

# Enumerações

Pedro Horchulhack

# Enumerações

- São valores que podem de forma nominal representar um tipo em Rust
  - Podemos usar quando queremos representar um conjunto de itens únicos, nomeados e constantes
- Podemos declarar uma enumeração com a palavra chave **enum**

```
enum Veiculo {  
    Carro,  
    Onibus  
}  
  
fn main() {  
    let mut v: Veiculo = Veiculo::Carro;  
    v = Veiculo::Onibus;  
}
```

# Enumerações

- Enumerações **unitárias** são representadas por números inteiros (`usize`) onde, quando os valores não possuem um valor associado, inicializam com o valor 0 (zero).

```
enum Veiculo {  
    Carro, // 0  
    Onibus // 1  
}  
  
fn main() {  
    let mut v: Veiculo = Veiculo::Carro;  
    v = Veiculo::Onibus;  
}
```

# Enumerações

- Também podemos dizer qual os valores os nossos tipos enumerados armazenarão

```
enum Veiculo {  
    Carro = 10,  
    Onibus = 42  
}  
  
fn main() {  
    let mut v: Veiculo = Veiculo::Carro;  
    v = Veiculo::Onibus;  
}
```

# Enumerações

- Também podemos associar construtores aos tipos enumerados, ou seja, inicializar variáveis junto ao tipo enumerado.

```
enum Veiculo {  
    Carro (String, i32),  
    Onibus {placa: String, num_portas: i32}  
}  
  
fn main() {  
    let mut v: Veiculo = Veiculo::Carro("MWB7H21".to_string(), 4);  
    v = Veiculo::Onibus{ placa: "JVJ4G10".to_string(), num_portas: 6 };  
}
```

# Enumerações

- Como já vimos, existem **três** formas de criarmos tipos enumerados. Ou seja, temos três variantes:
  - **Tuplas()**
  - **Structs{ }**
  - **Unidades**

```
enum SemCampos {  
    Tupla(),  
    Struct{},  
    Unidade  
}
```

# Enumerações

- Também podemos colocar valores enumerados dentro de vetores

```
enum Animal {  
    Cachorro,  
    Cobra(bool),  
    Passaro{raca: String},  
}  
  
fn main() {  
    let mut animais: Vec<Animal> = vec![];  
  
    animais.push(Animal::Cobra(true)); // Trocou de pele  
    animais.push(Animal::Passaro{raca: "João-de-barro".to_string()});  
    animais.push(Animal::Passaro{raca: "Sabiá".to_string()});  
    animais.push(Animal::Cachorro);  
}
```

# Pattern Matching


- Na linguagem Rust temos a peculiaridade de enumerações não serem comparáveis (por padrão) através do operador `==`.
- Isso acontece porque temos variantes de enumerações, portanto o compilador não saberá dizer qual a variante está sendo solicitada em tempo de execução. Vejamos o exemplo abaixo:

```
enum Animal {  
    Cachorro,  
    Cobra(bool),  
    Passaro{raca: String},  
}
```



# Pattern Matching

```
enum Animal {  
    Cachorro,  
    Cobra(bool),  
    Passaro{raca: String},  
}  
  
fn main() {  
    let mut animais: Vec<Animal> = vec![];  
  
    animais.push(Animal::Cobra(true)); // Trocou de pele  
    animais.push(Animal::Passaro{raca: "João-de-barro".to_string()});  
    animais.push(Animal::Passaro{raca: "Sabiá".to_string()});  
    animais.push(Animal::Cachorro);  
  
    for animal in animais {  
        // ?  
    }  
}
```



O que colocamos aqui se quisermos mostrar os animais e seus campos?

# Pattern Matching

```
enum Animal {
    Cachorro,
    Cobra(bool),
    Passaro{raca: String},
}

fn main() {
    let mut animais: Vec<Animal> = vec![];

    animais.push(Animal::Cobra(true)); // Trocou de pele
    animais.push(Animal::Passaro{raca: "João-de-barro".to_string()});
    animais.push(Animal::Passaro{raca: "Sabiá".to_string()});
    animais.push(Animal::Cachorro);

    for animal in animais {
        if animal == Animal::Cachorro {
            println!("Isso é um cachorro");
        } else if animal == Animal::Cobra {
            println!("A cobra trocou de pele? {}", ? );
        }
        ...
    }
}
```

A primeira coisa que vem em mente é, tendo em vista que `Animal::Cobra` é uma enumeração que armazena uma tupla, então podemos acessar através dos indexadores (`animal.0` ou `animal.1`). Dará certo?

Isso pode funcionar num dado momento, mas como nós acessamos o valor armazenado na enumeração `Animal::Cobra`?

