# TP1: Références et mémoire

Florent Becker

## 1 Durée de vie et responsabilité des valeurs: l'IUT

### 1A Un type pour les étudiant·es

On se donne un type `Student` suivant pour représenter des étudiant·es d'un IUT.

```
#[derive(Debug, Clone)]
struct Student = {
       nom: String,
       prenom: String,
}
```

La mention `#[derive(Debug)]` indique au compilateur Rust de prévoir pour nous un affichage de débuggage pour le type `Student`, accessible ainsi:

```
let s = Student {nom: String::from("Lapin"),
                 prenom: String::from("Jeannot")
                };
println!("Affichage moche d'un Student: {:?}", &s);
```

La mention `#[derive(Clone)]` quant à elle, demande la création d'une méthode `clone()` permettant d'obtenir une copie d'un `Student`.

**Question 1**

Dans un nouveau projet, créer un étudiant et le cloner. Modifier le prénom du clone puis afficher les deux étudiants.

On représente les filières de l'IUT par le type somme suivant:

```
enum Filiere { Chimie, GEA, GMP, GTE, Info, QLIO}
```

Un élément d'un tel type peut être représenté par un entier donnant le numéro du constructeur utilisé.

**Question 2**

Ajouter un champ `filiere: Filiere` au type `Student`; qu'observez-vous à propos du clonage et de l'affichage de débug?

**Question 3**

Ajouter au type `Filiere` une méthode `fn appreciation(self)` qui affiche votre appréciation sur la filière[1].

[1] tous les départements sont égaux dans nos cœurs, mais certains sont plus égaux que d'autres

**Question 4**

Créer une filière et appeler la fonction `appreciation_filiere` sur cette filière une fois, puis deux fois. Que se passe-t-il?

En Rust, pour éviter les fuites mémoires, chaque valeur est sous la responsabilité d'un morceau de code, son *owner* qui est chargé de sa destruction. Quand un objet est créé, son responsable est le bloc de code qui l'a créé. Quand on passe une valeur `x` en argument à une fonction `f`, c'est `f` qui devient responsable de `x`. Celui-ci n'est donc plus accessible dans la suite de la fonction appelante, puisqu'il aura été désalloué par `f`.

Dans le cas de `Filiere`, dont la représentation en mémoire tient en un entier sur la pile, ces considération sont un peu oiseuses. On peut ajouter une mention `#[derive(Copy)]` au-dessus de la déclaration de ce type pour assurer à Rust qu'il est possible de copier une `Filiere` simplement en copiant telle quelle sa représentation sur la pile.

**Question 5**

Ajouter cette mention `#[derive(Debug, Clone, Copy)]` au type `Filiere`. Vérifier qu'il est désormais possible d'appeler plusieurs fois `appreciation_filiere` sur la même valeur.

Comme indiqué par l'attribut `Copy`, il est possible de copier implicitement les éléments de `Filiere`. Ainsi, le code suivant:

```
let f : Filiere = Filiere::Info;
affiche_filiere(f);
// f n'est plus disponible ici
```

est transformé par le compilateur en l'équivalent du code suivant:

```
let f : Filiere = Filiere::Info;
{
  let copie_de_f = f.clone();
  affiche_filiere(copie_de_f);
}
// f est toujours disponible ici.
```

**Question 6**

Que se passe-t-il si on essaye d'ajouter l'annotation `#[derive(Copy)]` sur le type `Student`?

Le type `String` n'est pas copiable. En effet, puisqu'une chaîne peut grandir, son contenu est situé dans le *tas* et il est susceptible d'être déplacé (réalloué). Comme en Python, ou comme les `StringBuilders` et `StringBuffers` en Java, une `String` est à peu de chose prêt constituée de:

- dans le tas, d'une zone de mémoire où sont stockés les caractères de la chaîne.
- sur la pile:
  - d'un entier représentant sa taille, et
  - de l'adresse du début de son contenu sur le tas;

**Question 7**

Représenter par un schéma l'état mémoire d'un programme après la création d'un élément de type `Student`.

Si on copiait la partie *pile* d'une chaine de caractère `s0`, on obtiendrait une nouvelle chaine `s1`, avec un nouveau pointeur vers le contenu de `s0`.

**Question 8**

Représenter l'état mémoire ainsi obtenu. Que se passe-t-il si `s0` est désallouée? Si `s1` est modifiée?

Comme Rust garantit les accès mémoire ainsi que l'absence de *data races* en présence d'accès concurrents, on ne peut pas se permettre les copies d'une String. Ainsi, la méthode `s.clone()` du type `String` réalise une copie *en profondeur* de l'objet `s`.

**Question 9**

Vérifier ce qui se passe si on clone une valeur de type `String` puis qu'on la modifie (ou si on modifie le clone).

