Guide non-officiel du développeur Elementary OS

Version 0.1

Communauté francophone Elementary OS

18 February 2015

Table des matières

1 Introduction **1**
- 1.1 Licences . . . 1
- 1.2 Auteurs . . . 1

2 Le langage Vala **3**
- 2.1 Introduction . . . 3
- 2.2 Premiers pas en Vala . . . 4
- 2.3 La syntaxe générale du Vala . . . 6
- 2.4 Les variables . . . 9
- 2.5 Maîtrise des variables : avancée . . . 13
- 2.6 Les instructions conditionnelles . . . 15
- 2.7 Les tableaux . . . 21

3 Les interfaces graphique avec Gtk **25**
- 3.1 Introduction . . . 25

4 Gestion de projet **27**
- 4.1 Introduction . . . 27
- 4.2 Mise en route . . . 27
- 4.3 Première application . . . 33
- 4.4 Premier projet . . . 36
- 4.5 Plus de possibilités avec Vala . . . 43

5 Les technologies elementary **49**
- 5.1 Introduction . . . 49
- 5.2 Contractor . . . 49
- 5.3 Les Plugs Switchboard . . . 50

6 Lignes directrices **51**
- 6.1 Introduction . . . 51
- 6.2 Style de code . . . 51

7 Indices and tables **55**

CHAPITRE 1

Introduction

Ce guide a pour but de guider un développeur Linux vers le kit de développement Elementary OS.

Une attention particulière a été prise pour aider les débutants. Toutefois, si vous avez besoin d'aide, n'hésitez pas à en demander dans le forum de la communauté francophone d'Elementary OS.

Vous accèderez au forum en suivant le lien suivant : http://forum.elementaryos-fr.org

Important : Ce guide est encore en cours d'écriture. Il peut contenir des informations erronées.

Les erreurs éventuelles peuvent être signalées à l'adresse suivante : https://github.com/Elementary-fr/elementaryos-fr-doc/issues

1.1 Licences

Ce guide reprend des idées, du code et du texte depuis divers sites anglophones ou francophones.

Les parties créées entièrement pour les besoins de cette documentation sont placées sous licence **CC BY-SA 3.0** - https://creativecommons.org/licenses/by-sa/3.0/

Les parties traduites ou adaptées à partir d'une source externe peuvent avoir une licence différente. Cette license est en générale précisée dans la partie introduction de la partie en question.

1.2 Auteurs

Ce guide a été créé par la communauté francophone ElementaryOS-FR et ne dois pas être considéré comme une publication officiel de l'équipe Elementary OS.

CHAPITRE 2

Le langage Vala

2.1 Introduction

2.1.1 Présentation

Vala est un langage de programmation apparu en 2006. Il a été créé par Jürg Billeter.

Depuis toujours, le langage de prédilection de Gnome est le C. C'est un des langages de programmation les plus utilisés de nos jours. Le C est un langage très puissant. Pour preuve : c'est le langage du noyau Linux.

L'équipe de Gnome avait décidée d'utiliser le C pour construire son bureau libre. Cependant, lorsqu'il a fallu améliorer la librairie permettant de créer l'interface graphique, ils se sont rendus compte qu'ils devaient réécrire le code avec un style orienté objet.

Le C ne permettant pas de base la création d'objets, ils ont créés leur système, appelé GObject, par dessus le C.

Ainsi, Gnome a pu faire de l'orienté objet et profiter des autres avantages du C.

Le principal avantage : les librairies Gnome peuvent être utilisées facilement dans presque tous les autres langages, y compris les tout derniers langages à la mode, les langages interprétés comme le Python.

Les langages interprétés ne sont pas directement convertis en langage machine par le dévelopeur mais seulement lors de l'exécution, par l'interpréteur, qui convertit le code à la volée.

Cette méthode permet de faire des langages très flexibles et simples, mais en perdant un peu de performance.

Vala a été créé pour palier au problème de performances des langages interprétés. Le compilateur Vala traduit le code en language C, puis le compilateur C, en langage machine. De plus, le langage Vala amène toutes les principales nouveautés des langages modernes, ce qui permet d'écrire des programmes facilement tout en gardant une vitesse d'exécution proche du C.

2.1.2 Installation

Pour utiliser le langage Vala, nous devons installer le compilateur Valac, qui va convertir nos fichiers .vala en ficier .c. Il nous faut aussi le compilateur Gcc qui va convertir les fichier .c en programme exécutable.

```
$ sudo apt-get install valac build-essential
```

2.1.3 Auteurs et licences

Cette partie est une adaptation du cours “Apprenez à développer en C#” de Nicolas Hilaire, disponible à l’adresse suivante : http://fr.openclassrooms.com/informatique/cours/apprenez-a-developper-en-c

Conformément à la licence d’origine, cette partie est placée sous licence **CC BY-NC-SA 2.0** - http://creativecommons.org/licenses/by-nc-sa/2.0/

2.2 Premiers pas en Vala

2.2.1 Un programme ?

Avant de commencer à coder quoi que ce soit il serait déjà bien de savoir de quoi on parle, non ?

Alors un programme qu’est-ce que c’est ?

Un programme est un ensemble de **fonctions**. Chaque fonction regroupe une suite d’**instructions** qui permettent de faire une chose précise : afficher un bonjour, faire un calcul et j’en passe. Une fois construite, une fonction peut être utilisée dans une autre fonction. Pendant l’exécution de notre programme, celui ci lance automatiquement la fonction “**main**”, qui est la fonction principale du programme, c’est dans celle-ci que se déroule la partie la plus attractive ! ;)

- Un programme en Vala est composé de fonctions.
- Chaque fonction est composée d’instructions.
- Chaque fonction est indépendante.
- La première fonction appelée dans un programme s’appelle “main”.

```
void main (){

}
```

Structure d’une fonction

Il y a plusieurs types de fonctions. On ne va pour le moment parler que des trois basiques.

Une fonction est composée de deux parties :

- **Son prototype qui est lui-même composé de 3 parties :**
  - **Le type de retour :**
    - int qui retournera un entier.
    - char qui retournera un caractère ou une chaîne.
    - void qui ne demande pas de valeur de retour.
  - Le nom (pas d’espace entre le nom, pas de majuscule)
  - Les paramètres, entre parenthèses (dépend du type de la fonction)

    Exemple : `int nom_de_la_fonction(int a, int b)`
- **Son corps :**
  - Délimité par des accolades
  - Un return, qui retourne le type de la fonction. Sauf pour la fonction void qui n’en demande pas.

```
int function_addition(int a, int b, int c)
{
    c = a + b;
    return(c);
}
```

Tout ce qui va être exécuté se trouve dans le corps de la fonction.

Ne pas oublier :

Voici une petite liste des choses importantes à ne pas oublier quand vous allez coder. vous allez, au moins une fois, faire chacune des erreurs listées ci-dessous :

- Ne surtout pas oublier le **point virgule ;** à la fin de vos ligne de code.
- N'oubliez pas le return quand votre fonction en demande un.
- Ouvrez et fermez vos parenthèses et accolades avant d'écrire quoi que ce soit dedans. Cela vous évitera d'en oublier une ou de vous perdre au moment de toutes les fermer.

La fonction main()

```
int main  (string[] args) {
   print ("Salut le peuple de la Terre!\n");
   return(0);
}
```

Voici un exemple avec le type de retour **void** :

```
void main (string[] args) {
   print ("Je ne retourne rien ! Mais je suis très utile!\n");
}
```

Note : Le **\n** permet de faire un retour à la ligne, cela évite d'avoir une phrase qui se colle dans le prompt du terminal.

2.2.2 Hello World en console

La tradition veut que lorsqu'on apprend un nouveau langage de programmation, on commence par écrire un programme qui affiche *Hello World !* dans la console.

On commence par créer un dossier Projets, avec un sous dossier *console-hello*. Dans ce sous dossier, on va créer un fichier hello.vala et on va l'ouvrir avec notre éditeur de texte.

```
$ cd ~  # Permet de revenir dans notre dossier personnel.
$ mkdir Projets # Crée le dossier Projets.
$ mkdir Projets/console-hello # Crée le dossier console-hello dans Projets.
$ cd Projets/console-hello # On se déplace dans le nouveau répertoire.
$ touch hello.vala # On crée le fichier.
$ xdg-open hello.vala # On ouvre le fichier avec le programme par défaut.
```

L'éditeur de text *Scratch* à dû s'ouvrir. Copiez dans le fichier le code suivant :

```
void main (){
   print("Hello world!\n");
}
```

Ensuite, on retourne dans le terminal et on lance la compilation. Si nécessaire, un appui sur la touche *enter* permet de créer une nouvelle ligne de commande.

```
$ valac hello.vala
$ ./hello
```

La deuxième ligne lance notre premier programme. Si tout se passe bien, la phrase *Hello World !* devrait s'être affichée dans le terminal.

Note : Le code source des mini-projets de ce guide peut être consulté à l'adresse suivante :

https://github.com/Elementary-fr/elementaryos-fr-exemple

2.3 La syntaxe générale du Vala

Nous allons aborder ici la syntaxe générale du langage de programmation Vala dans le cadre d'une application console. Il est en effet possible de créer plein de choses différentes avec le Vala comme une application web, des jeux, etc.

Dans cette optique, nous allons utiliser très souvent l'instruction : **print("...");** que nous avons vue au chapitre précédent et qui est une instruction dédiée à l'affichage sur la console. C'est une instruction qui va s'avérer très pratique pour notre apprentissage car nous pourrons avoir une représentation visuelle de ce que nous allons apprendre.

Il est globalement rare qu'une application ne doive afficher que du texte, sans aucune mise en forme. Vous verrez dans une autre partie comment réaliser des applications un peu plus évoluées graphiquement.

Préparez vous, nous plongeons petit à petit dans l'univers du Vala. Dans ce chapitre, nous allons nous attaquer à la syntaxe générale du Vala et nous serons capable de reconnaître les lignes de code et de quoi elles se composent.

2.3.1 Ecrire une ligne de code

Les lignes de code écrites avec le langage de développement Vala doivent s'écrire dans des fichiers dont l'extension est .vala, comme dans le chapitre précédent, lorsque nous avons créer un fichier hello.vala. Nous y avons notamment rajouté une instruction permettant d'afficher du texte.

Les lignes de code Vala se lisent et s'écrivent de haut en bas et de gauche à droite, comme un livre normal. Aussi, une instruction écrite avant une autre sera en général exécutée avant celle-ci.

Important : Attention, chaque ligne de code doit être syntaxiquement correcte sinon le compilateur ne saura pas la traduire en langage exécutable.

Par exemple, si à la fin de mon instruction, je retire le point-virgule ou si j'orthographie mal le mot print, j'aurai le message d'erreur suivant :

Ce sont des erreurs de compilation qu'il va falloir résoudre si l'on souhaite que l'application console puisse s'exécuter.

Nous allons voir dans les chapitres suivants comment écrire correctement des instructions en Vala. Mais il est important de noter à l'heure actuelle que le Vala est sensible à la casse, ce qui veut dire que les majuscules comptent !

Ainsi le mot « print » et le mot « Print » sont deux mots bien distincts et peuvent potentiellement représenter deux instructions différentes. Ici, le deuxième mot est incorrect car il n'existe pas.

Rappelez-vous bien que la casse est déterminante pour que l'application puisse compiler.

2.3.2 Le caractère de terminaison de ligne

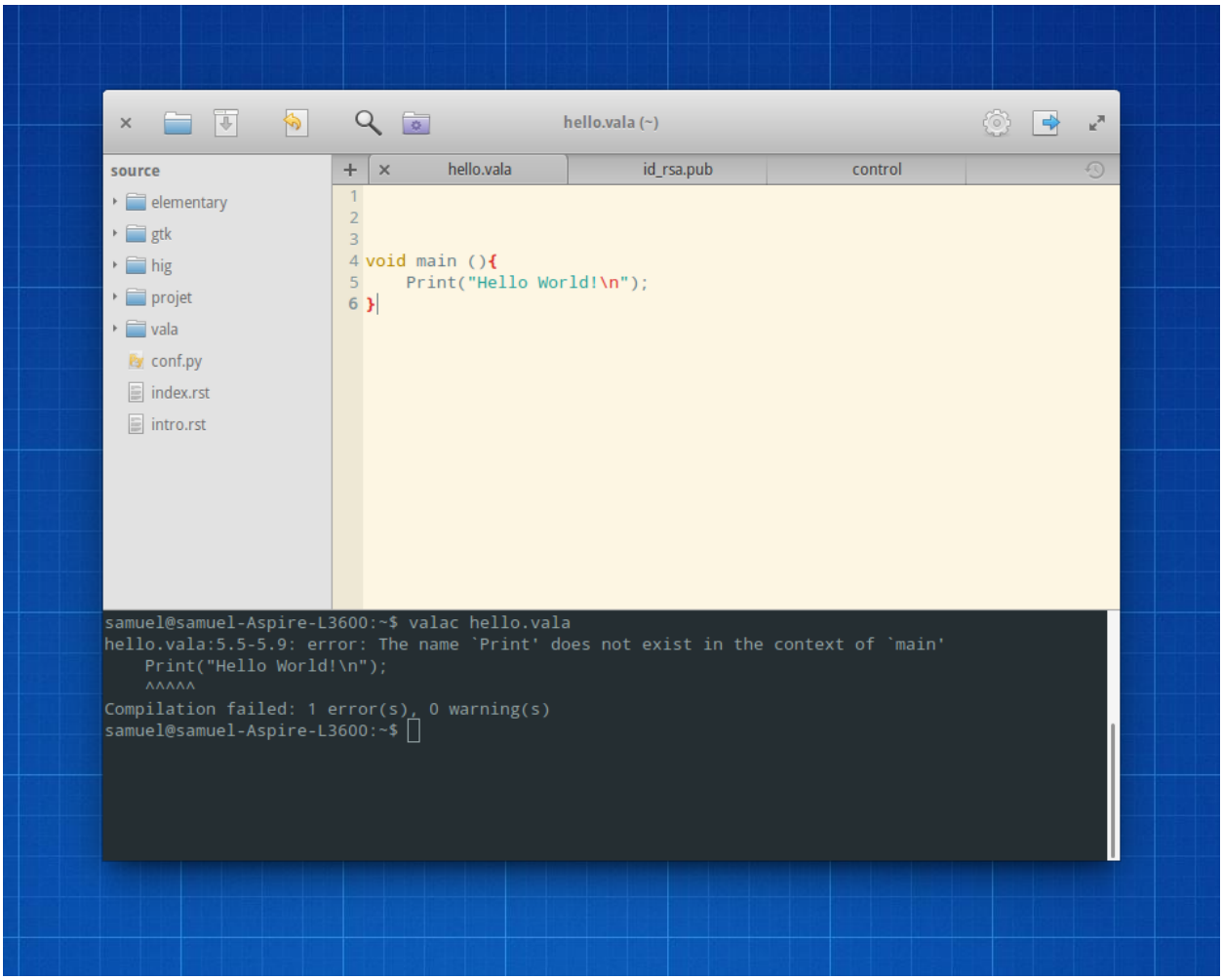
En général, une instruction en code Vala s'écrit sur une ligne et se termine par un point-virgule.

