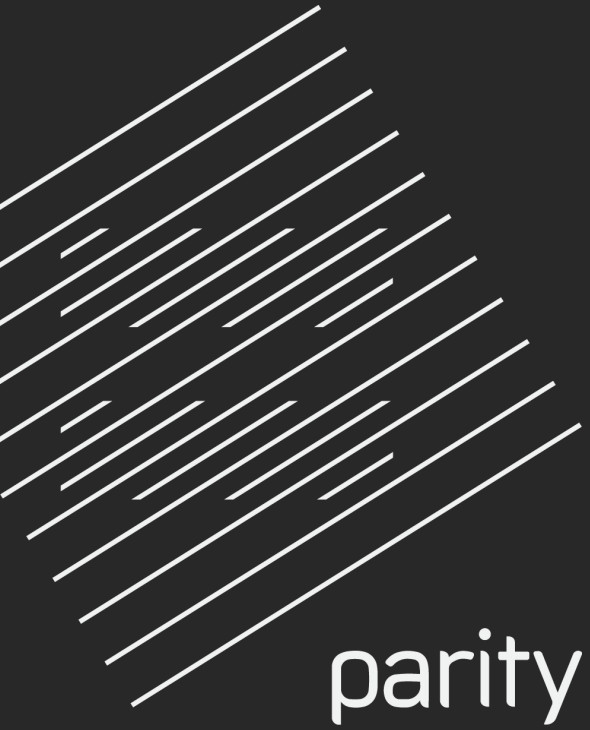


Intro to Rust for Substrate Developers

Or: how I learned to stop worrying and love lifetimes

Maciej Hirsz
Software developer @ Parity Technologies Ltd.

maciej@parity.io | @MaciejHirsz

Who is this for?

- Coming C / C++ or Python / Ruby / JS
- Completely new or beginner at Rust
- Want to work on **Substrate** modules or **ink!**
- Might have something for intermediate folks
- Discover the unknown unknowns

What we are going to cover here

- Rust philosophy
- Rust primitives and value types
- Error handling
- Implementing methods
- Trait system
- Lifetimes

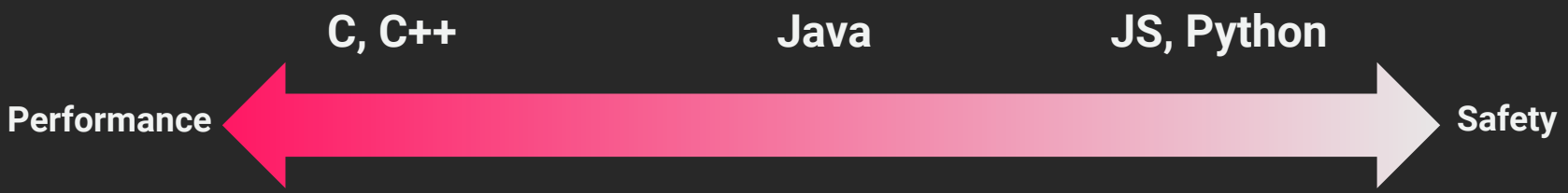
A COVER, GET IT?

