PRESIDENTIAL INITIATIVE
ARTIFICIAL INTELLIGENCE AND COMPUTING

Async

An Introduction

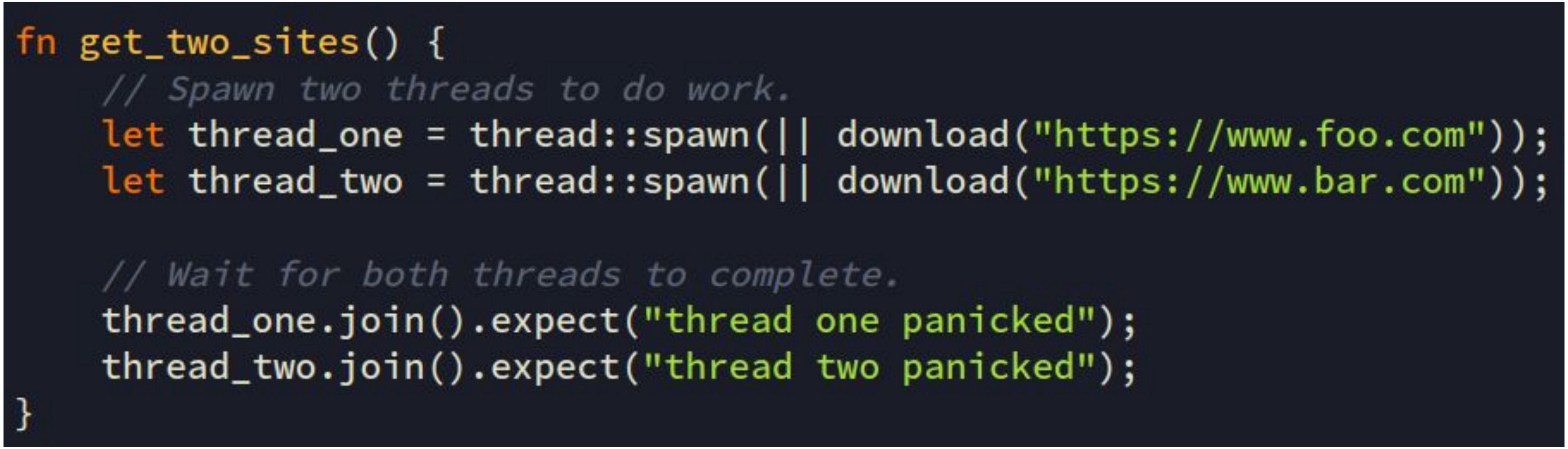
Introduction to Async Programming

- Async (or asynchronous) programming is a technique to run simultaneous operations in **single thread**
- Whatever the nature of your application a web server, a database or an operating system, using async programing you can get most out of the underlying hardware

Why Async Rust

- Asynchronous Rust allows us to run multiple tasks concurrently on the same OS thread
- In a typical threaded application, if you wanted to download two different webpages at the same time, you would spread the work across two different threads
- Example on the following screen

Why Async Rust



```
fn get_two_sites() {
    // Spawn two threads to do work.
    let thread_one = thread::spawn(|| download("https://www.foo.com"));
    let thread_two = thread::spawn(|| download("https://www.bar.com"));

    // Wait for both threads to complete.
    thread_one.join().expect("thread one panicked");
    thread_two.join().expect("thread two panicked");
}
```

