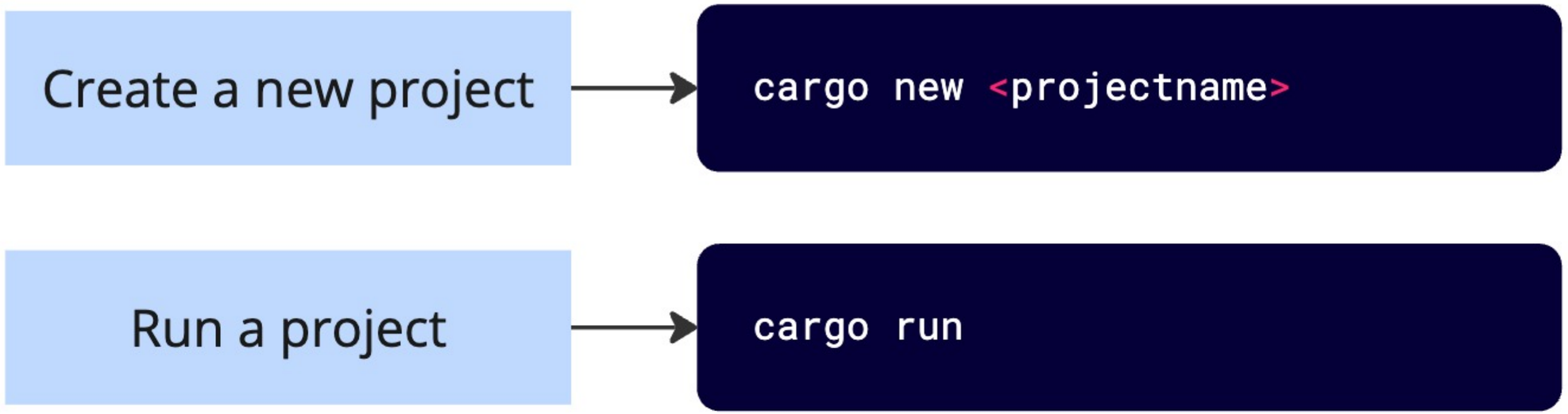
1	Install Rust	→	**rust-lang.org/tools/install**
2	Create a Project	→	Run *cargo new <projectname>* at the terminal
3	Setup VSCode	→	Install the *rust-analyzer* extension
4	Write code!	→	Let's do it!


1
Install Rust
rust-lang.org/tools/install

Project #1

Write a Rust program that simulates a deck of playing cards

Cargo is used to create, compile, run, and manage projects



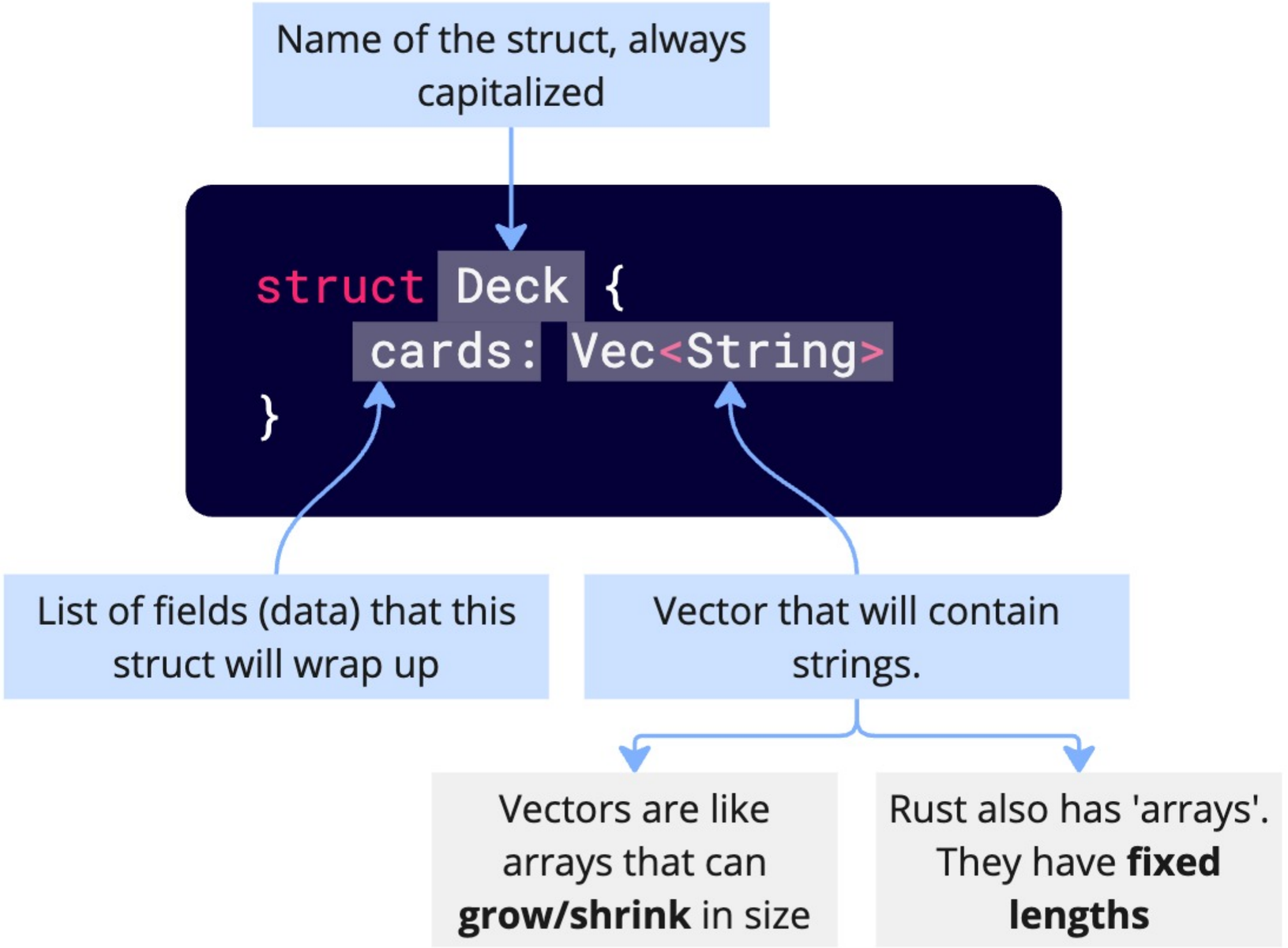
We want to store some data and attach some functionality to it

