

Rust Trainings All in One

- High-level intro about Rust
- Ownership, borrow check, and lifetime
- Typesystem and data structures
- Concurrency - primitives
- Concurrency - async/await
- Networking and security
- FFI with C/Elixir/Swift/Java
- WASM/WASI
- Rust for real-world problems

High-level Intro About Rust

Why Rust?

Let's talk about values and tradeoffs first

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoperability
- Integrity
- Maintainability
- Measurability
- Operability
- Performance
- Portability
- Productivity
- Resiliency
- Rigor
- Safety
- Security
- Simplicity
- Stability
- Thoroughness
- Transparent
- Velocity

C

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoperability

- Integrity
- Maintainability
- Measurability
- Operability
- **Performance**
- Portability
- Productivity
- Resiliency

- Rigor
- Safety
- Security
- **Simplicity**
- Stability
- Thoroughness
- **Transparent**
- Velocity

Erlang/Elixir

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoperability
- Integrity
- Maintainability
- Measurability
- Operability
- Performance
- Portability
- **Productivity**
- **Resiliency**
- Rigor
- **Safety**
- Security
- **Simplicity**
- Stability
- Thoroughness
- Transparent
- Velocity

Python

- **Approachability**
- Availability
- Compatibility
- Composability
- Debuggability
- **Expressiveness**
- Extensibility
- Interoperability
- Integrity
- Maintainability
- Measurability
- Operability
- Performance
- Portability
- **Productivity**
- Resiliency
- Rigor
- Safety
- Security
- **Simplicity**
- Stability
- Thoroughness
- Transparent
- Velocity

Java (in early days)

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoperability
- Integrity
- Maintainability
- Measurability
- Operability
- **Performance**
- **Portability**
- Productivity
- Resiliency
- Rigor
- **Safety (memory)**
- **Security**
- Simplicity
- Stability
- Thoroughness
- Transparent
- Velocity

