Interior vs. exterior mutability

Rust Vienna

Exterior mutability

A simple Counter

```rust
pub struct Counter {
    value: u64,
}

impl Counter {
    pub fn new() -> Self {
        Counter { value: 0 }
    }

    pub fn get(&self) -> u64 {
        self.value
    }

    pub fn inc(&mut self, amount: u64) -> u64 {
        self.value = self.value.saturating_add(amount);
        self.value
    }
}
```

A simple Counter

```rust
pub struct Counter {
    value: u64,
}

impl Counter {
    pub fn new() -> Self {
        Counter { value: 0 }
    }

    pub fn get(&self) -> u64 {
        self.value
    }

    pub fn inc(&mut self, amount: u64) -> u64 {
        self.value = self.value.saturating_add(amount);
        self.value
    }
}
```

Using the Counter

```
fn main() {
    let mut counter = Counter::new();
    println!("get(): {}", counter.get());
    println!("inc(1): {}", counter.inc(1));
    println!("inc(4): {}", counter.inc(4));
}
```

```
$ cargo run
get(): 0
inc(1): 1
inc(4): 5
```

Using the Counter

```
fn main() {
    let mut counter = Counter::new();
    println!("get(): {}", counter.get());
    println!("inc(1): {}", counter.inc(1));
    println!("inc(4): {}", counter.inc(4));
}
```

```
$ cargo run
get(): 0
inc(1): 1
inc(4): 5
```

Interior mutability

`RefCell<T>`

- Allows mutation with immutable references
- Borrow-checks at runtime instead of compile time
- **Caution:** borrowing panics when it is already mutably borrowed
- `RefCell<T>` is `!Sync`

Embedding the Counter

```
struct CounterService {
    counter: RefCell<Counter>,
}

impl CounterService {
    fn new() -> Self {
        CounterService { counter: RefCell::new(Counter::new()) }
    }

    fn get(&self) -> u64 {
        self.counter.borrow().get()
    }

    fn inc(&self, amount: u64) -> u64 {
        self.counter.borrow_mut().inc(amount)
    }
}
```

Embedding the Counter

```
struct CounterService {
    counter: RefCell<Counter>,
}

impl CounterService {
    fn new() -> Self {
        CounterService { counter: RefCell::new(Counter::new()) }
    }

    fn get(&self) -> u64 {
        self.counter.borrow().get()
    }

    fn inc(&self, amount: u64) -> u64 {
        self.counter.borrow_mut().inc(amount)
    }
}
```

Embedding the Counter

```
fn main() {
    let counter = CounterService::new();
    println!("get(): {}", counter.get());
    println!("inc(1): {}", counter.inc(1));
    println!("inc(4): {}", counter.inc(4));
}
```

```
$ cargo run
get(): 0
inc(1): 1
inc(4): 5
```

`Mutex<T>`

- Like `RefCell<T>` but blocks instead of panicking
- Can be passed between threads (it is `Send`)
- Mutex can get poisoned

Using `Mutex<T>`

```
struct CounterService {
    counter: Mutex<Counter>,
}

impl CounterService {
    fn new() -> Self {
        CounterService {
            counter: Mutex::new(Counter::new()),
        }
    }

    fn get(&self) -> u64 {
        let counter = self.counter.lock().expect("mutex poisoned");
        counter.get()
    }

    fn inc(&self, amount: u64) -> u64 {
        let mut counter = self.counter.lock().expect("mutex poisoned");
        counter.inc(amount)
    }
}
```

Using `Mutex<T>`

```
struct CounterService {
    counter: Mutex<Counter>,
}

impl CounterService {
    fn new() -> Self {
        CounterService {
            counter: Mutex::new(Counter::new()),
        }
    }

    fn get(&self) -> u64 {
        let counter = self.counter.lock().expect("mutex poisoned");
        counter.get()
    }

    fn inc(&self, amount: u64) -> u64 {
        let mut counter = self.counter.lock().expect("mutex poisoned");
        counter.inc(amount)
    }
}
```

Using `Mutex<T>`

```
struct CounterService {
    counter: Mutex<Counter>,
}

impl CounterService {
    fn new() -> Self {
        CounterService {
            counter: Mutex::new(Counter::new()),
        }
    }

    fn get(&self) -> u64 {
        let counter = self.counter.lock().expect("mutex poisoned");
        counter.get()
    }

    fn inc(&self, amount: u64) -> u64 {
        let mut counter = self.counter.lock().expect("mutex poisoned");
        counter.inc(amount)
    }
}
```

RwLock<T>

- Allows concurrent immutable borrows
- Can also get poisoned, but only due to a panic while holding a write lock
- Acquiring a read lock panics when the thread already holds a read lock

Using RwLock<T>

```rust
struct CounterService {
    counter: RwLock<Counter>,
}

impl CounterService {
    fn new() -> Self {
        CounterService {
            counter: RwLock::new(Counter::new()),
        }
    }

    fn get(&self) -> u64 {
        let counter = self.counter.read().expect("rw-lock poisoned");
        counter.get()
    }

    fn inc(&self, amount: u64) -> u64 {
        let mut counter = self.counter.write().expect("rw-lock poisoned");
        counter.inc(amount)
    }
}
```

Using RwLock<T>

```
struct CounterService {
    counter: RwLock<Counter>,
}

impl CounterService {
    fn new() -> Self {
        CounterService {
            counter: RwLock::new(Counter::new()),
        }
    }

    fn get(&self) -> u64 {
        let counter = self.counter.read().expect("rw-lock poisoned");
        counter.get()
    }

    fn inc(&self, amount: u64) -> u64 {
        let mut counter = self.counter.write().expect("rw-lock poisoned");
        counter.inc(amount)
    }
}
```

Using RwLock<T>

```
struct CounterService {
    counter: RwLock<Counter>,
}

impl CounterService {
    fn new() -> Self {
        CounterService {
            counter: RwLock::new(Counter::new()),
        }
    }

    fn get(&self) -> u64 {
        let counter = self.counter.read().expect("rw-lock poisoned");
        counter.get()
    }

    fn inc(&self, amount: u64) -> u64 {
        let mut counter = self.counter.write().expect("rw-lock poisoned");
        counter.inc(amount)
    }
}
```

Conclusion

- Exterior mutability is the idiomatic approach
- Use the borrow-checker to your advantage!
- But if you are limited to immutable references, there are options available

Bonus Slides

Actor model

Actor model

- Concurrent tasks
- Communication via channels
- Local mutability

Handling messages

```
struct Get;

impl Handler<Get> for Counter {
    type Result = u64;

    fn handle(&mut self, _m: Get) -> Self::Result {
        self.value
    }
}

struct Inc(u64);

impl Handler<Inc> for Counter {
    type Result = u64;

    fn handle(&mut self, Inc(amount): Inc) -> Self::Result {
        self.value = self.value.saturating_add(amount);
        self.value
    }
}
```

Usage

```rust
#[tokio::main]
async fn main() {
    let counter = actor::spawn(Counter::new());

    println!("Get: {}", counter.send(Get).await);
    println!("Inc(1): {}", counter.send(Inc(1)).await);
    println!("Inc(4): {}", counter.send(Inc(4)).await);
}
```