Lecture 2: Traits

Methods, traits, functional programming and standard library

Willem Vanhulle

DevLab Rust 2025

Tuesday November 11, 2025

github.com/wvhulle/rust-course-ghent

1.1. Disclaimer

- These are difficult questions. Just try your best!
- Will not be graded.

1.2. Loop

1.2.1. Question

```
struct S;

fn main() {
    let [x, y] = &mut [S, S];

    let eq = x as *mut S == y as *mut S;
    print!("{}", eq as u8);
}
```

(playground link)

What does this program print?

1.2. Loop

1.2.1. Question

```rust
struct S;

fn main() {
    let [x, y] = &mut [S, S];

    let eq = x as *mut S == y as *mut S;
    print!("{}", eq as u8);
}
```

(playground link)

What does this program print?

…

1.2. Loop

1.2.2. Explanation

What are zero-sized types?

1.2. Loop

1.2.2. Explanation

What are zero-sized types?

Types that occupy no space at runtime.

```rust
struct S;
fn main() {
    let [x, y] = &mut [S, S];
    let eq = x as *mut S == y as *mut S;
    print!("{}", eq as u8);
}
```

(playground link)

1.2. Loop

1.2.2. Explanation

What are zero-sized types?

Types that occupy no space at runtime.

```rust
struct S;
fn main() {
    let [x, y] = &mut [S, S];
    let eq = x as *mut S == y as *mut S;
    print!("{}", eq as u8);
}
```
(playground link)

Desugaring pattern match →

```rust
fn main() {
    let tmp: [S; 2] = [S, S];
    let x = &mut tmp[0];
    let y = &mut tmp[1];
    let eq = x as *mut S == y as *mut S;
    print!("{}", eq as u8);
}
```
(playground link)

1.2. Loop

1.2.2. Explanation

What are zero-sized types?

Types that occupy no space at runtime.

```rust
struct S;
fn main() {
    let [x, y] = &mut [S, S];
    let eq = x as *mut S == y as *mut S;
    print!("{}", eq as u8);
}
```
(playground link)

Desugaring pattern match

```rust
fn main() {
    let tmp: [S; 2] = [S, S];
    let x = &mut tmp[0];
    let y = &mut tmp[1];
    let eq = x as *mut S == y as *mut S;
    print!("{}", eq as u8);
}
```
(playground link)

Operator precedence

```rust
fn main() {
    let tmp: [S; 2] = [S, S];
    let x = &mut tmp[0];
    let y = &mut tmp[1];
    let eq = (x as *mut S) == (y as *mut S);
    print!("{}", eq as u8);
}
```
(playground link)

1.2. Loop

1.2.2. Explanation

What are zero-sized types?

Types that occupy no space at runtime.

```rust
struct S;
fn main() {
    let [x, y] = &mut [S, S];
    let eq = x as *mut S == y as *mut S;
    print!("{}", eq as u8);
}
```
(playground link)

Desugaring pattern match

```rust
fn main() {
    let tmp: [S; 2] = [S, S];
    let x = &mut tmp[0];
    let y = &mut tmp[1];
    let eq = x as *mut S == y as *mut S;
    print!("{}", eq as u8);
}
```
(playground link)

Operator precedence

```rust
fn main() {
    let tmp: [S; 2] = [S, S];
    let x = &mut tmp[0];
    let y = &mut tmp[1];
    let eq = (x as *mut S) == (y as *mut S);
    print!("{}", eq as u8);
}
```
(playground link)

Explicit casts

```rust
fn main() {
    let tmp: [S; 2] = [S, S];
    let x = &mut tmp[0];
    let y = &mut tmp[1];
    let x_ptr: *mut S = x;
    let y_ptr: *mut S = y;
    let eq = x_ptr == y_ptr;
    print!("{}", eq as u8);
}
```
(playground link)

Are `x_ptr` and `y_ptr` equal?

1.2. Loop

1.2.2. Explanation

What are zero-sized types?

Types that occupy no space at runtime.

```rust
struct S;
fn main() {
    let [x, y] = &mut [S, S];
    let eq = x as *mut S == y as *mut S;
    print!("{}", eq as u8);
}
```

(playground link)

Desugaring pattern match

```rust
fn main() {
    let tmp: [S; 2] = [S, S];
    let x = &mut tmp[0];
    let y = &mut tmp[1];
    let eq = x as *mut S == y as *mut S;
    print!("{}", eq as u8);
}
```

(playground link)

Operator precedence

```rust
fn main() {
    let tmp: [S; 2] = [S, S];
    let x = &mut tmp[0];
    let y = &mut tmp[1];
    let eq = (x as *mut S) == (y as *mut S);
    print!("{}", eq as u8);
}
```

(playground link)

Explicit casts

```rust
fn main() {
    let tmp: [S; 2] = [S, S];
    let x = &mut tmp[0];
    let y = &mut tmp[1];
    let x_ptr: *mut S = x;
    let y_ptr: *mut S = y;
    let eq = x_ptr == y_ptr;
    print!("{}", eq as u8);
}
```

(playground link)

Are `x_ptr` and `y_ptr` equal?

Yes, since the compiler can see that `S` is a zero-sized type.

1.3. Patterns

1.3.1. Question

Last session I mentioned that variable assignments are actually patterns.

```
fn main() {
    let (.., x, y) = (0, 1, ..);
    print!("{}", b"066"[y][x]);
}
```

(playground link)

What does this program print?

1.3. Patterns

1.3.1. Question

Last session I mentioned that variable assignments are actually patterns.

```rust
fn main() {
    let (.., x, y) = (0, 1, ..);
    print!("{}", b"066"[y][x]);
}
```

(playground link)

What does this program print?

…

1.3. Patterns

1.3.2. Explanation

```
fn main() {
    let (.., x, y) = (0, 1, ..);
    print!("{}", b"066"[y][x]);
}
```

(playground link)

1.3. Patterns

1.3.2. Explanation

```rust
fn main() {
    let (.., x, y) = (0, 1, ..);
    print!("{}", b"066"[y][x]);
}
```
(playground link)

Desugaring →

```rust
fn main() {
    let x = 1;
    let y: RangeFull = ..;
    print!("{}", b"066"[y][x]);
}
```
(playground link)

1.3. Patterns

1.3.2. Explanation

```rust
fn main() {
    let (.., x, y) = (0, 1, ..);
    print!("{}", b"066"[y][x]);
}
```
(playground link)

Desugaring →

```rust
fn main() {
    let x = 1;
    let y: RangeFull = ..;
    print!("{}", b"066"[y][x]);
}
```
(playground link)

Type inference →

```rust
fn main() {
    let x = 1;
    let y = ..;
    let bytes: &'static [u8; 3] = b"066";
    print!("{}", bytes[..][x]);
}
```
(playground link)

1.3. Patterns

1.3.2. Explanation

```rust
fn main() {
    let (.., x, y) = (0, 1, ..);
    print!("{}", b"066"[y][x]);
}                                  (playground link)
```

Desugaring

```rust
fn main() {
    let x = 1;
    let y: RangeFull = ..;
    print!("{}", b"066"[y][x]);
}                                  (playground link)
```

Type inference

```rust
fn main() {
    let x = 1;
    let y = ..;
    let bytes: &'static [u8; 3] = b"066";
    print!("{}", bytes[..][x]);
}                                  (playground link)
```

Slice indexing

```rust
fn main() {
    let x = 1;
    let bytes = b"066";
    print!("{}", bytes[x]);
}                                  (playground link)
```

What is the decimal representation of the ASCII character ‘6’?

1.3. Patterns

1.3.2. Explanation

```rust
fn main() {
    let (.., x, y) = (0, 1, ..);
    print!("{}", b"066"[y][x]);
}                                   (playground link)
```

Desugaring →

```rust
fn main() {
    let x = 1;
    let y: RangeFull = ..;
    print!("{}", b"066"[y][x]);
}                                   (playground link)
```

Type inference →

```rust
fn main() {
    let x = 1;
    let y = ..;
    let bytes: &'static [u8; 3] = b"066";
    print!("{}", bytes[..][x]);
}                                   (playground link)
```

Slice indexing →

```rust
fn main() {
    let x = 1;
    let bytes = b"066";
    print!("{}", bytes[x]);
}                                   (playground link)
```

What is the decimal representation of the ASCII character '6'?

54

1.4. Bool

1.4.1. Question

```rust
fn check(x: i32) -> bool {
    print!("{}", x);
    false
}

fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x) => {
            print!("3")
        }
        _ => print!("4"),
    }
}
```

(playground link)

What does this program print?

1.4. Bool

1.4.1. Question

```
fn check(x: i32) -> bool {
    print!("{}", x);
    false
}

fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x) => {
            print!("3")
        }
        _ => print!("4"),
    }
}
```

(playground link)

What does this program print?

...

1.4. Bool

1.4.2. Explanation

```rust
fn check(x: i32) -> bool {
    print!("{}", x);
    false
}
```

(playground link)

1.4. Bool

1.4.2. Explanation

```rust
fn check(x: i32) -> bool {
    print!("{}", x);
    false
}
```

(playground link)

→

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x) => {
            print!("3")
        }
        _ => print!("4"),
    }
}
```

(playground link)

1.4. Bool

1.4.2. Explanation

```rust
fn check(x: i32) -> bool {
    print!("{}", x);
    false
}
```
(playground link)

⟶

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x) => {
            print!("3")
        }
        _ => print!("4"),
    }
}
```
(playground link)

→

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x) => {
            print!("3")
        }
        _ => print!("4"),
    }
}
```
(playground link)

1.4. Bool

1.4.2. Explanation

```
fn check(x: i32) -> bool {
    print!("{}", x);
    false
}                                (playground link)
```

```
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x) =>
        {
            print!("3")
        }
        _ => print!("4"),
    }
}                                (playground link)
```

```
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x)
        => {
            print!("3")
        }
        _ => print!("4"),
    }
}                                (playground link)
```

```
fn check(x: i32) -> bool {
    print!("{}", x);
    false
}                                (playground link)
```

1.4. Bool

1.4.2. Explanation

```rust
fn check(x: i32) -> bool {
    print!("{}", x);
    false
}
```
(playground link)

→

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x) =>
        {
            print!("3")
        }
        _ => print!("4"),
    }
}
```
(playground link)

→

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x)
        => {
            print!("3")
        }
        _ => print!("4"),
    }
}
```
(playground link)

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x)
        => {
            print!("3")
        }
        _ => print!("4"),
    }
}
```
(playground link)

```rust
fn check(x: i32) -> bool {
    print!("{}", x);
    false
}
```
(playground link)

1.4. Bool

1.4.2. Explanation

```rust
fn check(x: i32) -> bool {
    print!("{}", x);
    false
}
```
(playground link)

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x) =>
        {
            print!("3")
        }
        _ => print!("4"),
    }
}
```
(playground link)

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x)
        => {
            print!("3")
        }
        _ => print!("4"),
    }
}
```
(playground link)

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x)
        => {
            print!("3")
        }
        _ => print!("4"),
    }
}
```
(playground link)

```rust
fn check(x: i32) -> bool {
    print!("{}", x);
    false
}
```
(playground link)

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x) =>
        {
            print!("3")
        }
        _ => print!("4"),
    }
}
```
(playground link)

What is going on here?

1.4. Bool

1.4.2. Explanation

```rust
fn check(x: i32) -> bool {
    print!("{}", x);
    false
}
```
(playground link)

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x) =>
        {
            print!("3")
        }
        _ => print!("4"),
    }
}
```
(playground link)

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x)
        => {
            print!("3")
        }
        _ => print!("4"),
    }
}
```
(playground link)

```rust
fn check(x: i32) -> bool {
    print!("{}", x);
    false
}
```
(playground link)

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x)
        => {
            print!("3")
        }
        _ => print!("4"),
    }
}
```
(playground link)

```rust
fn main() {
    match (1, 2) {
        (x, _) | (_, x) if check(x) =>
        {
            print!("3")
        }
        _ => print!("4"),
    }
}
```
(playground link)

What is going on here?

the guard is being run multiple times, once per |-separated alternative in the match-arm.

2.1. Irrefutable patterns

Info

An **irrefutable pattern** is a pattern that always matches.

```rust
fn takes_tuple(tuple: (char, i32, bool)) {
    let a = tuple.0;
    let b = tuple.1;
    let c = tuple.2;

    // This does the same thing as above.
    let (a, b, c) = tuple;

    // Ignore the first element, only bind the second and third.
    let (_, b, c) = tuple;

    // Ignore everything but the last element.
    let (.., c) = tuple;
}
```

(playground link)

2.1. Irrefutable patterns

Info

An **irrefutable pattern** is a pattern that always matches.

```rust
fn takes_tuple(tuple: (char, i32, bool)) {
    let a = tuple.0;
    let b = tuple.1;
    let c = tuple.2;

    // This does the same thing as above.
    let (a, b, c) = tuple;

    // Ignore the first element, only bind the second and third.
    let (_, b, c) = tuple;

    // Ignore everything but the last element.
    let (.., c) = tuple;
}
```

(playground link)

What happens when adding or removing an element to the tuple and look at the resulting compiler errors

2.1. Irrefutable patterns

Info

An **irrefutable pattern** is a pattern that always matches.

```rust
fn takes_tuple(tuple: (char, i32, bool)) {
    let a = tuple.0;
    let b = tuple.1;
    let c = tuple.2;

    // This does the same thing as above.
    let (a, b, c) = tuple;

    // Ignore the first element, only bind the second and third.
    let (_, b, c) = tuple;

    // Ignore everything but the last element.
    let (.., c) = tuple;
}
```

(playground link)

What happens when adding or removing an element to the tuple and look at the resulting compiler errors

The destructuring pattern must match the structure of the value exactly.

2.1. Irrefutable patterns

Info

An **irrefutable pattern** is a pattern that always matches.

```rust
fn takes_tuple(tuple: (char, i32, bool)) {
    let a = tuple.0;
    let b = tuple.1;
    let c = tuple.2;

    // This does the same thing as above.
    let (a, b, c) = tuple;

    // Ignore the first element, only bind the
    second and third.
    let (_, b, c) = tuple;

    // Ignore everything but the last element.
    let (.., c) = tuple;
}                                  (playground link)
```

What happens when adding or removing an element to the tuple and look at the resulting compiler errors

The destructuring pattern must match the structure of the value exactly.

Info

A variable binding is actually a pattern itself!

The _ pattern matches anything.

2.1. Irrefutable patterns

2.1.1. Advanced usage of ..

Ignoring middle element:

```rust
fn takes_tuple(tuple: (char, i32, bool, u8)) {
    let (first, .., last) = tuple;
}
```

(playground link)

2.1. Irrefutable patterns

2.1.1. Advanced usage of ..

Ignoring middle element:

```rust
fn takes_tuple(tuple: (char, i32, bool, u8)) {
    let (first, .., last) = tuple;
}
```

(playground link)

Works with arrays as well:

```rust
fn takes_array(array: [u8; 5]) {
  let [first, .., last] = array;
}
```

(playground link)

2.1. Irrefutable patterns

2.1.1. Advanced usage of ..

Ignoring middle element:

```
fn takes_tuple(tuple: (char, i32, bool, u8)) {
    let (first, .., last) = tuple;
}
```

(playground link)

Works with arrays as well:

```
fn takes_array(array: [u8; 5]) {
  let [first, .., last] = array;
}
```

(playground link)

Warning

Pattern matching works on all data types in Rust!

2.2. Matching values

Patterns can also be simple values like characters:

```rust
#[rustfmt::skip]
fn main() {
    let input = 'x';
    match input {
        'q'                     => println!("Quitting"),
        'a' | 's' | 'w' | 'd'   => println!("Moving around"),
        '0'..='9'               => println!("Number input"),
        key if key.is_lowercase() => println!("Lowercase: {key}"),
        _                       => println!("Something else"),
    }
}
```

(playground link)

A variable in the pattern (key in this example) will create a binding that can be used within the match arm.

2.2. Matching values

A **match guard**:

- causes the arm to match only if the condition is true.
- If the condition is false the match will continue checking later cases.

