Rust Workshop

Day 2

Recap of Day 1

Variables

```rust
let x = 5;
let mut x = 5;
let x: i32 = 5;

const MAGIC_NUMBER: i32 = 42;
static FAST_DATA: &str = "Performante Daten"
```

Types

```
let b: u8 = b'A';
let l: usize = "some string".len();
let c: char = '🦀';
let x: (u8, char) = (b'A', '🦀');
let void: () = ();
let arr: [i32; 5] = [100; 5];
```

Functions

```
1  fn identity(x: i32) -> i32 { x }
```

Control Flow

```
let abs = if x < 0 { -x } else { x };
loop {
    if done() {
        break;
    } else {
        continue;
    }
}
while true {}
for elem in [10, 20, 30] {
    println!("the number is: {}", elem);
}
```

Ownership Rules

1. Every value has exactly one owner.
2. When the owner goes out of scope, the destructor is run.

Borrrowing Rules

1. ONE mutable reference OR many immutable ones.
2. References always point to valid memory.

```
let read_only_ref: &String = &owned_string;
let mutable_ref: &mut String = &mut owned_string;
```

Slices

```
let byte_slice: &[u8] = &[1, 2, 3, 4, 5];
let middle_part = &byte_slice[1..4];
let string_slice: &str = "valid UTF-8";
```

Language Basics 2

book chapters 5 – 9

- structs & methods
- enums & pattern matching
- modules
- collections
- error handling

Structs & Methods

book chapter 5

Declaration

```
struct Rectangle {
    width: u32,
    height: u32,
}
```

Instantiation

```
let rect = Rectangle {
    width: 200,
    height: 100,
};
```

Field Access

```
1  rect.width = 120;
2  let area = rect.width * rect.height;
```

Tuple Structs

```
1  struct Color(u8, u8, u8);
2
3  fn main() {
4      let color = Color(12, 200, 85);
5      let green = color.1;
6  }
```

Tuple Structs: Encapsulation

a.k.a. the newtype pattern

```
1  struct NonZeroByte(u8);
2
3  fn into_non_zero_byte(raw: u8) -> NonZeroByte {
4      if raw == 0 {
5          panic!("You had one job!")
6      }
7      NonZeroByte(raw)
8  }
```

Unit-Like Structs

```
1  struct OnePossibleValue;
2
3  fn main() {
4      let seems_useless: OnePossibleValue = OnePossibleValue;
5  }
```

Extremely useful with traits!

Sit tight for day 3.

Ownership of Struct Data

```
1  struct User {
2      name: &str,
3  }
```

It’s possible, but don’t try it yet!

Sit tight for day 3.

Deriving Traits

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}
fn main() {
    let rect = Rectangle { width: 30, height: 50 };
    println!("rect has the value: {:?}", rect);
    //                                  ^^^^
    //     note the debug-print syntax
}
// output:
// rect has the value: Rectangle { width: 30, height: 50 }
```

Sit tight for traits on day 3! 😄

Methods

book chapter 5.3

Syntax

```
1  struct Rectangle {
2      width: u32,
3      height: u32,
4  }
5  impl Rectangle {
6      fn area(&self) -> u32 { // area(self: &Rectangle)
7          self.width * self.height
8      }
9  }
10 fn main() {
11     let rect = Rectangle { width: 30, height: 50 };
12     let a: u32 = rect.area();
13     // this works:
14     let a: u32 = Rectangle::area(&rect);
15 }
```

Automatic (De-)Referencing

```
1  fn main() {
2      let rect = Rectangle { width: 30, height: 50 };
3
4      println!("{}", rect.area()); // adds 1 reference
5
6      let rect: &&Rectangle = &&rect;
7      println!("{}", rect.area()); // removes 1 reference
8
9      let rect = &&&&&&&&rect;
10     println!("{}", rect.area()); // ok we get it
11 }
```

Additional Parameters

```
impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
fn main() {
    let rect_1: Rectangle;
    let rect_2: Rectangle;
    if rect_1.can_hold(&rect_2) {
        // ...
    }
}
```

Associated Functions

```
1  impl Rectangle {
2      fn square(size: u32) -> Self {
3          Self {
4              width: size,
5              height: size,
6          }
7      }
8  }
9  fn main() {
10     let square = Rectangle::square(3);
11 }
```

Multiple `impl` Blocks

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
```

Summary

structs and methods

- group data meaningfully
- combine data with related behavior

Enums & Pattern Matching

book chapter 6

Declaration

```
1  enum IpAddrKind {
2      V4,
3      V6,
4  }
```

Instantiation

```
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

Storing Data ?

