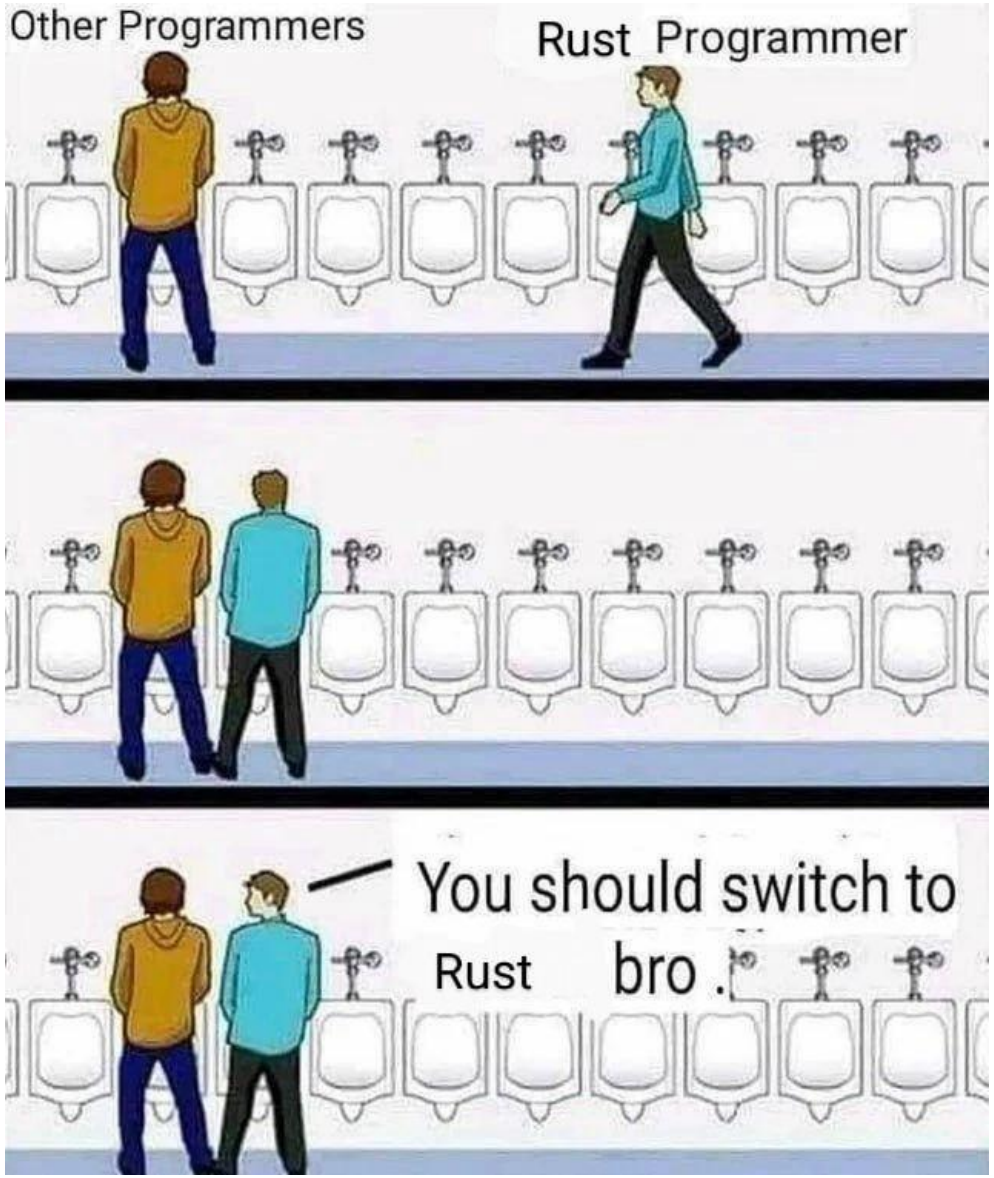
PROGRAMMING PARADIGMS

CS 3022 – Imperative