# Pattern Matching

- Para resolvermos o problema anterior, precisamos de uma expressão **match**

```
fn formatar_animal(animal: Animal) -> String {  
    match animal {  
        Animal::Cachorro => format!("O animal é um cachorro!"),  
        Animal::Cobra(trocou_pele) => {  
            let mut sim_nao = "sim";  
            if !trocou_pele { sim_nao = "não"}  
            format!("O animal é uma cobra e {} trocou de pele!", sim_nao)  
        },  
        Animal::Passaro { raca } => format!("O animal é um pássaro da raça {}", raca)  
    }  
}
```

# Pattern Matching

- Vamos pegar um caso mais simples
  - Padrões **consomem** valores
  - Expressões **produzem** valores

```
match animal {  
    Animal::Cachorro => format!("O animal é um cachorro!")  
}
```

Padrão (*pattern*)                      Expressão (retorno)

# Pattern Matching

- Vamos assumir que animal é
  - `Animal::Passaro{raca: "Sabiá".to_string()})`

<code>Animal::Passaro{raca: "Sabiá".to_string()})</code>	← Valor
↓ <span style="color: red; font-size: 2em;">✗</span>	
<code>Animal::Cachorro</code>	← Padrão

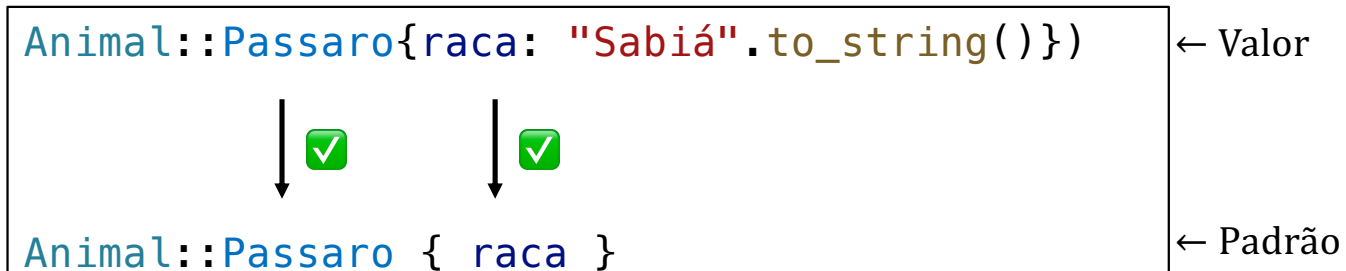
# Pattern Matching

- Vamos assumir que animal é
  - `Animal::Passaro{raca: "Sabiá".to_string()})`

<code>Animal::Passaro{raca: "Sabiá".to_string()})</code>	← Valor
↓ <b>×</b>	
<code>Animal::Cobra(trocou_pele)</code>	← Padrão

# Pattern Matching

- Vamos assumir que animal é
  - `Animal::Passaro{raca: "Sabiá".to_string()})`



# Pattern Matching

–Podemos utilizar o match com tipos primitivos também

```
fn main() {  
    let nota = 5;  
    match nota {  
        1 => { println!("Horrível") },  
        2 => { println!("Ok...") },  
        3 => { println!("Razoável") },  
        4 => { println!("Bom") },  
        5 => { println!("Excelente") },  
        _ => { }  
    }  
}
```



# Pattern Matching

–Podemos verificar um intervalo (1 até 5)

```
fn main() {  
    let nota = 5;  
    match nota {  
        1..=5 => {println!("Uma nota qualquer...")}  
        _ => {}  
    }  
}
```

# Pattern Matching

–Podemos verificar um intervalo (caracteres)

```
fn main() {  
    let letra = 'a';  
    match letra {  
        'a' ..= 'z' | 'A' ..= 'Z' => {println!("Uma letra qualquer...")},  
        '0' ..= '9' => {println!("Um dígito qualquer...")},  
        _ => {}  
    }  
}
```

# Pattern Matching

–Podemos verificar um intervalo e ainda termos *guards* (condições)

```
fn main() {  
    let idade = 18;  
    match idade {  
        i if i < 18 => { println!("Menor de idade") },  
        i if i >= 18 => { println!("Maior de idade") },  
        i if i >= 65 => { println!("Idoso") },  
        _ => {}  
    }  
}
```

# Exercícios

- Crie uma enumeração chamada Voto que represente diferentes tipos de votos, como voto a favor, voto contra e voto em branco. Em seguida, crie uma função que aceite uma lista de votos representados como uma lista de variantes da enumeração. Use a correspondência de padrões (*pattern matching*) para contar e imprimir o número total de votos a favor, votos contra e votos em branco.
- Crie uma enumeração chamada Carta que represente cartas de um baralho, incluindo naipes (paus, copas, espadas, ouros) e valores (Ás, 2 a 10, Valete, Dama, Rei). Em seguida, escreva uma função que aceite uma variante da enumeração Carta e retorne o valor da carta em pontos em um jogo de cartas simples. Use a correspondência de padrões para atribuir pontos com base na carta fornecida.

Contato: [p.horchulhack@pucpr.br](mailto:p.horchulhack@pucpr.br)