Enums

Andreea Costea

Delft University of Technology

2025-09-11

TU Delft

Enums - one of the more powerful kinds of types in Rust

```Rust
enum IpAddress {
    Ipv4,
    Ipv6,
}
```

Enums - one of the more powerful kinds of types in Rust

```Rust
enum IpAddress {
    Ipv4,
    Ipv6,
}
```

```Rust
enum HomeAddress {
        Ipv4Home,
        Ipv6Home,
        NotHome,
}
```

Enums - one of the more powerful kinds of types in Rust

```Rust
enum IpAddress {
    Ipv4,
    Ipv6,
}
```

- a list of values that are possible to represent.
- each possibility is called a “variant”.

Enums - one of the more powerful kinds of types in Rust

```Rust
enum IpAddress {
    Ipv4,
    Ipv6,
}
```

- a list of values that are possible to represent.
- each possibility is called a “variant”.
- each variant is an alternative value; pick a single value to instantiate:

```Rust
fn main() {
    let ip_type = IpAddress::Ipv4;
}
```

Enums

Each variant can have associated data with it:

```Rust
enum IpAddress {
    Ipv4(u8, u8, u8, u8),
    Ipv6(u16, u16, u16, u16, u16, u16, u16, u16),
}
```

Enums

Each variant can have associated data with it:

```rust
enum IpAddress {
    Ipv4(u8, u8, u8, u8),
    Ipv6(u16, u16, u16, u16, u16, u16, u16, u16),
}
```

- the associated data and the variant are bound together:

```rust
fn main() {
    let ipv4_home = IpAddress::Ipv4(127, 0, 0, 1);
    let ipv6_home = IpAddress::Ipv6(0, 0, 0, 0, 0, 0, 0, 1);
}
```

Enums

Each variant has a discriminant (hidden by default) – an isize number representing the index of the variant

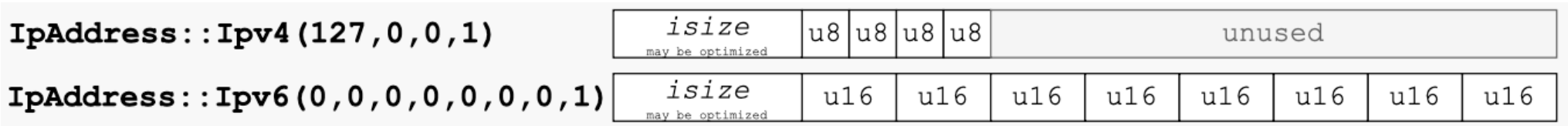
```rust
enum IpAddress {
    Ipv4(u8, u8, u8, u8),                          // = 0
    Ipv6(u16, u16, u16, u16, u16, u16, u16, u16),  // = 1
}
```

Enums

Each variant has a discriminant (hidden by default) – an isize number representing the index of the variant

```rust
enum IpAddress {
    Ipv4(u8, u8, u8, u8),                          // = 0
    Ipv6(u16, u16, u16, u16, u16, u16, u16, u16),  // = 1
}
```

An enum always is as large as the largest variant (plus the size of the discriminant)



Pattern Match: Extracting Data from an Enum (Cond)

Pattern Match: Extracting Data from an Enum (Cond)

Using the `if let [pattern] = [value]` expression:

```rust
fn accept_ipv4(ip: IpAddress) {
  if let IpAddress::Ipv4(a, b, _, _) = ip {
    println!("Accepted, first octet is {} and second is {}", a, b);
  }
}
```

Pattern Match: Extracting Data from an Enum (Cond)

Using the `if let [pattern] = [value]` expression:

```rust
fn accept_ipv4(ip: IpAddress) {
  if let IpAddress::Ipv4(a, b, _, _) = ip {
    println!("Accepted, first octet is {} and second is {}", a, b);
  }
}
```

- `a` and `b` introduce local variables binding values of corresponding variant fields.
- the underscore _ can be used to accept any value.