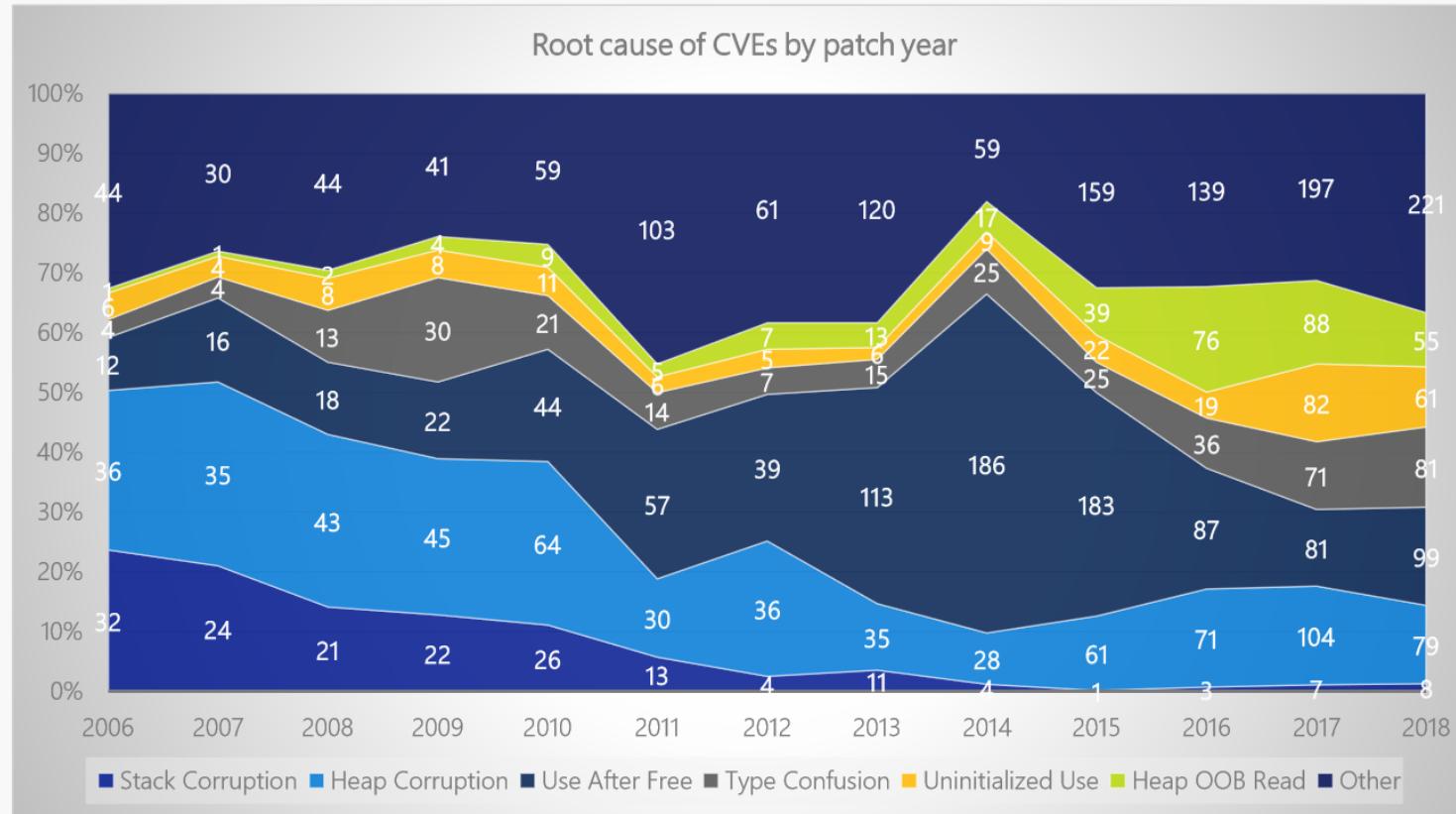
Rust

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- **Expressiveness**
- Extensibility
- Interoperability
- Integrity
- Maintainability
- Measurability
- Operability
- **Performance**
- Portability
- **Productivity**
- Resiliency
- Rigor
- **Safety!!!**
- Security
- Simplicity
- Stability
- Thoroughness
- Transparent
- Velocity



Why safety is important?

Drilling down into root causes



Stack corruptions are essentially dead

Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

Top root causes since 2016:

#1: heap out-of-bounds

#2: use after free

#3: type confusion

#4: uninitialized use

Note: CVEs may have multiple root causes, so they can be counted in multiple categories

Safety is hard!

- memory safety is not easy (you need to understand the corner cases)
- concurrency safety is really hard (without certain tradeoffs)
- Often you have to bear the extra layer of abstractions
 - normally it means performance hit

Memory safety

- Manually - C/C++: painful and error-prone
- Smart Pointers - C++/ObjC/Swift: be aware of cyclical references
- GC - Java/DotNet/Erlang: much bigger memory consumption, and STW
- Ownership - Rust: learning curve

Concurrency safety

- single-threaded - Javascript: cannot leverage multicore
- GIL - Python/Ruby: multithreading is notorious inefficient
- Actor model - Erlang/Akka: at the cost of memory copy and heap allocation
- CSP - Golang: at the cost of memory copy and heap allocation
- Ownership + Type System - Rust: super **elegant** and **no extra cost!**

