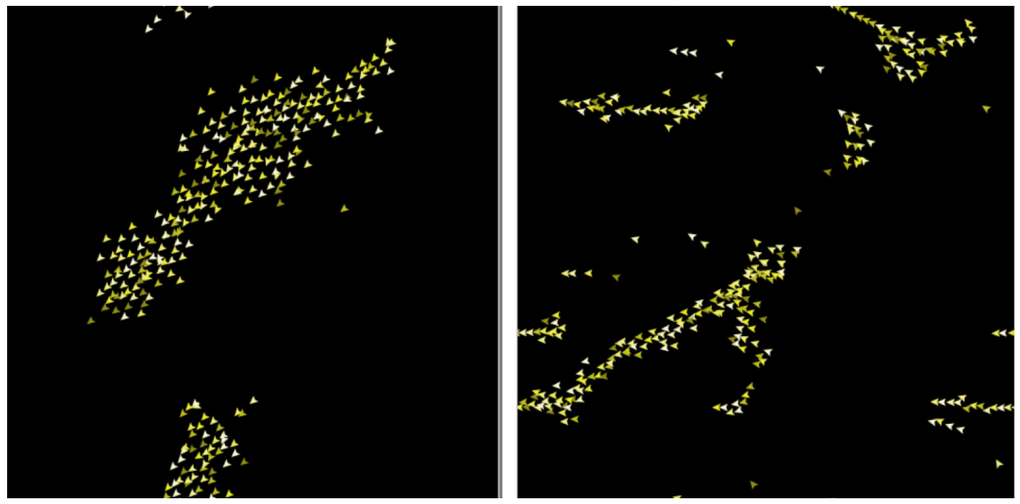


# PROGRAMMING IN

# RUST

Daniele De Vinco

# INDICE

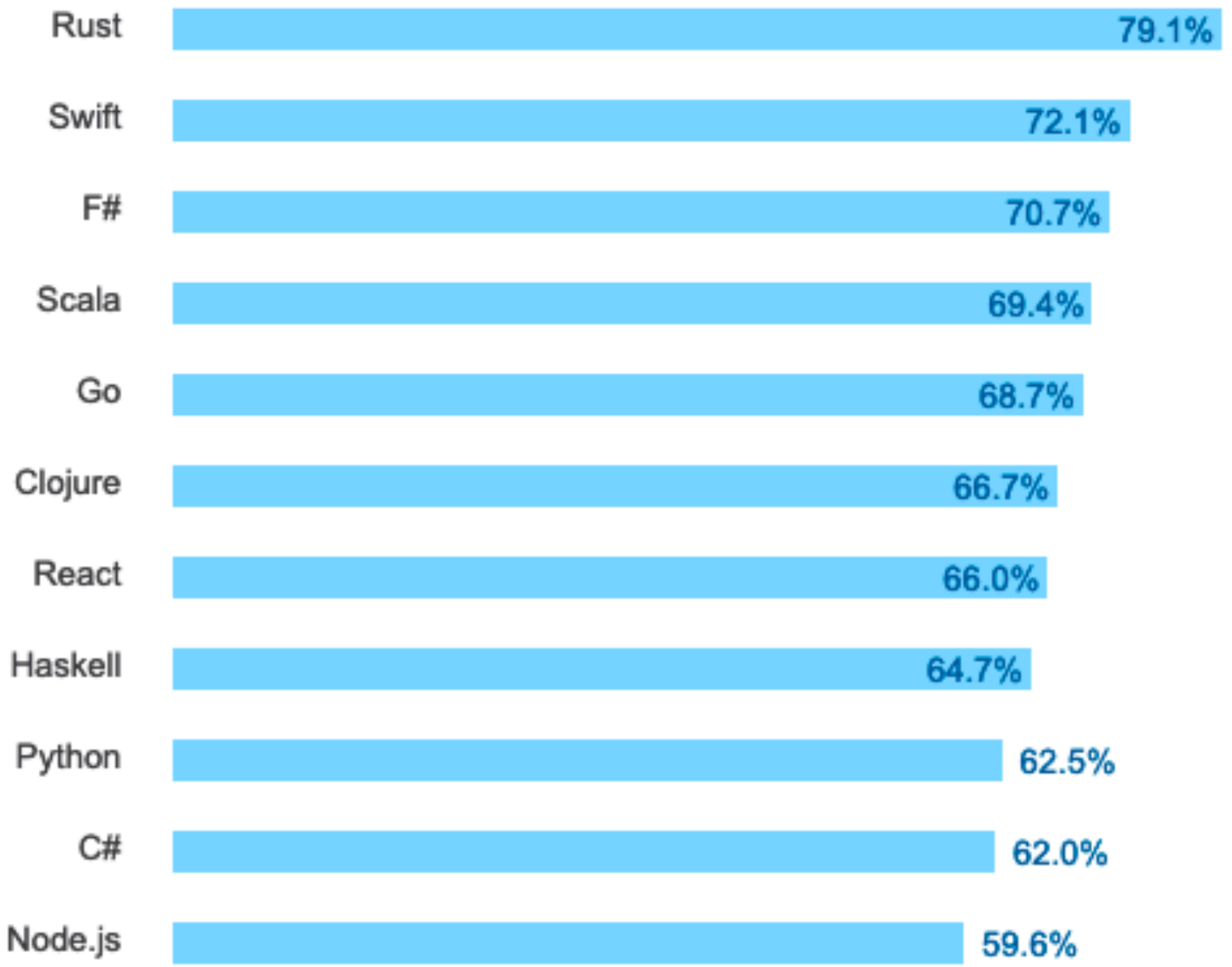

- Intro
- Variabili e mutability
- Memory safe
- Struct - Enum - Match
- Scope
- Collections
- Error Handling
- Generics
- Closure
- Thread
- Unsafe Rust



