Characteristics of Object-Oriented Languages

There is no consensus in the programming community about what features a language must have to be considered object-oriented. Rust is influenced by many programming paradigms, including OOP; for example, we explored the features that came from functional programming in Chapter 13. Arguably, OOP languages share certain common characteristics, namely objects, encapsulation, and inheritance. Let's look at what each of those characteristics means and whether Rust supports it.

Objects Contain Data and Behavior

The book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1994), colloquially referred to as *The Gang of Four* book, is a catalog of object-oriented design patterns. It defines OOP this way:

> Object-oriented programs are made up of objects. An *object* packages both data and the procedures that operate on that data. The procedures are typically called *methods* or *operations*.

Using this definition, Rust is object-oriented: structs and enums have data, and `impl` blocks provide methods on structs and enums. Even though structs and enums with methods aren't *called* objects, they provide the same functionality, according to the Gang of Four's definition of objects.

Encapsulation that Hides Implementation Details

Another aspect commonly associated with OOP is the idea of *encapsulation*, which means that the implementation details of an object aren't accessible to code using that object. Therefore, the only way to interact with an object is through its public API; code using the object shouldn't be able to reach into the object's internals and change data or behavior directly. This enables the programmer to change and refactor an object's internals without needing to change the code that uses the object.

We discussed how to control encapsulation in Chapter 7: we can use the `pub` keyword to decide which modules, types, functions, and methods in our code should be public, and by

default everything else is private. For example, we can define a struct `AveragedCollection` that has a field containing a vector of `i32` values. The struct can also have a field that contains the average of the values in the vector, meaning the average doesn't have to be computed on demand whenever anyone needs it. In other words, `AveragedCollection` will cache the calculated average for us. Listing 17-1 has the definition of the `AveragedCollection` struct:

Filename: src/lib.rs

```
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}
```

Listing 17-1: An `AveragedCollection` struct that maintains a list of integers and the average of the items in the collection

The struct is marked `pub` so that other code can use it, but the fields within the struct remain private. This is important in this case because we want to ensure that whenever a value is added or removed from the list, the average is also updated. We do this by implementing `add`, `remove`, and `average` methods on the struct, as shown in Listing 17-2:

Filename: src/lib.rs

```
impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }

    pub fn remove(&mut self) -> Option<i32> {
        let result = self.list.pop();
        match result {
            Some(value) => {
                self.update_average();
                Some(value)
            }
            None => None,
        }
    }

    pub fn average(&self) -> f64 {
        self.average
    }

    fn update_average(&mut self) {
        let total: i32 = self.list.iter().sum();
        self.average = total as f64 / self.list.len() as f64;
    }
}
```

Listing 17-2: Implementations of the public methods `add`, `remove`, and `average` on `AveragedCollection`

The public methods `add`, `remove`, and `average` are the only ways to access or modify data in an instance of `AveragedCollection`. When an item is added to `list` using the `add` method or removed using the `remove` method, the implementations of each call the private `update_average` method that handles updating the `average` field as well.

We leave the `list` and `average` fields private so there is no way for external code to add or remove items to or from the `list` field directly; otherwise, the `average` field might become out of sync when the `list` changes. The `average` method returns the value in the `average` field, allowing external code to read the `average` but not modify it.

Because we've encapsulated the implementation details of the struct `AveragedCollection`, we can easily change aspects, such as the data structure, in the future. For instance, we could use a `HashSet<i32>` instead of a `Vec<i32>` for the `list` field. As long as the signatures of the `add`, `remove`, and `average` public methods stay the same, code using `AveragedCollection` wouldn't need to change in order to compile. If we made `list` public instead, this wouldn't necessarily be the case: `HashSet<i32>` and `Vec<i32>` have different methods for adding and removing items, so the external code would likely have to change if it were modifying `list` directly.

If encapsulation is a required aspect for a language to be considered object-oriented, then Rust meets that requirement. The option to use `pub` or not for different parts of code enables encapsulation of implementation details.

Inheritance as a Type System and as Code Sharing

Inheritance is a mechanism whereby an object can inherit elements from another object's definition, thus gaining the parent object's data and behavior without you having to define them again.

If a language must have inheritance to be an object-oriented language, then Rust is not one. There is no way to define a struct that inherits the parent struct's fields and method implementations without using a macro.

However, if you're used to having inheritance in your programming toolbox, you can use other solutions in Rust, depending on your reason for reaching for inheritance in the first place.

