Rust Nation Workshop

Intermediate Track

16/02/2023

Image by Chandler Cruttenden on Unsplash

Planning

9am - 10am	Generics & Closures
	Refreshment break
10:30am - 11am	Generics & Closures
11am - 12:30pm	Threads

1:30pm - 3pm	Macros
	Refreshment break
3:30pm - 5pm	Web Assembly

Generics

- Generics
- Closures
- Threads
- Macros
- Web Assembly

Polymorphism - Trait

```rust
trait Widget {
    fn width(&self) -> usize;

    fn draw_into(&self, buffer: &mut dyn Write);

    fn draw(&self) {
        let mut buffer = String::new();
        self.draw_into(&mut buffer);
        println!("{}", &buffer);
    }
}
```

- Defines a contract of methods that Types must implement
- Default methods can be implemented and overridden
- Can implement Trait in a crate for any type that can be outside of the crate

Polymorphism to remove duplication

```rust
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```rust
fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```rust
fn largest<T: PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

- Function **largest** is generic over **T**
- Any type **T** must implement **PartialOrd** trait
- Compiler is producing a copy of the function for each type using it
- Known as **Monomorphization**
- **Static Dispatch**

```rust
fn debug_value(value: &dyn Debug) {
    println!("Debug [{:?}]", value);
}
```

- Uses a virtual table to find the method to call
- Not sized as compiler does not know the concrete type
- **Dynamic Dispatch**

Generics - Trait bound

```rust
fn debug_value<T: Debug>(value: T) {
    println!("Debug [{:?}]", value);
}
```

```rust
fn debug_value(value: impl Debug) {
    println!("Debug [{:?}]", value);
}
```

```rust
fn debug_value<T>(value: T)
where
    T: Debug,
{
    println!("Debug [{:?}]", value);
}
```

Trait bound - Multiple Traits for a Generic Parameter

```rust
fn debug_value<T: Display + Debug>(value: T) {
    println!("Display {}, Debug [{:?}]", value, value);
}
```

```rust
fn debug_value(value: impl Display + Debug) {
    println!("Display {}, Debug [{:?}]", value, value);
}
```

```rust
fn debug_value<T>(value: T)
where
    T: Display + Debug,
{
    println!("Display {}, Debug [{:?}]", value, value);
}
```

```rust
fn debug_value<T>(value: T)
where
    T: Display,
    T: Debug,
{
    println!("Display {}, Debug [{:?}]", value, value);
}
```

Trait bound - Ergonomics with **where** keyword

```rust
fn is_equal_to_hello<T>(a: T) -> bool
where
    String: PartialEq<T>,
{
    String::from("hello") == a
}

fn main() {
    println!("{:?}", is_equal_to_hello("hello"));
    println!("{:?}", is_equal_to_hello(String::from("hello")));
}
```

Generics - impl Trait

```rust
use std::fmt::Debug;

fn generate_value<T: Debug>() -> T
{
    String::from("Hello")
}

fn main() {
    let value = generate_value();
    println!("Value: {:?}", value);
}
```

```
error[E0308]: mismatched types
 --> src/main.rs:5:5
  |
3 | fn generate_value<T: Debug>() -> T
  |                   -              -
  |                   |              |
  |                   |              expected `T` because
    of return type
  |                   |              help: consider using
    an impl return type: `impl Debug`
  |                   this type parameter
4 | {
5 |     String::from("Hello")
  |     ^^^^^^^^^^^^^^^^^^^^^ expected type parameter
  `T`, found struct `String`
  |
  = note: expected type parameter `T`
                     found struct `String`
```

Generics - impl Trait (opaque type)

```rust
use std::fmt::Debug;

fn generate_value() -> impl Debug {
    String::from("Hello")
}

fn main() {
    let value = generate_value();
    println!("Value: {:?}", value);
}
```

Generics - In Struct and Enum

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

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

Generics - Impl trait for structure with generics

```rust
impl Display for Point<u8> {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        writeln!(f, "{{x: {}, y: {}}}", self.x, self.y)
    }
}
```

Generics - Blanket implementation

```rust
impl<T> Display for Point<T>
where
    T: Display,
{
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        writeln!(f, "{{x: {}, y: {}}}", self.x, self.y)
    }
}
```

```rust
trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    //...
}
```

- Associated types make it easy to express a trait with some type used in the Output of a method
- Type is defined when implementing the Trait
- Only one type for a specific implementation

```rust
impl<T: Display> MyIterator for Vec<T> {
    type Item = String;

    fn next(&mut self) -> Option<Self::Item> {
        if self.is_empty() {
            None
        } else {
            Some(format!("Value: {}", self.remove(0)))
        }
    }
}
```

Traits - Associated types

```rust
impl<T> MyIterator for Vec<T> {
    type Item = T;

    fn next(&mut self) -> Option<Self::Item> {
        if self.is_empty() {
            None
        } else {
            Some(self.remove(0))
        }
    }
}
```

Traits - Associated types

```rust
trait HasAssociatedType {
    type SomeType;
}

impl<T: HasAssociatedType> MyIterator for Vec<T> {
    type Item = T::SomeType;

    //...
}
```

```rust
use std::fmt::Display;

fn print_all<T: Display>(list: Vec<T>) {
    for elt in list {
        println!("Value: {}", elt);
    }
}

fn main() {
    print_all(vec!["a", "b", "c"]);
}
```

- Vec only contains elements of the same type
- How can allow multiple types?

```rust
use std::fmt::Display;

fn print_all(list: Vec<Box<dyn Display>>) {
    for elt in list {
        println!("Value: {}", elt);
    }
}

fn main() {
    print_all(vec![
        Box::new("a"),
        Box::new(1),
        Box::new(true),
    ]);
}
```

- Trait objects are not Sized
- They need to be behind a pointer or a reference
- **Dynamic Dispatch**

Closures

- ~~Generics~~
- Closures
- Threads
- Macros
- Web Assembly

Closures

```rust
let list = vec![1, 2, 3, 4];
let even_numbers: Vec<_> = list
    .into_iter()
    .filter(
        |&item| {
            item % 2 == 0
        }
    )
    .collect();
println!("{:?}", even_numbers);
```

```rust
let list = vec![1, 2, 3, 4];
let even_numbers: Vec<_> = list
    .into_iter()
    .filter(|&item| item % 2 == 0)
    .collect();
println!("{:?}", even_numbers);
```

Closures - Capture

```rust
let list = vec![1, 3, 5, 6];
let divisor = 3;
let filtered: Vec<_> = list
    .into_iter()
    .filter(
        |&item| {
            item % divisor == 0
        }
    )
    .collect();
println!("{:?}", filtered);
```

Closures - Capture (Borrow or Ownership)