Programming with Rust

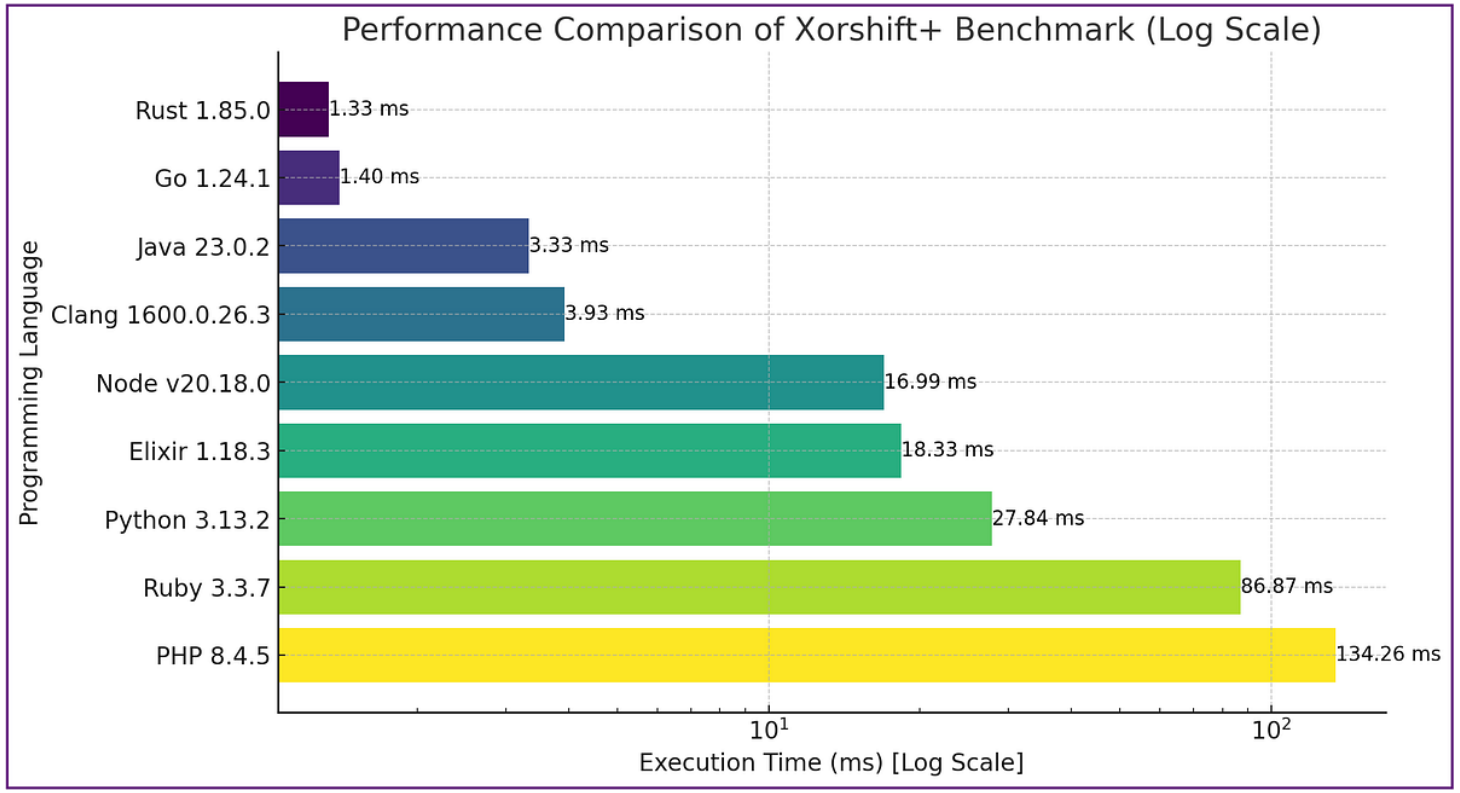


HOUSEKEEPING

