Une **structure**, est un type de données personnalisé qui vous permet de nommer et de rassembler plusieurs valeurs associées qui forment un groupe cohérent. Si vous êtes familier avec un langage orienté objet, une **structure** est en quelque sorte l'ensemble des attributs d'un objet.

*Dans cette étude, nous comparerons les **tuples** avec les **structures**, nous montrerons comment utiliser les structures et nous aborderons la définition des méthodes et des fonctions associées pour spécifier le comportement associé aux données d'une structure. Les **structures** et les **énumérations** (que nous traiterons ultérieurement) sont les fondements de la création de nouveaux types au sein de votre programme effectuées par **Rust**.*

## Définir et instancier des structures

Les **structures** sont similaires aux **tuples**, que nous avons abordé lors de la première étude. Comme pour les **tuples**, les éléments d'une **structure** peuvent être de différents types. Par contre, contrairement aux **tuples**, nous devons nommer chaque élément des données afin de clarifier le rôle de chaque valeur. Grâce à ces noms, les **structures** sont plus flexibles que les **tuples** : nous n'avons pas à utiliser l'ordre des données pour spécifier ou accéder aux valeurs d'une instance.

*Pour définir une **structure**, nous utilisons le mot-clé **struct** suivi du nom unique de la structure. Ce nom est un nouveau type et à ce titre il est judicieux d'avoir la première an majuscule afin de différencier un type d'une variable. Le nom de la structure devrait décrire l'utilisation des éléments des données regroupés. Ensuite, entre des accolades, nous définissons le nom et le type de chaque élément des données, que nous appelons un champ (ou attribut dans le cadre d'une philosophie objet).*

**Création et utilisation de structures**

```
struct Complexe {
    reel: f64,
    imaginaire: f64
}

struct Eleve {
    nom: String,
    prenom: String,
    notes: Vec<f64>
}

fn main() {
    let i = Complexe {
        imaginaire: 1.0,
        reel: 0.
    };
    let mut elle = Eleve {
        nom: "BORÉALE".to_string(),
        prenom: "Aurore".to_string(),
        notes: [5., 10., 15.0].to_vec()
    };
    elle.notes.push(18.5);
    println!("(reel={}, imaginaire={})", i.reel, i.imaginaire);
    println!("(Prénom={}, Nom={}, Notes={:?})", elle.prenom, elle.nom, elle.notes);
}
```

**Résultat**

```
(reel=0, imaginaire=1)
(Prénom=Aurore, Nom=BORÉALE, Notes=[5.0, 10.0, 15.0, 18.5])
```

*Pour obtenir une valeur spécifique d'un champ particulier depuis une structure, nous utilisons l'opérateur point « **.** ». Lorsque nous désirons la valeur de chaque élément de la structure, il suffit de spécifier le champ concerné à la suite du nom de la variable et du point séparateur. C'est ce que nous faisons lors de la gestion des affichages. Si l'instance est **mutable**, nous pourrions changer une valeur en utilisant la notation avec le point et assigner une valeur à ce champ en particulier, ce que nous faisons lors du rajout d'une nouvelle **note** à l'élève « **elle** ».*

*À noter que l'instance tout entière doit être **mutable** ; **Rust** ne nous permet pas de marquer seulement certains champs comme **mutables**. Comme pour toute expression, nous pouvons construire une nouvelle instance de la structure comme dernière expression du corps d'une fonction pour retourner implicitement cette nouvelle instance.*

## Utiliser le raccourci d'initialisation des champs lorsque les variables possèdent le même nom

Il est logique de nommer les paramètres de fonction ou des variables annexes avec le même nom que les champs de la **structure**, mais devoir répéter les noms de variables et des champs respectifs est un peu pénible. Si la **structure** avait plus de champs, répéter chaque nom serait encore plus fatigant. Heureusement, il existe un raccourci pratique !