↓

In Rust, a good tool for this is a **struct**

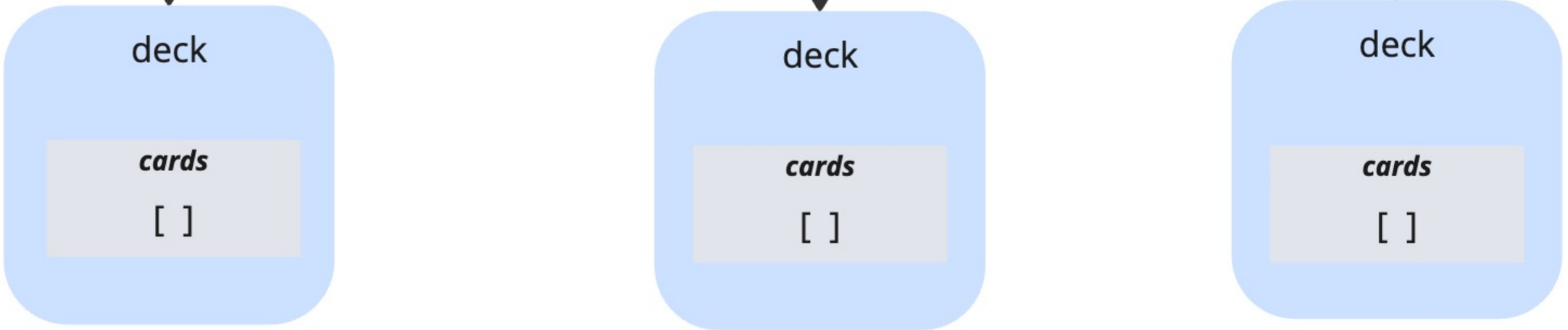
↓

Structs are kind of like classes from other languages

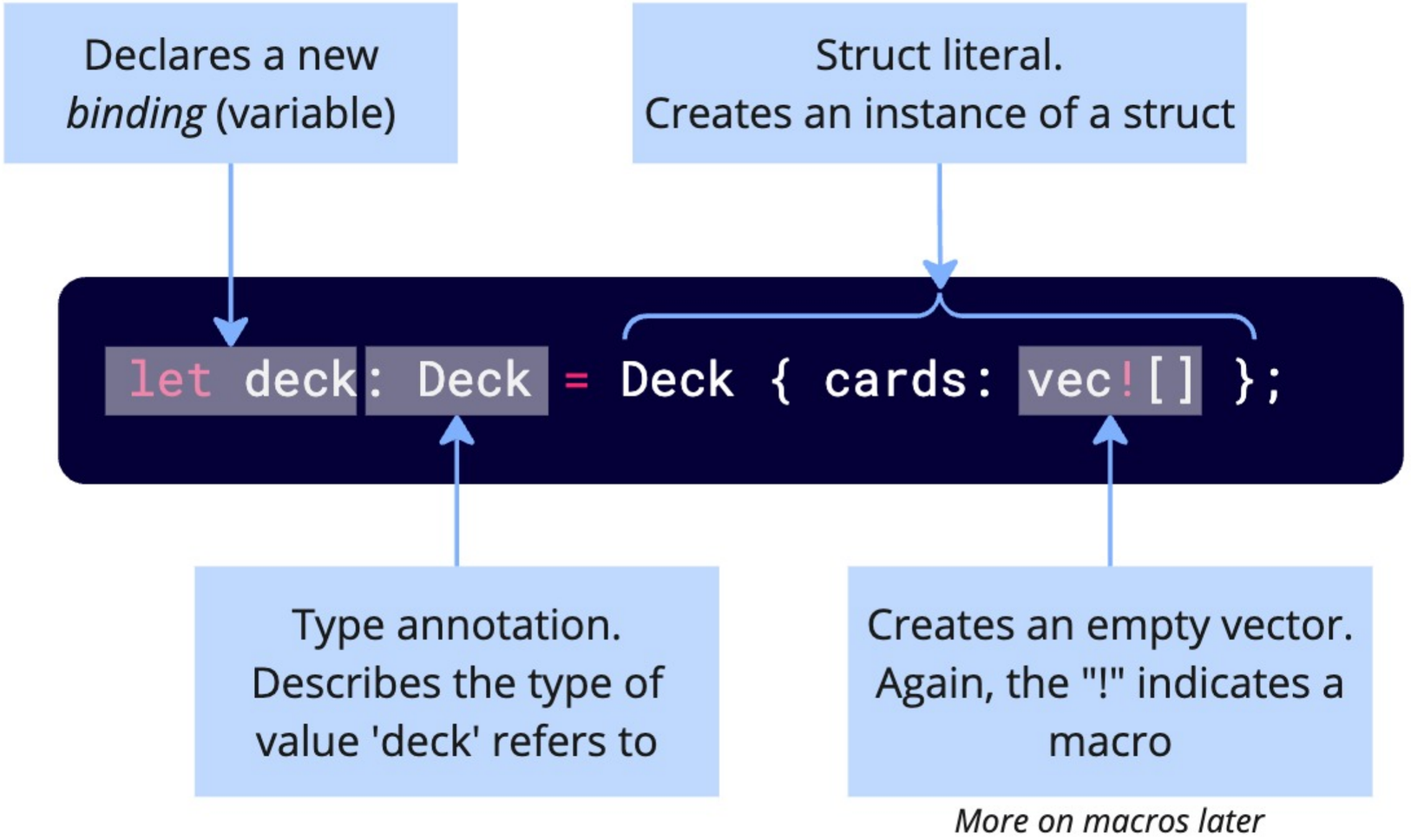

Name of the struct, always capitalized
struct Deck {
cards: Vec<String>
}
List of fields (data) that this struct will wrap up
Vector that will contain strings.
Vectors are like arrays that can **grow/shrink** in size
Rust also has 'arrays'. They have **fixed lengths**