```rust
fn main() {
    let hello = String::from("Hello");
    let execute = || {
        println!("I borrow {}", hello);
    };
    execute();
    println!("Still available: {}", hello);
}
```

```rust
fn main() {
    let hello = String::from("Hello");
    let execute = || {
        let take_ownership = hello;
        println!("I own {}", take_ownership);
    };
    execute();
    // hello is not accessible anymore
    // println!("Not available: {}", hello);
}
```

Closures - Ownership

```rust
fn main() {
    let list = vec![1, 2, 3];
    let print_list = || {
        for i in list {
            println!("{}", i);
        }
    };
    print_list();
    println!("Printed {:?}", list);
}
```

for loop is calling `into_iter(self)`

```
error[E0382]: borrow of moved value: `list`
 --> src/main.rs:9:30
  |
2 |     let list = vec![1, 2, 3];
  |         ---- move occurs because `list` has type
  `Vec<i32>`, which does not implement the `Copy` trait
3 |     let print_list = || {
  |                      -- value moved into closure here
4 |         for i in list {
  |                  ---- variable moved due to use in closure
...
9 |     println!("Printed {:?}", list);
  |                              ^^^^ value borrowed here
  after move
```

Closures - Ownership

```rust
fn main() {
    let list = vec![1, 2, 3];
    let print_list = || {
        for i in &list {
            println!("{}", i);
        }
    };
    print_list();
    println!("Printed {:?}", list);
}
```

for loop is calling `iter(&self)`

Closures - Force ownership

```rust
fn main() {
    let say_hi = build_hi();

    say_hi();
}

fn build_hi() -> impl Fn() {
    let name = String::from("John");
    || {
        println!("Hello {}", name);
    }
}
```

```
error[E0373]: closure may outlive the current function, but it
   borrows `name`, which is owned by the current function
  --> src/main.rs:10:5
   |
10 |     || {
   |     ^^ may outlive borrowed value `name`
11 |         println!("Hello {}", name);
   |                              ---- `name` is borrowed here
   |
note: closure is returned here
  --> src/main.rs:10:5
   |
10 | /     || {
11 | |         println!("Hello {}", name);
12 | |     }
   | |_____^
help: to force the closure to take ownership of `name` (and
  any other referenced variables), use the `move` keyword
   |
10 |     move || {
   |     ++++
```

Closures - Force ownership

```
fn build_hi() -> impl Fn() {
    let name = String::from("John");

    move || {
        println!("Hello {}", name);
    }
}
```

```
fn main() {
    let name = String::from("John");
    print_value(|| {
        format!("Hello {}", name)
    });
}

fn print_value<T>(get_value: T)
where
    T: Fn() -> String,
{
    println!("{}", get_value());
}
```

- Closure types are dynamically created depending on the environment they capture
- Dynamically implements these Traits
  - **Fn()** : Closures that are Immutable
  - **FnMut()** : Closures that are Mutable
  - **FnOnce()** : Closures that consume themselves

Closures - **Fn() Traits** (Immutable)

```rust
fn main() {
    let name = String::from("John");
    let say_hi = || {
        println!("Hello {}", name);
    };
    exec(say_hi);
}
```

```rust
fn main() {
    let name = String::from("John");
    let say_hi = move || {
        println!("Hello {}", name);
    };
    exec(say_hi);
}
```

```rust
fn exec<F: Fn()>(f: F) {
    f();
    f();
}
```

Returns

> Hello John
> Hello John

Closures - **FnMut() Traits (Mutable)**

```rust
fn main() {
    let mut name = String::from("John");
    let say_hi = || {
        name.push_str(" Doe");
    };
    exec(say_hi);
    println!("Hello {}", name);
}
```

```rust
fn main() {
    let mut name = String::from("John");
    let say_hi = move || {
        name.push_str(" Doe");
        println!("Hello {}", name);
    };
    exec(say_hi);
}
```

```rust
fn exec<F: FnMut()>(mut f: F) {
    f();
    f();
}
```

Returns

> Hello John Doe

> Hello John Doe Doe

Closures - **FnOnce()** Traits (Consume)

```rust
fn main() {
    let name = String::from("John");
    let say_hi = || {
        drop(name);
    };
    exec(say_hi);
}
```

```rust
fn exec<F: FnOnce()>(f: F) {
    f();
}
```

Closures - **Traits definitions**

```rust
pub trait FnOnce<Args> {
    // ...
    type Output;

    // ...
    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
```

```rust
pub trait FnMut<Args>: FnOnce<Args> {
    // ...
    extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output;
}
```

```rust
pub trait Fn<Args>: FnMut<Args> {
    // ...
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}
```

Closures - Trait objects

```rust
fn main() {
    let name = String::from("John");
    print_value(&|| {
        format!("Hello {}", name)
    });
}

fn print_value(get_value: &dyn Fn() -> String)
{
    println!("{}", get_value());
}
```

Closures - Traits syntax

```rust
fn execute<T>(work: T)
    where
        T: Fn(),
```

```rust
fn execute<T>(work: T)
    where
        T: Fn(String),
```

```rust
fn execute<T>(work: T)
    where
        T: Fn() -> String,
```

Closures - Stored in Structs

```rust
struct CallMe<F> {
    f: F
}

fn main() {
    let cm = CallMe {
        f: || {
            println!("Hello John");
        }
    };
    (cm.f)();
}
```

Implementing Map and Filter iterators

Clone the following repository:

https://github.com/codurance/rust-nation-intermediate-workshop

Open the folder **1_generics-traits** with your preferred IDE

Threads

- ~~Generics~~
- ~~Closures~~
- Threads
- Macros
- Web Assembly

Threads

```
thread::spawn(|| {
    println!("Hello thread");
});
```

Threads - JoinHandle<()>

```rust
let handle = thread::spawn(|| {
    println!("Hello thread");
});

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

Threads - JoinHandle<String>

```rust
let handle = thread::spawn(|| {
    println!("Hello thread");

    String::from("Hello")
});

if let Ok(result) = handle.join() {
    println!("{} from thread", result);
}
```

```rust
let mut handles = vec![];
for _ in 0..10 {
    handles.push(thread::spawn(|| {
        format!("Hello")
    }));
}

for handle in handles {
    if let Ok(result) = handle.join() {
        println!("{} from thread ", result);
    }
}
```

