Rust for Java Developers

Simon Chemouil
@simach

Toulouse JUG, 2019

"The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry."

- Henry Petroski

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01

Energy Efficiency across Programming Languages: How does Energy, Time and Memory Relate?, SLE'17

Outline

1. Data, Composition and Abstraction
   - Ownership and Borrowing
   - Data, Types and Composition
   - Abstraction Using Traits
2. Multithreading and Concurrency
   - Multithreading
   - Concurrency
3. Community, Ecosystem and Future
   - Community and Ecosystem
   - What's Next?

Hello World!

- **10 years on the JVM**
  - Java, Scala, bits of Kotlin
  - Performance, middleware and architecture
- **Before**
  - C++, FP (OCaml)
  - Fundamental CS (PLT, proofs, etc)
- **Now**
  - 1+ year all-in on Rust; full-time Rust developer
  - Toulouse Rust Meetup co-organizer

Three Laws of Informatics

Dimensions that matter

1. **Correction:** spec-compliance, reliability, execution safety
2. **Maintainability:** productivity, scalable design primitives, documentation
3. **Effiency:** runtime speed, memory footprint, power consumption

- **Corollary**: *Ecosystem + Community*

Trade-offs

- **Speed and control at your own risk**
  - C, C++

Trade-offs

- **Speed and control at your own risk**
  - C, C++
- **Execution safety (safe memory management)**
  - Increases reuse, sharing and ecosystem
  - Managed industry languages (Java, C#, Go, JavaScript, . . .)
  - Typed FP languages: increased spec-compliance through types

Trade-offs

- **Speed and control at your own risk**
  - C, C++
- **Execution safety (safe memory management)**
  - Increases reuse, sharing and ecosystem
  - Managed industry languages (Java, C#, Go, JavaScript, . . .)
  - Typed FP languages: increased spec-compliance through types
- **Fast, reliable, productive — pick three**
  - Execution safety without GC - fostering code reuse and sharing
  - Design primitives inspired by typed FP languages
  - Zero-Cost Abstractions

How?

Outline

1. Data, Composition and Abstraction
   - Ownership and Borrowing
   - Data, Types and Composition
   - Abstraction Using Traits
2. Multithreading and Concurrency
   - Multithreading
   - Concurrency
3. Community, Ecosystem and Future
   - Community and Ecosystem
   - What's Next?

Implicit copying

Example

```
// Java
public int getFoo() {
    int bar = 21;
    int foo = bar * 2;
    return foo; // bar is dropped, foo is copied
}

// Rust
pub fn get_foo() -> u32 {
    let bar = 21;
    let foo = bar * 2;
    foo // bar is dropped, foo is copied
}
```

Copying

- **Values everywhere**
  - Held on the stack

Copying

- **Values everywhere**
  - Held on the stack
- **Automatic memory management**
  - Values are dropped when they go out of scope
  - Copied otherwise (could get expensive)

Copying

- **Values everywhere**
  - Held on the stack
- **Automatic memory management**
  - Values are dropped when they go out of scope
  - Copied otherwise (could get expensive)
- **Poor for complex values**
  - State management

Implicit sharing

Example

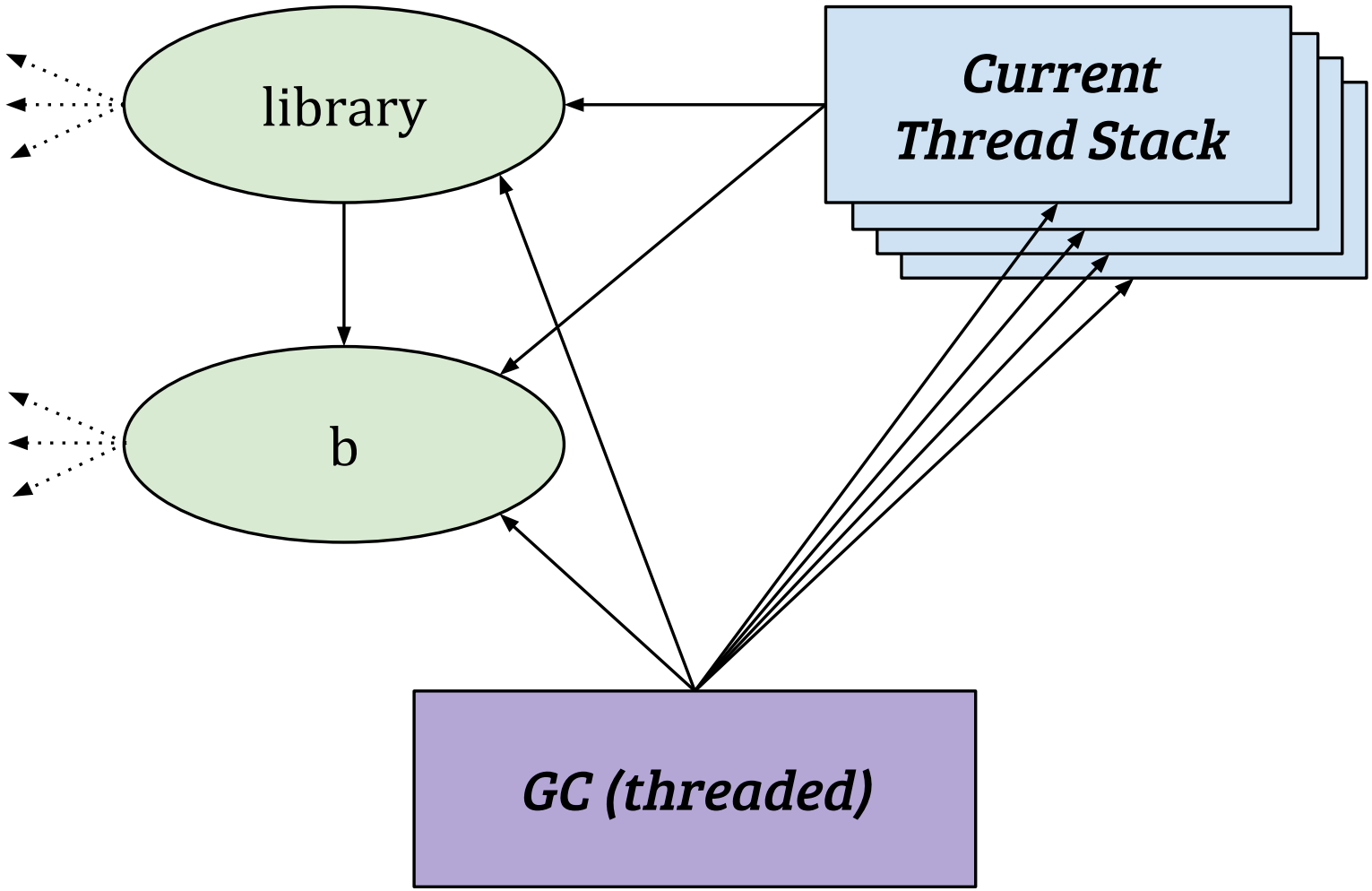
```
public Library init() {
    Library library = new Library(); // (1)
    Book b = new Book("Hyperion"); // (2)
    library.add(b);  // (3)
    // what is `b` and I allowed to mutate it now?
    return library;
}

// Called later
public void removeBook(Library library, String bookName) {
    library.removeByName(bookName)
}
```