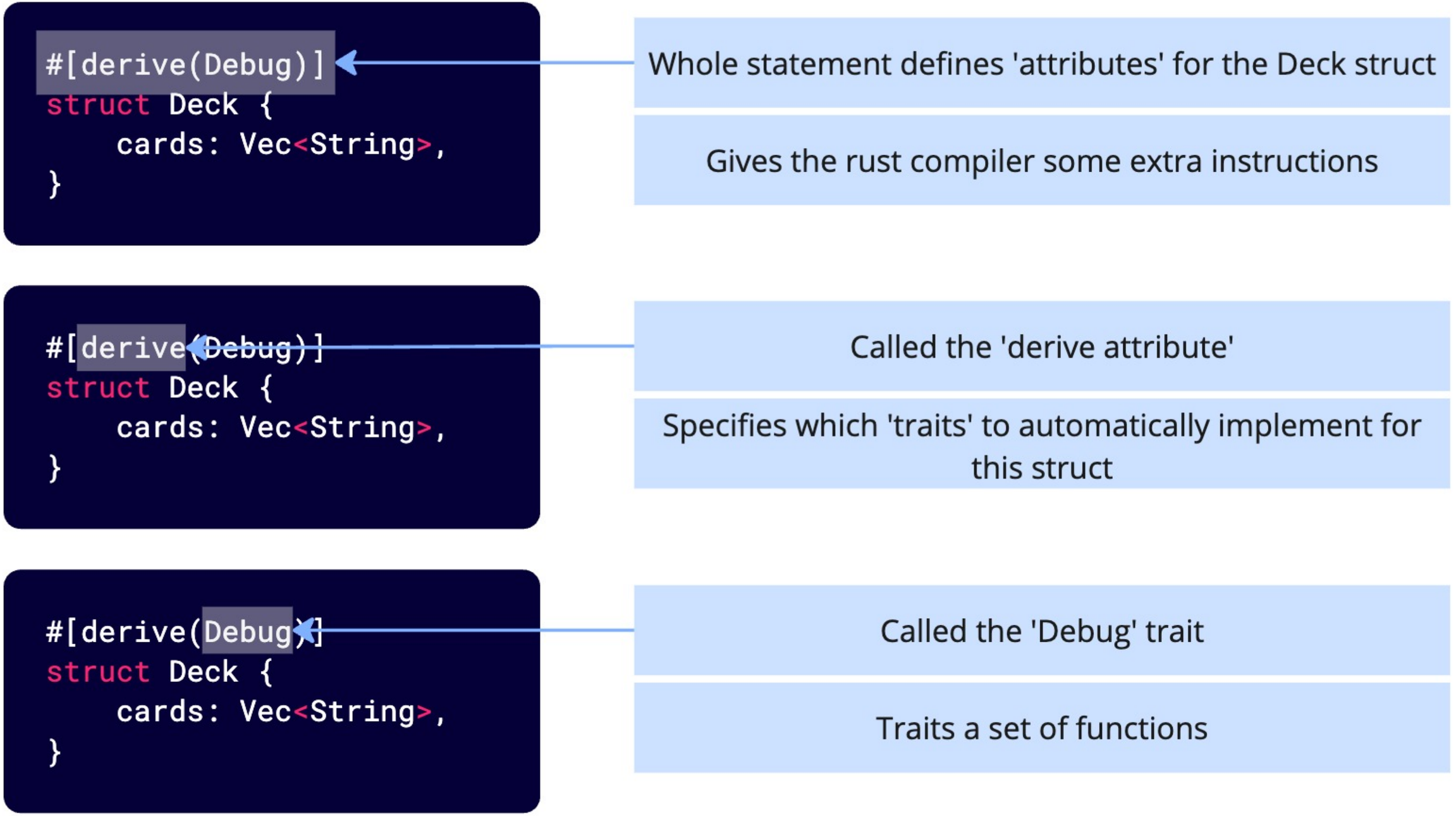
Struct Definition

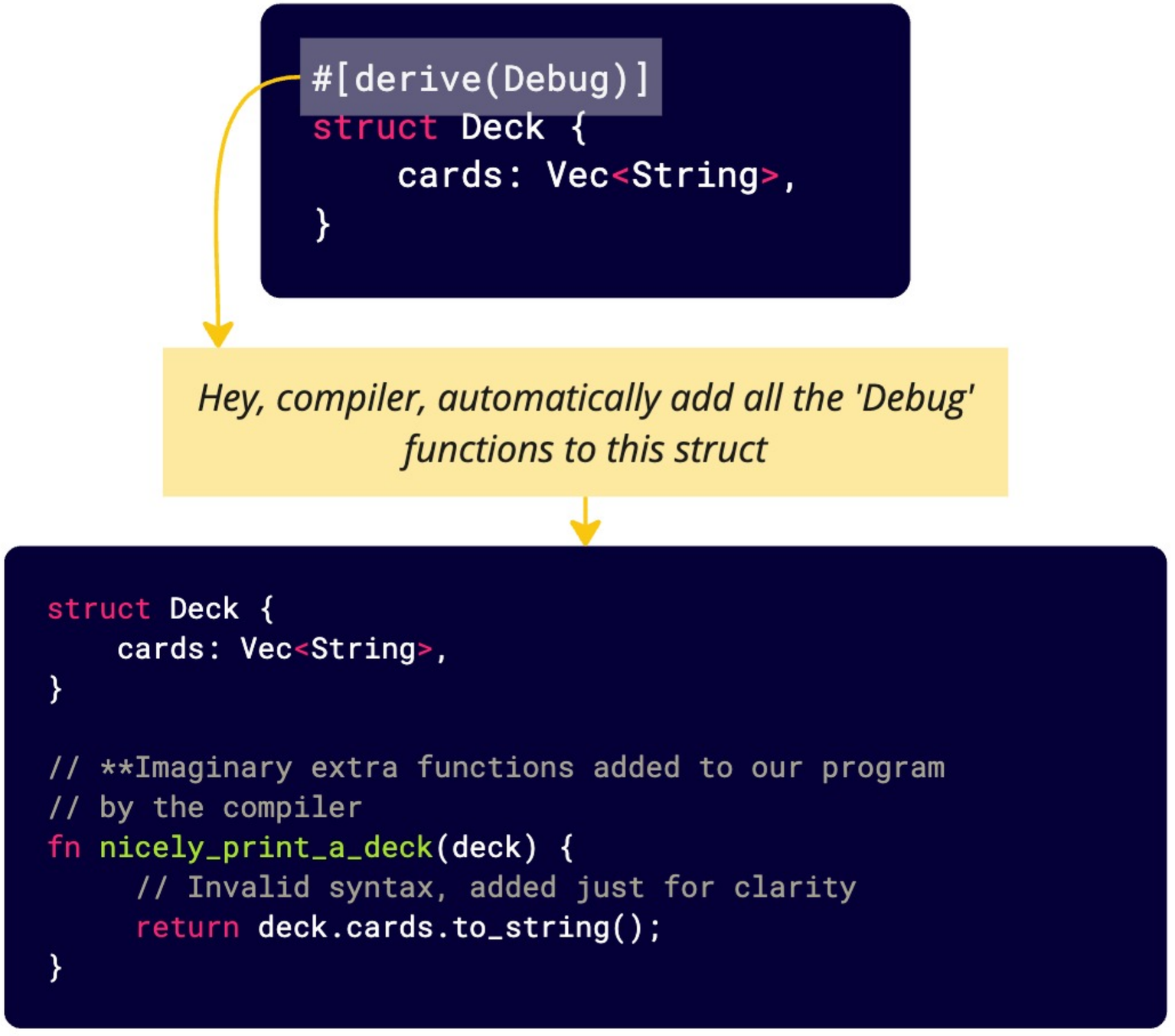
```
struct Deck {
    cards: Vec<String>
}
```



Instances of the 'Deck' struct


Declares a new
binding (variable)
Struct literal.
Creates an instance of a struct
let deck: Deck = Deck { cards: vec![] };
Type annotation.
Describes the type of
value 'deck' refers to
Creates an empty vector.
Again, the "!" indicates a
macro
More on macros later


#[derive(Debug)]
struct Deck {
cards: Vec<String>,
}
Whole statement defines 'attributes' for the Deck struct
Gives the rust compiler some extra instructions
#[derive(Debug)]
struct Deck {
cards: Vec<String>,
}
Called the 'derive attribute'
Specifies which 'traits' to automatically implement for this struct
#[derive(Debug)]
struct Deck {
cards: Vec<String>,
}
Called the 'Debug' trait
Traits a set of functions


#[derive(Debug)]
struct Deck {
cards: Vec<String>,
}
Hey, compiler, automatically add all the 'Debug' functions to this struct
struct Deck {
cards: Vec<String>,
}
// **Imaginary extra functions added to our program
// by the compiler
fn nicely_print_a_deck(deck) {
// Invalid syntax, added just for clarity
return deck.cards.to_string();
}