*Puisque les noms des variables et les noms de champs de la **structure** sont exactement les mêmes, nous pouvons utiliser la syntaxe de **raccourci d'initialisation des champs** pour réécrire la structure **elle** de sorte qu'elle se comporte exactement de la même façon sans avoir à répéter **nom** et **prénom**, comme le montre le programme ci-dessous :*

**La structure Eleve**

```
struct Eleve {
    nom: String,
    prenom: String,
```

```
    notes: Vec<f64>
}

fn main() {
    let nom = "BORÉALE".to_string();
    let prenom = "Aurore".to_string();
    let mut elle = Eleve {
        nom,
        prenom,
        notes: Vec::new()
    };
    elle.notes = [15.5, 8.5, 12.5, 17.].to_vec();
    println!("(Prénom={}, Nom={}, Notes={:?})", elle.prenom, elle.nom, elle.notes);
}
```

**Résultat**

```
(Prénom=Aurore, Nom=BORÉALE, Notes=[15.5, 8.5, 12.5, 17.0])
```

*Ici, nous créons une nouvelle instance de la structure **Eleve**, qui possède deux champs nommés **nom** et **prenom**. Nous voulons donner au champ **nom** la valeur de la variable **nom**, ainsi que le champ **prenom** avec la valeur de la variable **prenom**. Comme les champs et les variables possèdent les mêmes intitulés, nous avons besoin uniquement d'écrire **nom** plutôt que **nom: nom** ainsi que **prenom** plutôt que **prenom : prenom**.*

## Créer des objets à partir d'autres objets avec la syntaxe de mise à jour de structure

Il est souvent utile de créer une nouvelle instance de **structure** qui utilise la plupart des valeurs d'une ancienne **instance** tout en changeant certaines parties. Nous utiliserons pour cela la syntaxe de **mise à jour de structure**.

*En utilisant la syntaxe de **mise à jour de structure**, nous pouvons produire beaucoup moins de code que la copie intégrale de chacun des champs respectifs, comme l'exemple suivant. La syntaxe **« .. »** indique que les autres champs auxquels nous ne donnons pas explicitement de valeur devraient avoir la même valeur que dans l'instance copiée.*

**Mise à jour de structure**

```
struct Date {
    jour: u8,
    mois: u8,
    annee: i16
}

struct Complexe {
    reel: f64,
    imaginaire: f64
}

fn main() {
    let aujourdui = Date {jour:16, mois:4, annee:2021};
    let demain = Date { jour: aujourdui.jour+1, ..aujourdui };
    let i = Complexe {imaginaire: 1., reel:0.};
    let mut x = Complexe{ ..i };
    x.reel = 1.;
    afficheDate(aujourdui);
    afficheDate(demain);
    afficheComplexe(i);
    afficheComplexe(x);
}

fn afficheDate(date: Date) {
    println!("{}/{}/{}", date.jour, date.mois, date.annee);
}

fn afficheComplexe(c:Complexe) {
    println!("(réel={}, imaginaire={})", c.reel, c.imaginaire);
}
```

**Résultat**

```
16/4/2021
17/4/2021
(réel=0, imaginaire=1)
(réel=1, imaginaire=1)
```

## Structures tuples sans champ nommé pour créer de nouveaux types

Nous pouvons aussi définir des **structures** qui ressemblent à des **tuples**, appelées **structures tuples**. La signification d'une **structure tuple** est donnée par son nom. En revanche, ses champs ne sont pas nommés ; nous ne précisons que leurs types. Les **structures tuples** servent lorsque nous désirons créer un nouveau type, sans que nous ayons à nommer chaque champ comme dans une structure classique, ce qui serait trop verbeux ou redondant.