Pattern Match: Extracting Data from an Enum (Match)

```Rust
fn accept_home(ip: IpAddress) -> HomeAddress {
  match ip {
    IpAddress::Ipv4(127, 0, 0, 1) => HomeAddres::Ipv4Home,
    IpAddress::Ipv6(0, 0, 0, 0, 0, 0, 0, 1) => HomeAddres::Ipv6Home,
    _ => HomeAddres::NotHome,
  }
}
```

Pattern Match: Extracting Data from an Enum (Match)

```Rust
fn accept_home(ip: IpAddress) -> HomeAddress {
  match ip {
    IpAddress::Ipv4(127, 0, 0, 1) => HomeAddres::Ipv4Home,
    IpAddress::Ipv6(0, 0, 0, 0, 0, 0, 0, 1) => HomeAddres::Ipv6Home,
    _ => HomeAddres::NotHome,
  }
}
```

- every part of the match is called an arm.
- a match is exhaustive.
- a catch-all _ arm catches remaining cases if there are any left.

Expression Oriented Language? Say no more...

```Rust
fn get_first_byte(ip: IpAddress) {

  let first_byte = match ip {
    IpAddress::Ipv4(a, _, _, _) => a,
    IpAddress::Ipv6(a, _, _, _, _, _, _, _) => (a / 256) as u8,
  };

  println!("The first byte is: {}", first_byte);
}
```

Adding Behaviour

```rust
impl IpAddress {
 fn as_u32(self) -> Option<u32> {
  match self {
    IpAddress::Ipv4(a,b,c,d) =>
        Some( (a as u32) << 24 | (b as u32) << 16 |
              (c as u32) << 8  | (d as u32))
    _ => None,
}}}
fn main() {
  let addr = IpAddress::Ipv4(127, 0, 0, 1);
  println!("{:?}", addr.as_u32());
}
```

Options are Enums

```Rust
enum Option<T> {
    Some(T),
    None,
}
```

Options are Enums

```rust
enum Option<T> {
  Some(T),
  None,
}
```

```rust
fn print_option(x: Option<u32>) {
  match x {
    Some(value) => println!("the value inside is {value}"),
    None => println!("there was no value"),
  }
}
```

Utilities for Option

There are behaviours defined on options

```rust
impl<T> Option<T> {
  // all kinds of functions to work with options
}
```

Utilities for Option

There are behaviours defined on options—Important one: **map**

```Rust
impl<T> Option<T> {
    fn map<U>(self, /*...*/) -> Option<U>;
}
```

Utilities for Option

There are behaviours defined on options—Important one: **map**

```rust
impl<T> Option<T> {
    fn map<U>(self, /*...*/) -> Option<U>;
}
```

Map: translate the contents of an option

```rust
fn double(input: Option<u32>) -> Option<u32> {
    input.map(|inner| inner * 2)
}
```

Utilities for Option

There are behaviours defined on options—Important one: **map**

```rust
impl<T> Option<T> {
    fn map<U>(self, /*...*/) -> Option<U>;
}
```

Map: translate the contents of an option

```rust
fn double(input: Option<u32>) -> Option<u32> {
    input.map(|inner| inner * 2)
}
```

```rust
println!("{:?}", double(Some(2));  // Some(4)
```

Utilities for Option

When we want to work with what is **inside** an option:

```Rust
fn double(input: Option<u32>) -> Option<u32> {
    if let Some(inner) = input {
        Some(inner * 2)
    } else {
        None
    }
}
```

Utilities for Option

When we want to work with what is **inside** an option:

```rust
fn double(input: Option<u32>) -> Option<u32> {
  if let Some(inner) = input {
    Some(inner * 2)
  } else {
    None
  }
}
```

```rust
fn double(input: Option<u32>) -> Option<u32> {
  Some(input? * 2)
}
```

Utilities for Option

How to read the documentation

- `map` means to translate what's inside
- `unwrap` means to get what' inside
- `or` means to do something specific when there's nothing inside
- `is` gets some property of the value

