Week 05 Weekly Exercises

Objectives

- Use and understand the strength of Traits for defining common behaviours
- Investigate the smart pointers Cell, and RefCell
- Introduction to generic programming, trait bounds, and trait objects

---

Activities To Be Completed

The following is a list of all the markedactivities available to complete this week...

- **Pointy**
- **Dungeons and Dragons!**
- **Languages!**

The following practice activities are optional and are not **marked, or required** to be completed for the week.

- **Mitochondria (Investigating the Cell smart pointer)**

Preparation

Before attempting the weekly exercises you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this week's exercises called `lab05`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab05
$ cd lab05
$ 6991 fetch lab 05
```

Or, if you're not working on CSE, you can download the provided code as a tar file.

EXERCISE:
Pointy

NOTE:

Objectives

- Understand how and why Rust uses generics
- Write a generic DataType, Function, and understand the problems which they solve
- Implement a method on a generic DataType

Basically, get you to write some very basic generic code in Rust!

So far in our Rust journey, we've always had to write our types, functions, and methods for a specific type.

For example, we would write a struct that would store two $i32$ values, and then we would write a function that would take two $i32$ values and return an $i32$ value (maybe doing some sort of addition).

This is fine for simple programs, but what if we wanted to write a function that would take two $f32$ values, perform some operation, and then return an $f32$ value? We would have to write a new function for that. What if we wanted to write a function that would take two String values, perform the same operation, and then return a String value? We would have to write a new function for that, even though the only thing that changed were the types inputted and outputted to the function!

Suddenly, we have multiple functions that all do the same thing - resulting in a lot of duplicated code, more code to maintain, test and write! There are many ways we could make writing this code easier; we can use generics to solve this problem!

> HINT:
>
> If you've never encountered "generic programming" or "generics" before, you may find the following resources useful:
> - What is Generic Programming?
> - Rust Book - Generics

In this exercise you will be writing a generic function, a generic type, and a method on a generic type. You should complete the below tasks in `src/lib.rs`

- **First** - implement a generic function called `first` that takes a slice of some type T, and returns a shared borrow of first value in the slice. Be sure to use the provided doctests to help you write your function.
- **Second** - modify the given `Point` struct to be generic. It should be able to store two values of one type `T`. Be sure to use the provided doctests to help you write your struct.
- **Third** - modify the existing method on the `Point` struct called `distance` that takes a second `Point` and returns the distance between the two points. Modify this method to be only be implemented for `Point` instances where the type of the `x` and `y` values are `f32`
- **Finally** - create a generic method on the `Point` struct called `new` that creates a new `Point`. This method should take two values of type T and return a new `Point` instance with the given values.

  Ideally, your method should return `Self`, and construct a `Self`.

> HINT:
>
> The documentation for the `traits` chapter of the book may be useful.
>
> The documentation for the `generics` chapter of the book may be useful.
>
> The documentation for the `methods` chapter of the book may be useful.
>
> The documentation for the `slices` chapter of the book may be useful.
>
> The documentation for the `self` keyword may be useful.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 7 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:

Dungeons and Dragons!

> NOTE:
>
> **Objectives** Understanding, practice and reasoning about:
> - Traits
> - Trait bounds
>
> Additionally, exposure to the rustyline crate.
>
> e should provide you with the when and why behind writing your first trait in Rust.

> It should be an opportunity for you to create, reason about and apply simple traits - useful for your assignment!

In this exercise, you will be harnessing the power of traits, in order to emulate a subset of a role playing game (RPG) inspired by dungeons and dragons - dice rolling!

The provided code will handle all of the input parsing using the awesome rustyline crate. You will be modifying the code in `die.rs`

> WARNING:
>
> If you have not watched the week 5 lectures, the following short talk should give enough context to start.
> It's also in theme!
>
> Traits and You: A Deep Dive — Nell Shamrell-Harrington

In this exercise, you will need to complete the following tasks, by modifying the code in `die.rs` ONLY.

1. Create a Trait with a single function , that describes the behaviour of getting the value of a roll of a dice/coin.
2. Implement the trait for `Coin` (a coin "roll" will either be 1 or 2).
3. Implement the trait for `Dice` (each dice will be a number between 1 and the number of sides on the dice (inclusive) - a D4 for example, has 4 sides).
4. Add a generic trait bound for the roll function - such that your function accepts any type T - where T implements your trait!

> NOTE:
>
> In order to pass the autotests, you **must** use the given `get_random_value` function to generate random values (why might this be?).
> You should not touch the code inside `main.rs`

