Breve intro a Rust

Agenda

- Por qué Rust?
- Instalación
- Sintaxis
- Packages, crates & modules
- Ownership & borrowing
- Testing
- Recursos

Acerca de mi,

- Ingeniero de sistemas
- Entusiasta del código abierto
- Mayormente escribo sobre Rust, web3, infra and node.js

Por qué Rust?

- *Rust is a system programming language*
- ***safety, concurrency, speed***
- *Zero cost abstractions*
- Empowering everyone to build reliable and efficient software.

Por qué Rust?

- ***Performance***: *increíblemente rápido y eficiente en términos de manejo de memoria (no GC ni runtime)*

- **Seguridad**: memory-safety and thread-safety (previene errores en tiempo de compilación como dangling-pointers, data-race, etc)

- ***Productividad:*** ***excelente documentación, errores de compilación amigables y herramientas de primer nivel:***

Origen

2006: Comenzó en el 2006 como un proyecto personal de Graydon Hoare (empleado de Mozilla)

2009: Mozilla comenzó a patrocinar el proyecto oficialmente en el 2009

2015: Se lanza oficialmente la versión 1.0

En la actualidad la última versión es la 1.59.0.

Instalación

Instalación

La forma recomendada es utilizar *rustup* (toolchain manager).

En macOs, Linux u otros Unix-like OS se puede instalar desde la terminal corriendo:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

En Windows se puede utilizar alguno de los métodos alternativos (https://forge.rust-lang.org/infra/other-installation-methods.html)

Nota: Rust tiene un proceso de release cada 6 semanas.

Instalación cont.

Una vez realizada la instalación vamos a tener acceso a:

- rustup (toolchain/version manager)

- rustc (compilador)

- cargo (package manager, build system, y mucho más)

Cargo

Cargo es una herramienta que permite realizar muchas tareas como compilar, descargar dependencias y muchas otras.

Algunos comandos útiles

- cargo new <nombre> /// Crea un nuevo proyecto
- cargo build [--release] /// Compila el proyecto
- cargo run /// Compila y ejecuta
- cargo test /// Ejecuta los tests
- cargo fmt /// Formatea el código
- cargo check /// Realiza un análisis sin compilar

Cargo cont.

Cargo puede ser extendido también, hay varios `*crates*` útiles que agregan funcionalidades a cargo

- cargo-edit /// Permite agregar or eliminar dependencias de forma simple
- cargo-chef /// Permite realizar builds de deps utilizando Docker.
- cargo-expand /// Muestra la *expansión* de macros.

https://crates.io/categories/development-tools::cargo-plugins?sort=downloads

Sintaxis

Sintaxis - Variables

- Las variables son declaradas utilizando el keyword *let* y por defecto son *inmutables*
- Para hacerlas *mutables* se utiliza *mut*
- Se las llama *bindings* y permite primero hacer la declaración y luego el *bind*
- Permite reutilizar el nombre de una variable, descartando la anterior (*shadow*)
- Implementa inferencia de tipos

Sintaxis - Variables

```rust
fn main() {
    // declare and bind
    let var_name: i32 = 12;
    // declare as mutable
    let mut another_var = 1;
    // mutate
    another_var = 2;

    println!("{}", var_name);
    println!("{}", another_var);
}
```

Sintaxis - Tipos de datos

Numéricos

- i8, i16, i32, i64
- u8, u16, u32, u64
- usize, isize /// depende del target de compilación:

Bool

Char (Unicode)

Compuestos:

- Tuplas
- Array [u8;3]

```
# let tup: (i32, f64, u8) = (500, 6.4, 1);
# let a: [i32; 5] = [1, 2, 3, 4, 5];
```

Sintaxis - Tipos de datos

Collections

Tipos *dinámicos*, no es necesario saber su *tamaño* en tiempo de compilación y pueden crecer y reducirse en tiempo de ejecución.