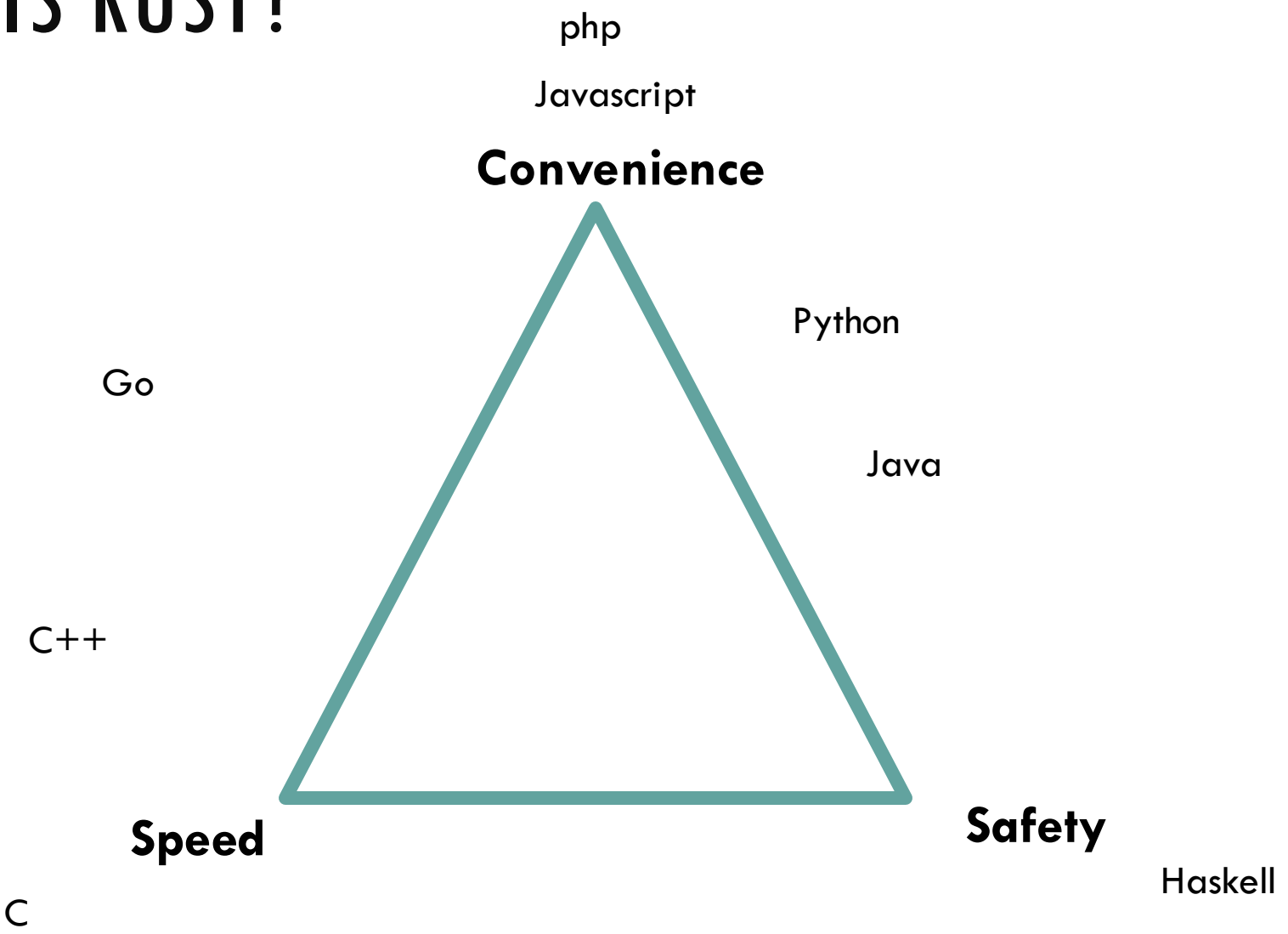
* Exam date time. November 5th?

