3_Result type and errors_fr

May 2, 2023

1 Gérer les erreurs potentielles avec `Result`

[ ]:
```
use std::io;

fn main() {
    println!("Devinez le nombre!");

    println!("Entrez votre tentative.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("Vous avez deviné: {}", guess);
}
main();
```

> Note: Exécuter la cellule ci-dessus va bloquer le kernel, redémarrer avec le menu Kernel, restart

`read_line` place ce que l'utilisateur a tapé en entré dans la chaîne de caractère qui lui est passé en paramètre et retourne une valeur de type `io::Result`. Rust a dans sa librairie standard un certain nombre de types `Result`, en premier lieu le générique `Result`, mais aussi d'autres plus spécialisés pour des modules spécifiques comme `io::Result`.

Les types `Result` sont tous des énumérations avec deux variantes, `Ok` et `Err`. La variante `Ok` indique que l'opération s'est passé correctement. Cette variante peut contenir la valeur de retour "normale". A l'inverse, la variante `Err` signifie que l'opération a échoué. Cette variante contient une valeur descriptive de l'erreur.

Une instance de `Result` possède une méthode `expect`. Si l'instance de la variante `Err`, la méthode va faire crasher le programme (en utilisant `panic`) en utilisant le message passé en argument.

Si l'instance est de la variante `Ok`, `expect` va retourner la valeur contenue par la variante pour qu'elle puisse être utilisée. Dans ce cas précis, il s'agit du nombre d'octets lus.

[ ]:
```
use std::fs::File;
```

```
fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("There was a problem opening the file: {:?}", error)
        },
    };
}
main();
```

2 Gérer les erreurs avec `Result`

2.1 Match de types d'erreurs

[ ]:
```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Tried to create file but there was a problem:␣
 ↪{:?}", e),
            },
            other_error => panic!("There was a problem opening the file: {:?}",␣
 ↪other_error),
        },
    };
}
main();
```

Le type de valeur produite par `File::open` dans sa variante `Err` est `io::Error`, qui est une structure fournie par la librairie standard. Cette structure a une méthode `kind` qui renvoie une instance de `io::ErrorKind`, une énumération représentant les différents types d'erreur qui peuvent arriver lors d'une opérations I/O. Comme `File::create` peut aussi échouer, nous avons besoin d'une seconde expression `match`.

2.2 Propagation d'erreurs

Lorsque vous écrivez une fonction qui utilise d'autres fonctions qui peuvent échouer avec `Result`, au lieu de gérer dans votre fonction ces erreurs potentielles, elles peuvent être propagées au code

qui appele votre fonction pour que celui-ci décide.

[ ]:
```rust
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) { // lit le contenu du fichier dans une␣
 ↪String
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

Dans cet exemple, notre fonction retourne une valeur de type `Result< T, E >` où `T` correspond au type en cas de succès (`String` ici) et `E` le type pour la variante `Err` (ici `io::Error`). Si tout se passe bien, le code qui appele cette fonction va recevoir une variante `Result::Ok` avec une `String` à l'intérieur. A l'inverse, en cas d'erreur, la fonction retournera une variante `Result::Err` contenant l'erreur `io::Error` à l'origine de l'échec.

2.3 Propagation raccourcie avec l'opérateur ?

L'opérateur `?` placé après une valeur `Result` est défini pour fonctionner d'une manière similaire aux expression `match` dans l'exemple précédent. La différence se trouve dans le fait que l'opérateur `?` peut en plus convertir implicitement l'erreur trouvée vers le type d'erreur attendu en retour de la fonction en utilisant le trait `From`. Aussi longtemps que les erreurs qui peuvent survenir dans la fonction peuvent être converties en erreur attendues par le retour de la fonction, la conversion est automatique.

[ ]:
```rust
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
```

```
}
```

Note : L'opérateur `?` ne peut être utilisé que dans des fonctions qui retourne un type `Result`.