Exercise 1

Its time to put all this Rust nonsense to good use. For our first exercise, we are going to build a little LevelSpec api. As a refresher, The LevelSpec should model a level in a job system. To begin with, said level can be one of three things - a show, a Sequence, or a Shot. Because we are talking about a closed set of variants, the best way of modeling this is with an enum. So lets get started...

First steps

Create a new library project, and name it levelspec. You remember how to do that, right? In case I haven't mentioned it before, to create a library, you use the --lib flag with... that command I have already demonstrated.

Your project should look something like this:

```
levelspec/
    src/
       lib.rs
    Cargo.toml
```

In src, we will be working in the lib.rs file. The file will have a boilerplate test in it. Go ahead do your work above the boilerplate. We will use it later.

The LevelSpec

Go ahead and **create a LevelSpec enum**. It should have 3 variants - Show, Sequence, and Shot. You are free to use tuplestruct notation or struct notation. The choice is up to you. As a reminder, here are what each type looks like:

```
pub enum Foo {
  TupleStruct(String, String),
  Other{first: String, last: String}
}
```

So now we have a POD. Lets add some constructor functions for practice. Specifically, **add an impl** block and these **three functions**:

1. fn from_show(show: String) -> Self
2. fn from sequence(show: String, sequence: String) -> Self
3. fn from_shot(show: String, sequence: String, shot: String) -> Self

I wont lie to you. It would be nice if Rust supported function overloading. There is nothing philosophical stopping them from adding it. It is just not high on their list.

Ok. Now that you have added a minimal api of sorts, lets write some tests.

Testing

If you followed the first step correctly, when you opened the lib.rs file, you probably saw something like this:

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Hopefully, you added your code above this, as opposed to removing it. Otherwise, you have some typing to do. Either way, an explanation is in order. The `#[cfg(test)]` is a pragma for the compiler that sets up conditional compilation for the tests module, which is declared on the line below. If we didn't have this pragma, all of the tests within the tests module would be baked into the library, which we don't necessarily want. There is nothing special about the module name by the way. We could call it levelspectests or whatever. Doesn't matter. What does matter is that this module doesn't inherit anything from its parent's scope. So, right under `mod tests` type the following:

```
use super::*;
```

Now we can see LevelSpec from the parent scope.

Every test we write will be a function, with a `#[test]` on top of it, letting rust know that it is in fact a test. Without that pragma, the test will not be found by the test runner. To run a test, type:

```
cargo test
```

This will compile the code and execute all of the tests in parallel (by default). That last bit is important if you intend to write tests which interact with the environment in any way. We wont have to worry about that now though.

Ok. lets replace that first test with something meaningful.

```
fn can_create_show() {
  let levelspec = LevelSpec::from_show("DEV01");
  let expected = <YOUR CODE>; // initialize directly depending upon what
LevelSpec looks like (ie ```LevelSpec::Show(String) vs
LevelSpec::Show{name:String}``)

  assert_eq!(levelspec, expected);
}
```

Tests

Rust provides a number of macros that are useful for testing. Among them are:

macro	description
assert!	Test that some expression evaluates to true
assert_eq!	Test that two expressions are equal
assert_ne!	Test that two expressions are not equal

If you find the errors less than helpful when you run `cargo test`, the three macros outlined above each take optional parameters that get passed to `format!`; the resultant string is presented to the user in the case of a test failure. For example:

```
let result = greeting("foo");
assert!(result.contains("foo"), "result didn't contain foo. value was
`{:?}`", result);
```

Ok, with that explanation out of the way, lets create two more tests, one for sequences, and one for shots. For each test, you will have to create a function adorned with `#[test]`. Lets follow the first test and call them "can_create_sequence" and "can_create_shot". Populate them with relevant tests.

After getting your tests to pass, its time to move on to a bit more interesting challenge.

LevelSpec from_str

We would like to be able to create a levelspec from a string. Of course, not all strings are valid, so we will want to create a fallible function to handle this. Fortunately, Rust has a trait that fits the bill. It looks like this:

```
trait FromStr {
  type Err;
  fn from_str(s: &str) -> Result<Self, self::Err>;
}
```

Right off the bat, we realize that we are going to need to pick a strategy for dealing with errors. To begin with, we are simply going to return a String wrapped in an Error. Later on, we will refactor to use a custom error. In case we have not covered it, the *type* above is called an *associated type*. It is used in lieu of a generic when one has to choose a single type for a trait implementation. In our case, we will be setting `type Err=String;`.

Anyway, back to the task at hand. Implement the FromStr trait for LevelSpec. Just as a reminder, a levelspec string may have between 0 and two periods separating level components. `SHOW[.SEQUENCE[.SHOT]]`. Legal characters include upper and lower case letters and underscores. Certainly no spaces. You are welcome to look into external crates to help you validate, although you can also apply more brute force methodologies. Be sure to look into both the String and str documentation to see what sort of help is available.