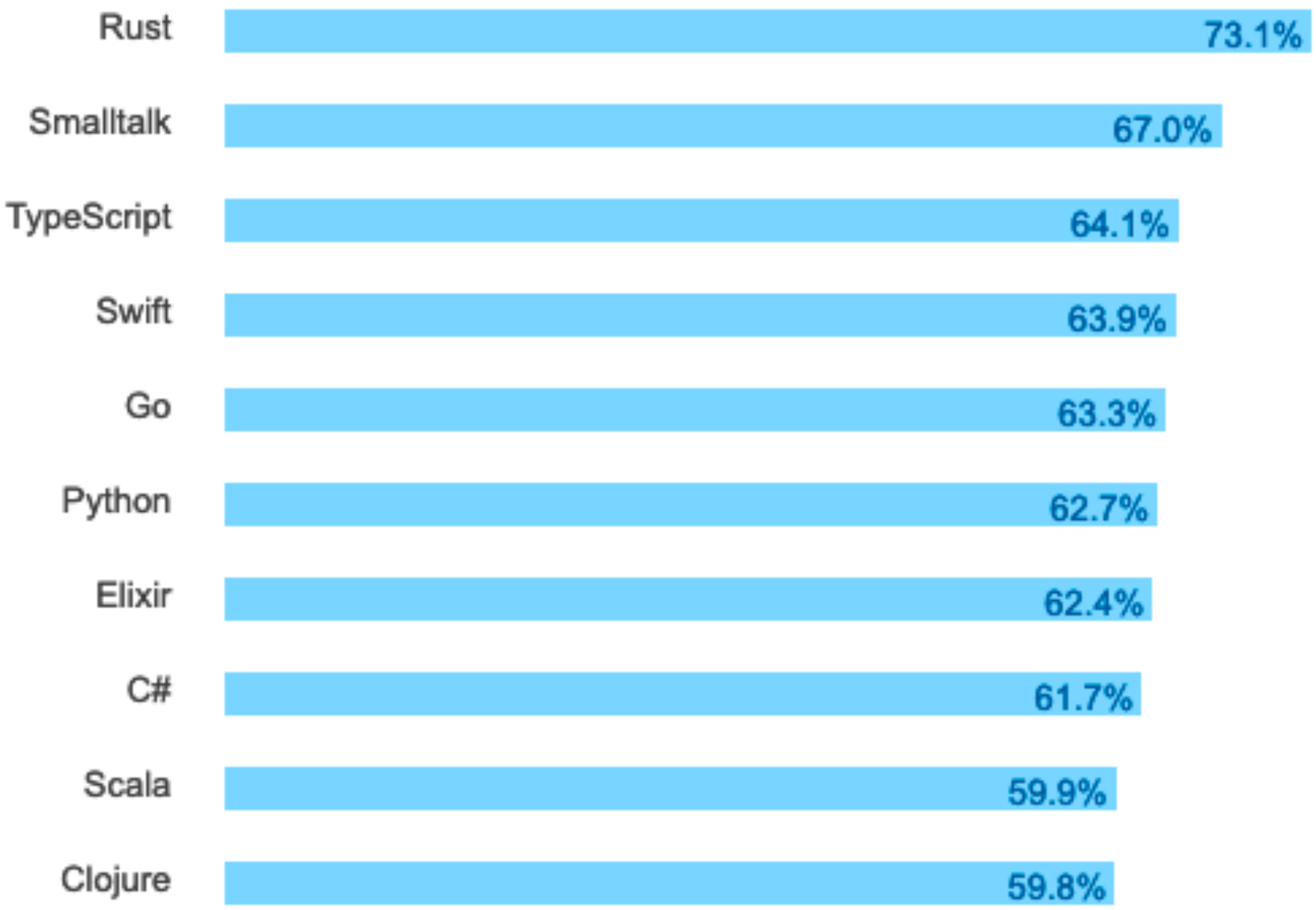

# STORIA DI RUST

- Sviluppato da Mozilla Research
- Open-Source
- Multi-paradigma
  - imperativo-procedurale, funzionale,  e object-oriented
- Versione stabile 1.0 nel 2015
- Most loved programming language