Il peut sembler contraignant que les fonctions prennent la responsabilité de leurs arguments. L'exercice 2 vous présente l'utilisation de références qui permet d'accéder à des valeurs sans en prendre la responsablité. Pour motiver ce choix, dans l'exercice 3 nous allons trier le bon grain de l'ivraie en sélectionnant les étudiant·es qui vont obtenir leur diplôme.

## 1B Fonctions et références

**Question 10**

Écrire une fonction `fn affiche_etu(e: Student)` qui affiche un·e étudiant·e dans la console. Vérifier qu'elle prend bien la responsabilité (ou *ownership*) de son argument.

Cette fonction n'a pas de raison de modifier son argument, ni de le désallouer, ni de transférer sa responsabilité à un autre objet. Au lieu de prendre l'argument `e` *par valeur*, et d'en assumer la responsabilité, on peut écrire une version de la fonction qui prend l'argument `e` par référence en lecture seule. Pour celà, on indique le type d'argument `&Student` plutôt que `Student`.

**Question 11**

Modifier la fonction `affiche_etu` en conséquence

**Question 12**

Transformer la fonction `affiche_etu` en une méthode `affiche` du type `Student`. Pour prendre l'argument `self` par référence, on l'écrit `&self` dans l'en-tête de la fonction

**Question 13**

Vérifier que les appels `jlapin.affiche()` et `Student::affiche(&jlapin)` sont possibles (ils sont équivalents).

Quand on appelle une méthode avec la notation `s.affiche()`, Rust fait un référencement / déréférencement automatique, c'est à dire qu'il remplace cet appel par l'un des suivants:

- `Student::affiche(s)`
- `Student::affiche(&s)`
- `Student::affiche(&mut s)`
- `Student::affiche(*s)`

Cette règle ne s'applique que pour l'argument à gauche de l'appel de méthode. Si ce n'était pas le cas, il faudrait définir des règles complexe de priorité entre `f(&x, y)` et `f(x, &y)`.

**Question 14**

On veut ajouter une méthode `fn japanifie(??self)` au type `Student` qui ajoute la particule "さん" après son nom de famille. Quel type de référence faut-il utiliser pour ce faire?

## 1C La diplômation

**Question 15**

Ajouter un champ `nb_competences_validees: u8` à la structure `Student`.

**Question 16**

Créer un vecteur d'une dizaine d'étudiant·es qui représente la promotion de l'IUT.

Pour créer un vecteur non vide, on peut le créer vide et le remplir:

```
let v = Vec::new();
v.push(valeur0);
v.push(valeur1);
v.push(valeur2);
```

ou l'écrire directement

```
let v = vec![valeur0, valeur1]
```

Quand on a insère des valeurs dans un vecteur `v`, la méthode `v.push()` prend la responsabilité de ces valeurs. Par la suite c'est `v` qui est responsable de ces valeurs: elles seront désallouées quand le vecteur le sera.

On veut écrire une fonction `jury` qui prend en argument un vecteur de `Students` et renvoie ceux qui ont leur 6 compétences validées.

Considérons le code suivant:

```
1     fn main() {
2       // Création de joeLapin, sophieLajirHaff et billCanard
3       let tous_etudiants = vec![joeLapin, sophieLajirHaff, billCanard];
4       // Joe et Sophie valident leur compétences, pas Bill
5       let etu_ok = jury(tous_etudiants);
6       for e in &etu_ok {
7           affiche_etu(e);
8       }
9     }
```

Lors de la création des 3 étudiants, c'est la fonction `main` qui est chargée de les désallouers. À la ligne 3, avec la création du vecteur `tous_etudiants`, cette responsabilité est subsumée par celle de désallouer le vecteur; on dit que les valeurs sont *déplacées* dans le vecteur. À la ligne 5, avec l'appel à `jury`, la responsabilité du vecteur et donc de ses éléments passe à la fonction `jury`. Celle-ci va placer les étudiant·es méritant·es dans sa valeur de retour, qui deviendra le vecteur `etu_ok` et désallouer le vecteur et Bill qui y est resté[2]. La responsabilité de désallouer Joe et Sophie revient ainsi de nouveau à `main` quand la fonction `jury` renvoie sa valeur de retour. Ainsi, ceux-ci sont bien accessibles pour l'affichage des lignes 6 à 8.

**Question 17**

Écrire la fonction `fn jury(vecteur_etudiants: Vec<Etudiants>) -> Vec<Etudiants>`.

Que se passe-t-il si on essaie de passer son argument par référence plutôt que par valeur (avec la responsabilité des valeurs)?

**Question 18**

Il y a en fait un nombre variable de compétences à valider suivant la filière, comment en tenir compte?

## 1D Implémentation de traits prédéfinis

### 1DA Affichage utilisateur: le trait `Display`

Pour pouvoir afficher dans une instance du type `Student` dans une application réelle (avec `println!` ou `format!`), il suffit que ce type implémente le *trait* `Display`.

