메소드 문법

메소드(method) 는 함수와 유사합니다: 이들은 `fn` 키워드와 이름을 가지고 선언되고, 파라미터와 반환값을 가지고 있으며, 다른 어딘가로부터 호출되었을때 실행될 어떤 코드를 담고 있습니다. 하지만, 메소드는 함수와는 달리 구조체의 내용 안에 정의되며 (혹은 열거형이나 트레잇 객체 안에 정의되는데, 이는 6장과 17장에서 각각 다루겠습니다), 첫번째 파라미터가 언제나 `self`인데, 이는 메소드가 호출되고 있는 구조체의 인스턴스를 나타냅니다.

메소드 정의하기

Listing 5-12에서 보는 바와 같이 `Rectangle` 인스턴스를 파라미터로 가지고 있는 `area` 함수를 바꿔서 그 대신 `Rectangle` 구조체 위에서 정의된 `area` 메소드를 만들어 봅시다:

Filename: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    length: u32,
    width: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.length * self.width
    }
}

fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Listing 5-12: `Rectangle` 구조체 상에 `area` 메소드 정의하기

`Rectangle`의 내용 안에 함수를 정의하기 위해서, `impl` (*구현: implementation*) 블록을 시작합니다. 그 다음 `area` 함수를 `impl` 중괄호 안으로 옮기고 시그니처 및 본체 내의 모든 곳에 있는 첫번째 파라미터 (지금의 경우에는 유일한 파라미터)를 `self`로 변경시킵니다. 우리가 `area` 함수를 호출하고 여기에 `rect1`을 인자로 넘기고 있는 `main` 함수에서는, 이 대신 `Rectangle` 인스턴스 상의 `area` 메소드를 호출하기 위해서 *메소드 문법(method syntax)* 를 이용할 수 있습니다. 메소드

문법은 인스턴스 다음에 위치합니다: 점을 추가하고 그 뒤를 이어 메소드 이름, 괄호, 인자들이 따라옵니다.

`area`의 시그니처 내에서는, `rectangle: &Rectangle` 대신 `&self`가 사용되었는데 이는 메소드가 `impl Rectangle` 내용물 안에 위치하고 있어 러스트가 `self`의 타입이 `Rectangle` 라는 사실을 알 수 있기 때문입니다. 우리가 `&Rectangle`이라고 썼던 것 처럼, `self` 앞에도 여전히 `&`를 사용할 필요가 있음을 주목하세요. 메소드는 `self`의 소유권을 가져갈 수도, 여기서처럼 `self`를 변경 불가능하게 빌릴 수도, 혹은 다른 파라미터와 비슷하게 변경이 가능하도록 빌려올 수도 있습니다.

여기서는 함수 버전에서 `&Rectangle`을 이용한 것과 같은 이유로 `&self`를 택했습니다: 우리는 소유권을 가져오는 것을 원하지 않으며, 다만 구조체 내의 데이터를 읽기만 하고, 쓰고 싶지는 않습니다. 만일 그 메소드가 동작하는 과정에서 메소드 호출에 사용된 인스턴스가 변하기를 원했다면, 첫번째 파라미터로 `&mut self`를 썼을테지요. 그냥 `self`을 첫번째 파라미터로 사용하여 인스턴스의 소유권을 가져오는 메소드를 작성하는 일은 드뭅니다; 이러한 테크닉은 보통 해당 메소드가 `self`을 다른 무언가로 변형시키고 이 변형 이후에 호출하는 측에서 원본 인스턴스를 사용하는 것을 막고 싶을 때 종종 쓰입니다.

함수 대신 메소드를 이용하면 생기는 주요 잇점은, 메소드 문법을 이용하여 모든 메소드 시그니처 내에서마다 `self`를 반복하여 타이핑하지 않아도 된다는 점과 더불어, 조직화에 관한 점입니다. 우리 코드를 향후 사용할 사람들이 우리가 제공하는 라이브러리 내의 다양한 곳에서 `Rectangle`이 사용 가능한 지점을 찾도록 하는 것보다 하나의 `impl` 블록 내에 해당 타입의 인스턴스로 할 수 있는 모든 것을 모아두었습니다.