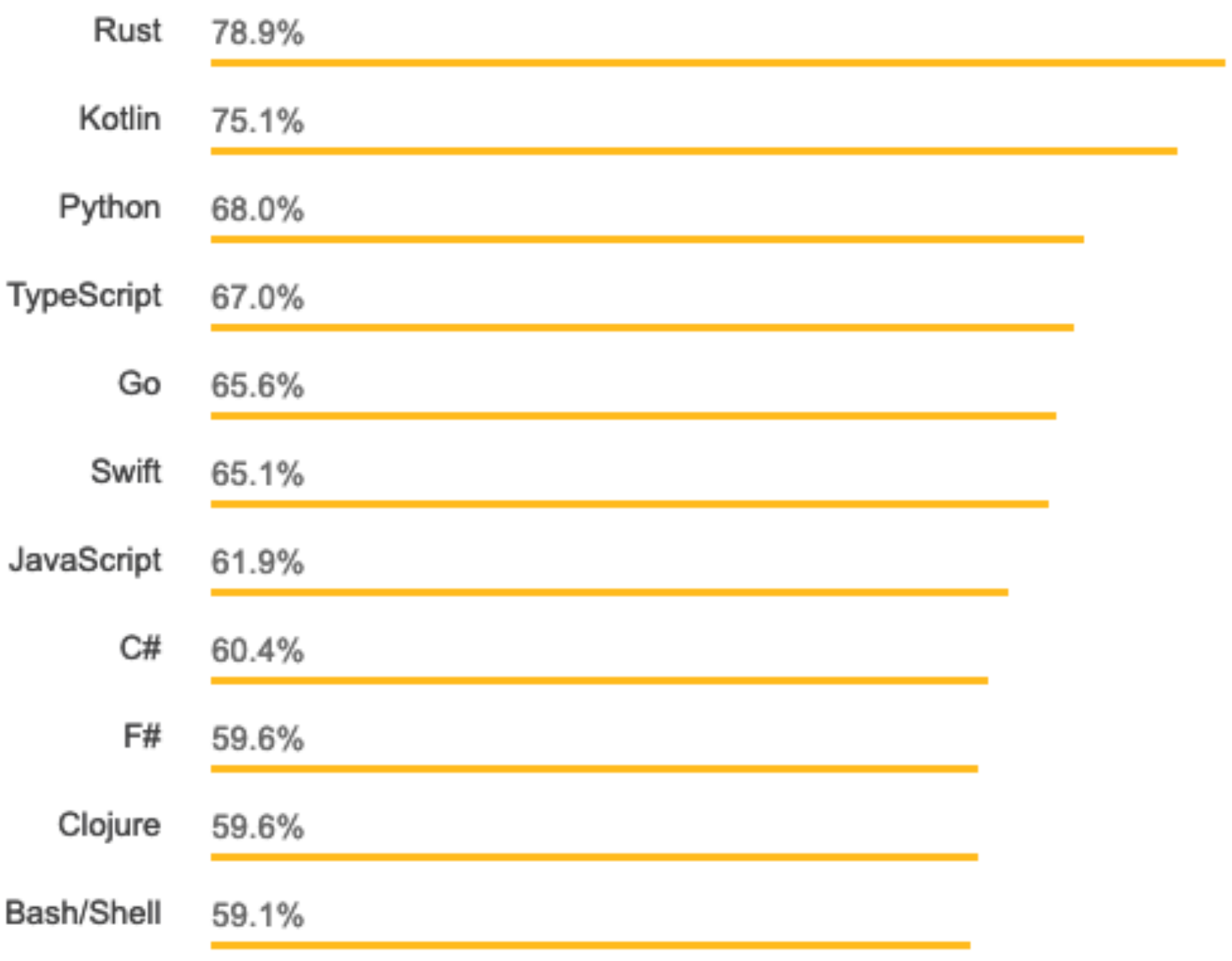



Rust
79.1%
Swift
72.1%
F#
70.7%
Scala
69.4%
Go
68.7%
Clojure
66.7%
React
66.0%
Haskell
64.7%
Python
62.5%
C#
62.0%
Node.js
59.6%

ISISLab


Rust
73.1%
Smalltalk
67.0%
TypeScript
64.1%
Swift
63.9%
Go
63.3%
Python
62.7%
Elixir
62.4%
C#
61.7%
Scala
59.9%
Clojure
59.8%

ISISLab


Rust 78.9%
Kotlin 75.1%
Python 68.0%
TypeScript 67.0%
Go 65.6%
Swift 65.1%
JavaScript 61.9%
C# 60.4%
F# 59.6%
Clojure 59.6%
Bash/Shell 59.1%


ISISLab

# PERCHÉ RUST

- Codice più sicuro e affidabile
- Build System
  - Cargo
- No Data Race



# PERCHÉ RUST

- Performance
- Affidabilità
- Produttività



# BENCHMARK

reverse-complement

| source | secs | mem | gz | cpu | cpu load |
|---|---|---|---|---|---|
| Rust | **1.61** | 994,796 | 1376 | 2.74 | 28% 24% 96% 24% |
| C gcc | 1.76 | 994,488 | 1438 | 3.98 | 97% 43% 44% 46% |

binary-trees

| source | secs | mem | gz | cpu | cpu load |
|---|---|---|---|---|---|
| Rust | **3.50** | 134,208 | 754 | 12.73 | 89% 89% 99% 89% |
| C gcc | 3.73 | 116,600 | 836 | 11.74 | 76% 71% 98% 72% |

spectral-norm

| source | secs | mem | gz | cpu | cpu load |
|---|---|---|---|---|---|
| Rust | **1.97** | 2,312 | 1126 | 7.86 | 100% 100% 99% 99% |
| C gcc | 1.98 | 1,168 | 1139 | 7.87 | 100% 99% 100% 99% |

# BENCHMARK

regex-redux

| source | secs | mem | gz | cpu | cpu load |
|---|---|---|---|---|---|
| Rust | **2.51** | 191,144 | 765 | 3.93 | 16% 84% 58% 0% |
| Java | 10.50 | 571,736 | 929 | 31.63 | 71% 73% 66% 92% |

n-body

| source | secs | mem | gz | cpu | cpu load |
|---|---|---|---|---|---|