Threads

```rust
let mut handles = vec![];
for i in 0..10 {
    handles.push(thread::spawn(|| {
        format!("Hello {}", i)
    }));
}

for handle in handles {
    if let Ok(result) = handle.join() {
        println!("{} from thread ", result);
    }
}
```

```
error[E0373]: closure may outlive the current function, but it borrows `i`,
 which is owned by the current function
 --> src/main.rs:7:36
  |
7 |         handles.push(thread::spawn(|| {
  |                                    ^^ may outlive borrowed value `i`
8 |             format!("Hello {}", i)
  |                                 - `i` is borrowed here
  |
note: function requires argument type to outlive `'static`
```

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

- **F** can only capture values that are thread safe
- **F** can only own values or hold references that can live until the end of the program

Threads

```
 --> src/main.rs:7:22
  |
7 |           handles.push(thread::spawn(|| {
  |  ______________________^
8 | |             format!("Hello {}", i)
9 | |         }));
  | |_________^
help: to force the closure to take ownership of `i` (and any other referenced
 variables), use the `move` keyword
  |
7 |         handles.push(thread::spawn(move || {
  |                                    ++++
```

```rust
let count = 2;
let handle = thread::spawn(move || {
    println!("{}", count);
});
handle.join().unwrap();
println!("Done counting until {}", count);
```

This compiles because **i32** implements the **Copy** auto trait ✅

```rust
let hello = String::from("Hello");
let handle = thread::spawn(move || {
    println!("{}", hello);
});
handle.join().unwrap();
println!("Done saying {}", hello);
```

This does not compile because **String** does not implement **Copy.** You cannot reuse it later ❌

```
error[E0382]: borrow of moved value: `hello`
  --> src/main.rs:10:32
   |
5  |     let hello = String::from("Hello");
   |         ----- move occurs because `hello` has type `String`, which does not implement the `Copy` trait
6  |     let handle = thread::spawn(move || {
   |                                ------- value moved into closure here
7  |         println!("{}", hello);
   |                        ----- variable moved due to use in closure
...
10 |     println!("Done saying {}", hello);
   |                                ^^^^^ value borrowed here after move
   |
```

```
let hello = Arc::new(String::from("Hello"));
let hello_cloned = hello.clone();
let handle = thread::spawn(move || {
    println!("{}", hello_cloned);
});

handle.join().unwrap();
println!("Done saying {}", hello);
```

- **Arc** is a thread safe version of the smart pointer **Rc**
  - **Atomic Reference Counted**
- Allows you to share an immutable piece of data between threads

Threads - Interior mutability with Mutex

```rust
let hello = Arc::new(Mutex::new(String::from("Hello")));

let hello_cloned = hello.clone();
let handle = thread::spawn(move || {
    let mut hello = hello_cloned.lock().unwrap();
    println!("Updating {}", hello);
    hello.push('a');
});

handle.join().unwrap();
println!("Done saying {}", hello.lock().unwrap());
```

Threads - Interior mutability with RwLock

```
let hello = Arc::new(RwLock::new(String::from("Hello")));

let hello_cloned = hello.clone();
let handle = thread::spawn(move || {
    let mut hello = hello_cloned.write().unwrap();
    println!("Updating {}", hello);
    hello.push('a');
});

handle.join().unwrap();
println!("Done saying {}", hello.read().unwrap());
```

Mutex<T>	RwLock<T>
Only one reader or writer at a time	Multiple readers or one writer at a time
T just need to be Send	**T** needs to be Send and Sync

Send	Sync
Indicates that a type is safe to be send into another thread	Indicates that a type is safe to be shared between threads
	T is **Sync** only if **&T** is **Send**

These traits are auto-trait. This means types are automatically implementing these, as long as they are composed with other types that implement these Trait as well.

Threads - Examples of !Send and !Sync types

!Send	!Sync
`std::rc::`**`Rc<T>`**	`std::rc::`**`Rc<T>`**
	`std::cell::`**`Cell<T>`**
	`std::cell::`**`RefCell<T>`**
	`std::sync::mpsc::`**`Receiver<T>`**

Threads - Send & Sync in trait bounds

```rust
fn print_in_thread<T>(value: T) -> JoinHandle<()>
where
    T: Display,
{
    thread::spawn(move || {
        println!("Saying {}", value);
    })
}
```

```
error[E0277]: `T` cannot be sent between threads safely
   --> src/main.rs:22:19
    |
22  |         thread::spawn(move || {
    |         ------------- ^------
    |         |             |
    |  _______|_____________within this `[closure@src/main.rs:22:19: 22:26]`
    | |       |
    | |       required by a bound introduced by this call
23  | |           println!("Saying {}", value);
24  | |       })
    | |_____^ `T` cannot be sent between threads safely
    |
note: required because it's used within this closure
```

Threads - Send & Sync in trait bounds

```rust
fn print_in_thread<T>(value: T) -> JoinHandle<()>
where
    T: Display + Send,
{
    thread::spawn(move || {
        println!("Saying {}", value);
    })
}
```

```
error[E0310]: the parameter type `T` may not live long enough
  --> src/main.rs:22:5
   |
22 | /     thread::spawn(move || {
23 | |         println!("Saying {}", value);
24 | |     })
   | |______^ ...so that the type `T` will meet its required lifetime bounds
   |
help: consider adding an explicit lifetime bound...
   |
20 |     T: Display + Send + 'static,
   |                       +++++++++
```

Threads - Send & Sync in trait bounds

```rust
fn print_in_thread<T>(value: T) -> JoinHandle<()>
where
    T: Display + Send + 'static,
{
    thread::spawn(move || {
        println!("Saying {}", value);
    })
}
```

Threads - Communication with Channels

```rust
use std::sync::mpsc::channel;

let (sender, receiver) = channel();
```

Multi-**p**roducer, **s**ingle-**c**onsumer FIFO queue communication primitives.