You would choose inheritance for two main reasons. One is for reuse of code: you can implement particular behavior for one type, and inheritance enables you to reuse that implementation for a different type. You can do this in a limited way in Rust code using default trait method implementations, which you saw in Listing 10-14 when we added a default implementation of the `summarize` method on the `Summary` trait. Any type implementing the `Summary` trait would have the `summarize` method available on it without any further code. This is similar to a parent class having an implementation of a method and an inheriting child class also having the implementation of the method. We can also override the default implementation of the `summarize` method when we implement the `Summary` trait, which is similar to a child class overriding the implementation of a method inherited from a parent class.

The other reason to use inheritance relates to the type system: to enable a child type to be used in the same places as the parent type. This is also called *polymorphism*, which means that you can substitute multiple objects for each other at runtime if they share certain characteristics.

Polymorphism

To many people, polymorphism is synonymous with inheritance. But it's actually a more general concept that refers to code that can work with data of multiple types. For inheritance, those types are generally subclasses.

> Rust instead uses generics to abstract over different possible types and trait bounds to impose constraints on what those types must provide. This is sometimes called *bounded parametric polymorphism*.

Inheritance has recently fallen out of favor as a programming design solution in many programming languages because it's often at risk of sharing more code than necessary. Subclasses shouldn't always share all characteristics of their parent class but will do so with inheritance. This can make a program's design less flexible. It also introduces the possibility of calling methods on subclasses that don't make sense or that cause errors because the methods don't apply to the subclass. In addition, some languages will only allow single inheritance (meaning a subclass can only inherit from one class), further restricting the flexibility of a program's design.

For these reasons, Rust takes the different approach of using trait objects instead of inheritance. Let's look at how trait objects enable polymorphism in Rust.

Using Trait Objects That Allow for Values of Different Types

In Chapter 8, we mentioned that one limitation of vectors is that they can store elements of only one type. We created a workaround in Listing 8-9 where we defined a `SpreadsheetCell` enum that had variants to hold integers, floats, and text. This meant we could store different types of data in each cell and still have a vector that represented a row of cells. This is a perfectly good solution when our interchangeable items are a fixed set of types that we know when our code is compiled.

However, sometimes we want our library user to be able to extend the set of types that are valid in a particular situation. To show how we might achieve this, we'll create an example graphical user interface (GUI) tool that iterates through a list of items, calling a `draw` method on each one to draw it to the screen—a common technique for GUI tools. We'll create a library crate called `gui` that contains the structure of a GUI library. This crate might include some types for people to use, such as `Button` or `TextField`. In addition, `gui` users will want to create their own types that can be drawn: for instance, one programmer might add an `Image` and another might add a `SelectBox`.

We won't implement a fully fledged GUI library for this example but will show how the pieces would fit together. At the time of writing the library, we can't know and define all the types other programmers might want to create. But we do know that `gui` needs to keep track of many values of different types, and it needs to call a `draw` method on each of these differently typed values. It doesn't need to know exactly what will happen when we call the `draw` method, just that the value will have that method available for us to call.

To do this in a language with inheritance, we might define a class named `Component` that has a method named `draw` on it. The other classes, such as `Button`, `Image`, and `SelectBox`, would inherit from `Component` and thus inherit the `draw` method. They could each override the `draw` method to define their custom behavior, but the framework could treat all of the types as if they were `Component` instances and call `draw` on them. But because Rust doesn't have inheritance, we need another way to structure the `gui` library to allow users to extend it with new types.

Defining a Trait for Common Behavior

To implement the behavior we want `gui` to have, we'll define a trait named `Draw` that will have one method named `draw`. Then we can define a vector that takes a *trait object*. A trait object points to both an instance of a type implementing our specified trait and a table used to look

up trait methods on that type at runtime. We create a trait object by specifying some sort of pointer, such as a `&` reference or a `Box<T>` smart pointer, then the `dyn` keyword, and then specifying the relevant trait. (We'll talk about the reason trait objects must use a pointer in Chapter 19 in the section "Dynamically Sized Types and the `Sized` Trait.") We can use trait objects in place of a generic or concrete type. Wherever we use a trait object, Rust's type system will ensure at compile time that any value used in that context will implement the trait object's trait. Consequently, we don't need to know all the possible types at compile time.

We've mentioned that, in Rust, we refrain from calling structs and enums "objects" to distinguish them from other languages' objects. In a struct or enum, the data in the struct fields and the behavior in `impl` blocks are separated, whereas in other languages, the data and behavior combined into one concept is often labeled an object. However, trait objects *are* more like objects in other languages in the sense that they combine data and behavior. But trait objects differ from traditional objects in that we can't add data to a trait object. Trait objects aren't as generally useful as objects in other languages: their specific purpose is to allow abstraction across common behavior.

