Methods of Enums

We studied methods in structs in the last chapter, let's explore methods in enums in this lesson.

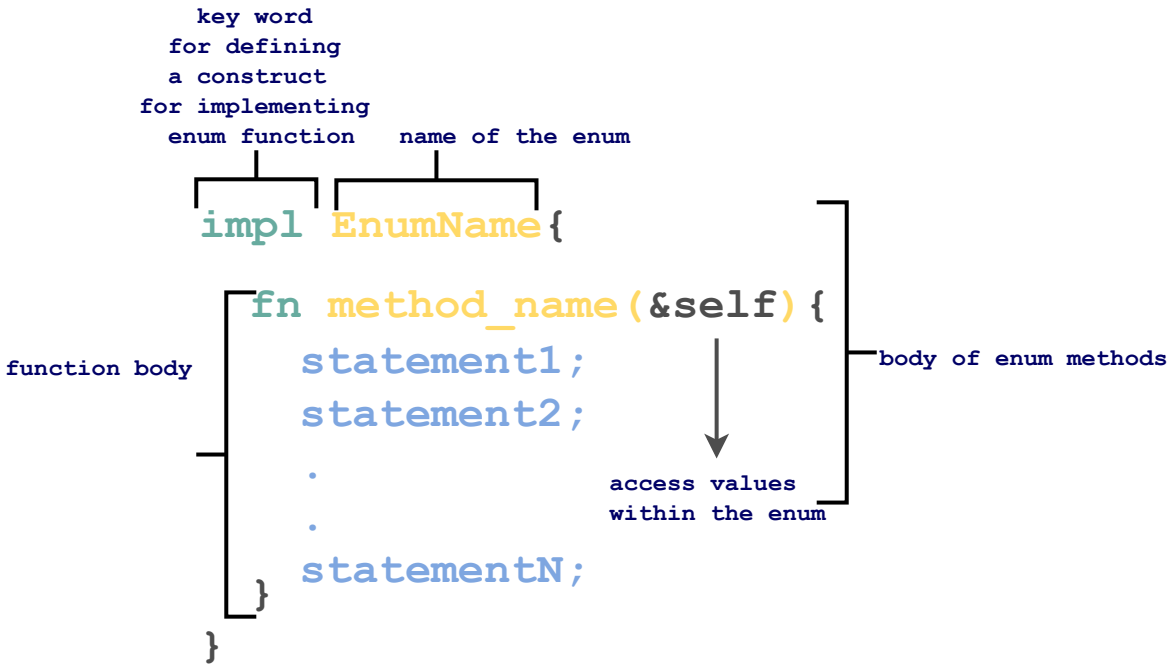
We'll cover the following

What Are Methods?

Just like structs, methods are functions specific to enums.

Syntax

To define methods of `enum` write the functions within the `impl` followed by the `enum` name and then the functions within the `impl` block.



imply construct for defining enum methods

> Move `enum` related logic in the `impl` construct

Example

The example below declares an `enum`, named `TrafficSignal` and defines an enum method `is_stop` within the `impl` construct:

```
#![allow(dead_code)]
#[derive(Debug)]
// declare an enum
enum TrafficSignal{
  Red, Green, Yellow
}
//implement a Traffic Signal methods
impl TrafficSignal{
  // if the signal is red then return
   fn is_stop(&self)->bool{
     match self{
       TrafficSignal::Red=>return true,
       _=>return false
     }
   }
}
fn main(){
  // define an enum instance
  let action = TrafficSignal::Red;
  //print the value of action
  println!("What is the signal value? - {:?}", action);
  //invoke the enum method 'is_stop' and print the value
  println!("Do we have to stop at signal? - {}", action.is_stop());
}
```

Explanation

- **`main` Function**

  The body of the `main` function is defined from **line 17 to line 24**.

  - On **line 19**, `enum` is initialized and the value is saved in the variable `action`.
  - On **line 21**, the value of `action` is printed.
  - On **line 23**, `action` invokes the function `is_stop` within the `impl` construct.

- On **line 2**, `#![allow(dead_code)]` is declared which helps to remove warning if

any variable is left uninitialized.

- On **line 4**, `#[derive(Debug)]` is declared which helps to print the values of the `enum`.
- `enum`
  - On **line 3**, `enum` `TrafficSignal` is defined
  - On **line 5**, **variants** of enum `Red` , `Yellow` , and `Green` are defined.
- `impl` construct

  The `impl` construct is defined from **line 8 to line 16**.
  - A method `is_stop` is defined within the construct.
    - The function takes in a parameter `self` passed by reference and returns a **boolean** value.
    - On **line 11**, the `match` construct takes the parameter `&self`
      - On **line 12**, it checks if the function is invoked with the value `Red` , i.e., the value of `TrafficSignal` is `Red`. If yes, returns `true` and `false` otherwise.

Quiz

Test your understanding of methods of `enum` in Rust.

Quick Quiz on Enum Methods!

What is the output of the following code?

```
#![allow(dead_code)]
#[derive(Debug)]
enum TrafficSignal {
  Red, Green, Yellow
}
impl TrafficSignal{
   fn is_stop(&self)->bool{
     match self{
       &TrafficSignal::Red=>return true,
```

```
        _=>return false
      }

    }
}
fn main(){
  let action = TrafficSignal::Yellow;
  println!("What is the signal value? - {:?}", action);
  println!("Do we have to stop at signal? - {}", action.is_stop
());
}
```

Retake Quiz

Now that you have learned about methods in enums, let's move on to learn how match flow operators work with enum types.