Threads - Communication with Channels

```rust
let receiver = {
    let (sender, receiver) = channel();

    let mut handles = Vec::new();
    for i in 0..10 {
        let sender_cloned = sender.clone();
        handles.push(thread::spawn(move || {
            sender_cloned
                .send(format!("Hello {}", i))
                .unwrap();
        }));
    }

    receiver
};

while let Ok(value) = receiver.recv() {
    println!("Received {}", value);
}
```

```rust
let (sender, receiver) = channel();

let handle = thread::spawn(move || {
    let value = receiver.recv().unwrap();
    println!("Received {}", value);
});
sender.send(String::from("Hello")).unwrap();

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

Threads - Channels

```rust
let (sender, receiver) = channel();

let mut handles = Vec::new();
for _ in 0..10 {
    handles.push(thread::spawn(move || {
        let value = receiver.recv().unwrap();
        println!("Received {}", value);
    }));
}

for _ in 0..10 {
    sender.send(String::from("Hello")).unwrap();
}

for handle in handles {
    handle.join().unwrap();
}
```

```
error[E0382]: use of moved value: `receiver`
  --> src/main.rs:10:36
   |
6  |     let (sender, receiver) = channel();
   |                  -------- move occurs because `receiver`
   has type `std::sync::mpsc::Receiver<String>`, which does
   not implement the `Copy` trait
...
10 |         handles.push(thread::spawn(move || {
   |                                    ^^^^^^^ value moved
   into closure here, in previous iteration of loop
11 |             let value = receiver.recv().unwrap();
   |                         -------- use occurs due to use in
   closure
```

Threads - Channels

```rust
let (sender, receiver) = channel();
let receiver = Arc::new(receiver);

let mut handles = Vec::new();
for _ in 0..10 {
    let receiver_cloned = receiver.clone();
    handles.push(thread::spawn(move || {
        let value = receiver_cloned.recv().unwrap();
        println!("Received {}", value);
    }));
}
// ...
```

```
error[E0277]: `std::sync::mpsc::Receiver<String>` cannot be shared
 between threads safely
   --> src/main.rs:12:36
    |
12  |         handles.push(thread::spawn(move || {
    |  ______________________-------------_^
    | |                     |
    | |                     required by a bound introduced by this call
13  | |           let value = receiver_cloned.recv().unwrap();
14  | |           println!("Received {}", value);
15  | |       }));
    | |_________^ `std::sync::mpsc::Receiver<String>` cannot be shared
   between threads safely
    |
    = help: the trait `Sync` is not implemented for
   `std::sync::mpsc::Receiver<String>`
```

Threads - Channels

```rust
let (sender, receiver) = channel();
let receiver = Arc::new(Mutex::new(receiver));

let mut handles = Vec::new();
for i in 0..10 {
    let receiver_cloned = receiver.clone();
    handles.push(thread::spawn(move || {
        let value = receiver_cloned.lock().unwrap().recv().unwrap();
        println!("[Thread {}] Received {}", i, value);
    }));
}
// ...
```

Implementing Thread Pool

Open the **2_threads** folder

Macros

- ~~Generics~~
- ~~Closures~~
- ~~Threads~~
- Macros
- Web Assembly

What *are* macros?

https://matt.si/2022-05/macros-what-why-how/

What are they and why would I use them?

Macros

Macros are functions that output code- aka code generation. They run at compile time and replace some pieces of code with code they've generated.
In C/C++ macros were infamously unsafe. You could replace any keyword, such as replacing true with false. But in Rust macros have many more safety guarantees.

Image by Егор Камелев, Unsplash

Domain-specific Languages (DSL)

With macros you can specify a bespoke syntax outside of regular Rust syntax for your specific needs

```
fn main() {
    calculate!(eval 1 + 2);
}
```

Example from “Rust By Example”

Deduplication / DRY

Automatically generate verbose, boilerplate code to save you time & reduce duplication

```rust
fn main() {
    let mut m = ::std::collections::HashMap::new();
    m.insert(1, "one");
    m.insert(2, "two");
    //
}
```

```rust
fn main() {
    let names = map!{ 1 => "one", 2 => "two" };
}
```

Example from StackOverflow,
users/155423/shepmaster

Deduplication / DRY

Automatically generate verbose, boilerplate code to save you time & reduce duplication

```
#[derive(ToUrl)]
struct Request {
    response_type: String,
    client_id: String
}
```

Example adapted from DareDevDiary, niilz

Variadic Interfaces

Interfaces, such as `println!`, that take a variable number of parameters

```
fn main() {
    calculate! {
        eval 1 + 2,
        eval 3 + 4,
        eval (2 * 3) + 1
    }
}
```

Example from “Rust By Example”

Function-like

Compile

```rust
fn main() {
    calculate!(eval 1 + 2);
}
```

```rust
fn main() {
    {
        let val: usize = 1 + 2;
        {
            ::std::io::_print(::core::fmt::Arguments::new_v1(
                &["", " = ", "\n"],
                &[
                    ::core::fmt::ArgumentV1::new_display(&"1 + 2"),
                    ::core::fmt::ArgumentV1::new_display(&val)
                ],
            ))
        }
    }
}
```

Outputs “1 + 2 = 3”

Mid-Level Intermediary Language

One of several intermediary languages Rust compiles to before compiling to assembly

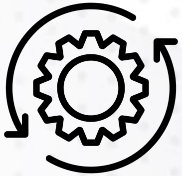
What are they and why would I use them?

Attribute macro

```rust
#[logfn_inputs(Info)]
#[logfn(ok = "TRACE", err = "ERROR")]
fn call_isan(num: &str) -> Result<Success, Error> {
    if num.len() >= 10 && num.len() <= 15 {
        Ok(Success)
    } else {
        Err(Error)
    }
}
```

Example from LogRocket blog

Compile



```rust
fn call_isan(num: &str) -> Result<Success, Error> {
    let result = (move || {
        if num.len() >= 10 && num.len() <= 15 {
            Ok(Success)
        } else {
            Err(Error)
        }
    })();

    result
        .map(|result| {
            {
                let lvl = log::Level::Trace;
                if lvl <= ::log::STATIC_MAX_LEVEL && lvl < ::log::max_level() {
                    ::log::__private_api_log(
                        ::core::fmt::Arguments::new_v1(
                            &["call_isan() => "],
                            &[::core::fmt::ArgumentV1::new_debug(&result)],
                        ),
                        lvl,
                        &("playground", "playground", "src/main.rs", 32u32),
                        ::log::__private_api::Option::None,
                    );
                }
            };
            result
        })
        .map_err(|err| {
            {
                // error handling, omitted
            };
            err
        })
}
```

What are they and why would I use them?

Derive macro

```rust
#[derive(ToUrl)]
struct Request {
    response_type: String,
    client_id: String
}
```

Example from DareDevDiary, niilz

Compile