**How Rust achieves
memory and concurrency safety
without extra cost?**



Show me the code!

```
fn main() {
    let mut arr: Vec<i32> = vec![1, 2, 3];      move occurs because `arr` has type `Vec<i32>`, which does not implement the `Copy` trait
    arr.push(4);

    let _result: Result<(), Error> = process(arr);    value moved here
    let _v: Option<i32> = arr.pop(); // failed since arr is moved    borrow of moved value: `arr`

    // you can have multiple immutable references
    let mut arr1: Vec<i32> = vec![1, 2, 3];
    let ir1: &Vec<i32> = &arr1;
    let ir2: &Vec<i32> = &arr1;    immutable borrow occurs here

    println!("ir1: {:?} ir2: {:?}", ir1, ir2);

    // but you can't have both mutable and immutable references
    let mr1: &mut Vec<i32> = &mut arr1;    cannot borrow `arr1` as mutable because it is also borrowed as immutable
    // let mr2 = &mut arr1;

    println!("mr1: {:?} mr2: {:?}", mr1, ir2);    immutable borrow later used here

    // by default, closure borrows the data
    let mut arr2: Vec<i32> = vec![1, 2, 3];
    thread::spawn(|| {    closure may outlive the current function, but it borrows `arr2`, which is owned by the current function
        ... arr2.push(4);    `arr2` is borrowed here
    });
}

// we shall move the data explicitly
let mut arr3: Vec<i32> = vec![1, 2, 3];
thread::spawn(move || arr3.push(4));
}

fn thread_safety() {
    // but certain types cannot be moved to other thread safely
    let mut rc1: Rc<Vec<i32>> = Rc::new(vec![1, 2, 3]);
    thread::spawn(move || {    `Rc<Vec<i32>>` cannot be sent between threads safely
        ... rc1.push(4);
    });
}
```

```
fn thread_safety_reasoning() {
    let mut map: HashMap<&str, &str> = HashMap::new();      move occurs because `map` has type `HashMap<&str, &str>`, which does not implement
    map.insert(k: "hello", v: "world");

    // Arc is an atomic reference counter which can be moved safely across threads
    let mut ir: Arc<HashMap<&str, &str>> = Arc::new(data: map);      variable does not need to be mutable