Listing 17-3 shows how to define a trait named `Draw` with one method named `draw`:

Filename: src/lib.rs

```
pub trait Draw {
    fn draw(&self);
}
```

Listing 17-3: Definition of the `Draw` trait

This syntax should look familiar from our discussions on how to define traits in Chapter 10. Next comes some new syntax: Listing 17-4 defines a struct named `Screen` that holds a vector named `components`. This vector is of type `Box<dyn Draw>`, which is a trait object; it's a stand-in for any type inside a `Box` that implements the `Draw` trait.

Filename: src/lib.rs

```
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

Listing 17-4: Definition of the `Screen` struct with a `components` field holding a vector of trait objects that implement the `Draw` trait

On the `Screen` struct, we'll define a method named `run` that will call the `draw` method on each of its `components`, as shown in Listing 17-5:

Filename: src/lib.rs

```
impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

Listing 17-5: A `run` method on `Screen` that calls the `draw` method on each component

This works differently from defining a struct that uses a generic type parameter with trait bounds. A generic type parameter can only be substituted with one concrete type at a time, whereas trait objects allow for multiple concrete types to fill in for the trait object at runtime. For example, we could have defined the `Screen` struct using a generic type and a trait bound as in Listing 17-6:

Filename: src/lib.rs

```
pub struct Screen<T: Draw> {
    pub components: Vec<T>,
}

impl<T> Screen<T>
where
    T: Draw,
{
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

Listing 17-6: An alternate implementation of the `Screen` struct and its `run` method using generics and trait bounds

This restricts us to a `Screen` instance that has a list of components all of type `Button` or all of type `TextField`. If you'll only ever have homogeneous collections, using generics and trait bounds is preferable because the definitions will be monomorphized at compile time to use the concrete types.

On the other hand, with the method using trait objects, one `Screen` instance can hold a `Vec<T>` that contains a `Box<Button>` as well as a `Box<TextField>`. Let's look at how this works, and then we'll talk about the runtime performance implications.

Implementing the Trait

Now we'll add some types that implement the `Draw` trait. We'll provide the `Button` type. Again, actually implementing a GUI library is beyond the scope of this book, so the `draw` method won't have any useful implementation in its body. To imagine what the implementation might look like, a `Button` struct might have fields for `width`, `height`, and `label`, as shown in Listing 17-7:

Filename: src/lib.rs

```
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // code to actually draw a button
    }
}
```

Listing 17-7: A `Button` struct that implements the `Draw` trait

The `width`, `height`, and `label` fields on `Button` will differ from the fields on other components; for example, a `TextField` type might have those same fields plus a `placeholder` field. Each of the types we want to draw on the screen will implement the `Draw` trait but will use different code in the `draw` method to define how to draw that particular type, as `Button` has here (without the actual GUI code, as mentioned). The `Button` type, for instance, might have an additional `impl` block containing methods related to what happens when a user clicks the button. These kinds of methods won't apply to types like `TextField`.

If someone using our library decides to implement a `SelectBox` struct that has `width`, `height`, and `options` fields, they implement the `Draw` trait on the `SelectBox` type as well, as shown in Listing 17-8:

Filename: src/main.rs

```
use gui::Draw;

struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select box
    }
}
```

Listing 17-8: Another crate using `gui` and implementing the `Draw` trait on a `SelectBox` struct

Our library's user can now write their `main` function to create a `Screen` instance. To the `Screen` instance, they can add a `SelectBox` and a `Button` by putting each in a `Box<T>` to become a trait object. They can then call the `run` method on the `Screen` instance, which will call `draw` on each of the components. Listing 17-9 shows this implementation:

Filename: src/main.rs

```
use gui::{Button, Screen};

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No"),
                ],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };

    screen.run();
}
```

Listing 17-9: Using trait objects to store values of different types that implement the same trait

When we wrote the library, we didn't know that someone might add the `SelectBox` type, but our `Screen` implementation was able to operate on the new type and draw it because `SelectBox` implements the `Draw` trait, which means it implements the `draw` method.

This concept—of being concerned only with the messages a value responds to rather than the value's concrete type—is similar to the concept of *duck typing* in dynamically typed languages: if it walks like a duck and quacks like a duck, then it must be a duck! In the implementation of `run` on `Screen` in Listing 17-5, `run` doesn't need to know what the concrete type of each component is. It doesn't check whether a component is an instance of a `Button` or a `SelectBox`, it just calls the `draw` method on the component. By specifying `Box<dyn Draw>` as the type of the values in the `components` vector, we've defined `Screen` to need values that we can call the `draw` method on.

The advantage of using trait objects and Rust's type system to write code similar to code using duck typing is that we never have to check whether a value implements a particular method at runtime or worry about getting errors if a value doesn't implement a method but we call it anyway. Rust won't compile our code if the values don't implement the traits that the trait objects need.

For example, Listing 17-10 shows what happens if we try to create a `Screen` with a `String` as a component:

Filename: src/main.rs