Implicit sharing

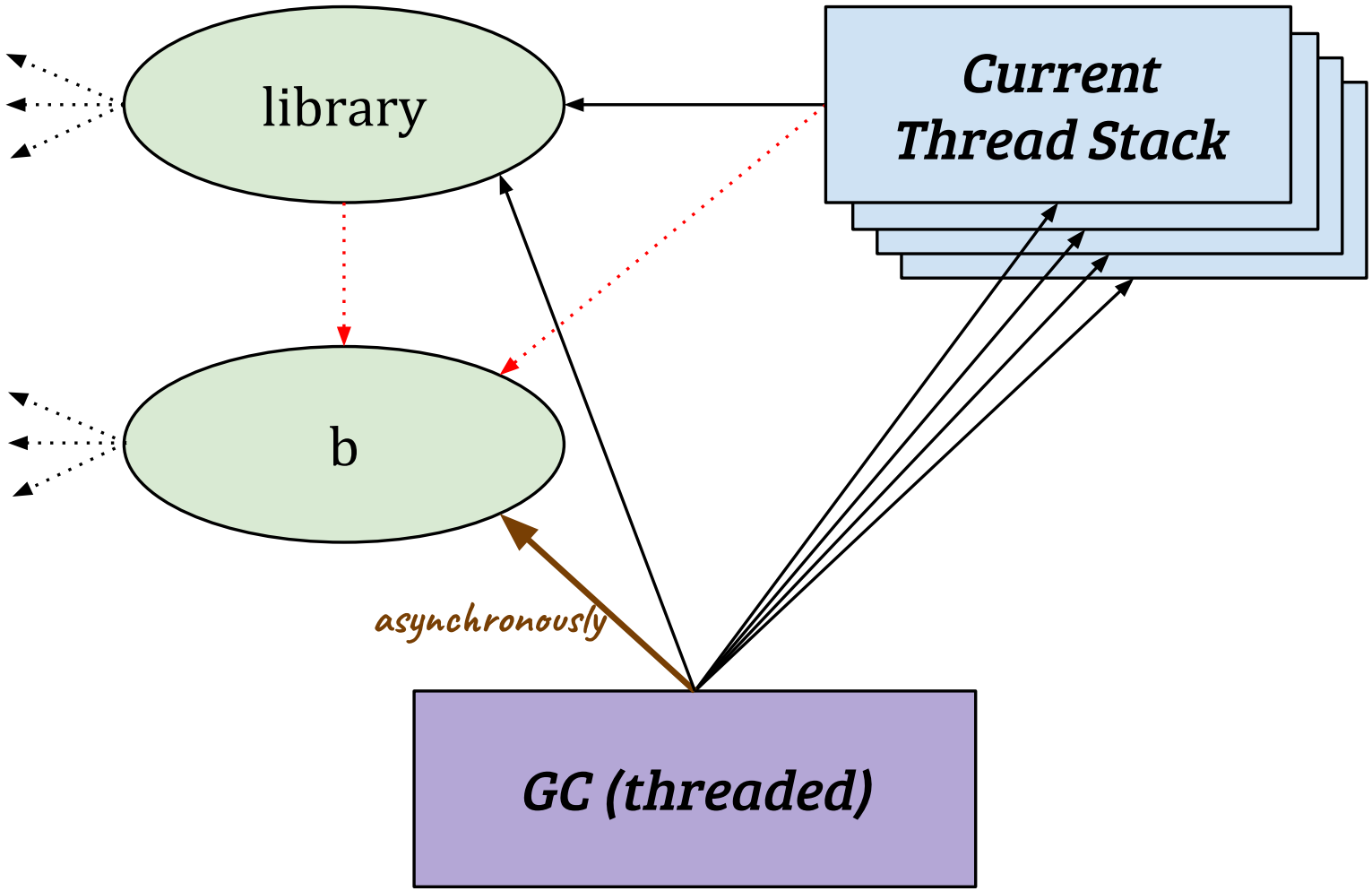
Example

```
public Library init() {
    Library library = new Library(); // (1)
    Book b = new Book("Hyperion"); // (2)
    library.add(b);  // (3)
    b.setName("Endymion"); // (4)
    // Did we just break something?
    // Does ‘library‘ contain a book
    // named "Hyperion" or "Endymion"?
    return library;
}

// Called later
public void removeBook(Library library, String bookName) {
    library.removeByName(bookName)
}
```

Implicit sharing and GC



Implicit sharing and GC



Managed Runtimes

- **Refs everywhere**
  - All structured data ("objects") are only accessible through references (pointers)
  - Allocated on the heap by default (possible escape analysis)

Managed Runtimes

- **Refs everywhere**
  - All structured data ("objects") are only accessible through references (pointers)
  - Allocated on the heap by default (possible escape analysis)
- **Refs can (and must be) shared without restriction**
  - Strong access coupling
  - No mutation control (immutable+builder pattern)
  - Accross threads as well

Managed Runtimes

- **Refs everywhere**
  - All structured data ("objects") are only accessible through references (pointers)
  - Allocated on the heap by default (possible escape analysis)
- **Refs can (and must be) shared without restriction**
  - Strong access coupling
  - No mutation control (immutable+builder pattern)
  - Accross threads as well
- **Memory safety and quick ramp-up**
  - Hard perf tuning, insidious concurrency bugs

Ownership


DUNE
Frank Herbert
ACE SCIENCE FICTION
CHAPTERHOUSE DUNE
FRANK HERBERT
Le seigneur des anneaux
Les deux tours
J.R.R. Tolkien
Power Games 2 · Ruthless.com
Tom Clancy
DUNE MESSIAH
Frank Herbert
Octobre rouge
Tom Clancy
Philip José Farmer
Le monde du fleuve
Le seigneur des anneaux
Le retour du roi
J.R.R. Tolkien
HYPERION
DAN SIMMONS
J.R.R. Tolkien/le Silmarillion-1
J.R.R. Tolkien
Robert Ludlum

Ownership

Example

```
pub fn init() -> Library {
    let mut library = Library::new(); // (1)
    let b = Book::new("Hyperion"); // (2)
    library.insert(b);  // (3)
    // what is `b` and I allowed to mutate it now?
    library
}

// Called later
pub fn remove_book(library: &mut Library, book_name: &str) {
    library.remove_by_name(book_name)
}
```

Use after move

Example

```
pub fn init()-> Library {
    let mut library = Library::new(); // (1)
    let b = Book::new("Hyperion"); // (2)
    library.insert(b);  // (3)
    // what is ‘b‘ and I allowed to mutate it‘ now?
    let name = b.name;
    library
}

// Called later
pub fn remove_book(library: &mut Library, book_name: &str) {
    library.remove_by_name(book_name)
}
```