-> 연산자는 어디로 갔나요?

C++ 같은 언어에서는, 메소드 호출을 위해서 서로 다른 두 개의 연산자가 사용됩니다: 만일 어떤 객체의 메소드를 직접 호출하는 중이라면 `.`를 이용하고, 어떤 객체의 포인터에서의 메소드를 호출하는 중이고 이 포인터를 역참조할 필요가 있다면 `->`를 쓰지요. 달리 표현하면, 만일 `object`가 포인터라면, `object->something()`은 `(*object).something()`과 비슷합니다.

러스트는 `->` 연산자와 동치인 연산자를 가지고 있지 않습니다; 대신, 러스트는 *자동 참조 및 역참조 (automatic referencing and dereferencing)* 이라는 기능을 가지고 있습니다. 메소드 호출은 이 동작을 포함하는 몇 군데 중 하나입니다.

동작 방식을 설명해보겠습니다: 여러분이 `object.something()`이라고 메소드를 호출했을 때, 러스트는 자동적으로 `&`나 `&mut`, 혹은 `*`을 붙여서 `object`가 해당 메소드의 시그니처와 맞도록 합니다. 달리 말하면, 다음은 동일한 표현입니다:

```
p1.distance(&p2);
(&p1).distance(&p2);
```

첫번째 표현이 훨씬 깔끔해 보입니다. 이러한 자동 참조 동작은 메소드가 명확한 수신자-즉 `self`의 타입을 가지고 있기 때문에 동작합니다. 수신자와 메소드의 이름이 주어질 때, 러스

트는 해당 메소드가 읽는지 (`&self`) 혹은 변형시키는지 (`&mut self`), 아니면 소비하는지 (`self`)를 결정론적으로 알아낼 수 있습니다. 러스트가 메소드 수신자를 암묵적으로 빌리도록 하는 사실은 실사용 환경에서 소유권을 인간공학적으로 만드는 중요한 부분입니다.

더 많은 파라미터를 가진 메소드

`Rectangle` 구조체의 두번째 메소드를 구현하여 메소드 사용법을 연습해 봅시다. 이번에는 `Rectangle`의 인스턴스가 다른 `Rectangle` 인스턴스를 가져와서 이 두번째 `Rectangle`이 `self` 내에 완전히 안에 들어갈 수 있다면 `true`를 반환하고, 그렇지 않으면 `false`를 반환하고 싶어합니다. 즉, `can_hold` 메소드를 정의했다면, Listing 5-13에서 제시하는 프로그램을 작성할 수 있기를 원합니다:

Filename: src/main.rs

```
fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };
    let rect2 = Rectangle { length: 40, width: 10 };
    let rect3 = Rectangle { length: 45, width: 60 };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

Listing 5-13: 아직 작성하지 않은 `can_hold` 메소드를 이용하는 데모

그리고 기대하는 출력은 아래와 같게 될 것인데, 이는 `rect2`의 두 차원축은 모두 `rect1`의 것보다 작지만, `rect3`은 `rect1`에 비해 가로로 더 넓기 때문입니다:

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

메소드를 정의하기를 원한다는 것을 인지하고 있으니, 이는 `impl Rectangle` 블록 내에 오게될 것입니다. 메소드의 이름은 `can_hold`이고, 또다른 `Rectangle`의 불변 참조자를 파라미터로 갖을 것입니다. 파라미터의 타입이 어떤 것이 될지는 메소드를 호출하는 코드를 살펴봄으로써 알 수 있습니다: `rect1.can_hold(&rect2)`는 `&rect2`를 넘기고 있는데, 이는 `Rectangle`의 인스턴스인 `rect2`의 불변성 빌림입니다. 이는 우리가 `rect2`를 그냥 읽기만 하길 원하기 때문에 타당하며 (쓰기를 원하는 것은 아니지요. 이는 곧 가변 빌림이 필요함을 의미합니다), `main`이 `rect2`의 소유권을 유지하여 `can_hold` 메소드 호출 이후에도 이를 다시 사용할 수 있길 원합니다. `can_hold`의 반환값은 부울린이 될 것이고, 이 구현은 `self`의 길이와 너비가 다른 `Rectangle`의 길이와 너비보다 둘다 각각 큰지를 검사할 것입니다. Listing 5-14에서 보는 것처럼, 이 새로운 `can_hold` 메소드를 Listing 5-12의 `impl` 블록에 추가해 봅시다:

Filename: src/main.rs