*Comme pour toute **structure**, la définition d'une **structure tuple** commence par le mot-clé **struct** avec le nom de la **structure** suivis des types des champs du **tuple,** sans spécification du nom de chaque champ puisqu'il s'agit d'un **tuple**. Par exemple, voici une définition et une utilisation de deux **structures tuples** nommées **Couleur** et **Point** :*

**Structures tuples**

```
fn main() {
    struct Couleur(i32, i32, i32);
    struct Point(i32, i32, i32);

    let noir = Couleur(0, 0, 0);
    let origine = Point(0, 0, 0);
}
```

*Notez que les valeurs **noir** et **origine** sont de types différents parce que ce sont des instances de **structures tuples** différentes. Chaque **structure** que nous définissons constitue son propre type, même si les champs au sein de la structure possède exactement les mêmes types.*
*Par exemple, une fonction qui prend un paramètre de type **Couleur** ne peut pas prendre un argument de type **Point** à la place, bien que ces deux types soient tous les deux constitués de trois valeurs **i32**.*
*Mis à part cela, les instances de **stuctures tuples** se comportent comme des **tuples** : nous ne pouvons les déstructurer en éléments individuels, nous pouvons utiliser un **.** suivi de l'indice pour accéder individuellement à une valeur, et ainsi de suite.*

## Ajouter de fonctionnalités utiles avec les traits prédéfinis

L'utilisation de nouveaux types que nous élaborons va vite sembler inconfortables. En effet, la structure **Rectangle** proposée ci-dessous ne peut être ni copiée, ni clonée. Vous ne pouvez pas afficher son contenu avec la syntaxe classique de la macro ***println(« {:?} », rectangle) ;*** et enfin elle ne supporte pas les opérateurs relationnels « **==** » et «**!=** ».

*Chacune de ces quatre opérations porte un nom prédéfini : **Copy**, **Clone**, **Debug** et **PartialEq**. Ce sont des **traits**. Nous verrons dans une autre étude comment créer nos propres **traits** pour vos **structures**. Mais sauf si vous avez un besoin très spécial, vous avez un besoin très spécial, vous avez tout intérêt à réutiliser les **traits standards**.*

***Rust** peut les appliquer pour vous avec une précision toute mécanique. Il suffit d'ajouter une ligne d'attributs #[derive] juste devant la structure qui doit implémenter ces **traits standards**.*

**Affichage automatique, permissions du clonage et de la copie associée à la structure Rectangle**

```
#[derive(Copy, Clone, Debug, PartialEq)]
struct Rectangle {
    largeur: u32,
    hauteur: u32,
}

fn surface(rectangle: &Rectangle) -> u32 {
    rectangle.largeur * rectangle.hauteur
}

fn main() {
    let rect1 = Rectangle {
        largeur: 30,
        hauteur: 50
    };
    let mut rect2 = rect1.clone();
    rect2.largeur = 40;
    let rect3 = rect2;

    println!("rect1={:?}", rect1);
    println!("rect2={:?}", rect2);
    println!("rect3={:?}", rect3);
    println!("rect1!=rect2 ? {}", rect1!=rect2);
    println!("rect2==rect3 ? {}", rect2==rect3);
    println!("surface = {}", surface(&rect1));
}
```

**Résultat**

```
rect1=Rectangle { largeur: 30, hauteur: 50 }
rect2=Rectangle { largeur: 40, hauteur: 50 }
rect3=Rectangle { largeur: 40, hauteur: 50 }
rect1!=rect2 ? true
rect2==rect3 ? true
surface = 1500
```

*Chacun des traits standard peut être implémenté automatiquement , à condition que chacun des champs implémente ce trait. Nous avons par exemple de demander le trait **PartialEq** pour la structure puisque les deux champs sont de type **u32**, type qui implémente déjà **PartialEq**.*

*Nous pourrions aussi faire dériver le trait **PartialOrd** afin de pouvoir utiliser les opérateurs de comparaison « **<,>, ≤** et **=>** ». Nous ne l'avons pas fait, parce que comparer deux rectangles sur une surface n'a vraiment pas de sens (il n'y a pas d'ordre particulier entre des rectangles).*

***C'est pour éviter d'associer inutilement certains traits que Rust a choisi d'obliger le programmeur à utiliser l'attribut #[derive] au lieu de dériver automatiquement tous les traits possibles.***

## La syntaxe des méthodes – programmation objet

L'évocation de méthode fait bien entendu référence à la notion de programmation objet. Elles sont beaucoup plus adaptées au fonctionnement même de la structure. Les méthodes ont en effet un accès direct aux différents attributs. Pas besoin d'avoir une multitude de paramètres, puisque bien souvent les calculs sont fait à partir des attributs de la structure elle-même. Par ailleurs, la notion de surface peut être associée à bien d'autres structures. La surface d'un rectangle n'a rien à voir avec la surface d'un triangle. Il est donc plus judicieux que cette fonction **surface()** devienne plutôt une méthode spécifique de cette structure particulière.