Use after move

Example

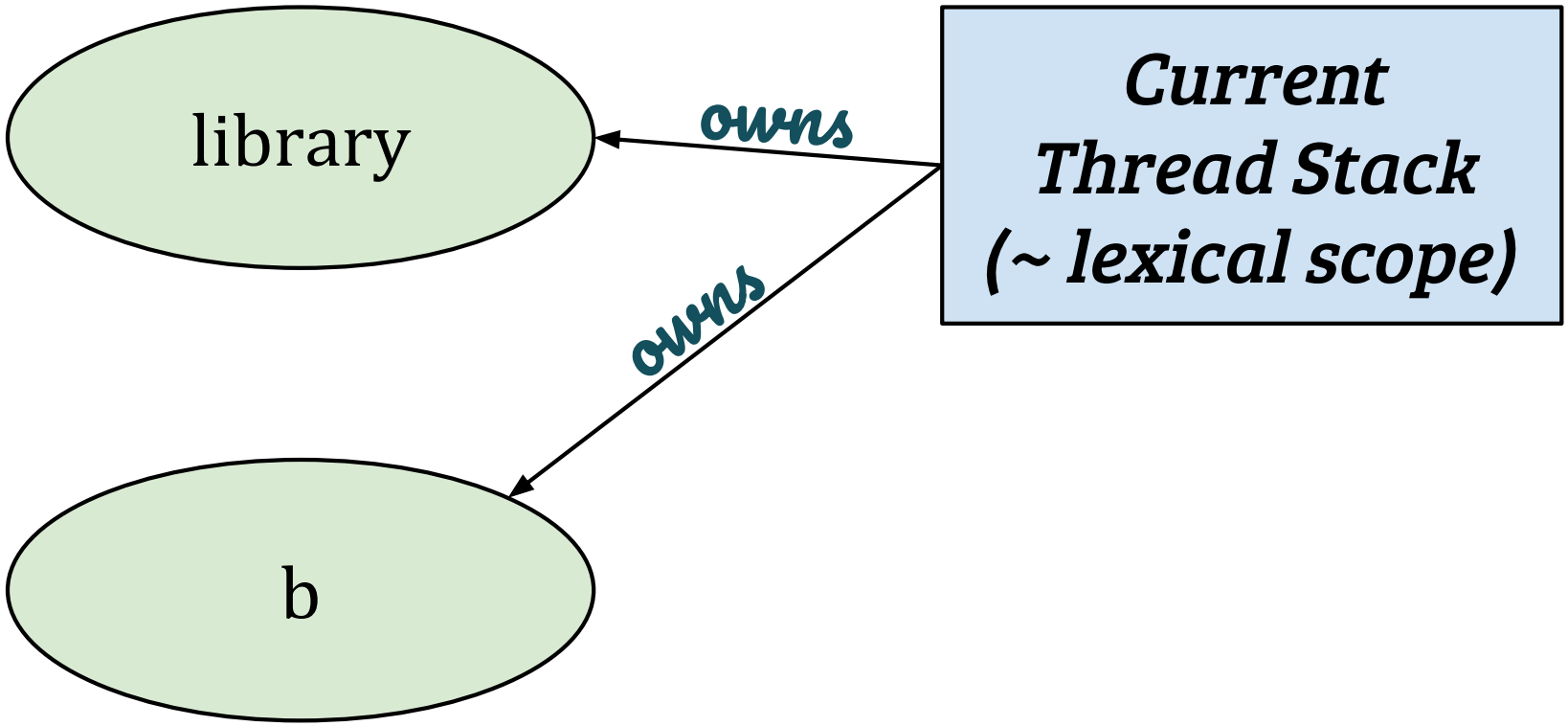
```
33 |     let b = Book::new("Hyperion"); // (2)
   |         - move occurs because `b` has type `Book`,
   |           which does not implement the `Copy` trait
34 |     library.insert(b); // (3)
   |                    - value moved here ...
37 |     let name = b.name;
   |                ^^^^^^ value used here after move
```

Ownership, not magic

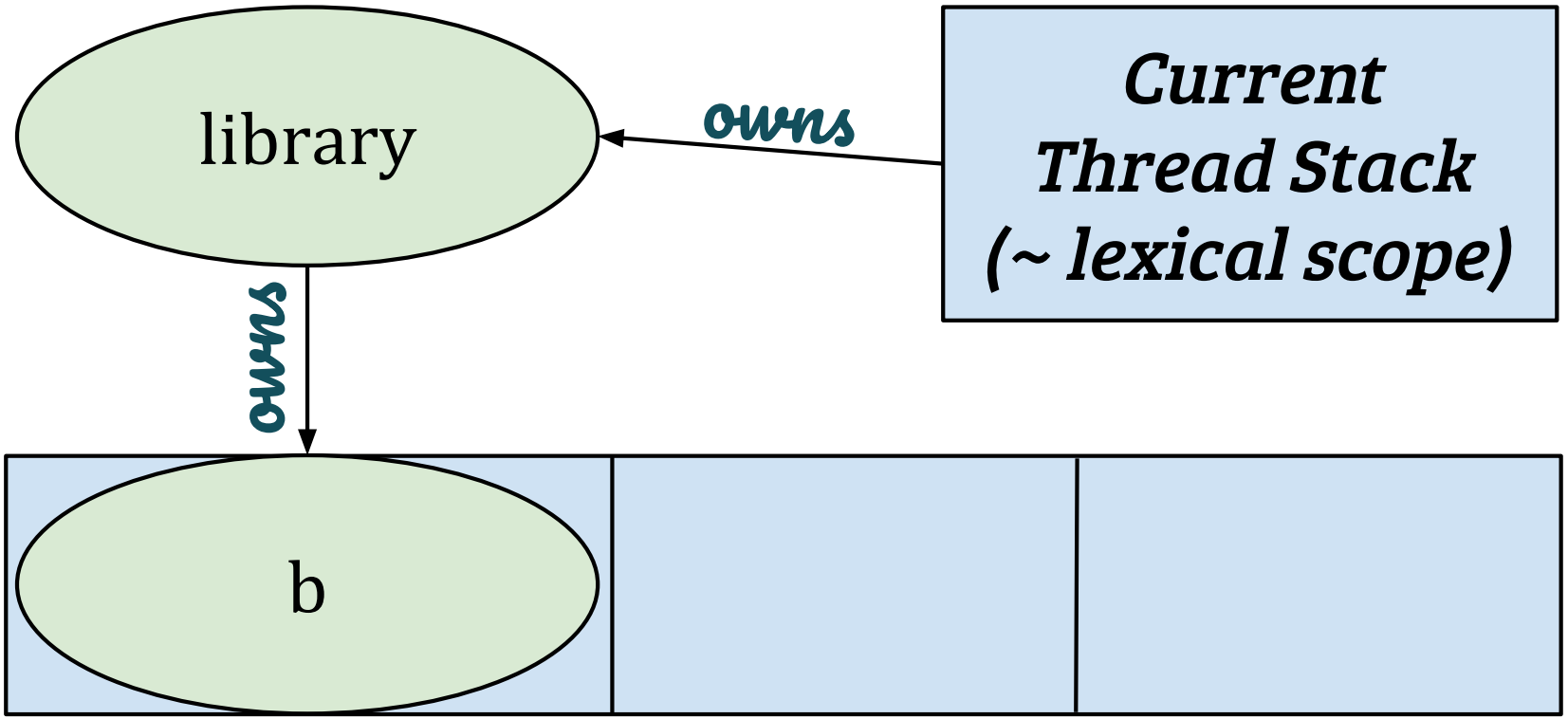
Example

```
#[derive(Debug, Default, Eq, PartialEq)]
pub struct Library {
    books: Vec<Book>,
}

impl Library {
    pub fn new() -> Self { Default::default() }
    pub fn insert(&mut self, book: Book) { self.books.push(book); }
    // parameters: mutable borrow (`self`), shared borrow (`book_name`)
    pub fn remove_by_name(&mut self, book_name: &str) {
        // No `ConcurrentModificationException` possible!
        self.books.retain(|b| b.name != book_name);
    }
}

#[derive(Debug, Eq, PartialEq)]
pub struct Book {
    name: String,
}

impl Book {
    pub fn new(name: impl Into<String>) -> Self { Book { name: name.into() } }
}
```