| Rust | **5.72** | 916 | 1753 | 5.72 | 0% 100% 1% 0% |
| Java | 22.00 | 32,272 | 1489 | 22.07 | 1% 0% 0% 100% |

mandelbrot

| source | secs | mem | gz | cpu | cpu load |
|---|---|---|---|---|---|
| Rust | **2.00** | 11,740 | 1007 | 7.93 | 99% 100% 99% 99% |
| Java | 6.96 | 76,316 | 796 | 27.06 | 97% 97% 97% 97% |

# A CHI È RIVOLTO RUST

- Team
- Studenti
- Aziende



# A CHI È RIVOLTO RUST

- Team
- Studenti
- Aziende











# VARIABILI E MUTABILITY

- Variabili
- Shadowing
- Tipizzazione statica e inferita



# MUTABILITY

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```



# MUTABILITY

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

```
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: make this binding mutable: `mut x`
3 |     println!("The value of x is: {}", x);
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable

error: aborting due to previous error
```



# MUTABILITY

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```



# MUTABILITY

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

```
The value of x is: 5
The value of x is: 6
wifistud8246:example_rust Dani$
```



# SHADOWING

```
fn main() {
    let x = 5;
    let x = x + 1;
    let x = x * 2;
    println!("The value of x is: {}", x);
}
```



# SHADOWING

```
fn main() {
    let x = 5;
    let x = x + 1;
    let x = x * 2;
    println!("The value of x is: {}", x);
}
```

```
The value of x is: 12
wifistud8246:example_rust Dani$
```



# MEMORY SAFE

- Inizializzazione forzata + restricted aliasing + ownership
- Reference & Borrowing