    map.insert(k: "hello1", v: "world1"); // you can't do this since map is moved      borrow of moved value: `map`
    let ir1: Arc<HashMap<&str, &str>> = ir.clone(); // this is cheap, just reference counter clone
    thread::spawn(move || assert_eq!(ir1.get("hello"), Some(&"world")));
    // but arc is immutable, so this would fail
    thread::spawn(move || ir.insert(k: "hello2", v: "world2"));      cannot borrow data in an `Arc` as mutable

    // the compiler guide you to use types that provides mutable reference for threads

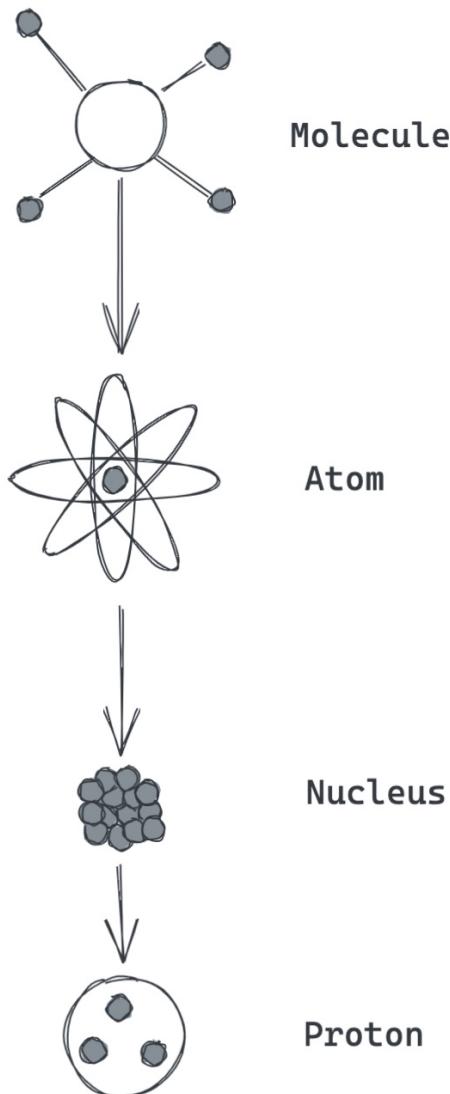
    // use Mutex - you can't clone a Mutex, thus you can't make it available for multiple threads
    let mut map1: HashMap<&str, &str> = HashMap::new();
    map1.insert(k: "hello", v: "world");
    let mr: Mutex<HashMap<&str, &str>> = Mutex::new(map1);
    let mr1 = mr.clone();      no method named `clone` found for struct `Mutex<HashMap<&str, &str>>` in the current scope
    thread::spawn(move || mr.lock().unwrap().insert(k: "hello1", v: "world1"));
    mr1.lock().unwrap().insert("hello2", "world2");

    // use Mutex with Arc - now you have mutable access and multi-thread cloning
    let mut map2: HashMap<&str, &str> = HashMap::new();
    map2.insert(k: "hello", v: "world");
    let mr: Arc<Mutex<HashMap<&str, &str>>> = Arc::new(data: Mutex::new(map2));
    let mr1: Arc<Mutex<HashMap<&str, &str>>> = mr.clone();

    thread::spawn(move || mr.lock().unwrap().insert(k: "hello1", v: "world1"));
    thread::spawn(move || mr1.lock().unwrap().insert(k: "hello2", v: "world2"));

    // can I use Box (smart pointer for heap allocation)?
    let mut map1: HashMap<&str, &str> = HashMap::new();
    map1.insert(k: "hello", v: "world");
    let mr: Arc<Box<HashMap<&str, &str>>> = Arc::new(data: Box::new(map1));
    let mr1: Arc<Box<HashMap<&str, &str>>> = mr.clone();
    thread::spawn(move || (**mr).insert(k: "hello1", v: "world1"));      cannot borrow data in an `Arc` as mutable
    mr1.insert(k: "hello2", v: "world2");      cannot borrow data in an `Arc` as mutable
}
```