- Vec
- String
- HashMap

```
# let v: Vec<i32> = Vec::new();
# let s = String::from("initial contents");
# let mut scores = HashMap::new();
```

Sintaxis - Funciones

- Se definen con el *keyword* `fn`
- Nombre de la función
- Argumentos y tipo del valor de retornado.
- La última línea, *sin ;* y sin return es el valor de retorno

Sintaxis - Estructuras

Una estructura es un tipo que se compone de otros tipos. Los elementos de una estructura se denominan *fields*. Al igual que las tuplas, los campos de una estructura pueden tener diferentes tipos de datos, pero cada uno de ellos puede ser nombrado.

Sintaxis - Estructuras

Hay tres posibles variantes

- Con *nombres de campos*
- Tipo *tupla*
- Tipo *unit*

```
// Classic struct with named fields
struct Person {
    name: String,
    age: u8,
}

// A unit struct
struct Unit;

// A tuple struct
struct Pair(i32, f32);
```

Sintaxis - Enums

- Las enumeraciones son tipos que pueden ser cualquiera de varias variantes definidas.

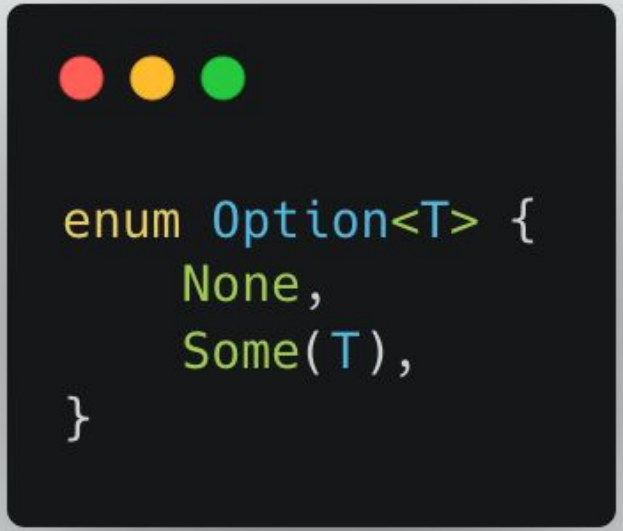
- Cada variante de enumeración puede tener datos que la acompañen.

```
enum WebEvent {
    // Unit
    WELoad,
    // Tuple
    WEKeys(String, char),
    // Classic struct
    WEClick { x: i64, y: i64 }
}
```

Sintaxis - Option

- Es un *enum* definido en la "*standard library*", permite representar un escenario muy común en el que un valor podría ser algo o podría no ser nada (`None`).

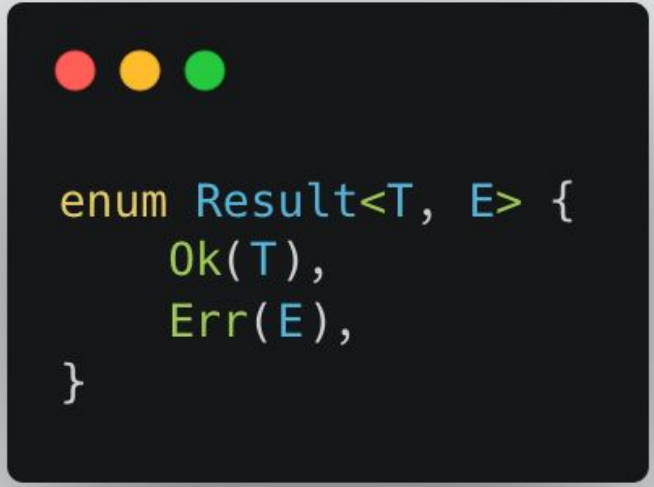
- `T` es un *generic*, que puede ser cualquier valor.



Sintaxis - Result

- Es un *enum* definido en la "*standard library*", permite representar un escenario en donde una función podría retornar con éxito o fallar

- `T` y `E` son *generics*
- `T` puede ser cualquier valor
- `E` puede ser cualquier error



Sintaxis - if/else

- Crea una bifurcación dependiendo de las condiciones

- Permite combinar *if* y *else* para formar una expresión *else if*.