Bindings are **immutable** (can't change) by default

Immutable

```
let numbers = vec![];

// Error! Can't change the value
numbers.push(1); // Error!

// Error! Can't reassign either!
numbers = vec![];
```

Mutable

```
let mut numbers = vec![];

// Ok!
numbers.push(1);

// Ok!
numbers = vec![];
```

```
let suits = ["Diamonds", "Clubs"];
let values = ["2", "3", "4", "5"];
```

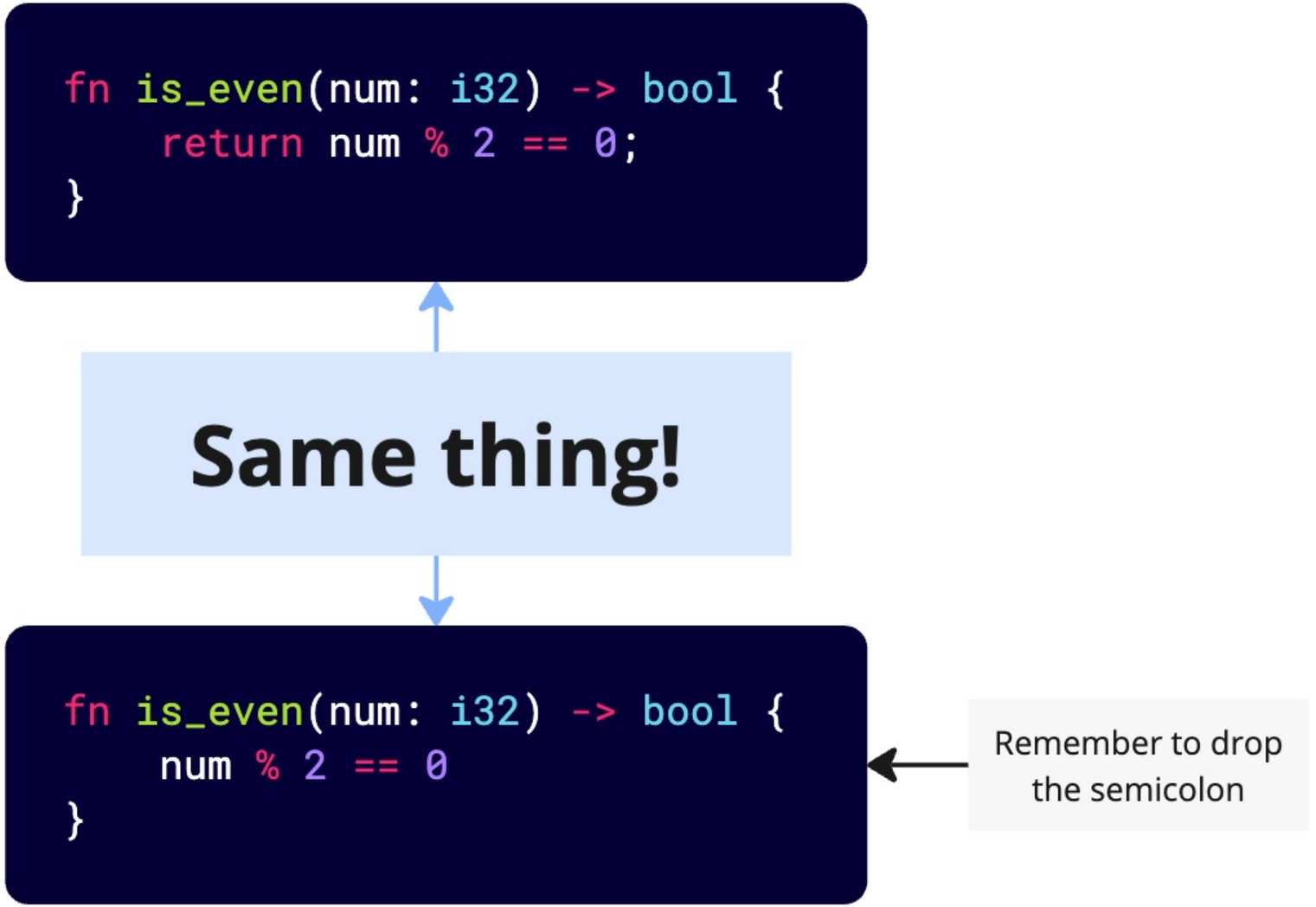
Creates an **array** of strings.
Arrays are fixed in size (can't grow/shrink)

```
let suits = vec!["Diamonds", "Clubs"];
let values = vec!["2", "3", "4", "5"];
```

Creates a **Vector** of strings.
Vectors are dynamic (they can grow/shrink)

Are the suits/values lists going to change over time? If not, maybe use an array!

Implicit Return



```rust
fn is_even(num: i32) -> bool {
    if num % 2 == 0 {
        true
    } else {
        false
    }
}
```

Implicit return will automatically return the last executed expression in the function

Impl definition. Same name as struct

```
impl Deck {
    fn new() {
        // stuff...
    }

    fn shuffle(&self) {
        // stuff...
    }
}

fn main() {
    let deck = Deck::new();

    deck.shuffle();
}
```

Inherent Implementations

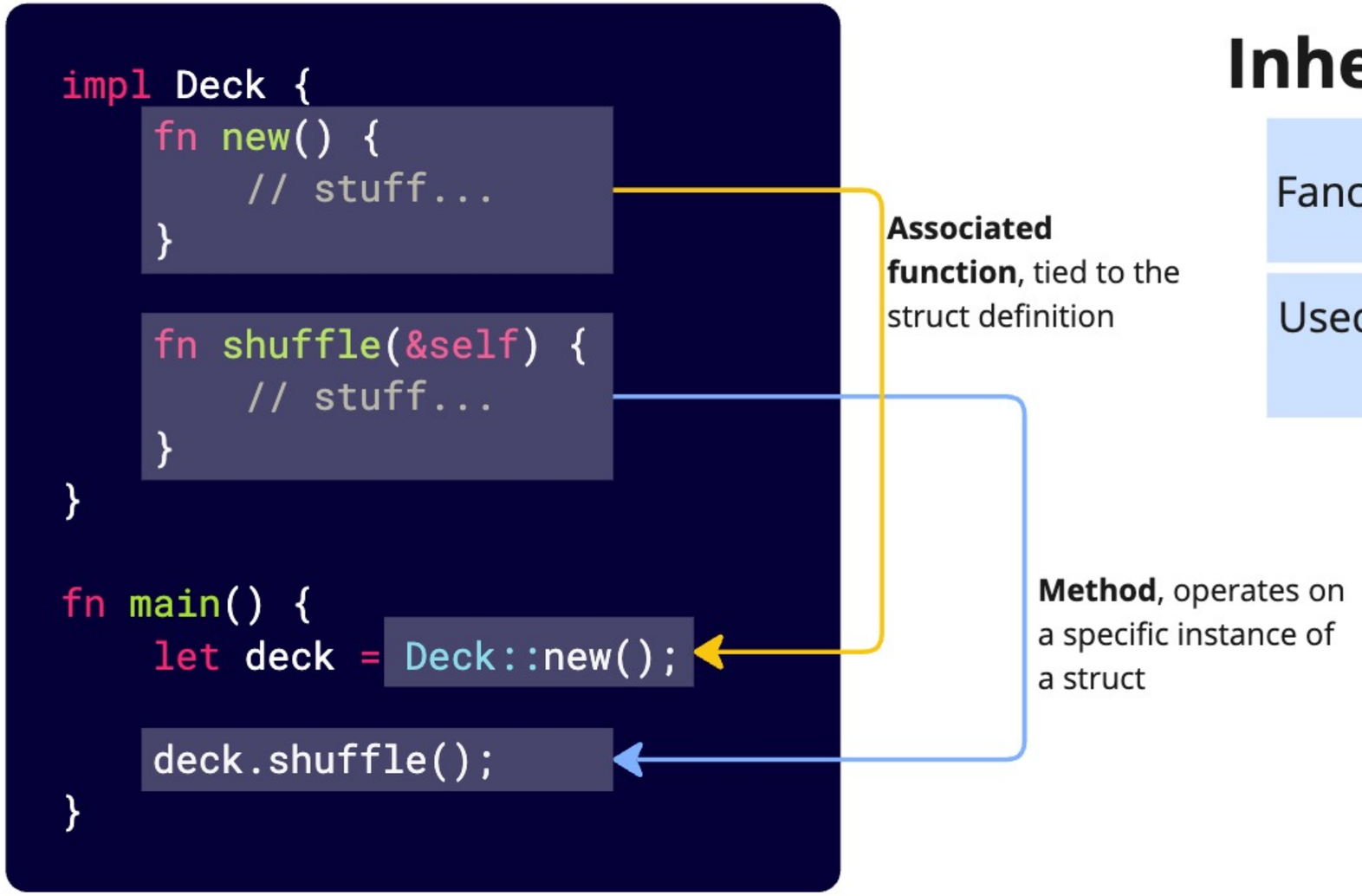
Fancy term for 'add a function to a struct'