Ainsi, l'instruction que nous avons vue plus haut :

```
print("Hello World!");
```

se termine au niveau du point-virgule.

Il aurait été possible de remplacer le code écrit :


hello.vala (~)
source
elementary
gtk
hig
projet
vala
conf.py
index.rst
intro.rst
hello.vala
id_rsa.pub
control
1
2
3
4 void main (){
5 Print("Hello World!\n");
6 }
samuel@samuel-Aspire-L3600:~$ valac hello.vala
hello.vala:5.5-5.9: error: The name `Print' does not exist in the context of `main'
Print("Hello World!\n");
^^^^^
Compilation failed: 1 error(s), 0 warning(s)
samuel@samuel-Aspire-L3600:~$

```
void main (){
    print("Hello World!");
}
```

par :

```
void main (){print("Hello World!");}
```

ou encore :

```
void main (){


    print("Hello World!");


            }
```

Note : En général, pour que le code soit le plus lisible possible, on écrit une instruction par ligne et on indente le code de façon à ce que les blocs soient lisibles.

Note : Un bloc de code est délimité par des accolades { }. Nous y reviendrons plus tard.

Indenter signifie que chaque ligne de code qui fait partie d'un même bloc de code commence avec le même retrait sur l'éditeur. Ce sont soit des tabulations, soit des espaces qui permettent de faire ce retrait.

En général, les programmeurs Vala utilisent quatre espaces par niveau d'indentation.

Décortiquons à présent cette ligne de code :

```
print("Hello World!");
```

Pour simplifier, nous dirons que nous appelons la méthode print qui permet d'écrire une chaîne de caractères sur la console.

Une méthode représente une fonctionnalité, écrite avec du code, qui est utilisable par d'autres bouts de code (par exemple, calculer la racine carrée d'un nombre ou afficher du texte ...).

L'instruction "Hello World !" représente une chaine de caractères et est passée en paramètre de la méthode print à l'aide des parenthèses. La chaîne de caractères est délimitée par les guillemets. Enfin, le point-virgule permet d'indiquer que l'instruction est terminée et que l'on peut enchaîner sur la suivante.

Certains points ne sont peut-être pas encore tout à fait clairs, comme ce qu'est vraiment une méthode, ou comment utiliser des chaînes de caractères. Mais ne vous inquiétez pas, nous allons y revenir plus en détail dans les chapitres suivants et découvrir au fur et à mesure les arcanes du Vala.

2.3.3 Les commentaires

Pour faciliter la compréhension du code ou pour se rappeler un point précis, il est possible de mettre des commentaires dans son code. Les commentaires sont ignorés par le compilateur et n'ont qu'une valeur informative pour le développeur.

Dans un fichier de code Vala (.vala), on peut écrire des commentaires de deux façons différentes :

Soit en commençant son commentaire par /* et en le terminant par */. Ceci permet d'écrire un commentaire sur plusieurs lignes.

Soit en utilisant //. Ainsi, tout ce qui se trouve après sur la même ligne est alors un commentaire.

L'éditeur Scratch colore les commentaires en *gris* pour faciliter leur identification.

```
/* permet d'afficher du texte
   sur la console */
print("Hello World !!"); // ne pas oublier le point virgule
```

2.3.4 En résumé

— Le code Vala est composé d'une suite d'instructions qui se terminent par un point virgule.
— La syntaxe d'un code Vala doit être correcte sinon nous aurons des erreurs de compilation.
— Il est possible de commenter son code grâce aux caractères « // », « /* » et « */ ».

2.4 Les variables

2.4.1 Qu'est-ce qu'une variable ?

Comme tous les langages de programmation, le Vala va pouvoir conserver des données grâce à des variables. Ce sont en fait des blocs de mémoire qui vont contenir des nombres, des chaînes de caractères, des dates ou plein d'autres choses.

Les variables vont nous permettre d'effectuer des calculs mathématiques, d'enregistrer l'âge du visiteur, de comparer des valeurs, ...etc.

Une variable est représentée par son nom, caractérisée par son type et contient une valeur.

Note : Le type correspond à ce que la variable représente : un entier, une chaîne de caractères, une date, ...etc.

2.4.2 Déclaration/initialisation de variable

Déclaration

Voici un exemple de **déclaration** pour chaque type de variable :

```
int nombre;
```

Note : On appelle ceci la déclaration de la variable.

Important : *int* correspond au début de "integer" qui veut dire "entier" en anglais.

Initialisation

Dans cette exemple plus haut, la variable n'a pas été initialisé. Elle ne pourra pas être utilisée car le compilateur ne sait pas quelle valeur il y a dans la variable.

Pour l'initialiser (on parle également d'affecter une valeur) on utilisera l'opérateur "égal à" (=) :

```
int nombre;

nombre = 30;
```

Notre variable *nombre* possède désormais l'entier numérique *30* comme valeur.

L'initialisation d'une variable peut également se faire au même moment que sa déclaration. Ainsi, on pourra remplacer le code précédent par :

```
int nombre = 30;
```

Note : Nous ne recommandons pas cette façon de faire, nous reparlerons de la norme dans un autre chapitre.

Nous pouvons à tout moment demander la valeur contenue dans la variable *nombre_entier*, par exemple :

```
int nombre_entier = 10;
print("%d \n", nombre); // Affiche 10 dans la console.
```

Note : Les caractères *%d* sont la pour indiquer que *nombre* est un chiffre entier décimal.

Les caractères *n*, eux, indiquent à la console de faire un saut de ligne. Nous en reparlerons dans un chapitre plus en détails sur la lecture/écriture.

La norme des noms

Vous pouvez nommer vos variables à peu près n'importe comment, à quelques détails près. Les noms de variables ne peuvent pas avoir le même nom qu'un type. Il sera alors impossible d'appeler une variable *int*.

Il est également impossible d'utiliser des caractères spéciaux, comme des espaces, des tirets, des lettres accentuées ou des caractères de ponctuation.

De même, on ne pourra pas nommer une variable en commençant par des chiffres.

Note : Il est souvent recommandé de nommer ses variables en anglais (langue qui ne contient pas d'accent). Vous aurez noté que je ne le fais pas volontairement dans ce tutoriel afin de ne pas rajouter une contrainte supplémentaire lors de la lecture du code. Mais libre à vous de le faire.

En général, une variable commence par une minuscule et si son nom représente plusieurs mots, on les séparera par des "underscore" à savoir ces caractères la : "_" (Sans les guillemets) . Celui qui le nommera par "tiret du bas" se fera lyncher par ses collèges. Exemple :

```
int age_du_visiteur;
```

Important : A noter un détail qui peut paraître évident, mais toutes les variables sont réinitialisées à chaque nouvelle exécution du programme. Dès qu'on démarre le programme, l'emplacement de mémoire vive est vidé, comme si on emménageait dans des nouveaux locaux à chaque fois.

Il est donc impossible de faire persister une information entre deux exécutions du programme en utilisant des variables normales. Pour ceci, on utilisera d'autres solutions, comme enregistrer des valeurs dans un fichier ou dans une base de données. Nous y reviendrons ultérieurement.

2.4.3 Affectations, opérations, concaténation

Nous allons voir comment intéragir avec les variable. Nous pouvons également faire des opérations +, ‘‘*‘‘, / ou encore -, par exemple :

```
int resultat = 2 * 3;
```

ou encore :

```
int age1 = 20;
int age2 = 30;
int moyenne = (age1 + age2) / 2;
```

Vous aurez donc surement deviné que la variable *resultat* contient 6 et que la moyenne vaut 25.

Il est à noter que les variables contiennent une valeur qui ne peut évoluer qu'en affectant une nouvelle valeur à cette variable.

Ainsi, si j'ai le code suivant :

```
int age1 = 20;
int age2 = 30;
int moyenne = (age1 + age2) / 2;
age2 = 40;
```

Les initialisations de variable auront toujours la même valeur définie, les calculs dans le programme ne seront pas sauvegardé.

Concaténation

D'autres opérateurs particuliers, que nous ne trouvons pas dans les cours de mathématiques, existent. Par exemple, l'opérateur ++ qui permet de réaliser une **incrémentation** de 1, ou l'opérateur – qui permet de faire une **décrémentation** de 1.

Une incrémentation de 1 revient à faire `x = x + 1;` Une décrémentation de 1 revient à faire `x = x - 1;`

De même, les opérateurs que nous avons déjà vus peuvent se cumuler à l'opérateur pour simplifier une opération qui prend une variable comme opérande et cette même variable comme résultat.

Par exemple :

```
int age = 20;
age = age + 10; // age contient 30 (addition)
age++; // age contient 31 (incrémentation de 1)
age--; // age contient 30 (décrémentation de 1)
age += 10; // équivalent à age = age + 10 (age contient 40)
age /= 2; // équivalent à age = age / 2 => (age contient 20)
```

2.4.4 Notion de math

Comme nous avons pu le voir dans nos cours de mathématiques, il est possible de grouper des opérations avec des parenthèses pour agir sur leurs priorités.

Ainsi, l'instruction précédemment vue :

```
int moyenne = (age1 + age2) / 2;
```

effectue bien la somme des deux âges avant de les diviser par 2, car les parenthèses sont prioritaires.

Cependant, l'instruction suivante :

```
int moyenne = int moyenne = age1 + age2 / 2;
```

aurait commencé par diviser l'age2 par 2 et aurait ajouté l'age1, ce qui n'aurait plus rien à voir avec une moyenne.

En effet, la division est prioritaire par rapport à l'addition.

Important : Attention, la division ici est un peu particulière.

Prenons cet exemple :

```
int moyenne = 5 / 2;
print("%d \n", moyenne);
```

Si nous l'exécutons, nous voyons que moyenne vaut 2.

Indice : 2 ? Si je me rappelle bien de mes cours de math ... N'est-ce pas plutôt 2.5 ?

Oui et non.

Si nous divisions 5 par 2, nous obtenons bien 2.5.

Par contre, ici nous divisons l'entier 5 par l'entier 2 et nous stockons le résultat dans l'entier moyenne.

Le Vala réalise en fait une division entière, c'est-à-dire qu'il prend la partie entière de 2.5, c'est-à-dire 2.

De plus, l'entier moyenne est incapable de stocker une valeur contenant des chiffres après la virgule. Il ne prendrait que la partie entière.

Pour avoir 2.5, il faudrait utiliser le code suivant :

```
double moyenne = 5.0 / 2.0;
print("%f \n", moyenne);
```

Ici, nous divisons deux « doubles » entre eux et nous stockons le résultat dans un « double ». (Rappelez-vous, le type de données « double » permet de stocker des nombres à virgule.)

Note : Le Vala comprend qu'il s'agit de double car nous avons ajouté un .0 derrière. Sans ça, il considère que les chiffres sont des entiers.

2.4.5 Les caractères spéciaux dans les chaines de caractères

En ce qui concerne l'affectation de chaînes de caractères, vous risquez d'avoir des surprises si vous tentez de mettre des caractères spéciaux dans des variables de type string.

En effet, une chaîne de caractères étant délimitée par des guillemets " ", comment faire pour que notre chaîne de caractères puisse contenir des guillemets ?

C'est là qu'intervient le caractère spécial (backslash) qui sera à mettre juste devant le guillemet, par exemple le code suivant :

```
string phrase = "Mon prénom est \"Nicolas\" \n";
print(phrase);
```

affichera :

```
Mon prénom est "Nicolas"
```

Si vous avez testé par vous-même, vous avez pu remarquer que le caractère spécial *n* effectue un saut de ligne.

Ainsi, le code suivant :

```
string phrase = "Mon prénom est \"Nicolas\"\n";
print(phrase);
print("Passage\nà\nla\nligne\n");
```

affichera :

```
Mon prénom est "Nicolas"
Passe
à
la
ligne
```

où nous remarquons bien les divers passages à la ligne.

On peut aussi utiliser le caractère spécial *t* pour effectuer une tabulation. Le code suivant :

```
print("Choses à faire :");
print("\t - Arroser les plantes");
print("\t - Laver la voiture");
```

permettra d'afficher des tabulations, comme illustré ci-dessous :

```
Choses à faire :
        - Arroser les plantes
        - Laver la voiture
```

Nous avons vu que le caractère "backslash" était un caractère spécial et qu'il permettait de dire au compilateur que nous voulions l'utiliser combiné à la valeur qui le suit, permettant d'avoir une tabulation ou un retour à la ligne.

Comment pourrons-nous avoir une chaîne de caractères qui contienne ce fameux caractère ?

Le principe est le même, il suffira de faire suivre ce fameux caractère spécial de lui-même. Exemple :

```
print("Le caractère backslash \\");
```

2.4.6 En résumé

- Une variable est une zone mémoire permettant de stocker une valeur d'un type particulier.
- Le Vala possède plein de types prédéfinis, comme les entiers (int), les chaînes de caractères (string), etc.
- On utilise l'opérateur = pour affecter une valeur à une variable.
- Il est possible de faire des opérations entre les variables.

2.5 Maîtrise des variables : avancée

2.5.1 Les différents types de variables

Nous avons vu juste au-dessus que la variable *nombre* pouvait être un entier numérique grâce au mot clé *int*. Le langage Vala dispose de beaucoup de types permettant de représenter beaucoup de choses différentes.

Par exemple, nous pouvons stocker une chaîne de caractères grâce au type *string*.

```
string prenom = "nicolas";
```

ou encore un boolean (qui représente une valeur vraie ou fausse) avec :

```
bool est_vrai= true;
bool est_faux = false;
```

Warning : Il est important de stocker des données dans des variables ayant le bon type.
On ne peut par exemple pas stocker le prénom “Nicolas” dans un entier.

