Structures, implémentations et traits

> *No Boilerplate →* ***Building a space station in Rust (Simple Rust patterns) [RUST-8]***

Structure

Un `struct` ou *structure* est un type de donnée personnalisé qui permet de rassembler plusieurs propriétés. (C'est un peu comme un "objet" dans la POO)

```
// Création d'une première structure "Chapitre"
struct Chapitre {
  titre: String,
  numero: u64,
  contenu: String
}

// Création d'une deuxième structure "Livre" qui contient entreautre, une liste de "Chapitre"
struct Livre {
  titre: String,
  auteur: String,
  contenu: Vec<Chapitre>
}

fn main() {
    // Création d'un chapitre et d'un livre
    let chapitre = Chapitre {
        titre: "Mon super petit chapitre".to_string(),
        numero: 1,
        contenu: "voilà, le chapitre est fini...".to_string()
    };
```

```
    let livre = Livre {
        titre: "Livre court".to_string(),
        auteur: "Anonyme".to_string(),
        contenu: vec![chapitre]
    };

    println!("Mon livre est '{}' et le premier chapitre est '{}'", livre.titre,
livre.contenu[0].titre);
}
```

Implémentation et trait

Un *trait* sert à définir un comportement partagé de manière abstraite par plusieurs types.

Une *implémentation* permet de grouper des méthodes et fonctions pour un type donné. Par exemple sur un *struct*, il va y avoir à la fois les propriétés du *struct* et les méthodes de son implémenation. La différence entre une fonction et une méthode c'est que:

- Une méthode prends comme argument elle même (`self`) et on l'utilise en finissant par `.nom_de_la_methode()`
- Une fonction n'a pas besoin d'accéder au contenu (donc pas de `self`), et on l'utilise en finissant par `::nom_de_la_fonction()`

Ainsi une implémentation peut être utilisée pour implémenter un *trait* sur un type.

```
// Le trait "Resumable" dit que tout type implémentant ce trait doit avoir la méthode
"resumer"
pub trait Resumable {
    // Le &self est important car sinon il devient propriétaire de lui même et on ne peut donc
plus l'utiliser
    fn resumer(&self) -> String;
    // On utilise &mut pour pouvoir faire en sorte que l'objet puisse modifier une instance de
lui même
    fn anonymiser(&mut self);

    // L'absence de self signifie que ceci est une fonction et non pas une méthode
    fn new(titre: &str, lieu: &str, auteur: &str, contenu: &str) -> ArticleDePresse;
}
```

```
// La structure ArticleDePresse contient 4 propriétés de type String
pub struct ArticleDePresse {
    pub titre: String,
    pub lieu: String,
    pub auteur: String,
    pub contenu: String,
}

// Implémente le trait Resumable pour le type ArticleDePresse
impl Resumable for ArticleDePresse {
    // Implémente la méthode "résumer"
    // Le &self est important car si on met juste 'self' il devient propriétaire de lui même et on ne peut donc plus l'utiliser
    fn resumer(&self) -> String {
        format!("{}, par {} ({})", self.titre, self.auteur, self.lieu)
    }

    // On utilise &mut pour pouvoir faire en sorte que l'objet puisse modifier une instance de lui même
    fn anonymiser(&mut self) {
        self.auteur = "Anonyme".to_string();
        self.lieu = "Inconnu".to_string();
    }

    // Ceci ne prends pas self, donc c'est une fonction et pas une méthode
    // Cette fonction va créer un élément de type ArticleDePresse à partir de valeurs données
    fn new(titre: &str, lieu: &str, auteur: &str, contenu: &str) -> ArticleDePresse {
        ArticleDePresse {
            titre: titre.to_string(),
            lieu: lieu.to_string(),
            auteur: auteur.to_string(),
            contenu: contenu.to_string()
        }
    }
}

// Crée ArticleDePresse dans la variable "article" et exécute la méthode "resumer"
fn main() {
    // On va utiliser la méthode anonymiser plus tard donc on met "mut" sur notre variable pour pouvoir la modifier
```

```
    // On utilise notre fonction new avec ::
    let mut article = ArticleDePresse::new("Mon titre", "Liège", "John Doe",
"ayayyayayayaya");

    // On utilise nos méthodes avec .
    // On modifie notre article pour modifier les valeurs lieu et auteur
    article.anonymiser();

    // Enfin on retourne le résumé de notre article
    println!("{}", article.resumer());
}
```

Mais il est aussi possible de faire la même chose, sans trait.

```
// La structure ArticleDePresse contient 4 propriétés de type String
pub struct ArticleDePresse {
    pub titre: String,
    pub lieu: String,
    pub auteur: String,
    pub contenu: String,
}

// Implémente le type ArticleDePresse
impl ArticleDePresse {
    fn resumer(&self) -> String {
        format!("{}, par {} ({})", self.titre, self.auteur, self.lieu)
    }

    fn anonymiser(&mut self) {
        self.auteur = "Anonyme".to_string();
        self.lieu = "Inconnu".to_string();
    }

    fn new(titre: &str, lieu: &str, auteur: &str, contenu: &str) -> ArticleDePresse {
        ArticleDePresse {
            titre: titre.to_string(),
            lieu: lieu.to_string(),
            auteur: auteur.to_string(),
```

```
            contenu: contenu.to_string()
        }
    }
}

// Crée ArticleDePresse dans la variable "article" et exécute la méthode "resumer"
fn main() {
    // On va utiliser la méthode anonymiser plus tard donc on met "mut" sur notre variable pour pouvoir la modifier
    let mut article = ArticleDePresse::new("Mon titre", "Liège", "John Doe", "ayayyayayayaya");

    article.anonymiser();
    println!("{}", article.resumer());
}
```

En savoir plus

- *Le language de programmation Rust* - Définir et instancier des structures
- *Le language de programmation Rust* - Définir des comportements partagés avec les traits
- *Rust By Example* - Structures
- *Rust By Example* - Traits
- *Rust By Example* - Implementation

Revision #1
Created 27 April 2023 06:12:11 by SnowCode
Updated 27 April 2023 06:12:49 by SnowCode