```
enum IpAddrKind {
    V4,
    V6,
}
struct IpAddr {
    kind: IpAddrKind,
    address: [u8; 16], // v4 and v6 both fit in here
}
fn main() {
    let home = IpAddr {
        kind: IpAddrKind::V4,
        address: [127, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    };
    home.address[8]; // ???
}
```

Enums are Tagged Unions

```
enum IpAddr {
    V4([u8; 4]),
    V6([u8; 16]),
}
fn main() {
    let home = IpAddr::V4([127, 0, 0, 1]);

    home.0[8]; // nice try :-)
}
```

Flexible Data Modelling

```
enum Message {
    Quit,                       // ~= unit-like struct
    Move { x: i32, y: i32 },    // ~= regular struct
    Write(String),              // ~= tuple struct
    ChangeColor(i32, i32, i32),
}
```

Billion-Dollar Mistake

The C pointer mixes two unrelated concepts:

- Indirection
- Optionality (something or nothing)

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965.
>
> *Tony Hoare*

Rust:

Indirection:	Reference `&`
Optionality:	❓

The `Option` Type

```
enum Option<T> {
    None,
    Some(T),
}
main() {
    let some_number = Option::Some(5);
    let some_char = Some('e');

    let absent_number: Option<i32> = None;
}
```

Pattern Matching

book chapter 6.2

The `match` Expression

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Patterns that Bind to Values

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
fn receive(msg: Message) {
    match msg {
        Message::Quit => println!("bye bye!"),
        Message::Move { x, y } => set_position(x, y),
        Message::Write(text) => println!("{}", text),
        Message::ChangeColor(r, g, b) => {
            set_red(r);
            set_green(g);
            set_blue(b);
        }
    }
}
```

Matching on `Option`

```
1  fn plus_one(x: Option<i32>) -> Option<i32> {
2      match x {
3          None => None,
4          Some(i) => Some(i + 1),
5      }
6  }
7  fn main() {
8      let five = Some(5);
9      let six = plus_one(five);
10     let none = plus_one(None);
11 }
```

Exhaustive Matching

```
1  fn plus_one(x: Option<i32>) -> Option<i32> {
2      match x {
3          Some(i) => Some(i + 1),
4          // forgot about None!
5      }
6  }
```

compiler says:

```
non-exhaustive patterns: None not covered
```

Exhaustive Matching

demo

Catch-all Patterns

```
match 0_i32 {
    7  => println!("You are lucky! 🥳"),
    13 => println!("You are unlucky! 😢"),
    x  => println!("squared: {}", x * x),
}
```

Catch-all Patterns

```
match 0_i32 {
    7  => println!("You are lucky! 🥳"),
    13 => println!("You are unlucky! 😢"),
    x  => println!("squared: {}", x * x),
}
```

Catch-all Patterns

```
match 0_i32 {
    7  => println!("You are lucky! 🥳"),
    13 => println!("You are unlucky! 😢"),
    _  => {}
}
```

Boilerplate...

```
match option_num {
    Some(num) => {
        println!("So much typing 🤧");
    }
    _ => {}
}
```

Matching on a Single Pattern

```
let maybe_num: Option<i32> = Some(10);
if let Some(num) = maybe_num {
    println!("number detected: {}", num);
}
```

Looping `while` a Pattern Matches

```
let mut numbers = vec![1, 2, 3];
while let Some(n) = numbers.pop() {
    println!("removed from vec: {}", n);
}
```