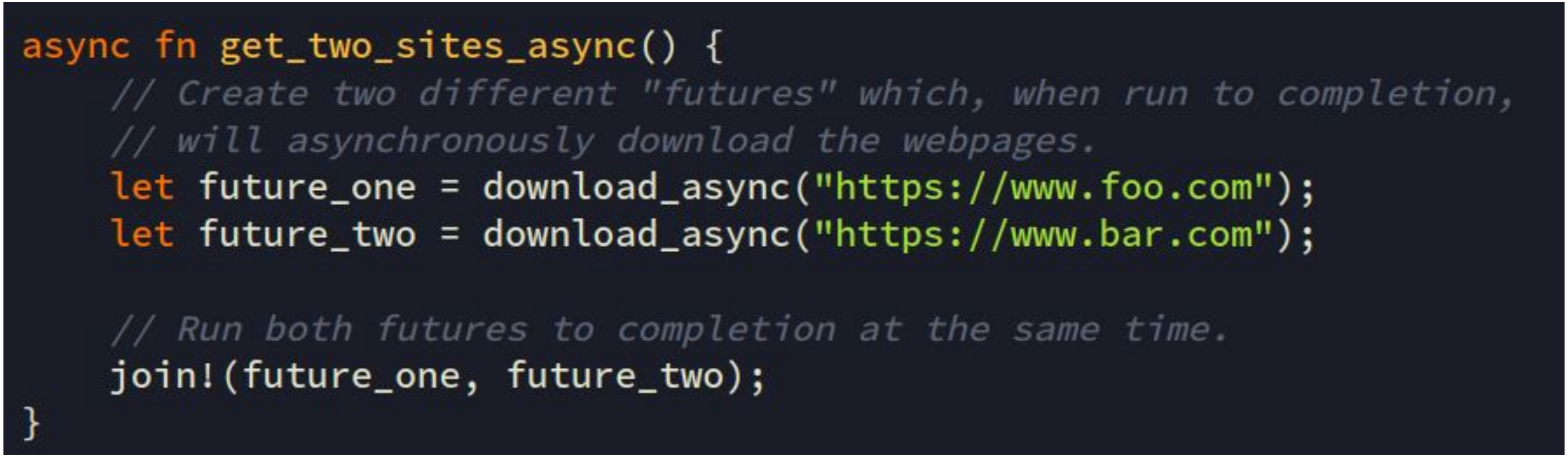
Why Async Rust

- The code on previous slide works fine for many applications
- Since threads were designed to run multiple different tasks at once
- However, they also come with some limitations
- There's a lot of overhead in switching between different threads and sharing data between them
- Even a thread which just sits and does nothing uses up valuable system resources

Why Async Rust

- These are the costs that asynchronous Rust is designed to eliminate
- We can rewrite the function above using Rust's async/.await notation
- Which will allow us to run multiple tasks at once without creating multiple threads

Why Async Rust



```rust
async fn get_two_sites_async() {
    // Create two different "futures" which, when run to completion,
    // will asynchronously download the webpages.
    let future_one = download_async("https://www.foo.com");
    let future_two = download_async("https://www.bar.com");

    // Run both futures to completion at the same time.
    join!(future_one, future_two);
}
```

Why Async Rust

- Overall, asynchronous applications have the potential to be **much faster and use fewer resources** than a corresponding threaded implementation
- However, **there is a cost**
- Threads are **natively supported by the OS** (operating system), and using them doesn't require any special programming model

Why Async Rust

- Any function can create a thread
- A function that uses threads is usually just as easy as calling any normal function
- However, **asynchronous functions** require special support from the language or libraries

Why Async Rust

- In Rust, **async fn** creates an asynchronous function which returns a Future
- To execute the body of the function, the returned Future must be run to completion

Why Async Rust

- It's important to remember that traditional threaded applications can be quite effective
- The increased complexity of the asynchronous programming model isn't always worth it
- it's important to consider whether your application would be better served by using a simpler threaded model

The State of Asynchronous Rust

The State of Asynchronous Rust

- Asynchronous Rust ecosystem has undergone a lot of evolution over time
- So it can be hard to know what tools to use, what libraries to invest in, or what documentation to read
- However, the **Future trait** inside the standard library and the **async**/**await** language feature has recently been stabilized.

The State of Asynchronous Rust

- The ecosystem as a whole is therefore in the midst of migrating to the newly-stabilized API
- After which point churn will be significantly reduced
- At the moment, however, the ecosystem is still undergoing rapid development and the asynchronous Rust experience is unpolished
- Most libraries use the **0.1** definitions of the **futures crate**, meaning that to interoperate developers frequently need to reach for the compat functionality from the **0.3 futures crate**

The State of Asynchronous Rust

- The **async/await** language feature is still new
- Important extensions like **async fn syntax** in trait methods are still **unimplemented**
- Current compiler error messages can be difficult to parse
- In short, Rust is well on its way to having some of the most performant and ergonomic support for **asynchronous programming**

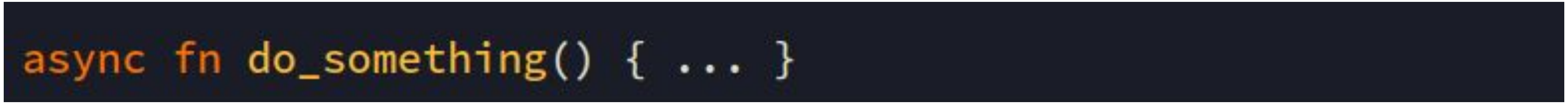
Async/.await Primer

Async/.await Primer

- Async/.await is **Rust's built-in tool** for writing asynchronous functions that look like synchronous code
- Async transforms a block of code into a state machine that implements a trait called Future
- Whereas calling a **blocking function** in a synchronous method would block the whole thread
- **Blocked Futures** will yield control of the thread, allowing other Futures to run