```
#[rustfmt::skip]
fn main() {
    let input = 'a';
    match input {
        key if key.is_uppercase() => println!("Uppercase"),
        key => if input == 'q' { println!("Quitting") },
        _   => println!("Lowercase"),
    }
}
```

(playground link)

What is the bug in this program?

2.2. Matching values

A **match guard**:

- causes the arm to match only if the condition is true.
- If the condition is false the match will continue checking later cases.

```rust
#[rustfmt::skip]
fn main() {
    let input = 'a';
    match input {
        key if key.is_uppercase() => println!("Uppercase"),
        key => if input == 'q' { println!("Quitting") },
        _   => println!("Lowercase"),
    }
}
```

(playground link)

What is the bug in this program?

The second arm will always match, so the third arm is unreachable. Write match guard in front of the second arm: `key if key == 'q' => ...`

2.2. Matching values

Warning

Can't use an existing variable as the condition in a match arm, as it will instead be interpreted as a variable name pattern, which creates a new variable that will shadow the existing one.

```
let expected = 5;
match 123 {
    expected => println!("Expected value is 5, actual is {expected}"),
    _ => println!("Value was something else"),
}
```

(playground link)

What is the output of this program?

2.2. Matching values

Warning

Can't use an existing variable as the condition in a match arm, as it will instead be interpreted as a variable name pattern, which creates a new variable that will shadow the existing one.

```
let expected = 5;
match 123 {
    expected => println!("Expected value is 5, actual is {expected}"),
    _ => println!("Value was something else"),
}
```

(playground link)

What is the output of this program?

Will always print "Expected value is 5, actual is 123".

2.2. Matching values

Use @ to bind to a value while also testing it:

```rust
let expected = 5;
match 123 {
    val @ 1..=10 => println!("Value {val} is between 1 and 10"),
    val @ 11..=20 => println!("Value {val} is between 11 and 20"),
    val => println!("Value {val} is something else"),
}
```

(playground link)

2.3. Structs

Like tuples, structs can also be destructured by matching:

```rust
struct Foo {
    x: (u32, u32),
    y: u32,
}

#[rustfmt::skip]
fn main() {
    let foo = Foo { x: (1, 2), y: 3 };
    match foo {
        Foo { y: 2, x: i }   => println!("y = 2, x = {i:?}"),
        Foo { x: (1, b), y } => println!("x.0 = 1, b = {b}, y = {y}"),
        Foo { y, .. }        => println!("y = {y}, other fields were ignored"),
    }
}
```

(playground link)

Add or remove fields to `Foo` and see what happens!

2.3. Structs

2.3.1. Reference patterns

When matching on a reference, Rust **automatically dereferences** it for you:

```
fn main() {
    let foo = Foo { x: (1, 2), y: 3 };
    match &foo {  // `foo` is turned into a reference
        Foo { y: 2, x: i }   => println!("y = 2, x = {i:?}"),
        Foo { x: (1, b), y } => println!("x.0 = 1, b = {b}, y = {y}"),
        Foo { y, .. }        => println!("y = {y}, other fields were ignored"),
    }
}
```

(playground link)

Info

The pattern `Foo { ... }` works on `&foo` because Rust automatically dereferences. You could also write `&Foo { ... }` explicitly, but it's not required.

2.3. Structs

Two equivalent ways to match on a reference:

```rust
match &foo {
    Foo { y, .. } => println!("y = {y}"),  // Implicit dereference
}

match &foo {
    &Foo { y, .. } => println!("y = {y}"), // Explicit dereference pattern
}
```

(playground link)

Pick the one you like best!

Warning

In the implicit version, bound variables like `y` will be references. In the explicit version with `&Foo`, `y` will be a value (copied from the struct).

2.3.2. Mutable reference patterns

When matching on &mut, bound variables are mutable references:

```
struct Foo {
    x: (u32, u32),
    y: u32,
}
fn main() {
    let mut foo = Foo { x: (1, 2), y: 3 };
    match &mut foo {
        Foo { x: (1, 2), y } => *y = 4,  // y is &mut u32, need * to assign
        Foo { y, .. }        => println!("y = {y}"),
    }
}
```

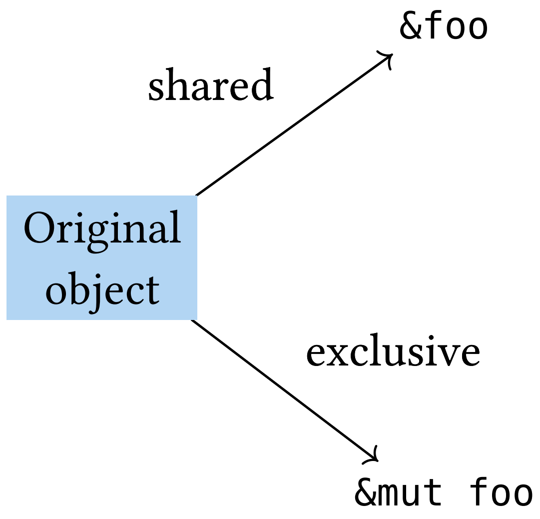
(playground link)

Warning

Bound variables like y are &mut u32, not u32. You must dereference with * to assign values.

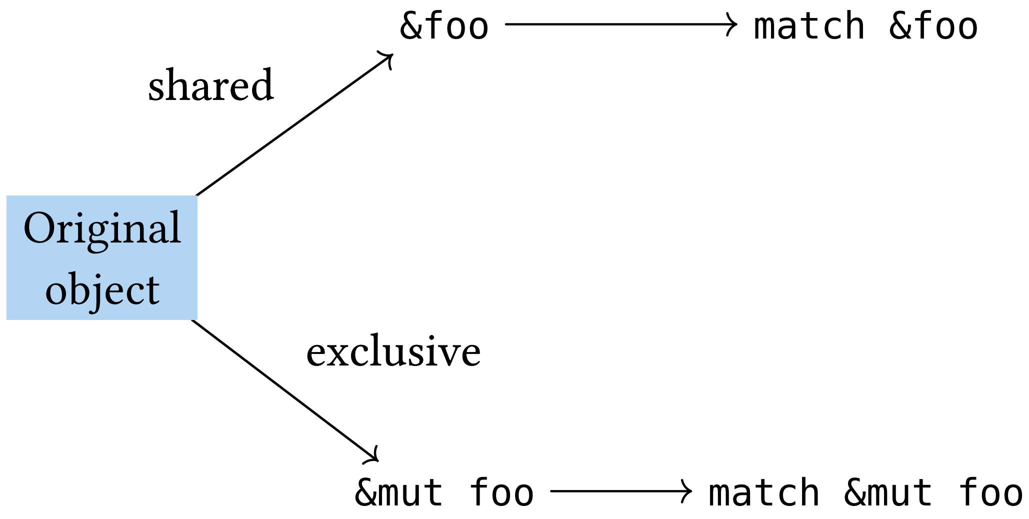
2.3. Structs

2.3.3. Intuition



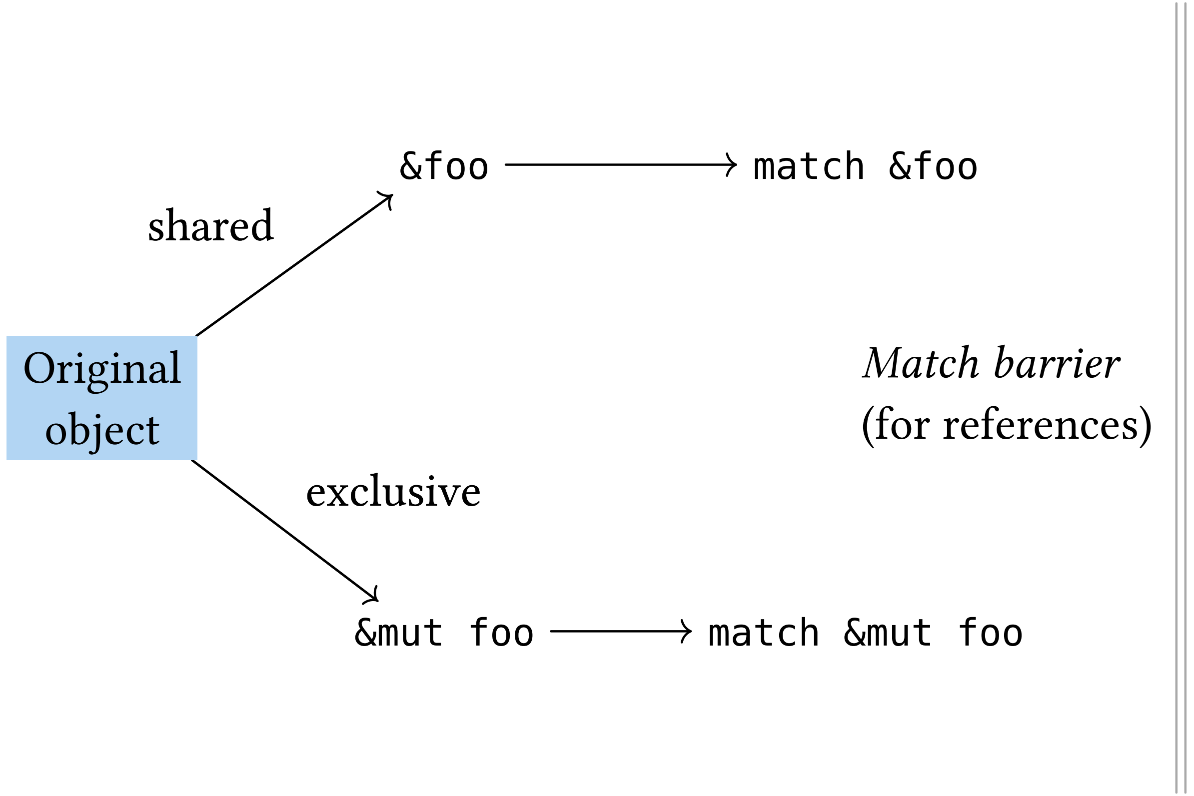
2.3. Structs

2.3.3. Intuition



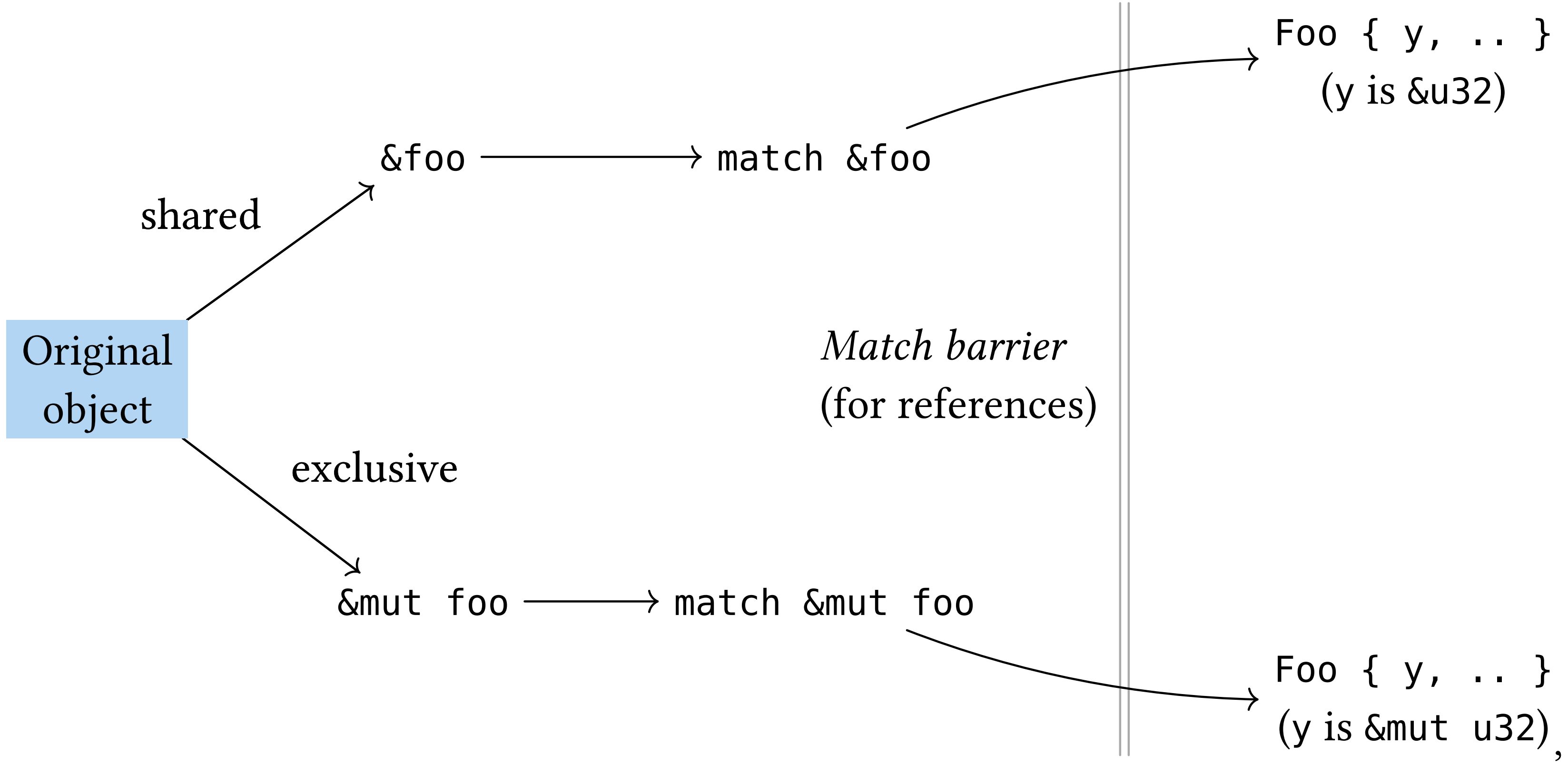
2.3. Structs

2.3.3. Intuition



2.3. Structs

2.3.3. Intuition



2.4. Enums

Like tuples, enums can also be destructured by matching:

```
enum Result {
    Ok(i32),
    Err(String),
}

fn divide_in_two(n: i32) -> Result {
    if n % 2 == 0 {
        Result::Ok(n / 2)
    } else {
        Result::Err(format!("cannot divide {n} into two equal parts"))
    }
}
```

(playground link)

2.4. Enums

Like tuples, enums can also be destructured by matching:

```rust
enum Result {
    Ok(i32),
    Err(String),
}

fn divide_in_two(n: i32) -> Result {
    if n % 2 == 0 {
        Result::Ok(n / 2)
    } else {
        Result::Err(format!("cannot divide {n} into two equal parts"))
    }
}
```

(playground link)

```rust
fn main() {
    let n = 100;
    match divide_in_two(n) {
        Result::Ok(half) => println!("{n} divided in two is {half}"),
        Result::Err(msg) => println!("sorry, an error happened: {msg}"),
    }
}
```

(playground link)

Warning

Rust does not allow non-exhaustive matches

2.5. Let control flow

```rust
use std::time::Duration;

fn sleep_for(secs: f32) {
    let result = Duration::try_from_secs_f32(secs);

    if let Ok(duration) = result {
        std::thread::sleep(duration);
        println!("slept for {duration:?}");
    }
}

fn main() {
    sleep_for(-10.0);
    sleep_for(0.8);
}
```

(playground link)

When to use `if let` over `match`?

2.5. Let control flow

```rust
use std::time::Duration;

fn sleep_for(secs: f32) {
    let result = Duration::try_from_secs_f32(secs);

    if let Ok(duration) = result {
        std::thread::sleep(duration);
        println!("slept for {duration:?}");
    }
}

fn main() {
    sleep_for(-10.0);
    sleep_for(0.8);
}
```

(playground link)

When to use `if let` over `match`?

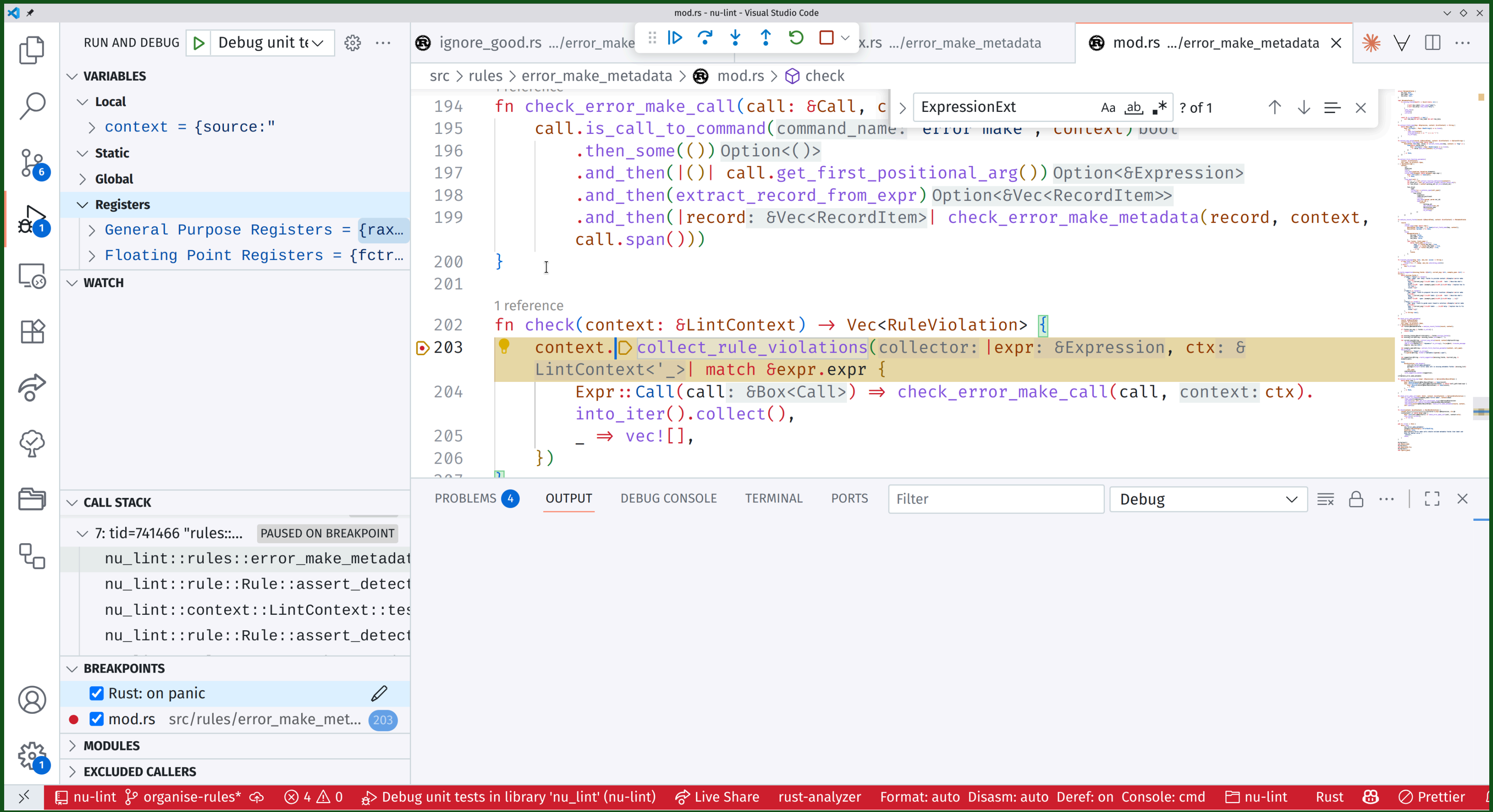
Use `if let` when you only care about one specific pattern and want to ignore all others.

2.6. Exercise: Expression evaluation

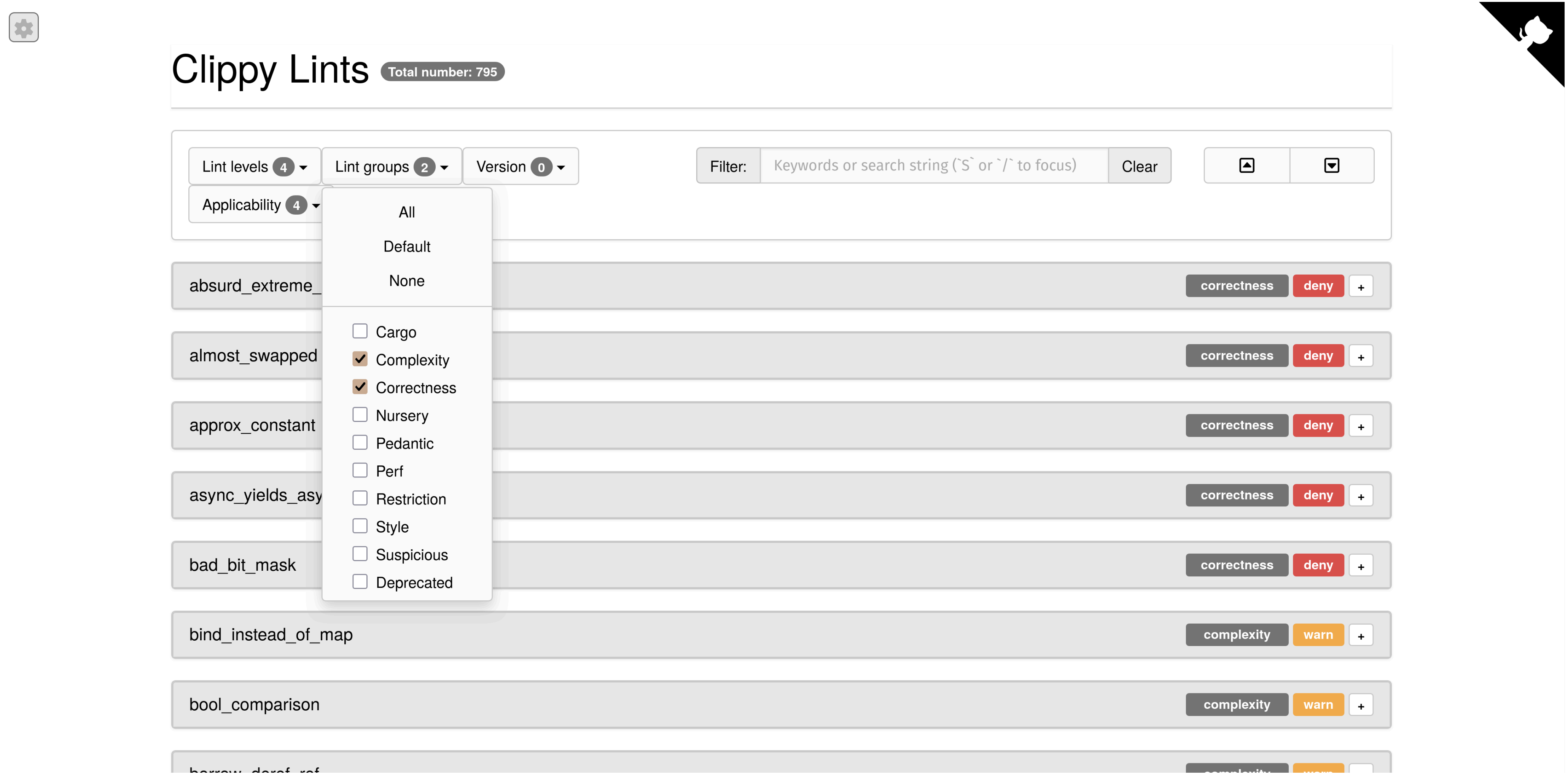
Let's write a simple recursive evaluator for arithmetic expressions.

Complete the `eval` function in the file `session-2/tests/s2e2-evaluation.rs`.

Run tests with `cargo test --test s2e2-evaluation`.


mod.rs - nu-lint - Visual Studio Code
RUN AND DEBUG
Debug unit te
ignore_good.rs .../error_make
x.rs .../error_make_metadata
mod.rs .../error_make_metadata
src > rules > error_make_metadata > mod.rs > check
VARIABLES
Local
context = {source:"
Static
Global
Registers
General Purpose Registers = {rax…
Floating Point Registers = {fctr…
WATCH
ExpressionExt
? of 1
194 fn check_error_make_call(call: &Call, c
195 call.is_call_to_command(command_name:
196 .then_some(()) Option<()>
197 .and_then(|()| call.get_first_positional_arg()) Option<&Expression>
198 .and_then(extract_record_from_expr) Option<&Vec<RecordItem>>
199 .and_then(|record: &Vec<RecordItem>| check_error_make_metadata(record, context,
call.span()))
200 }
201
1 reference
202 fn check(context: &LintContext) → Vec<RuleViolation> {
203 context.collect_rule_violations(collector: |expr: &Expression, ctx: &
LintContext<'_>| match &expr.expr {
204 Expr::Call(call: &Box<Call>) ⇒ check_error_make_call(call, context: ctx).
into_iter().collect(),
205 _ ⇒ vec![],
206 })
CALL STACK
7: tid=741466 "rules::...
PAUSED ON BREAKPOINT
nu_lint::rules::error_make_metadat
nu_lint::rule::Rule::assert_detect
nu_lint::context::LintContext::tes
nu_lint::rule::Rule::assert_detect
BREAKPOINTS
Rust: on panic
mod.rs src/rules/error_make_met... 203
MODULES
EXCLUDED CALLERS
PROBLEMS 4
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Filter
Debug
nu-lint organise-rules* 4 0 Debug unit tests in library 'nu_lint' (nu-lint) Live Share rust-analyzer Format: auto Disasm: auto Deref: on Console: cmd nu-lint Rust Prettier

3.2. Clippy


Clippy Lints
Total number: 795
Lint levels 4
Lint groups 2
Version 0
Applicability 4
Filter:
Keywords or search string (`S` or `/` to focus)
Clear
All
Default
None
Cargo
Complexity
Correctness
Nursery
Pedantic
Perf
Restriction
Style
Suspicious
Deprecated
absurd_extreme_
almost_swapped
approx_constant
async_yields_asy
bad_bit_mask
bind_instead_of_map
bool_comparison
correctness
deny
complexity
warn

3.2. Clippy

Place clippy rules in your `Cargo.toml` like this:

```toml
[lints.clippy]
complexity = { level = "deny", priority = -1 }
pedantic = { level = "deny", priority = -1 }
style = { level = "deny", priority = -1 }

absolute_paths = "deny"
allow_attributes_without_reason = "deny"
...
```

(playground link)

Find more lint rules at rust-lang.github.io/rust-clippy.

3.3. Summary

Typical development workflow in Rust:

1. Debug with GDB / LLDB or VS Code
2. Format with `cargo fmt`
3. Lint with `cargo clippy`
4. Test with `cargo test`
5. Build with `cargo build` or `cargo build --release`

4.1. Methods

In Rust, methods are separated from fields:

```rust
#[derive(Debug)]
struct CarRace {
    name: String,
    laps: Vec<i32>,
}

impl CarRace {
    // No receiver, a static method
    fn new(name: &str) -> Self {
        Self { name: String::from(name), laps: Vec::new() }
    }
    // Exclusive borrowed read-write access to self
    fn add_lap(&mut self, lap: i32) {
        self.laps.push(lap);
    }
}

fn main() {
    let mut race = CarRace::new("Monaco Grand Prix");
    race.add_lap(70);
}
```

(playground link)

4.1. Methods

Like in other languages, methods group functionality around *acting* data types, types that *do* related things.

How are methods in Rust different from methods in other languages?

4.1. Methods

Like in other languages, methods group functionality around *acting* data types, types that *do* related things.

How are methods in Rust different from methods in other languages?

Methods cannot be overridden.

```rust
impl CarRace {
    fn add_lap(&mut self, lap: i32) {
        self.laps.push(lap);
    }
}
```

(playground link)

What is the `&mut self` parameter syntax sugar for?

4.1. Methods

Like in other languages, methods group functionality around *acting* data types, types that *do* related things.

How are methods in Rust different from methods in other languages?

Methods cannot be overridden.

```rust
impl CarRace {
    fn add_lap(&mut self, lap: i32) {
        self.laps.push(lap);
    }
}
```
(playground link)

What is the `&mut self` parameter syntax sugar for?

For `self: &mut Self`.