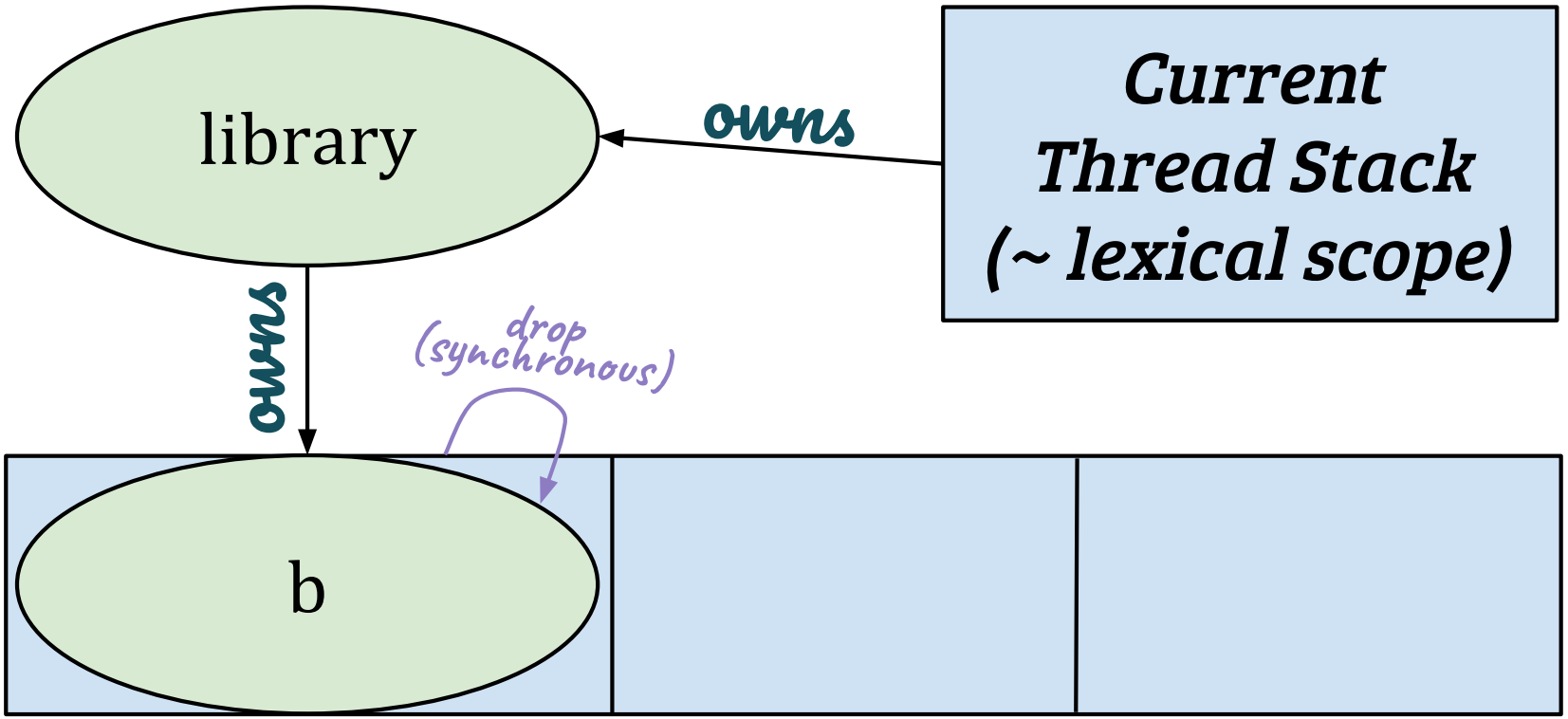
Ownership



Moving ownership



Dropping


library
owns
Current
Thread Stack
(~ lexical scope)
owns
drop
(synchronous)
b

Ownership

- **Always exactly one owner**
  - Matches the lexical scope, spot owner easily
  - Memory is freed when the owner drops the value
  - Stack by default (as long as the data is *Sized*), can be refs

Ownership

- **Always exactly one owner**
  - Matches the lexical scope, spot owner easily
  - Memory is freed when the owner drops the value
  - Stack by default (as long as the data is *Sized*), can be refs
- **Change owner by *moving* value**
  - Regular assignment or passing to a function
  - Generally very fast (on stack)
  - Large values can be costly (allocate and move the smart pointer)

Ownership

- **Always exactly one owner**
  - Matches the lexical scope, spot owner easily
  - Memory is freed when the owner drops the value
  - Stack by default (as long as the data is *Sized*), can be refs
- **Change owner by *moving* value**
  - Regular assignment or passing to a function
  - Generally very fast (on stack)
  - Large values can be costly (allocate and move the smart pointer)
- **Memory safe and *natural* to reason with**
  - Deterministic behavior, stronger invariants

Borrowing

- **Values can be *borrowed***
  - Borrowing gives a *safe* reference that is cheap to pass around
  - Loose-coupling because sharing is explicit
  - Shared vs Exclusive access (Readers / Writers)
  - Borrow can only live as long as the owner

Borrowing

- **Values can be *borrowed***
  - Borrowing gives a *safe* reference that is cheap to pass around
  - Loose-coupling because sharing is explicit
  - Shared vs Exclusive access (Readers / Writers)
  - Borrow can only live as long as the owner
- **Shared borrowing**
  - Several readers, immutable by default