Async/.await Primer

- To **create** an **asynchronous function**, you can use the **async fn** syntax:

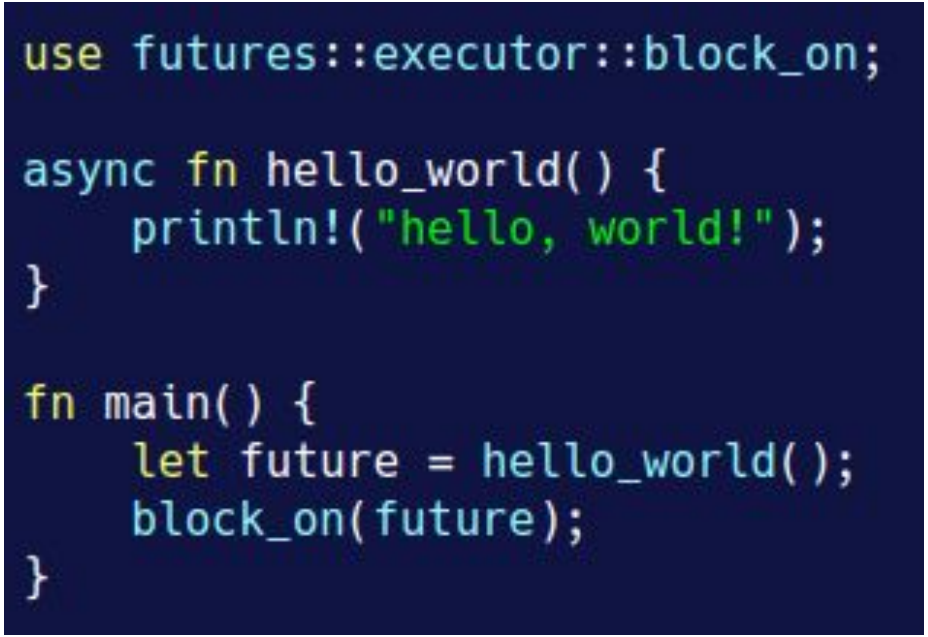


```
async fn do_something() { ... }
```

- The **value** returned by async fn is a **Future**
- **Future** needs to be run **on an executor**, so that a task may be done

Async/.await Primer

- “**block_on**” blocks the current thread until the provided future has run to completion
- Other executors provide more complex behavior, like scheduling multiple futures onto the same thread



```
use futures::executor::block_on;

async fn hello_world() {
    println!("hello, world!");
}

fn main() {
    let future = hello_world();
    block_on(future);
}
```

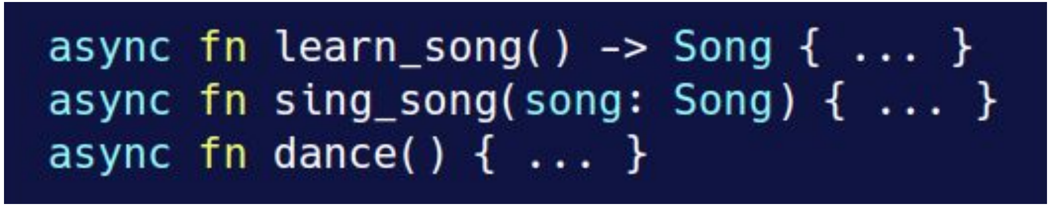
Async/.await Primer

- We can also use **.await** instead of **block_on** inside **async fn**
- **.await** doesn’t **block** the whole thread but wait for the **specific Future**
- Allows the other tasks to run if the future unable to progress or busy
- Imagine we have three async fn: **learn_song**, **sing_song**, and **dance**