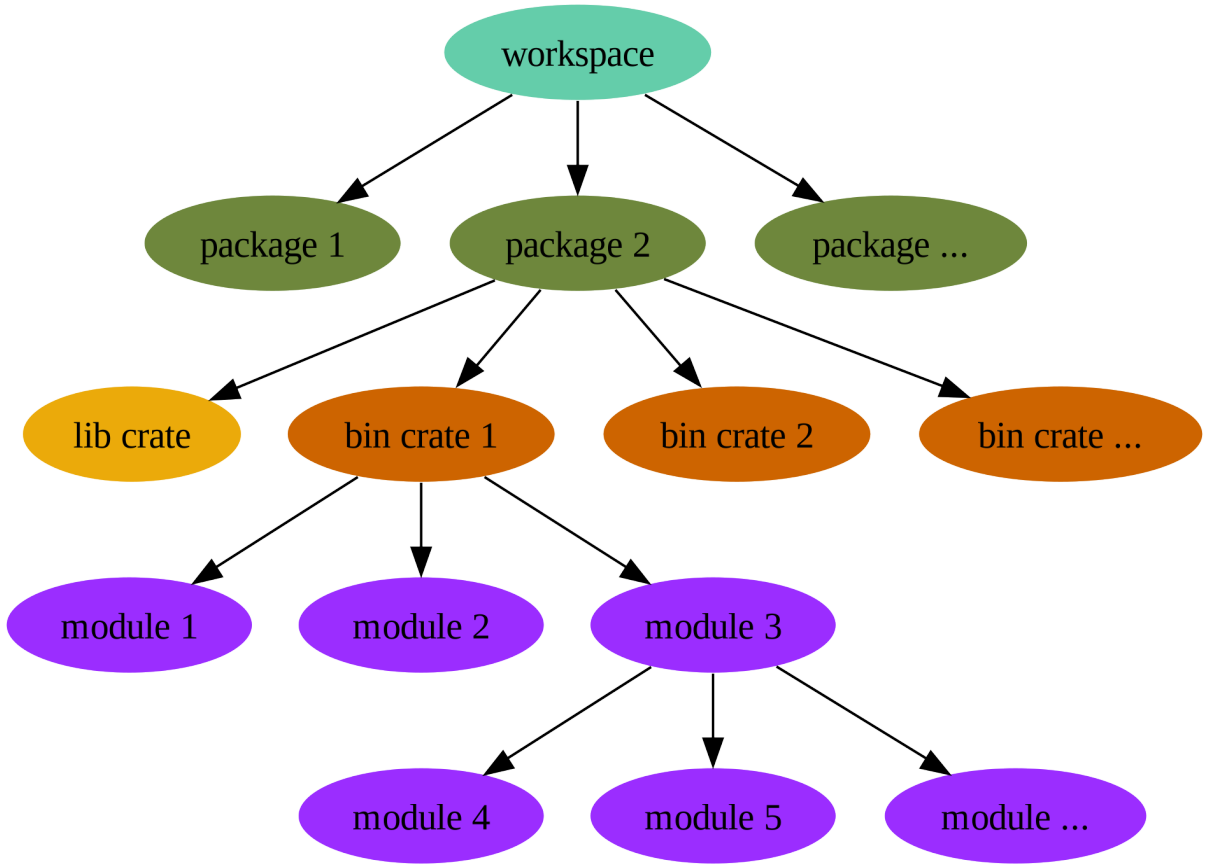
Summary

enums and pattern matching

- model alternatives in your data
- prevent invalid data access bugs
- branch over data structures with `match`

Project Organization

book chapter 7


workspace
package 1
package 2
package ...
lib crate
bin crate 1
bin crate 2
bin crate ...
module 1
module 2
module 3
module 4
module 5
module ...

The Crate

- executable crate: `main.rs` – library crate: `lib.rs`
- the basic compilation unit for `rustc`
- can comprise multiple `.rs` files

Simplification: `main.rs` is the crate!