Used to define **methods** and **associated functions**

Inherent Implementations

Fancy term for 'add a function to a struct'

Used to define **methods** and **associated functions**



```rust
impl Deck {
    fn new() -> Self {
        // stuff...
    }
}

fn main() {
    Deck::new();
}
```

Associated Functions

Use when you have functionality not tied to a specific instance

Examples: 'full_deck()', 'with_n_cards(10)', 'empty_deck()'

```rust
impl Deck {
    fn shuffle(&self) {
        // stuff...
    }
}

fn main() {
    deck.shuffle();
}
```

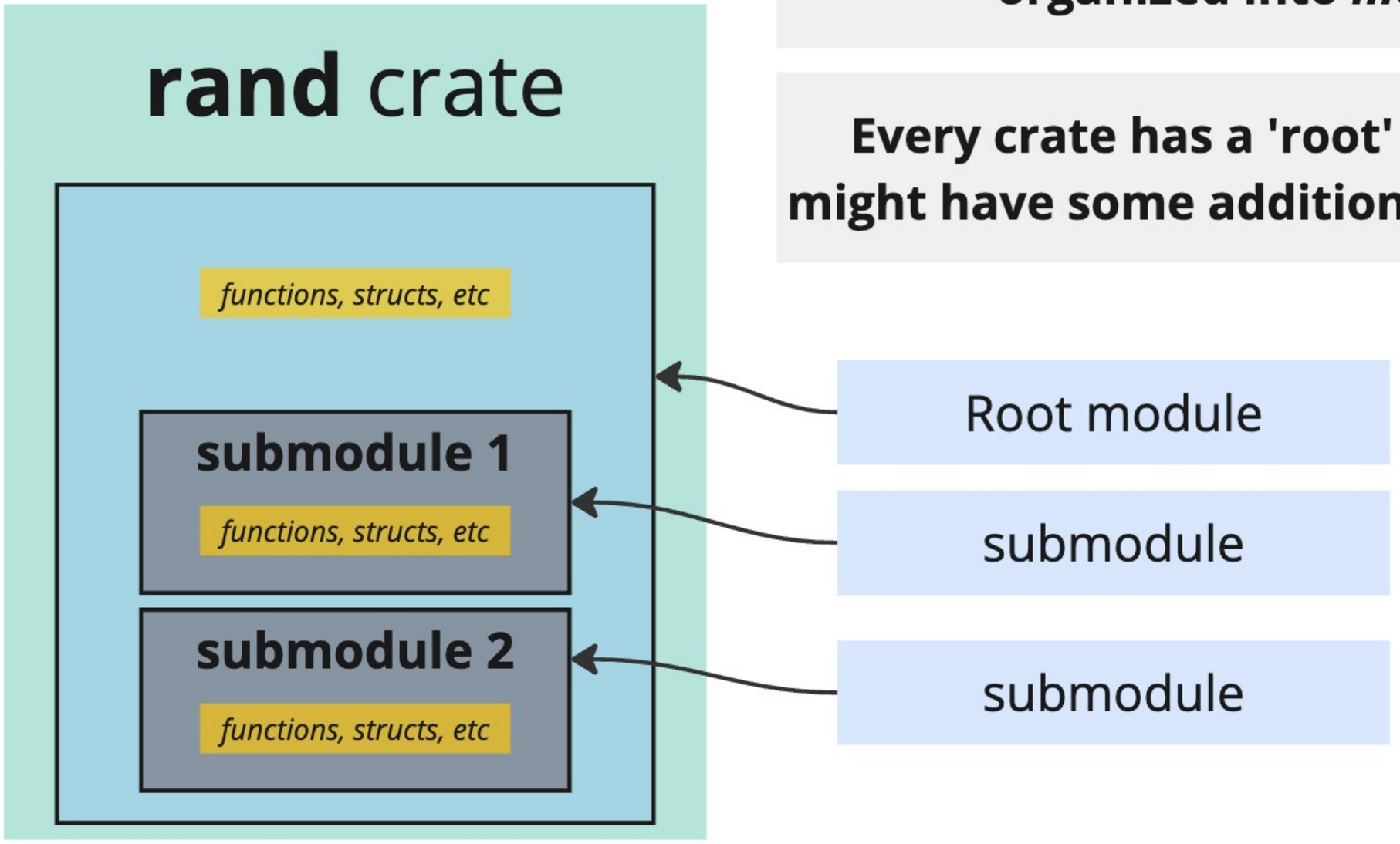
Methods

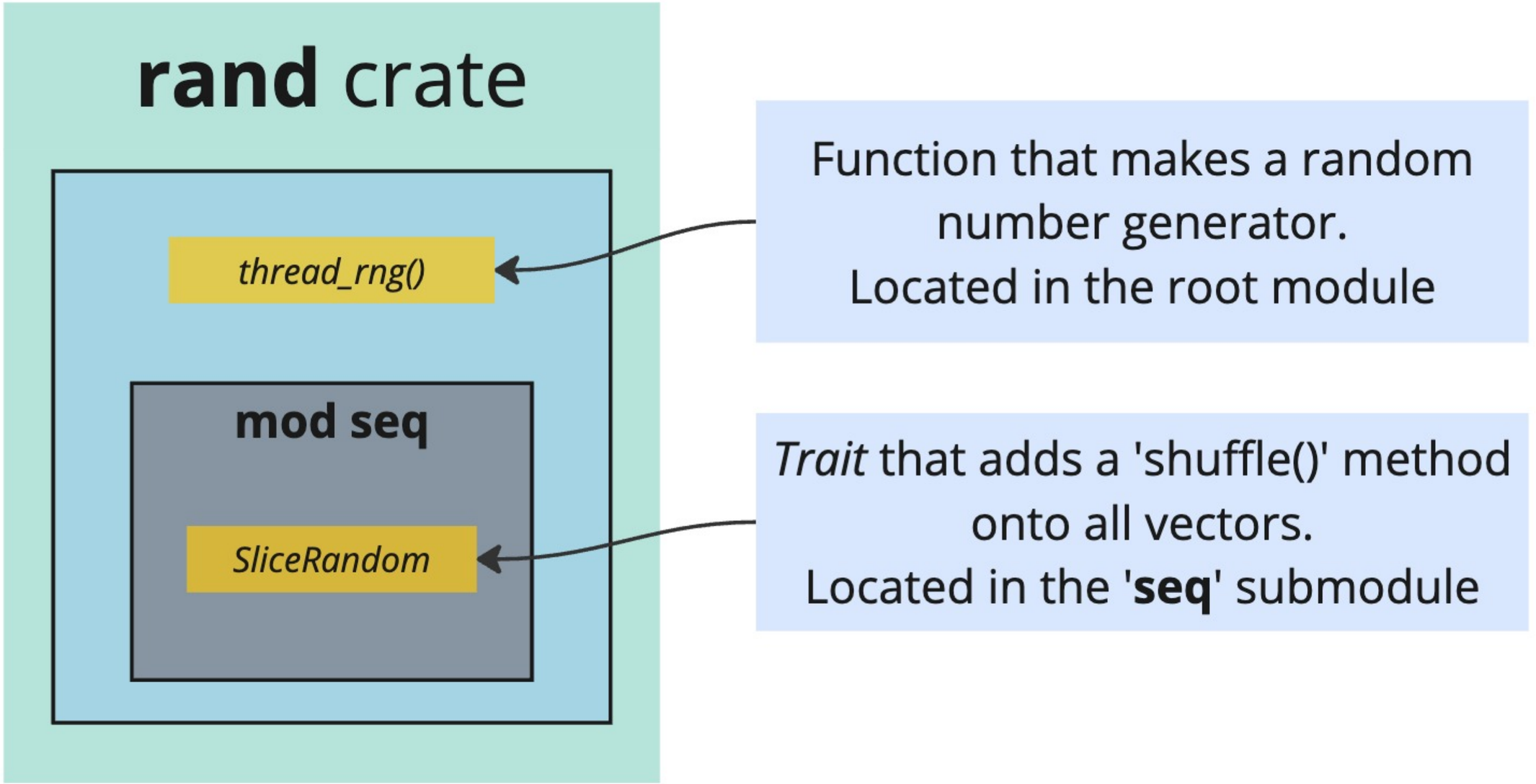
Use when you need to read or change fields on a specific instance

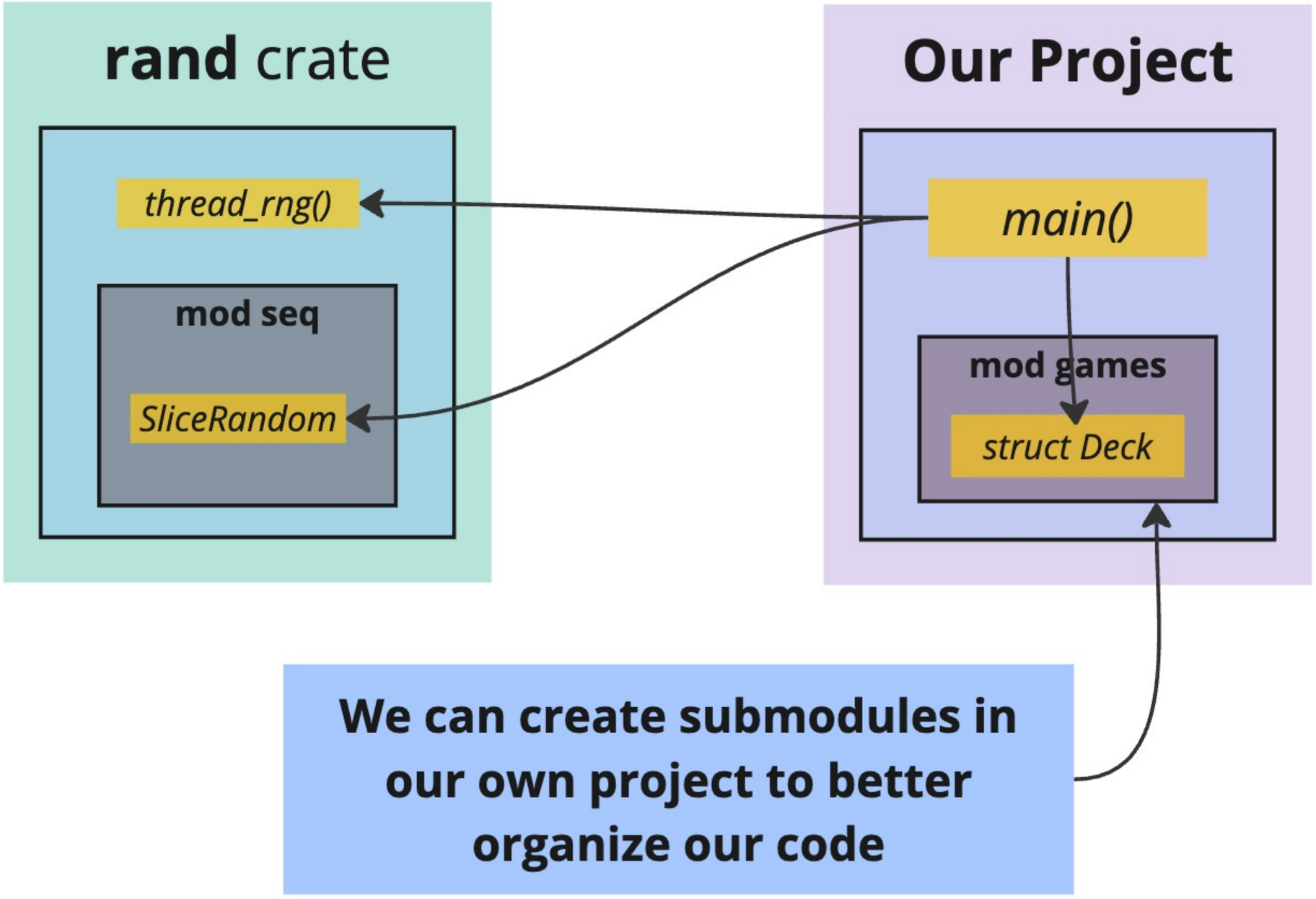
Examples: shuffling cards, adding a card, removing a card, checking if a card exists