```
use gui::Screen;

fn main() {
    let screen = Screen {
        components: vec![Box::new(String::from("Hi"))],
    };

    screen.run();
}
```

Listing 17-10: Attempting to use a type that doesn't implement the trait object's trait

We'll get this error because `String` doesn't implement the `Draw` trait:

```
$ cargo run
   Compiling gui v0.1.0 (file:///projects/gui)
error[E0277]: the trait bound `String: Draw` is not satisfied
 --> src/main.rs:5:26
  |
5 |         components: vec![Box::new(String::from("Hi"))],
  |                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `Draw` is not
implemented for `String`
  |
  = help: the trait `Draw` is implemented for `Button`
  = note: required for the cast from `Box<String>` to `Box<dyn Draw>`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `gui` (bin "gui") due to 1 previous error
```

This error lets us know that either we're passing something to `Screen` we didn't mean to pass and so should pass a different type or we should implement `Draw` on `String` so that `Screen` is able to call `draw` on it.

Trait Objects Perform Dynamic Dispatch

Recall in the "Performance of Code Using Generics" section in Chapter 10 our discussion on the monomorphization process performed by the compiler when we use trait bounds on generics: the compiler generates nongeneric implementations of functions and methods for each concrete type that we use in place of a generic type parameter. The code that results from monomorphization is doing *static dispatch*, which is when the compiler knows what method you're calling at compile time. This is opposed to *dynamic dispatch*, which is when the compiler can't tell at compile time which method you're calling. In dynamic dispatch cases, the compiler emits code that at runtime will figure out which method to call.

When we use trait objects, Rust must use dynamic dispatch. The compiler doesn't know all the types that might be used with the code that's using trait objects, so it doesn't know which method implemented on which type to call. Instead, at runtime, Rust uses the pointers inside the trait object to know which method to call. This lookup incurs a runtime cost that doesn't occur with static dispatch. Dynamic dispatch also prevents the compiler from choosing to inline a method's code, which in turn prevents some optimizations. However, we did get extra flexibility in the code that we wrote in Listing 17-5 and were able to support in Listing 17-9, so it's a trade-off to consider.

Implementing an Object-Oriented Design Pattern

The *state pattern* is an object-oriented design pattern. The crux of the pattern is that we define a set of states a value can have internally. The states are represented by a set of *state objects*, and the value's behavior changes based on its state. We're going to work through an example of a blog post struct that has a field to hold its state, which will be a state object from the set "draft", "review", or "published".

The state objects share functionality: in Rust, of course, we use structs and traits rather than objects and inheritance. Each state object is responsible for its own behavior and for governing when it should change into another state. The value that holds a state object knows nothing about the different behavior of the states or when to transition between states.

The advantage of using the state pattern is that, when the business requirements of the program change, we won't need to change the code of the value holding the state or the code that uses the value. We'll only need to update the code inside one of the state objects to change its rules or perhaps add more state objects.

First, we're going to implement the state pattern in a more traditional object-oriented way, then we'll use an approach that's a bit more natural in Rust. Let's dig in to incrementally implementing a blog post workflow using the state pattern.

The final functionality will look like this:

1. A blog post starts as an empty draft.
2. When the draft is done, a review of the post is requested.
3. When the post is approved, it gets published.
4. Only published blog posts return content to print, so unapproved posts can't accidentally be published.

Any other changes attempted on a post should have no effect. For example, if we try to approve a draft blog post before we've requested a review, the post should remain an unpublished draft.

Listing 17-11 shows this workflow in code form: this is an example usage of the API we'll implement in a library crate named `blog`. This won't compile yet because we haven't implemented the `blog` crate.

Filename: src/main.rs