WHY IS RUST SO POPULAR

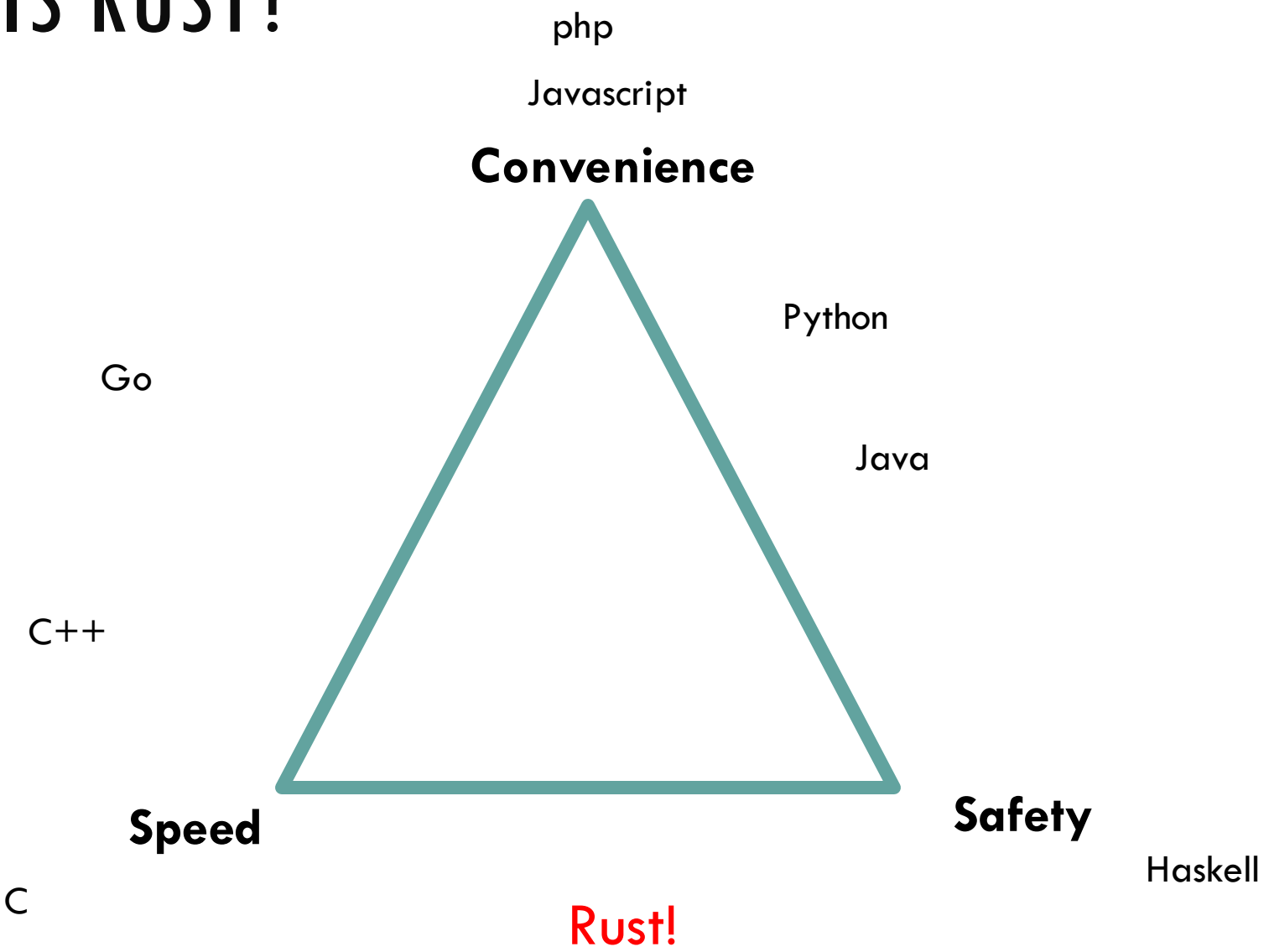
- Memory safe
- Type safe
- Data race-free
- Ahead-of-time compiled
- Built on and encourages zero-cost abstractions
- Minimal runtime (no garbage collection, no JIT compiler, no VM)
- FAST!!!
- Its new (2012)



WHERE IS RUST?



WHERE IS RUST?



LECTURE THIS WEEK

- Homework: Install Rust

https://doc.rust-lang.org/book/ch01-01-installation.html

- Lectures are from Rust handbook:

https://doc.rust-lang.org/book/title-page.html

Chapters 1 - 4.

HELLO WORLD

```rust
fn main() {
    println!("Hello, world!");
}
```

Macros – code that is generated with a script before compilation.

Here is a macro in c. SQUARE(x) is literally replaced with ((x) * (x)).

```c
#define SQUARE(x) ((x) * (x))
```

Rust macros are called with a '!' character. println! Is replaced with more in depth code before compile time. Often used as a type of syntactic sugar.