Borrowing

- **Values can be *borrowed***
  - Borrowing gives a *safe* reference that is cheap to pass around
  - Loose-coupling because sharing is explicit
  - Shared vs Exclusive access (Readers / Writers)
  - Borrow can only live as long as the owner
- **Shared borrowing**
  - Several readers, immutable by default
- **Mutable / exclusive borrowing**
  - Always exactly **one** reader/writer, no exceptions

Borrowing (live coding)

Example

```
impl Library {
    pub fn get_book_by_name(&self, book_name: &str) -> Option<&Book> {
        self.books.iter().find(|b| b.name == book_name)
    }

    pub fn get_book_mut_by_name(&mut self, book_name: &str) -> Option<&mut Book> {
        self.books.iter_mut().find(|b| b.name == book_name)
    }
}
```

Borrowing limitations

- **Borrows can only live as long as their owner**
  - That's the *lifetime* of a borrow
  - Known statically and proven by the compiler
  - **Fits 90+% of the use cases**

Borrowing limitations

- **Borrows can only live as long as their owner**
  - That's the *lifetime* of a borrow
  - Known statically and proven by the compiler
  - **Fits 90+% of the use cases**
- **Dynamic lifetimes?**
  - We don't know statically own long a value will live (e.g caches)
  - *Shared ownership* similar to GC through reference counting (library constructs, **Rc** and **Arc**)

Borrowing limitations

- **Borrows can only live as long as their owner**
  - That's the *lifetime* of a borrow
  - Known statically and proven by the compiler
  - **Fits 90+% of the use cases**
- **Dynamic lifetimes?**
  - We don't know statically own long a value will live (e.g caches)
  - *Shared ownership* similar to GC through reference counting (library constructs, **Rc** and **Arc**)
- **When reference counting isn't enough?**
  - 1% use cases: cyclic data, high-allocation patterns, ...
  - Other approaches, often packaged libs, e.g for graphs, arenas, ...

Ownership and Borrowing

- **Safer and more expressive**
  - You can model how values can be consumed (how *and* when)
  - Stronger invariants and loose-coupling

Ownership and Borrowing

- **Safer and more expressive**
  - You can model how values can be consumed (how *and* when)
  - Stronger invariants and loose-coupling
- **Natural and ergonomic**
  - A physical take on data
  - *Shared ownership* through reference counting (library constructs, **Rc** and **Arc**)

Ownership and Borrowing

- **Safer and more expressive**
  - You can model how values can be consumed (how *and* when)
  - Stronger invariants and loose-coupling
- **Natural and ergonomic**
  - A physical take on data
  - *Shared ownership* through reference counting (library constructs, **Rc** and **Arc**)
- **Enabler for more**
  - Data race freedom, resource management, ...
  - Native performance

Outline

1. Data, Composition and Abstraction
   - Ownership and Borrowing
   - Data, Types and Composition
   - Abstraction Using Traits
2. Multithreading and Concurrency
   - Multithreading
   - Concurrency
3. Community, Ecosystem and Future
   - Community and Ecosystem
   - What's Next?

Enums and Generics

Example

```
// No null!
pub enum Option<T> {
    None,
    Some(T),
}

// No exceptions!
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Error-handling + RAII

Example

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

pub fn file_to_upper() -> Result<String, std::io::Error> {
    let mut file = File::open("foo.txt")?;  // (1)
    let mut contents = String::new();  // (2)
    file.read_to_string(&mut contents)?;  // (3)
    Ok(contents.to_uppercase())  // (4)
}
```

Types

- **struct** and **enum**
  - Accurate modelling and built-in pattern matching
- **Generics**
  - *Monomorphized* at compile time (no boxing!)
- **A bit more**
  - References, lifetimes, trait objects

Composition Over Inheritance

Example

```
pub enum Origin {
    Gift(Gifter),
    Bought(Shop)
}

pub struct Dated<T> {
    pub date: Date<Utc>,
    pub item: T,
}

pub type DatedLibrary = Vec<Dated<(Book, Origin)>>;
```

Outline

1. Data, Composition and Abstraction
   - Ownership and Borrowing
   - Data, Types and Composition
   - Abstraction Using Traits
2. Multithreading and Concurrency
   - Multithreading
   - Concurrency
3. Community, Ecosystem and Future
   - Community and Ecosystem
   - What's Next?

Interfaces vs Traits

Example

```
// Java
public interface Named {
    String getName();
}

// Rust
pub trait GetName {
    fn get_name(&self) -> &str;
}
```

Interfaces vs Traits

Example

```
// Java
public class Book implements Named {
    public String getName() { return this.name; }
    ...
}

// Rust
impl GetName for Book {
    fn get_name(&self) -> &str { &self.name }
}

// Because why not?
impl GetName for String {
    fn get_name(&self) -> &str { &self }
}

impl<T: GetName> GetName for Option<T> {
    fn get_name(&self) -> &str {
        match self {
            Some(s) => s.get_name(),
            None => "<none>",
        }
    }
}
```

Interfaces vs Traits

Example

```
// (1)
pub fn print_names(names: &[impl GetName]) {
    for (i, named) in names.iter().enumerate() {
        println!("{}. {}", i, named.get_name());
    }
}

// (2)
let books = vec![Book::new("Hyperion"), Book::new("Dune")];
print_names(&books);

// (3)
let v_str = vec!["Foo", "Bar"];
print_names(&v_str);

// (4)
// let v_mixed = vec!["Foo", Book::new("Hyperion")];
// expected &str, found struct `Book`

// (5)
let v_mixed: Vec<Box<dyn GetName>> = vec![Box::new("Foo"),
                                          Box::new(Book::new("Hyperion"))];
print_names(&v_mixed);
```

Interfaces vs Traits

- **Traits can do more**
  - Similar to Haskell's type classes
  - Implement trait on existing types, including quantified types

Interfaces vs Traits

- **Traits can do more**
  - Similar to Haskell's type classes
  - Implement trait on existing types, including quantified types
- **Zero-Cost Abstractions or Dynamic Dispatch**
  - Trait as a compile time concept, or as a type for virtual tables
  - Same abstractions, pick your trade-off

Interfaces vs Traits

- **Traits can do more**
  - Similar to Haskell's type classes
  - Implement trait on existing types, including quantified types
- **Zero-Cost Abstractions or Dynamic Dispatch**
  - Trait as a compile time concept, or as a type for virtual tables
  - Same abstractions, pick your trade-off
- **A bit more**
  - Associated types, trait constants, automatic derivation using macros, ...

Outline

1. Data, Composition and Abstraction
   - Ownership and Borrowing
   - Data, Types and Composition
   - Abstraction Using Traits
2. Multithreading and Concurrency
   - Multithreading
   - Concurrency
3. Community, Ecosystem and Future
   - Community and Ecosystem
   - What's Next?