Les principaux types de base du langage Vala sont :

Nombre décimal à taille garantie :

Type	Description
int8	Entier de -128 à +127
int16	Entier de -32768 à 32767
int32	Entier de -2147483648 à 2147483647
int64	Entier de -9223372036854775808 à 9223372036854775807
uint8	Entier non-signé (uniquement positif) de 0 à 255
uint16	Entier non-signé de 0 à 65535
uint32	Entier non-signé de 0 à 4294967295
uint64	Entier non-signé de 0 à 18446744073709551615

Nombre décimal à taille variable :

Les nombres ci-dessous ont une taille qui varie en fonction de la plateforme sur laquelle le code a été compilé. Les valeurs limitées sont données, à titre indicatif, pour un ordinateur 32 bits.

Type	Description
short	Entier de -32768 à 32767
int	Entier de -2147483648 à 2147483647
long	Entier de -9223372036854775808 à 9223372036854775807

Nombre à virgule

Type	Description
float	Nombre simple précision de -3,402823e38 à 3,402823e38
double	Nombre double précision de -1,79769313486232e308 à 1,79769313486232e308

Types spécifiques

Type	Description
bool	Boolean, vrai ou faux
unichar	Un caractère (utilisant l’encodage *unicode*)
string	Une chaîne de caractère

Vous verrez plus loin qu’il existe encore d’autres types dans le langage Vala et qu’on peut également construire les siens.

2.6 Les instructions conditionnelles

Dans nos programmes Vala, nous allons régulièrement avoir besoin de faire des opérations en fonction d'un résultat précédent. Par exemple, lors d'un processus de connexion à une application, si le login et le mot de passe sont bons, alors nous pouvons nous connecter, sinon nous afficherons une erreur.

Il s'agit de ce que l'on appelle une condition. Elle est évaluée lors de l'exécution et en fonction de son résultat (vrai ou faux) nous ferons telle ou telle chose.

Bien que relativement court, ce chapitre est très important. N'hésitez pas à le relire et à vous entraîner.

2.6.1 Les opérateurs de comparaison

Une condition se construit grâce à des opérateurs de comparaison. On dénombre plusieurs opérateurs de comparaisons, les plus courants sont :

Opérateur	Description
==	Egalité
!=	Différence
>	Supérieur à
<	Inférieur à
>=	Supérieur ou égal
<=	Inférieur ou égal
&&	ET logique
||	OU logique
!	Négation

Nous allons voir comment les utiliser en combinaison avec les instructions conditionnelles.

2.6.2 L'instruction *if*

L'instruction **if** permet d'exécuter du code si une condition est vraie (if = si en anglais).

Par exemple :

```
int compte_en_banque = 300;
if (compte_en_banque >= 0)
    print("Votre compte est créditeur\n");
```

Ici, nous avons une variable contenant le solde de notre compte en banque. Si notre solde est supérieur ou égal à 0 alors nous affichons que le compte est créditeur.

Pour afficher que le compte est débiteur, on pourrait tester si la valeur de la variable est inférieure à 0 et afficher que le compte est débiteur :

```
1 int compte_en_banque = 300;
2 if (compte_en_banque >= 0)
3     print("Votre compte est créditeur\n");
4 if (compte_en_banque < 0)
5     print("Votre compte est débiteur\n");
```

Une autre solution est d'utiliser le mot clé **else**, qui veut dire « sinon » en anglais.

« Si la valeur est vraie, alors on fait quelque chose, sinon, on fait autre chose », ce qui se traduit en Vala par :

```
int compte_en_banque = 300;
if (compte_en_banque >= 0)
    print("Votre compte est créditeur\n");
else
    print("Votre compte est débiteur\n");
```

Il faut bien se rendre compte que l'instruction if teste si une valeur est vraie (dans l'exemple précédent la comparaison compte_en_banque >= 0).

On a vu rapidement dans les chapitres précédents qu'il existait un type de variable qui permettait de stocker une valeur vraie ou fausse : le type bool, autrement appelé booléen (boolean en anglais).

Ainsi, il sera également possible de tester la valeur d'un booléen. L'exemple précédent peut aussi s'écrire :

```
int compte_en_banque = 300;
int est_crediteur = (compte_en_banque >= 0);
if (est_crediteur)
    print("Votre compte est créditeur\n");
else
    print("Votre compte est débiteur\n");
```

À noter que les parenthèses autour de l'instruction de comparaison sont facultatives, je les ai écrites ici pour clairement identifier que la variable *est_crediteur* va contenir une valeur qui est le résultat de l'opération de comparaison « compte en banque est supérieur ou égal à 0 », en l'occurrence vrai.

Voici d'autres exemples pour vous permettre d'appréhender plus précisément le fonctionnement du type bool :

```
int age = 30;
bool est_age_de_30_ans = (age == 30); // Valeur vrai (true)
bool est_superieur_a_10 = age > 10; // Valeur vrai (true)
bool est_different_de_30 = age != 30; // Valeur fausse (false)
```

Combinaison

Il est également possible de combiner les tests grâce aux opérateurs logique conditionnels, par exemple && qui correspond à l'opérateur *ET*.

Dans l'exemple qui suit, nous affichons le message de bienvenue uniquement si le login est « Nicolas » ET que le mot de passe est « test ». Si l'un des deux ne correspond pas, nous irons dans l'instruction **else**.

```
string login = "Nicolas";
string mot_de_passe = "test";
if (login == "Nicolas" && mot_de_passe == "test")
    print("Bienvenue Nicolas\n");
else
    print("Login incorrect\n");
```

Note : Remarquons ici que nous avons utilisé le test d'égalité **==**, à ne pas confondre avec l'opérateur d'affectation **=**. C'est une erreur classique de débutant.

D'autres opérateurs logiques existent, nous avons notamment l'opérateur || qui correspond au *OU* logique :

```
if (civilite == "Mme" || civilite == "Mlle")
    print("Vous êtes une femme\n");
else
    print("Vous êtes un homme\n");
```

L'exemple parle de lui-même, si la civilité de la personne est *Mme* ou *Mlle*, alors nous avons à faire à une femme.

A noter ici que si la première condition du **if** est vraie alors la deuxième ne sera pas évaluée. C'est un détail ici, mais cela peut s'avérer important dans certaines situations dont une que nous verrons un peu plus loin.

Un autre opérateur très courant est la négation que l'on utilise avec l'opérateur **!**. Par exemple :

```
bool est_vrai = true;
if (!est_vrai)
    print("C'est faux !\n");
else
    print("C'est vrai !\n");
```

Ce test pourrait se lire ainsi : « Si la négation de la variable *est_vrai* est vraie, alors on écrira c'est faux ».

La variable « est_vrai » étant égale à *true*, sa négation vaut *false*.

Dans cet exemple, le programme nous affichera donc l'instruction correspondant au **else**, à savoir « C'est vrai ! ».

Rappelez-vous, nous avons dit qu'une instruction se finissait en général par un point-virgule. Comment cela se fait-il alors qu'il n'y ait pas de point-virgule à la fin du if ou du else ?

Et si nous écrivions l'exemple précédent de cette façon ?

```
bool est_vrai = true;
if (!est_vrai) print("C'est faux !\n");
else print("C'est vrai !\n");
```

Ceci est tout à fait valable et permet de voir où s'arrête vraiment l'instruction grâce au point-virgule. Cependant, nous écrivons en général ces instructions de la première façon afin que celles-ci soient plus lisibles.

Vous aurez l'occasion de rencontrer dans les chapitres suivants d'autres instructions qui ne se terminent pas obligatoirement par un point-virgule.

Remarquons enfin qu'il est possible d'enchaîner les tests de manière à traiter plusieurs conditions en utilisant la combinaison **else if**. Cela donne :

```
if (civilite == "Mme")
    print("Vous êtes une femme\n");
else if (civilite == "Mlle")
    print("Vous êtes une femme non mariée\n");
else if (civilite == "M.")
    print("Vous êtes un homme\n");
else
    print("Je n'ai pas pu déterminer votre civilité\n");
```

2.6.3 Notion très avancée : Les ternaires

Les ternaires sont ni plus ni moins qu'un if/else qui a la particularitée de tenir sur une ligne. on les utilise pour envoyer une information dans une variable voir un return

---

Note : Si vous venez tout juste d'apprendre la programmation, vous pouvez passer cette partie, cependant il est important de savoir les utiliser à la fin de ce cours.

```
void main () {
   string nom = "Nolan";
   print ((nom == "Nolan")? "oui \n" : "non \n");
}
```

Le résultat est :

```
oui
```

Explication

Pour mon exemple, je retourne une condition à un print, la lecture en pseudo code serait :

```
affiche ((condition)si "alors" sinon "ça");
```

La condition est avant le **si (?)**, et pour le **sinon (:)** ne se ferme que avec la première paranthese.

Important : Contrairement au **if**, il est obligatoire d'avoir un **else** dans une ternaire. Dans le cas ou vous utilisez une ternaire sur une variable, je vous conseil de retourner la même variable.

Voici un exemple un peu plus concret :

```
void main () {
   int nb = -5;

   nb = ((nb < 0)? nb * -1 : nb);
   print ("%d \n", nb);
}
```

Admettons que pour une raison x ou y je cherche à convertir un nombre négatif en nombre positif, la solution la plus ergonomique serait de faire une ternaire. Dans le cas contraire, si nb est superieur à 0, nb serait devenu nb.

Une, deux ou trois ternaire dans une ternaire

Le prototype d'une ternaire imbriqué est la suivante :

```
affiche ((condition)si "alors" sinon ((condition2)si "alors" sinon "ca"));
```

Pour rendre plus lisible la deuxieme ternaire, je vous conseil de le placer dans une parenthèse.

2.6.4 L'instruction switch

L'instruction **switch** peut être utilisée lorsqu'une variable peut prendre beaucoup de valeurs. Elle permet de simplifier l'écriture.

Ainsi, le code précédent peut aussi s'écrire de la manière suivante :

```
switch(civilite) {
    case "Mme":
        print("Vous êtes une femme\n");
        break;
    case "Mlle":
        print("Vous êtes une femme non mariée\n");
        break;
    case "m.":
        print("Vous êtes un homme\n");
        break;
    default:
         print("Je n'ai pas pu déterminer votre civilité\n");
```

```
13          break;
14  }
```

switch commence par évaluer la variable qui lui est passée entre parenthèses. Avec le mot clé **case** on énumère les différents cas possible pour la variable et on exécute les instructions correspondante jusqu'au mot clé **break** qui signifie que l'on sort du **switch**.

Nous pouvons également enchaîner plusieurs cas pour qu'ils fassent la même chose, ce qui reproduit le fonctionnement de l'opérateur logique || (OU).

Par exemple :

```
1   switch (mois) {
2       case "Mars":
3       case "Avril":
4       case "Mai":
5           print("C'est le printemps\n");
6           break;
7       case "Juin":
8       case "Juillet":
9       case "Aout":
10          print("C'est l'été\n");
11          break;
12      case "Septembre":
13      case "Octobre":
14      case "Novembre":
15          print("C'est l'automne\n");
16          break;
17      case "Décembre":
18      case "Janvier":
19      case "Février":
20          print("C'est l'hiver\n");
21          break;
22  }
```

Note : Il n'est pas obligatoire de retenir le switch, il reste très pratique pour faire les menus, mais facilement remplacable par une boucle avec des if.

2.6.5 Les boucles

Cette instruction permet de répéter une action tant que la condition est vraie, il en existe trois, la troisième étant quelque peu spécial, nous la verrons dans le chapitre des tableaux.

L'instruction while

while (condition) sigifie **tant que la condition est vrai, on revient à la même ligne**.

```
1   void main () {
2      int entier = 0;
3      while (entier <= 5) {
4        print ("%d \n", entier);
5        entier++;
6      }
7   }
```

Le résultat attendu :

```
0
1
2
3
4
5
```

Note : c'est clairement la plus importante instruction de boucle, avec un peu de réflexion, il est possible de se passer des deux prochaines instruction.

---

l'instruction do ... while

La **do ... while** est la même chose que notre while plus haut. Cependant, elle ne réagit pas aux même règles : la particularité de cette dernière est de faire un tour dans la boucle avant toutes choses, puis, si la condition est vérifiée, de continuer tant qu'elle est vraie.

```
void main  () {
   int entier = 0;
   do {
      print ("%d \n", entier);
      entier++;
   } while (entier <= 10);
}
```

Avec cette condition, le résultat sera :

```
0
1
2
3
4
5
6
7
8
9
10
```

Maintenant, si je change la condition :

```
void main  () {
   int entier = 0;
   do {
      print ("%d \n", entier);
      entier++;
   } while (entier >= 10);
}
```

Le résultat sera :

```
0
```

L'instruction for

La **for** est un peu particulière, elle permet de créer une variable et de l'initialiser, très utile pour faire un compteur. L'avantage de cette méthode est de ne pas perdre une ligne avec la création de la variable.

```
void main () {
   int entier = 10;
   for (int i = 0; entier >= i ; i++ ) {
      print ("%d \n", i);
   }
}
```

Le résultat de ce code sera :

```
0
1
2
3
4
5
6
7
8
9
10
```

En résumé

- Les instructions conditionnelles permettent d'exécuter des instructions seulement si une condition est vérifiée.
- On utilise en général le résultat d'une comparaison dans une instruction conditionnelle.
- Le Vala possède beaucoup d'opérateurs de comparaison, comme l'opérateur d'égalité **==**, l'opérateur de supériorité **>**, d'infériorité **<**, etc.

2.7 Les tableaux

2.7.1 Qu'est-ce qu'un tableau en programmation ?

Pour faire simple un tableau est une variable un peu spéciale qui permet de stocker plusieurs valeurs du même type dans ce dernier.

Une chose primordiale à retenir, la première valeur d'un tableau est à la position zéro :

Valeur	a	b	c	d	e	f	i
Position	0	1	2	3	4	5	6

2.7.2 Comment créer un tableau ?

Pour faire ce tableau en Vala, voici comment faire :

```
string[] tab;

tab = {"a", "b", "c", "d", "e", "f", "i"};
```

Ce qu'il faut retenir :

1. On reconnait un tableau par son type suivi des crochets.
2. En Vala, le tableau est dynamique, il n'est pas recommandé de définir la taille du tableau a l'avance (string[7] par exemple).
3. Pour initialiser le tableau, ne jamais oublier les accolades.

Note : Le nom de la variable utilisée ci-dessus, "tab" est juste un exemple.

2.7.3 Comment lire une information dans un tableau ?

Souvenez vous bien du tableau Position/Valeur, un tableau commence toujours par un 0 et fini toujours par "le nombre d'éléments" - 1.