- Los bloques if pueden actuar como expresiones. Todos los bloques deben devolver el mismo tipo

```
let formal = true;

// if used as an expression
let greeting = if formal {
    "Good day to you."
} else {
    "Hey!"
};
println!("{}", greeting)
```

Sintaxis - loop

- Ejecuta un bloque infinitamente hasta que explícitamente que se indique que se detenga.

- Permite retornar un valor

```rust
let mut counter = 0;

let result = loop {
    counter += 1;

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

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

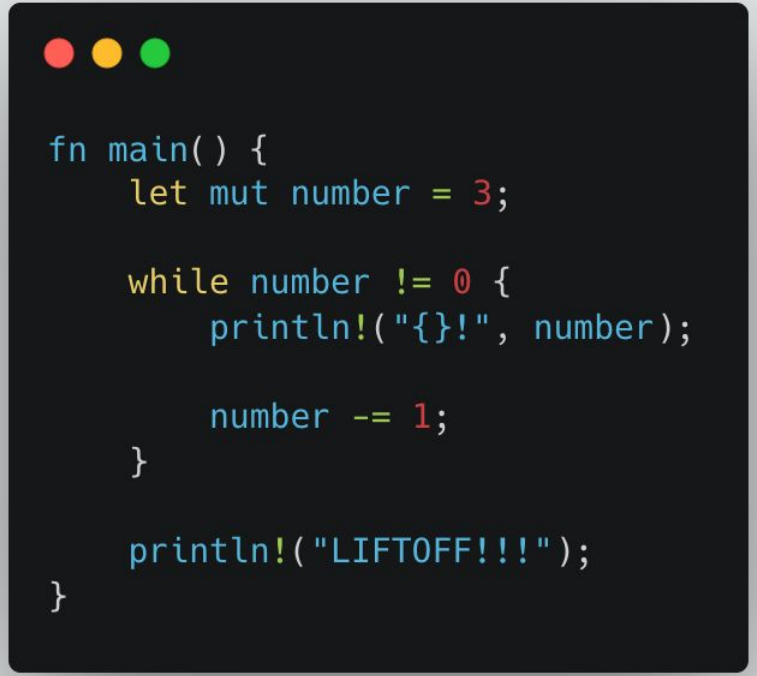
Sintaxis - loop cont.

- Permite especificar una etiqueta al `loop` que luego podemos usar con `break` o `continue` para ser aplicado sobre el loop marcado con el label.

```rust
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);
```

Sintaxis - While

- Permite ejecutar un bloque mientras que la condición se evalúe como verdadera.



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

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

        number -= 1;
    }

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

Sintaxis - for

- Permite iterar sobre un iterador (vectores, arrays, range, etc)

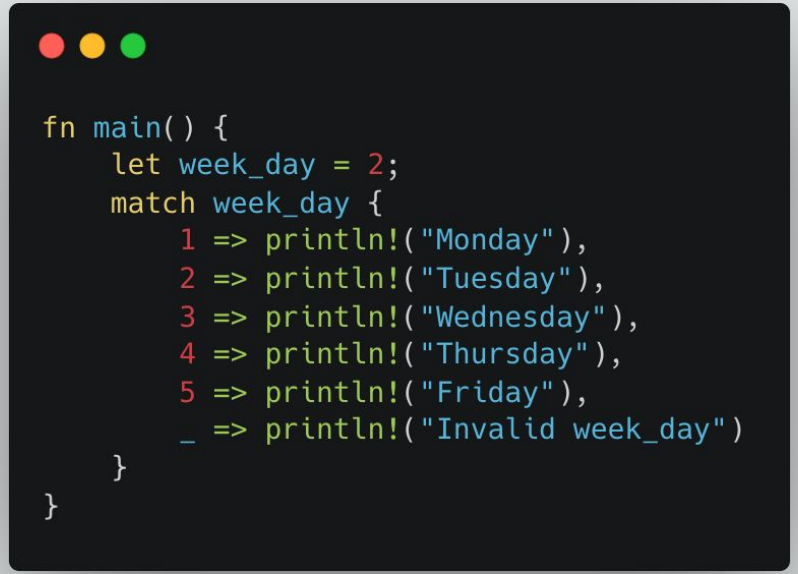
```rust
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("{}", element);
    }

