Midterm 2 - Mockup

Ex 1

- define an i32 constant named "CONSTANT" inside a module named "odd_module" and assign to it the value 123
- define an i32 constant named "CONSTANT" inside a module named "even_module" and assign to it the value 246
- define a public function "get_constant" inside the module "getter_function" that take as input an u32 named "value", and return
- the constant inside "odd_module" if "value" is odd. otherwise it returns the constant inside "even_module"
- just to avoid confusion remember that in Italian: odd = dispari, even = pari

Ex 2

define a trait CloneAndDouble with a function `clone_and_double(&self)->Self`
the function clone_and_double clone the item and double it.
Implement the trait for all items that implement the traits Clone and Add (use a simple addition to double)

Ex 3

The trait `Unknown` defines a method `serialize` that returns the implementer's `String` representation.

- implement it for `i32`
- implement it for `String`
- implement it for `Vec<T>`, where T implements Debug
- write a function `get_vec` that returns an empty vec of `Unknown` data
- write a function `print_vec` that takes as input a reference of a vec of `Unknown` data and prints its content

```
trait Unknown {
    fn serialize(&self) -> String;
}
```

Ex 4

Write a struct `BinIter` that implements `Iterator` over `bool`s.

- `BinIter` must have a function `new` that takes as input `n` the number and `l` the length.
- The iterator must yield bits according to the binary form of `n`, after returning the `l`-th bit the iterator stops.
- The bits yielded must be in "little-endian" order, so the most significant bit must be yielded last.

Ex 5

Implement a doubly linked list
Create the necessary structs to represent it

- `Node<T>` with an element of type `T` and two fields, `prev` and `next`, both of type `Option<Rc<RefCell<Node<T>>>>`.
- `List<T>` with two fields, `head` and `tail`, both of type `Option<Rc<RefCell<Node<T>>>>`, and a size field of type `usize`.

Implement the following traits for `Node<T>`:

- `PartialEq` that compares the elements of two nodes.

- `Display` that prints the element of a node.

Implement the following traits for `List<T>`:

- `PartialEq` that checks if two lists are equal, by comparing the elements of the nodes, one by one.
- `Debug` that prints the elements of the list.

Implement the following methods for `List<T>`:

- `new()` that creates a new empty list.
- `print_list(&self)` that prints the elements of the list.
- `push(&mut self, element: T)` that adds an element to the front of the list.
- `pop(&mut self) -> Option<T>` that removes an element from the front of the list.
- `push_back(&mut self, element: T)` that adds an element to the back of the list.
- `pop_back(&mut self) -> Option<T>` that removes an element from the back of the list.
- `print_list(&self)` that prints the elements of the list.

Ex 6

Write the necessary structs to represent an oriented graph generic over `T`, where `T` implements `Hash`, `PartialEq` and `Eq`.

- `Node`, with a value of type `T` and a vector of adjacent nodes
- `Graph`, with a vector of nodes

Then, implement the following methods for `Node`:

- `new`, which creates a new `Node` with the given value and the given vector of adjacents
- `get_value`, which returns a reference to the value of the node

Implement `Debug` for `Node`, so that it prints the value of the node and the values of its adjacents.
For example, if the node has value `1` and its adjacents are `2` and `3`, it should print:

> [value: 1, adjacents: [2, 3]]

Then, implement the following methods for `Graph`:

- `new`, which creates a `Graph` from a vector of nodes, with the respective adjacents set
- `dfs`, which performs a depth-first search on the graph, starting from the given node. It returns a vector of nodes, in the order in which they were visited.

Ex 7

Write a trait `Task` that define a method `execute(&self)->usize`.

implement the `Task` trait for the following structs:

- `SumTask` is a struct with a method `new(n1: usize, n2: usize)` were executing task returns the sum of n1 and n2
- LenTask is a struct with a method `new(s: String)` were executing task returns the len of s

Write two structs: `Tasker` and `Executer`, that interact following this protocol:

- At any given time any number of tasker and executer can be linked together.
- `Tasker` can ask for a task to be scheduled using the method `schedule_task(&mut self, task: ...)` that take as input a

box with inside an object that implements Task

- `Executer` can execute a task using the method `execute_task(&mut self)->Option<usize>`. this method can fail if no task is scheduled
- Tasks are executed inf a FIFO queue
- `Tasker` has a method `new` that return am instance with an empty queue, linked to no one.
- `Tasker` has a method `get_tasker(&self)->Tasker` that return a new `Tasker` linked with self.
- `Tasker` has a method `get_executer(&self)->Executer` that return a new `Executer` linked with self.