*La syntaxe des méthodes sont similaires aux fonctions : nous les déclarons avec le mot-clé **fn** suivi de leur nom, elles peuvent avoir des paramètres et une valeur de retour, et elles contiennent du code qui est exécuté quand nous les appelons depuis un autre endroit.*

*Cependant, les **méthodes** diffèrent des **fonctions** parce qu'elles sont définies dans le contexte d'une **structure** et que leur premier paramètre est toujours **self**, un mot-clé qui **représente l'instance de la structure** sur laquelle nous appelons la méthode.*

**Méthodes associées à la structure Rectangle**

```
#[derive(Debug)]
struct Rectangle {
    largeur: u32,
    hauteur: u32,
}

impl Rectangle {
    fn surface(&self) -> u32 {
        self.largeur * self.hauteur
    }
    fn perimetre(&self) -> u32 {
        (self.largeur + self.hauteur) * 2
    }
}

fn main() {
    let rectangle = Rectangle {
        largeur: 30,
        hauteur: 50
    };
    println!("Notre rectangle = {:?}", rectangle);
    println!("Surface du rectangle = {}", rectangle.surface());
    println!("Périmètre du rectangle = {}", rectangle.perimetre());
}
```

**Résultat**

```
Notre rectangle = Rectangle { largeur: 30, hauteur: 50 }
Surface du rectangle = 1500
Périmètre du rectangle = 160
```

*Pour définir la **méthode** dans le contexte de **Rectangle**, nous démarrons un bloc **impl** (implémentation). Puis nous déplaçons la fonction **surface** entre les accolades du **impl** et nous remplaçons le premier paramètre (et dans notre cas, le seul) par **self** dans la signature et dans tout le corps.*

*L'appel des méthodes se place après l'instance : nous ajoutons un point suivi du nom de la méthode et des parenthèses contenant les arguments s'il y en a.*

*Nous avons choisi **&self** ici pour la même raison que nous avions utilisé **&Rectangle** quand il s'agissait d'une fonction ; nous ne voulons pas en prendre possession, et nous voulons seulement lire les données de la structure, pas les modifier. Si nous voulions que la méthode modifie l'instance sur laquelle on l'appelle, nous utiliserions **&mut self** comme premier paramètre.*

## Où est l'opérateur -> que nous utilisions dans le langage C++ ?

En C++, deux opérateurs différents sont utilisés pour appeler les méthodes : nous utilisons « **.** » si nous appelons une méthode directement sur l'objet et « **->** » si nous appelons la méthode sur un pointeur vers l'objet et qu'il faut d'abord déréférencer le pointeur. En d'autres termes, si objet est un pointeur, **objet->methode()** est similaire à **(*objet).methode()**.

***Rust** n'a pas d'équivalent à l'opérateur **->** ; à la place, **Rust** a une fonctionnalité appelée **référencement et déréférencement automatiques**. L'appel de méthodes est l'un des rares endroits de **Rust** où nous retrouvons ce comportement.*

*Voilà comment cela fonctionne : quand nous appelons une méthode avec **objet.methode()**, **Rust** ajoute automatiquement le **&**, **&mut** ou ***** pour que **objet** corresponde à la signature de la méthode. Autrement dit, les deux dernières lignes de la fonction **main()** sont identiques :*