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

Sintaxis - match

- Permite comparar un valor con una serie de patrones y luego ejecutar código según el patrón que coincida

- Similar a *Switch*

- Los patrones pueden estar formados por valores literales, nombres de variables, comodines, etc.



```
fn main() {
    let week_day = 2;
    match week_day {
        1 => println!("Monday"),
        2 => println!("Tuesday"),
        3 => println!("Wednesday"),
        4 => println!("Thursday"),
        5 => println!("Friday"),
        _ => println!("Invalid week_day")
    }
}
```

match cont.

```rust
fn main() {
    let number = 21;

    println!("Tell me about {}", number);
    match number {
        // Match a single value
        1 => println!("One!"),
        // Match several values
        2 | 3 | 5 | 7 | 11 => println!("This is a prime"),
        // Match an inclusive range
        13..=19 => println!("A teen"),

        // Handle the rest of cases
        _ => println!("Ain't special"),
    }
}
```

Sintaxis - if let

- Permite combinar `if` y `let`, para usar los valores que coinciden con el patrón.

- Se puede agregar `else` para manejar los casos donde el patrón no coincide

```
let config_max: Option<i32> = Some(3);
if let Some(max) = config_max {
    println!("Maximum {}", max);
} else {
    println!("Maximum not set");
}
```

Sintaxis - impl

- `impl` nos permite definir métodos a `enums` y `structs`

- Los declaramos usando `fn`

- Su primer parámetro siempre es `*self*`, que representa la instancia del `*struct*`

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

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

Sintaxis - impl cont.

- `impl` nos permite definir también `*funciones asociadas*`

- No tienen `*self*` como primer parámetro

- Se utiliza `::` junto al nombre del struct para *llamarla*

```
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle {
            width: size,
            height: size,
        }
    }
}

let sq = Rectangle::square(3);
```

Sintaxis - otros temas a explorar

- Traits
- Trait bounds
- Generics

Packages

crates

modules

Paquetes, *crates* y módulos

- Paquete:
  - Puede contener uno o más *crates*
  - Incluye la información sobre cómo compilar esos *crates*. *(Cargo.toml)*
- Crate:
  - Es una unidad de compilación, la unidad más pequeña sobre la que trabaja el compilador.
  - Una vez compilado, produce un ejecutable o una lib.
  - Contiene de forma implícita y sin nombre un *top-level module*
- Módulo:
  - Es una unidad de organización de código dentro de un *crate*
  - Puede tener definiciones recursivas que abarquen módulos adicionales.

Package and crates

- Contiene uno o más *crates*
- Como máximo un *lib*.
- Cuantos *binary crates* quieras
- Siempre al menos un crate