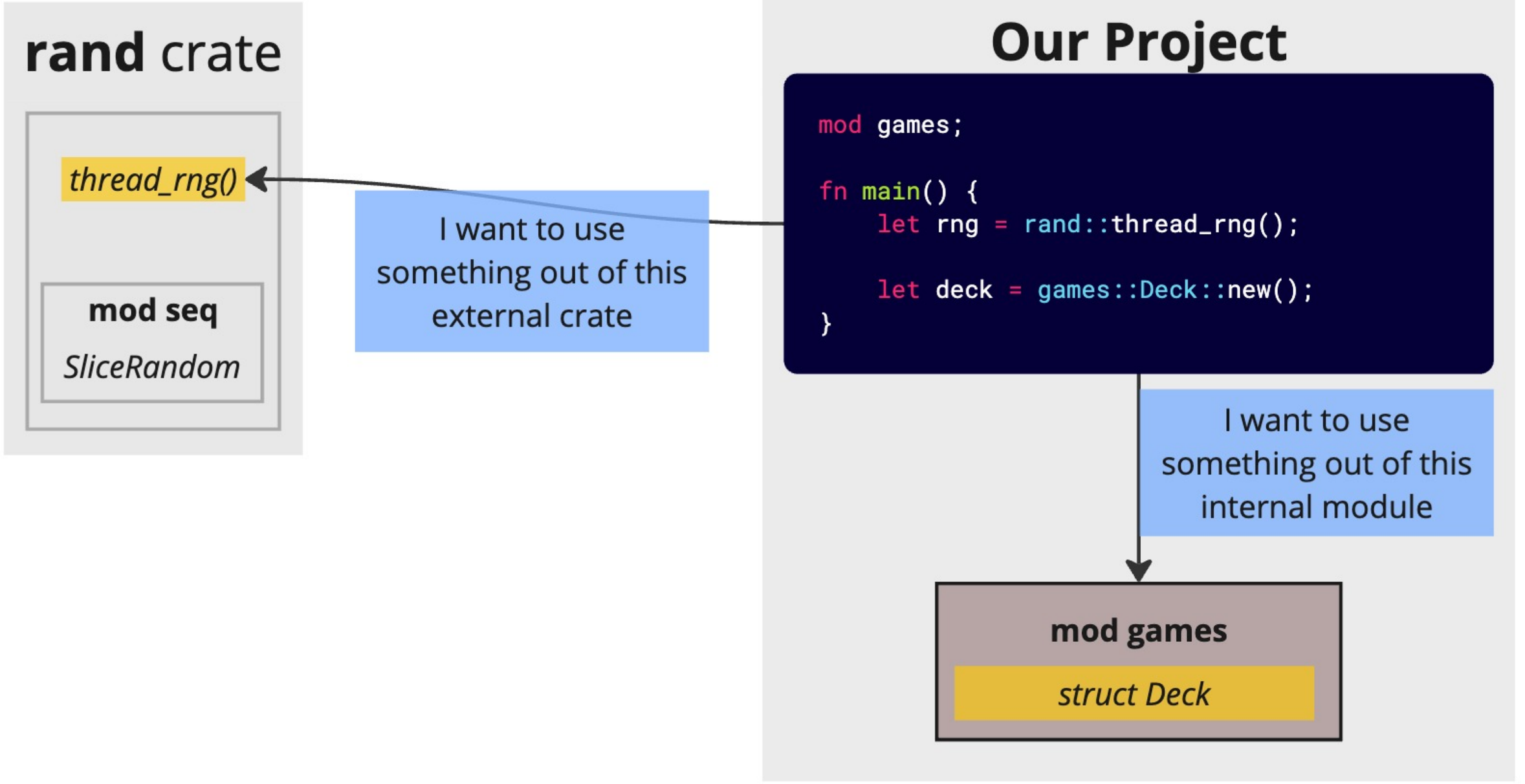
Code in all crates + programs is organized into *modules*

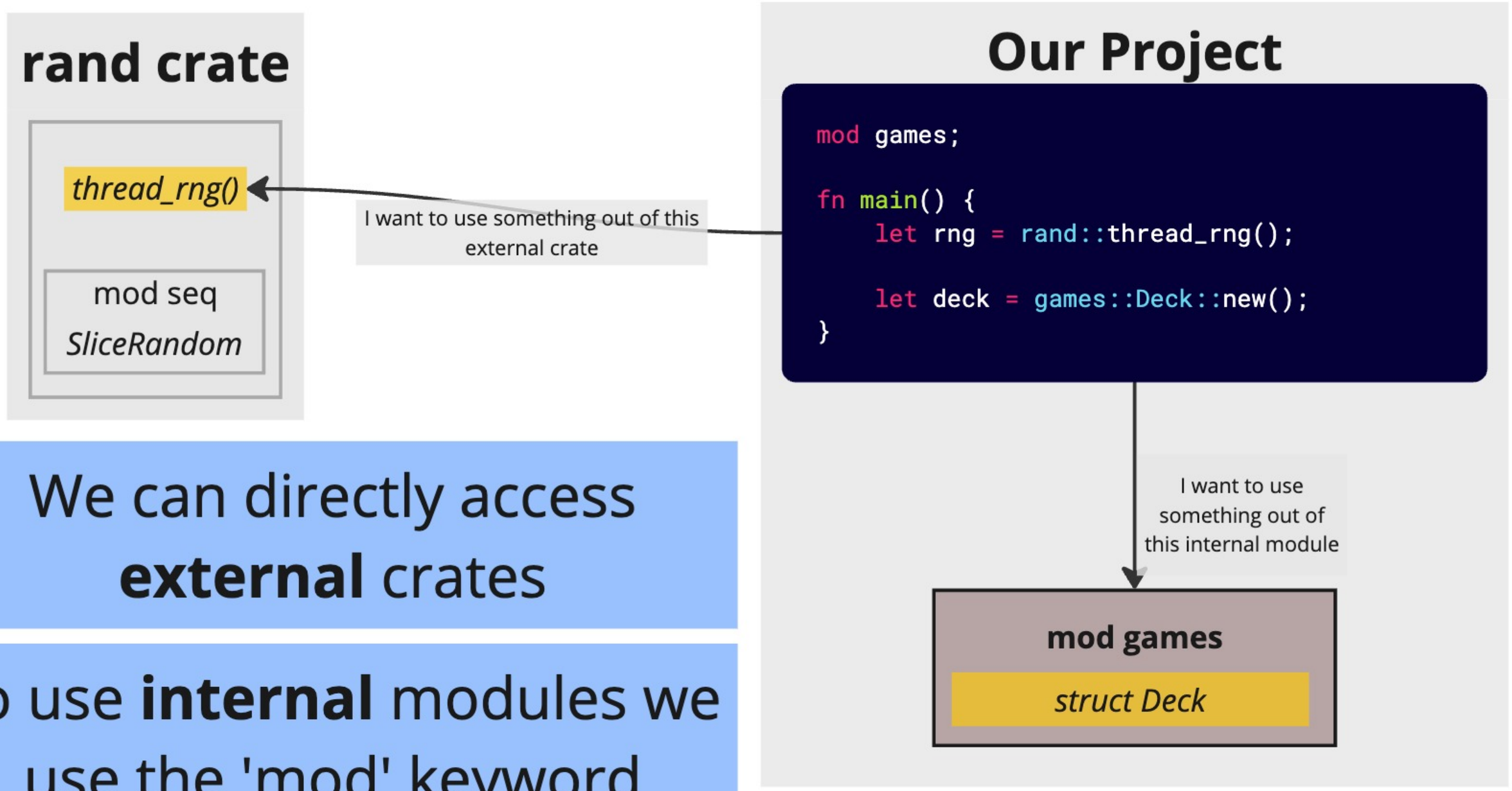
Every crate has a 'root' module and might have some additional submodules