```rust
impl CarRace {
    fn add_lap(self: &mut Self, lap: i32) {
        self.laps.push(lap);
    }
}
```
(playground link)

4.1. Methods

Methods can **consume** `self`:

```rust
#[derive(Debug)]
struct CarRace {
    name: String,
    laps: Vec<i32>,
}

impl CarRace {
    // Exclusive ownership of self (covered later)
    fn finish(self) {
        let total: i32 = self.laps.iter().sum();
        println!("Race {} is finished, total lap time:
        {}", self.name, total);
    }
}

fn main() {
    let mut race = CarRace::new("Monaco Grand Prix");
    race.add_lap(70);
    race.finish();
    // race.add_lap(42);
}
```

(playground link)

4.1. Methods

Methods can **consume** `self`:

```rust
#[derive(Debug)]
struct CarRace {
    name: String,
    laps: Vec<i32>,
}

impl CarRace {
    // Exclusive ownership of self (covered later)
    fn finish(self) {
        let total: i32 = self.laps.iter().sum();
        println!("Race {} is finished, total lap time: {}", self.name, total);
    }
}

fn main() {
    let mut race = CarRace::new("Monaco Grand Prix");
    race.add_lap(70);
    race.finish();
    // race.add_lap(42);
}
```

(playground link)

What is the name given to the `self` parameter in methods?

4.1. Methods

Methods can **consume** `self`:

```rust
#[derive(Debug)]
struct CarRace {
    name: String,
    laps: Vec<i32>,
}

impl CarRace {
    // Exclusive ownership of self (covered later)
    fn finish(self) {
        let total: i32 = self.laps.iter().sum();
        println!("Race {} is finished, total lap time: {}", self.name, total);
    }
}

fn main() {
    let mut race = CarRace::new("Monaco Grand Prix");
    race.add_lap(70);
    race.finish();
    // race.add_lap(42);
}
```

(playground link)

What is the name given to the `self` parameter in methods?

`self` is called the **receiver**.

After calling a method with `object.finish(self)`, you can no longer use `object`. It has been *consumed*.

Warning

Use the `self` receiver to define destructors or functionality that should happen only once at the end.

4.2. Receivers

Rust allows receivers to be one of the following:

- `self`, consuming self
- any kind of reference to self: `&self`, `&mut self`
- exceptions (covered later)

Name a few reference types in Rust.

4.2. Receivers

Rust allows receivers to be one of the following:

- `self`, consuming self
- any kind of reference to self: `&self`, `&mut self`
- exceptions (covered later)

Name a few reference types in Rust.

`&T`, `&mut T`, `Box<T>`, `Rc<T>`, `Arc<T>`, …

Info

Reference types are also called **wrappers** and some are **smart pointers**.

4.3. Traits

Rust lets you abstract over types with traits. Similar to interfaces:

```rust
trait Pet {
    /// Return a sentence from this pet.
    fn talk(&self) -> String;

    /// Print a string to the terminal greeting this pet.
    fn greet(&self);
}
```

(playground link)

Properties of traits:

4.3. Traits

Rust lets you abstract over types with traits. Similar to interfaces:

```rust
trait Pet {
    /// Return a sentence from this pet.
    fn talk(&self) -> String;

    /// Print a string to the terminal greeting this pet.
    fn greet(&self);
}
```

(playground link)

Properties of traits:

- A trait is a list of methods that a type **must implement**

4.3. Traits

Rust lets you abstract over types with traits. Similar to interfaces:

```rust
trait Pet {
    /// Return a sentence from this pet.
    fn talk(&self) -> String;

    /// Print a string to the terminal greeting this pet.
    fn greet(&self);
}
```

(playground link)

Properties of traits:

- A trait is a list of methods that a type **must implement**
- The **signatures** of the type's methods **must be identical** to the trait's method signatures

4.3. Traits

Rust lets you abstract over types with traits. Similar to interfaces:

```rust
trait Pet {
    /// Return a sentence from this pet.
    fn talk(&self) -> String;

    /// Print a string to the terminal greeting this pet.
    fn greet(&self);
}
```

(playground link)

Properties of traits:

- A trait is a list of methods that a type **must implement**
- The **signatures** of the type's methods **must be identical** to the trait's method signatures

Info

Default implementations can be provided

4.4. Implementing traits

```rust
trait Pet {
  fn talk(&self) -> String;

  fn greet(&self) {
      println!("Oh you're a cutie! What's your name? {}", self.talk());
  }
}

struct Dog {
  name: String,
  age: i8,
}
```

(playground link)

4.4. Implementing traits


```rust
trait Pet {
    fn talk(&self) -> String;

    fn greet(&self) {
        println!("Oh you're a cutie! What's your name? {}", self.talk());
    }
}

struct Dog {
    name: String,
    age: i8,
}
```

(playground link)

```rust
impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Woof, my name is {}!", self.name)
    }
}

fn main() {
    let fido = Dog { name: String::from("Fido"), age: 5 };
    dbg!(fido.talk());
    fido.greet();
}
```

(playground link)

4.4. Implementing traits

```rust
trait Pet {
  fn talk(&self) -> String;

  fn greet(&self) {
      println!("Oh you're a cutie!
      What's your name? {}",
      self.talk());
  }
}

struct Dog {
  name: String,
  age: i8,
}
```
(playground link)

```rust
impl Pet for Dog {
  fn talk(&self) -> String {
      format!("Woof, my name is {}!", self.name)
  }
}

fn main() {
  let fido = Dog { name: String::from("Fido"),
  age: 5 };
  dbg!(fido.talk());
  fido.greet();
}
```
(playground link)

Warning

a Cat type with a `talk()` method would not automatically satisfy Pet unless it is in an impl Pet block

4.4. Implementing traits

You can:

- split trait implementation blocks
- override default method implementations

```rust
impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Woof, my name is {}!", self.name)
    }
}

impl Pet for Dog {
    fn greet(&self) -> String {
        format!("Woof, my name is {}!", self.name)
    }
}
```

(playground link)

level of pointer indirection each time a new object is added to the mix (and, practically, a heap allocation).

Although there were other reasons as well, this issue of expensive composition is the key thing that drove Rust towards adopting a different model. It is particularly a problem when one considers, for example, the implications of composing together the `Future`s which will eventually make up an asynchronous task (including address-sensitive `async fn` state machines). It is plausible that there could be many layers of `Future`s composed together, including multiple layers of `async fn`s handling different parts of a task. It was deemed unacceptable to force indirection and allocation for each layer of composition in this case.

`Pin<Ptr>` is an implementation of the third option. It allows us to solve the issues discussed with the second option by building a *shared contractual language* around the guarantees of "pinning" data.

Using `Pin<Ptr>` to pin values

In order to pin a value, we wrap a *pointer to that value* (of some type `Ptr`) in a `Pin<Ptr>`. `Pin<Ptr>` can wrap any pointer type, forming a promise that the **pointee** will not be *moved* or otherwise invalidated.

We call such a `Pin`-wrapped pointer a **pinning pointer,** (or pinning reference, or pinning `Box`, etc.) because its existence is the thing that is conceptually pinning the underlying pointee in place: it is the metaphorical "pin" securing the data in place on the pinboard (in memory).

Notice that the thing wrapped by `Pin` is not the value which we want to pin itself, but rather a pointer to that value! A `Pin<Ptr>` does not pin the `Ptr`; instead, it pins the pointer's ***pointee** value*.

Pinning as a library contract

Pinning does not require nor make use of any compiler "magic"[2], only a specific contract between the `unsafe` parts of a library API and its users.

It is important to stress this point as a user of the `unsafe` parts of the `Pin` API. Practically, this means that performing the

4.5. Contracts

Rust, being a systems programming language, focuses on safety.

4.5. Contracts

Rust, being a systems programming language, focuses on safety.

In Rust documentation, you will often encounter the word **contract**.

Info

A contract is an agreement between two parties. There are mainly two kinds of contracts in Rust:

- Contracts between **two pieces of code**
- Contracts between **the programmer and the compiler**

4.5. Contracts

Rust, being a systems programming language, focuses on safety.

In Rust documentation, you will often encounter the word **contract**.

Info

A contract is an agreement between two parties. There are mainly two kinds of contracts in Rust:

- Contracts between **two pieces of code**
- Contracts between **the programmer and the compiler**

Contracts between two pieces of code are generally safe (can be enforced automatically).

4.5. Contracts

Rust, being a systems programming language, focuses on safety.

In Rust documentation, you will often encounter the word **contract**.

Info

A contract is an agreement between two parties. There are mainly two kinds of contracts in Rust:

- Contracts between **two pieces of code**
- Contracts between **the programmer and the compiler**

Contracts between two pieces of code are generally safe (can be enforced automatically).

Warning

… but contracts between the programmer and the compiler can be unsafe or **may have to be checked by the programmer**.

Examples of such contracts: `Pin`, `Send`, `Sync` traits (advanced topic).

4.6. Super-traits

```rust
trait Animal {
    fn leg_count(&self) -> u32;
}

trait Pet: Animal {
    fn name(&self) -> String;
}

struct Dog(String);

impl Animal for Dog {
    fn leg_count(&self) -> u32 {
        4
    }
}

impl Pet for Dog {
    fn name(&self) -> String {
        self.0.clone()
    }
}
```

(playground link)

Super-traits are an extra constraint on traits that say: “to implement this trait, you must also implement that other trait”.

4.6. Super-traits

```rust
trait Animal {
    fn leg_count(&self) -> u32;
}

trait Pet: Animal {
    fn name(&self) -> String;
}

struct Dog(String);

impl Animal for Dog {
    fn leg_count(&self) -> u32 {
        4
    }
}

impl Pet for Dog {
    fn name(&self) -> String {
        self.0.clone()
    }
}
```

(playground link)

Super-traits are an extra constraint on traits that say: "to implement this trait, you must also implement that other trait".

Warning

Super traits are the kind of language feature you should **avoid as long as you are stuck with the OOP mindset**.

Once you are willing to forget OOP, you can see super-traits are actually easy.

4.6. Super-traits


```rust
trait Animal {
    fn leg_count(&self) -> u32;
}

trait Pet: Animal {
    fn name(&self) -> String;
}

struct Dog(String);

impl Animal for Dog {
    fn leg_count(&self) -> u32 {
        4
    }
}

impl Pet for Dog {
    fn name(&self) -> String {
        self.0.clone()
    }
}
```

(playground link)

Super-traits are an extra constraint on traits that say: “to implement this trait, you must also implement that other trait”.

Warning

Super traits are the kind of language feature you should **avoid as long as you are stuck with the OOP mindset**.

Once you are willing to forget OOP, you can see super-traits are actually easy.

Advanced

… at least as long as you don’t constrain **associated types** of super traits in subtraits

4.7. Associated Types

```
#[derive(Debug)]
struct Meters(i32);
#[derive(Debug)]
struct MetersSquared(i32);

trait Multiply {
    type Output;
    fn multiply(&self, other: &Self) -> Self::Output;
}

impl Multiply for Meters {
    type Output = MetersSquared;
    fn multiply(&self, other: &Self) -> Self::Output {
        MetersSquared(self.0 * other.0)
    }
}

fn main() {
    println!("{:?}", Meters(10).multiply(&Meters(20)));
}
```

(playground link)

Associated types are **placeholder types that** are supplied by the trait implementation.

4.7. Associated Types

```rust
#[derive(Debug)]
struct Meters(i32);
#[derive(Debug)]
struct MetersSquared(i32);

trait Multiply {
    type Output;
    fn multiply(&self, other: &Self) -> Self::Output;
}

impl Multiply for Meters {
    type Output = MetersSquared;
    fn multiply(&self, other: &Self) -> Self::Output {
        MetersSquared(self.0 * other.0)
    }
}

fn main() {
    println!("{:?}", Meters(10).multiply(&Meters(20)));
}
```

(playground link)

Associated types are **placeholder types that** are supplied by the trait implementation.

Why are associated types are sometimes also called "output types"

4.7. Associated Types

```rust
#[derive(Debug)]
struct Meters(i32);
#[derive(Debug)]
struct MetersSquared(i32);

trait Multiply {
    type Output;
    fn multiply(&self, other: &Self) -> Self::Output;
}

impl Multiply for Meters {
    type Output = MetersSquared;
    fn multiply(&self, other: &Self) -> Self::Output {
        MetersSquared(self.0 * other.0)
    }
}

fn main() {
    println!("{:?}", Meters(10).multiply(&Meters(20)));
}
```

(playground link)

Associated types are **placeholder types that** are supplied by the trait implementation.

Why are associated types are sometimes also called "output types"

The implementer, not the caller, chooses the concrete associated type. (Compare with generics.)

4.7. Associated Types


```rust
#[derive(Debug)]
struct Meters(i32);
#[derive(Debug)]
struct MetersSquared(i32);

trait Multiply {
    type Output;
    fn multiply(&self, other: &Self) -> Self::Output;
}

impl Multiply for Meters {
    type Output = MetersSquared;
    fn multiply(&self, other: &Self) -> Self::Output {
        MetersSquared(self.0 * other.0)
    }
}

fn main() {
    println!("{:?}", Meters(10).multiply(&Meters(20)));
}
```

(playground link)

Associated types are **placeholder types that** are supplied by the trait implementation.

Why are associated types are sometimes also called “output types”

The implementer, not the caller, chooses the concrete associated type. (Compare with generics.)

Iterators from the standard library have an associated type `Item`:

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) ->
    Option<Self::Item>;
}
```

(playground link)

4.8. Deriving

Supported traits can be automatically implemented for your custom types, as follows:

```rust
#[derive(Debug, Clone, Default)]
struct Player {
    name: String,
    strength: u8,
    hit_points: u8,
}

fn main() {
    let p1 = Player::default(); // Default trait adds `default` constructor.
    let mut p2 = p1.clone(); // Clone trait adds `clone` method.
    p2.name = String::from("EldurScrollz");
    // Debug trait adds support for printing with `{:?}`.
    println!("{p1:?} vs. {p2:?}");
}
```

(playground link)

How is the derive functionality implemented in Rust?

4.8. Deriving

Supported traits can be automatically implemented for your custom types, as follows:

```rust
#[derive(Debug, Clone, Default)]
struct Player {
    name: String,
    strength: u8,
    hit_points: u8,
}

fn main() {
    let p1 = Player::default(); // Default trait adds `default` constructor.
    let mut p2 = p1.clone(); // Clone trait adds `clone` method.
    p2.name = String::from("EldurScrollz");
    // Debug trait adds support for printing with `{:?}`.
    println!("{p1:?} vs. {p2:?}");
}
```

(playground link)

How is the derive functionality implemented in Rust?

With **procedural macros**, and many crates provide useful derive macros to add useful functionality.

4.8. Deriving

Supported traits can be automatically implemented for your custom types, as follows:

```rust
#[derive(Debug, Clone, Default)]
struct Player {
    name: String,
    strength: u8,
    hit_points: u8,
}

fn main() {
    let p1 = Player::default(); // Default trait adds `default` constructor.
    let mut p2 = p1.clone(); // Clone trait adds `clone` method.
    p2.name = String::from("EldurScrollz");
    // Debug trait adds support for printing with `{:?}`.
    println!("{p1:?} vs. {p2:?}");
}
```

(playground link)

How is the derive functionality implemented in Rust?

With **procedural macros**, and many crates provide useful derive macros to add useful functionality.

For example, `serde` can derive serialization support for a struct using `#[derive(Serialize)]`.

4.8. Deriving

4.8.1. Why is deriving useful?

A manual implementation of the `Clone` trait for the `Player` struct would look like this:

```rust
impl Clone for Player {
    fn clone(&self) -> Self {
        Player {
            name: self.name.clone(),
            strength: self.strength.clone(),
            hit_points: self.hit_points.clone(),
        }
    }
}
```

(playground link)

It is easier to just write `#[derive(Clone)]`

Info

The derive attribute is similar to `deriving` in Haskell.