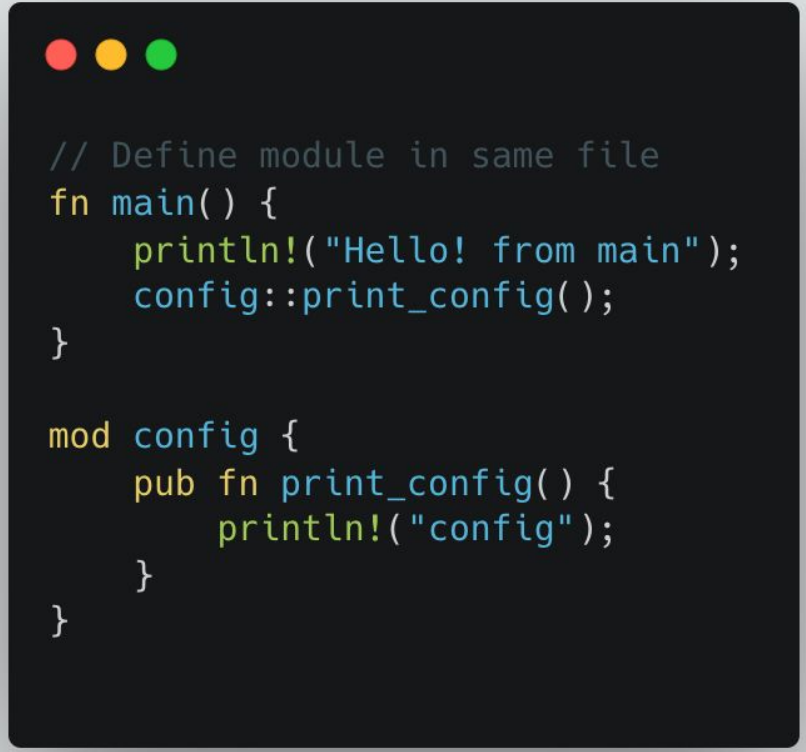
```
$ cargo new my-project
     Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs

$ cargo new --lib my-lib
     Created library `my-lib` package
$ ls my-lib/src
lib.rs
```

Módulos

- Permite organizar el código en espacios de nombres

- Se definen con el *keyword* `mod`

- Definen la privacidad de los *ítems* que contiene, por *default* todo es privado



```rust
// Define module in same file
fn main() {
    println!("Hello! from main");
    config::print_config();
}

mod config {
    pub fn print_config() {
        println!("config");
    }
}
```

Módulos cont.

- Necesitamos construir explícitamente el árbol de módulos

- No hay una asignación implícita al sistema de archivos

- Para agregar un archivo al árbol de módulos, debemos declarar ese archivo como un submódulo usando la palabra clave *mod*.

Módulos cont.

- Pero necesitamos declarar esto en un archivo diferente.

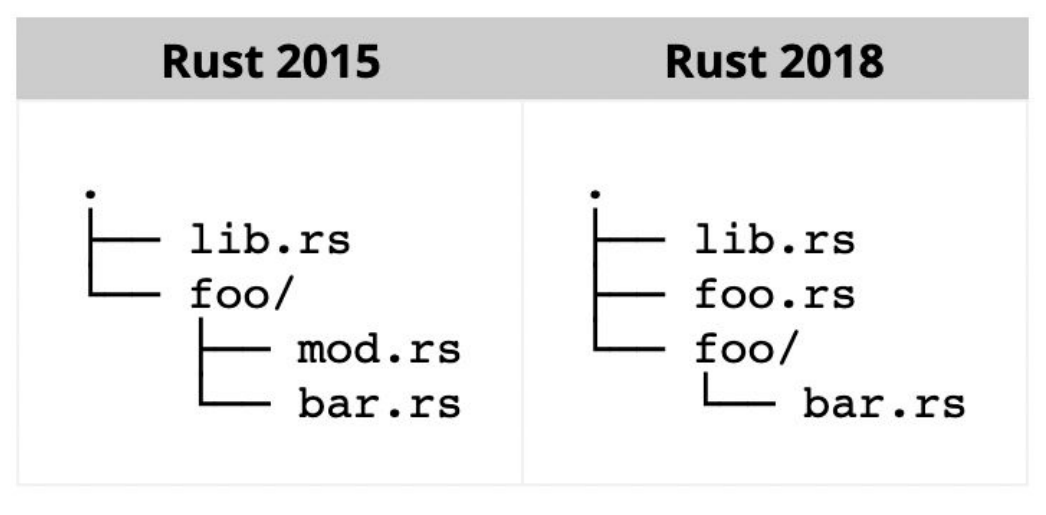
- Accedemos a los *items* que tenemos acceso (*pub*) *con la sintaxis* ::

```rust
// main.rs
mod config;
fn main() {
    println!("Hello! from main");
    config::print_config();
}

// config.rs
pub fn print_config() {
    println!("config");
}
```

Módulos cont.