First Principles Thinking



Boiling problems down to their most fundamental truth.

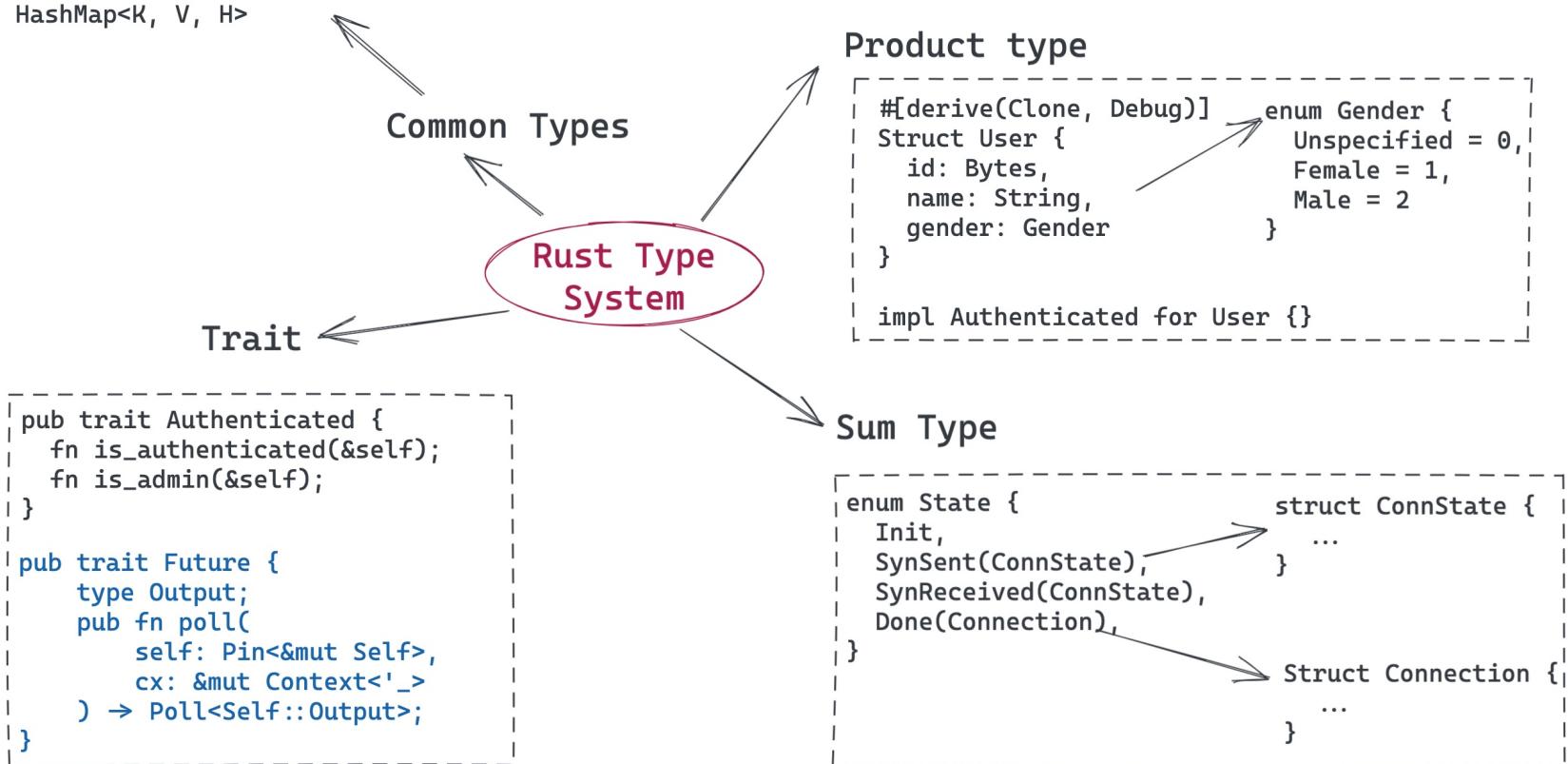
Recap

- One and only one owner
- Multiple immutable references
- mutable reference is mutual exclusive
- Reference cannot outlive owner
- **use type safety for thread safety**

With these simple rules, Rust achieved safety with
zero cost abstraction

A glance at Rust Type System

```
Option<T> = T | None  
Result<T, E> = Ok(T) | Err(E)  
Vec<T>  
HashMap<K, V, H>
```