rand crate
thread_rng()
mod seq
SliceRandom
Function that makes a random number generator.
Located in the root module
Trait that adds a 'shuffle()' method onto all vectors.
Located in the 'seq' submodule


rand crate
thread_rng()
mod seq
SliceRandom
Our Project
main()
mod games
struct Deck
We can create submodules in our own project to better organize our code


rand crate
thread_rng()
mod seq
SliceRandom
I want to use something out of this external crate
Our Project
mod games;
fn main() {
let rng = rand::thread_rng();
let deck = games::Deck::new();
}
I want to use something out of this internal module
mod games
struct Deck



We can directly access **external** crates

To use **internal** modules we use the 'mod' keyword

rand crate

thread_rng()

random()

mod seq

SliceRandom

mod rngs

OsRng

```
fn main() {
    let rng = rand::thread_rng();
    let rand_number = rand::random();
    let rand_u64 = rand::rngs::OsRng.next_u64();
}
```

Gets tedious to write out 'rand::xyz' repeatedly

rand crate

thread_rng()

random()

mod seq

SliceRandom

mod rngs

OsRng

```
use rand::{thread_rng, random, rngs::OsRng};

fn main() {
    let rng = thread_rng();
    let rand_number = random();
    let rand_u64 = OsRng.next_u64();
}
```

'use' pulls specific things into the scope of this file

```
impl Deck {
    fn shuffle(&self) {

    }
}
```

Give me a **read only** reference to the deck

```
impl Deck {
    fn shuffle(&mut self) {

    }
}
```

Give me a **read/write** reference to the deck

Deck Object

Represents a collection of playing cards

Functionality I want this deck to have

new() →	Creates a new 'deck' object that contains a list of playing cards
shuffle() →	Shuffles the order of cards in this deck
deal(5) →	Removes some playing cards from this deck and returns them in a new Vector

Crate == Package

Rust Standard Library

- Included with every project without any additional install
- Docs at ***doc.rust-lang.org/std***

External Crates

- Have to be installed into our project with *cargo add rand*
- Crate listing at ***crates.io***
- Docs also at ***docs.rs***

'Exclusive' range

```
0..4 // 0, 1, 2, 3
```

'Inclusive' range

```
0..=4 // 0, 1, 2, 3, 4
```

```rust
#[derive(Debug, PartialEq)]
enum DeckState {
    Initialized,
    Shuffled,
}

fn main() {
    let deck = Deck::new();

    if deck.state == DeckState::Initialized {
        println!("Deck not yet shuffled");
    } else if deck.state == DeckState::Shuffled {
        println!("Deck has been shuffled!");
    }
}
```

Enums

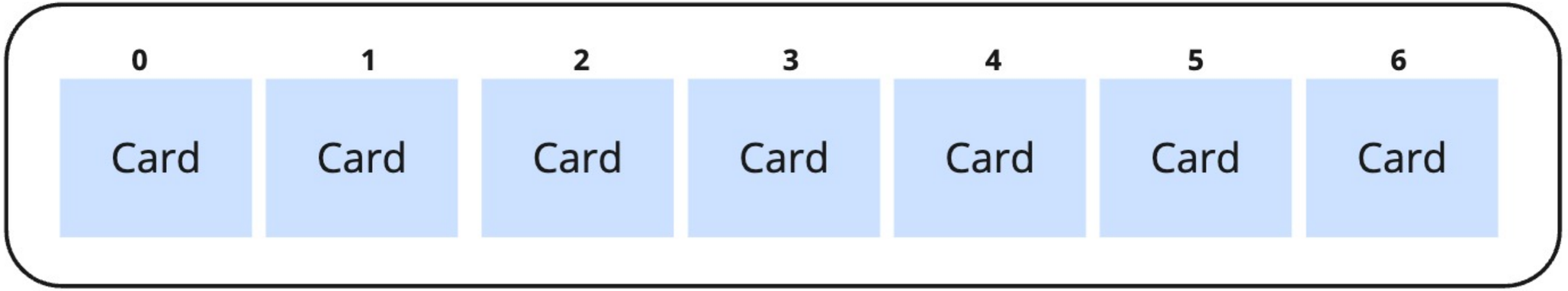
A set of values that are related together in some way

A set of values that are related together in some way

Types of Numbers

Types	Description	Range
i8	Positive or negative integers	-128 to 127
i16	Positive or negative integers	-32,768 to 32,767
i32	Positive or negative integers	2,147,483,648 to 2,147,483,647
i64	Positive or negative integers	-9.2E18 to 9.2E18
i128	Positive or negative integers	-1.7E38 to 1.7E38
isize	Positive or negative integers	-9.2E18 to 9.2E18
u8	Unsigned integers	0 to 255
u16	Unsigned integers	0 to 65,535
u32	Unsigned integers	0 to 4.29E9
u64	Unsigned integers	0 to 1.84E19
u128	Unsigned integers	0 to 3.4E38
usize	Unsigned integers	0 to 1.84E19
f32	Decimal values	-3.4E38 to 3.4E38
f64	Decimal values	-1.7E308 to 1.7E308

Vector




```
self.cards.split_off(
    self.cards.len() - num_cards
)
```

```
use rand::{seq::SliceRandom, thread_rng};
```

'use' Statements

Import code from crates (packages) or other files in your project

You can import multiple things on a single line using curly braces

Attributes

```
#[derive(Debug)]
struct Deck {
    cards: Vec<String>,
}
```

Provide extra instructions to the compiler

'derive' is an attribute. Tells the compiler to add additional code to the struct

'Debug' is a trait. Has functions included that aid in debugging (like printing a struct)

Structs

```
struct Deck {
    cards: Vec<String>,
}
```

Defines a collection of fields (data) that are related in some way

Can be used to tie together data + functionality if we add an 'impl' block

```rust
impl Deck {
    fn new() -> Self {
        // code
    }

    fn shuffle(&self) {
        // code
    }
}
```

Inherent Implementations

Defines methods + associated functions tied to a struct

First argument determines whether we are making a method or an associated function

'**main**' is called automatically when our program starts

'**let mut**' used when we want to make a variable that can be reassigned or when we want to be able to change the value

Error handling is critical, we will spend a lot of time investigating it!

'associated functions' called using the '::' syntax

```
fn main() {
    let mut deck = Deck::new();

    deck.shuffle();
    // Probably need to add error handling!!!!
    let cards = deck.deal(3);

    println!("Heres your hand: {:#?}", cards);
    println!("Heres your deck: {:#?}", deck);
}
```

{:#?} is a formatter. Prints a struct in a human-readable way