4.9. Exercise: Logger trait

Complete the test code in `session-2/examples/s2e3-logger.rs`.

Run code with `cargo run --example s2e3-logger`.

5.1. Generic functions

```rust
fn pick<T>(cond: bool, left: T, right: T) -> T {
    if cond { left } else { right }
}

fn main() {
    println!("picked a number: {:?}", pick(true, 222, 333));
    println!("picked a string: {:?}", pick(false, 'L', 'R'));
}
```

(playground link)

Bodies of generic functions need to be well-defined for all possible types T.

5.2. Monomorphisation

The compiler **generates concrete versions** of generic functions for each used type.

```rust
fn pick_i32(cond: bool, left: i32, right: i32) -> i32 {
    if cond { left } else { right }
}

fn pick_char(cond: bool, left: char, right: char) -> char {
    if cond { left } else { right }
}
```
(playground link)

Generics are a zero-cost abstraction.

5.3. Trait bounds

Require the types to implement some trait, so that you can call this trait's methods

```
fn duplicate<T: Clone>(a: T) -> (T, T) {
    (a.clone(), a.clone())
}

struct NotCloneable;

fn main() {
    let foo = String::from("foo");
    let pair = duplicate(foo);
    println!("{pair:?}");
}
```

(playground link)

What happens if we pass `NotCloneable` to duplicate? (Try in playground!)

5.3. Trait bounds

Require the types to implement some trait, so that you can call this trait's methods

```rust
fn duplicate<T: Clone>(a: T) -> (T, T) {
    (a.clone(), a.clone())
}

struct NotCloneable;

fn main() {
    let foo = String::from("foo");
    let pair = duplicate(foo);
    println!("{pair:?}");
}
```

(playground link)

What happens if we pass `NotCloneable` to duplicate? (Try in playground!)

A compile-time error, because `NotCloneable` does not implement the Clone trait.

5.4. Combining traits

When multiple traits are necessary, use + to join them.

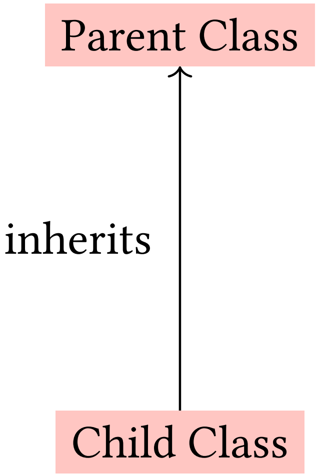
```rust
fn compare_and_print<T: PartialOrd + Display>(a: T, b: T) {
    if a < b {
        println!("{} is less than {}", a, b);
    } else {
        println!("{} is not less than {}", a, b);
    }
}
```

(playground link)

5.5. Composition over inheritance

Rust forbids object inheritance completely.

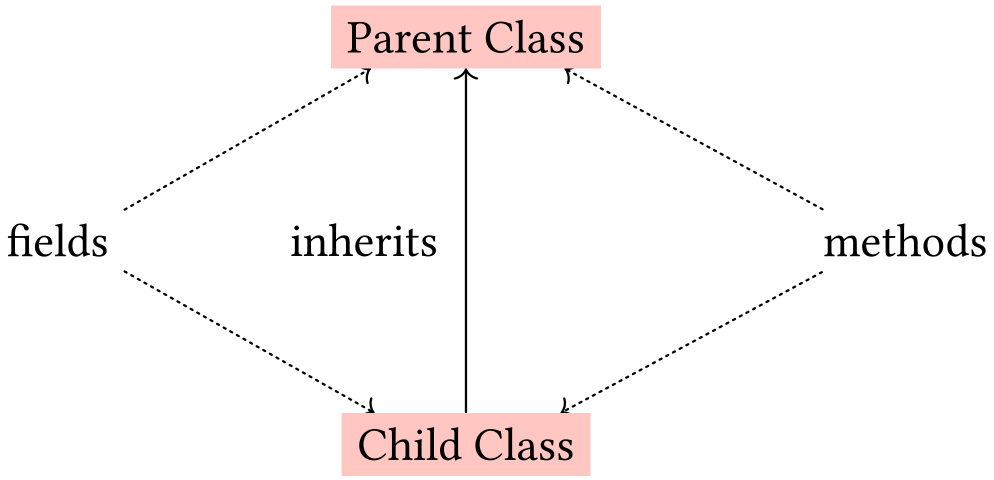
Class inheritance



5.5. Composition over inheritance

Rust forbids object inheritance completely.

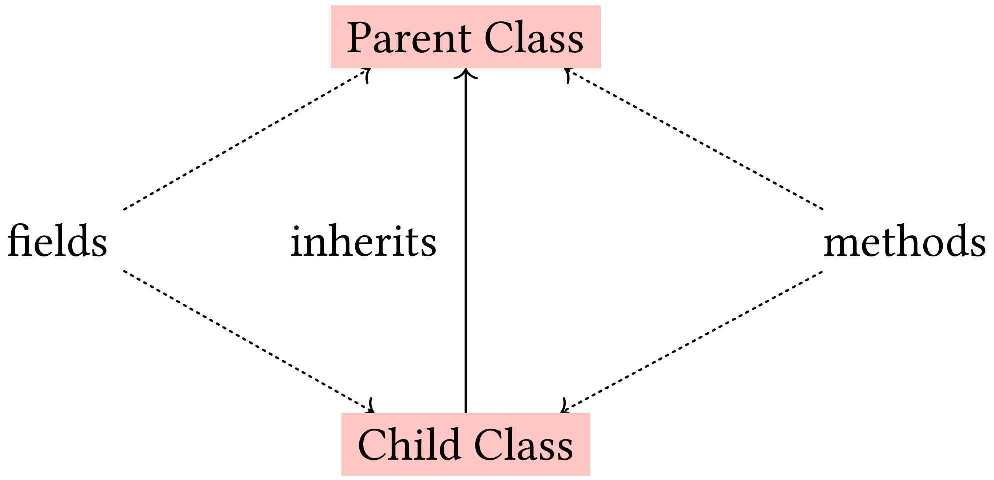
Class inheritance



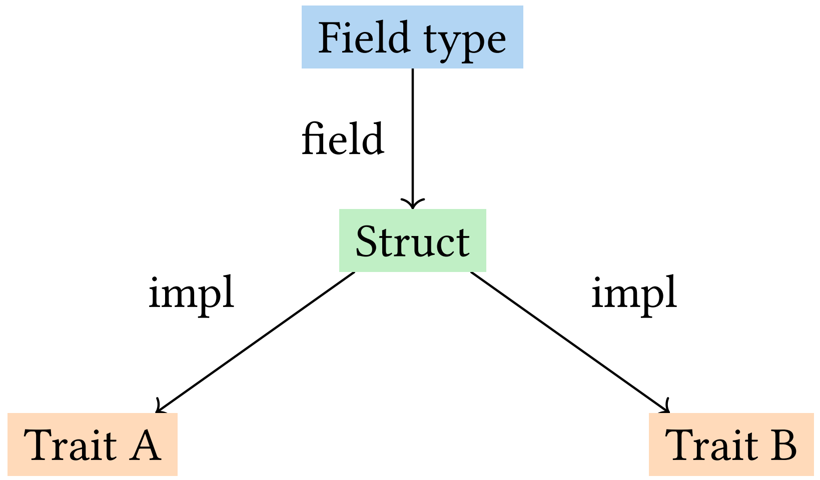
5.5. Composition over inheritance

Rust forbids object inheritance completely.

Class inheritance



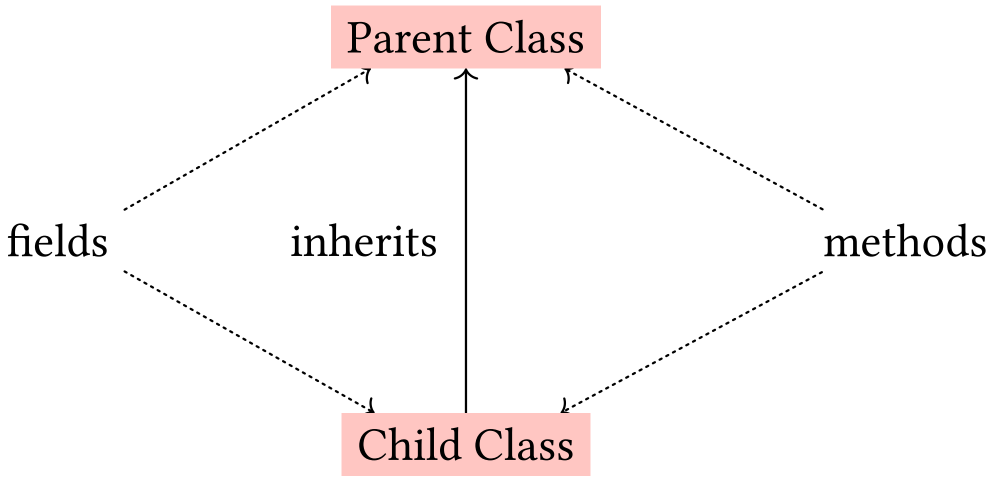
Composition wit traits



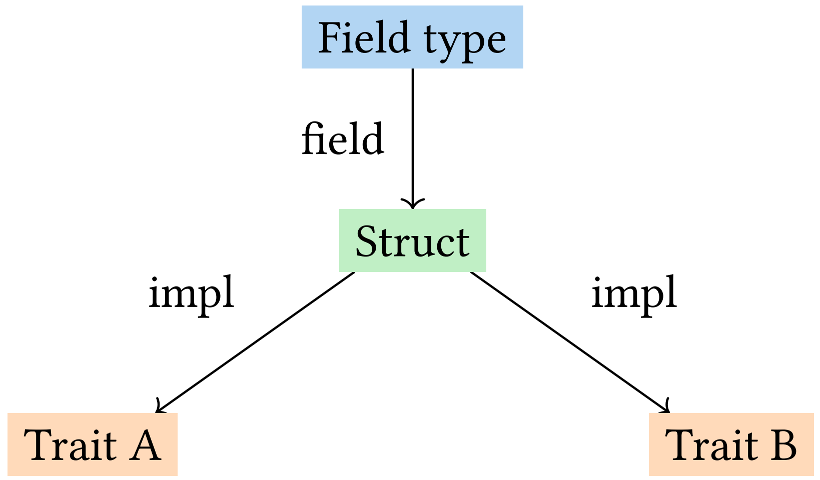
5.5. Composition over inheritance

Rust forbids object inheritance completely.

Class inheritance



Composition wit traits



Rust approach: Embed types as fields + implement multiple traits

Info

Types and traits are completely separate. No data inheritance.

5.6. where clauses

Declutters the function signature if you have many parameters

```rust
fn duplicate<T>(a: T) -> (T, T)
where
    T: Clone,
{
    (a.clone(), a.clone())
}
```

(playground link)

5.7. Feature of where clauses


```rust
fn duplicate_option<T>(a: Option<T>) -> (Option<T>, Option<T>)
where
  Option<T>: Clone,
{
  (a.clone(), a.clone())
}

fn main() {
  let s = Some(String::from("hello"));
  let pair = duplicate_option(s);
  println!("{pair:?}");
}
```

(playground link)

5.8. Generic datatypes


```rust
pub trait Logger {
    /// Log a message at the given verbosity level.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("verbosity={verbosity}: {message}");
    }
}

/// Only log messages up to the given verbosity level.
struct VerbosityFilter<L> {
    max_verbosity: u8,
    inner: L,
}
```

(playground link)

```rust
impl<L: Logger> Logger for VerbosityFilter<L> {
    fn log(&self, verbosity: u8, message: &str) {
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    logger.log(5, "FYI");
    logger.log(2, "Uhoh");
}
```

(playground link)

Use generics to abstract over the concrete field type.

5.9. If `X` impl, then `Y` impl

5.9.1. Generic impl block

```rust
struct VerbosityFilter<L> {
    max_verbosity: u8,
    inner: L,
}

impl<L: Logger> Logger for VerbosityFilter<L> {
    fn log(&self, verbosity: u8, message: &str) {
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}
```

(playground link)

Why is `L` specified twice in `impl<L: Logger> .. VerbosityFilter<L>`? Isn't that redundant?

5.9. If `X` impl, then `Y` impl

5.9.1. Generic impl block

```rust
struct VerbosityFilter<L> {
    max_verbosity: u8,
    inner: L,
}

impl<L: Logger> Logger for VerbosityFilter<L> {
    fn log(&self, verbosity: u8, message: &str) {
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}
```

(playground link)

Why is `L` specified twice in `impl<L: Logger> .. VerbosityFilter<L>`? Isn't that redundant?

`impl` parameters are separate and usually carries trait bounds (not the datatype).

What happens if you would just use a concrete type as in `impl VerbosityFilter<StderrLogger> { .. }`

5.9. If `X` impl, then `Y` impl

5.9.1. Generic impl block

```rust
struct VerbosityFilter<L> {
    max_verbosity: u8,
    inner: L,
}

impl<L: Logger> Logger for VerbosityFilter<L> {
    fn log(&self, verbosity: u8, message: &str) {
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}
```

(playground link)

Why is `L` specified twice in `impl<L: Logger> .. VerbosityFilter<L>`? Isn't that redundant?

`impl` parameters are separate and usually carries trait bounds (not the datatype).

What happens if you would just use a concrete type as in `impl VerbosityFilter<StderrLogger> { .. }`

Would only work with `StderrLogger` instances, not with any other type that implements `Logger`.