# INIZIALIZZAZIONE FORZATA

```
fn main() {
    let x;
    println!("the value of x is : {}", x);
}
```



# INIZIALIZZAZIONE FORZATA

```
fn main() {
    let x;
    println!("the value of x is : {}", x);
}
```

```
error[E0282]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let x;
  |         ^
  |         |
  |         cannot infer type
  |         consider giving `x` a type
```



# OWNERSHIP

```
fn main() {                 // s is not valid here
    let s = "hello";        // s is valid from here

    // do stuff with s
}                           // s is no longer valid
```



# OWNERSHIP

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;

    println!("{}, world!", s1);
}
```



# OWNERSHIP

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;

    println!("{}, world!", s1);
}
```

```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move
  |
  = note: move occurs because `s1` has type `std::string::String`, which does not implement the `Copy` trait
```



# REFERENCE AND BORROWING

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);
}

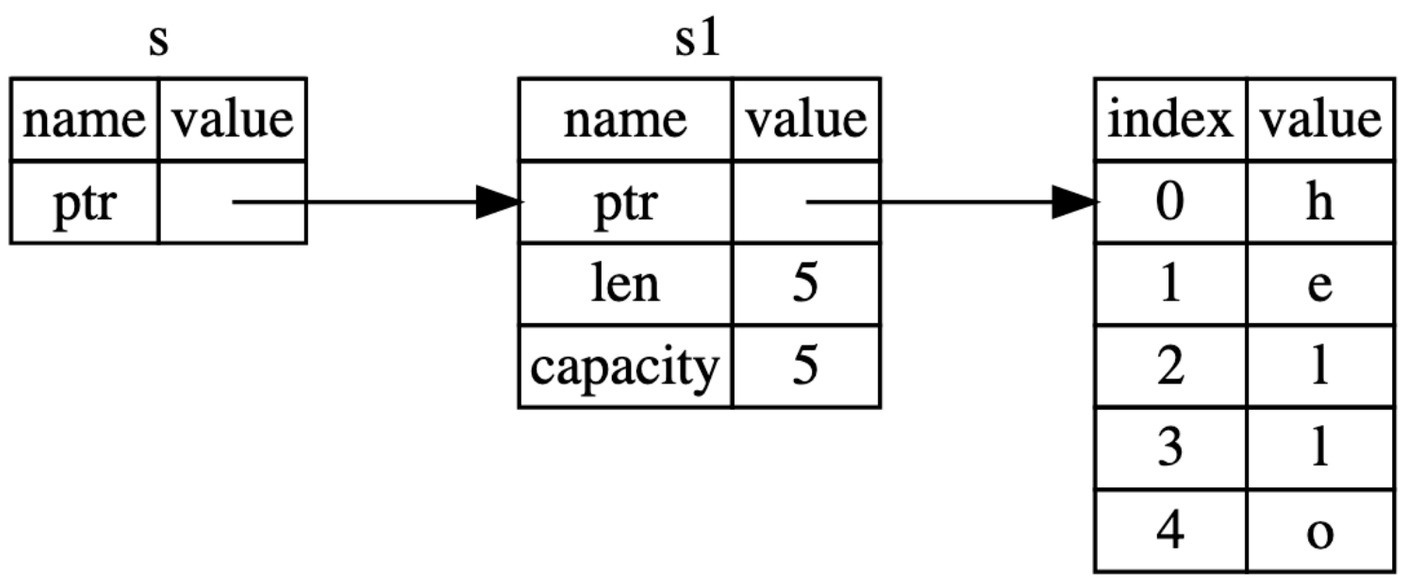
fn calculate_length(s: &String) -> usize {
    s.len()
}
```



# REFERENCE AND BORROWING

```rust
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```




ISISLab

# RESTRICTED ALIASING

```
fn main() {
    let mut s = String::from("hello");
    let r1 = &s;
    let r2 = &s;
    let r3 = &mut s;
    println!("{} {} {}", r1, r2, r3);
}
```



# RESTRICTED ALIASING

```rust
fn main() {
    let mut s = String::from("hello");
    let r1 = &s;
    let r2 = &s;
    let r3 = &mut s;
    println!("{} {} {}", r1, r2, r3);
}
```

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:14
  |
4 |     let r1 = &s; // no problem
  |              -- immutable borrow occurs here
5 |     let r2 = &s; // no problem
6 |     let r3 = &mut s; // BIG PROBLEM
  |              ^^^^^^ mutable borrow occurs here
7 |     println!("{} {} {}", r1, r2, r3);
  |                          -- immutable borrow later used here
```



# RESTRICTED ALIASING