```
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());

    post.request_review();
    assert_eq!("", post.content());

    post.approve();
    assert_eq!("I ate a salad for lunch today", post.content());
}
```

Listing 17-11: Code that demonstrates the desired behavior we want our `blog` crate to have

We want to allow the user to create a new draft blog post with `Post::new`. We want to allow text to be added to the blog post. If we try to get the post's content immediately, before approval, we shouldn't get any text because the post is still a draft. We've added `assert_eq!` in the code for demonstration purposes. An excellent unit test for this would be to assert that a draft blog post returns an empty string from the `content` method, but we're not going to write tests for this example.

Next, we want to enable a request for a review of the post, and we want `content` to return an empty string while waiting for the review. When the post receives approval, it should get published, meaning the text of the post will be returned when `content` is called.

Notice that the only type we're interacting with from the crate is the `Post` type. This type will use the state pattern and will hold a value that will be one of three state objects representing the various states a post can be in—draft, waiting for review, or published. Changing from one state to another will be managed internally within the `Post` type. The states change in response to the methods called by our library's users on the `Post` instance, but they don't have to manage the state changes directly. Also, users can't make a mistake with the states, like publishing a post before it's reviewed.

Defining `Post` and Creating a New Instance in the Draft State

Let's get started on the implementation of the library! We know we need a public `Post` struct that holds some content, so we'll start with the definition of the struct and an associated public `new` function to create an instance of `Post`, as shown in Listing 17-12. We'll also make a private `State` trait that will define the behavior that all state objects for a `Post` must have.

Then `Post` will hold a trait object of `Box<dyn State>` inside an `Option<T>` in a private field named `state` to hold the state object. You'll see why the `Option<T>` is necessary in a bit.

Filename: src/lib.rs

```
pub struct Post {
    state: Option<Box<dyn State>>,
    content: String,
}

impl Post {
    pub fn new() -> Post {
        Post {
            state: Some(Box::new(Draft {})),
            content: String::new(),
        }
    }
}

trait State {}

struct Draft {}

impl State for Draft {}
```

Listing 17-12: Definition of a `Post` struct and a `new` function that creates a new `Post` instance, a `State` trait, and a `Draft` struct

The `State` trait defines the behavior shared by different post states. The state objects are `Draft`, `PendingReview`, and `Published`, and they will all implement the `State` trait. For now, the trait doesn't have any methods, and we'll start by defining just the `Draft` state because that is the state we want a post to start in.

When we create a new `Post`, we set its `state` field to a `Some` value that holds a `Box`. This `Box` points to a new instance of the `Draft` struct. This ensures whenever we create a new instance of `Post`, it will start out as a draft. Because the `state` field of `Post` is private, there is no way to create a `Post` in any other state! In the `Post::new` function, we set the `content` field to a new, empty `String`.

Storing the Text of the Post Content

We saw in Listing 17-11 that we want to be able to call a method named `add_text` and pass it a `&str` that is then added as the text content of the blog post. We implement this as a method, rather than exposing the `content` field as `pub`, so that later we can implement a method that

will control how the `content` field's data is read. The `add_text` method is pretty straightforward, so let's add the implementation in Listing 17-13 to the `impl Post` block:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

Listing 17-13: Implementing the `add_text` method to add text to a post's `content`

The `add_text` method takes a mutable reference to `self`, because we're changing the `Post` instance that we're calling `add_text` on. We then call `push_str` on the `String` in `content` and pass the `text` argument to add to the saved `content`. This behavior doesn't depend on the state the post is in, so it's not part of the state pattern. The `add_text` method doesn't interact with the `state` field at all, but it is part of the behavior we want to support.

Ensuring the Content of a Draft Post Is Empty

Even after we've called `add_text` and added some content to our post, we still want the `content` method to return an empty string slice because the post is still in the draft state, as shown on line 7 of Listing 17-11. For now, let's implement the `content` method with the simplest thing that will fulfill this requirement: always returning an empty string slice. We'll change this later once we implement the ability to change a post's state so it can be published. So far, posts can only be in the draft state, so the post content should always be empty. Listing 17-14 shows this placeholder implementation:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        ""
    }
}
```

Listing 17-14: Adding a placeholder implementation for the `content` method on `Post` that always returns an empty string slice

With this added `content` method, everything in Listing 17-11 up to line 7 works as intended.

Requesting a Review of the Post Changes Its State

Next, we need to add functionality to request a review of a post, which should change its state from `Draft` to `PendingReview`. Listing 17-15 shows this code:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn request_review(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.request_review())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        Box::new(PendingReview {})
    }
}