parity

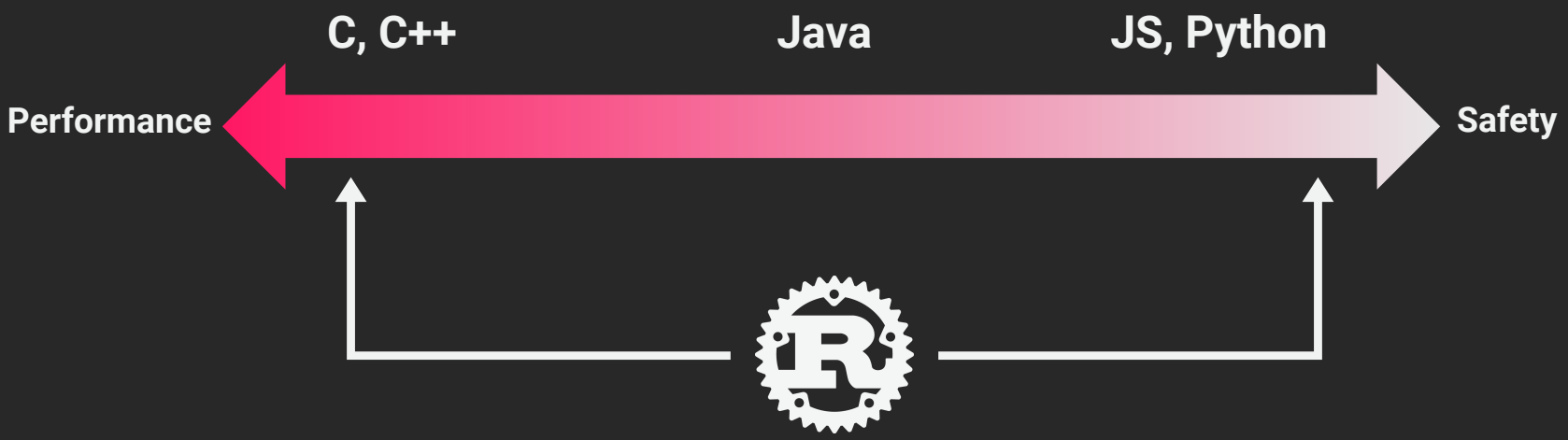
What we are NOT going to cover here

- Closures
- Multithreading, Mutexes, MPSC message passing
- Unsafe Rust
- Macros
- How types are represented in memory and more