**Référencement et déréférencement automatique**

```
fn main() {
    #[derive(Debug,Copy,Clone)]
    struct Point {
        x: f64,
        y: f64,
    }

    impl Point {
        fn distance(&self, autre: &Point) -> f64 {
            let x_carre = f64::powi(autre.x - self.x, 2);
            let y_carre = f64::powi(autre.y - self.y, 2);
            f64::sqrt(x_carre + y_carre)
        }
    }
    let p1 = Point { x: 0.0, y: 0.0 };
    let p2 = Point { x: 5.0, y: 6.5 };
    let mut p3 = p2.clone();
    p3.x -= 2.5;
    let p4 = p3;
    println!("{:?}, {:?}, {:?}, {:?}, distance={:.2}", p1, p2, p3, p4, p1.distance(&p3));
    p1.distance(&p2);
    (&p1).distance(&p2);
}
```

**Résultat**

```
Point { x: 0.0, y: 0.0 }, Point { x: 5.0, y: 6.5 }, Point { x: 2.5, y: 6.5 }, Point { x: 2.5, y: 6.5 }, distance=6.96
```

*Sur ces deux dernières lignes, la première semble bien plus propre. Ce comportement du **(dé)référencement automatique** fonctionne parce que les méthodes ont une cible claire : le type de **self**. Compte tenu du nom de la méthode et de l'instance sur laquelle elle s'applique, **Rust** peut déterminer de manière irréfutable si la méthode lit (**&self**), modifie (**&mut self**) ou consomme (**self**) l'instance. Le fait que **Rust** rend implicite **l'emprunt** pour les instances sur lesquelles nous appelons les méthodes améliore significativement l'ergonomie de la **possession**.*

## Les méthodes avec davantage de paramètres

Dans le chapitre précédent, nous avons utilisé la méthode ***distance()*** de la structure **Point** qui est capable de calculer la distance entre deux points, dont le deuxième point est passé en argument. Comme pour toutes les fonctions, il est bien sûr possibles d'avoir plusieurs paramètres pour les méthodes autre que **&self** ou **&mut self**.

*Je vous propose de continuer sur ce sujet en revenant sur la structure **Rectangle** en implémentant une troisième méthode sur la structure **Rectangle**. Cette fois-ci, nous voulons qu'une instance de **Rectangle** prenne une autre instance de **Rectangle** au travers de la méthode **contient()** qui retourne **true** si le second **Rectangle** peut se dessiner intégralement à l'intérieur du premier ; sinon, renvoie **false**.*

*Nous désirons donc définir une nouvelle méthode qui doit se trouver dans le bloc **impl** de **Rectangle**. Le nom de la méthode est **contient()** et prend une **référence immuable** vers un autre **Rectangle** en paramètre.*

*Nous pouvons déterminer le type du paramètre en regardant le code qui appelle la méthode : **rect1.contient(&rect2)** prend en argument **&rect2**, une **référence immuable** vers **rect2**, une instance de **Rectangle**. Cela est logique puisque nous voulons uniquement lire **rect2** (plutôt que de le **modifier**, ce qui aurait nécessité une **référence mutable**) et nous souhaitons que **main()** garde **possession** de **rect2** pour que nous puissions le réutiliser par la suite.*

*La valeur de retour de **contient()** est un booléen et l'implémentation de la méthode vérifie si la largeur et la hauteur de **self** sont respectivement plus grandes que la largeur et la hauteur de l'autre **Rectangle**.*

**Méthodes avec plusieurs paramètres**

```
struct Rectangle {
    largeur: u32,
    hauteur: u32,
}

impl Rectangle {
    fn surface(&self) -> u32 {
        self.largeur * self.hauteur
    }
    fn perimetre(&self) -> u32 {
        (self.largeur + self.hauteur) * 2
    }
    fn contient(&self, autre: &Rectangle) -> bool {
      self.largeur > autre.largeur && self.hauteur > autre.hauteur
    }
}
```

```
fn main() {
    let rect1 = Rectangle {
        largeur: 30,
        hauteur: 50
    };
    let rect2 = Rectangle {
        largeur: 10,
        hauteur: 40
    };
    let rect3 = Rectangle {
        largeur: 60,
        hauteur: 45
    };

    println!("rect1 peut-il contenir rect2 ? {}", rect1.contient(&rect2));
    println!("rect1 peut-il contenir rect3 ? {}", rect1.contient(&rect3));
}
```