5.9. If `X` impl, then `Y` impl

5.9.2. Blanket impls

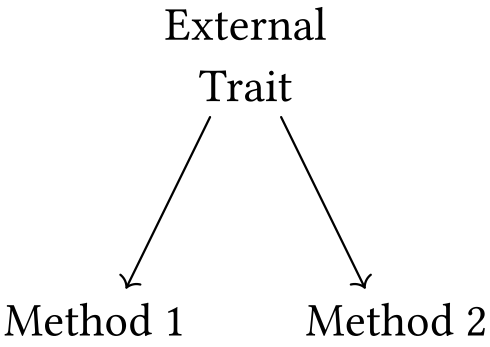
External libary

External Trait

5.9. If `X` impl, then `Y` impl

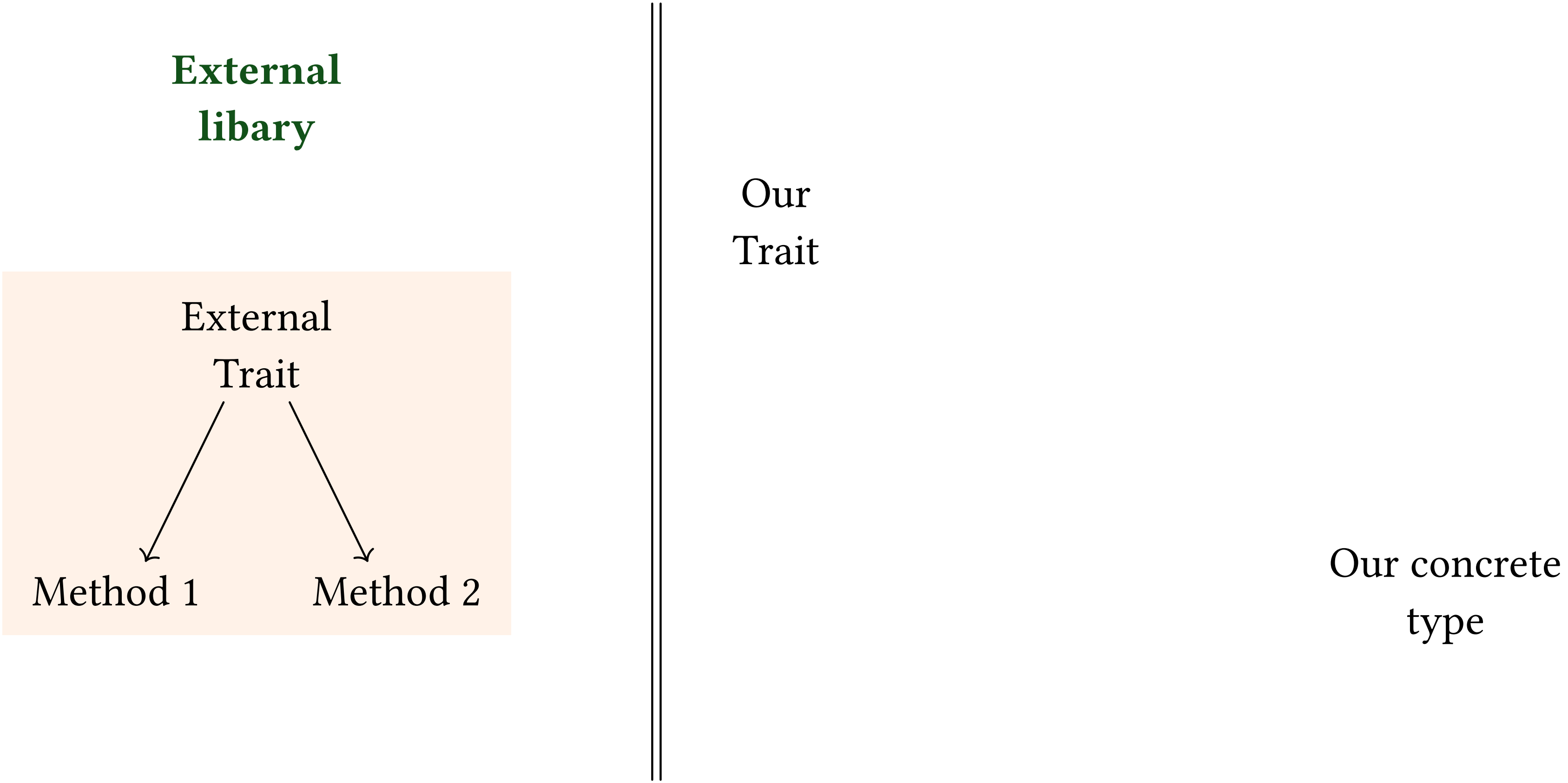
5.9.2. Blanket impls

External libary



5.9. If `X` impl, then `Y` impl

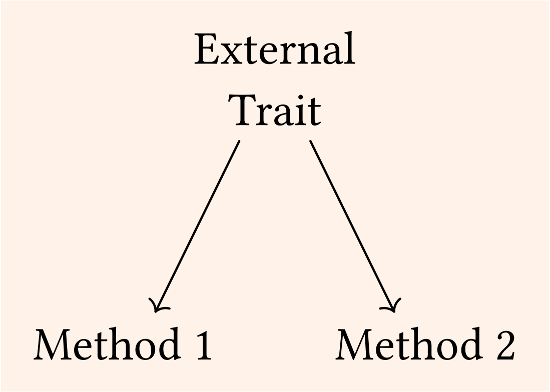
5.9.2. Blanket impls

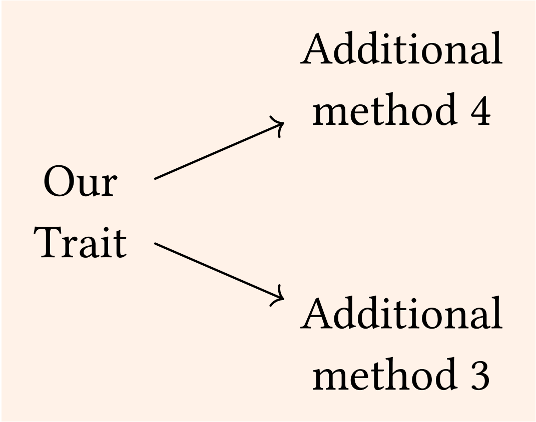


5.9. If `X` impl, then `Y` impl

5.9.2. Blanket impls

External libary

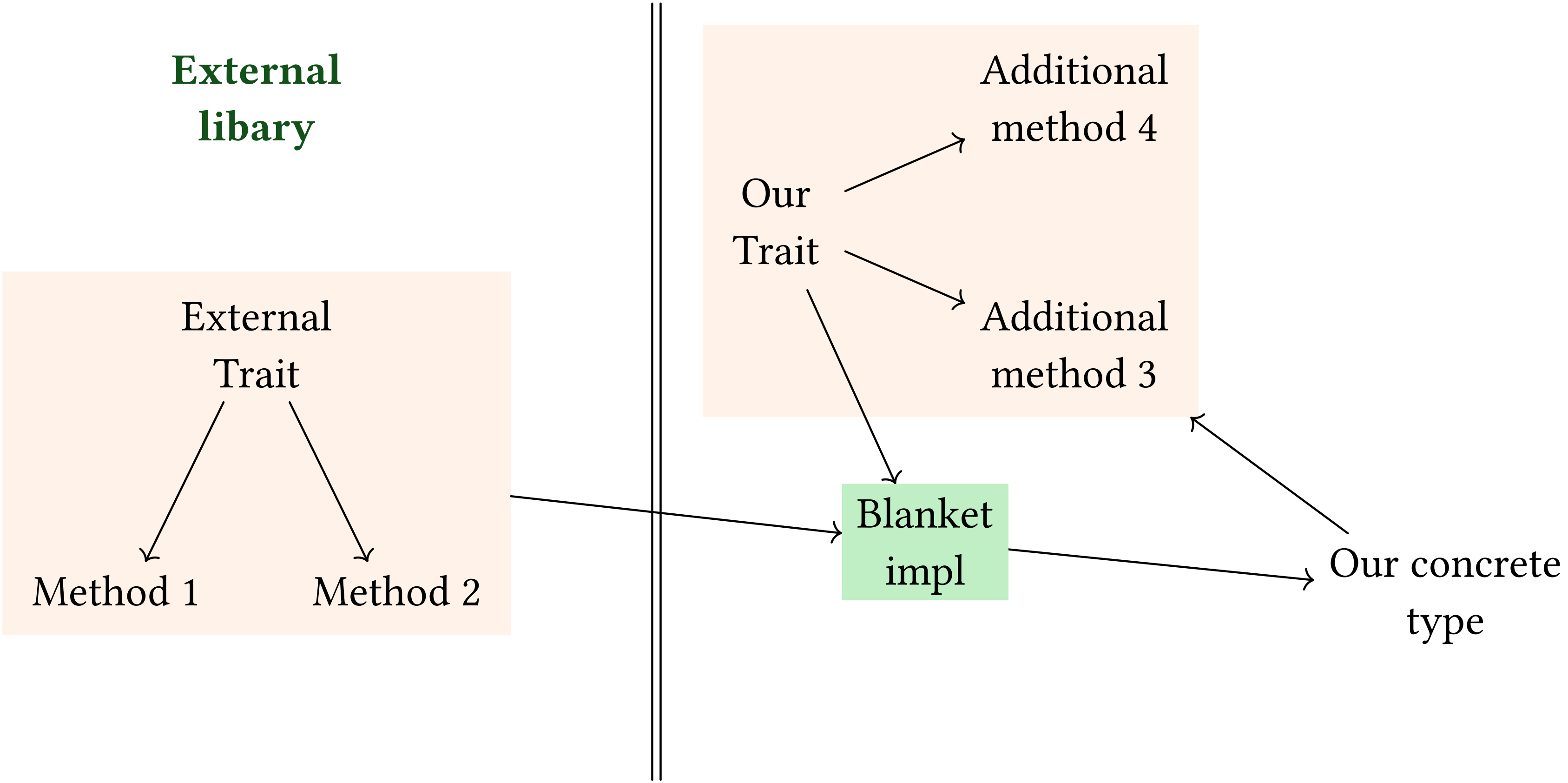




Our concrete type

5.9. If `X` impl, then `Y` impl

5.9.2. Blanket impls



5.9. If `X` impl, then `Y` impl

5.9.3. Putting constraints on super-traits (advanced)

A subtrait can add constraints on the associated types of its supertrait.

```
trait Container {
    type Item;
    fn get(&self) -> &Self::Item;
}

trait PrintableContainer: Container
where
    Self::Item: Display,
{
    fn print_item(&self) {
        println!("Item: {}", self.get());
    }
}

struct Box<T> {
    value: T,
}
```

(playground link)

```
impl<T> Container for Box<T> {
    type Item = T;
    fn get(&self) -> &Self::Item {
        &self.value
    }
}

impl<T: Display> PrintableContainer for Box<T> {}

fn main() {
    let b = Box { value: 42 };
    b.print_item();
}
```

(playground link)

Why does `PrintableContainer` require `Self::Item: Display`?

5.9. If `X` impl, then `Y` impl

5.9.3. Putting constraints on super-traits (advanced)

A subtrait can add constraints on the associated types of its supertrait.

```
trait Container {
    type Item;
    fn get(&self) -> &Self::Item;
}

trait PrintableContainer: Container
where
    Self::Item: Display,
{
    fn print_item(&self) {
        println!("Item: {}", self.get());
    }
}

struct Box<T> {
    value: T,
}
```

(playground link)

```
impl<T> Container for Box<T> {
    type Item = T;
    fn get(&self) -> &Self::Item {
        &self.value
    }
}

impl<T: Display> PrintableContainer for Box<T> {}

fn main() {
    let b = Box { value: 42 };
    b.print_item();
}
```

(playground link)

Why does `PrintableContainer` require `Self::Item: Display`?

It constrains the supertrait's associated type so that `print_item` can use the `Display` trait.

5.10. Generic traits

The `From` trait is a standard library trait for type conversion.

```rust
pub trait From<T>: Sized {
    fn from(value: T) -> Self;
}

#[derive(Debug)]
struct Foo(String);
```

(playground link)

Info

Common traits in the standard library will be covered later on.

5.10. Generic traits

5.10.1. Example `From`

```rust
impl From<u32> for Foo {
    fn from(from: u32) -> Foo {
        Foo(format!("Converted from integer: {from}"))
    }
}

impl From<bool> for Foo {
    fn from(from: bool) -> Foo {
        Foo(format!("Converted from bool: {from}"))
    }
}
fn main() {
    dbg!(Foo::from(123));
    dbg!(Foo::from(true));
}
```

(playground link)

Warning

Notice that a **generic trait can be implemented multiple times** for the same type!

5.11. Associated types vs. generic type parameters

It may not always be clear when to pick an associated type or a generic type.

Associated types

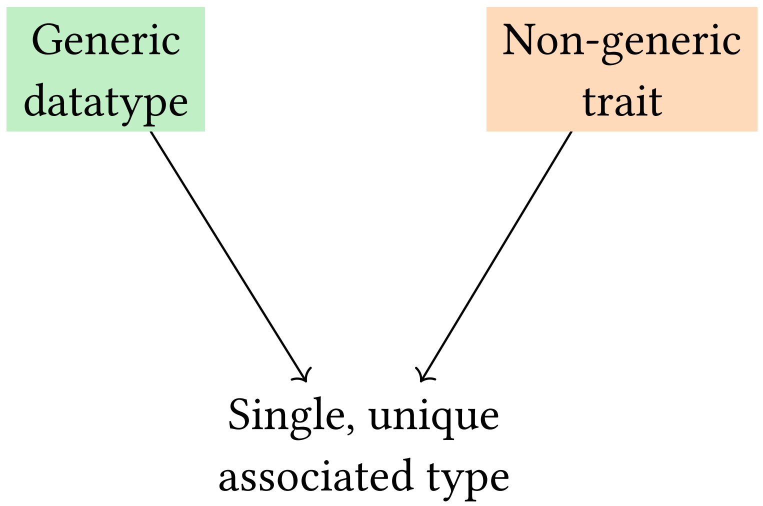
Generic datatype

Non-generic trait

5.11. Associated types vs. generic type parameters

It may not always be clear when to pick an associated type or a generic type.

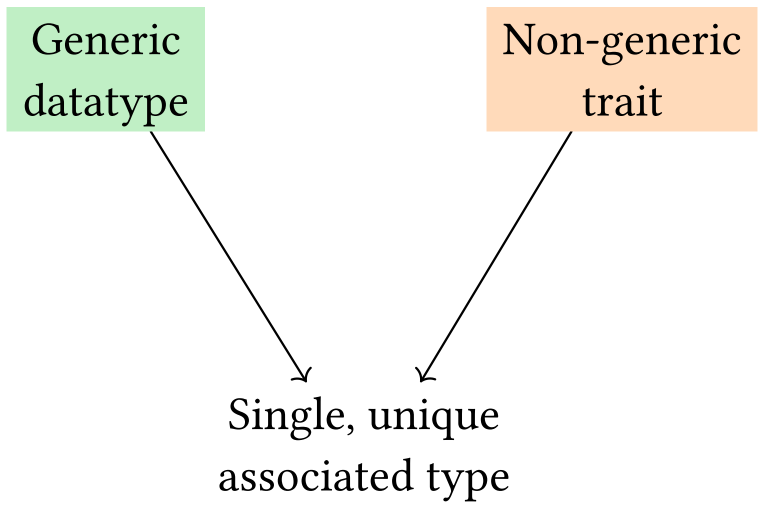
Associated types



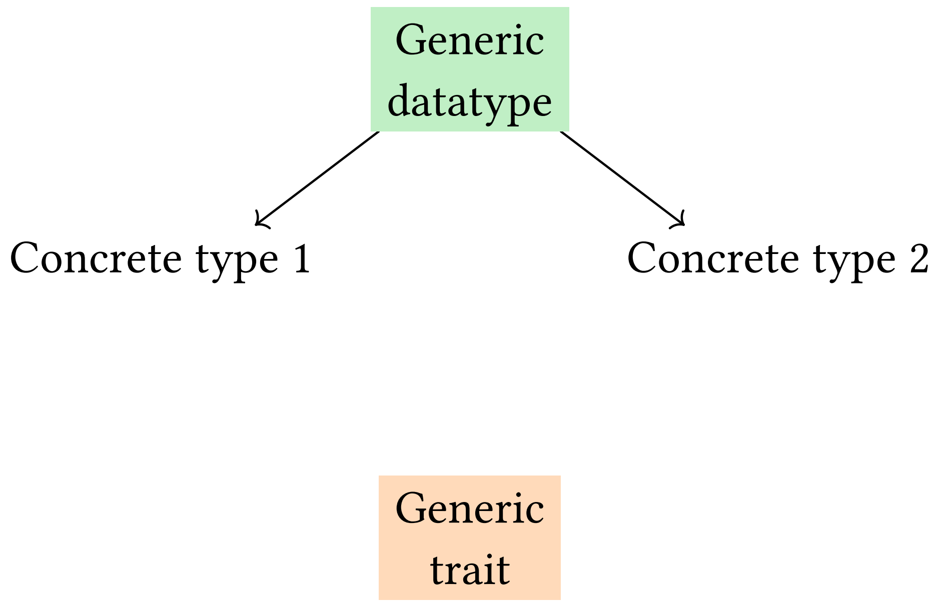
5.11. Associated types vs. generic type parameters

It may not always be clear when to pick an associated type or a generic type.