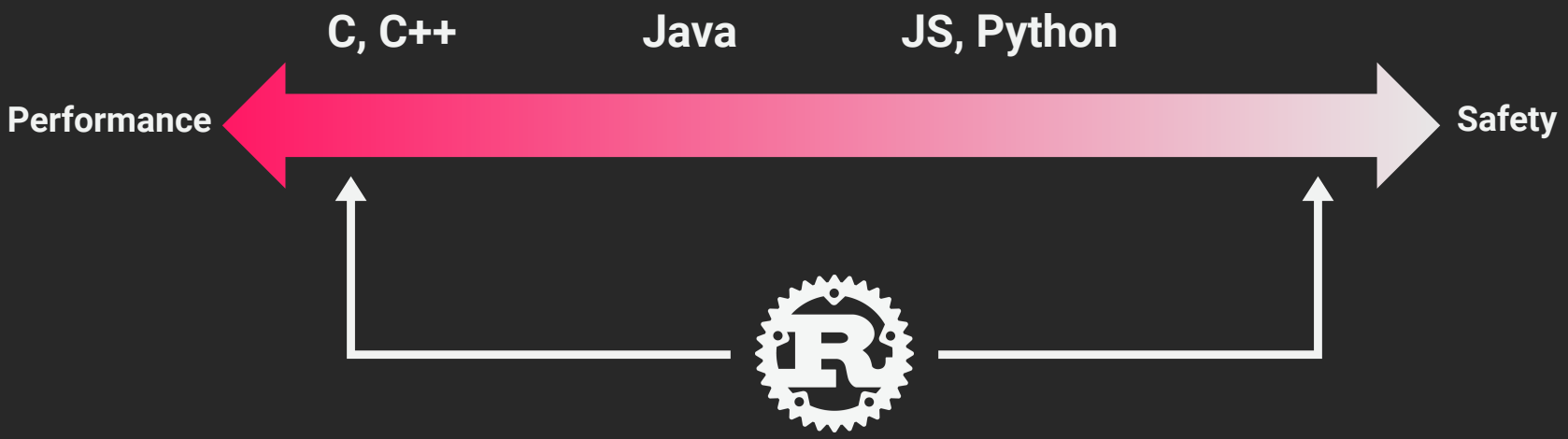
Rust philosophy



Rust philosophy



Rust philosophy



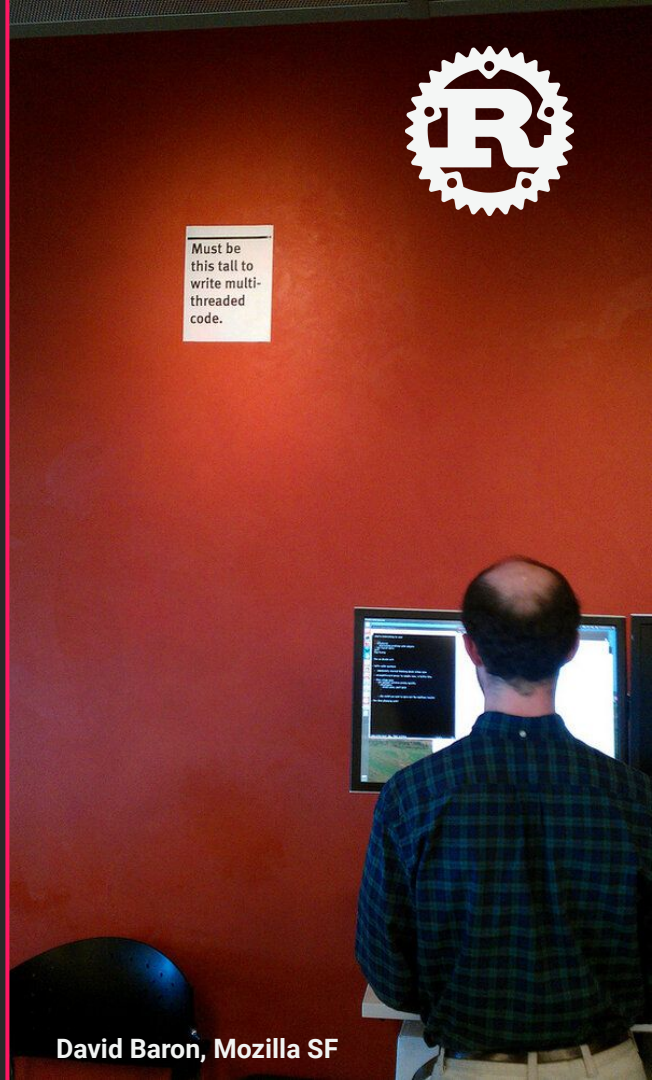
Rust philosophy

- Safe
- Concurrent
- Fast
- *Pick Three*

http://leftoversalad.com/c/015_programmingpeople/

Rust philosophy

- No Runtime overhead, no GC, C FFI
- Zero-Cost abstractions (like C++)
- Unique Ownership model (RAII)
- *Will hurt your feelings*
- *Will empower you*



Rust philosophy

```rust
fn bunch_of_numbers() -> Vec<u32> {
    let mut nums = Vec::new();
    for i in 0..10 {
        nums.push(i);
    }
    nums
}

fn main() {
    let nums = bunch_of_numbers();

    match nums.last() {
        Some(&0) => println!("Last number is zero"),
        Some(n)  => println!("Last number is {}", n),
        None     => println!("There are no numbers"),
    }
}
```

parity

Rust philosophy

```rust
fn bunch_of_numbers() -> Vec<u32> {
    let mut nums = Vec::new();
    for i in 0..10 {
        nums.push(i);
    }
    nums
}

fn main() {
    let nums = bunch_of_numbers();

    match nums.last() {
        Some(&0) => println!("Last number is zero"),
        Some(n)  => println!("Last number is {}", n),
        None     => println!("There are no numbers"),
    }
}
```

(re-)allocation

move

obtain ownership

deallocation

parity

Rust philosophy

```rust
fn bunch_of_numbers() -> Vec<u32> {
    let mut nums = Vec::with_capacity(10);
    for i in 0..10 {
        nums.push(i);
    }
    nums
}

fn main() {
    let nums = bunch_of_numbers();

    match nums.last() {
        Some(&0) => println!("Last number is zero"),
        Some(n)  => println!("Last number is {}", n),
        None     => println!("There are no numbers"),
    }
}
```

parity

Rust philosophy

```rust
fn bunch_of_numbers() -> Vec<u32> {
    let mut nums = Vec::with_capacity(10);
    for i in 0..10 {
        nums.push(i);
    }
    nums
}

fn main() {
    let nums = bunch_of_numbers();

    match nums.last() {
        Some(&0) => println!("Last number is zero"),
        Some(n)  => println!("Last number is {}", n),
        None     => println!("There are no numbers"),
    }
}
```

allocation

move

obtain ownership

deallocation

parity

Rust philosophy

```rust
fn bunch_of_numbers() -> Vec<u32> {
    (0..10).collect()
}

fn main() {
    let nums = bunch_of_numbers();

    match nums.last() {
        Some(&0) => println!("Last number is zero"),
        Some(n)  => println!("Last number is {}", n),
        None     => println!("There are no numbers"),
    }
}
```

Rust philosophy

```rust
fn bunch_of_numbers() -> Vec<u32> {
    (0..10).collect()
}

fn main() {
    let nums = bunch_of_numbers();

    match nums.last() {
        Some(&0) => println!("Last number is zero"),
        Some(n)  => println!("Last number is {}", n),
        None     => println!("There are no numbers"),
    }
}
```

allocation + move

obtain ownership

deallocation

parity

Intermission

Questions so far?