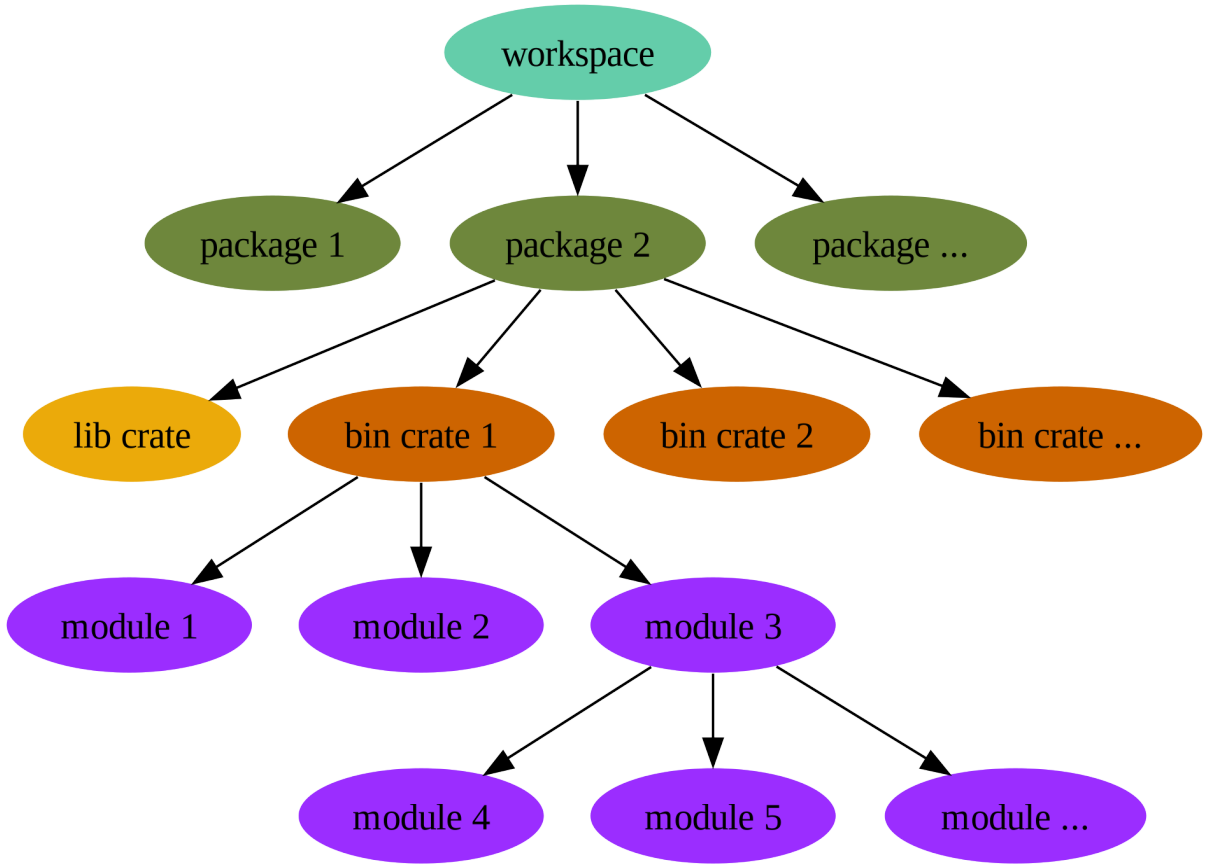
Above the Crate: The Package

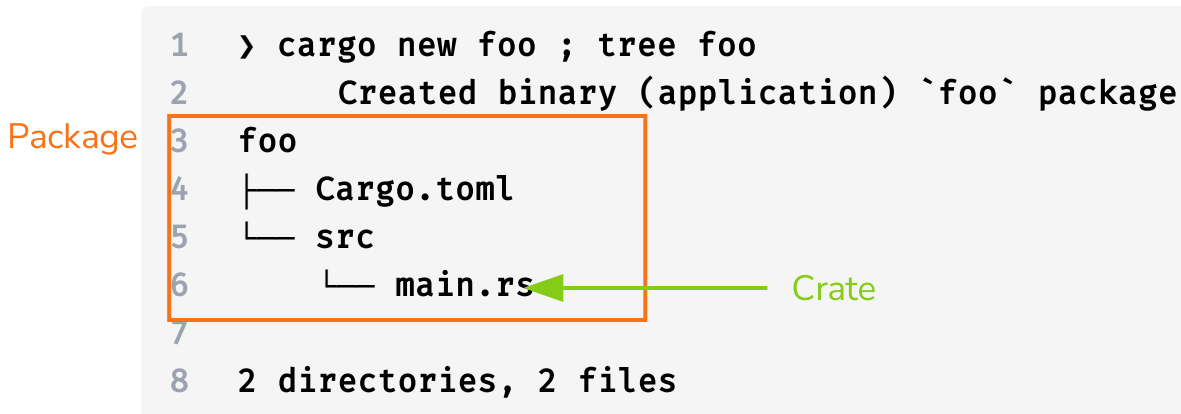
- `Cargo.toml`, the basic build unit for `cargo`
- build scripts, e.g. for linking against C-libs
- tests against the public API of a crate
- semantically versioned

Simplification: `Cargo.toml` is the package!

Below the Crate: Modules

- Code organziation
- Visibility & Encapsulation
- What we'll focus on next!


workspace
package 1
package 2
package ...
lib crate
bin crate 1
bin crate 2
bin crate ...
module 1
module 2
module 3
module 4
module 5
module ...