Multithreading in Java

Example

```
class Server {
    public Library library = new Library();

    // Coming from a multithreaded REST server...
    public void newBookRequest(Book book) {
        this.library.add(book);
    }
}

public class Library {
    private List<Book> books = new ArrayList<>(); // /!\

    public void add(Book b) { this.books.add(b); }
}
```

Multithreading in Rust, from naive

Example

```
let mut library = init();

for i in 0..5 {
    let b = Book::new(format!("Book_{}", i));
    library.insert(b);
}
```

Multithreading in Rust, first try

Example

```
let mut library = init();

for i in 0..5 {
    // cannot borrow ‘library‘ as mutable more than once at a time
    thread::spawn(|| {
        let b = Book::new(format!("Book_{}", i));
        library.insert(b);
    });
}
```

Multithreading in Rust, no loop?

Example

```
let mut library = init();

// closure may outlive the current function, but it borrows `library`,
// which is owned by the current function
thread::spawn(|| {
    let b = Book::new(format!("Book"));
    library.insert(b);
});
```

Multithreading in Rust, almost

Example

```
// Arc<Library>
let library = Arc::new(init());

for i in 0..5 {
    let library = Arc::clone(&library);
    thread::spawn(move || {
        let b = Book::new(format!("Book_{}", i));
        // error: cannot borrow data in an ‘Arc‘ as mutable
        library.insert(b);
    });
}
```

Multithreading in Rust, working

Example

```
// Arc<Mutex<Library>>
let library = Arc::new(Mutex::new(init()));

for i in 0..5 {
    let library = Arc::clone(&library);
    thread::spawn(move || {
        let b = Book::new(format!("Book_{}", i));
        library.lock().insert(b);
    });
}
```

Multithreading in Rust, ergonomics

Example

```
let library = SyncLibrary::from(init());

for i in 0..5 {
    let library = library.clone();
    thread::spawn(move || {
        let b = Book::new(format!("Book_{}", i));
        library.insert(b);
    });
}
```

Multithreading in Rust, ergonomics

Example

```
#[derive(Default, Debug, Clone)]
pub struct SyncLibrary {
    inner: Arc<Mutex<Library>>,
}

impl SyncLibrary {
    pub fn insert(&self, book: Book) {
        self.inner.lock().insert(book);
    }
}

impl From<Library> for SyncLibrary {
    fn from(library: Library) -> SyncLibrary {
        SyncLibrary {
            inner: Arc::new(Mutex::new(library)),
        }
    }
}
```

Multithreading in Rust

- **Compile time guarantees**
  - Sharing, Mutation and No Ordering: pick two!
  - Data-race freedom through the type and ownership system

Multithreading in Rust

- **Compile time guarantees**
  - Sharing, Mutation and No Ordering: pick two!
  - Data-race freedom through the type and ownership system
- **High ergonomics through composition and RAII**
  - *Mutex + MutexGuard, Channels, ...*

Multithreading in Rust

- **Compile time guarantees**
  - Sharing, Mutation and No Ordering: pick two!
  - Data-race freedom through the type and ownership system
- **High ergonomics through composition and RAII**
  - *Mutex* + *MutexGuard*, *Channels*, ...
- **A bit more**
  - *Send* and *Sync* traits
  - Similar to ownership and borrowing

Outline

1. Data, Composition and Abstraction
   - Ownership and Borrowing
   - Data, Types and Composition
   - Abstraction Using Traits
2. Multithreading and Concurrency
   - Multithreading
   - Concurrency
3. Community, Ecosystem and Future
   - Community and Ecosystem
   - What's Next?

Concurrency in Rust

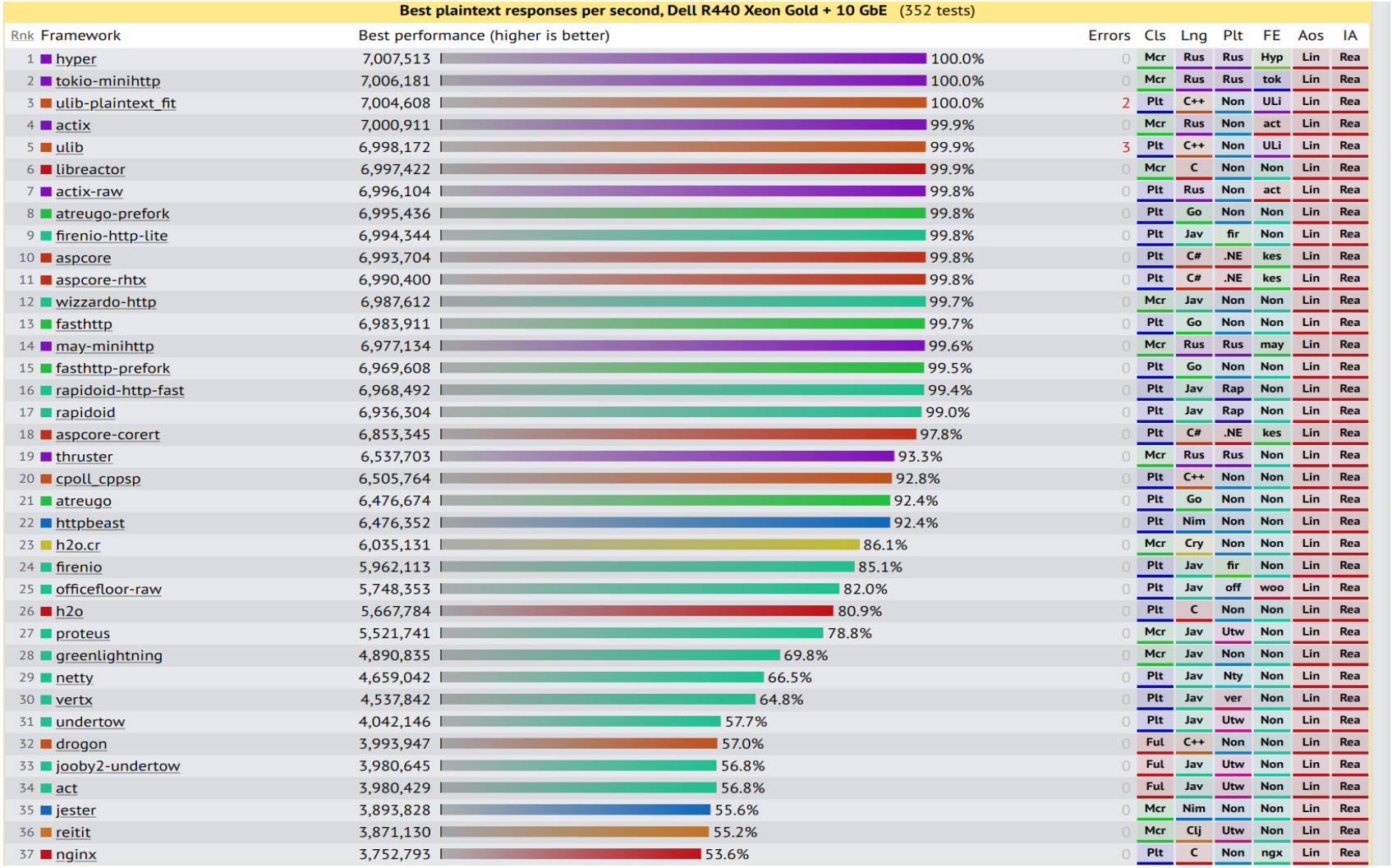
- **Back-pressure aware, non-blocking concurrency**
  - Using async / await in Rust
  - Concurrency on single thread or threadpool
- **Sequential code ergonomics**
  - Same strong guarantees as before
  - No mutation + async problem anymore (Actors?)
- **Zero-Cost Futures and Streams**
  - Compiled-down to a state machine, allocated once
  - Poll+Waker model