Primitives

- Boolean type: **`bool`** (**`true`**, **`false`**)
- Unicode codepoint: **`char`** (4 bytes, '❤️')
- Unsigned integers: **`u8`**, **`u16`**, **`u32`**, **`u64`**, **`u128`**, **`usize`**
- Signed integers: **`i8`**, **`i16`**, **`i32`**, **`i64`**, **`i128`**, **`isize`**
- IEEE floating point numbers: **`f32`**, **`f64`**

Choosing the right number type

- Need floating point? **`f64`**, use **`f32`** for games
- Need a length or index into array? **`usize`**
- Need negative integers? Smallest usable: **`i8`** - **`i128`**
- No negative integers? Smallest usable: **`u8`** - **`u128`**
- Bytes are always **`u8`**
- **`isize`** is rarely used (pointer arithmetic)

BLOCKCHAIN ACHTUNG!

`f64` and `f32` are **verboten!**

They are not deterministic across platforms.

parity

Simple value types

- Array (sized): `[T; N]` eg: `[u8; 5]`
- Tuple: `(T, U, ...)`, eg: `(u8, bool, f64)`
- “Void” tuple: `()`, default return type

parity

Slices

- Similar to arrays, but “unsized” (size unknown to compiler)
- `[T]` eg: `[u8]`, in practice mostly: `&[u8]`
- String slice: `str`, in practice mostly: `&str`

Slices

```
let mut foo = [0u8; 5];
foo[1] = 1;
foo[2] = 2;

let bar = &foo[..3];    // [u8] length of 3
println!("{:?}", bar); // [0, 1, 2]
```

Type inference

```rust
let foo = 10;
let bar: &str = "Hello SubZero";
let baz: &[u8; 13] = b"Hello SubZero";
let tuple: (u8, bool) = (b'0', true);
let heart: char = '❤️';
```

parity

Type inference

```
let foo = 10u32; // Would default to i32
let bar = "Hello SubZero";
let baz = b"Hello SubZero";
let tuple = (b'0', true);
let heart = '❤';
```

parity

Structs

```
struct Foo; // 0-sized
struct Bar(usize, String); // Tuple-like
struct Baz { // With field names
    id: usize,
    name: String, // Owned, growable str
}
```

parity

Structs

```
let baz = Baz {
    id: 42,
    name: "Owned Name".to_owned(),
};
// Access fields by names
println!("Id {} is {}", baz.id, baz.name);
// Id 42 is Owned Name
```

parity

Enums

- Like structs, but value is always one of many *variants*
- Stack size is largest *variant* + *tag*
- Values accessed by pattern matching

Enums

```
enum Animal {
    Cat,
    Dog,
    Fish,
}
let animal = Animal::Dog;
```

parity

Enums

```
enum Number {
    Integer(i64), // Tuple-esque variants
    Float { // Variant with fields
        inner: f64
    },
}
let a = Number::Integer(10);
let b = Number::Float {
    inner: 3.14
};
```

parity

Enums

```
// Match expression
match a {
    Number::Integer(n) => println!("a is integer: {}", n),
    Number::Float { inner } => println!("a is float: {}", inner),
}

// If-let if you want to check for a single variant
if let Number::Float { inner } = b {
    println!("b is float: {}", inner);
}
```

Intermission

Questions so far?

