Cours d'Initiation au Rust

Téo Bernier

Contents

Introduction

Rust est un langage de programmation bas niveau. Bien qu'il offre une multitude de fonctionnalités assez puissantes (Possibilité de `Typage dynamique`, `Itérateurs`, `Collections`, `Async`, `Multi-Threading`, ...), il reste tout de même un langage assez proche de la machine quand il s'agit de la mémoire. Son efficacité vient surtout du fait que les abstractions qu'il propose sont astucieusement gérées à la compilation et non pendant le runtime.

Par exemple, même si Rust propose du typage dynamique, la plupart des codes en Rust utilisera principalement du `typage statique` (potentiellement jusqu'à 10 fois plus rapide). De plus, il assure certaines sécurités statiquement (analyse pendant la compilation) comme les `data races`, les `dangling pointers`, et plus généralement, les `segmentation fault`.

Ressources

- Playlist des Cours
- GitHub

Références

- Rust Book
- Rust By Example
- Easy Rust

Partie I

L'utilisation de Cargo

Cargo est le gestionnaire de package / gestionnaire de projet de Rust.

C'est un outil très simple à utiliser, il vous suffira de connaître :

- Gestion de projet :
  - `cargo new project_name`
- Debug static du code :
  - `cargo check`
  - `cargo fmt`
  - `cargo clippy`
- Création d'exécutables :
  - `cargo build`
  - `cargo run`
  - `cargo clean`
- Release d'un exécutable optimisé :
  - `cargo build --release`
  - `cargo run --release`

Les Bases

`println!()`

La première chose que l'on peut voir est évidemment comment faire un `hello world`. On voit ici que la commande utilisée est `println!`.

Le `!` indique ici que ce n'est pas une fonction, mais une macro. Un peu comme `printf` en C.

```rust
fn main() {
    println!("Hello, world !");
    // Output -> "Hello, world !"
}
```

`println!` peut aussi permettre d'afficher des valeurs :

```rust
fn main() {
    println!("{} + {} = {}", 2, 2, 2 + 2);
    // Output -> "2 + 2 = 4"
}
```

On peut aussi afficher des structures plus complexes :

```rust
fn main() {
    println!("{:?}", [1, 2, 3, 4, 5]);
    // Output -> "[1, 2, 3, 4, 5]"
}
```

Le `:` indique qu'il y a des options d'affichage, et le `?` indique que c'est un affichage de Debug. En effet, les arrays n'ont pas d'affichage classique.

Les variables

Assignation

On peut utiliser des variables en Rust en utilisant le mot-clé `let`.

```rust
fn main() {
    let x = 4;
    println!("{}", x);
    // Output -> "4"
}
```

Le type des variables est souvent déterminé par inférence. Mais on peut tout de même le préciser :

```rust
fn main() {
    let x: u8 = 4;
    println!("{}", x);
    // Output -> "4"
}
```

u8 est un type représentant un entier non-signé sur 8 bits.

Types Primitifs

Il y a ainsi pas mal de types primitifs :

- `u8, u16, u32, u64, u128, usize` (unsigned int)
- `i8, i16, i32, i64, i128, isize` (int)
- `f32, f64` (floats)
- `char` (utf-8)
- `bool`
- `[type; N]` (Où N est le nombre d'éléments dans le tableau)
- `(type, type, ..., type)` (Un tuple avec possiblement différents types)
- `[type], str` (Les Slices, ils ont une taille inconnue à la compilation (ne sont pas utilisables comme tels))
- `()` (Représent un type vide (un peu comme None en Python))
- Référence / Pointeur et Fonction (On verra ça plus tard)

Immutabilité & Mutabilité

Une variable est de base immutable. C'est à dire qu'il est interdit de la modifier.

```rust
fn main() {
    let x = 4;
    x = 8; // Ne compilera pas
    println!("{}", x);
}
```

Il faut alors utiliser le mot-clé `mut`.

```rust
fn main() {
    let mut x = 4;
    x = 8;
    println!("{}", x);
    // Output -> "8"
}
```

Les fonctions

La définition de fonction n'est pas très compliquée :

En Rust, tout est expression, fonction comprise. (Ainsi, on peut ne pas mettre de ; à la fin d'une ligne. Elle sera alors considérée comme une valeur de retour).

```rust
fn return_2() -> usize {
    2
}

fn main() {
    let x = return_2();
    // C'est comme si return_2 était
    // évaluée comme un expression
    // étant égale à 2.

    println!("{}", x);
    // Output -> "2"
}
```