An echo HTTP handler in Rust

Example

```
async fn echo(req: Request<Body>) -> Result<Response<Body>, hyper::Error> {
    match (req.method(), req.uri().path()) {
        // Simply echo the body back to the client.
        (&Method::POST, "/echo") => {
            Ok(Response::new(req.into_body()))
        }

        (&Method::POST, "/echo/reversed") => {
            let whole_chunk = req.into_body().try_concat().await?;
            let reversed_chunk = whole_chunk.iter()
                                            .rev().cloned().collect::<Vec<u8>>();
            Ok(Response::new(Body::from(reversed_chunk)))
        }

        // Return the 404 Not Found for other routes.
        _ => {
            let mut not_found = Response::default();
            *not_found.status_mut() = StatusCode::NOT_FOUND;
            Ok(not_found)
        }
    }
}
```

That HTTP server. . .



Best plaintext responses per second, Dell R440 Xeon Gold + 10 GbE (352 tests)

Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	IA
1	hyper	7,007,513	100.0%	0	Mcr	Rus	Rus	Hyp	Lin	Rea
2	tokio-minihttp	7,006,181	100.0%	0	Mcr	Rus	Rus	tok	Lin	Rea
3	ulib-plaintext_fit	7,004,608	100.0%	2	Plt	C++	Non	ULi	Lin	Rea
4	actix	7,000,911	99.9%	0	Mcr	Rus	Non	act	Lin	Rea
5	ulib	6,998,172	99.9%	3	Plt	C++	Non	ULi	Lin	Rea
6	libreactor	6,997,422	99.9%	0	Mcr	C	Non	Non	Lin	Rea
7	actix-raw	6,996,104	99.8%	0	Plt	Rus	Non	act	Lin	Rea
8	atreugo-prefork	6,995,436	99.8%	0	Plt	Go	Non	Non	Lin	Rea
9	firenio-http-lite	6,994,344	99.8%	0	Plt	Jav	fir	Non	Lin	Rea
10	aspcore	6,993,704	99.8%	0	Plt	C#	.NE	kes	Lin	Rea
11	aspcore-rhtx	6,990,400	99.8%	0	Plt	C#	.NE	kes	Lin	Rea
12	wizzardo-http	6,987,612	99.7%	0	Mcr	Jav	Non	Non	Lin	Rea
13	fasthttp	6,983,911	99.7%	0	Plt	Go	Non	Non	Lin	Rea
14	may-minihttp	6,977,134	99.6%	0	Mcr	Rus	Rus	may	Lin	Rea
15	fasthttp-prefork	6,969,608	99.5%	0	Plt	Go	Non	Non	Lin	Rea
16	rapidoid-http-fast	6,968,492	99.4%	0	Plt	Jav	Rap	Non	Lin	Rea
17	rapidoid	6,936,304	99.0%	0	Plt	Jav	Rap	Non	Lin	Rea
18	aspcore-corert	6,853,345	97.8%	0	Plt	C#	.NE	kes	Lin	Rea
19	thruster	6,537,703	93.3%	0	Mcr	Rus	Rus	Non	Lin	Rea
20	cpoll_cppsp	6,505,764	92.8%	0	Plt	C++	Non	Non	Lin	Rea
21	atreugo	6,476,674	92.4%	0	Plt	Go	Non	Non	Lin	Rea
22	httpbeast	6,476,352	92.4%	0	Plt	Nim	Non	Non	Lin	Rea
23	h2o.cr	6,035,131	86.1%	0	Mcr	Cry	Non	Non	Lin	Rea
24	firenio	5,962,113	85.1%	0	Plt	Jav	fir	Non	Lin	Rea
25	officefloor-raw	5,748,353	82.0%	0	Plt	Jav	off	woo	Lin	Rea
26	h2o	5,667,784	80.9%	0	Plt	C	Non	Non	Lin	Rea
27	proteus	5,521,741	78.8%	0	Mcr	Jav	Utw	Non	Lin	Rea
28	greenlightning	4,890,835	69.8%	0	Mcr	Jav	Non	Non	Lin	Rea
29	netty	4,659,042	66.5%	0	Plt	Jav	Nty	Non	Lin	Rea
30	vertx	4,537,842	64.8%	0	Plt	Jav	ver	Non	Lin	Rea
31	undertow	4,042,146	57.7%	0	Plt	Jav	Utw	Non	Lin	Rea
32	drogon	3,993,947	57.0%	0	Ful	C++	Non	Non	Lin	Rea
33	jooby2-undertow	3,980,645	56.8%	0	Ful	Jav	Utw	Non	Lin	Rea
34	act	3,980,429	56.8%	0	Ful	Jav	Utw	Non	Lin	Rea
35	jester	3,893,828	55.6%	0	Mcr	Nim	Non	Non	Lin	Rea
36	reitit	3,871,130	55.2%	0	Mcr	Clj	Utw	Non	Lin	Rea
37	nginx	3,752,793	53.6%	0	Plt	C	Non	ngx	Lin	Rea

That HTTP server. . .