Lire une valeur

Pour lire un élément dans le tableau, il suffit de donner sa position :

```
void main () {
string[] tab;

tab = {"a", "b", "c", "d", "e", "f", "i"};
print ("%s \n%s \n", tab[0], tab[5]);
}
```

Le résultat sera :

```
a
f
```

Cela ne nous choque pas, mais imaginons si je lui demande une position d'une valeur non déclarée ?

```
void main () {
   string[] tab;

   tab = {"a", "b", "c", "d", "e", "f", "i"};
   print ("%s \n, tab[42]);
}
```

Donnera :

```
(null)
```

Il ne nous donne aucune erreur de compilation, son résultat est juste nul.

Important : Le vala est vraiment bien, le même exemple en c serait un massacre : arrêt du programme, et un joli "segfault" (segmentation fault).

Note : Cependant, c'est un peu à double tranchant : je vous recommande de faire attention avec les valeurs de tableau et de toujours vérifier les valeurs de chaque variable avant de faire la fonction.

Lire plusieurs valeurs

Cela n'est pas plus dur que pour une seule valeur, admettons que je souhaite prendre plusieurs valeurs dans un tableau pour les copier vers un autre tableau :

```
void main () {
   string[] tab;
   string[] exp;

   tab = {"a", "b", "c", "d", "e", "f", "i"};
   exp = tab[1:3];
}
```

Quel sera le résultat ? Et bien, **exp** a maintenant **"b"** et **"c"**. Pourquoi pas le **"d"** ? On peut dire qu'il fait quelque chose comme : "on démarre à l'index 1, et on prend juste avant l'index 3".

Note : Les tableaux sont un élément important, je vous invite à faire des tests pour mieux comprendre leurs comportements.

Ajouter une valeur dans un tableau déjà créé

Voici un exemple en Vala pour ajouter des valeurs à la suite d'un tableau :

```
void main () {
   string[] tab;

   tab = {"a", "b", "c", "d"};
   tab += "e";
}
```

2.7.4 Le foreach

Voici la dernière boucle que je vais vous apprendre, le foreach. Elle est parfaite pour parcourir et retourner les valeurs existantes du tableau.

```
foreach (string key in tab){

}
```

Le prototype differt selon le langage, mais voici la version de Vala :

1. Création d'une variable du même type que le tableau : **string key**.
2. **"in"** pour préciser la variable du tableau, et **"tab"** est le nom de notre exemple plus haut.
3. Ce qu'il faut retenir c'est qu'à chaque tour de boucle, le foreach déplace la position vers le prochain élément existant et le stocke dans la variable "key".

```
void main () {
   string[] tab;

   tab = {"a", "b", "c", "d", "e", "f", "i"};
   foreach (string key in tab) {
      print ("%s \n", key);
   }
}
```

Le résultat :

```
a
b
c
d
e
f
i
```

2.7.5 Exercice du chapitre

Un peu plus, je vous ai montré comment faire pour ajouter simplement un élément dans le tableau, mais je ne vous ai pas montré comment faire l'inverse !

C'est un bon entrainement que je vous propose la, vous avez toute les cartes en main pour :

1. Créer une petite fonction qui prend en paramètre un tableau et un string (vive l'informatique...), cette fonction a pour but de supprimer le mot (dans le string) et de refaire un nouveau tableau.

2.7.6 Correction de l'entrainement :

Fonction pour retirer une valeur dans un tableau

Voici la correction du premier exercice :

```
string[] supMot(string[] tab, string mot) {
   string[] newTab = {};

   foreach (string key in tab)
      if (key !=  mot)
         newTab += key;
}

void main () {
   string[] tab;

   tab = {"a", "b", "c", "d", "e", "f", "i"};
   tab = supMot(tab, "b");
   foreach (string key in tab)
      print ("%s \n", key);
}
```

Note : À retenir :

1. Une fonction est toujours au dessus de la fonction "main".
2. Je n'utilise pas d'accolade si le contenu tiens en une ligne.
3. Je respecte l'ordre du prototype de la fonction.

CHAPITRE 3

Les interfaces graphique avec Gtk

3.1 Introduction

Important : Cette partie n'est pas encors écrite.

3.1.1 Auteurs et Licences

Cette partie est une adaptation du tutoriel "Python GTK+ 3 Tutorial", disponible à l'adresse suivante : http://python-gtk-3-tutorial.readthedocs.org/en/latest/

Conformément à la licence d'origine, cette partie est placée sous Licence GNU Free Documentation License 1.3 - http://www.gnu.org/copyleft/fdl.html

CHAPITRE 4

Gestion de projet

4.1 Introduction

Important : Ce guide est encore en cours d'écriture. Il peut contenir des informations erronées.

Les erreurs éventuelles peuvent être signalée à l'adresse suivante : https://github.com/Elementary-fr/elementaryos-fr-doc/issues

4.1.1 Auteurs et Licences

Cette partie reprend des idées, du codes et du texte depuis le site officiel de développement d'Elementary OS, disponible à l'adresse suivante : http://elementaryos.org/developer

Ce guide a été créer par la communauté francophone ElementaryOS-FR et ne doit pas être considéré comme une publication officielle de l'équipe elementary OS.

Le contenu de cette partie est placé sous Licence CC BY-SA 3.0 - https://creativecommons.org/licenses/by-sa/3.0/

4.2 Mise en route

4.2.1 Gestion du codes

Installation et configuration de Git

Tous projet logiciel sérieux doit utiliser un logiciel de versioning. Ce logiciel va vous permettre de sauvegarder votre code, de revenir en arrière en cas d'erreur et même de travailler en équipe.

Dans la communauté francophone, nous préférons utiliser le logiciel Git. On peut facilement l'installer avec la commande suivante :

```
$ sudo apt install git
```

Avant de passer à l'étape suivante, il faut encore configurer Git.

Nous allons tout d'abord ajouter notre nom réel et notre adresse email. C'est très important, d'un point de vue légal, il est important de savoir qui a fait quoi dans un logiciel. Pour se faire, il faut lancer les commandes suivantes :

```
$ git config --global user.name "John Smith"
$ git config --global user.email "votre_email@example.com"
```

Et avant de continuer, une dernière petite configuration. On va demander à Git d'utiliser le mode simple pour la synchronisation avec les serveurs git.

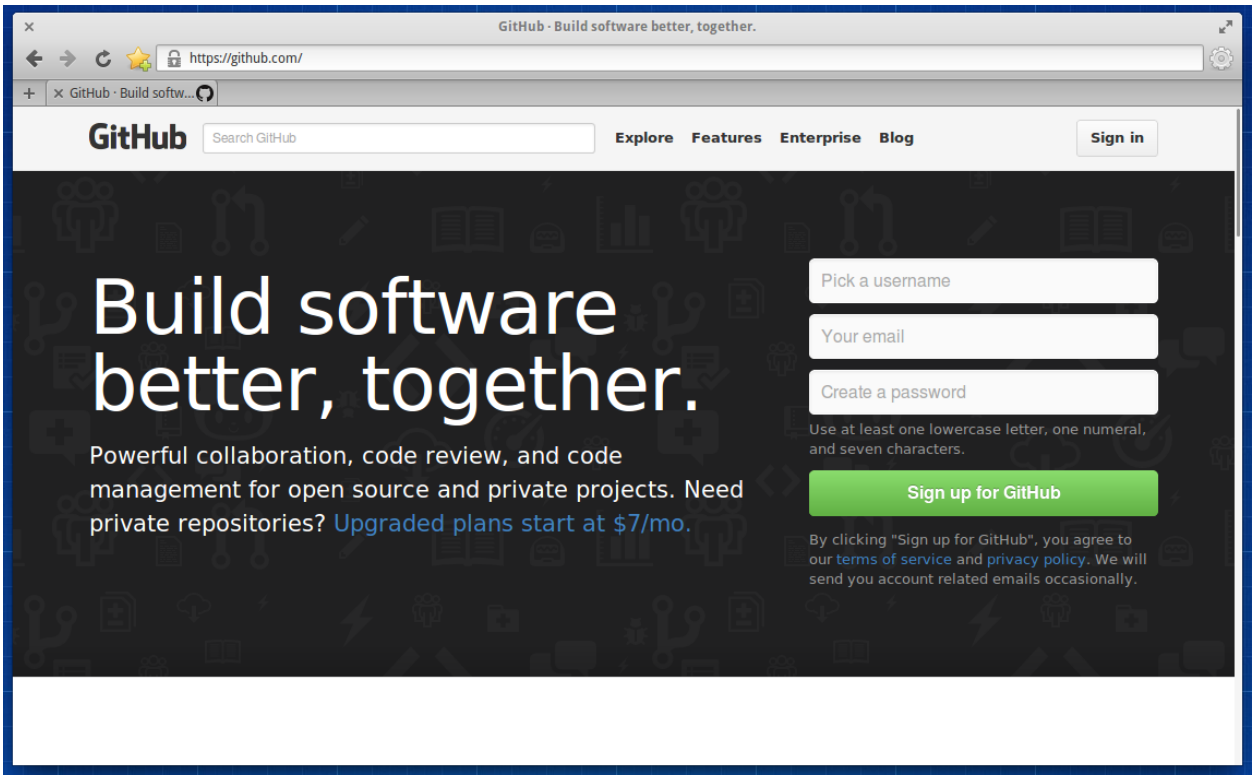
```
git config --global push.default simple
```

Création d'un compte Github

Afin de toujours avoir une sauvegarde en ligne et de pouvoir travailler à plusieur, il est important de publier nos fichiers sur internet. Pour se faire, nous utilisons Github, qui est gratuit pour les projets sous licence libre.

Si vous n'avez pas déjà un compte, rendez vous à l'adresse suivante : https://github.com/

Sur la page qui apparait, remplissez le formulaire avec un nom d'utilisateur, votre email et un mot de passe.



Il ne vous reste plus qu'à cliquer sur le bouton signup.

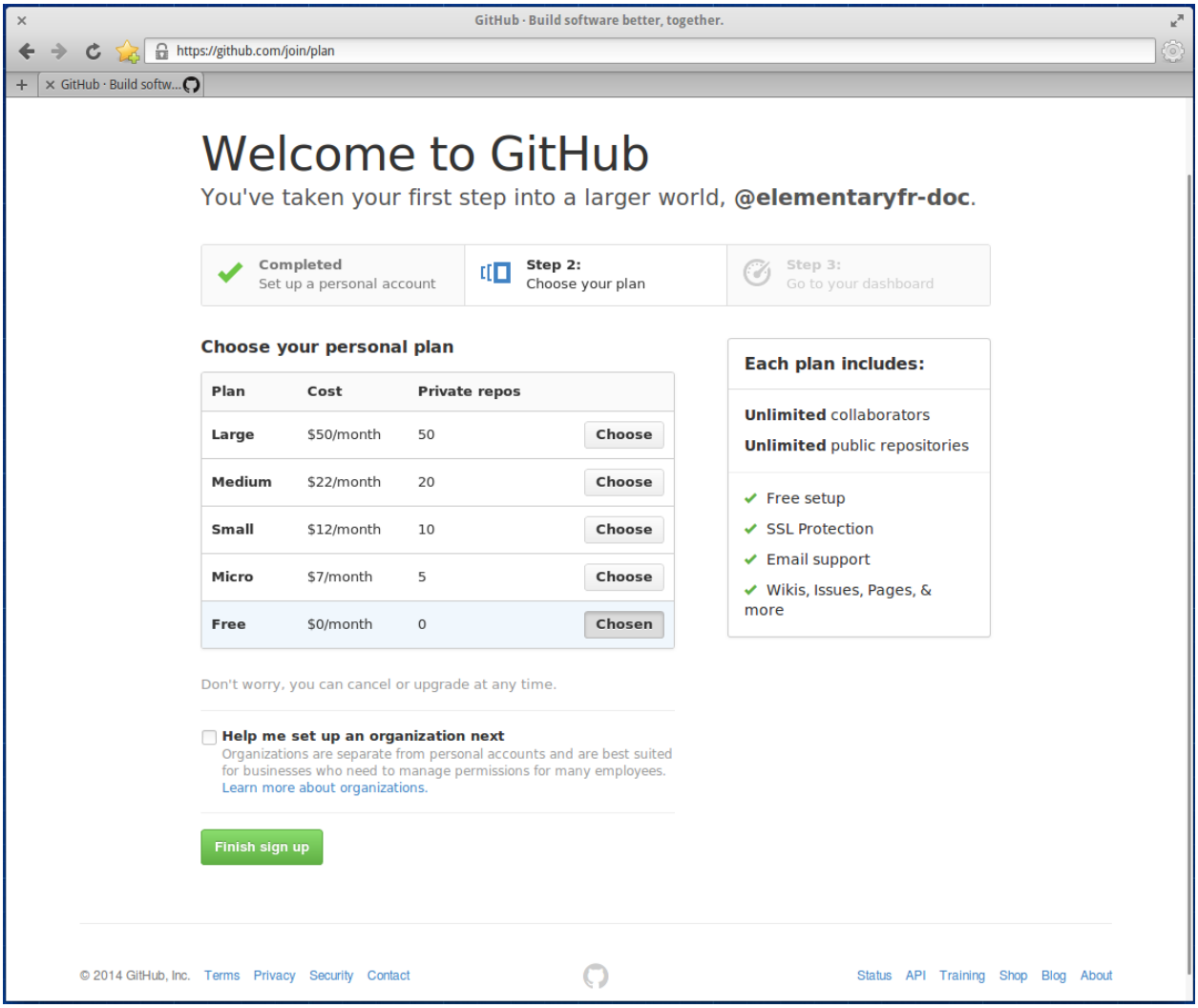
Sur la page suivante, cliquer sur Finish sign up en laissant les choix par défaut.