```
fn main() {
    let mut v = Vec::new();
    v.push("Hello");
    let x = &v[0];
    v.push("world");
    println!("{:?}", x);
}
```



# RESTRICTED ALIASING

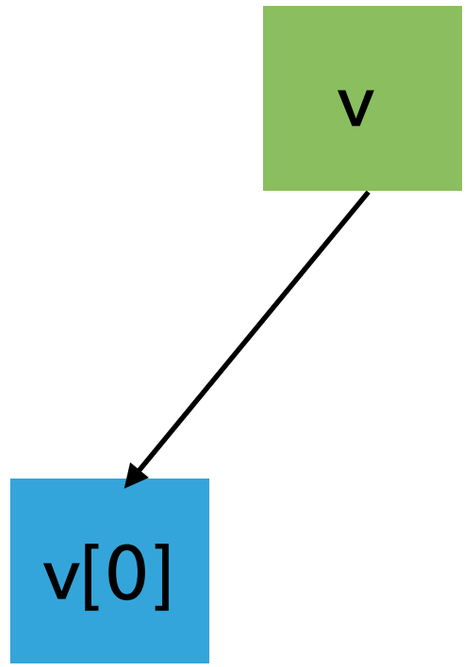
```
fn main() {
    let mut v = Vec::new();
    v.push("Hello");
    let x = &v[0];
    v.push("world");
    println!("{:?}", x);
}
```






# RESTRICTED ALIASING

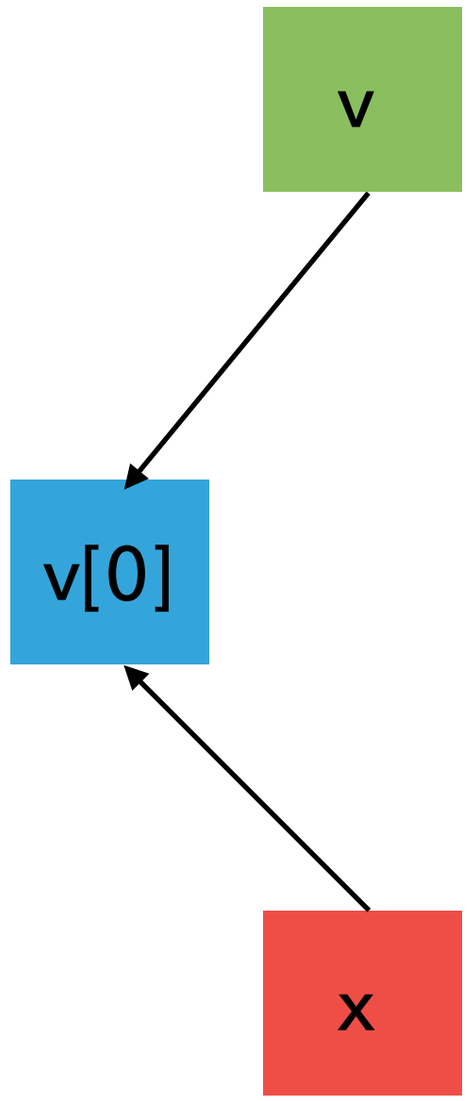
```
fn main() {
    let mut v = Vec::new();
    v.push("Hello");
    let x = &v[0];
    v.push("world");
    println!("{:?}", x);
}
```






# RESTRICTED ALIASING

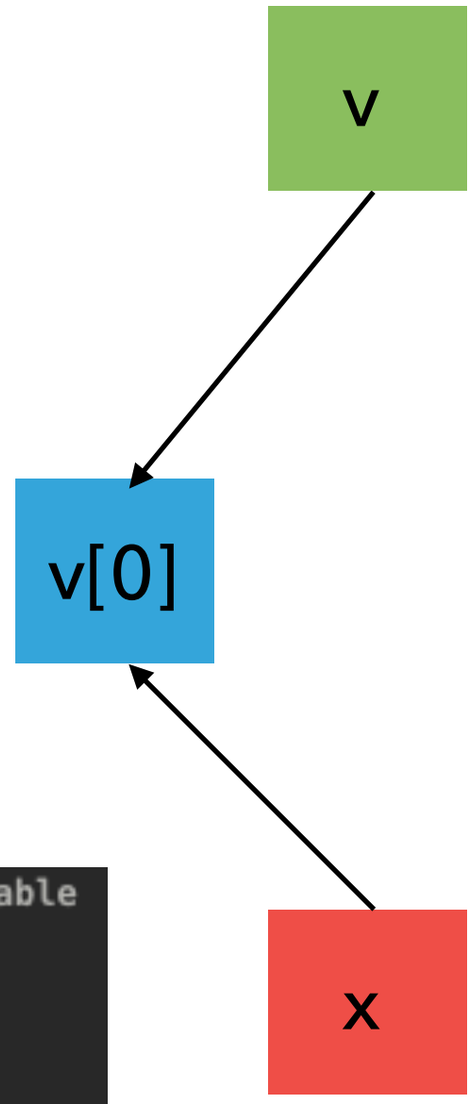
```
fn main() {
    let mut v = Vec::new();
    v.push("Hello");
    let x = &v[0];
    v.push("world");
    println!("{:?}", x);
}
```






# RESTRICTED ALIASING