HELLO WORLD

```rust
fn main() {
    println!("Hello, world!");
}
```

```rust
fn main() {
    {
        ::std::io::_print(::core::fmt::Arguments::new_v1(
            &["Hello, world!\n"],
            &[],
        ));
    }
}
```

```
$ rustc main.rs
$ ./main
Hello, world!
```

CARGO – RUST BUILD SYSTEM AND PACKAGE MANAGER

Any non-trivial program should use Cargo. Three important commands:

$ cargo new <name of project>

$ cargo build

$ cargo run

toml file manages the build

Cargo.toml U

Cargo.toml

```
[package]
name = "rust_server"
version = "0.1.0"
edition = "2021"

[dependencies]
anyhow = "1.0"
thiserror = "1.0"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
chrono = { version = "0.4", features = ["serde"] }
uuid = { version = "1.0", features = ["serde", "v4"] }
log = "0.4"
env_logger = "0.10"
parking_lot = "0.12"
arc-swap = "1.1"
```

GUESSING GAME

What is a mut???? →

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

MUTABILITY

Immutable means that the variable can never change. By default, all variables are immutable.

```
let apples = 5; // immutable
let mut bananas = 5; // mutable
```

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

← This is an error!!!!

GUESSING GAME

The <whole>::<part> syntax allows access to an associated part of the larger whole.

std::io – imports the io library in the std package

String::new() – calls the 'new()' function that is part of the String type

io::stdin() – calls the 'stdin()' function that is part of the io library

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

GUESSING GAME

io::stdin() returns an object with input reading functions.

```
io::stdin()
    .read_line(&mut guess)
```

.read_line() is called with the returned object.

```
.expect("Failed to read line");
```

.read_line() takes the input and sets it to guess.
It also returns a 'Result' object. Result has a function called expect() that will print out the message if reading fails.

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

GUESSING GAME

What is “&mut guess” ?

```
io::stdin()
    .read_line(&mut guess)
```

& - passes by *reference*.
- It does not make a copy of guess
- Multiple parts of code access same data

mut – means guess can be changed, the read_line() function can modify it.

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

GUESSING GAME – ADDING RANDOM

Don't forget to add to your .toml file

```
[dependencies]
rand = "0.8.5"
```

This

```
rand::thread_rng()
```

returns a Rng object.

Range expressions take the form `start..=end` and is inclusive.

```
use std::io;

use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    println!("The secret number is: {secret_number}");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {guess}");
}
```

GUESSING GAME – ADD VERIFICATION

We have 2 variables named ‘guess’. This variable replacement is called ‘**shadowing**’. The String ‘guess’ is no longer valid after the u32 ‘guess’ is created.

Note: the type is explicit here (unsigned 32 bit integer)

```
let guess: u32
```

In the first declaration of guess, it was implied to be a String.

```
// --snip--

let mut guess = String::new();

io::stdin()
    .read_line(&mut guess)
    .expect("Failed to read line");

let guess: u32 = guess.trim().parse().expect("Please type a number!");

println!("You guessed: {guess}");

match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal => println!("You win!"),
}
```

GUESSING GAME – MATCHES

- **match** is incredibly common in Rust.
- guess.cmp(&secret_number) returns an enum called Ordering.

```
pub enum Ordering {
    Less = -1,
    Equal = 0,
    Greater = 1,
}
```

- **match** is exhaustive. It requires a code response for each possible value in the enum.

```
match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal => println!("You win!"),
}
```

SIDEBAR – FORMAL METHODS WITH DIJKSTRA

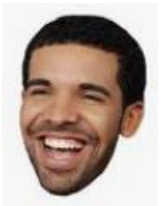
"Programmers should be regarded as mathematicians who produce proofs, and programs are the texts of those proofs"

"The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness."

WRAPPING

* Wrapping values in enums is very common
* Wrapped values are NOT functions
* When matching, values must be “unwrapped”

```
enum Operation {
    Add(i32, i32),      // Wraps two numbers for addition
    Multiply(i32, i32), // Wraps two numbers for multiplication
    Invalid(String),    // Wraps an error message
}
```



```
fn perform(op: Operation) -> i32 {
    match op {
        Operation::Add(a, b) => a + b,
        Operation::Multiply(a, b) => a * b,
        Operation::Invalid(_) => 0, // or handle error differently
    }
}

fn main() {
    let a = Operation::Add(2, 3);
    let b = Operation::Multiply(4, 5);
    let c = Operation::Invalid(String::from("oops"));

    println!("Add result: {}", perform(a));        // 5
    println!("Multiply result: {}", perform(b));   // 20
    println!("Invalid result: {}", perform(c));    // 0
}
```

UNWRAPPING

- Unwrapping allows access to the value wrapped inside the enum (if implemented by the enum).