On peut aussi explicitement écrire :

```rust
fn return_2() -> usize {
    return 2;
}
```

Ainsi, une fonction sera toujours de la forme :

```
fn function_name(param1: Type1, param2: Type2, ...) -> ReturnType {
    ...
}
```

Les Structures de Contrôle

Comme pour les fonctions, les structures de contrôle sont évalués comme des expressions.

```
// Ce if-else sera alors une expression
// évaluée comme égale à 2
if 3 < 4 {
    1
} else if 4 < 5 {
    2
} else {
    3
}
```

Il y a aussi des boucles :

```
// Ce loop sera alors une expression
// évaluée comme égale à 4
loop {
    break 4;
}
```

Et du pattern matching (On notera qu'on peut aussi matcher une variable : ici _ est comme une variable dont la valeur n'a pas d'importance) :

```
// Ce match sera alors une expression
// évaluée comme égale à "5"
match 5 {
    0 => "0",
    1 => "1",
    2 => "2",
    3 => "3",
    4 => "4",
    5 => "5",
    _ => "Some numbers",
}
```

Il y a aussi des boucles plus habituelles :

```
while cond {
    body
}

// for i in (0..150) revient
// à for i in range(0, 150) en python
// ou à for(i = 0; i < 150; i++) en C

// On peut mettre n'importe quel
// itérateur à la place de (0..150)
for i in (0..150) {
    body
}
```

Les Structures & Enumérations

Définition

En Rust, comme en C, il est possible de définir des `struct`.

```
// On accède aux champs via
// variable.name, variable.age
struct Person {
    name: String,
    age: u8,
}

// On accède aux champs via
// variable.0, variable.1
struct Person(String, u8);
```

Pour les `enum`, c'est un peu pareil, sauf que l'on peut avoir plusieurs structures différentes :

```
enum Color {
    Red,
    Green,
    Blue,
    Black,
    // ...
}
```

```
enum ColorEncoding {
    RGBA(u8, u8, u8, u8),
    RGB(u8, u8, u8),
    GreyScale(u8),
    // ...
}

// On peut aussi avoir des champs nommés

enum ColorEncoding {
    RGBA {
        red: u8,
        green: u8,
        blue: u8,
        alpha: u8,
    },

    GreyScale {
        grey: u8,
    },
    // ...
}
```

Construction

Pour les `struct` :

```
struct Person {
    name: String,
    age: u8,
}

let john = Person {
    name: "John Doe".to_string(),
    age: 32,
};

// On accède aux champs via
// variable.0, variable.1
struct Person(String, u8);

let john = Person("John Doe".to_string(), 32);
```

Pour les `enum` :

```
enum Color {
    Red,
    Green,
    Blue,
    Black,
    // ...
}

let c = Color::Red;

enum ColorEncoding {
    RGBA {
        red: u8,
        green: u8,
        blue: u8,
        alpha: u8,
    },
    GreyScale {
        grey: u8,
    },
    // ...
}
```

```
let c = ColorEncoding::GreyScale {
    grey : 255,
};

// Très inmportant pour après :
use ColorEncoding::*;
let c = GreyScale {
    grey : 255,
};
```

Décomposition

Il est aussi possible de décomposer des `struct` avec un `let` ou un `match` :

Pour les `struct` :

```
struct Person {
    name: String,
    age: u8,
}
```

```rust
let john = Person {
    name: "John Doe".to_string(),
    age: 32,
};

let Person {
    name: john_name,
    age: john_age
} = john;
```

Ici, on décompose alors la variable `john` pour créer les variables `john_name` et `john_age`. Il faut bien noter qu'il ne sera plus possible d'utiliser `john` après. A moins que les valeurs extraites soient copiables (elles seront donc juste copiées, et `john` existera toujours (C'est une des conséquences du système d'ownership que l'on verra plus en détail plus tard)).