Utilities for Option

How to read the documentation

- `map` means to translate what's inside
- `unwrap` means to get what' inside
- `or` means to do something specific when there's nothing inside
- `is` gets some property of the value

https://doc.rust-lang.org/stable/std/option/enum.Option.html

Examples

```rust
enum Coin {
 Heads,
 Tails,
}
```

Examples

```Rust
enum Coin {
 Heads,
 Tails,
}
```

```Rust
enum HttpRequest {
 Get(String),
 Post {url: String, body: String},
 Put  {url: String, body: String},
 Patch  {url: String, body: String},
 Delete {url: String, body: String},
}
```

Assignment: 10 minutes

Template:

```Rust
enum List {
  End,
  Item(u32, Box<List>)
}

impl List {
  fn create(&[u32]) -> List;
  fn length(&self) -> usize;
}
```

Use Case: Error Handling

Error Handling

Another `enum` in the standard library:

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

It's a bit like `Option`, if `None` were to have a value—same, same but different.

Error Handling

Another `enum` in the standard library:

```rust
enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

You'll often see functions like this:

```rust
fn do_thing(some_input: X) -> Result<Y, Error> {
  // if it goes well: Ok
  // if it goes bad: Err
}
```

Error Handling

Imagine reading from a file:

```Rust
fn read_from_file(filename: &Path) -> Result<String, Error> {
    // what could go wrong?
}
```

Error Handling

```rust
enum Error {
    DoesntExist,
    PermissionDenied,
    MemoryFull,
    InvalidName,
    TimedOut,
    ComputerOnFire, // ... etc
}

fn read_from_file(filename: &Path) -> Result<String, Error> {
    // what could go wrong?
}
```

Error Handling

```rust
enum MyError {
  DoesntExist,
  ComputerOnFire, // ... etc
}
fn read_from_file(filename: &Path) -> Result<String, MyError> {
  if !check_if_file_exists(filename) {
    return Err(MyError::DoesntExist)
  }
  // ... some more checks
  // and then finally:
  Ok(contents)
}
```

Error Handling

Match on the result of reading from file:

```rust
match read_from_file("data.txt") {
  Ok(contents)              => println!("the file contains {contents}"),
  Err(MyError::DoesntExist) => println!("the file doesn't exist"),
  Err(e)                    => println!("another error occurred: {e}"),
}
```

Common Error Handling Pattern

```Rust
fn do_big_thing() -> Result<Output, Error> {
  let outcome1 = match do_small_thing1() {
    Ok(i) => i,
    Err(e) => return Err(e),
  };
  let outcome2 = match do_small_thing2(outcome1) {
    Ok(i) => i,
    Err(e) => return Err(e),
  };
  Ok(outcome2) }
```

Common Error Handling Pattern

```rust
fn do_big_thing() -> Result<Output, Error> {
    let outcome1 = match do_small_thing1() { Ok(i) => i, Err(e) => return Err(e)};
    let outcome2 = match do_small_thing2(outcome1) { Ok(i) => i, Err(e) => return Err(e)};
    let outcome3 = match do_small_thing3(outcome2) { Ok(i) => i, Err(e) => return Err(e)};
    let outcome4 = match do_small_thing4(outcome3) { Ok(i) => i, Err(e) => return Err(e)};
    let outcome5 = match do_small_thing5(outcome4) { Ok(i) => i, Err(e) => return Err(e)};
    Ok(outcome5) }
```

Common Error Handling Pattern

So much boilerplate!

Common Error Handling Pattern

Use ? to do exactly the same:

```rust
let outcome1 = do_small_thing1()?;
```

means

```rust
let outcome1 = match do_small_thing1() {
    Ok(i) => i,
    Err(e) => return Err(e)
};
```

Common Error Handling Pattern

So the big example from before becomes:

```rust
fn do_big_thing() -> Result<Output, Error> {
    let outcome1 = do_small_thing1()?;
    let outcome2 = do_small_thing2(outcome1)?;
    let outcome3 = do_small_thing3(outcome2)?;
    let outcome4 = do_small_thing4(outcome3)?;
    let outcome5 = do_small_thing5(outcome4)?;

    Ok(outcome5)
}
```

Error handling in real life:

```Rust
pub enum FromFileError {
    Io(io::Error),
    Json(serde_json::Error),
    Yaml(serde_yaml::Error),
    Plist(plist::Error),
}

fn from_file(path: &Path) -> Result<Something, FromFileError>;
```

- This is an error for a single function `from_file`
- A different function might have different ways to fail
- It can fail in 4 different major ways

Error handling in real life:

```Rust
use thiserror::Error;

#[derive(Error, Debug)]
pub enum FromFileError {
    Io(#[from] io::Error), // ...
}
```

Error handling in real life:

```rust
use thiserror::Error;

#[derive(Error, Debug)]
pub enum FromFileError {
    Io(#[from] io::Error), // ...
}
```

`#[derive(Error, Debug)]`

- Error: generates an implementation of `std::error::Error` for your enum.
- Debug: lets you print the enum with the `{:?}` formatter for debugging.

Together, these two macros save you from manually implementing the boilerplate needed to make your enum a proper error type.

Error handling in real life:

```rust
use thiserror::Error;

#[derive(Error, Debug)]
pub enum FromFileError {
    Io(#[from] io::Error), // ...
}
```

`#[from]` automatically implements `From <io::Error>` for `FromFileError`.

```rust
impl From<io::Error> for FromFileError {
    fn from(e: io::Error) -> Self {
        FromFileError::Io(e)     } }
```

Error handling in real life:

```Rust
fn read_file() -> Result<String, io::Error> {/* ... */}

fn from_file() -> Result<String, FromFileError> {
  let contents = read_file()?; // does this work?


}
```

Error handling in real life:

```Rust
fn read_file() -> Result<String, io::Error> {/* ... */}

fn from_file() -> Result<String, FromFileError> {
    let contents = match read_file() {
        Ok(i) => i,
        //  v-- io::Error
        Err(e) => return Err(e),
        //    FromFileError--^
    }
}
```

Error handling in real life:

```rust
fn read_file() -> Result<String, io::Error> {/* ... */}

fn from_file() -> Result<String, FromFileError> {
    let contents = match read_file() {
        Ok(i) => i,
        //  v-- io::Error       vvvv-- automatically convert
        Err(e) => return Err(e.into()),         // FromFileError::Io(e)
        //    FromFileError--^^^^^^^^
    }
}
```

Error handling in real life:

```Rust
fn read_file() -> Result<String, io::Error> {/* ... */}

fn from_file() -> Result<String, FromFileError> {
    // this works! automatic conversion
    let contents = read_file()?;


}
```

Note: '?' also works for options!

```rust
fn double(input: Option<u32>) -> Option<u32> {
  Some(inner? * 2)
}
```

equivalent to:

```rust
fn double(input: Option<u32>) -> Option<u32> {
  if let Some(inner) = input {
    Some(inner * 2)
  } else {
    None
  }
```

Note: '?' also works for options!

```
7   }
```

Note: '?' also works for options!

```rust
fn first_char(s: &str) -> Option<char> {
    s.chars().next()? // returns None if empty, Some(c) otherwise
}

fn main() {
    println!("{:?}", first_char("hi")); // Some('h')
    println!("{:?}", first_char(""));   // None
}
```

Misc

Utilities for Result

How to read the documentation

https://doc.rust-lang.org/stable/std/result/enum.Result.html

Enums

We call `NoConstructors` the “never type” and you can write it as `!`

```rust
fn example() -> ! {
    // never returns
    loop {}
}
```

- `panic!()` returns `!`
- `std::process::exit()` returns `!`
- `loop{}` returns `!`
- When there’s no operating system, `main` returns `!`

Enums and Patterns

- Enums have multiple constructors
- Patterns can discern between constructors of a type

```rust
enum WeekDay {
    Monday,
    Tuesday,
    // ...
    Friday,
}

let x = WeekDay::Friday;

```

Enums and Patterns

```
10  match x {
11    WeekDay::Saturday | WeekDay::Sunday => println!("weekend"),
12    _ => println!("not weekend"),
13  }
```