**Résultat**

```
rect1 peut-il contenir rect2 ? true
rect1 peut-il contenir rect3 ? false
```

*Lorsque nous exécutons ce code avec la fonction main(), nous obtenons l'affichage attendu. Les **méthodes** peuvent prendre plusieurs paramètres que nous pouvons ajouter à la signature après le paramètre **self**, et ces paramètres ont exactement le même comportement que les paramètres des **fonctions**.*

## Les fonctions associées

Une autre aptitude utile des blocs **impl** est que nous pouvons définir des fonctions dans des blocs **impl** qui ne prennent pas **self** en paramètre. Cela s'appelle des **fonctions associées** parce qu'elles sont associées à la structure. Cela reste des **fonctions**, pas des **méthodes**, parce qu'elles ne s'appliquent pas à une instance de structure. Nous avons déjà utilisé la fonction associée ***from()*** de la classe **String** → **String::*from()***. Cela correspond aux méthodes statiques dans le langage C++.

*Les fonctions associées sont souvent utilisées comme constructeurs qui retourne une nouvelle instance de la structure. Par exemple, nous pourrions écrire une fonction associée qui prend une unique dimension en paramètre utile à la fois pour la largeur et pour la hauteur, ce qui rend plus aisé la création d'un carré plutôt que d'avoir à spécifier la même valeur deux fois :*

*Pour appeler cette fonction associée, nous utilisons l'opérateur de séparation « **::** » avec le nom de la structure. Cette fonction est cloisonnée dans l'espace de noms de la structure : le séparateur **::** s'utilise aussi bien pour les fonctions associées que pour les espaces de noms créés par des modules (voir dans une autre étude).*

**Méthodes statiques**

```
#[derive(Debug)]
struct Rectangle {
    largeur: u32,
    hauteur: u32,
}

impl Rectangle {
    fn carre(cote: u32) -> Rectangle {
        Rectangle { largeur:cote, hauteur:cote }
    }
    fn surface(&self) -> u32 {
        self.largeur * self.hauteur
    }
    fn perimetre(&self) -> u32 {
        (self.largeur + self.hauteur) * 2
    }
    fn contient(&self, autre: &Rectangle) -> bool {
      self.largeur > autre.largeur && self.hauteur > autre.hauteur
    }
}

fn main() {
    let carre = Rectangle::carre(30);
    println!("Notre carré = {:?}", carre);
    println!("Surface du carré = {}", carre.surface());
    println!("Périmètre du carré = {}", carre.perimetre());
}
```

**Résultat**

```
Notre carré = Rectangle { largeur: 30, hauteur: 30 }
Surface du carré = 900
Périmètre du carré = 120
```

*Par convention, les constructeurs **Rust** portent toujours le nom **new()** que nous connaissons déjà grâce à **Vec::new()**.*

*Il est possible pour chaque **structure** d'avoir plusieurs blocs **impl**. Ici, nous n'avons aucune raison de séparer ces méthodes dans plusieurs blocs **impl**, mais c'est une syntaxe valide. Nous verrons un exemple de l'utilité d'avoir plusieurs blocs **impl** lorsque nous aborderons les types génériques et les traits.*

## Construction et inférence de type

Une autre abréviation intéressante est que chaque bloc **impl** définit un paramètre de type spécial **Self** (notez la majuscule) qui correspond au type (à la structure) pour lequel nous créons les méthodes :