```rust
pub struct Request {
   response_type: String,
   client_id: String,
}
impl ToUrl for Request {
   pub fn to_url(&self, base_url: String) -> String {
       let url = {
           let res = ::alloc::fmt::format(::core::fmt::Arguments::new_v1(
               &[""],
               &[::core::fmt::ArgumentV1::new_display(&base_url)],
           ));
           res
       } + &{
           let res = ::alloc::fmt::format(::core::fmt::Arguments::new_v1(
               &["", "=", ""],
               &[
                   ::core::fmt::ArgumentV1::new_display(&"response_type"),
                   ::core::fmt::ArgumentV1::new_display(&self.response_type),
                   ::core::fmt::ArgumentV1::new_display(&"&"),
               ],
           ));
           res
       } + &{
           let res = ::alloc::fmt::format(::core::fmt::Arguments::new_v1(
               &["", "=", ""],
               &[
                   ::core::fmt::ArgumentV1::new_display(&"client_id"),
                   ::core::fmt::ArgumentV1::new_display(&self.client_id),
                   ::core::fmt::ArgumentV1::new_display(&"&"),
               ],
           ));
           res
       }; // Details omitted
       url
   }
}
```

Declarative Macros

The simpler and easier to use type of macro. Type a macro function call denoted by an exclamation mark, like: `my_macro!(params)` and the callsite will be replaced by the result of the macro. This is called *function-like*.

Procedural Macros

Lower level macros that are more complex but provide more flexibility and control. While declarative can only create new code, procedural macros can modify existing code.

They can also be executed via:

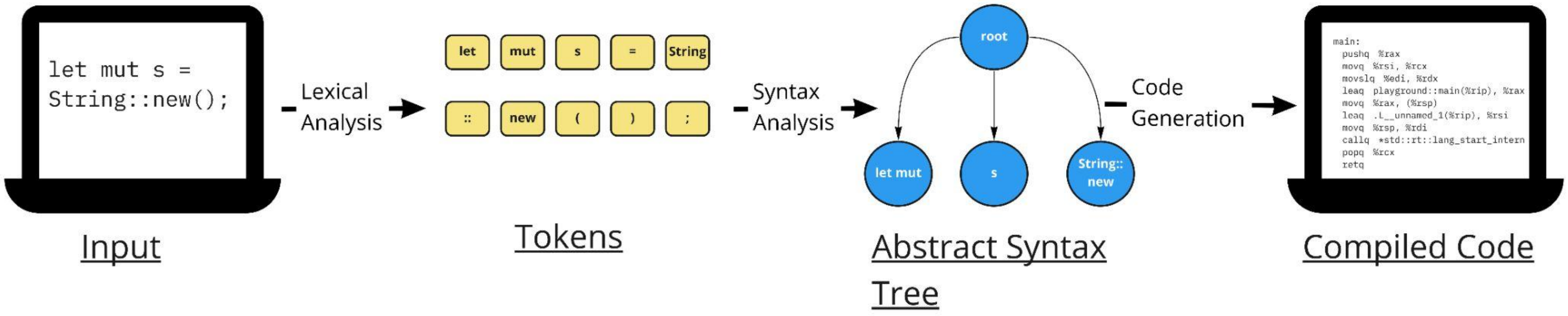
- Function-like
  - The same as Declarative Macros
- As an attribute to a unit of code
  - eg `#[test]`
- As a derive attribute
  - eg `#[derive(Serialize)]`

How Do Macros Work?

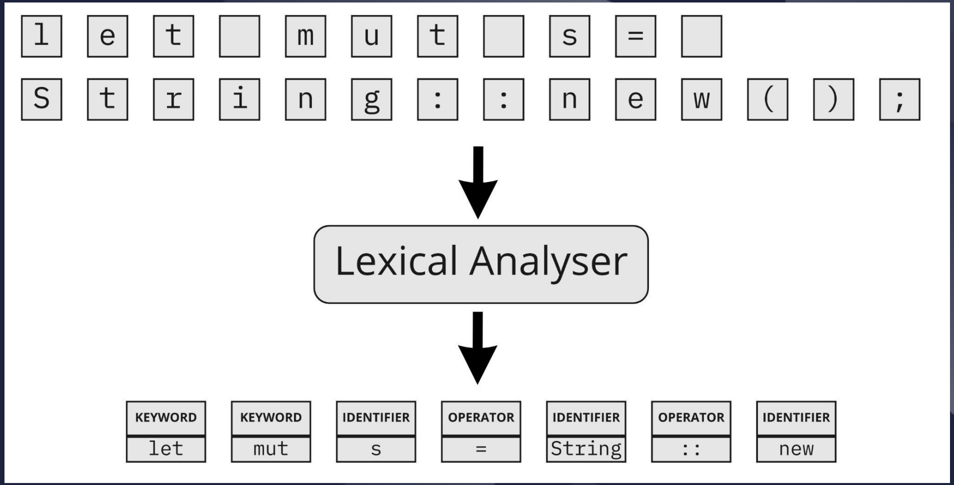
How Do Macros Work?

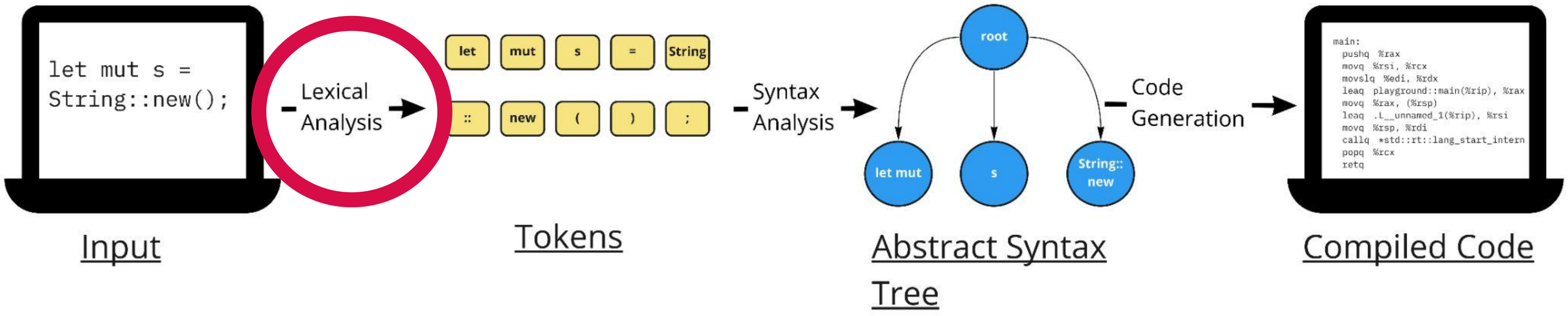
- Rust compilation begins with lexical analysis which splits the input into *tokens*.
- These tokens are then arranged into an abstract syntax tree
- This tree is used to generate the code.

NB: Very simplified- in reality Rust uses intermediary languages



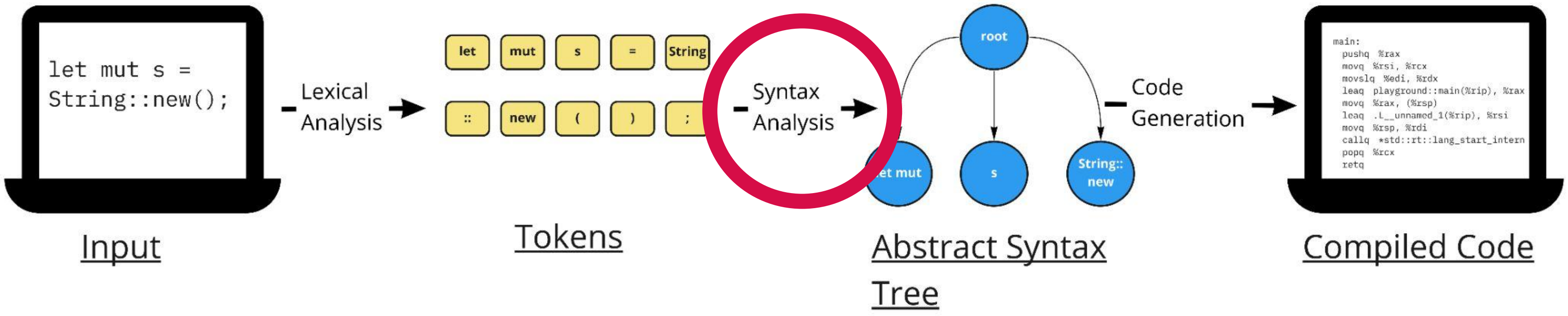
How Do Macros Work?





How Do Macros Work?

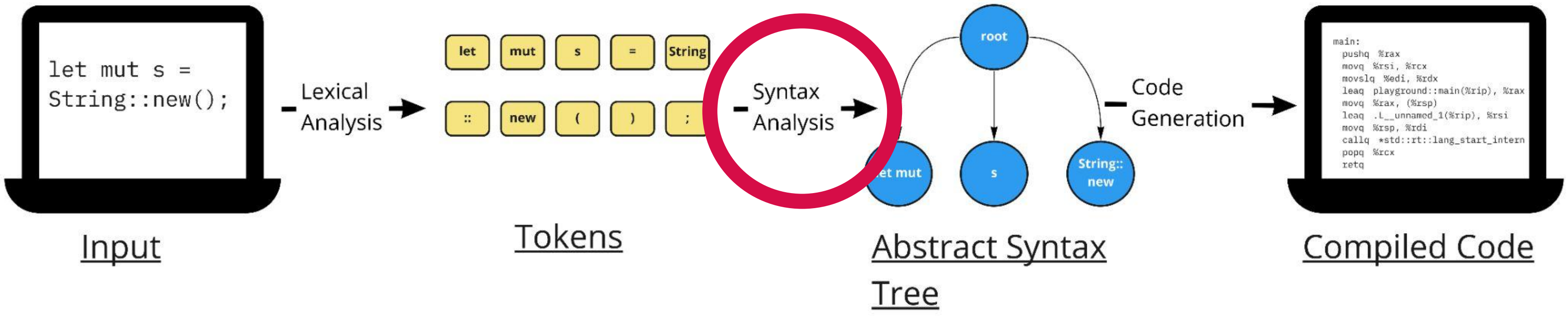
- Macros are evaluated *after* lexical analysis is complete, during syntax analysis.
  - Macros can add, modify or remove tokens from the set of tokens.
- When macros generate code they are working at a higher level of abstraction than raw characters and strings. Instead, they generate *tokens*



How Do Macros Work?

This has some impact on what macros can do:

- They **cannot** accept anything that would not pass lexical analysis because that step has already run.
    - For example, all open braces most have closing braces because that is managed by lexical analysis.
- Macros can accept parameters that would be illegal Rust syntax because the syntax analysis step has not run yet
- Tokens produced by macros **must** follow valid Rust syntax



Declarative macros

Syntax

- `macro_rules!` declares a new declarative macro
- Followed by name of macro and curly brace entering new block
- Block contains one or more *matchers*
- Each matcher is paired with a function that will determine the resulting tokens for that matcher, known as a *transcriber*.