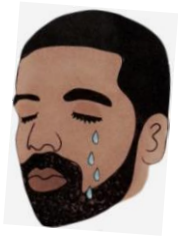
- Panics if there is an error.

- For example, Option enum.

- <T> means generic type. This means we don't specify that it is an integer or Boolean, or whatever.

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

```rust
fn main() {
    // Create a vector of numbers
    let numbers = vec![10, 20, 30];

    // 1️⃣ Safe unwrap: index exists
    let third = numbers.get(2).unwrap(); // Some(&30) → unwrap extracts value
    println!("Safe unwrap, third number: {}", third);

    // 2️⃣ Unsafe unwrap: index does NOT exist → causes panic
    let fifth = numbers.get(4).unwrap(); // None → panic!
    println!("This line will never run due to panic");

    // 3️⃣ Using match to handle Option safely
    match numbers.get(4) {
        Some(n) => println!("Match, fifth number: {}", n),
        None => println!("Match, fifth number not found"), // handles None safely
    }
}
```



RESULT ENUM

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

`Ok(T)`

Contains the success value

`Err(E)`

Contains the error value

Ok and Err are like wrapping paper around a Christmas present.

<T, E> is a generic type. This means we don't know what the specific types are, they could be Strings, Booleans, whatever.

GUESSING GAME – MATCHES

Lets improve this line:

```
let guess: u32 = guess.trim().parse().expect("Please type a number!");
```

We can use a **match** to ensure that a user types a number.

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```

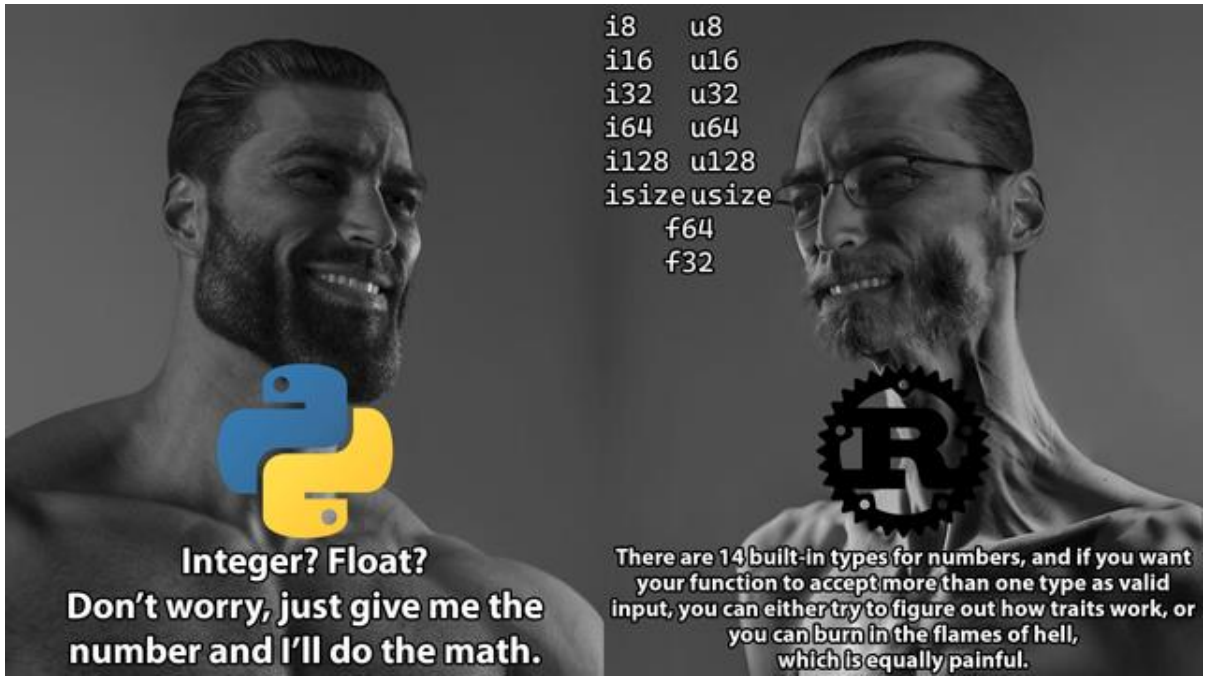
Because **match** forces handling of every scenario, strange bugs are less likely.