**Construction et inférence de type**

```
struct Eleve {
    nom: String,
    prenom: String,
    notes: Vec<f64>
}

impl Eleve {
    fn new(nom: &str, prenom: &str) -> Self {
        Eleve {
            nom: nom.to_string(),
            prenom: prenom.to_string(),
            notes: Vec::new()
        }
    }
    fn identite(&self) -> String {
        format!("{} {}", self.prenom, self.nom)
    }
    fn moyenne(&self) -> f64 {
        match self.notes.len() {
          0 => 0.,
          _ => {
             let mut somme = self.notes[0];
             for note in &self.notes[1..] { somme += note; }
             somme/self.notes.len() as f64
          }
        }
    }
}

fn main() {
    let mut elle = Eleve::new("BORÉALE", "Aurore");
    elle.notes = [15.5, 8.5, 12.5, 17.5].to_vec();
    println!("(Identité={}, Notes={:?}, Moyenne={})", elle.identite(), elle.notes, elle.moyenne());
}
```

**Résultat**

```
(Identité=Aurore BORÉALE, Notes=[15.5, 8.5, 12.5, 17.5], Moyenne=13.5)
```

*Vous remarquez que dans le corps de la méthode, nous n'avons pas besoin de répéter le paramètre de type dans l'expression du constructeur. Nous profitons du mécanisme d'**inférence de type** de **Rust** : puisqu'un seul type est acceptable pour la valeur renvoyée, **Eleve**, **Rust** déduit le paramètre de lui-même.*

## Structures contenant des références

Voyons maintenant comment Rust surveille l'utilisation des références lorsqu'elles sont implantées dans une structure de données. Les contraintes de sécurité de Rust concernant les références ne disparaissent pas comme par magie parce que nous avons caché la référence dans une structure. Elles doivent donc s'appliquer systématiquement à toutes structures qui comporte les références.

*Je vous propose d'expérimenter cette problématique en reprenant la structure **Eleve** que nous venons d'exploiter. Cette fois-ci, au lieu de prendre des chaînes de type **String**, nous allons plutôt prendre des chaînes statique de type **&str**, puisque normalement une fois que l'identité est donnée, elle n'évolue plus au cours du temps. Ce choix paraît plus logique.*

**Structure avec des références de chaîne**

```
struct Eleve {
    nom: &str,
    prenom: &str,
    notes: Vec<f64>
}

impl Eleve {
    fn new(nom: &str, prenom: &str) -> Self {
        Eleve {
            nom,
            prenom,
            notes: Vec::new()
```

```
        }
    }
    fn identite(&self) -> String {
        format!("{} {}", self.prenom, self.nom)
    }
    fn moyenne(&self) -> f64 {
        match self.notes.len() {
            0 => 0.,
            _ => {
                let mut somme = self.notes[0];
                for note in &self.notes[1..] { somme += note; }
                somme/self.notes.len() as f64
            }
        }
    }
}

fn main() {
    let mut elle = Eleve::new("BORÉALE", "Aurore");
    let lui = Eleve {
        nom: "TÉRIEUR",
        prenom: "Alain",
        notes: vec![12., 15., 18., 7.]
    };
    elle.notes = [15.5, 8.5, 12.5, 17.5].to_vec();
    println!("(Identité={}, Notes={:?}, Moyenne={})", elle.identite(), elle.notes, elle.moyenne());
    println!("(Identité={}, Notes={:?}, Moyenne={})", lui.identite(), lui.notes, lui.moyenne());
}
```