```
macro_rules! print_result {
    ($expression:expr) => {
        // `stringify` will expand the expression
        // *as it is* into a string.
        println!(
            "{:?} = {:?}",
            stringify!($expression),
            $expression
        );
    };
}
```

Example from “Rust By Example”

Matcher syntax

- Matchers match the *argument* tokens in the macro callsite, and bind some of these token as *metavariables* that can be accessed by the transcriber
    - Metavariables are denoted by `$`
- Metavariables are denoted with `$` and must have a *token type* denoted with `:`, like so: `$argument_name:token_type`
- Hence, in this example instead of using arguments separated by commas the matcher is able to separate the arguments by `; and`

```
macro_rules! test {
    ($left:expr; and $right:expr) => {
        println!(
            "{:?} and {:?} is {:?}",
            stringify!($left),
            stringify!($right),
            $left && $right
        );
    };
}

fn main() {
    test!(1i32 + 1 == 2i32; and 2i32 * 2 == 4i32);
}
```

Example from “Rust By Example”

Matcher syntax (2)

- This example uses the `expr` token type, which matches any *expression* but there are many others such as…
  - `item` a struct, function or module
  - `stmnt` a statement
  - `block` a statement surrounded by curly braces
  - `pat` a pattern
  - `ty` a variable type name (eg u32)
  - `tt` a token tree
- The compiler ensures that all calls to the macro use the correct token types for each argument

```
macro_rules! map {
    ( $( $key:expr => $value:expr ),+ ) => {
        {
            let mut m = ::std::collections::HashMap::new();
            $(
                m.insert($key, $value);
            )+
            m
        }
    };
}

fn main() {
    let names = map!{ 1 => "one", 2 => "two" };
}
```

Example from StackOverflow, users/155423/shepmaster

Multiple Matchers

- When invoked the compiler will check the literal argument tokens from the callsite against every matcher starting from the top. It stops when it hits the first one that matches, returning the result the transcriber assigned to that matcher.
  - Some patterns may match multiple matchers, so the order can be important
- As such macros can be overloaded, accepting multiple different sets of parameters
- Each matcher/transcriber key/pair ends with a semicolon