Package
1 ❯ cargo new foo ; tree foo
2 Created binary (application) `foo` package
3 foo
4 ├── Cargo.toml
5 └── src
6 └── main.rs
7
8 2 directories, 2 files
Crate

Modules & Visibility

book chapter 7

Declaring a Module

demo

The `pub` Keyword

```
1  fn main() {
2      garden::carrot();
3      garden::lettuce(); // error
4  }
5  mod garden {
6      pub fn carrot() {}
7      fn lettuce() {}
8  }
```

compiler says:

```
function `lettuce` is private
```

Nesting

```
fn main() {
    garden::fruits::orange();   // ✅
    garden::veggies::lettuce(); // error
}
mod garden {
    pub mod fruits {
        pub fn orange() {}
    }
    mod veggies {
        pub fn lettuce() {}
    }
}
```

compiler says:

```
module veggies is private
```

Paths

```
mod garden {
    pub mod fruits {
        pub fn orange() {}
    }
    fn do_stuff() {
        // relative path
        fruits::orange();
    }
    mod veggies {
        fn lettuce() {
            // absolute path
            crate::garden::fruits::orange();
            // walking the module tree backwards
            super::fruits::orange()
        }
    }
}
```

Struct Field Visibility

```
mod garden {
    pub struct Melon {
        pub size: u32,
        ripeness: u32,
    }
}
fn main() {
    let m: garden::Melon; // imagine initialization

    m.size; // ok
    m.ripeness; // error, field is private
}
```

The `use` Keyword

```
1  mod garden {
2      pub struct Melon;
3  }
4  fn main() {
5      let m: garden::Melon;
6
7      use garden::Melon;
8      let m: Melon;
9  }
```

Using External Packages

```
# Cargo.toml
[dependencies]
rand = "0.8.5"
```

```
fn main() {
    let rand_num: i32 = rand::random();
    println!("your lucky number: {}", rand_num);
}
```

Glob Imports

```
use std::collections::*;
fn main() {
    let hm: HashMap;
    let bm: BTreeMap;
}

// common pattern for framework-like libraries:
use leptos::prelude::*;
```

Re-Exporting with `pub use`

```
mod garden {
    mod fruits {
        pub struct Melon;
        pub struct Orange;
    }
    pub use fruits::Melon;
}
fn main() {
    let m: garden::Melon;
    // doesn't work, because garden doesn't re-export Orange
    let o: garden::Orange;
    // doesn't work, because fruits is private
    let o: garden::fruits::Orange;
}
```

Renaming Imports with `as`

```
mod german {
    pub struct Kartoffel;
}
mod swiss_german {
    pub use super::german::Kartoffel as Herdoepfel;
}
fn main() {
    let same_thing: german::Kartoffel = swiss_german::Herdoepfel;
}
```

Summary

modules and visibility

- `main.rs` is the root of the crate.
- Modules can be defined inline or in separate files.
- The module tree is traversed with `::`, `super` and `crate`.
- Items are private unless made `pub`-lic.
- Verbose module path specifiers are avoided with `use`.
- Re-exporting allows fine-grained control over visibility.

Collections

book chapter 8

Reading from Vectors Safely

```rust
fn main() {
    let v = vec![1, 2, 3, 4, 5];

    v[10]; // panics immediately! 😱

    let tenth: Option<&i32> = v.get(10); // safety: 💯
    match tenth {
        Some(tenth) => println!("The tenth element is {}", tenth),
        None => println!("There is no tenth element."),
    }
}
```

HashMap

a.k.a. `dict`, `map`, hash table, associative array

```rust
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();

    scores.insert("Brazil", 1);
    scores.insert("Germany", 7);

    let germany_score: Option<&i32> = scores.get(&"Germany");

    for (key, value) in scores {
        println!("{} scored {} points!", key, value);
    }
}
```

`std::collections::*`

- Queue
- Linked-List
- Set
- Heap

Error Handling

book chapter 9

Quick and Dirty

stack unwinding

```
fn main() {
    let v = vec![1, 2, 3];
    let x: i32 = v[100]; // <- panic
    let x: Option<i32> = None;
    let x: i32 = x.unwrap(); // <- panic
    panic!("custom panic message");
    todo!(); // shush compiler, I'm not done
}
```

The `Result` Enum

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

recall the `Option` type:

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

Error Handling

demo

Practice 

`rust-exerices/day_2/README.md`

Please suggest improvements for next week!

Check the readme of your repository for the form link.