```
impl Rectangle {
```

```
    fn area(&self) -> u32 {
        self.length * self.width
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.length > other.length && self.width > other.width
    }
}
```

Listing 5-14: 또다른 `Rectangle` 인스턴스를 파라미터로 갖는 `can_hold` 메소드를 `Rectangle` 상에 구현하기

Listing 5-13에 있는 `main` 함수와 함께 이 코드를 실행하면, 원하는 출력을 얻을 수 있을 것입니다. 메소드는 `self` 파라미터 뒤에 추가된 여러 개의 파라미터를 가질 수 있으며, 이 파라미터들은 함수에서의 파라미터와 동일하게 기능합니다.

연관 함수

`impl` 블록의 또다른 유용한 기능은 `self` 파라미터를 *갖지 않는* 함수도 `impl` 내에 정의하는 것이 허용된다는 점입니다. 이를 *연관 함수 (associated functions)* 라고 부르는데, 그 이유는 이 함수가 해당 구조체와 연관되어 있기 때문입니다. 이들은 메소드가 아니라 여전히 함수인데, 이는 함께 동작할 구조체의 인스턴스를 가지고 있지 않아서 그렇습니다. 여러분은 이미 `String::from` 연관 함수를 사용해본 적이 있습니다.

연관 함수는 새로운 구조체의 인스턴스를 반환해주는 생성자로서 자주 사용됩니다. 예를 들면, 하나의 차원값 파라미터를 받아서 이를 길이와 너비 양쪽에 사용하여, 정사각형 `Rectangle`을 생성할 때 같은 값을 두번 명시하도록 하는 것보다 쉽게 해주는 연관 함수를 제공할 수 있습니다:

Filename: src/main.rs

```
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle { length: size, width: size }
    }
}
```

이 연관 함수를 호출하기 위해서는 `let sq = Rectangle::square(3);` 처럼, 구조체 이름과 함께 `::` 문법을 이용합니다. 이 함수는 구조체의 이름공간 내에 있습니다: `::` 문법은 연관 함수와 모듈에 의해 생성된 이름공간 두 곳 모두에서 사용되는데, 모듈에 대해서는 7장에서 다룰 것입니다.

정리

구조체는 우리의 문제 영역에 대해 의미있는 커스텀 타입을 만들수 있도록 해줍니다. 구조체를 이용함으로써, 우리는 연관된 데이터의 조각들을 서로 연결하여 유지할 수 있으며 각 데이터 조각에

이름을 붙여 코드를 더 명확하게 만들어 줄 수 있습니다. 메소드는 우리 구조체의 인스턴스가 가지고 있는 동작을 명시하도록 해주며, 연관 함수는 이용 가능한 인스턴스 없이 우리의 구조체에 특정 기능을 이름공간 내에 넣을 수 있도록 해줍니다.

하지만 구조체가 커스텀 타입을 생성할 수 있는 유일한 방법은 아닙니다: 러스트의 열거형 기능으로 고개를 돌려 우리의 도구상자에 또다른 도구를 추가하도록 합니다.