On peut faire pareil avec les pattern matching, et encore même mieux avec les enums :

```rust
enum ColorEncoding {
    RGBA(u8, u8, u8, u8),
    RGB(u8, u8, u8),
    GreyScale(u8),
    // ...
}

use ColorEncoding::*;
let c = RGB(255,0, 0);

match c {
    RGB(r, g, b) =>
        println!("r: {}, g: {}, b: {}", r, g, b),

    RGBA(r, g, b, a) =>
        println!("r: {}, g: {}, b: {}, a: {}", r, g, b, a),

    GreyScale(g) => println!("Grey : {}", g),
}
```

Un autre cas particulier qui est très intéressant, et d'utiliser (un peu à l'idée du pattern matching) la décomposition dans des stuctures de contrôles. On peut alors avoir quelque chose comme :

```
let c = RGB(255,0, 0);

if let RGB(r, g, b) = c {
    // r, g et b ne sont accessibles que dans cette scope.
    // si l'on sort de ce bloc de code, on ne peut plus y accéder
} else {
    ...
}

// Ou du même type :

while let RGB(r, g, b) = c {
    // Ici, ce n'est pas très intéressant...
    // Mais on verra que l'on peut trouver
    // des moyens assez bien pour s'en servir.
}
```

Méthodes

Il est possible, comme dans les langages objets (Rust n'est pas un langage objet), de définir des méthodes pour nos types personnalisés :

```
struct Person {
    name: String,
    age: u8,
}

impl Person {
    fn new(name: String, age: u8) -> Person {
        // Comme les params ont le même nom que les champs,
        // pas besoin d'écrire name: name
        Person {
            name,
            age,
        }
    }
}

let john = Person::new("John Doe".to_string(), 32);
```

Public / Privée / Use

Rust a un système de module assez simple. Ainsi, de base, chaque fichier est considéré comme un module. Ainsi, tous ce qui est dans un module peut-être utilisé sans problème, que ce soit public ou privé. Mais quand on importe des structures, il faut que ces structures soit publiques, et on ne pourra alors appeler que leurs fonctions publiques.

Tout est de base privé en Rust. Il faut alors utiliser le mot-clé `pub` afin de définir que quelque chose est public. Le constructeur de structure (`struct_name(expr1, expr2, ...)`) est d'ailleurs privé, d'où l'utilité de définir des méthodes `new` publiques.

```rust
// fichier person.rs
pub struct Person {
    name: String,
    age: u8,
}

impl Person {
    pub fn new(name: String, age: u8) -> Person {
        Person {
            name,
            age,
        }
    }
}

// autre fichier
use person::Person;
let john = Person::new("John Doe".to_string(), 32);
```

Structures intéressantes de la std

String

Une String est constitué d'un buffer contenant des charactères, et d'une capacité actuelle.

Si l'on rajoute plus de charactères qu'elle n'a de place, elle réallouera un buffer plus grand et recopiera tout son contenu.

```rust
fn main() {
    let mut s = String::new();
    s += "Hello";
    s += ", world!";
    println!("{}", s);

    let s = "Hello, world!".to_string();
    println!("{}", s);
}
```

Je vous conseille vraiment de prendre l'habitude d'aller regarder la doc : String.

Vector

Un Vecteur suit la même idée que les String, sauf qu'à la place, il peut contenir n'importe quel type.

```rust
fn main() {
    let mut v = Vec::new();
    v.push(4);
    v.push(5);
    println!("{:?}", v);
    // Output -> "[4, 5]"
}
```

Si on veut préciser le type d'un Vec :

```rust
fn main() {
    // C'est une macro pour créer
    // les vecteurs plus simplement.
    let mut v: Vec<u64> = vec![4, 5];
    println!("{:?}", v);
    // Output -> "[4, 5]"
}
```

Ici, la doc des Vecteurs.

Premier Concept Important : Ownership

Rust n'utilise pas de Garbage Collector, c'est aussi une des raisons pour lesquelles il est assez efficace ! En effet, c'est le compilateur qui gérera les allocations et les désallocations. En contre-partie, on devra respecter certaines règles afin que le compilateur sache toujours quand allouer et quand désallouer.

Une de ces règles est le principe d'Ownership (possession en Français). Il nous dit que toute variable est possédée par une fonction / scope (`{ }`).

Ainsi, à la fin d'une fonction, toute variable qui n'est pas renvoyée, qui n'est plus utilisée, sera alors désallouée. Aussi, chaque fois qu'on passera une variable en paramètre, si cette variable n'est pas copiable, on ne pourra plus y accéder après.

```rust
fn print_vecteur(v: Vec<u64>) {
    println!("{:?}", v);
}

fn main() {
    let mut v: Vec<u64> = Vec::new();
    v.push(4);
    v.push(5);
    print_vecteur(v);
    // Output -> "[4, 5]"
    // print_vecteur(v);
    // ne compilera pas car v ne nous appartient plus...
}
```

Comme dit au-dessus, certaints type sont copiables, ainsi, ils seront copié avant d'être passé en paramètre, on pourra toujours réutiliser la version originale plus tard.

```rust
fn print_int(n : u64) {
    println!("{}", n);
}

fn main() {
    let n = 8;

    // On copie le n comme variable locale pour print_int.
    print_int(n);
    // Output -> "8"

    // On peut donc le réutiliser ici.
    print_int(n);
    // Output -> "8"
}
```

Macro et Propriété

Les types copiables ont la propriété `Copy` (mais on verra les propriétés plus tard).