How's Productivity of Rust?

```
1 # client configuration
2
3 domain = "localhost"
4
5 [cert]
6 pem = """-----BEGIN CERTIFICATE-----
7 MIIBeTCCASugAwIBAgIBKjAFBgMrZXAwNzELMAkGA1UEBgwCVVMxFDASBgNVBAoM
8 C0RvbWFpbijBjmMuMRIwEAYDVQQDDAlEb21haW4gQ0EwHhcNMjEwMzE0MTg0NTU2
9 WhcNMzEwMzEyMTg0NTU2WjA3MQswCQYDVQQGDAJVUzEUMBIGA1UECgwLRG9tYWlu
10 IEluYy4xExAQBgNVBAMMCURvbWFpbjBDQTaqMAUGAytlcAMhAAZhorM9IPsXjBTx
11 ZxykG15xZrsj3X2XqKjaAVutnf7po1wwjAUBgNVHREEDTALgglsb2Nhbgvc3Qw
12 HQYDVR00BYEFd+NqChBZD0s5MfMgefHJSIWirthXMBIGA1UDewEB/wQIMAYBaF8C
13 ARAwDwYDVR0PAQH/BAUDAwcGADAFBgMrZXADQQA9sIlgQcYGaBqTxR1+JadSelMK
14 Wp35+yhVvuu4PTL18kWdU819w3cVlRe/GHt+jjlbk1i22Tvf05AaNmdxySk0
15 -----END CERTIFICATE-----"""
16
17 # server configuration
18
19 [identity]
20 key = """-----BEGIN PRIVATE KEY-----
21 MFCAQEwBQYDK2VwBCIEII0kozd0PJsbnfNUS/oqI/Q/enDiLwmdw+JUnTLpR9xs
22 oSMDIQAtkhJiFdF9SYBIMcLikWPRIgca/Rz9ngIgd6HuG6HI3g==
23 -----END PRIVATE KEY-----"""
24
25 [identity.cert]
26 pem = """-----BEGIN CERTIFICATE-----
27 MIIBAzCAR2gAwIBAgIBKjAFBgMrZXAwNzELMAkGA1UEBgwCVVMxFDASBgNVBAoM
28 C0RvbWFpbijBjmMuMRIwEAYDVQQDDAlEb21haW4gQ0EwHhcNMjEwMzE0MTg0NTU2
29 WhcNMjIwMzE0MTg0NTU2WjA5MQswCQYDVQQGDAJVUzEUMBIGA1UECgwLRG9tYWlu
30 IEluYy4xFDASBgNVBAMMC0dSUEmgU2VydMVyMcwBQYDK2VwAyEALZISYhXRFuM
31 SDHC4pFj0SIHGv0c/Z4CIHeh7huhyN6jTDBKMBQGA1UdEQQNMAuCCWxvY2FsaG9z
32 dDATBgnVHSUEDDAKBgggrBgfFBQcDATAMBgnVHRMEBTADAQEA8GA1UdDwEB/wQF
33 AwMH4AwBQYDK2VwA0EAy7EOIZp73XtcqaSopQDGWU7Umi4DVvIgjmY6qbJZP0sj
34 ExGdaVq/7M01ZlI+vY7G0NSZWIZUilX0Co0krn0DA==
35 -----END CERTIFICATE-----"""
36
37
```

```
9   ````rust
10  // you could also build your config with cert and identity separately. See tests.
11  let config: ServerTlsConfig = toml::from_str(config_file).unwrap();
12  let acceptor = config.tls_acceptor().unwrap();
13  let listener = TcpListener::bind(addr).await.unwrap();
14  tokio::spawn(async move {
15      loop {
16          let (stream, peer_addr) = listener.accept().await.unwrap();
17          let stream = acceptor.accept(stream).await.unwrap();
18          info!("server: Accepted client conn with TLS");
19
20          let fut = async move {
21              let (mut reader, mut writer) = split(stream);
22              let n = copy(&mut reader, &mut writer).await?;
23              writer.flush().await?;
24              debug!("Echo: {} - {}", peer_addr, n);
25          }
26
27          tokio::spawn(async move {
28              if let Err(err) = fut.await {
29                  error!("{}: {:?}", err);
30              }
31          });
32      }
33  });
34  ````
35
36 Client: You, a month ago • init the project
37
38 ````rust
39 let msg = b"Hello world\n";
40 let mut buf = [0; 12];
41
42 // you could also build your config with cert and identity separately. See tests.
43 let config: ClientTlsConfig = toml::from_str(config_file).unwrap();
44 let connector = config.tls_connector(Uri::from_static("localhost")).unwrap();
45
46 let stream = TcpStream::connect(addr).await.unwrap();
47 let mut stream = connector.connect(stream).await.unwrap();
48 info!("client: TLS conn established");
49
50 stream.write_all(msg).await.unwrap();
51
52 info!("client: send data");
53
54 let (mut reader, _writer) = split(stream);
55
56 reader.read_exact(buf).await.unwrap();
57
58 info!("client: read echoed data");
59  ````
```