Associated types



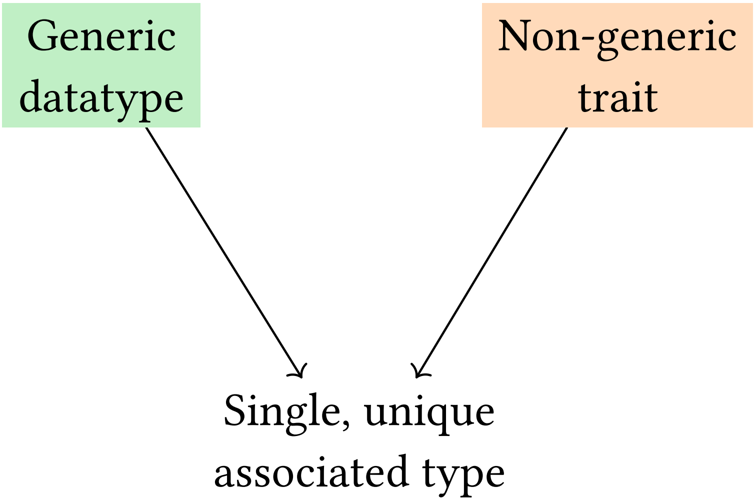
Generic types parameters



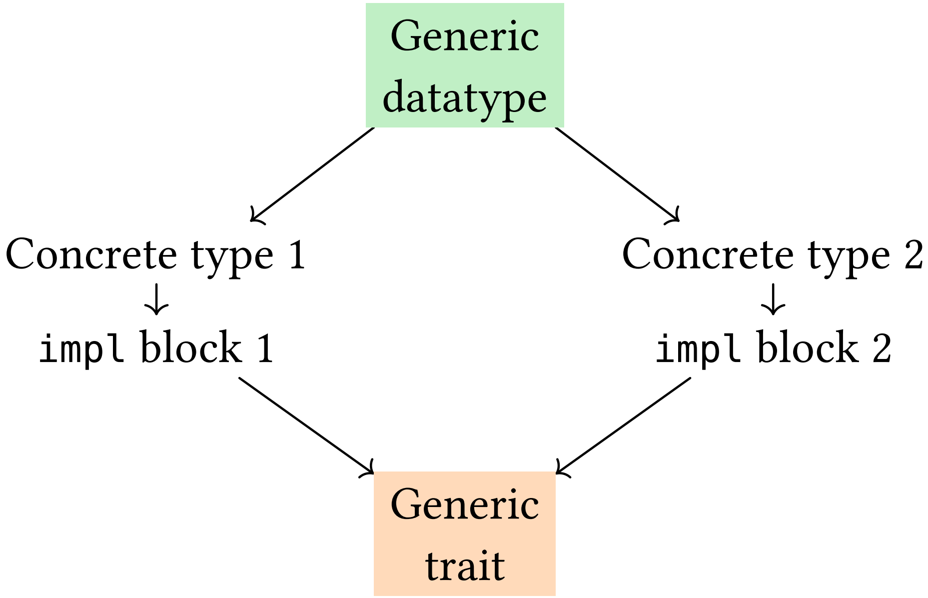
5.11. Associated types vs. generic type parameters

It may not always be clear when to pick an associated type or a generic type.

Associated types



Generic types parameters



- Associated types behave like output (first choice).
- Generic type parameters behave like input.

5.12. Multiple generic `impl` blocks

```rust
struct Container<T> {
  value: T,
}

impl<T> Container<T> {
  fn process(&self) -> &'static str { "generic implementation" }
}

impl Container<i32> {
  fn process(&self) -> &'static str { "specialized implementation for i32" }
}
```

(playground link)

Will this code compile?

5.12. Multiple generic `impl` blocks

```rust
struct Container<T> {
  value: T,
}

impl<T> Container<T> {
  fn process(&self) -> &'static str { "generic implementation" }
}

impl Container<i32> {
  fn process(&self) -> &'static str { "specialized implementation for i32" }
}
```

(playground link)

Will this code compile?

No. Rust does not allow overlapping implementations.

5.13. `impl Trait`

Info

`impl Into<i32>` is syntactic sugar for: `fn add_42_millions<T: Into<i32>>(x: T) -> i32`.
`T` is an anonymous and **hidden generic type**.

```rust
fn add_42_millions(x: impl Into<i32>) -> i32 {
    x.into() + 42_000_000
}

fn pair_of(x: u32) -> impl Debug {
    (x + 1, x - 1)
}

fn main() {
    let many = add_42_millions(42_i8);
    dbg!(many);
    let many_more = add_42_millions(10_000_000);
    dbg!(many_more);
    let debuggable = pair_of(27);
    dbg!(debuggable);
}
```

(playground link)

5.13. impl Trait

5.13.1. Inference for return type impl Trait

```rust
fn returns_impl_trait(x: i32) -> impl std::fmt::Display {
    if x > 0 {
        x
    } else {
        -x
    }
}

fn main() {
    let result = returns_impl_trait(-5);
    println!("{}", result);
}
```

(playground link)

What is the return type of `returns_impl_trait`?

5.13. `impl Trait`

5.13.1. Inference for return type `impl Trait`

```rust
fn returns_impl_trait(x: i32) -> impl std::fmt::Display {
    if x > 0 {
        x
    } else {
        -x
    }
}

fn main() {
    let result = returns_impl_trait(-5);
    println!("{}", result);
}
```

(playground link)

What is the return type of `returns_impl_trait`?

`i32`, because both branches return an `i32`.

5.14. Exercise

Please find the exercise in `./session-2/tests/s2e4-min.rs`.

Test using the command: `cargo test --test s2e4-min -- --nocapture`

6.1. Syntax

Info

Closures are anonymous functions that can capture variables from their surrounding scope.

```rust
fn main() {
    // Argument and return type can be inferred for lightweight syntax:
    let double_it = |n| n * 2;
    dbg!(double_it(50));

    // Or we can specify types and bracket the body to be fully explicit:
    let add_1f32 = |x: f32| -> f32 { x + 1.0 };
    dbg!(add_1f32(50.));
}
```

(playground link)

Properties:

6.1. Syntax

Info

Closures are anonymous functions that can capture variables from their surrounding scope.

```rust
fn main() {
  // Argument and return type can be inferred for lightweight syntax:
  let double_it = |n| n * 2;
  dbg!(double_it(50));

  // Or we can specify types and bracket the body to be fully explicit:
  let add_1f32 = |x: f32| -> f32 { x + 1.0 };
  dbg!(add_1f32(50.));
}
```

(playground link)

Properties:

- The body may be be surrounded by `{ .. }`

6.1. Syntax

Info

Closures are anonymous functions that can capture variables from their surrounding scope.

```rust
fn main() {
  // Argument and return type can be inferred for lightweight syntax:
  let double_it = |n| n * 2;
  dbg!(double_it(50));

  // Or we can specify types and bracket the body to be fully explicit:
  let add_1f32 = |x: f32| -> f32 { x + 1.0 };
  dbg!(add_1f32(50.));
}
```

(playground link)

Properties:

- The body may be be surrounded by `{ .. }`
- Argument and return types are optional, and are inferred if not given.

6.2. Capturing

```rust
fn main() {
    let max_value = 5;
    let clamp = |v| {
        if v > max_value
        { max_value } else { v }
    };

    dbg!(clamp(1));
    dbg!(clamp(3));
    dbg!(clamp(5));
    dbg!(clamp(7));
    dbg!(clamp(10));
}
```

(playground link)

By default, a closure captures values by reference. Here max_value is captured by clamp, but still available to main for printing.

> Try making max_value mutable, changing it, and printing the clamped values again. Why doesn't this work?

6.2. Capturing


```rust
fn main() {
    let max_value = 5;
    let clamp = |v| {
        if v > max_value { max_value } else { v }
    };

    dbg!(clamp(1));
    dbg!(clamp(3));
    dbg!(clamp(5));
    dbg!(clamp(7));
    dbg!(clamp(10));
}                                   (playground link)
```

By default, a closure captures values by reference. Here max_value is captured by clamp, but still available to main for printing.

Try making max_value mutable, changing it, and printing the clamped values again. Why doesn't this work?

The closure captures by reference. No mutable borrow may coexist with an immutable borrow.

6.2. Capturing

6.2.1. Mutable capture

```rust
fn main() {
    let mut max_value = 5;
    let mut clamp = |v| {
        max_value += 1;
        if v > max_value { max_value } else { v }
    };

    dbg!(clamp(1));
    dbg!(clamp(3));
    dbg!(clamp(5));
    dbg!(clamp(7));
    dbg!(clamp(10));
}
```

(playground link)

If a closure mutates captured values, it captures them by mutable reference.

6.2. Capturing

6.2.1. Mutable capture

```rust
fn main() {
    let mut max_value = 5;
    let mut clamp = |v| {
        max_value += 1;
        if v > max_value { max_value } else { v }
    };

    dbg!(clamp(1));
    dbg!(clamp(3));
    dbg!(clamp(5));
    dbg!(clamp(7));
    dbg!(clamp(10));
}
```

(playground link)

If a closure mutates captured values, it captures them by mutable reference.

Why does the closure need to be `mut`?

6.2. Capturing

6.2.1. Mutable capture

```rust
fn main() {
    let mut max_value = 5;
    let mut clamp = |v| {
        max_value += 1;
        if v > max_value { max_value } else { v }
    };

    dbg!(clamp(1));
    dbg!(clamp(3));
    dbg!(clamp(5));
    dbg!(clamp(7));
    dbg!(clamp(10));
}
```

(playground link)

If a closure mutates captured values, it captures them by mutable reference.

Why does the closure need to be `mut`?

The closure's internal state changes, so it must be declared mutable.

6.2. Capturing

6.2.1. Mutable capture

```rust
fn main() {
    let mut max_value = 5;
    let mut clamp = |v| {
        max_value += 1;
        if v > max_value
        { max_value } else { v }
    };

    dbg!(clamp(1));
    dbg!(clamp(3));
    dbg!(clamp(5));
    dbg!(clamp(7));
    dbg!(clamp(10));
}
```

(playground link)

If a closure mutates captured values, it captures them by mutable reference.

Why does the closure need to be `mut`?

The closure's internal state changes, so it must be declared mutable.

What happens if you try to access `max_value` directly while the closure exists?

6.2. Capturing

6.2.1. Mutable capture

```rust
fn main() {
    let mut max_value = 5;
    let mut clamp = |v| {
        max_value += 1;
        if v > max_value { max_value } else { v }
    };

    dbg!(clamp(1));
    dbg!(clamp(3));
    dbg!(clamp(5));
    dbg!(clamp(7));
    dbg!(clamp(10));
}
```

(playground link)

If a closure mutates captured values, it captures them by mutable reference.

Why does the closure need to be `mut`?

The closure's internal state changes, so it must be declared mutable.

What happens if you try to access `max_value` directly while the closure exists?

Compile error: the closure holds a mutable borrow, preventing other accesses.

6.3. Moving

6.3.1. Move closures

```rust
fn main() {
    let name = String::from("Alice");
    let greet = move || {
        println!("Hello, {}", name);
    };

    greet();
    greet();

    // println!("{}", name); // Error!
}
```

(playground link)

The `move` keyword forces a closure to take ownership of captured values.

6.3. Moving

6.3.1. Move closures

```rust
fn main() {
    let name = String::from("Alice");
    let greet = move || {
        println!("Hello, {}", name);
    };

    greet();
    greet();

    // println!("{}", name); // Error!
}
```

(playground link)

The `move` keyword forces a closure to take ownership of captured values.

What happens if you uncomment the last line?

6.3. Moving

6.3.1. Move closures

```rust
fn main() {
    let name = String::from("Alice");
    let greet = move || {
        println!("Hello, {}", name);
    };

    greet();
    greet();

    // println!("{}", name); // Error!
}
```
(playground link)

The `move` keyword forces a closure to take ownership of captured values.

What happens if you uncomment the last line?

Compile error: `name` was moved into the closure and can no longer be used.

6.3. Moving

6.3.1. Move closures

```rust
fn main() {
    let name = String::from("Alice");
    let greet = move || {
        println!("Hello, {}", name);
    };

    greet();
    greet();

    // println!("{}", name); // Error!
}
```

(playground link)

The `move` keyword forces a closure to take ownership of captured values.

What happens if you uncomment the last line?

Compile error: `name` was moved into the closure and can no longer be used.

Why can we call `greet()` multiple times?

6.3. Moving

6.3.1. Move closures

```rust
fn main() {
    let name = String::from("Alice");
    let greet = move || {
        println!("Hello, {}", name);
    };

    greet();
    greet();

    // println!("{}", name); // Error!
}
```

(playground link)

The `move` keyword forces a closure to take ownership of captured values.

What happens if you uncomment the last line?

Compile error: `name` was moved into the closure and can no longer be used.

Why can we call `greet()` multiple times?

The closure owns `name`, which implements `Clone`, so it can be called repeatedly.

6.3. Moving

6.3.1. Move closures

```rust
fn main() {
    let name = String::from("Alice");
    let greet = move || {
        println!("Hello, {}", name);
    };

    greet();
    greet();

    // println!("{}", name); // Error!
}
```
(playground link)

The `move` keyword forces a closure to take ownership of captured values.

What happens if you uncomment the last line?

Compile error: `name` was moved into the closure and can no longer be used.

Why can we call `greet()` multiple times?

The closure owns `name`, which implements `Clone`, so it can be called repeatedly.

Common use case: passing closures to threads or async tasks that must outlive their parent scope.