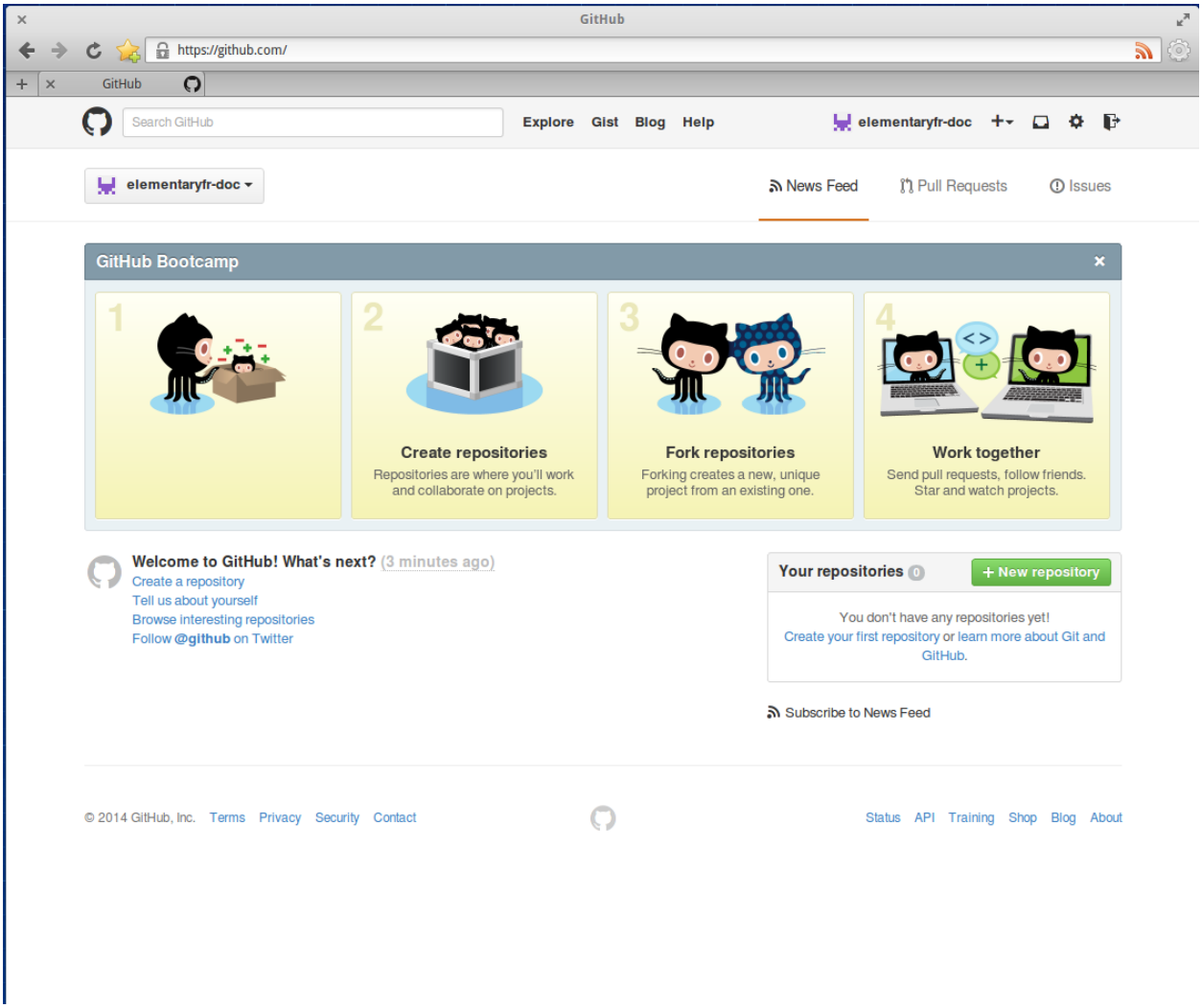
Et voilà, la création du compte est finie.

4.2.2 Gestion de la distribution

Création d'un compte launchpad

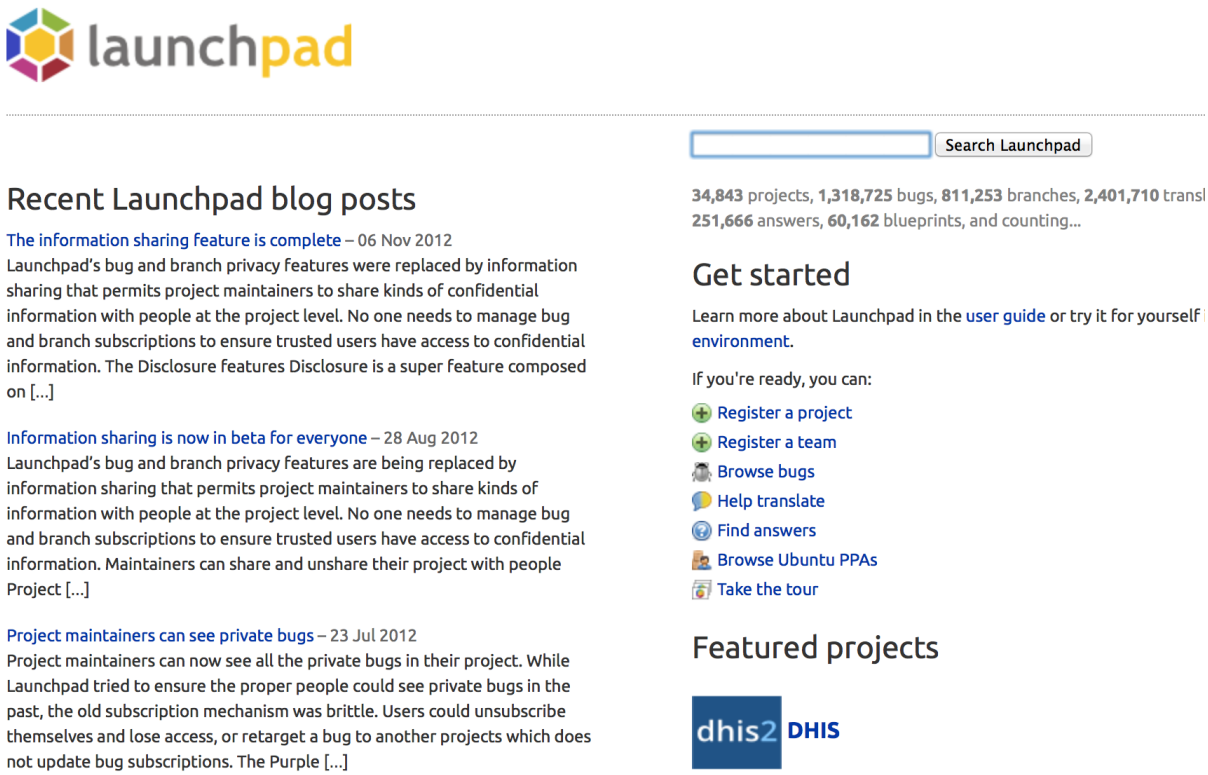
Tout développeur elementary se doit d'avoir un compte sur Launchpad. Launchpad est la plateforme de gestion du projet Ubuntu, sur lequel se base Elementary OS. Cette plateforme permet de distribuer facilement nos logiciels en les


GitHub · Build software better, together.
https://github.com/join/plan
GitHub · Build softw...
Welcome to GitHub
You've taken your first step into a larger world, @elementaryfr-doc.
Completed
Set up a personal account
Step 2:
Choose your plan
Step 3:
Go to your dashboard
Choose your personal plan
Plan
Cost
Private repos
Large
$50/month
50
Choose
Medium
$22/month
20
Choose
Small
$12/month
10
Choose
Micro
$7/month
5
Choose
Free
$0/month
0
Chosen
Each plan includes:
Unlimited collaborators
Unlimited public repositories
Free setup
SSL Protection
Email support
Wikis, Issues, Pages, & more
Don't worry, you can cancel or upgrade at any time.
Help me set up an organization next
Organizations are separate from personal accounts and are best suited for businesses who need to manage permissions for many employees.
Learn more about organizations.
Finish sign up
© 2014 GitHub, Inc.
Terms
Privacy
Security
Contact
Status
API
Training
Shop
Blog
About


GitHub
https://github.com/
GitHub
Search GitHub
Explore
Gist
Blog
Help
elementaryfr-doc
elementaryfr-doc
News Feed
Pull Requests
Issues
GitHub Bootcamp
1
2
Create repositories
Repositories are where you'll work and collaborate on projects.
3
Fork repositories
Forking creates a new, unique project from an existing one.
4
Work together
Send pull requests, follow friends. Star and watch projects.
Welcome to GitHub! What's next? (3 minutes ago)
Create a repository
Tell us about yourself
Browse interesting repositories
Follow @github on Twitter
Your repositories 0
+ New repository
You don't have any repositories yet!
Create your first repository or learn more about Git and GitHub.
Subscribe to News Feed
© 2014 GitHub, Inc.
Terms
Privacy
Security
Contact
Status
API
Training
Shop
Blog
About

intégrants dans l'application Software Center.

Bien sûr si vous avez déjà un compte, vous pouvez passer cette partie.



Pour créer un compte cliquez en haut à droite sur Create account. Ou allez directement sur https://login.launchpad.net/+login

Après rien de bien compliqué, vous pouvez compléter les infos de votre compte à votre guise.

4.2.3 Authentification par clé SSH

Création d'une clé SSH

Maintenant nous avons un compte Launchpad, nous allons en profiter pour lui ajouter une paire de clé ssh. Dans un premier temps, nous allons générer ces clés. Ces clés sont une sorte de reconnaissance digitale de votre ordinateur. Une fois la clé publique importée sur Launchpad, celui-ci reconnaitra automatiquement votre machine, lorsque vous communiquerez avec lui depuis la ligne de commande.

On va installer l'outil nécessaire pour générer la clé et ensuite créer cette dernière, avec Terminal :

```
$ sudo apt-get install openssh-client
$ ssh-keygen -t rsa
```

Note : Par convention, on préfixe souvent les commandes au terminal par un $. Il ne faut pas copier ce caractère, sinon la commande ne fonctionnera pas.

Suivez les instructions, à un moment on vous demandera un mot de passe, retenez le bien car il vous sera demandé lorsque vous enverrez vos modifications de code vers Launchpad.

Ajout de la clef sur Launchpad

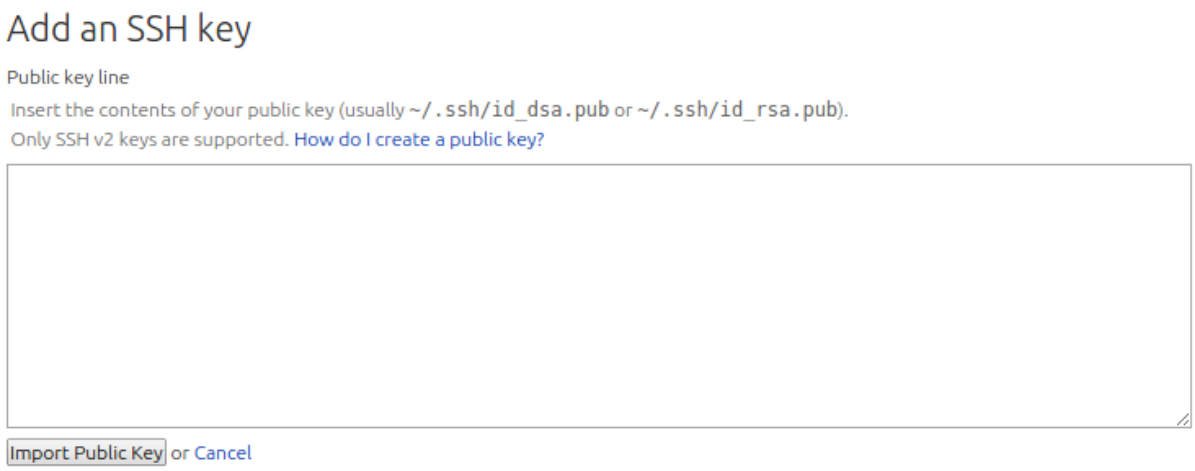
Votre clé publique est donc créée et nous allons l'ajouter à Launchpad.

Rendez-vous sur la page de gestion des clés SSH de votre compte : https://launchpad.net/people/+me/+editsshkeys

Avec Terminal, nous allons ouvrir le fichier comportant la clé publique :

```
$ scratch-text-editor ~/.ssh/id_rsa.pub
```

Copier le contenu du fichier pour le coller dans la zone adéquate sur la page Launchpad :



Ensuite vous n'avez plus qu'à cliquer sur le bouton. Si vous comptez développer sur plusieurs machines, il faudra importer chaque clé par ordinateur.

Maintenant on peut vérifier si tout va bien avec bzr. Bazaar (ou bzr en ligne de commande) est le gestionnaire de gestion de code utilisé sur Launchpad, nous verrons plus loin comment l'utiliser avec Launchpad. Toutefois, on peut déjà signaler à bzr quelle est notre identité Launchpad, Remplacez your-launchpad-id par votre nom d'utilisateur Launchpad.

```
$ sudo apt-get install bzr
$ bzr launchpad-login your-launchpad-id
```

Ajout de la clef sur Github

De la manière, aller sur la page suivante : https://github.com/settings/ssh

Une fois dessus, un clic sur le bouton "Add SSH Key" va vous permettre d'ajouter votre clef.

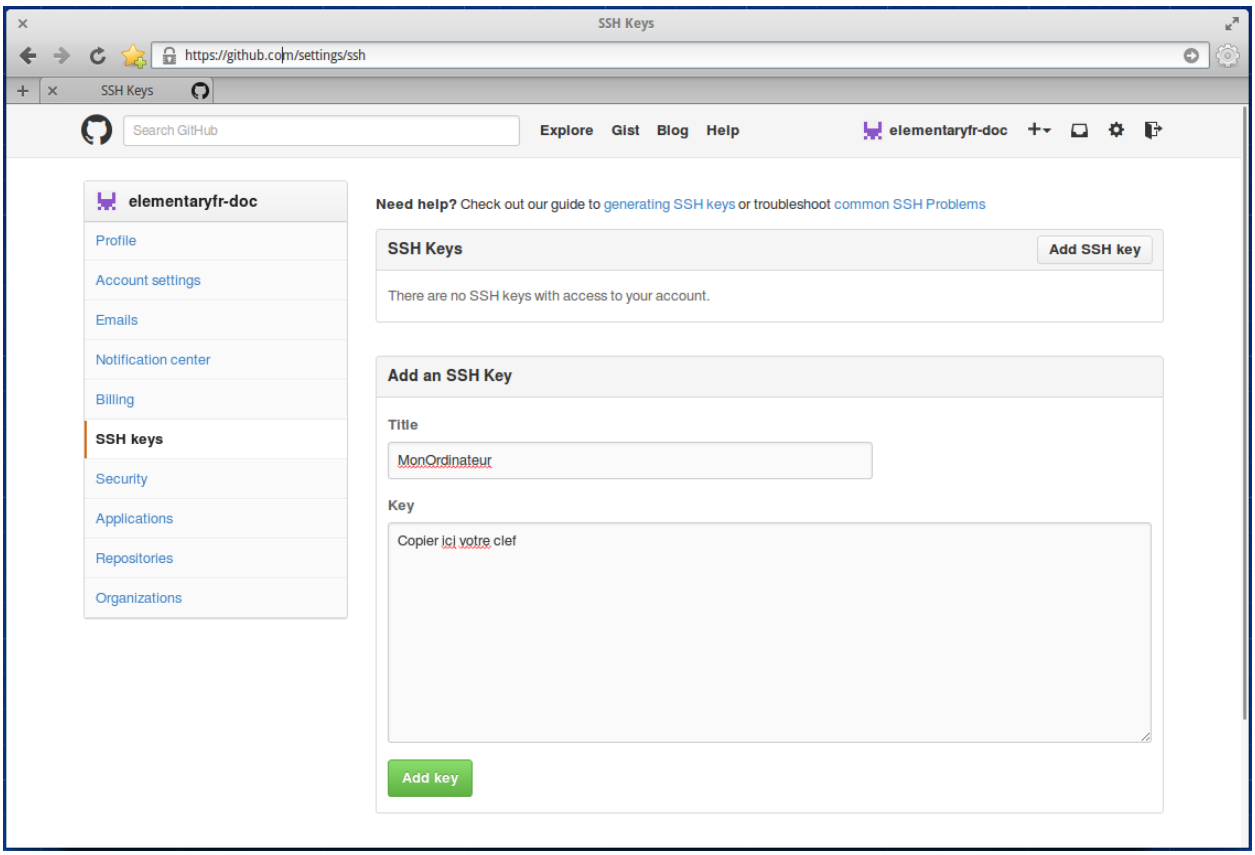
4.2.4 Installation du SDK d'Elementary OS

Une application Elementary est crée en utilisant certaines technologies, qui rendront votre projet intégré au système. En réutilisant ces outils, vos applications ressembleront à celle que vous avez déjà sur votre machine.