Ici, je vais juste vous expliquer comment on demande au compilateur de donner certaines propriétés à nos structures.

Si vous vous rappelez, je parlais de `Debug` pour l'affichage. Cela correspond à la propriété `Debug`.

On a aussi une propriété `Clone`, qui permet de cloner une structure. La grande différence entre `Copy` et `Clone`, pour simplifier, est que `Copy` doit toujours être en temps constant (le temps de copie ne doit pas varier selon ce que contient la structure), alors que le clone peut prendre autant de temps qu'il veut. Il faut donc voir `Copy` comme un cas particulier de `Clone`.

Ainsi, pour demander au compilateur de donner ces propriétés on utilise :

```
// On ne peut pas demander la copie,
// car copier une string ne se fait pas
// en temps constant, ça dépend de la
// taille de son buffer.

#[derive(Debug, Clone)]
struct Person {
    name: String,
    age: u8,
}

#[derive(Debug, Clone, Copy)]
enum ColorEncoding {
    RGBA(u8, u8, u8, u8),
    RGB(u8, u8, u8),
    GreyScale(u8),
    // ...
}
```

Partie 2

Références : Understanding Ownership & Validating References with Lifetimes

Retour sur l'ownership

La Pile et le Tas

La Pile et le Tas sont deux zones mémoires différentes qui servent deux objectifs bien différents : * La pile sert à l'allocation de zones de mémoires définies à la compilation. Chaque fonction agrandit la pile autant qu'elle veut, afin d'y stocker ses variables locales, mais tout ça doit être défini à la compilation. De plus, à la sortie de la fonction tout ce qui était stocké dans la pile sera désalloué. C'est donc un bon moyen de stockage temporaire, qui nécessite de connaître à l'avance la taille de chaque élément qu'elle devra contenir. * Le tas en revanche permet des allocations de mémoires dont on ne connait pas forcément la taille à la compilation mais qu'on connaîtra pendant le runtime. Ainsi, tout type contenant des attributs de taille variable (Sting, Vecteur, etc. . .) auront forcément une partie de leur structure dans le tas.

Les String

Pour les String, par exemple, on utilise une structure de contrôle dans la pile qui contient :

- la capacité maximale et la capacité actuelle (en bytes, pas en charactères)
- un pointeur (ou référence) vers le buffer de charactères

Ainsi, le buffer de charactères sera stocké dans le tas, car il a une taille variable.

Les Entiers

Un entier n'est lui constitué que d'une valeur sur 8, 16, 32, 64 ou 128 bits. De plus, cette taille sera définit précisément par le type de l'entier.

Imaginons que l'on veuille un type Entier, qui puisse avoir la taille que l'on souhaite :

Ce type sera-t'il stocké dans la pile ou dans le tas ?

```
// Si on reprend les nomenclatures de C
// On aura ça, à peu de chose près
enum Entier {
    Char(i8),
    Short(i16),
    Int(i32),
    LongInt(isize),
    LongLongInt(i64),
    VeryLongInt(i128),
}
```

Et bien ce type Entier sera stocké dans la pile car peu importe la variante dans laquelle on est, il aura toujours la même taille : 128 bits (+ 8 bits si nécéssaire pour stocker la variante actuelle).

En revanche, les calculs que l'on fera seront plus rapide que de toujours prendre directement un `i128`, car si l'on fait une addition, on pourra faire directement une addition entre deux `i32` ou deux `i64`, au lieu de toujours utiliser des `i128`.

Copy, Move et Clone

Dans la partie précédente, on avait très vite parlé de `Copy` et de `Clone`. On avait dit que `Copy` devait toujours être en temps constant, alors que ça n'avait pas trop d'importance pour `Clone`.

Cette différence entre `Copy` et `Clone` vient du fait que `Clone` doit copier toute la structure, même la partie dans le tas dont la taille peut changer (deep copy), alors que `Copy` ne fait qu'une copie des champs de la structure (shallow copy).

Ainsi, chaque structure voulant implémenter `Copy` doit vérifier, pour chacun de ses champs qu'il n'utilise que la pile (équivalent à implémenter `Copy`).