```rust
macro_rules! test {
    ($left:expr; and $right:expr) => {
        println!(
            "{:?} and {:?} is {:?}",
            stringify!($left),
            stringify!($right),
            $left && $right
        );
    };
    ($left:expr; or $right:expr) => {
        println!(
            "{:?} and {:?} is {:?}",
            stringify!($left),
            stringify!($right),
            $left || $right
        );
    };
}

fn main() {
    test!(1i32 + 1 == 2i32; and 2i32 * 2 == 4i32);
    test!(true; or false);
}
```

Example from “Rust By Example”

Fully qualified namespaces

- When a macro generates code, you can't be sure that the scope the generated code is being inserted into will have all the imports you need
- Especially true of exporting macros to be used by someone consuming your library
- Best practice is to use fully-qualified namespaces always inside macros, to avoid relying on imports altogether

```rust
macro_rules! find_min {
    ($x:expr) => ($x);
    ($x:expr, $( $y:expr ),+) => {
        ::std::cmp::min($x, find_min!($( $y ),+))
    }
}

fn main() {
    println!("{}", find_min!(1));
    println!("{}", find_min!(1 + 2, 2));
    println!("{}", find_min!(5, 2*3, 4));
}
```

Example from "Rust By Example"

Declarative Macros - Hygiene

```rust
macro_rules! let_foo {
    ($x:expr) => {
        let foo = $x;
        println!("Macro foo: {}", foo);
    };
}

fn main() {
    let foo = 1;
    let_foo!("a");
    println!("Original foo: {}", foo);
}
```

Example inspired from Rust for Rustaceans,
Jon Gjengset

When to use... **Declarative macros**

- Reduce boilerplate duplication
- Test generation
- Domain Specific Languages

```
macro_rules! test_battery {
    ($( $t:ty as $name:ident ),*) => {
        $(
            mod $name {
                #[test]
                fn frobnified() { test_inner::<$t>(1, true) }
                #[test]
                fn unfrobnified() { test_inner::<$t>(2, false) }
            }
        )*
    }
}

test_battery! {
    u8 as u8_tests,
    i128 as i128_tests
}
```

Example adapted from Rust for Rustaceans,
Jon Gjengset

Implementing a JSON like structure parser macro

Open the **3_macros** and follow through these steps.

Web Assembly

- ~~Generics~~
- ~~Closures~~
- ~~Threads~~
- ~~Macros~~
- Web Assembly

“WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications.”

[WebAssembly org]

“WebAssembly has huge implications for the web platform — it provides a way to run code written in multiple languages on the web at near native speed, with client apps running on the web that previously couldn't have done so.”

WebAssembly | MDN

“WASI (WebAssembly System Interface) is a set of interfaces that provide a standard way for WebAssembly modules to interact with the host operating system and other external resources.

It allows WebAssembly programs to access system calls, file system, and other features of the host environment, making it possible to run WebAssembly programs outside the browser environment.

This allows for more versatile usage of WebAssembly, such as in servers, command-line tools, and other non-browser contexts.”

ChatGPT

To build Web Assembly with Rust you will need one of these tools:

- **wasm-pack** is command-line tool for building and packaging Rust-generated WebAssembly (wasm) modules
- **Trunk** is a WASM web application bundler for Rust. Trunk uses a simple, optional-config pattern for building & bundling WASM, JS snippets & other assets (images, css, scss) via a source HTML file.

Web Assembly - wasm-bindgen

```rust
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console)]
    fn log(s: &str);
}

#[wasm_bindgen(start)]
pub fn run() {
    log("Hello, World!");
}
```

- **js-sys** Bindings to JavaScript's standard, built-in objects, including their methods and properties.
  - Javascript Types (**Array**, **Boolean**, **JsString**, …)
  - Modules (**JSON**, **Math**, **Intl**, …)
  - Errors
  - …
- **web-sys** Raw API bindings for Web APIs that browsers provide on the web.
  - Document
  - Element
  - Event
  - EventTarget
  - HtmlInputElement
  - …

Web Assembly - web-sys Features

```toml
[dependencies.web-sys]
version = "0.3.60"
features = [
   "HtmlInputElement",
   "HtmlAudioElement",
   # etc..
]
```

Gloo is a modular toolkit for building fast and reliable libraries and apps with Rust and WebAssembly.

```rust
use web_sys::console::{log_1, log_2};

let object = JsValue::from("any JsValue can be logged");
log_1(&JsValue::from("hello"));
log_2(&JsValue::from("text"), &object);
```

```rust
use gloo_console::log;

let object = JsValue::from("any JsValue can be logged");
log!("text", object);
```

- gloo_console
- gloo_dialogs
- gloo_events
- gloo_file
- gloo_history
- gloo_net
- gloo_render
- gloo_storage
- gloo_timers
- gloo_utils
- gloo_worker

Yew Framework

- Rust Web framework via WASM
- Inspired by React

```rust
use yew::prelude::*;

#[function_component(App)]
pub fn app() -> Html {
    html! {
        <main>
            <img src="https://yew.rs/img/logo.png" alt="Yew logo" />
            <h1>{ "Hello world!" }</h1>
            <span>{ "from Yew with " }<i class="heart" /></span>
        </main>
    }
}
```

Yew Framework - **html!** macro

```
html! {
    <div>
        <h1>{"Hello"}</h1>
        <div>
    </div>
}
```

```
error: this opening tag has no corresponding closing tag
 --> src/app.rs:6:9
  |
6 |         <div>
  |         ^^^^^
```

Yew Framework - **html!** macro - **Variable**

```rust
let hello = String::from("hello");
html! {
    <h1>{hello}</h1>
}
```

html! takes ownership of the variable

```rust
let hello = String::from("hello");
html! {
    <h1>{&hello}</h1>
}
```

html! borrows the variable

```
let can_display = true;
html! {
    if can_display {
        <h1>{"hello"}</h1>
    } else {
        <h1>{"-"}</h1>
    }
}
```

Yew Framework - **html! Macro - List**

```rust
use yew::prelude::*;

#[function_component(App)]
pub fn app() -> Html {
    let items = (0..10).map(|value| {
        html! {
            <li>{value}</li>
        }
    });

    html! {
        <ul class="item-list">
            { items.collect::<Html>() }
        </ul>
    }
}
```

```rust
use yew::prelude::*;

#[function_component(App)]
pub fn app() -> Html {
    let items = (0..10).map(|value| {
        html! {
            <li>{value}</li>
        }
    });

    html! {
        <ul class="item-list">
            { for items }
        </ul>
    }
}
```

```rust
let can_hide = true;
html! {
    <div hidden={can_hide}>
        { "This div is hidden." }
    </div>
}
```

- Boolean attribute will be added only if the boolean is **true**
- Applies as well to other attributes:
  - **hidden**
  - **checked**
  - **required**

Yew Framework - **classes!** macro