Things built with Rust



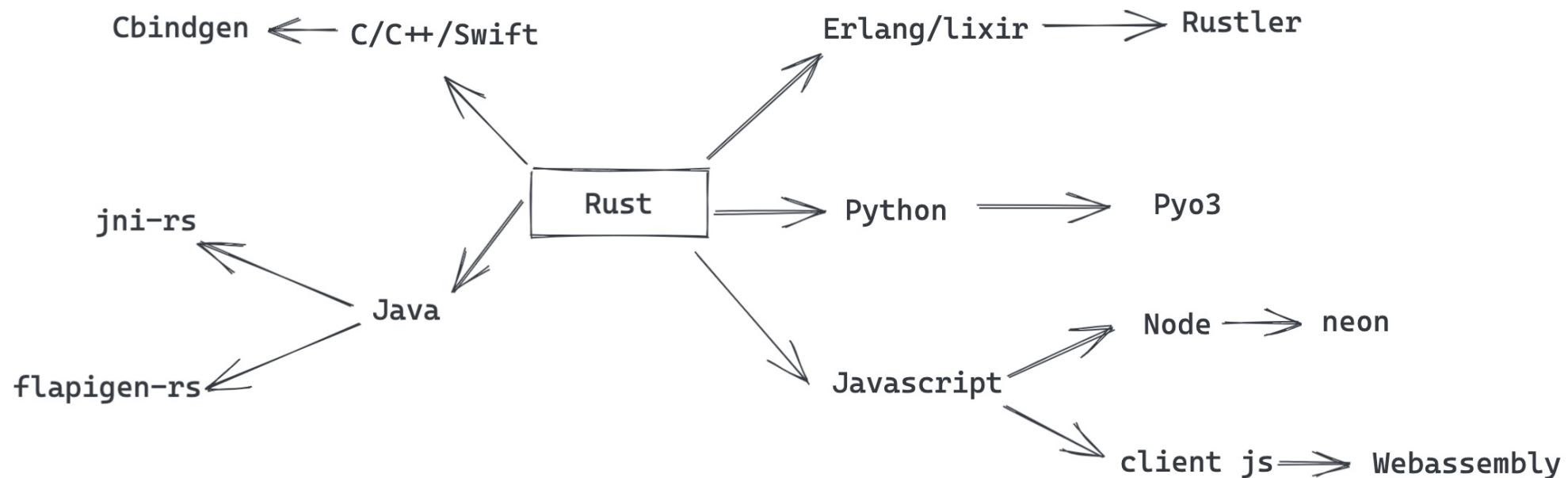
Should I use Rust?

- Rust is ideal when you need a system that reliable and performant
- Sometimes you don't, sometimes you do, sometimes you need that later
- it's all about tradeoffs

Rust for our use cases

- parts of the system that are bottlenecks
 - bottleneck on computation
 - bottleneck on memory consumption
 - bottleneck on I/O
- parser/decoder/encoder
- wants to leverage existing C/C++/Rust ecosystem (e.g. you need blake3 for hashing)

Rust FFI



Learning rust as a(n)...

- Elixir eng: ownership model, type system, oh no mutation
- Scala eng: ownership model, oh no mutation
- Typescript eng: ownership model, multi-threaded programming
- Swift/Java eng: ownership model
- Python eng: ownership model, type system

**The common
misunderstandings**

1. Rust is super hard to learn...



Rust is explicit

- Lots of knowledge about computer science is suddenly explicit to you
- If all your pain to learn a lang is 100%:
 - Rust:
 - Compiler help to reduce that to 90%
 - Then you suffer 70% the pains in first 3-6 months
 - Then the rest 20% in 3-5 years
 - Other:
 - You suffer 10-30% in first 3-6 months
 - Then 70%-90% in next 3-5 years

2. Unsafe Rust is evil...

Safe Rust	Unsafe Rust
<p>Compiler will make sure dereference is valid</p> 	<p>dereference raw pointer is OK</p> 
<p>Compiler will guarantee it is OK</p> 	<p>deal with FFI</p> <p>impl unsafe trait</p> 
<p>peace of mind</p> <p>The compiler will enforce memory and thread safety</p> 	<p>The dev guarantees memory/thread safety</p> <p>Need extra code review Static analysis</p> 

References

- The pain of real linear types in Rust
- Substructural type system
- Rust official book
- Rust official site
- Awesome Rust
- Are we web yet?
- Are we async yet?
- Are we gui yet?
- Are we learning yet?
- Are we game yet?
- Are we quantum yet?
- Are we IDE yet?

Ownership, borrow check, and
lifetime

Typesystem and data structures

Concurrency - primitives

Concurrency - `async/await`

Networking and security

FFI with C/Elixir/Swift/Java

WASM/WASI

Rust for real-world problems

May the **Rust** be with you