Mon projet Tamagotchi en Rust (POO – ligne de commande)

Pourquoi j'ai fait ce projet

J'ai voulu faire ce petit projet Tamagotchi pour apprendre concrètement la programmation orientée objet (POO) en Rust, mais sans partir sur quelque chose de trop compliqué.
Je voulais aussi que ce soit **en ligne de commande**, pour me concentrer uniquement sur la logique du code et pas sur une interface graphique.

Création de la struct `Tamagotchi`

J'ai commencé par créer une structure `Tamagotchi` qui représente mon petit personnage virtuel.
Il a 4 attributs principaux : un nom, une énergie, une faim et un niveau de bonheur.

```
#[derive(Debug)]
struct Tamagotchi {
    nom: String,
    energie: i32,
    faim: i32,
    bonheur: i32,
}
```

Chaque attribut est un entier compris entre 0 et 100. Au départ, tout est à 100.

Le constructeur : `new`

Ensuite, j'ai ajouté une méthode `new` dans un bloc `impl` pour créer un Tamagotchi avec un nom choisi par l'utilisateur.

```
fn new(nom: String) -> Self {
    Self {
        nom,
        energie: 100,
        faim: 100,
        bonheur: 100,
    }
}
```

J'ai utilisé `Self` pour renvoyer une nouvelle instance de ma structure, ce qui est l'équivalent d'un constructeur en POO classique.

🎮 Les actions du Tamagotchi

J'ai créé plusieurs méthodes qui modifient l'état du Tamagotchi :

- `manger()` : augmente la faim et un peu le bonheur
- `dormir()` : recharge l'énergie mais baisse un peu la faim
- `jouer()` : consomme un peu d'énergie et de faim mais rend heureux
- `sport()` : effort intense, plus de fatigue, plus de faim, mais encore plus de bonheur

Voici un exemple :

```
fn manger(&mut self) {
    self.faim = (self.faim + 10).min(100);
    self.bonheur = (self.bonheur + 5).min(100);
    println!("{} a mangé !", self.nom);
}
```

J'ai utilisé `.min(100)` et `.max(0)` pour garder les valeurs dans des bornes valides (de 0 à 100), ce qui évite les bugs.

⌨️ Interaction en ligne de commande

Le jeu se lance avec une boucle `loop` et propose à l'utilisateur de saisir une commande : `manger`, `dormir`, `jouer`, `sport` ou `quitter`.

```
let mut action = String::new();
io::stdin().read_line(&mut action).unwrap();
let action = action.trim();

match action {
    "manger" => tamagotchi.manger(),
    "quitter" => break,
    _ => println!("Commande inconnue."),
}
```

J'utilise `read_line()` pour récupérer l'entrée utilisateur, et `match` pour exécuter la bonne méthode.

🎨 L'ASCII art pour donner vie au Tamagotchi

J'ai aussi ajouté une méthode `afficher_ascii()` pour afficher un petit visage différent selon l'humeur du Tamagotchi :

```
fn afficher_ascii(&self) {
    if self.energie <= 10 || self.faim <= 10 {
        println!("  ( T_T ) ");
```

```
        } else if self.bonheur >= 80 {
            println!("  ( ^_^ ) ");
        } else if self.energie <= 30 {
            println!("  ( -_- ) ");
        } else {
            println!("  ( o_o ) ");
        }
    }
```

Ça rend le jeu plus vivant, même en ligne de commande.

☑ Ce que j'ai appris avec ce projet

Ce projet m'a vraiment aidé à :

- mieux comprendre les **structs** et les **impl** en Rust
- utiliser le mot-clé `self` pour accéder ou modifier les attributs d'un objet
- gérer des **entrées utilisateur** et une boucle interactive en CLI
- utiliser des méthodes comme `.min()` et `.max()` pour sécuriser les valeurs
- structurer mon code de façon propre, à la manière objet

Je suis parti d'un concept simple, et j'ai pu ajouter petit à petit de nouvelles fonctionnalités. Je pourrais même aller plus loin : détecter la mort du Tamagotchi, sauvegarder l'état, ou ajouter des effets visuels supplémentaires.

🎉 Conclusion

C'est un petit projet, mais il m'a permis d'apprendre plein de choses sur Rust tout en m'amusant un peu.
Je pense que c'est une super manière de progresser en POO, surtout dans un langage comme Rust qui oblige à bien réfléchir à la gestion de la mémoire et des états.

🤯 Quelques explications sur la syntaxe Rust utilisée

◇ `struct` et `impl`

En Rust, une `struct` est utilisée pour définir une structure de données (similaire à une classe sans comportement).
Le mot-clé `impl` est utilisé pour définir les **méthodes associées** à cette struct, comme les actions du Tamagotchi (`manger`, `dormir`, etc.).

◇ `Self` vs `self`

- `Self` (avec un grand S) représente le **type courant** dans une implémentation.
  → utilisé surtout dans le constructeur pour retourner une nouvelle instance :

```
fn new(nom: String) -> Self { ... }
```

- `self` (avec un petit s) fait référence à **l'instance courante** dans une méthode.
  → utilisé pour accéder ou modifier les attributs :

```
self.faim = (self.faim + 10).min(100);
```

◇ `&mut self`

Quand une méthode modifie l'objet, elle doit prendre `&mut self` comme paramètre.
Cela signifie : "je vais modifier l'objet sur lequel cette méthode est appelée".

◇ `min()` et `max()`

Ce sont des méthodes très pratiques pour limiter une valeur entre deux bornes.
Exemple :

```
self.bonheur = (self.bonheur + 10).min(100); // pas plus que 100
self.energie = (self.energie - 20).max(0);   // pas moins que 0
```

◇ `io::stdin().read_line()`

Permet de lire ce que l'utilisateur tape au clavier.

```
let mut action = String::new();
io::stdin().read_line(&mut action).unwrap();
let action = action.trim();
```

◇ `match`

Permet de choisir un comportement en fonction d'une valeur (équivalent à `switch` dans d'autres langages).

```
match action {
    "manger" => tamagotchi.manger(),
    _ => println!("Commande inconnue."),
}
```

◇ `loop` et `break`

Une boucle infinie qui ne s'arrête que si on utilise `break`, par exemple quand l'utilisateur tape `quitter`.

Ces éléments font partie des bases du langage Rust. En les pratiquant dans un projet simple comme ce Tamagotchi, on les retient beaucoup plus naturellement.