```
html! {
    <h1 class="title">{"Hello!"}</h1>
}
```

```
html! {
    <h1 class={classes!("title", "bright")}>{"Hello!"}</h1>
}
```

```
html! {
    <h1 class={classes!("title", Some("bright"))}>{"Hello!"}</h1>
}
```

```
html! {
    <h1 class={classes!(vec!["title", "bright"])}>{"Hello!"}</h1>
}
```

Yew Framework - **html!** macro

```
html! {
    <>
        <h1>{"Hello!"}</h1>
        <p>{"This is a paragraph"}</p>
    </>
}
```

Yew Framework - Components

```rust
use yew::prelude::*;

#[function_component(Title)]
pub fn title() -> Html {
    html! {
        <h1>{"Hello"}</h1>
    }
}

#[function_component(App)]
pub fn app() -> Html {
    html! {
        <>
            <Title />
            <p>{"This is a paragraph"}</p>
        </>
    }
}
```

Yew Framework - Components properties

```rust
use yew::prelude::*;

#[derive(Properties, PartialEq)]
pub struct TitleProps {
    label: String,
}

#[function_component(Title)]
pub fn title(props: &TitleProps) -> Html {
    html! {
        <h1>{&props.label}</h1>
    }
}
```

```rust
#[function_component(App)]
pub fn app() -> Html {
    html! {
        <>
            <Title label={String::from("Hello")} />
            <p>{"This is a paragraph"}</p>
        </>
    }
}
```

Open the **4_web-assembly** folder

- Run the command: `trunk serve`
- Open the page: `http://127.0.0.1:8080`

Extract Components (20 minutes)

- TodoList
- NewTodoInput

Yew Framework - Hooks use_state

```rust
use web_sys::HtmlInputElement;
use yew::prelude::*;

#[function_component(App)]
pub fn app() -> Html {
    let name = use_state(|| String::new());

    html! {
        <>
            <h1>{"Please enter your name"}</h1>
            <p>{format!("Hello {}", *name)}</p>
        </>
    }
}
```

Yew Framework - Hooks use_state

```rust
let name = use_state(|| String::new());
let onkeyup = {
    let name = name.clone();
    Callback::from(move |event: KeyboardEvent| {
        let input = event.target_unchecked_into::<HtmlInputElement>();
        if event.key() == "Enter" {
            name.set(input.value());
            input.set_value("");
        }
    })
};
```

Create static list of todos content from a state (10 mins)

```rust
use yew::prelude::*;
use gloo_dialogs::alert;

#[function_component(App)]
pub fn app() -> Html {
    let notify_btn_clicked = Callback::from(|_| {
        alert("Button clicked");
    });
    html! {
        <button onclick={notify_btn_clicked}>
            {"Click here!"}
        </button>
    }
}
```

Full list of events is available on https://yew.rs/docs/concepts/html/events#available-events

Yew Framework - Events

```rust
use yew::prelude::*;
use gloo_console::log;

#[function_component(App)]
pub fn app() -> Html {
    let log_keypress = Callback::from(|event: KeyboardEvent| {
        log!(format!("{} pressed", event.key()));
    });
    html! {
        <input onkeypress={log_keypress} />
    }
}
```

Yew Framework - Events target *with JsCast*

```rust
use gloo_console::log;
use wasm_bindgen::JsCast;
use web_sys::{EventTarget, HtmlInputElement};
use yew::prelude::*;

#[function_component(App)]
pub fn app() -> Html {
    let log_keypress = Callback::from(|event: KeyboardEvent| {
        let target: Option<EventTarget> = event.target();

        let input = target.and_then(|t| t.dyn_into::<HtmlInputElement>().ok());

        if let Some(input_elt) = input {
            log!(format!("text: {}{}", input_elt.value(), event.key()));
        }
    });
    html! {
        <input onkeypress={log_keypress} />
    }
}
```

Yew Framework - Events target *with TargetCast*

```rust
use gloo_console::log;
use web_sys::HtmlInputElement;
use yew::prelude::*;

#[function_component(App)]
pub fn app() -> Html {
    let log_keypress = Callback::from(|event: KeyboardEvent| {
        let input = event.target_dyn_into::<HtmlInputElement>();

        if let Some(input) = input {
            log!(format!("text: {}{}", input.value(), event.key()));
        }
    });
    html! {
        <input onkeypress={log_keypress} />
    }
}
```

Prelude is importing **yew::TargetCast**
Trait that simplifies casting targets.

Yew Framework - Events target *with TargetCast*

```rust
use gloo_console::log;
use web_sys::HtmlInputElement;
use yew::prelude::*;

#[function_component(App)]
pub fn app() -> Html {
    let log_keypress = Callback::from(|event: KeyboardEvent| {
        let input = event.target_unchecked_into::<HtmlInputElement>();
        log!(format!("text: {}{}", input.value(), event.key()));
    });
    html! {
        <input onkeypress={log_keypress} />
    }
}
```

Yew Framework - Components properties

```rust
#[function_component(App)]
pub fn app() -> Html {
    let show_answer = Callback::from(|answer: String| {
        alert(&answer);
    });

    html! {
        <>
            <h1>{"Are you learning?"}</h1>
            <YesOrNoButton on_answer_clicked={show_answer} />
        </>
    }
}
```

Yew Framework - Components properties

```rust
#[derive(Properties, PartialEq)]
pub struct TitleProps {
    on_answer_clicked: Callback<String>,
}

#[function_component(YesOrNoButton)]
pub fn yes_or_no_btn(props: &TitleProps) -> Html {
    let emit_answer = {
        let on_answer_clicked = props.on_answer_clicked.clone();
        Callback::from(move |event: MouseEvent| {
            let button = event.target_unchecked_into::<HtmlButtonElement>();
            let answer = button.inner_text();
            on_answer_clicked.emit(answer);
        })
    };

    html! {
        <>
            <button onclick={emit_answer.clone()}>{"Yes"}</button>
            <button onclick={emit_answer.clone()}>{"No"}</button>
        </>
    }
}
```

Implementing TodoMVC (20 minutes)

- Add a Todo when pressing Enter key if input is not empty
- Delete a Todo from the Todo list
- Define a Todo as selected when clicking the select box
- Saving the Todo list Local Storage with gloo
- etc...

https://github.com/yewstack/yew/tree/master/examples/function_todomvc

Thank you

If you have any questions, please get in touch.

Jocelyn Facchini
Software Craftsman

jocelyn.facchini@codurance.com
Jocelyn.facchini
@jsfacchini
@jsfacchini@hachyderm.io