On a pas vraiment abordé le `Move` jusqu'à présent. Ce n'est pas une propriété contrairement à `Copy` et `Clone`. Mais une conséquence du fait de ne pas implémenter `Copy`. Je vous ai déjà dit que donner une structure non-copiable à une fonction est définitif, on ne peut plus l'utiliser après.

Cela est dû au fait que l'on a `Move` la structure. C'est à dire que l'on a copié ses champs un à un (comme `Copy`), mais que comme la structure n'avait pas la propriété `Copy`, les deux structures seront potentiellement en conflit (Deux String ayant le même buffer par exemple). Pour empêcher cela, Rust n'autorise donc plus l'accès à l'ancienne structure.

Notre première rencontre avec Move

Voilà un code que nous avions vu dans la première scéance.

```rust
fn main() {
    // On crée john
    let john = Person::default();

    // Ici, on move john car
    // john n'est pas copiable
    let john2 = john;

    // Donc john n'est plus accessible
    // Ca ne compilera donc pas
    // println!("{:?}", john);
}
```

Arrays

Ainsi, un array dont la taille est connue à la compilation et qui se trouve donc dans la pile devrait être copiable.

```
fn main() {
    // Crée un tableau de 200 000 cases.
    // a est ici de type : [i32, 200_000]
    // la taille de a est alors de 800 ko
    let a = [0; 200_000];

    // Si jamais on copie a dans b, on a alors besoin
    // de doubler la place dans la pile
    // On atteint alors 1.6 Mo
    //let b = a;

    // ne pas print le tableau
    // Ca sert juste à vérifier que
    // le compilo ne crie pas si on accède
    // à notre ancien tableau
    // preuve qu'il implémente Copy
    //println!("{:?}", a);
}
```

J'ai trouvé que la taille maximum de la pile pour Windows (gcc ou msvc) ne peut dépasser 1Mo, ainsi, si l'on copie l'array, on fait un stack overflow.

Sur Linux, il faudra probablement utiliser un array de 1 600 000 éléments.

Utilité de l'ownership

Il permet d'éviter les conflits entre plusieurs structures. Si l'on commence à utiliser plusieurs threads, il nous permettra donc d'éviter les data races. Il permet aussi de respecter les règles de borrowing (que l'on verra juste après). Et comme présenté la première fois, permet une gestion totale de la mémoire.

Les Références et le Borrowing

En plus de l'ownership, Rust utilise d'autres moyens afin de protéger les utilisateurs des data races et des segmentation faults tout en permettant de facilement faire du multi-threading très optimisé !

Rust introduit alors le Borrowing (emprunt) qui permet d'utiliser des références.

Références

Passer une structure par référence permet de ne pas effectuer de `Move` et donc de ne pas donner notre structure si celle-ci n'est pas copiable. En revanche, si cette dernière est copiable, alors on donnera une copie à la fonction, et la structure que l'on garde ne pourra donc pas être modifiée par la fonction qui n'aura alors qu'une copie locale. Pour la modifier, il faudra alors donner à la fonction une référence mutable.

De plus, si notre structure est énorme (Comme l'array qui faisait un stack overflow), il sera sûrement préférable de ne passer qu'un pointeur vers cet array, sans devoir tout recopier sur la pile.

Règles à respecter

Ainsi, pour avoir une référence sur une structure, il faut la borrow (l'emprunter). C'est à dire qu'on donne temporairement notre structure. On ne pourra alors plus la modifier, mais on pourra toujours y accéder de manière immutable.

Si on essaye de modifier une valeur borrowed :

```rust
fn main() {
    let mut a = 10;
    let b = &a;

    // a est toujours accessible
    println!("{}", a);

    // Ne compilera pas
    // car a est borrowed
    a = 12;
    println!("{}", b);
}
```

Si on attend avant de la modifier :

```
fn main() {
    let mut a = 10;
    let b = &a;

    // a est toujours accessible
    println!("{}", a);

    println!("{}", b);
    // b n'est plus utilisé après
    // a n'est donc plus borrowed

    // Compile sans problème
    a = 12;
}
```

De plus, pôur éviter les data races, on définit 2 règles :

Il peut y avoir soit :

- Un seul borrow mutable (dans ce cas là, la variable de base ne peut même plus lire son contenu le temps du borrow)
- Une multitude de borrow seulement immutables (comme vu précédemment, on peut accéder à la variable originale, mais on ne peut pas la modifier)