Pour installer ces outils de développement, lancer la commande suivante :

```
$ sudo apt-get build-dep granite-demo
```

Avec cette commande nous allons installer les dépendances de construction de granite-demo qui seront suffisantes pour compiler un projet de base.



4.3 Première application

Dans un premier temps nous allons créer dans le dossier personnel, un dossier où seront situés tout vos projets et on le nommera Projects comme les pros sur elementary :D

```
$ mkdir ~/Projects
```

Nos outils seront l'éditeur de texte Scratch pour écrire notre code et Terminal pour exécuter certaines commandes. Ces deux applications sont installées par défaut sur elementary OS.

4.3.1 Bases de Vala

Nous allons créer un petit projet tout simple en Vala, le code utilisé pour les applications elementary. Notre projet sera un simple Hello World dans une fenêtre GTK. Créons le dossier pour ce projet puis un sous-dossier src où nous mettrons notre fichier Vala comportant notre code.

```
$ cd ~/Projects
$ mkdir gtk-hello
$ cd gtk-hello
$ mkdir src
```

Avec Scratch nous allons créer notre fichier vala et commencer son écriture :

```
$ cd src
$ touch gtk-hello.vala
$ scratch-text-editor gtk-hello.vala
```

Passons à l'écriture, tout d'abord nous mettons ce code pour initialiser notre code vala :

```
int main (string[] args) {
   Gtk.init (ref args);
}
```

Maintenant nous allons insérer une fenêtre avec quelques paramètres à l'intérieur du code précédent, donc avant le } ajoutez ceci :

```
    var window = new Gtk.Window ();
    window.title = "Hello World!";
    window.set_border_width (12);
    window.set_position (Gtk.WindowPosition.CENTER);
    window.set_default_size (350, 70);
    window.destroy.connect (Gtk.main_quit);
    Gtk.main ();
    return 0;
```

Donc en résumé, on crée une fenêtre portant la variable window, qui aura pour titre HelloWorld !, avec une bordure de 12, positionner au centre de l'écran, avec une taille de 350x70, et enfin si on ferme cette fenêtre l'action sera que l'on quitte l'application.

Ce n'est pas encore terminé, nous avons fenêtre principale mais nous allons lui ajouter un bouton donc avant Gtk,main() on y ajoute ceci :

```
    var button_hello = new Gtk.Button.with_label ("Click me!");
    button_hello.clicked.connect (() => {
       button_hello.label = "Hello World!";
       button_hello.set_sensitive (false);
    });

    window.add (button_hello);
    window.show_all ();
```

Donc ici, on ajoute un bouton nommé par la variable button_hello, qui aura comme étiquette (label) Click Me ! Et quand on clique dessus cela affichera le label Hello World !.

Donc les deux dernières, on ajoute notre bouton à notre fenêtre, notez qu'on a réutilisé le même nom de variable entre les parenthèses. Au final vous devriez obtenir ceci :

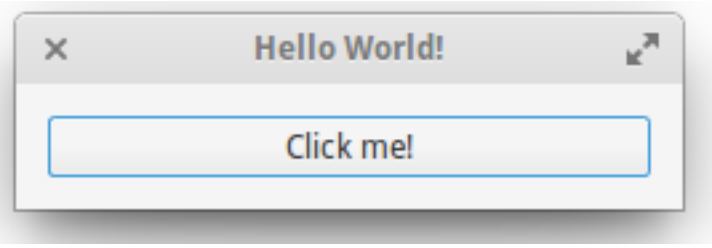
```
int main (string[] args){
    Gtk.init (ref args);

    var window = new Gtk.Window ();
    window.title = "Hello World!";
    window.set_border_width (12);
    window.set_position (Gtk.WindowPosition.CENTER);
    window.set_default_size (350, 70);
    window.destroy.connect (Gtk.main_quit);

    var button_hello = new Gtk.Button.with_label ("Click me!");
    button_hello.clicked.connect (() => {
        button_hello.label = "Hello World!";
        button_hello.set_sensitive (false);
    });

    window.add (button_hello);
    window.show_all ();

```

```
20      Gtk.main();
21      return 0;
22  }
```

Maintenant on va compiler notre fichier vala (qui va créer un fichier éxécutable) et le tester. Si des erreurs sont signalées, revérifiez votre code.

```
$ valac --pkg gtk+-3.0 gtk-hello.vala
$ ./gtk-hello
```

Et donc vous devriez avoir votre petite application :



Sympas non ? Poussons notre code sur Launchpad maintenant !

4.3.2 Bazaar

Comme je l'avais dit plutôt Bazaar est le gestionnaire qui va vous permettre d'envoyer votre code source sur un dépôt de code (à ne pas confondre avec les dépôts PPA) sur Launchpad. Placez votre terminal au niveau de ~/Projects/gtk-hello, On va déjà déclarer votre Pseudo et votre email, ceci est à faire une seule fois.

Important : Cette partie contient des exemples que vous devez modifier avec vos informations.

Et là on remplace "votre-Pseudo" par votre pseudo et le mail qui va bien ;-)

```
$ bzr whoami "votre-Pseudo <votre@email.publique>"
```

On va initialiser notre dossier pour bzr :

```
$ bzr init
```

Et on va ajouter notre fichier (le dossier src sera aussi ajouté) :

```
$ bzr add src/gtk-hello.vala
```

Puis on va écrire un commit (un bref résumé des modifications apportées au code) :

```
$ bzr commit -m "Create initial structure. Create window with button."
```

Et là on envoie notre fichier sur nôtre dépôt +junk/gtk-hello chez Launchpad, pensez à indiquer votre login Launchpad dans la commande :

```
$ bzr push lp:~votre-login-launchpad/+junk/gtk-hello
```

Et voilà, jetez un coup d'oeil sur Launchpad : https://code.launchpad.net/people/+me/

Bien joué ! Il est temps de faire une application un peu plus complexe et surtout plus complète, et d'ensuite d'en faire un paquet.

4.4 Premier projet

De retour dans notre dossier Projects, on créer un sous-dossier hello-world

```
$ mkdir hello-world
```

Puis dedans on créer un autre dossier pour notre nouvelle application.

```
$ cd hello-world
$ mkdir hello-again
```

Créons le dossier src puis éditons notre nouveau fichier hello-again.vala

```
$ cd hello-again
$ mkdir src
$ cd src
$ touch hello-again.vala
$ scratch-text-editor hello-again.vala
```

Nous allons écrire un entête pour le copyright comme pour une vraie application :

```
/***
 Copyright © 2014 Votre nom <email@example.com>

 This file is part of Hello Again.

 Hello Again is free software: you can redistribute it and/or
 modify it under the terms of the GNU General Public License as published
 by the Free Software Foundation, either version 3 of the License, or
 (at your option) any later version.

 Hello Again is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with Hello Again. If not, see <http://www.gnu.org/licenses/>.
***/
```

Cette fois on reprend la même structure de base que gtk-hello.vala mais on vas juste mettre un Gtk.Label au lieu d'un Gtk.Button :

```
var label = new Gtk.Label ("Hello Again World!");
```

Et pensez à l'ajouter à votre fenêtre ! Donc si vous avez compris, vous avez déjà écris : window.add (label) ;

Si vous avez fait comme il fallait, vous devriez avoir ceci :

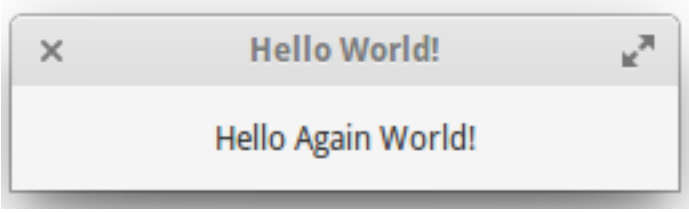
```
/***
 Copyright © 2014 Votre nom <email@example.com>

 This file is part of Hello Again.

 Hello Again is free software: you can redistribute it and/or
 modify it under the terms of the GNU General Public License as published
 by the Free Software Foundation, either version 3 of the License, or
 (at your option) any later version.

```

```
 Hello Again is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with Hello Again. If not, see <http://www.gnu.org/licenses/>.
***/

int main (string[] args) {
    Gtk.init (ref args);

    var window = new Gtk.Window ();
    window.title = "Hello World!";
    window.set_border_width (12);
    window.set_position (Gtk.WindowPosition.CENTER);
    window.set_default_size (350, 70);
    window.destroy.connect (Gtk.main_quit);

    var label = new Gtk.Label ("Hello Again World!");

    window.add (label);
    window.show_all ();

    Gtk.main ();
    return 0;
}
```

Compiler l'appli et testez là !

```
$ valac --pkg gtk+-3.0 hello-again.vala
$ ./hello-again
```

Tadaa !



Notre première application fonctionne et nous allons la mettre sur Launchpad, reprenez les commandes indiquées en partie 2.2 mais cette fois nous mettrons ce fichier sur un autre dépôt :

```
$ bzr push lp:~votre-login-launchpad/+junk/hello-again
```

Vous pouvez vérifier si cela a bien marché en allant sur launchpad : https://code.launchpad.net/people/+me/

Maintenant nous allons créer des fichiers indispensables pour agrémenter notre application

4.4.1 Les fichiers indispensables

Nous allons rajouter à notre application un raccourci qui apparaîtra dans le menu Applications. On vas déjà créer un dossier data dans notre dossier hello-again

```
$ mkdir data
```

Entrons dedans et créons un fichier .desktop

```
$ cd data
$ touch hello-again.desktop
$ scratch-text-editor hello-again.desktop
```

Ce type de fichier répond à des normes spécifiques pour elementary, elles sont consultables ici : http://elementaryos.org/docs/human-interface-guidelines/app-launchers

Passons à l'écriture :

```
[Desktop Entry]
Name=Hello Again
GenericName=Hello World App
Comment=Proves that we can use Vala and Gtk
Categories=GTK;Utility;
Exec=hello-again
Icon=application-default-icon
Terminal=false
Type=Application
X-GNOME-Gettext-Domain=hello-again
X-GNOME-Keywords=Hello;World;Example;
```

Donc rapidement, on a crée un raccourci par rapport à notre application Hello Again comportant un nom, un nom générique, une description, la commande d'exécution, l'icône qui sera utilisée, le type...

Vous pouvez déjà commiter ce nouveau fichier :

```
$ cd ..
$ bzr add data/hello.desktop
$ bzr commit -m "Added a .desktop file"
$ bzr push
```

Notez qu'il suffit juste d'utiliser push maintenant que bzr sait sur quel dépôt de code il doit envoyer les modifications.

Maintenant nous allons aborder la partie « juridique » de la chose. À la racine du projet on crée deux fichiers : AUTHORS et COPYING (en lettres capitales !)

Dans AUTHORS, on indique tous les participants du projet, de cette façon :

```
votre Nom <vous@emailaddress.com>
Votre Ami <ami@emailaddress.com>
```

Dans COPYING, on place la licence du projet, pour info les applications elementary utilisent généralement la GPL : http://www.gnu.org/licenses/quick-guide-gplv3.html

Vous pouvez récupérer le COPYING de mon dépôt : http://bazaar.launchpad.net/~devil505/+junk/hello-packaging/view/head:/COPYING

Maintenant vous pouvez commit ces deux fichiers comme nous l'avons fait pour le fichier .desktop

Maintenant attardons-nous la compilation de notre application.

4.4.2 La compilation

Pour la compilation nous allons utiliser Cmake http://www.cmake.org/ qui est utilisé par la plupart des applications elementary.

Cmake et ses modules se trouve sur un dépôt d'elementary que nous allons récupérer, placez- vous le dossier ~/Projects.

```
$ bzr branch lp:~elementary-apps/+junk/cmake-modules
```

Dans le dossier cmake-modules, vous avez un dossier cmake, copiez-le et collez-le dans le dossier hello-again.