Error handling

- Rust differentiates between *errors* and *panics*
- Errors are explicit, Rust will force you to handle them
- Panics cause thread to shut down unexpectedly and are almost always result of assumptions being violated
- When coding for Substrate your code should never panic, there are tools to check for that

Error handling

- Rust uses two built-in types to handle errors
- **`Option`** is either **`Some(T)`** or **`None`** and replaces **`null`**
- **`Result`** is either **`Ok(T)`** or **`Err(U)`** and replaces what would be exceptions in other languages
- We can propagate errors using the **`?`** operator

Error handling

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

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

parity

Error handling

```
fn add_numbers(numbers: &[i32]) -> i32 {
    let a = numbers[0];
    let b = numbers[1];

    a + b
}
```

parity

Error handling

```
fn add_numbers(numbers: &[i32]) -> i32 {
    let a = numbers[0]; // can panic!
    let b = numbers[1]; // can panic!

    a + b // can panic (debug build)
}         // or do wrapping addition (release build)
```

parity

Error handling

```
fn add_numbers(numbers: &[i32]) -> Option<i32> {
    let a = numbers.get(0)?; // `get` returns Option<i32>
    let b = numbers.get(1)?; // ? will early return on None

    a.checked_add(b) // returns None on overflow
}
```

parity

Error handling

```
fn add_numbers(numbers: &[i32]) -> i32 {
    let a = numbers.get(0).unwrap_or(0); // 0 for None
    let b = numbers.get(1).unwrap_or(0); // 0 for None

    a.saturating_add(b) // Caps to max value on overflow
}
```

Error handling

```rust
use std::io;
use std::fs::File;

fn read_file() -> Result<String, io::Error> {
    let mut file = File::open("./test.txt")?;
    let mut content = String::new();
    file.read_to_string(&mut content)?; // Err early returns
    Ok(content)
}
```

parity

Error handling

```rust
use std::io;
use std::fs::File;

fn read_file() -> io::Result<String> { // Alias type
    let mut file = File::open("./test.txt")?;
    let mut content = String::new();
    file.read_to_string(&mut content)?; // Err early returns
    Ok(content)
}
```

Error handling

```rust
use std::io;
use std::fs::File;

fn read_file() -> Option<String> { // Result to Option
    let mut file = File::open("./test.txt").ok()?;
    let mut content = String::new();
    file.read_to_string(&mut content).ok()?;
    Some(content)
}
```

parity

Intermission

Questions so far?

Implementing methods

- Using **`impl`** keyword
- Can be done for local **`enum`**s and **`struct`**s
- Can have multiple **`impl`** blocks for any type
- Each block can have different trait bounds for generics

Implementing methods

```
struct Duck {
    name: String,
}

impl Duck {
    fn new(name: &str) -> { // Convention, there are no constructors
        Duck { name: name.into() }
    }
    fn quack(&self) { // equates to `self: &Duck`, must be the first argument
        println!("{} quacks!", self.name);
    }
}
```

Implementing methods

```
let bob = Duck::new("Bob"); // associated function

bob.quack();        // automatically borrows
Duck::quack(&bob); // same as above, explicit borrow
```

parity

Implementing methods

```
impl Duck {
    fn borrowing(&self) {
    }
    fn mut_borrowing(&mut self) {
    }
    fn taking_ownership(self) { // drops the instance
    }
}
```

parity

Implementing methods

```
struct Duck<Name> {
    name: Name,
}

impl<Name> Duck<Name> {
    fn new(name: Name) -> {
        Duck { name } // Shorthand syntax, like JS ¯\_(ツ)_/¯
    }
}
```

Implementing methods

```
// Implement for Duck with a Name,
// where the Name implements Display
impl<Name: Display> Duck<Name> {
    fn quack(&self) {
        println!("{} quacks!", self.name);
    }
}
```

Implementing methods

```
// create a Duck<&str>
let bob = Duck::new("Bob");

bob.quack(); // &str implements Display
             // so this is cool
```

parity

Intermission

Questions so far?

Trait system

- Add methods and associated functions to types
- Extremely expressive when combined with generics
- Only active when imported in-scope
- You can implement your traits to external types,
- Or external traits to your types

Trait system

Some built-in traits:

- **`Default`**: create the type with default value
- **`From`**: create a type from another type
- **`Into`**: convert self into another type
- **`Display`**: provide formatting for “pretty” terminal printing
- **`Debug`**: provide formatting for debug terminal printing