On ne peut pas utiliser a :

```
fn main() {
    let mut a = 10;
    let b = &mut a;

    // Cannot use a as it is mutably borrowed
    let c = a;

    println!("{}", b);
}
```

On ne peut pas borrowed une structure déjà mutably borrowed :

```
fn main() {
    let mut a = 10;
    let b = &mut a;
    // cannot borrow `a` as mutable
    // more than once at a time
    let c = &mut a;
    println!("{}", b);
}
```

On ne peut pas mutably borrowed une structure déjà borrowed :

```
fn main() {
    let mut a = 10;
    let b = &a;
    let c = b;
    let d = c;

    // cannot borrow `a` as mutable
    // because it is also borrowed as immutable
    let e = &mut a;

    println!("{}", b);
    println!("{}", c);
    println!("{}", d);
}
```

Le Lifetime

Le `Lifetime` permet d'éviter les dangling pointers (pointeur pointant vers une structure que n'existe plus).

Ainsi, quand on borrow une variable, la référence ne pourra être utilisée que tant que la variable originale existera.

Par exemple :

```
fn main() {
    let reference;
    {
        // Ici, a n'existe que dans
        // dans cette scope
        let a = 0;
        reference = &a;
    }
    // a n'existe plus ici

    // on aura alors l'erreur :
    // borrowed value does
    // not live long enough

    println!("{}", reference);
}
```

Ainsi, si l'on veut créer une fonction qui renvoie une structure dépendante d'une référence, alors il faudra préciser que la structure renvoyée à le même `Lifetime` que la référence en paramètre.

Par exemple, si l'on prend une slice de String, et qu'on veut renvoyer la première partie du slice avant un certain pattern :

```
// parfois, le Lifetime est évident
// ici, on voit bien que le Lifetime du retour
// dépend de référence
fn before_dot(reference: &str) -> &str {
    reference.split('.').next().unwrap()
}

//en revanche, ici, on ne sait pas vraiment
//de quel Lifetime on dépend: celui de
// référence, ou celui de pattern ?
fn before_pattern<'a>(reference: &'a str, pattern: &str) -> &'a str {
    reference.split(pattern).next().unwrap()
}

fn main() {
    let hello;
    {
        let s = String::from("Hello. World!");
        hello = before_pattern(&s, ".");
    }

    // On a alors la même erreur qu'avant
    println!("{}", hello);
}
```

On peut aussi avoir un `Lifetime` dans une structure, alors, le `Lifetime` fait partie du type de la structure :

```
struct Person<'a> {
    name: &'a str,
    age: u8,
}

fn main() {
    // plus besoin de mettre des
    // to_string() partout !
    let john = Person {
        name: "John Doe",
        age: 32,
    };
}
```

Si on commence à définir des méthodes pour une structure avec `Lifetime` :

```rust
impl<'a> Person<'a> {
    // Ici, le lifetime est obvious
    // pas besoin de le spécifié
    pub fn new(name: &str, age: u8) -> Person {
        Person { name, age }
    }

    // Si on veut permettre à with_name de changer le lifetime
    // de notre person, il faut en créer une autre
    // Rust nous dit ici que le Lifetime est obvious
    pub fn with_name<'b>(self, name: &'b str) -> Person<'b> {
        Person {
            name,
            age: self.age,
        }
    }

    // Ici, on s'assure que le lifetime de
    // notre nouveau nom sera le même que celui
    // de l'ancien
    pub fn set_name(&mut self, name: &'a str) {
        self.name = name;
    }
}

fn main() {
    // plus besoin de mettre des
    // to_string() partout !
    let mut john = Person::new("John Doe", 32);

    {
        let joe = "joe".to_string();
        // ainsi, si le lifetime est différent,
        // on ne peut pas changer notre nom
        john = john.with_name(&joe);

        // Si on commente cette ligne, on a eu erreur
        // car joe ne vit pas assez longtemps
        john = john.with_name("John Doe");
    }

    println!("{:?}", john);
}
```

Pour la fonction `set_name`, je ne pense pas qu'il soit possible de modifier le `Lifetime` de la `Person` que l'on manipule. Car, quand on prend une `&'a str`, le `'a` est déjà défini car il correspond au `'a` de `Person<'a>`. Cela oblige la slice à avoir le même `Lifetime` que notre structure.

On pourrait, comme pour `with_name`, prendre un autre `Lifetime`, mais je ne vois pas comment changer le type de la `Person`, alors que l'on a qu'une référence mutable sur cette `Person`.