- Módulos en subdirectorios antes tenían la restricción de contener el archivo *mod.rs*

- En la versión 2018 se elimina esta restricción.



Módulos super/use

- Podemos usar *super* en el *path* para referirnos al *parent*

- Podemos usar *use* para vincular el *path* a un nuevo nombre o alias.

```
// Cargo.toml
mod config;
mod routes;

use routes::health_routes;

fn main() {
    println!("Hello! from main");
    config::print_config();
    health_routes::print_health_route();
}

// routes/user_routes.rs
pub fn print_user_route() {
    println!("user_route");
}

// routes/health_routes.rs
pub fn print_health_route() {
    super::user_routes::print_user_route();
    println!("health_route");
}
```

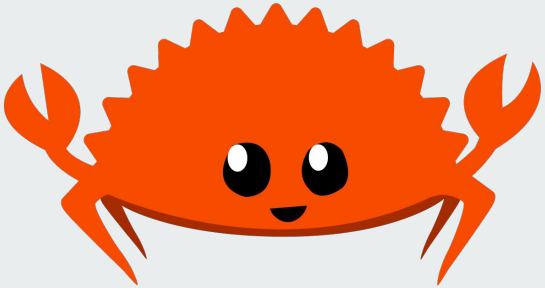
Módulos externos

- Las dependencias agregadas a *Cargo.toml* están disponibles globalmente

- También podemos usar *use* para acortar el *path*

```
// Cargo.toml
[dependencies]
rand = "0.8.5"

// main.rs
mod config;
mod routes;

use rand::random;
use routes::health_routes;
fn main() {
    let random_number: u8 = random();
    println!("{}", random_number);
    println!("Hello! from main");
    config::print_config();
    health_routes::print_health_route();
}
```

Ownership & borrowing



Ownership

- La forma en que maneja la memoria es lo que hace a Rust único

- Permite tener *Memory safety guarantees sin GC*

- *Ownership*, conjunto de reglas que que rigen cómo se administra la memoria.

```
// Ownership rules

1. Cada valor en Rust tiene
una variable que se llama
propietaria (owner).

2. Solo puede haber un owner
a la vez.

3. Cuando el owner queda
fuera del scope, el valor se
libera.
```

Ownership - Move

- Move semantics, **transferir** ownership de un binding a otro

- Una vez que se transfiere la propiedad, la variable anterior ya no es válida.

- Transferir la propiedad de un ítem es conocido como “moving”

```
fn main() {
    let s1 = String::from("Hello");
    let s2 = s1;
    println!("{}", s1); // Error!
}
```

```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:4:20
  |
2 |     let s1 = String::from("Hello");
  |         -- move occurs because `s1` has
type `String`, which does not implement the
`Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |     println!("{}", s1);
  |                    ^^ value borrowed
here after move
```

Ownership - Copy

- Los valores que implementan *Copy trait** no se mueven, sino que se copian.

- Tipos estándar (*know size), numéricos, bool, floating point, char*

- Si el tipo necesita que suceda algo especial cuando queda *out of scope,* no puede implementar *Copy*

```
fn main() {
    let s1 = 2;
    // Copy
    let s2 = s1;
    println!("{}", s1);
}
```

References & borrowing

- Es *como* un puntero que podemos seguir para acceder a los datos que pertenecen a *otra* variable.

- Se garantiza que la referencia apunte a un *valor válido*

- Llamamos *borrowing* a crear una referencia. Toma *prestado, no se adueña*

References - tipos

- Dos tipos de referencias:
  - Inmutable reference (shared)

    Puede leer el valor al que refiere pero no modificarlo.

  - Mutable reference (exclusive)

    Puede leer y modificar el valor que refiere

References - reglas

- Puede existir una *referencia mutable* o cualquier cantidad de *referencias inmutables*.