Trait system

```
struct Foo(usize);

impl Default for Foo {
    fn default() -> Foo {
        Foo(0)
    }
}
```

parity

Trait system

```
struct Foo(usize);

impl Default for Foo {
    fn default() -> Foo {
        Foo(usize::default()) // Use Default for usize
    }
}
```

parity

Trait system

```
struct Foo(usize);

impl Default for Foo {
    fn default() -> Foo {
        Foo(Default::default()) // Have compiler find impl
    }
}
```

parity

Trait system

```
// Replaces all code from previous slide
#[derive(Default)]
struct Foo(usize);
```

parity

Trait system

```
// From is one of the most useful built-in traits
trait From<Other> {
    // `Other` is generic, `Self` is a special type
    fn from(other: Other) -> Self;

    // No function body, although default implementation
    // could be provided
}
```

parity

Trait system

```
enum Count {
    Zero,
    One,
    Two,
    Many,
}
```

parity

Trait system

```
impl From<u32> for Count {
    fn from(n: u32) -> Count {
        match n {
            0 => Count::Zero,
            1 => Count::One,
            2 => Count::Two,
            _ => Count::Many,
        }
    }
}
```

parity

Trait system

```
use std::io;

struct MyError;

impl From<io::Error> for MyError {
    fn from(err: io::Error) -> MyError {
        MyError
    }
}
```

parity

Trait system

```rust
use std::fs::File;

// ? automatically convert errors!
fn read_file() -> Result<String, MyError> {
    let mut file = File::open("./test.txt")?;
    let mut content = String::new();
    file.read_to_string(&mut content)?;
    Ok(content)
}
```

Trait system

```rust
#[derive(Debug)]
struct Dummy;

// Static dispatch
fn debug<T: Debug>(val: T) {
    println!("{:?}", val);
}

debug(Dummy);
```

parity

Trait system

```rust
#[derive(Debug)]
struct Dummy;

// Dynamic dispatch
fn debug(val: &dyn Dummy) {
    println!("{:?}", val);
}

debug(&Dummy);
```

parity

Trait system

```
trait Duck: Display {
    fn quack(&self) {
        println!("{} quacks!", self);
    }
}

impl Duck for str {}

"Bob".quack();
```

parity

Trait system

```
trait Duck: Display { // trait bound: Self must impl Display
    fn quack(&self) { // default implementation
        println!("{} quacks!", self); // ok because self
    }                                   // impls Display
}

impl Duck for str {} // use default quack method

"Bob".quack(); // Bob quacks!
```

Intermission

Questions so far?

Lifetimes

- Bane of newcomers
- There is no analog in any other mainstream language
- Main source of “fighting the borrow checker”
- Once you grok it, you will be able to write code that in C would equate to magic, with complete confidence!

parity

Lifetimes

- *All* references/borrows (**&**) have a lifetime
- By default those lifetimes are anonymous
- Rust is typically good enough at working with anonymous lifetimes via elision (similar to inference)
- Examples often name lifetimes `'a`, `'b`, `'c`…
- **If you have more than one lifetime, this is a horrible idea!**

Lifetimes

```
impl Duck {
    // Borrow self with anonymous lifetime,
    // return a string slice with anonymous lifetime.
    // Rust will figure out that those are the same.
    fn name(&self) -> &str {
        &self.name
    }
}
```

parity

Lifetimes

```
impl Duck {
    // Borrow self with lifetime 'a,
    // return a string slice with lifetime 'a.
    fn name<'a>(&'a self) -> &'a str {
        &self.name
    }
}
```

parity

Lifetimes

```
impl Duck {
    // Define 'short and 'long lifetimes
    // 'long lifetime has to “outlive” the 'short lifetime
    // Borrow self with lifetime 'long,
    // return a string slice with lifetime 'short.
    fn name<'short, 'long: 'short>(&'long self)
    -> &'short str {
        &self.name
    }
}
```

parity

Lifetimes

There is a special **`'static`** lifetime that can save you a lot of time:

```
// No need to define any lifetimes
// in the struct definition
struct Duck {
    name: &'static str,
}

let bob = Duck { name: "Bob" }; // All literals are 'static
```

Compiler

- *Why doesn’t the stupid compiler let me do my thing?*
- The only way to fight the borrow checker is to realize:
- **YOU CAN’T WIN!**
- Compiler is smarter than you.
- Yes, really.

parity

Compiler

- The compiler is your friend.
- It’s a very good friend. It’s a very honest friend.
- It might be Dutch.
- Eventually writing code that satisfies the compiler becomes second nature.
- As a result, you will become a better programmer.

parity

Questions?

maciej@parity.io
@MaciejHirsz

parity