struct PendingReview {}

impl State for PendingReview {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }
}
```

Listing 17-15: Implementing `request_review` methods on `Post` and the `State` trait

We give `Post` a public method named `request_review` that will take a mutable reference to `self`. Then we call an internal `request_review` method on the current state of `Post`, and this second `request_review` method consumes the current state and returns a new state.

We add the `request_review` method to the `State` trait; all types that implement the trait will now need to implement the `request_review` method. Note that rather than having `self`, `&self`, or `&mut self` as the first parameter of the method, we have `self: Box<Self>`. This syntax means the method is only valid when called on a `Box` holding the type. This syntax takes ownership of `Box<Self>`, invalidating the old state so the state value of the `Post` can transform into a new state.

To consume the old state, the `request_review` method needs to take ownership of the state value. This is where the `Option` in the `state` field of `Post` comes in: we call the `take` method to take the `Some` value out of the `state` field and leave a `None` in its place, because Rust doesn't let us have unpopulated fields in structs. This lets us move the `state` value out of `Post` rather than borrowing it. Then we'll set the post's `state` value to the result of this operation.

We need to set `state` to `None` temporarily rather than setting it directly with code like `self.state = self.state.request_review();` to get ownership of the `state` value. This ensures `Post` can't use the old `state` value after we've transformed it into a new state.

The `request_review` method on `Draft` returns a new, boxed instance of a new `PendingReview` struct, which represents the state when a post is waiting for a review. The `PendingReview` struct also implements the `request_review` method but doesn't do any transformations. Rather, it returns itself, because when we request a review on a post already in the `PendingReview` state, it should stay in the `PendingReview` state.

Now we can start seeing the advantages of the state pattern: the `request_review` method on `Post` is the same no matter its `state` value. Each state is responsible for its own rules.

We'll leave the `content` method on `Post` as is, returning an empty string slice. We can now have a `Post` in the `PendingReview` state as well as in the `Draft` state, but we want the same behavior in the `PendingReview` state. Listing 17-11 now works up to line 10!

Adding `approve` to Change the Behavior of `content`

The `approve` method will be similar to the `request_review` method: it will set `state` to the value that the current state says it should have when that state is approved, as shown in Listing 17-16:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn approve(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.approve())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
    fn approve(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn State> {
        self
    }
}

struct PendingReview {}

impl State for PendingReview {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn State> {
        Box::new(Published {})
    }
}

struct Published {}

impl State for Published {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }

    fn approve(self: Box<Self>) -> Box<dyn State> {
        self
    }
}
```

Listing 17-16: Implementing the `approve` method on `Post` and the `State` trait

We add the `approve` method to the `State` trait and add a new struct that implements `State`, the `Published` state.

Similar to the way `request_review` on `PendingReview` works, if we call the `approve` method on a `Draft`, it will have no effect because `approve` will return `self`. When we call `approve` on

`PendingReview`, it returns a new, boxed instance of the `Published` struct. The `Published` struct implements the `State` trait, and for both the `request_review` method and the `approve` method, it returns itself, because the post should stay in the `Published` state in those cases.

Now we need to update the `content` method on `Post`. We want the value returned from `content` to depend on the current state of the `Post`, so we're going to have the `Post` delegate to a `content` method defined on its `state`, as shown in Listing 17-17:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        self.state.as_ref().unwrap().content(self)
    }
    // --snip--
}
```

Listing 17-17: Updating the `content` method on `Post` to delegate to a `content` method on `State`

Because the goal is to keep all these rules inside the structs that implement `State`, we call a `content` method on the value in `state` and pass the post instance (that is, `self`) as an argument. Then we return the value that's returned from using the `content` method on the `state` value.

We call the `as_ref` method on the `Option` because we want a reference to the value inside the `Option` rather than ownership of the value. Because `state` is an `Option<Box<dyn State>>`, when we call `as_ref`, an `Option<&Box<dyn State>>` is returned. If we didn't call `as_ref`, we would get an error because we can't move `state` out of the borrowed `&self` of the function parameter.

We then call the `unwrap` method, which we know will never panic, because we know the methods on `Post` ensure that `state` will always contain a `Some` value when those methods are done. This is one of the cases we talked about in the "Cases In Which You Have More Information Than the Compiler" section of Chapter 9 when we know that a `None` value is never possible, even though the compiler isn't able to understand that.

At this point, when we call `content` on the `&Box<dyn State>`, deref coercion will take effect on the `&` and the `Box` so the `content` method will ultimately be called on the type that implements the `State` trait. That means we need to add `content` to the `State` trait definition, and that is where we'll put the logic for what content to return depending on which state we have, as shown in Listing 17-18:

Filename: src/lib.rs

```
trait State {
    // --snip--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        ""
    }
}

