Exercise https://github.com/codurance/rust-nation-intermediate-workshop/tree/main/1_generics-traits

```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);
    }
}
```

Defining a trait

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

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

    largest
}
```

Creating a generic function

Traits & Generics

```rust
fn debug_value<T: Display + Debug>(value: T)
{
   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(value: impl 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 bounds definitions

Traits & Generics

```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")));
}
```

Trait bounds definition on other types

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

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

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

Generic return type with opaque type

Traits & Generics

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

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

Generic types in struct and enum

```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)
    }
}
```

Blanket implementation

Traits & Generics

```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)))
        }
    }
}
```

```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))
        }
    }
}
```

Implementing a trait with associated type

```rust
trait HasAssociatedType {
    type SomeType;
}

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

    //...
}
```

Using associated type from a trait bound

```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),
    ]);
}
```

Heterogeneous list of displayable items

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);
```

Using closures in iterators

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

Captures borrowed values from iterator

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

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

Forcing closures ownership

Closures - Closure Traits

```rust
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());
}
```

Using closure as function parameter with generics

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

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

Using closure as function parameter with trait objects

Closures

```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,
```

Defining a closure trait

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

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

Calling a closure stored in a struct

Threads

Exercise https://github.com/codurance/rust-nation-intermediate-workshop/tree/main/2_threads

```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);
    }
}
```

Spawning a list of threads

```rust
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);
```

Sharing immutable values with threads

Threads

```
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());
```

Sharing mutable values with threads

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

Defining trait bounds required by a thread

Threads

```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);
}
```

Using channel to communicate between threads

```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);
    }));
}
// ...
```

Sharing receiver of a channel between threads

Declarative Macros

Exercise https://github.com/codurance/rust-nation-intermediate-workshop/tree/main/3_macros

```
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);
}
```

Defining a macro matching expressions with “; and” separator

```
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" };
}
```

Using a repetition in a macro matcher

Declarative Macros

```
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));
}
```

Using fully qualified names for external dependencies

```
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);
}
```

Defining multiple matchers

Declarative Macros

```
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
}
```

Defining multiple modules for a give type and module identifier

Yew Framework - **html!** macro

```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>
    }
}
```

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

Exercise https://github.com/codurance/rust-nation-intermediate-workshop/tree/main/4_web-assembly

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>
    }
}
```

Yew Framework - **classes!** macro

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

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

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

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

Yew Framework - Components

```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>
      </>
  }
}
```

Yew Framework - Hooks use_state

```
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>
      </>
  }
}
```

```
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("");
       }
   })
};
```

Yew Framework - Events

```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>
  }
}
```

```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} />
  }
}
```

Yew Framework - Events target *with TargetCast*

```rust
use gloo::console::log;
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} />
        </>
    }
}
```

```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>
        </>
    }
}
```