```
$ 6991 cargo run
  Finished dev [unoptimized + debuginfo] target(s) in 0.02s
    Running `target/debug/dungeons_and_dragons`
>> d20
You rolled a d20: 7
>> d2
You rolled a d2: 1
>> d10
You rolled a d10: 4
>> d8
You rolled a d8: 3
>> Ctrl-D
Goodbye!
```

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 7 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:

Languages!

> NOTE:
>
> **Objectives**
>
> - Trait objects
> - The From trait
> - ction with associate trait types `impl Add<Foo> for Bar`

> Understanding of the above is super useful for the assignment! For example, you might want something like `impl Add<Direction> for Player`

In this exercise, you will be finishing a quick implementation of a program that uses some basic operations using trait objects.

You've been given a starter crate languages, which consists of the `Greeting` trait, and three implementations of it for the `English`, `French` and `Spanish` structs.

Your task is to complete the following tasks in `main.rs` such that it compiles and prints some basic output.

1. **First**, implement the `From` trait, such that you can convert a `&str` into a `Box<dyn Greeting>`. You can assume that only valid strings will be given. In a real codebase, you would want to handle errors (maybe by using the `TryFrom` trait), but for this exercise, you can assume that the input is valid.
2. **Second**, finish the implementation of the `speak_all_greetings` function, such that it prints out the greetings.

> HINT:
>
> The video linked in the exercise above should explain the concept of trait objects, alongside the week05 lectures.

```
    $ 6991 cargo run
    Compiling languages v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 0.21s
     Running `target/debug/languages`
John says:
Hello!
Hola!
Jane says:
Bonjour!
```

> HINT:
>
> You may find the following videos useful, although some may go into advanced trait usages.
> - Jon Gjengset - Dispatch and Fat Pointers
> - Tim Clicks - Generics and trait objects explained

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 7 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

(OPTIONAL) CHALLENGE EXERCISE:

Mitochondria (Investigating the Cell smart pointer)

> DANGER:
>
> This exercise is optional, and will not be marked. This will allow you to focus on your assignment1!
> If you're getting here and have time, you can still complete it, and discuss your solutions on the forum/with your tutor.
>
> However, we would recommend you instead spend your time watching some of the linked videos to get a better understanding of trait objects, if you have the time.
>
> If you've finished this and still want something to procrastinate your assignment on - I'd love to see an investigation of the thiserror and anyhow crates for error handling as blog post :D

> NOTE:

> In the field of biology, the mitochondria is often labelled as the "powerhouse of the cell", that is, it is the organelle that is responsible for how/why the Cell is able to work!
> This exercise name hence is `mitochondria`, as we investigate how rust's std::cell::Cell smart pointer (and it's close relatives) work, and what powers them!

This exercise is a theory exercise, aimed to explore the smart pointers `Cell` and `RefCell`. The exercise consists of 10 prompts/questions, in order for you to investigate `Cell`. Your task is to answer each question!

If you wish to discuss your solution/ask questions - do so on the course forum!

> HINT:
>
> Most of the questions in this exercise can be answered by watching the first 20 or 30 minutes of this video:
> Crust of Rust: Smart Pointers and Interior Mutability

Submission

When you are finished each exercise make sure you submit your work by running `give`.

You can run `give` multiple times.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

The due date for this week's exercises is **Week 7 Wednesday 21:00:00**.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

Automarking will be run continuously throughout the term, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run you can view your results here or by running this command on a CSE machine:

```
$ 6991 classrun -sturec
```