Un trait est l'équivalent en Rust d'une iterface Java. Il s'agit d'un ensemble de méthodes qu'un type peut implémenter.

**Question 19**

D'après la documentation https://doc.rust-lang.org/std/fmt/trait.Display.html, quelles sont les méthodes à implémenter pour implémenter le trait `Display` pour `Student`?

En plus de ce trait, la bibliothèque standard définit le type `fmt::Formatter` des objets chargés d'afficher des représentations chiadées de valeurs Rust. Il n'est pas nécessaire de connaitre le détail de ce type, il suffit de savoir que l'on peut écrire dans un formateur `f` par: `write!(f, "chaine avec des {}", x)`, avec un fonctionnement analogue à `println!`.

**Question 20**

Implémenter le trait `Display` pour `Student` et vérifier que l'on peut utiliser la syntaxe `println!("{}", s)` pour afficher une valeur de type `Student`

[2] *dura lex, sed lex.* Dans l'implémentation, le déplacement des étudiants méritants en dehors du vecteur se fait par une copie. L'exemplaire qui est réputé avoir été déplacé est rendu inaccessible par les règles de Rust, ainsi le code se comporte *comme si* ils avaient été déplacés. À la fin de la fonction, on désalloue le vecteur et toutes ses valeurs, à savoir les étudiants défaillants et les copies obsolètes des étudiant·es méritant·es –qui ont été sauvés ailleurs.

**Question 21**

Vérifier que `format!("... {} ...", s)` permet d'intégrer la représentation d'un student au sein d'une chaine de caractères. Quel est le type de la valeur obtenue?

# 2 Modifications concurrentes

## 2A Sur des vecteurs

Jeannot Lapin, étudiant de BUT1 info, est chargé de supprimer toutes les valeurs "a" d'une liste en Java.

Il propose le squelette de code suivant[3]

```
import java.util.ArrayList;
import java.util.List;

class Deleteur {
    public static void main(String args[]) {
        List<String> list = new ArrayList<>();
  list.add("a");
        list.add("b");

        for (String s: list) {
            if (s.equals("a")) {
    list.remove(s);
      }
        }

        System.out.println(list);
    }
}
```

**Question 22**

Que se passe-t-il quand on lance ce code?

**Question 23**

Sa camarade Sophie lui propose de tester son code sur les listes suivantes.

- `["a", "a"]`
- `["a", "b", "c", "d"]`
- `["d", "c", "b", "a"]`
- `["d", "c", "b", "a"]`

Que se passe-t-il? Pouvez-vous l'expliquer? Qu'en concluez-vous?

Leur camarade Camille Castor propose le code suivant en python:

```
l = ["b", "c", "a", "a", "d"]
```

[3]En début de BUT1, Jeannot n'a tpas encore entendu parler de classes, getters, setters, tests, intégration continue, conteneures... *Ignorance is bliss.*

```python
for s in l:
    if s == "a":
        l.remove(s)

print(l)
```

**Question 24**

À nouveau, expliquer ce qui se passe.

Forte de ce savoir, Valériane Vigogne se propose d'utiliser le code suivant pour éliminer d'un dictionnaire les paires clé-valeur avec une valeur paire:

```python
d = {"a": 1, "b": 2, "c": 3, "d": 4}
for (k, v) in d.items():
    if v % 2 == 0:
        del d[k]

print(d)
```

**Question 25**

Qu'obtient-elle, pourquoi?

Dans les deux cas, le comportement est peu intuitif. En Java, le fait de supprimer un élément de la liste *peut* invalider l'itérateur sur la liste si celui-ci n'est pas vers la fin de la liste; les détails dépendent de l'implémentation de l'itérateur. En Python, les itérateurs de liste sont robustes aux délétions[4], mais pas les itérateurs sur les dictionnaires. En règle générale, il faut éviter de modifier une collection sur laquelle on itère.

En Rust, cette *bonne pratique* devient une *règle*. Chaque référence, mutable ou non est associée à un *domaine de validité* (ou lifetime) correspondant à la partie du code où elle est valable. Le compilateur impose la règle suivante:

> Dans le domaine de validité d'une référence mutable, aucune autre référence au même objet ne peut être valable.

On propose la traduction suivante en Rust des exemples précédents sur les vecteurs de chaines de caractères:

```rust
fn main() {
  let mut v = vec!["a", "b", "c", "d"];

  for (i, s) in v.iter().enumerate() {
    if *s == "a" {
      v.remove(i);
    }
  }

  println!("{v:?}");
}
```

[4] et aux *insertions*, en prenant garde d'éviter les boucles infinies.

**Question 26**

Que se passe-t-il quand on compile cet exemple? Expliquer le message d'erreur en donnant le domaine de validité des références mises en jeu.

**Question 27**

Donner un code correct en Rust pour supprimer toutes les chaînes égales à `\"a\"` dans un vecteur de chaînes de caractères.