```
fn main() {
    let mut v = Vec::new();
    v.push("Hello");
    let x = &v[0];
    v.push("world");
    println!("{:?}", x);
}
```




```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:5:5
  |
4 |     let x = &v[0];
  |             - immutable borrow occurs here
5 |     v.push("World");
  |     ^^^^^^^^^^^^^^^ mutable borrow occurs here
6 |     println!("{:?}", x);
  |                      - immutable borrow later used here
```



# STRUCT

- Custom data type



# STRUCT

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```



# ENUM E MATCH

- enum
- match
- if let



# ENUM E MATCH

```
enum WebEvent {
    PageLoad,
    PageUnload,
    KeyPress(char),
    Paste(String),
    Click { x: i64, y: i64 },
}

fn inspect(event: WebEvent) {
    match event {
        WebEvent::PageLoad => println!("page loaded"),
        WebEvent::PageUnload => println!("page unloaded"),
        // Destructure `c` from inside the `enum`.
        WebEvent::KeyPress(c) => println!("pressed '{}'.", c),
        WebEvent::Paste(s) => println!("pasted \"{}\".", s),
        // Destructure `Click` into `x` and `y`.
        WebEvent::Click { x, y } => {
            println!("clicked at x={}, y={}.", x, y);
        },
    }
}
```



# IF LET

```
enum Foo {
    Bar,
    Fuzz,
    Qux(u32)
}

fn main() {
    let a = Foo::Bar;
    let b = Foo::Fuzz;
    let c = Foo::Qux(100);

    if let Foo::Bar = a { println!("a is foobar");}

    if let Foo::Fuzz = b { println!("never printed");}

    if let Foo::Qux(value) = c {
        println!("c is {}", value);
    }
}
```



# SCOPE

- Package
  - Cargo.toml
- Crates
- Modules
- Path



# SCOPE

```
mod sound {
    pub mod instrument {
        pub fn clarinet() {
            // code
        }
    }
}

fn main() {
    // Absolute path
    crate::sound::instrument::clarinet();

    // Relative path
    sound::instrument::clarinet();
}
```



# COLLECTIONS

- Vector
- String
- HashMap



# ERROR HANDLING

- Unrecoverable
- Recoverable errors with Result<T, E>
- To Panic! or not to Panic!



# RECOVERABLE ERRORS

```
fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!(“Error with file: {:?}", error)
        },
    };
}
```



# RECOVERABLE ERRORS

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

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```



# RECOVERABLE ERRORS

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

fn main() {
    let f = File::open("hello.txt").expect("Failed to open
hello.txt");
}
```



# RECOVERABLE ERRORS

```
fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```



# GENERICS

- Generic types
  - Monomorphization
- Trait
  - Coherence(Orphan rule)
- Lifetimes
  - Borrow checker
  - Lifetime elision
  - No dangling pointers



# GENERIC TYPES

```
struct Point<T, U> {
    x: T,
    y: U,
}
```

```
fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```



# GENERIC TYPES

```
impl Point<f32, f32> {
    fn sum_point_float(&self) -> f32 {
        self.x + self.y
    }
}
```

```
fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
    println!("{}", both_float.sum_point_float() );
    println!("{}", both_integer.sum_point_float() );
}
```



# TRAIT

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!(“(Read from {})", self.summarize_author())
    }
}
```



# TRAIT