Note: _ Is a “catch all”.

GUESSING GAME

```rust
use std::cmp::Ordering;
use std::io;

use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {guess}");

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
architecture dependent	`isize`	`usize`



INTEGER DATA TYPES

INTEGER LITERALS

Number literals	Example
Decimal	`98_222`
Hex	`0xff`
Octal	`0o77`
Binary	`0b1111_0000`
Byte (`u8` only)	`b'A'`

← 98,222. Equivalent to 98222

FLOATING POINTS

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

NUMERIC OPERATIONS

```rust
fn main() {
    // addition
    let sum = 5 + 10;

    // subtraction
    let difference = 95.5 - 4.3;

    // multiplication
    let product = 4 * 30;

    // division
    let quotient = 56.7 / 32.2;
    let truncated = -5 / 3; // Results in -1

    // remainder
    let remainder = 43 % 5;
}
```

BOOLEANS

```
fn main() {
    let t = true;

    let f: bool = false; // with explicit type annotation
}
```

CHARACTER DATA TYPE

```
fn main() {
    let c = 'z';
    let z: char = 'Z'; // with explicit type annotation
    let heart_eyed_cat = '😻';
}
```

4 bytes. Use single quotes when declaring.

TUPLES

```rust
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {y}");
}
```

```rust
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

ARRAYS

All of these arrays are stored on the stack. **Why not the heap?**

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

```
let months = ["January", "February", "March", "April", "May", "June", "July",
              "August", "September", "October", "November", "December"];
```

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

`let a = [3; 5];` is the same as `let a = [3, 3, 3, 3, 3];`

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

```rust
fn main() {
    println!("Hello, world!");

    another_function();
}

fn another_function() {
    println!("Another function.");
}
```

FUNCTIONS
NO RETURN VALUE

FUNCTIONS WITH PARAMETERS

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {x}");
}
```

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

FUNCTIONS WITH RETURN VALUES

There is NO semicolon on ‘x + 1’. This is called an expression, not a statement. Statements end in semi-colons.

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

Typing ‘return’ is optional!

STATEMENTS AND EXPRESSIONS

- **Statements** are instructions that perform some action and do not return a value.
- **Expressions** evaluate to a resultant value.

```
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {y}");
}
```

```
The value of y is: 4
```

This expression:

```
{
    let x = 3;
    x + 1
}
```

is a block that, in this case, evaluates to `4`.

```rust
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

```
number is divisible by 3
```

```rust
fn main() {
    let number = 3;

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

```
condition was true
```

CONTROL FLOW - IF EXPRESSIONS

CONTROL FLOW - IF STATEMENTS

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {number}");
}
```

```
The value of number is: 5
```

CONTROL FLOW - LOOPS

```
fn main() {
    loop {
        println!("again!");
    }
}
```

CONTROL FLOW - LOOPS WITH RETURN VALUES

```rust
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");
}
```

```
The result is 20
```

CONTROL FLOW - LOOPS WITH LABELS

```rust
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

```
count = 0
remaining = 10
remaining = 9
count = 1
remaining = 10
remaining = 9
count = 2
remaining = 10
End count = 2
```

CONTROL FLOW - CONDITIONAL LOOPS

```rust
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{number}!");

        number -= 1;
    }

    println!("LIFTOFF!!!");
}
```

```
3!
2!
1!
LIFTOFF!!!
```

```rust
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("the value is: {element}");
    }
}
```

```
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

```rust
fn main() {
    for number in (1..4).rev() {
        println!("{number}!");
    }
    println!("LIFTOFF!!!");
}
```

```
3!
2!
1!
LIFTOFF!!!
```

CONTROL FLOW - FOR LOOPS