// --snip--
struct Published {}

impl State for Published {
    // --snip--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        &post.content
    }
}
```

Listing 17-18: Adding the `content` method to the `State` trait

We add a default implementation for the `content` method that returns an empty string slice. That means we don't need to implement `content` on the `Draft` and `PendingReview` structs. The `Published` struct will override the `content` method and return the value in `post.content`.

Note that we need lifetime annotations on this method, as we discussed in Chapter 10. We're taking a reference to a `post` as an argument and returning a reference to part of that `post`, so the lifetime of the returned reference is related to the lifetime of the `post` argument.

And we're done—all of Listing 17-11 now works! We've implemented the state pattern with the rules of the blog post workflow. The logic related to the rules lives in the state objects rather than being scattered throughout `Post`.

Why Not An Enum?

You may have been wondering why we didn't use an `enum` with the different possible post states as variants. That's certainly a possible solution, try it and compare the end results to see which you prefer! One disadvantage of using an enum is every place that checks the value of the enum will need a `match` expression or similar to handle every possible variant. This could get more repetitive than this trait object solution.

Trade-offs of the State Pattern

We've shown that Rust is capable of implementing the object-oriented state pattern to encapsulate the different kinds of behavior a post should have in each state. The methods on `Post` know nothing about the various behaviors. The way we organized the code, we have to look in only one place to know the different ways a published post can behave: the implementation of the `State` trait on the `Published` struct.

If we were to create an alternative implementation that didn't use the state pattern, we might instead use `match` expressions in the methods on `Post` or even in the `main` code that checks the state of the post and changes behavior in those places. That would mean we would have to look in several places to understand all the implications of a post being in the published state! This would only increase the more states we added: each of those `match` expressions would need another arm.

With the state pattern, the `Post` methods and the places we use `Post` don't need `match` expressions, and to add a new state, we would only need to add a new struct and implement the trait methods on that one struct.

The implementation using the state pattern is easy to extend to add more functionality. To see the simplicity of maintaining code that uses the state pattern, try a few of these suggestions:

- Add a `reject` method that changes the post's state from `PendingReview` back to `Draft`.
- Require two calls to `approve` before the state can be changed to `Published`.
- Allow users to add text content only when a post is in the `Draft` state. Hint: have the state object responsible for what might change about the content but not responsible for modifying the `Post`.

One downside of the state pattern is that, because the states implement the transitions between states, some of the states are coupled to each other. If we add another state between `PendingReview` and `Published`, such as `Scheduled`, we would have to change the code in `PendingReview` to transition to `Scheduled` instead. It would be less work if `PendingReview` didn't need to change with the addition of a new state, but that would mean switching to another design pattern.

Another downside is that we've duplicated some logic. To eliminate some of the duplication, we might try to make default implementations for the `request_review` and `approve` methods on the `State` trait that return `self`; however, this would violate object safety, because the trait doesn't know what the concrete `self` will be exactly. We want to be able to use `State` as a trait object, so we need its methods to be object safe.

Other duplication includes the similar implementations of the `request_review` and `approve` methods on `Post`. Both methods delegate to the implementation of the same method on the

value in the `state` field of `Option` and set the new value of the `state` field to the result. If we had a lot of methods on `Post` that followed this pattern, we might consider defining a macro to eliminate the repetition (see the "Macros" section in Chapter 19).

By implementing the state pattern exactly as it's defined for object-oriented languages, we're not taking as full advantage of Rust's strengths as we could. Let's look at some changes we can make to the `blog` crate that can make invalid states and transitions into compile time errors.

Encoding States and Behavior as Types

We'll show you how to rethink the state pattern to get a different set of trade-offs. Rather than encapsulating the states and transitions completely so outside code has no knowledge of them, we'll encode the states into different types. Consequently, Rust's type checking system will prevent attempts to use draft posts where only published posts are allowed by issuing a compiler error.

Let's consider the first part of `main` in Listing 17-11:

Filename: src/main.rs

```
fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());
}
```

We still enable the creation of new posts in the draft state using `Post::new` and the ability to add text to the post's content. But instead of having a `content` method on a draft post that returns an empty string, we'll make it so draft posts don't have the `content` method at all. That way, if we try to get a draft post's content, we'll get a compiler error telling us the method doesn't exist. As a result, it will be impossible for us to accidentally display draft post content in production, because that code won't even compile. Listing 17-19 shows the definition of a `Post` struct and a `DraftPost` struct, as well as methods on each:

Filename: src/lib.rs

```
pub struct Post {
    content: String,
}

pub struct DraftPost {
    content: String,
}

impl Post {
    pub fn new() -> DraftPost {
        DraftPost {
            content: String::new(),
        }
    }

    pub fn content(&self) -> &str {
        &self.content
    }
}

impl DraftPost {
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

Listing 17-19: A `Post` with a `content` method and a `DraftPost` without a `content` method

Both the `Post` and `DraftPost` structs have a private `content` field that stores the blog post text. The structs no longer have the `state` field because we're moving the encoding of the state to the types of the structs. The `Post` struct will represent a published post, and it has a `content` method that returns the `content`.