- Las referencias deben ser siempre válidas.

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

println!("{}, {}", r1, r2);
```

```
let mut s = String::from("hello");

{
    let r1 = &mut s;
}
// r1 goes out of scope here,
//so we can make a new reference.

let r2 = &mut s;
```

Otros temas

- *Lifetimes*
- *Interior mutability*
- *Concurrencia*

Testing

Testing

Los *test* son funciones que verifican que el código, que no es de prueba funciona de la manera esperada.

Normalmente realizan estas tres acciones:

- Configurar cualquier dato o estado necesario.
- Ejecutar el código que se desea probar.
- *assert* que los resultados son los esperados

Testing - unit

- ***Unit test*** son funciones marcadas con el atributo `#[test]`

- Se utilizan `macros` para realizar las *afirmaciones* (`assert`, `assert_eq`)

- Otros atributos útiles:
  - `#[should_panic]`
  - `#[ignore]`

```
//main.rs
fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[test]
fn add_works() {
    assert_eq!(add(1, 2), 3);
    assert_eq!(add(10, 12), 22);
    assert_eq!(add(5, -2), 3);
}
#[test]
#[should_panic]
fn add_fails() {
    assert_eq!(add(2, 2), 7);
}

#[test]
#[ignore]
fn add_negatives() {
    assert_eq!(add(-2, -2), -4)
}


--

cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
     Running unittests (target/debug/deps/test_example-75fbd798f92bb8ae)

running 3 tests
test add_negatives ... ignored
test add_works ... ok
test add_fails - should panic ... ok

test result: ok. 2 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

Testing - module

- Organizamos los test en módulos utilizando el atributo `#[cfg(test)]`

- El atributo `cfg` controla la compilación condicional, sólo compilará el módulo cuando se ejecute `cargo test`

- `use super::*;` es necesario para acceder a las funciones en el *outer module*

```rust
// main.rs
fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod add_function_tests {
    use super::*;

    #[test]
    fn add_works() {
        assert_eq!(add(1, 2), 3);
        assert_eq!(add(10, 12), 22);
        assert_eq!(add(5, -2), 3);
    }

    #[test]
    #[should_panic]
    fn add_fails() {
        assert_eq!(add(2, 2), 7);
    }

    #[test]
    #[ignore]
    fn add_negatives() {
        assert_eq!(add(-2, -2), -4)
    }
}
```

Testing - integración

- Se utiliza el directorio `/tests` (mismo nivel que `/src`)

- Compila cada archivo como un *crate* separado y no hace falta el atributo `#[cfg(test)]`

- **Sólo** se puede testear *library crates*, no se pueden testear funciones en **src/main.rs**

```rust
// adder.rs
fn add(a: i32, b: i32) -> i32 {
    a + b
}

// tests/add-test.rs
use test_example::adder::*;

#[test]
fn add_works() {
    assert_eq!(add(1, 2), 3);
    assert_eq!(add(10, 12), 22);
    assert_eq!(add(5, -2), 3);
}

#[test]
#[should_panic]
fn add_fails() {
    assert_eq!(add(2, 2), 7);
}
```

Recursos

- Rustlings, https://github.com/rust-lang/rustlings
- Playground, https://play.rust-lang.org
- Exercism, https://exercism.org/tracks/rust

Recursos

- Rust book, https://doc.rust-lang.org/stable/book/
- Microsoft “Rust first Steps”, https://docs.microsoft.com/en-us/learn/paths/rust-first-steps/
- “A gentle introduction to Rust”, https://stevedonovan.github.io/rust-gentle-intro/readme.html
- Rust by example, https://doc.rust-lang.org/stable/rust-by-example/
- FIUBA - Taller de programación I, “https://taller-1-fiuba-rust.github.io/clases.html”
- Comunidad de Rust Argentina, https://rust-lang-ar.github.io/

Preguntas?

Gracias! Pueden encontrarme en,

github.com/pepoviola

twitter.com/pepoviola

javierviola.com