

Enums et gestion d'erreurs (Result et Option)

Rust exige toujours de gérer les cas où il y aurait une erreur d'une manière ou d'une autre. Toute commande qui pourrait donner une erreur rendra un type `Result<>` et toute commande qui pourrait ne donner rien retournera un type `Option<>`.

Il y a plusieurs manières de gérer ce genre de types :

- Avec `.unwrap()`, non-recommandé. Fait planter le programme quand il y a une erreur (pour les `Result`) ou quand il n'y a rien (dans les `Option`)
- Avec `.expect("message")`, fait également planter le programme, mais indique un message d'erreur personnalisé.
- Avec `.unwrap_or(valeur)` ou encore `.unwrap_or_else(|c| { valeur })`, dans le cas d'un `None` ou d'une erreur, il va retourner une valeur par défaut à la place
- Avec `match` ou `if let` pour faire des choses pour chaque cas ou pour définir une valeur par défaut
- Avec `?` dans une fonction et va donc créer une erreur au niveau du retour de la fonction en elle même

```
// Cette fonction peut retourner un booléen si tout se passe bien et peut planter avec un &str
fn beverage_checker(beverage: &str) -> Result<bool, &str> {
    []if beverage == "water" {
        [][]Ok(true)
    []} else if beverage == "coffee" {
        [][]Err("Wait, you actually drink that???)
    []} else {
        [][]Ok(true)
    []}
}

fn main() {
    []let mut status = beverage_checker("water").expect("Le développeur de ce programme n'aimais vraiment pas
cette boisson");
    []println!("Le status de 'water' est {}", status);
}
```

// Ici on va remplacer l'erreur par un false avec unwrap_or

```
status = beverage_checker("coffee").unwrap_or(false);  
println!("Le status de 'coffee' est {}", status);
```

// On peut aussi utiliser unwrap_or_else pour exécuter du code dans une closure en plus de retourner une autre valeur

```
status = beverage_checker("coffee").unwrap_or_else(|err| {  
    println!("Message: {}", err);  
    false  
});  
println!("Le status de 'coffee' est {}", status);
```

// Sinon on peut encore utiliser un match dans lequel on exécute du code

```
match beverage_checker("coffee") {  
    Ok(s) => println!("Le status de 'coffee' est {}", s),  
    Err(e) => println!("Voici le message du développeur : {}", e)  
}
```

// On peut aussi faire similaire à précédemment avec un if let

```
if let Ok(s) = beverage_checker("coffee") {  
    println!("Le status de 'coffee' est {}", s);  
} else {  
    println!("Le développeur n'aime pas le café");  
}
```

// Ou encore utiliser match comme on la fait avec unwrap_or ou unwrap_or_else

```
status = match beverage_checker("coffee") {  
    Ok(s) => s,  
    Err(e) => {  
        println!("Voici le message du développeur : {}", e);  
        false  
    }  
};  
println!("Le status de 'coffee' est {}", status);
```

// Cette ligne va planter le programme :

```
status = beverage_checker("coffee").expect("Le développeur de ce programme n'aimait vraiment pas cette  
boisson");  
println!("Le status de 'coffee' est {}", status);
```

```
□  
□// Cette ligne planterais le programme sans message d'erreur  
□status = beverage_checker("coffee").unwrap();  
□println!("Le status de 'coffee' est {}", status);  
}
```

Enum

Result et Option sont tous les deux des enums que voici :

```
pub enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}  
  
pub enum Option<T> {  
    None,  
    Some(T),  
}
```

Un enum permet d'énumérer plusieurs variantes. Voici un exemple d'enum personnalisé

```
enum IpAddrKind {  
    V4,  
    V6,  
}  
  
fn main() {  
    let four = IpAddrKind::V4;  
    let six = IpAddrKind::V6;  
}
```

Maintenant on peut aussi avoir des valeurs dans un enum

```
enum IpAddrKind {  
    V4(String),  
    V6(String),  
}
```

```
fn main() {
    let four = IpAddrKind::V4("127.0.0.1".to_string());
    let six = IpAddrKind::V6("::1".to_string());
}
```

Et quand on ne sait pas encore les types qui vont être compris dans notre enum on peut utiliser cette notation :

```
enum IpAddrKind<A, B> {
    V4(A),
    V6(B),
}

fn main() {
    let four: IpAddrKind<String, String> = IpAddrKind::V4("127.0.0.1".to_string());
    let six: IpAddrKind<String, String> = IpAddrKind::V6("::1".to_string());
}
```

On peut aussi gérer les enums avec des `match` ou encore avec des `if let`

```
enum IpAddrKind<A, B> {
    V4(A),
    V6(B),
}

fn main() {
    let four: IpAddrKind<&str, &str> = IpAddrKind::V4("127.0.0.1");
    let six: IpAddrKind<&str, &str> = IpAddrKind::V6("::1");

    // En utilisant match
    match four {
        IpAddrKind::V4(ip) => println!("ipv4: {}", ip),
        IpAddrKind::V6(ip) => println!("ipv6: {}", ip)
    }

    match six {
        IpAddrKind::V4(ip) => println!("ipv4: {}", ip),
        IpAddrKind::V6(ip) => println!("ipv6: {}", ip)
    }

    // En utilisant if let
```

```
if let IpAddrKind::V4(ip) = four {  
    println!("ipv4: {}", ip);  
}  
  
if let IpAddrKind::V6(ip) = six {  
    println!("ipv6: {}", ip);  
}  
}
```

Revision #1

Created 27 April 2023 06:10:32 by SnowCode

Updated 27 April 2023 06:12:49 by SnowCode