We still have a `Post::new` function, but instead of returning an instance of `Post`, it returns an instance of `DraftPost`. Because `content` is private and there aren't any functions that return `Post`, it's not possible to create an instance of `Post` right now.

The `DraftPost` struct has an `add_text` method, so we can add text to `content` as before, but note that `DraftPost` does not have a `content` method defined! So now the program ensures all posts start as draft posts, and draft posts don't have their content available for display. Any attempt to get around these constraints will result in a compiler error.

Implementing Transitions as Transformations into Different Types

So how do we get a published post? We want to enforce the rule that a draft post has to be reviewed and approved before it can be published. A post in the pending review state should still not display any content. Let's implement these constraints by adding another struct,

`PendingReviewPost`, defining the `request_review` method on `DraftPost` to return a `PendingReviewPost`, and defining an `approve` method on `PendingReviewPost` to return a `Post`, as shown in Listing 17-20:

Filename: src/lib.rs

```
impl DraftPost {
    // --snip--
    pub fn request_review(self) -> PendingReviewPost {
        PendingReviewPost {
            content: self.content,
        }
    }
}

pub struct PendingReviewPost {
    content: String,
}

impl PendingReviewPost {
    pub fn approve(self) -> Post {
        Post {
            content: self.content,
        }
    }
}
```

Listing 17-20: A `PendingReviewPost` that gets created by calling `request_review` on `DraftPost` and an `approve` method that turns a `PendingReviewPost` into a published `Post`

The `request_review` and `approve` methods take ownership of `self`, thus consuming the `DraftPost` and `PendingReviewPost` instances and transforming them into a `PendingReviewPost` and a published `Post`, respectively. This way, we won't have any lingering `DraftPost` instances after we've called `request_review` on them, and so forth. The `PendingReviewPost` struct doesn't have a `content` method defined on it, so attempting to read its content results in a compiler error, as with `DraftPost`. Because the only way to get a published `Post` instance that does have a `content` method defined is to call the `approve` method on a `PendingReviewPost`, and the only way to get a `PendingReviewPost` is to call the `request_review` method on a `DraftPost`, we've now encoded the blog post workflow into the type system.

But we also have to make some small changes to `main`. The `request_review` and `approve` methods return new instances rather than modifying the struct they're called on, so we need to add more `let post =` shadowing assignments to save the returned instances. We also can't have the assertions about the draft and pending review posts' contents be empty strings, nor

do we need them: we can't compile code that tries to use the content of posts in those states any longer. The updated code in `main` is shown in Listing 17-21:

Filename: src/main.rs

```
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");

    let post = post.request_review();

    let post = post.approve();

    assert_eq!("I ate a salad for lunch today", post.content());
}
```

Listing 17-21: Modifications to `main` to use the new implementation of the blog post workflow

The changes we needed to make to `main` to reassign `post` mean that this implementation doesn't quite follow the object-oriented state pattern anymore: the transformations between the states are no longer encapsulated entirely within the `Post` implementation. However, our gain is that invalid states are now impossible because of the type system and the type checking that happens at compile time! This ensures that certain bugs, such as display of the content of an unpublished post, will be discovered before they make it to production.

Try the tasks suggested at the start of this section on the `blog` crate as it is after Listing 17-21 to see what you think about the design of this version of the code. Note that some of the tasks might be completed already in this design.

We've seen that even though Rust is capable of implementing object-oriented design patterns, other patterns, such as encoding state into the type system, are also available in Rust. These patterns have different trade-offs. Although you might be very familiar with object-oriented patterns, rethinking the problem to take advantage of Rust's features can provide benefits, such as preventing some bugs at compile time. Object-oriented patterns won't always be the best solution in Rust due to certain features, like ownership, that object-oriented languages don't have.

Summary

No matter whether or not you think Rust is an object-oriented language after reading this chapter, you now know that you can use trait objects to get some object-oriented features in

Rust. Dynamic dispatch can give your code some flexibility in exchange for a bit of runtime performance. You can use this flexibility to implement object-oriented patterns that can help your code's maintainability. Rust also has other features, like ownership, that object-oriented languages don't have. An object-oriented pattern won't always be the best way to take advantage of Rust's strengths, but is an available option.

Next, we'll look at patterns, which are another of Rust's features that enable lots of flexibility. We've looked at them briefly throughout the book but haven't seen their full capability yet. Let's go!