Dans le dossier hello-again, on crée un fichier CmakeLists.txt

```
$ touch CmakeListe.txt
$ scratch-text-editor CmakeLists.txt
```

Dans ce fichier copiez ce qui suit, pas avec les commentaires qui sont là pour vous expliquer à quoi corresponde telles ou telles lignes :

```
# le nom du projet
project (hello-again)

# la version moins récente de cmake que nous pouvons supporter
cmake_minimum_required (VERSION 2.6)

# dire à cmake quels modules sont présents dans le dossier de notre projet
list (APPEND CMAKE_MODULE_PATH ${CMAKE_SOURCE_DIR}/cmake)

# où installer le dossier data si besoin est
set (DATADIR "${CMAKE_INSTALL_PREFIX}/share")

# ici on dit où installer le projet
set (PKGDATADIR "${DATADIR}/hello-again")

set (EXEC_NAME "hello-again")
set (RELEASE_NAME "A hello world.")
set (VERSION "0.1")
set (VERSION_INFO "whats up world")

# on vas utiliser pkgconfig pour vérifier que les dépendances sont installées, mais avant, allons le
find_package(PkgConfig)

# maintenant vérifions les dépendances requises
pkg_check_modules(DEPS REQUIRED gtk+-3.0)

add_definitions(${DEPS_CFLAGS})
link_libraries(${DEPS_LIBRARIES})
link_directories(${DEPS_LIBRARY_DIRS})

# pour être sûr que nous avons vala
find_package(Vala REQUIRED)
# pour être sûr qu'on utilise vala
include(ValaVersion)
# pour être sûr de la version de vala utilisée. ensure_vala_version("0.16" MINIMUM)

# les fichiers que nous voulons compiler
include(ValaPrecompile)
vala_precompile(VALA_C ${EXEC_NAME}
```

```
40
41  src/hello-again.vala
42
43  # on dit quelles bibliothèques utilisées pour la compilation
44  PACKAGES
45
46      gtk+-3.0
47  )
48
49  # on dit à cmake d'appeler l'éxécutable que nous venons de créer
50  add_executable(${EXEC_NAME} ${VALA_C})
51
52  # ceci installe le binaire résultant de la compilation
53  install (TARGETS ${EXEC_NAME} RUNTIME DESTINATION bin)
54
55  # ceci installe le fichier .desktop pourqu'il soit disponible dans le menu Applications
56  install (FILES ${CMAKE_CURRENT_SOURCE_DIR}/data/hello.desktop DESTINATION ${DATADIR}/applications/)
```

Toujours dans le dossier hello-again, on crée un dossier nommé build.

Puis avec le Terminal dans ce dossier, nous allons lancer la construction par Cmake :

```
$ cd build
$ cmake -DCMAKE_INSTALL_PREFIX=/usr ../
```

Puis on compile de manière classique :

```
$ make
```

Enfin, on vas même installer notre application :

```
$ sudo make install
```

Vous avez donc deux nouveaux fichiers dans votre système :

```
/usr/bin/hello-again
/usr/share/applications/hello.desktop
```

Regardez dans votre menu Applications :

4.4.3 Faire un paquet

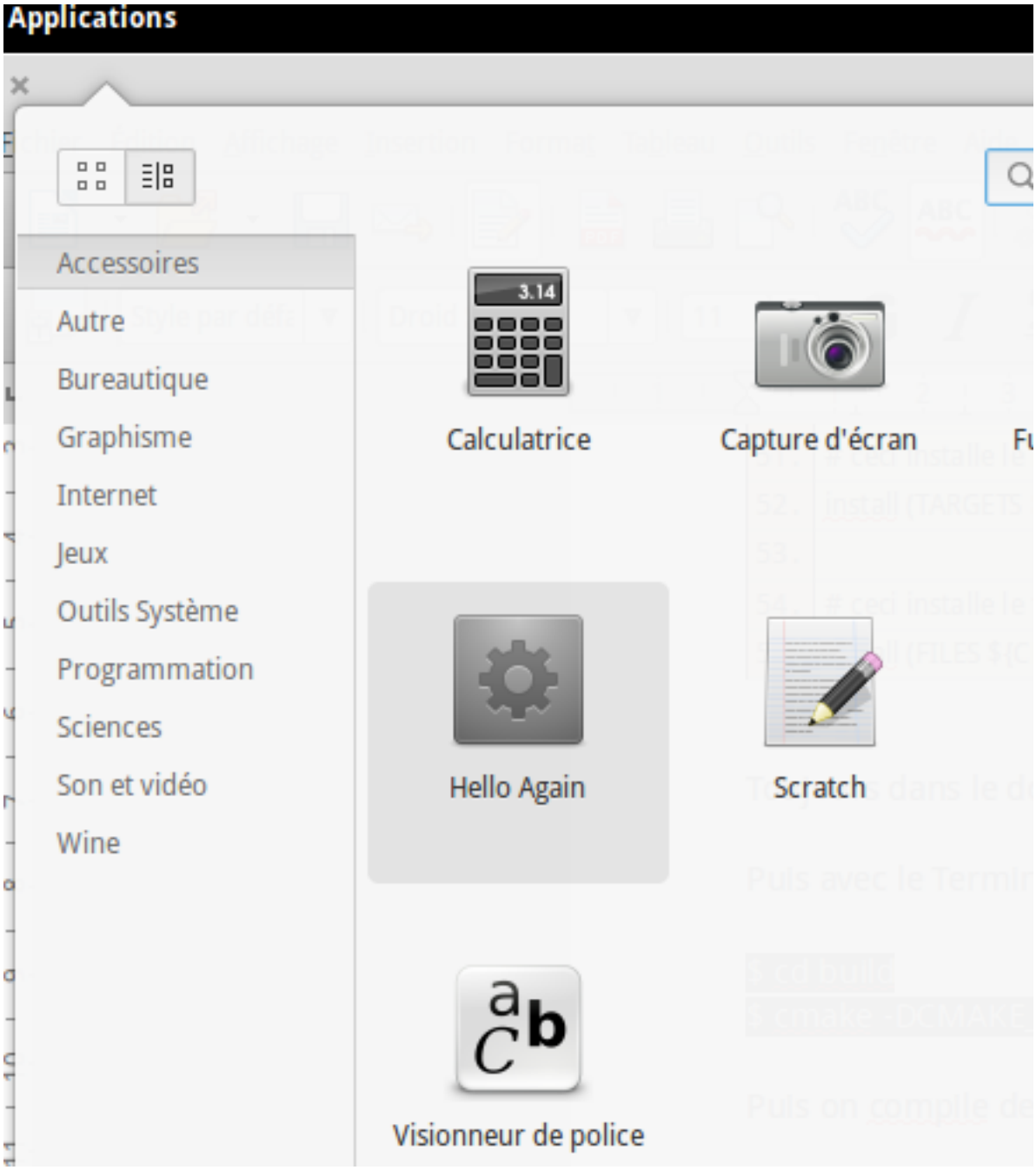
Nous avons maintenant une application simple mais complète. Essayons d'en faire un paquet qui sera disponible à tous sur votre dépôt PPA.

On va créer un nouveau dossier dans hello-world dédié au packaging et nous allons l'appeler : deb-packaging

Dans deb-packaging nous allons copier le contenu de notre application hello-again (src,data,cmake,AUTHORS,COPYING,CmakeLists,txt) sauf le dossier build.

Nous allons avoir besoin de fichiers spécifiques aux paquets debian et nous allons les chercher avec bzr mais avant revenez dans ~/Projects pour ne pas tout mélanger :

```
$ cd ~/Projects
$ bzr branch lp:~elementary-apps/+junk/debian-template
```


Applications
Accessoires
Autre
Bureautique
Graphisme
Internet
Jeux
Outils Système
Programmation
Sciences
Son et vidéo
Wine
Calculatrice
Capture d'écran
Hello Again
Scratch
Visionneur de police

Dans le dossier debian-template se trouve un dossier debian que vous copier dans deb- packaging.

Puis aller dans ~/Projects/hello-world/deb-packaging/debian et regardez le fichier changelog. Ce fichier doit être mis à jour à chaque nouvelle version de votre futur paquet.

Comme votre application en est sa premiès version, ce fichier doit ressembler à ceci :

```
hello-packaging (0.1) precise; urgency=low

* Initial Release.

-- Your Name <you@emailaddress.com> Tue, 9 Apr 2013 04:53:39 -0500
```

Astuce : On peut utiliser le programme dch pour générer les entées

En premier vous indiquez la version du paquet, la version d'ubuntu (Luna étant basée sur Precise) et l'urgence de la construction par les serveurs de Launchpad. Ce n'est pas critique, donc nous avons opté pour low,

Ensuite vous indiquez votre Nom et votre email. Enfin à la dernière ligne vous placez la date et l'heure (ne pas oublier le fuseau horaire) actuelles.

Si vous sortez une nouvelle version de votre paquet, il ne faudra pas effacer ou remplacer ses lignes, vous réécrivez un nouveau paragraphe de 5 lignes comme l'exemple en haut avec le nouveau numéro de version, le nouvel horodatage...

Maintenant allons nous occuper du fichier control, ce fichier recueille les infos de votre futur paquet .deb. Voici à qui il doit ressembler :

```
Source: hello-again
Section: x11
Priority: extra
Maintainer: Your Name <you@emailaddress.com>
Build-Depends: cmake (>= 2.8),
debhelper (>= 8.0.0),
valac-0.24 | valac (>= 0.24),
libgtk-3-dev (3.12.2)

Standards-Version: 3.9.3

Package: hello-again
Architecture: any
Depends: ${misc:Depends}, ${shlibs:Depends}
Description: Hey young world
This is a Hello World written in Vala using the Autovala build system.
```

Donc ici on indique le nom et l'email du mainteneur du paquet, les dépendances de construction, les dépendances (notez la séparation par des virgules), la description de l'application...

Passons au fichier copyright que l'on va modifier pour obtenir ceci :

```
Format: http://dep.debian.net/deps/dep5
Upstream-Name: hello-again3.
Source: https://code.launchpad.net/~junrrein/+junk/hello-packaging

Files: cmake/* data/* debian/* doc/* po/* src/*
Copyright: 2014 Votre Nom
License: GPL-3.0+


License: GPL-3.0+
This program is free software: you can redistribute it and/or modify
```

```
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
.
This package is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
.
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.
.
On Debian systems, the complete text of the GNU General
Public License version 3 can be found in "/usr/share/common-licenses/GPL-3".
```

Pour plus de détails sur le packaging Debian, consulter ce lien : http://www.debian.org/doc/debian-policy/

Maintenant, nous allons nous occuper des recettes Launchpad (recipes en anglais) pour la construction de votre paquet.

Rendez-vous sur cette page : https://code.launchpad.net/people/+me/+junk/deb-packaging/+new-recipe

Ici cochez les options comme vous le désirez, le plus important étant la case Recipe Text dont vous allez effacer le contenu par défaut pour le remplacer par :

```
# bzr-builder format 0.3 deb-version {debupstream}+r{revno}-0
```

Ensuite appuyer sur Create Recipe.

Les serveurs de Launchpad vont crée les paquets pour les différentes architectures et pour les différentes versions d'Ubuntu que vous aurez choisi au moment de la création de la recipe.

Vous ne devrez pas tarder à voir votre paquet sur votre dépôt PPA : https://launchpad.net/people/+me/+archive/ubuntu/deb-packaging

Et voilà vous avez packagé votre application !

4.5 Plus de possibilités avec Vala

Dans cette partie nous allons nous amuser un peu avec Vala et ses possibilités vis à vis de Gtk. Nous verrons la fonction Gtk.Grid et Attach. Bien sûr vous trouverez plein d'infos sur http://valadoc.org/ Reprenez votre fichier hello-again.vala

4.5.1 Gtk.Grid

Cette fois au lieu d'ajouter un bouton ou label, nous allons utiliser Gtk.Grid qui peut être considéré comme une sorte de grille. Voici le code à insérer :

```
var grid = new Gtk.Grid ();
grid.orientation = Gtk.Orientation.VERTICAL;
```

On a donc une nouvelle variable nommée grid qui est un Gtk.Grid ayant une orientation, si on avait pas mis cette seconde ligne, la grille serait automatiquement en position horizontale (valeur par défaut) On peut ajouter des labels à l'intérieur de cette grille :

```
grid.add (new Gtk.Label ("Label 1"));
grid.add (new Gtk.Label ("Label 2"));
```

Puis enfin on va ajouter notre variable grid à la fenêtre principale :

```
this.add (grid);
```

Au final, si vous avez tout compris vous devriez avoir un fichier comme ceci : Ensuite on compile et on teste, souvenez-vous :

```
$ valac --pkg gtk+-3.0 hello-again.vala
$ ./hello-again
```

Et voici l'application :

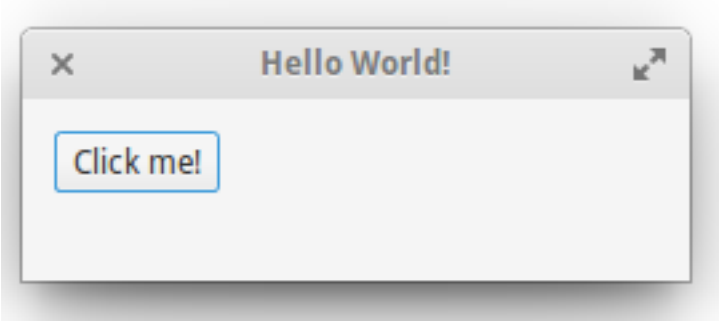
```
/* Copyright 2013 YourName
*
* This file is part of Hello Again
** Hello Again is free software: you can redistribute it
* and/or modify it under the terms of the GNU General Public License as
* published by the Free Software Foundation, either version 3 of the
* License, or (at your option) any later version.
*
* Hello Again is distributed in the hope that it will be
* useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
* Public License for more details
*
* You should have received a copy of the GNU General Public License along
* with Hello Again. If not, see http :// www . gnu . org / licenses /.
*/

int main (string[] args) {
    Gtk.init (ref args);

var window = new Gtk.Window ();
window.title= "Hello World!";
window.set_border_width (12);
window.set_position (Gtk.WindowPosition.CENTER);
window.set_default_size (350, 70);
window.destroy.connect (Gtk.main_quit);

var label = new Gtk.Label ("Hello Again World!");

window.add (label);
window.show_all ();

Gtk.main ();
return 0;
}
```

Regardez bien, on a donc une grille de position verticale et avec un label dans chaque cellule. Maintenant voyons un autre exemple :

```
var grid = new Gtk.Grid ();
grid.orientation = Gtk.Orientation.VERTICAL;
grid.row_spacing = 6;
```

```
4
5  var button = new Gtk.Button.with_label ("Click me!");
6  var label = new Gtk.Label (null);
7
8  grid.add (button);
9  grid.add (label);
10
11 window.add (grid);
```

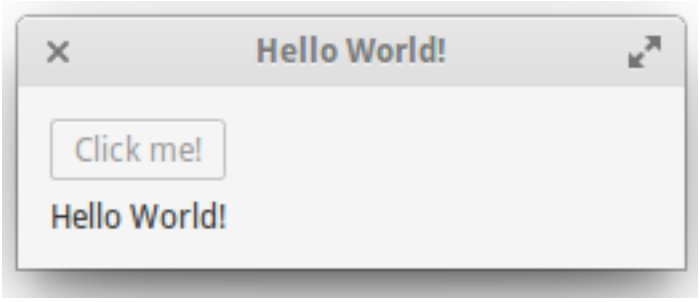
Toujours une grille verticale et avec un espacement de 6 pour chaque cellule. On ajoute un bouton (avec le label Click me !) et un label vide. Puis on ajoute les variables button et label à la grille. Voici le résultat :