**Résultat**

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:2:10
  |
2 |     nom: &str,
  |          ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 | struct Eleve<'a> {
2 |     nom: &'a str,
  |
```

*Lorsque nous exécutons le programme ci-dessus, nous remarquons tout de suite une erreur de compilation qui stipule que nous devons proposer une durée de vie pour les références internes à la structure avec une proposition de solution.*

***Il reste un détail que nous n'avons pas encore abordé, c'est que toutes les références dispose d'une durée de vie dans Rust, qui est la portée pour laquelle cette référence est en vigueur. La plupart du temps, les durées de vies sont implicites et sont déduit automatiquement, comme lors de la détermination des types implicites.***

*Nous devons renseigner le type lorsque plusieurs types sont possibles. De la même manière, nous devons renseigner les durées de vie lorsque les durées de vies des références peuvent être déduites de différentes manières. Rust nécessite que nous renseignons ces relations en utilisant des paramètres de durée de vie génériques pour s'assurer que les références utilisées au moment de la compilation restent bien en vigueur.*

***Le concept de la durée de vie est quelque chose de radicalement différent de ce que l'on retrouve dans les outils des autres langages de programmation, à un tel point que la durée de vie est la fonctionnalité qui distingue Rust des autres.***

## Structures avec paramètres de durée de vie

L'annotation des durées de vies a une syntaxe un peu inhabituelle. Nous utilisons la syntaxe des génériques « <> » avec le nom des paramètres de durées de vies qui commence par une apostrophe « ' » et sont habituellement en minuscule et très court. Vous pouvez choisir ce nom, mais la plupart des personnes utilisent le nom « **<'a>** » (si un seul paramètre). Nous plaçons le paramètre de type après le **&** d'une référence, en utilisant un **espace** pour **séparer l'annotation du type de la référence**.

*Une annotation de durée de vie toute seule n'a pas vraiment de sens, car les annotations sont faites pour indiquer à **Rust** quels paramètres de durée de vie génériques de plusieurs références sont liés aux autres. L'objectif est de préciser que les durées de vie des méthodes qui utilisent les références doivent être les mêmes que la structure elle-même.*

*Les instances de ces structures fixent la durée de vie à partir du moment où elles sont déclarée, jusqu'à ce qu'elles soient automatiquement supprimées lorsqu'elles sortent du bloc d'instructions où elles ont été déclarées.*

**Structure avec paramètre de durée de vie**

```
struct Eleve<'a> {
    nom: &'a str,
    prenom: &'a str,
    notes: Vec<f64>
}
```

```
impl<'a> Eleve<'a> {
    fn new(nom: &'a str, prenom: &'a str) -> Self {
        Eleve {
            nom,
            prenom,
            notes: Vec::new()
        }
    }
    fn identite(&self) -> String {
        format!("{} {}", self.prenom, self.nom)
    }
    fn moyenne(&self) -> f64 {
        match self.notes.len() {
            0 => 0.,
            _ => {
                let mut somme = self.notes[0];
                for note in &self.notes[1..] { somme += note; }
                somme/self.notes.len() as f64
            }
        }
    }
}

fn main() {
    let mut elle = Eleve::new("BORÉALE", "Aurore");
    let lui = Eleve {
        nom: "TÉRIEUR",
        prenom: "Alain",
        notes: vec![12., 15., 18., 7.]
    };
    elle.notes = [15.5, 8.5, 12.5, 17.5].to_vec();
    println!("(Identité={}, Notes={:?}, Moyenne={})", elle.identite(), elle.notes, elle.moyenne());
    println!("(Identité={}, Notes={:?}, Moyenne={})", lui.identite(), lui.notes, lui.moyenne());
}
```

**Résultat**

```
(Identité=Aurore BORÉALE, Notes=[15.5, 8.5, 12.5, 17.5], Moyenne=13.5)
(Identité=Alain TÉRIEUR, Notes=[12.0, 15.0, 18.0, 7.0], Moyenne=13)
```

*Cette structure possède deux champs avec des références de chaîne, **nom** et **prenom**. Comme pour les types de données génériques, nous déclarons le nom du paramètre de durée de vie générique entre des chevrons après le nom de la structure pour que nous puissions utiliser le paramètre de durée de vie dans le corps de la définition de la structure. Cette annotation signifie qu'une instance de **Eleve** ne **peut pas vivre plus longtemps** que les références qu'elle stocke dans ses champs **nom** et **prenom**.*

*La fonction **main()** crée ici deux instances de la structure **Eleve** qui stocke des références vers des constantes littérales chaîne de caractères. Les données de ces constantes littérales existent juste au moment de la création des deux élèves. De plus, les constantes littérales ne sortent pas de la portée avant que les instances de **Eleve** sortent elles-même de la portée, donc les références dans les instances de **Eleve** sont **toujours valides**.*