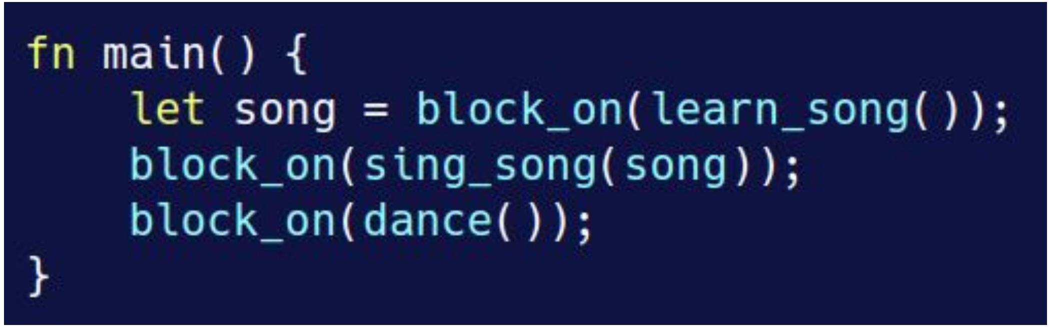
Async/.await Primer (Example)

- One way to do **learn**, **sing**, and **dance** would be to block on each of these individually

- However, we're not giving the best performance possible this way



```
async fn learn_song() -> Song { ... }
async fn sing_song(song: Song) { ... }
async fn dance() { ... }
```



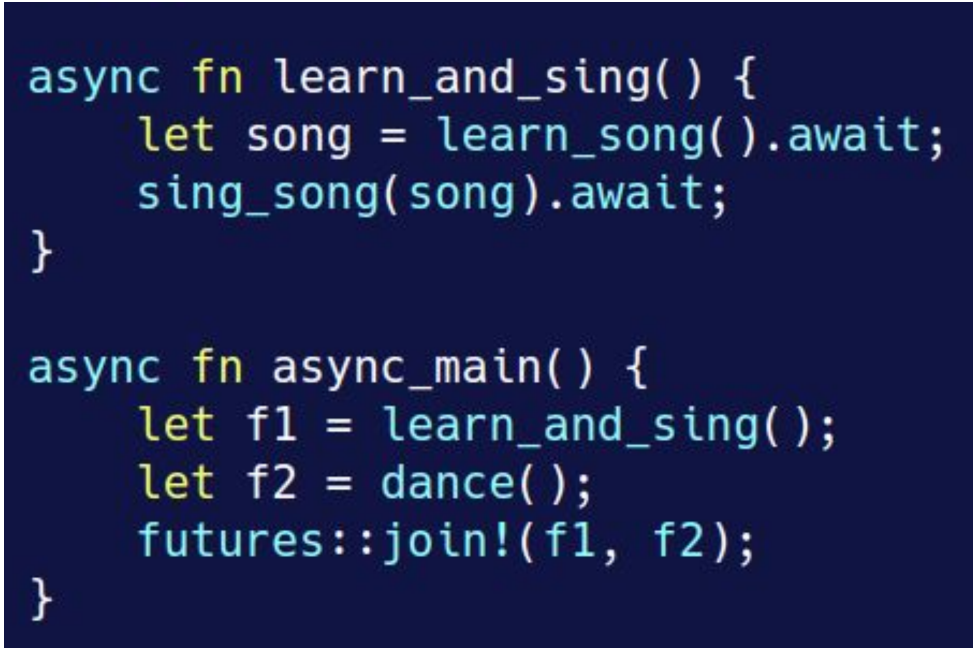
```
fn main() {
    let song = block_on(learn_song());
    block_on(sing_song(song));
    block_on(dance());
}
```

Async/.await Primer (Example)

- This way we're doing one thing at once
- Indeed, we have to learn before singing the song but it’s possible to dance at the same time as learning and singing!
- For this we can create two **concurrently** running **async fn**

Async/.await Primer (Example)

- We’ll wait for learning the song before singing
- Will be using .await to wait asynchronously rather than blocking whole thread
- This is how it can **dance** while learning and singing
- **join!** is like **.await** but can wait for multiple futures concurrently



```rust
async fn learn_and_sing() {
    let song = learn_song().await;
    sing_song(song).await;
}

async fn async_main() {
    let f1 = learn_and_sing();
    let f2 = dance();
    futures::join!(f1, f2);
}
```

Async/.await Primer (Example Summary)

- Learning the song must happen **before singing** the song
- But both learning and singing can happen at the same time as dancing
- Using **block_on** in **learning_and_singing** instead .**await** would have **blocked** the whole thread
- This would make it **impossible to dance** at the same time

Async/.await Primer (Example Summary)

- By **.await**-ing the learn_song future, other tasks can take over the current thread if **learn_song** is blocked
- This makes it possible to run multiple futures to completion concurrently on the same thread

Resources

Book : https://rust-lang.github.io/async-book/

Link to the article : https://thomashartmann.dev/blog/async-rust/

Source code repository :

https://github.com/mohammadrajabraza/class-codes-piaic-q3

Summary