On retrouve notre grille avec le bouton Click me dans la première cellule. Définissons une action sur le bouton, après window.add (grid), rajoutez ce code :

```
1  button.clicked.connect (() => {
2    label.label = "Hello World!";
3    button.sensitive = false;
4
5  });
```

Ici on veut qu'une fois cliqué le bouton affiche un label Hello World ! Avec le sensitive sur false, le bouton restera « enfoncé ». Démonstration :



Amusez-vous à changer les valeurs, par exemple le sensitive sur true, virer la ligne de position verticale...etc :-)

4.5.2 Attach

Nous allons voir la méthode de placement des widgets par la fonction attach. On vas tester une autre grille :

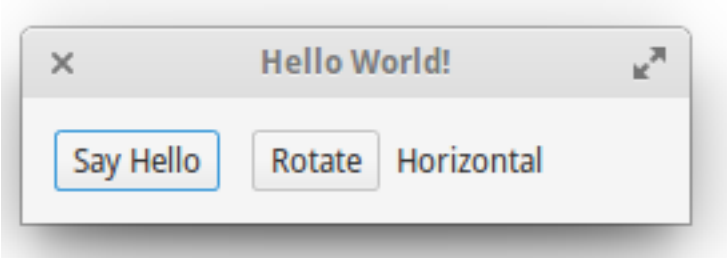
```
var layout = new Gtk.Grid ();
layout.column_spacing = 6;
layout.row_spacing = 6;

var hello_button = new Gtk.Button.with_label ("Say Hello");
var hello_label = new Gtk.Label (null);

var rotate_button = new Gtk.Button.with_label ("Rotate");
var rotate_label = new Gtk.Label ("Horizontal");
```

Nous avons : - une grille nommé layout - les colonnes et les cellules sont espacées de 6 - un bouton (hello_button) avec le label Say Hello - un label (hello_label) vide - un bouton (rotate_button) avec le label Rotate - un label (rotate_label) qui indique Horizontal

Ensuite n'oubliez pas qu'il faut ajouter tout ce petit monde en respectant le nom des variables :

```
layout.add (hello_button);
layout.add (hello_label);
layout.add (rotate_button);
layout.add (rotate_label);

window.add (layout);
```

Voilà le résultat :



Nous avons bien notre grille comportant la première cellule avec le bouton Say Hello, la seconde est visuellement omise car elle affiche un label vide, la troisième avec le bouton rotate et enfin la quatrième avec le label affichant le mot Horizontal.

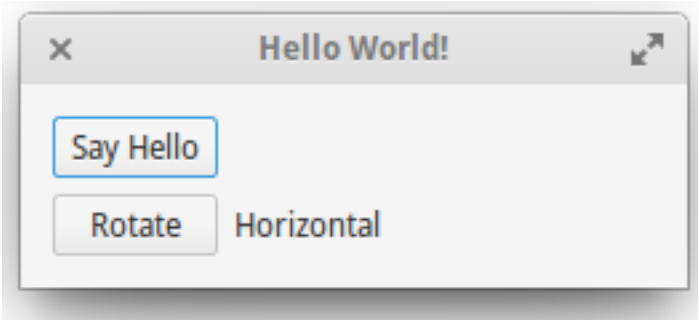
La fonction add rajoute bêtement vos widgets mais avec attach on peut avoir davantage de possibilités, regardons cela ! Remplacez ce morceau :

```
layout.add (hello_button);
layout.add (hello_label);
layout.add (rotate_button);
layout.add (rotate_label);
```

Par :

```
// ajout d'un premier lot de widgets
layout.attach (hello_button, 0, 0, 1, 1);
layout.attach_next_to (hello_label, hello_button, Gtk.PositionType.RIGHT, 1, 1);

// ajout d'un second lot de widgets
layout.attach (rotate_button, 0, 1, 1, 1);
layout.attach_next_to (rotate_label, rotate_button, Gtk.PositionType.RIGHT, 1, 1);
```

Dans un premier temps on rajoute hello_button puis on lui attache hello_label positionné à sa droite. Dans un second temps et avec le même principe, on ajoute rotate_button avec rotate_label situé à sa droite. La fonction attach_next_to permet de placer un widget à côté d'un autre. Le tout sera placé donc dans une grille 2x2. Le résultat :



Notez qu'attach répond à 5 arguments, exemple au-dessus avec : 0, 0, 1, 1 Dans l'ordre : 1 : Correspond au widget que vous rattachez à la grille. 2 : Le nombre de colonne (qui débute à 0). 3 : Le nombre de cellules (qui débute à 0). 4 : Le nombre de colonne où le widget peut s'étendre. 5 : Le nombre de cellule où le widget peut s'étendre. On peut aussi ajouter des fonctions à nos boutons le code suivant après le window.add (layout) ;

```
hello_button.clicked.connect (() => {
  hello_label.label = "Hello World!";
  hello_button.sensitive = false;

});

rotate_button.clicked.connect (() => {
  rotate_label.angle = 90;
  rotate_label.label = "Vertical";
  rotate_button.sensitive = false;

});
```

Le bouton hello_button affichera un label Hello World ! Il restera enfoncé. Le bouton rotate_button affichera un label Vertical pivoté à 90° et restera enfoncé. Démonstration :

Voilà, n'hésitez pas à consulter Valadoc pour découvrir d'autres fonctions :-)

CHAPITRE 5

Les technologies elementary

5.1 Introduction

Important : Ce guide est encore en cours d'écriture. Il peut contenir des informations erronées.

Les erreurs éventuelles peuvent être signaler à l'adresse suivante : https://github.com/Elementary-fr/elementaryos-fr-doc/issues

5.1.1 Auteurs et Licences

Cette partie reprend des idées, du codes et du texte depuis le site officiel de développement d'Elementary OS, disponible à l'adresse suivante : http://elementaryos.org/developer

Ce guide a été créer par la communauté francophone ElementaryOS-FR et ne dois pas être considérer comme une publication officiel de l'équipe Elementary OS.

Le contenu de cette partie est placé sous Licence CC BY-SA 3.0 - https://creativecommons.org/licenses/by-sa/3.0/

5.2 Contractor

Note : Ressource pour l'écriture d'une doc ou d'un tutoriel :

- Exemple de fichier : https://launchpadlibrarian.net/84018890/compress.contract
- Documentation officielle : http://elementaryos.org/docs/human-interface-guidelines/contractor

Les fichiers sont stockés dans */usr/share/contractor/*

5.2.1 Format des fichiers .contract

```
[Contractor Entry]
Name=Archives
Icon=file-roller
Description=Create archive
MimeType=all
Exec=file-roller -d %U
```

5.3 Les Plugs Switchboard

Note : Document en cours d'écriture

Les plugs (pligins) pour Switchboard se retrouvent dans Paramètres Système.

Les fichiers sont stockés dans */usr/lib/x86_64-linux-gnu/switchboard/* puis dans un dossier selon la catégorie.

Il est possible de les coder directement dans un fichier vala et à la compilation le fichier .plug sera crée.

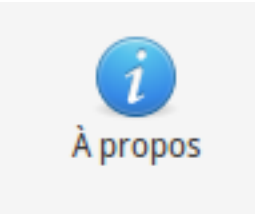
Il existe un template ici : http://bazaar.launchpad.net/~elementary-pantheon/pantheon-plugs/plug-template-isis/files

5.3.1 Format des fichiers .plug

Exemple ici avec About Plug (À Propos)

```
[Plugin]
Module=about
IAge=2
Name=About
Description=About
Authors=Switchboard Developers
Copyright=Copyright © 2013 Switchboard Developers
Website=http://launchpad.net/switchboard
Icon=help-info
X-Category=system
X-GNOME-Gettext-Domain=about-plug
```

On retrouve des similitudes avec les fichiers desktop avec le nom, la description du module, l'icone...



CHAPITRE 6

Lignes directrices

6.1 Introduction

Important : Ce guide est encore en cours d'écriture. Il peut contenir des informations erronées.

Les erreurs éventuelles peuvent être signaler à l'adresse suivante : https://github.com/Elementary-fr/elementaryos-fr-doc/issues

6.1.1 Auteurs et Licences

Cette partie reprend des idées, du codes et du texte depuis le site officiel de développement d'Elementary OS, disponible à l'adresse suivante : http://elementaryos.org/developer

Ce guide a été créer par la communauté francophone ElementaryOS-FR et ne dois pas être considérer comme une publication officiel de l'équipe Elementary OS.

Le contenu de cette partie est placé sous Licence CC BY-SA 3.0 - https://creativecommons.org/licenses/by-sa/3.0/

6.2 Style de code

Important : Cette section est une traduction non-officiel de la page "code style" du site d'elementary OS, disponible à l'adresse suivante : http://elementaryos.org/docs/code/code-style

Tous comme le texte original, et le reste de ce site, ce contenu est publié sous license Creative Commons CC BY-SA 3.0

Le but de ce guide est de fournir des intructions claires et précises sur la manière d'écrire du bon code dans tous les projets elementary. Ces lignes directrices doivent être suivie sur tous les fichiers, de manière à garder notre code cohérent et lisible. Nous héritons aussi de certaines des lignes directrices de Gnome pour Vala, de manière à garder notre code cohérent avec les autres programmes Vala.

Si les lignes directrices proposées ici sont suivie, les nouveaux arrivant pourront plus facilement se joindre au développement d'elementary et comprendre le code. En plus, celà rendra facilitera le travail des développeurs sur des applications dont ils n'ont pas l'habitude, car le code sera conforme à leur attente. Finallement, comme le dit Guido Van Rossum [1] *Le code est plus souvent lu qu'écrit*, il est ainsi crucial d'en avoir un bien écrit.

1. Créateur du language python

6.2.1 Information

Ce guide n'est pas encores finis, bien qu'il a été discuté et partiellement approuvé par les contributeurs d'elementary. [2] Il est sujet au changement dans un futur proche.

6.2.2 Espace blanc

Il n'y a pas d'espace blanc traînant à la fin d'une line, qu'elle soit vide ou non. Il n'y a pas de ligne vide après la déclaration de la fonction

```
public string get_text () {
    string text = search_entry.get_text ();
    return text;
}
```

Un espace est mis avant l'ouverture de parentaise :

```
public string get_text () {}
if (a == 5) return 4;
for (i = 0; i < maximum; i++) {}
my_function_name ();
Object my_instance = new Object ();
```

Un espace est mis dans tous les endrois ou des opérations mathématique sont utilisées, entre les nombres et les opérateurs.

```
c = n * 2 + 4;
```

6.2.3 Indentation

Le code en Vala est indenté en utilisant 4 espaces pour la cohérence et la lisibilité.

Dans les classes, les fonctions, les boucles et le contrôle du flux, la première accolade se met à la fin de la première ligne [3] , suivie par le code indenté et une ligne de fermeture de la fonction avec une accolade fermante.

```
public int my_function (int a, string b, long c, int d, int e) {
    if (a == 5) {
        b = 3;
        c += 2;
        return d;
    }

    return e;
}
```

Dans les conditions et les boucles, si il n'y a qu'une seule ligne de code, les accolades ne sont pas utilisées :

```
if (my_var > 2)
    print ("hello\n");
```

Pour les instructions else et else if, on utilise le style emboités.

2. Dans sa version anglaise
3. One True Brace Style

```
if (a == 4) {
    b = 1;
    print ("Yay");
} else if (a == 3) {
    b = 3;
    print ("Not so good...");
} else {
    b = 5;
    print ("Terrible!");
}
```

6.2.4 Classes et fichiers

Il est recommandé de n'avoir qu'une classe par fichier.

Tous les fichiers ont le nom de la classe qu'il contiennent.

Le code doit être séparé en classes pour permettre une évolution plus facile.

6.2.5 Commentaires

Les commentaires sont soit sur la même ligne que le code, soit sur une ligne à part.

Les commentaires sont indentés sur le coté du code, et les commentaires.

Les commentaires évident doivent être éviter. Il font plus de mal que de bien.

```
/* User chose number five */
if (a == 5) {
    B = 4;           // Update value of b
    c = 0;           // No need for c to be positive
    l = n * 2 + 4;   // Clear l variable
}
```

6.2.6 Nom de variables, de classes et de fonctions

```
my_variable = 5;      // Variable names
MyClass               // Class names
my_function_name ();  // Type/Function/Method names
MY_C       // Constants are all caps with underscores

/* For enum members, all uppercase and underscores */
enum OperatingSystem { // An enum name is the same as ClassesNames
    UBUNTU,
    ELEMENTARY_OS,
    VERY_LONG_OS_NAME
}
```

Il faut également convenir qu'il n'y as pas d'utilisation de la notation Hongroise [4] , ni de mélange entre plusieurs sorte de notation.

4. Voire http://fr.wikipedia.org/wiki/Notation_hongroise

6.2.7 Espace de noms de Vala

Se référer à la GLib n'est pas nécessaire. Si vous voulez afficher quelque chose :

```
1  GLib.print ("Hello World");
2  print ("Hello World");
```

Choisissez la deuxième méthode, elle est plus propre.

6.2.8 Nombre de colonne par ligne

Idéalement, il ne devrait pas y avoir plus de 80 caractères par lignes, car c'est la taille par défaut du terminal. Cependant, exceptionnellement, plus de caractère peuvent être ajouter, parce que les gens ont aujourd'hui des écrans suffisament large.

La limite maximale est de 120 caractère.

6.2.9 Entête de la license GPL

```
1  /***
2    Copyright (C) 2011-2012 Application Name Developers
3    This program is free software: you can redistribute it and/or modify it
4    under the terms of the GNU Lesser General Public License version 3, as published
5    by the Free Software Foundation.
6    This program is distributed in the hope that it will be useful, but
7    WITHOUT ANY WARRANTY; without even the implied warranties of
8    MERCHANTABILITY, SATISFACTORY QUALITY, or FITNESS FOR A PARTICULAR
9    PURPOSE. See the GNU General Public License for more details.
10   You should have received a copy of the GNU General Public License along
11   with this program. If not, see
12 ***/
```

CHAPITRE 7

Indices and tables

— *genindex*
— *modindex*
— *search*