Dive into async and Futures in Rust

1Password

What is all this?

- Async is an abstraction provided by Rust to express **concurrent units of work**.
- A `Future` is a single unit of work that can operate concurrently with other `Futures`.
- `async`/`await` are primitives for working with `Futures` in an imperative style.
- Usually used for i/o, but not limited to i/o.
- We're focusing on the underlying abstraction, not tokio or other runtimes.

The `Future` trait

- ```
  pub trait Future {
      type Output;

      fn poll(
          self: Pin<&mut Self>,
          cx: &mut Context<'_>,
      ) -> Poll<Self::Output>;
  }
  ```
  ```

The `Future` trait

- ```
  pub trait Future {
      type Output;

      fn poll(
          self: Pin<&mut Self>,
          cx: &mut Context<'_>,
      ) -> Poll<Self::Output>;
  }
  ```
- Output type returned by the future when its work is complete
  ```

The `poll` method

- ```
  pub trait Future {
      type Output;

      fn poll(
          self: Pin<&mut Self>,
          cx: &mut Context<'_>,
      ) -> Poll<Self::Output>;
  }
  ```
- The `poll` method is where the future does its work.
  ```

Pinned references

```
pub trait Future {
    type Output;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output>;
}
```

- Futures take a **pinned mutable reference** to `self`
- A `Pin` is a special kind of reference that promises that the referenced data will never be moved. This promise persists even after the `Pin` is dropped.
- Difficult to use safely, but libraries help.

The `Poll` result

- ```
  pub trait Future {
      type Output;

      fn poll(
          self: Pin<&mut Self>,
          cx: &mut Context<'_>,
      ) -> Poll<Self::Output>;
  }
  ```
- ```
  pub enum Poll<T> {
      Ready(T),
      Pending,
  }
  ```
- `Pending` means that the `Future` will need to be polled again.

Context and Waker

- ```
  pub trait Future {
      type Output;

      fn poll(
          self: Pin<&mut Self>,
          cx: &mut Context<'_>,
      ) -> Poll<Self::Output>;
  }
  ```
- Context contains a Waker.
- ```
  impl Waker {
      fn wake(self);
      fn wake_by_ref(&self);
  }
  ```

The contract of `Future`

- When polled, the `Future` attempts to make progress.
  - `Futures` do their work in the foreground, in the `poll` function.
- When it gets to a waiting point, it returns `Poll::Pending`.
- It arranges independently for `Waker::wake` to be called when it's ready to continue.
- The caller in turn promises that when wake is called, the `Future` will be polled again.
  - The `Future` may be polled spuriously.
  - The `Future` may be dropped or not polled.
- Key point: A Future is inert.

Polling? Wakers? What about `async/await`?

- An `async` block is an anonymously typed object that implements `Future`.
- As `async` function *returns* an anonymously typed object that implements `Future`.
- Any `Future` can be awaited with `.await`.
- These `Futures` have polling logic that resumes the most recent `await`.