6.4. Internal representation of closures

Closures are like structs with a function field:

1. Each captured variable is a reference field.
2. The closure body is a method that uses these fields.

For example, the closure `|x| x + offset` where `offset` is captured from the environment is represented as:

```rust
struct Closure {
    offset: &i32,
}

impl Closure {
    fn call(&self, x: i32) -> i32 {
        x + *self.offset
    }
}
```

(playground link)

6.4. Internal representation of closures

Closures are like structs with a function field:

1. Each captured variable is a reference field.
2. The closure body is a method that uses these fields.

For example, the closure `|x| x + offset` where `offset` is captured from the environment is represented as:

```rust
struct Closure {
    offset: &i32,
}

impl Closure {
    fn call(&self, x: i32) -> i32 {
        x + *self.offset
    }
}
```

(playground link)

Warning

This **representation is hidden**.

Closures belong to the category of **anonymous types**. Each closure has a unique, unnamed type generated by the compiler.

6.5. Closure traits

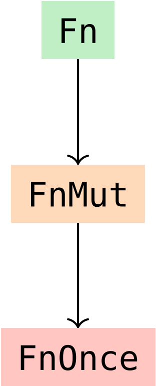
Please open demo file `session-2/examples/s2e5-closure-traits.rs`.

6.5. Closure traits

Closures have anonymous types that implement one or more of these traits:

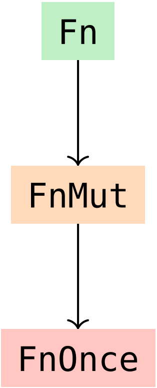
6.5. Closure traits

Closures have anonymous types that implement one or more of these traits:



6.5. Closure traits

Closures have anonymous types that implement one or more of these traits:



Trait hierarchy in Rust stdlib:

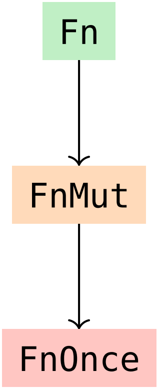
```
trait FnOnce { /* ... */ }
trait FnMut: FnOnce { /* ... */ }
trait Fn: FnMut { /* ... */ }
```

(playground link)

- **`Fn`**: Immutable capture, callable multiple times, even concurrently
- **`FnMut`**: Mutable capture, callable multiple times, not concurrently
- **`FnOnce`**: Ownership capture, callable once, consumes closure

6.5. Closure traits

Closures have anonymous types that implement one or more of these traits:



Trait hierarchy in Rust stdlib:

```
trait FnOnce { /* ... */ }
trait FnMut: FnOnce { /* ... */ }
trait Fn: FnMut { /* ... */ }
```

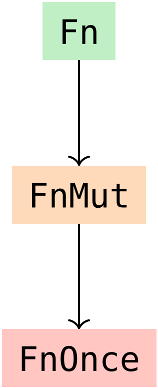
(playground link)

- **`Fn`**: Immutable capture, callable multiple times, even concurrently
- **`FnMut`**: Mutable capture, callable multiple times, not concurrently
- **`FnOnce`**: Ownership capture, callable once, consumes closure

Can a closure implementing `Fn` be used where `FnOnce` is required?

6.5. Closure traits

Closures have anonymous types that implement one or more of these traits:



Trait hierarchy in Rust stdlib:

```
trait FnOnce { /* ... */ }
trait FnMut: FnOnce { /* ... */ }
trait Fn: FnMut { /* ... */ }
```
(playground link)

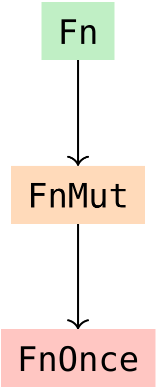
- **`Fn`**: Immutable capture, callable multiple times, even concurrently
- **`FnMut`**: Mutable capture, callable multiple times, not concurrently
- **`FnOnce`**: Ownership capture, callable once, consumes closure

Can a closure implementing `Fn` be used where `FnOnce` is required?

Yes. `Fn` is a subtrait of `FnMut`, which is a subtrait of `FnOnce`.

6.5. Closure traits

Closures have anonymous types that implement one or more of these traits:



Trait hierarchy in Rust stdlib:

```
trait FnOnce { /* ... */ }
trait FnMut: FnOnce { /* ... */ }
trait Fn: FnMut { /* ... */ }
```
(playground link)

- **`Fn`**: Immutable capture, callable multiple times, even concurrently
- **`FnMut`**: Mutable capture, callable multiple times, not concurrently
- **`FnOnce`**: Ownership capture, callable once, consumes closure

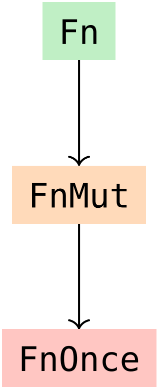
Can a closure implementing `Fn` be used where `FnOnce` is required?

Yes. `Fn` is a subtrait of `FnMut`, which is a subtrait of `FnOnce`.

What does the `:` in `trait Fn: FnMut` mean?

6.5. Closure traits

Closures have anonymous types that implement one or more of these traits:



Trait hierarchy in Rust stdlib:

```
trait FnOnce { /* ... */ }
trait FnMut: FnOnce { /* ... */ }
trait Fn: FnMut { /* ... */ }
```

(playground link)

- **`Fn`**: Immutable capture, callable multiple times, even concurrently
- **`FnMut`**: Mutable capture, callable multiple times, not concurrently
- **`FnOnce`**: Ownership capture, callable once, consumes closure

Can a closure implementing `Fn` be used where `FnOnce` is required?

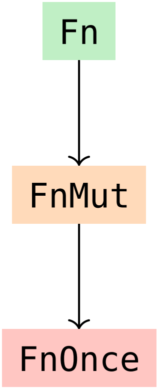
Yes. `Fn` is a subtrait of `FnMut`, which is a subtrait of `FnOnce`.

What does the `:` in `trait Fn: FnMut` mean?

`Fn` is a subtrait of `FnMut`, meaning anything implementing `Fn` must also implement `FnMut`.

6.5. Closure traits

Closures have anonymous types that implement one or more of these traits:



Trait hierarchy in Rust stdlib:

```rust
trait FnOnce { /* ... */ }
trait FnMut: FnOnce { /* ... */ }
trait Fn: FnMut { /* ... */ } (playground link)
```

- **`Fn`**: Immutable capture, callable multiple times, even concurrently
- **`FnMut`**: Mutable capture, callable multiple times, not concurrently
- **`FnOnce`**: Ownership capture, callable once, consumes closure

Can a closure implementing `Fn` be used where `FnOnce` is required?

Yes. `Fn` is a subtrait of `FnMut`, which is a subtrait of `FnOnce`.

What does the `:` in `trait Fn: FnMut` mean?

`Fn` is a subtrait of `FnMut`, meaning anything implementing `Fn` must also implement `FnMut`.

Info

All `Fn` closures are also `FnMut` and `FnOnce`. All `FnMut` closures are also `FnOnce`.

6.6. Picking the right closure trait

When accepting closures

Prefer the least restrictive trait:

1. `FnOnce` - called once (most flexible)
2. `FnMut` - called multiple times, may mutate
3. `Fn` - called multiple times, no mutation

6.6. Picking the right closure trait

When accepting closures

Prefer the least restrictive trait:

1. `FnOnce` - called once (most flexible)
2. `FnMut` - called multiple times, may mutate
3. `Fn` - called multiple times, no mutation

`FnOnce`

↑ is a subtrait of

`FnMut`

↑ is a subtrait of

`Fn`

Maximizes caller flexibility

When providing closures

Prefer the most restrictive trait:

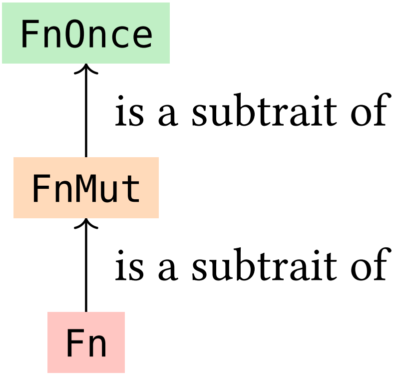
1. `Fn` - most flexible (most usable)
2. `FnMut` - medium flexibility
3. `FnOnce` - least flexible

6.6. Picking the right closure trait

When accepting closures

Prefer the least restrictive trait:

1. `FnOnce` - called once (most flexible)
2. `FnMut` - called multiple times, may mutate
3. `Fn` - called multiple times, no mutation

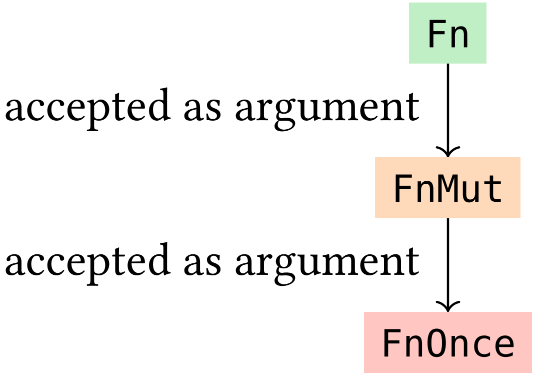


Maximizes caller flexibility

When providing closures

Prefer the most restrictive trait:

1. `Fn` - most flexible (most usable)
2. `FnMut` - medium flexibility
3. `FnOnce` - least flexible



Maximizes usage flexibility

6.6. Picking the right closure trait

When accepting closures

Prefer the least restrictive trait:

1. `FnOnce` - called once (most flexible)
2. `FnMut` - called multiple times, may mutate
3. `Fn` - called multiple times, no mutation

$$\begin{tikzcd} \texttt{FnOnce} \\ \texttt{FnMut} \arrow[u, "\text{is a subtrait of}"'] \\ \texttt{Fn} \arrow[u, "\text{is a subtrait of}"'] \end{tikzcd}$$

Maximizes caller flexibility

When providing closures

Prefer the most restrictive trait:

1. `Fn` - most flexible (most usable)
2. `FnMut` - medium flexibility
3. `FnOnce` - least flexible

$$\begin{tikzcd} \texttt{Fn} \arrow[d, "\text{accepted as argument}"'] \\ \texttt{FnMut} \arrow[d, "\text{accepted as argument}"'] \\ \texttt{FnOnce} \end{tikzcd}$$

Maximizes usage flexibility

Info

`Fn` closures can be used wherever `FnMut` or `FnOnce` is expected.

6.7. Auto-traits of closures

What are auto-traits?

6.7. Auto-traits of closures

What are auto-traits?

Traits that the compiler can automatically implement for types based on their structure.

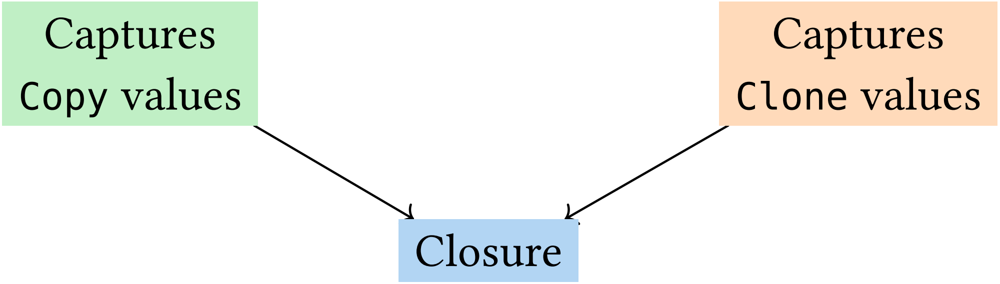
Closures automatically implement `Copy` and `Clone` based on captured values:

6.7. Auto-traits of closures

What are auto-traits?

Traits that the compiler can automatically implement for types based on their structure.

Closures automatically implement `Copy` and `Clone` based on captured values:

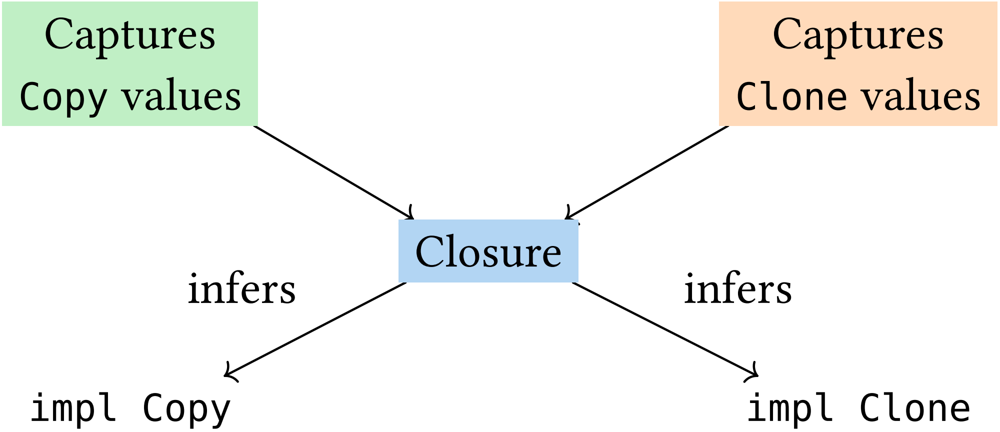


6.7. Auto-traits of closures

What are auto-traits?

Traits that the compiler can automatically implement for types based on their structure.

Closures automatically implement `Copy` and `Clone` based on captured values:

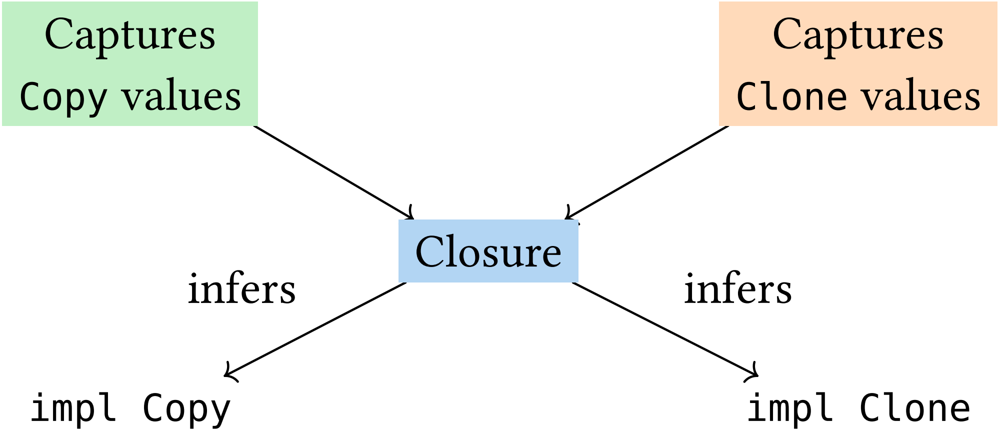


6.7. Auto-traits of closures

What are auto-traits?

Traits that the compiler can automatically implement for types based on their structure.

Closures automatically implement `Copy` and `Clone` based on captured values:



Info

Closures inherit properties from the values they capture.

6.8. Exercise

Please find the statement of the exercise at `session-2/examples/s2d6-closures.rs` in this repo.

Run it with `cargo run --example s2d6-closures`.

Homework: study standard library traits

Questions?

Feel free to contact me:

- Video meeting / email: willemvanhulle@protonmail.com
- Phone: +32 479 080 252