```
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```



# TRAIT

```rust
pub fn notify(item: impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

```rust
pub fn notify(item: impl Summary + Display)
```



# TRAIT

```rust
pub fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}
```

```rust
pub fn notify<T: Summary + Display>(item: T)
```

```rust
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```



# LIFETIME BORROW CHECKER

```
{
let r;                    // ----------+-- 'a
                          //           |
{                         //           |
    let x = 5;            // -+-- 'b   |
    r = &x;               //  |        |
}                         // -+        |
                          //           |
println!("r: {}", r);     //           |
}                         // ----------+
```



# LIFETIME ELISION

- Se un parametro è una reference, ha il suo lifetime
- Se c'è un solo parametro con lifetime in input, allora quel lifetime è assegnato a tutti i parametri di output
- Se ci sono multipli parametri di input con lifetime, ma uno di essi è '&self' o '&mut self', allora il lifetime di 'self' è assegnato a tutti i parametri di output



# LIFETIME ELISION

```rust
fn first_word(s: &str) -> &str {}
```

```rust
fn first_word<'a>(s: &'a str) -> &str {}
```

```rust
fn first_word<'a>(s: &'a str) -> &'a str {}
```



# LIFETIME

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```



# LIFETIME

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```



# LIFETIME

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

```
  --> src/main.rs:14:44
   |
14 |         result = longest(string1.as_str(), string2.as_str());
   |                                            ^^^^^^^ borrowed value does not live long enough
15 |     }
   |     - `string2` dropped here while still borrowed
16 |     println!("The longest string is {}", result);
   |                                          ------ borrow later used here
```



# CLOSURE

- Closure
  - Funzione anonima che può essere salvata in una variabile o passata come argomento ad altre funzioni



# CLOSURE

```
let expensive_closure = |num| {
    println!("calculating...");
    thread::sleep(Duration::from_secs(2));
    num
};
```



# CLOSURE

```
struct Cacher<T>
    where T: Fn(u32) -> u32
{
    calculation: T,
    value: Option<u32>,
}
```



# CLOSURE

```
impl<T> Cacher<T>
    where T: Fn(u32) -> u32
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            },
        }
    }
}
```



# THREAD

- Comunicazione
  - Tramite messaggi
  - Memoria condivisa



# THREAD

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

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });
    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
    handle.join().unwrap();
}
```



# THREAD

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

```
--> src/main.rs:6:32
  |
6 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `v`
7 |         println!("Here's a vector: {:?}", v);
  |                                           - `v` is borrowed here
  |
```



# THREAD

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```



# THREAD - MESSAGE PASSING

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```



# THREAD - MESSAGE PASSING

```rust
fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        println!("val is {}", val);
    });
    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```



# THREAD - MESSAGE PASSING

```rust
fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        println!("val is {}", val);
    });
    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

```
error[E0382]: borrow of moved value: `val`
  --> src/main.rs:10:31
   |
9  |         tx.send(val).unwrap();
   |                 --- value moved here
10 |         println!("val is {}", val);
   |                               ^^^ value borrowed here after move
   |
```



# THREAD – SHARED STATE

```rust
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);
    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }
    println!("m = {:?}", m);
}
```



# THREAD - SHARED STATE

```rust
fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }
    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```



# UNSAFE RUST

- Superpowers
  - Dereference a raw pointer
  - Call an unsafe function or method
  - Access or modify a mutable static variable
  - Implement an unsafe trait



# DEREFERENCE A RAW POINTER

```
fn main() {
    let mut num = 5;
    let r1 = &num as *const i32;
    let r2 = &mut num as *mut i32;
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```



# DEREFERENCE A RAW POINTER

```
fn main() {
    let mut num = 5;
    let r1 = &num as *const i32;
    let r2 = &mut num as *mut i32;
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

```
error[E0133]: dereference of raw pointer is unsafe and requires unsafe function or block
  --> src/main.rs:10:27
   |
10 |     println!("r2 is: {}", *r2);
   |                           ^^^ dereference of raw pointer
   |
   = note: raw pointers may be NULL, dangling or unaligned; they can violate aliasing rules and cause data races: all of the
se are undefined behavior
```



# DEREFERENCE A RAW POINTER

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```



# CALL UNSAFE METHOD OR FUNCTIONS

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```



# ACCESS OR MODIFY A MUTABLE STATIC VARIABLE

```
static mut COUNTER: u32 = 0;
fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}
fn main() {
    add_to_count(3);
    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```



# UNSAFE TRAIT

```
unsafe trait Foo {
    // methods go here
}

unsafe impl Foo for i32 {
    // method implementations go here
}
```



# GRAZIE PER L'ATTENZIONE