68	jetty	1,537,352	21.9%	0	Plt	Jav	Jty	Non	Lin	Rea
69	giraffe	1,532,734	21.9%	0	Ful	f#	.NE	kes	Lin	Rea
70	warp-hasql	1,521,044	21.7%	0	Mcr	Hkl	Wai	wai	Lin	Rea
71	jlhttp	1,448,316	20.7%	0	Plt	Jav	JLH	Non	Lin	Rea
72	tio-mvc	1,391,416	19.9%	0	Mcr	Jav	t-i	Non	Lin	Rea
73	hexagon-resin-postgresql	1,361,016	19.4%	0	Mcr	Kot	Svt	Non	Lin	Rea
74	hexagon-resin-mongodb	1,352,995	19.3%	0	Mcr	Kot	Svt	Non	Lin	Rea
75	gemini	1,345,068	19.2%	0	Ful	Jav	Svt	Res	Lin	Rea
76	h2o_mruby	1,311,907	18.7%	0	Plt	Rby	Non	h2o	Lin	Rea
77	jooby2-jetty	1,272,306	18.2%	0	Ful	Jav	Jty	Non	Lin	Rea
78	aspcore-mvc	1,255,837	17.9%	0	Ful	C#	.NE	kes	Lin	Rea
79	amber	1,232,049	17.6%	0	Ful	Cry	Non	Non	Lin	Rea
80	facil.io	1,221,373	17.4%	0	Mcr	C	Non	Non	Lin	Rea
81	vibed-ldc	1,168,920	16.7%	0	Plt	D	Non	Non	Lin	Rea
82	vibed-ldc-pgsql	1,148,524	16.4%	0	Plt	D	Non	Non	Lin	Rea
83	workerman	1,125,572	16.1%	0	Plt	PHP	Non	wor	Lin	Rea
84	ratpack	1,107,946	15.8%	0	Mcr	Jav	Nty	Non	Lin	Rea
85	finatra	1,053,298	15.0%	0	Mcr	Sca	Nty	Non	Lin	Rea
86	swift-nio	1,010,416	14.4%	0	Plt	Swi	Non	Non	Lin	Rea
87	nodejs	997,327	14.2%	0	Plt	JS	njs	Non	Lin	Rea
88	nodejs-chakra	994,225	14.2%	0	Plt	JS	njs	Non	Lin	Rea
89	finch	993,490	14.2%	0	Mcr	Sca	Nty	Non	Lin	Rea
90	servlet3-sync	987,384	14.1%	0	Plt	Jav	Svt	tom	Lin	Rea
91	finagle	975,009	13.9%	0	Mcr	Sca	Nty	Non	Lin	Rea
92	kemal	967,739	13.8%	0	Ful	Cry	Non	Non	Lin	Rea
93	akka-http	938,771	13.4%	0	Mcr	Sca	Akk	Non	Lin	Rea
94	gramework	920,600	13.1%	0	Plt	Go	Non	Non	Lin	Rea
95	go-pgx-prefork	913,917	13.0%	0	Plt	Go	Non	Non	Lin	Rea
96	jetty-servlet	905,369	12.9%	0	Plt	Jav	Jty	Non	Lin	Rea
97	play2-scala-netty	865,420	12.3%	0	Ful	Sca	Nty	Non	Lin	Rea
98	polkadot	864,834	12.3%	0	Plt	JS	Non	Non	Lin	Rea
99	fastify	858,416	12.2%	0	Mcr	JS	Non	Non	Lin	Rea
100	ktor-reactivepg	839,050	12.0%	0	Ful	Kot	Non	Non	Lin	Rea
101	vibed-dmd-pgsql	829,320	11.8%	0	Plt	D	Non	Non	Lin	Rea
102	go	817,743	11.7%	0	Plt	Go	Non	Non	Lin	Rea
103	gin	815,866	11.6%	0	Mcr	Go	Non	Non	Lin	Rea
104	ktor	815,432	11.6%	0	Mcr	Kot	Nty	Non	Lin	Rea

Outline

1. Data, Composition and Abstraction
   - Ownership and Borrowing
   - Data, Types and Composition
   - Abstraction Using Traits
2. Multithreading and Concurrency
   - Multithreading
   - Concurrency
3. Community, Ecosystem and Future
   - Community and Ecosystem
   - What's Next?

Built For Reuse

- **Cargo**
  - Project and dependency management
  - Semantic versioning, multiple versions supported
  - crates.io
- **Precise semantics and strong invariants**
  - Hard to misuse libraries, usually get a compile error instead
  - Hard to cut corners, well-designed crates
- **Very strong community**
  - A dynamic that reminds of Java's better days
  - No "Us vs Them" split as in Core JVM vs Java devs

Crates We Love

- **Serde**
  - Serialization and deserialization done right
- **Tokio**
  - Future-based framework
- **Macro crates**
  - Clap (now with Structopts inside!)
  - Many more

Serialization and Deserialization (live demo)

Example

```
#[derive(Debug, Eq, PartialEq, Serialize, Deserialize)]
pub struct Book {
    name: String,
    author: Author
}

#[derive(Debug, Eq, PartialEq, Serialize, Deserialize)]
pub struct Author {
    first_name: String,
    last_name: String
}
```

Parsing Command-Line Arguments

Example

```
/// A basic example
#[derive(StructOpt, Debug)]
#[structopt(name = "basic")]
struct Opt {
    /// Activate debug mode
    #[structopt(short, long)]
    debug: bool,

    /// Verbose mode (-v, -vv, -vvv, etc.)
    #[structopt(short, long, parse(from_occurrences))]
    verbose: u8,
}

fn main() {
    let opt = Opt::from_args();
    println!("{:#?}", opt);
}
```

Parsing Command-Line Arguments

Example

```
$ ./my-program --help
basic 0.1.0
A basic example

USAGE:
    clap-demo [FLAGS]

FLAGS:
    -d, --debug       Activate debug mode
    -h, --help        Prints help information
    -V, --version     Prints version information
    -v, --verbose     Verbose mode (-v, -vv, -vvv, etc.)
```

Notable Users

- **Mozilla**
  - Rust was born out of Firefox yak shaving
- **Google Fuchsia**
  - Google's Next-Gen OS
- **AWS**
  - Firecracker, the AWS Lambda runtime
- **CloudFlare**
  - 1.1.1.1 VPN, QUIC/HTTP3 services, ...

Outline

1. Data, Composition and Abstraction
   - Ownership and Borrowing
   - Data, Types and Composition
   - Abstraction Using Traits
2. Multithreading and Concurrency
   - Multithreading
   - Concurrency
3. Community, Ecosystem and Future
   - Community and Ecosystem
   - What's Next?

WebAssembly

- **#1 source language for WASM**
  - Supported out-of-the box
  - Lack of runtime = small modules
- **WebApps and Serverless**
  - "Universal / Isomorphic" apps in Rust
  - Some React-like frameworks and serverless platforms
- **Used for sandboxing code**
  - Write Once, Run Everywhere!

Maturity

- **Rust 2021**
  - Implement last few missing pieces
  - Specialization, GATs, ...
- **Even better tooling**
  - On-demand compiler infrastructure
  - Strong IDE support
- **It's time to get involved! :-)**

Conclusion

- **Lower-level *and* higher-level than Java**
  - Learned from C++, Java, Scala, Haskell, Ruby... to propose something new
  - Very robust solutions

Conclusion

- **Lower-level *and* higher-level than Java**
  - Learned from C++, Java, Scala, Haskell, Ruby... to propose something new
  - Very robust solutions
- **Easy to write applications, harder to write libraries**
  - Read the *Rust Book*
  - Practice makes perfect, ***very productive*** language

Conclusion

- **Lower-level *and* higher-level than Java**
  - Learned from C++, Java, Scala, Haskell, Ruby... to propose something new
  - Very robust solutions
- **Easy to write applications, harder to write libraries**
  - Read the *Rust Book*
  - Practice makes perfect, ***very productive*** language
- **Write Rust and...**
  - Stop shipping bugs or insecure software
  - **Save the planet! :-)**

Questions...

Questions?

Twitter:
@simach

See you at the Toulouse Rust Meetup!