객체지향 요소들

2020.4

객체지향 이란?

- 개념에 대해서 의견이 일치되지 않음

객체-지향 프로그램은 객체로 구성된다. 객체는 데이터 및 이 데이터를 활용하는 프로시저를 묶는다. 이 프로시저들은 보통 메소드 혹은 연산 *(operation)* 으로 불린다.

- 이 정의에 따르면 러스트는 객체 지향적임.
- 구조체와 열거형은 데이터와 구현(메소드)를 가짐.

캡슐화

- 객체를 이용하는 외부 코드가 객체 내부에 접근하지 못하도록 하는 특성.
- 객체 내부에 접근하기 위해 API를 제공함.

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

```
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;
    }
}
```

상속

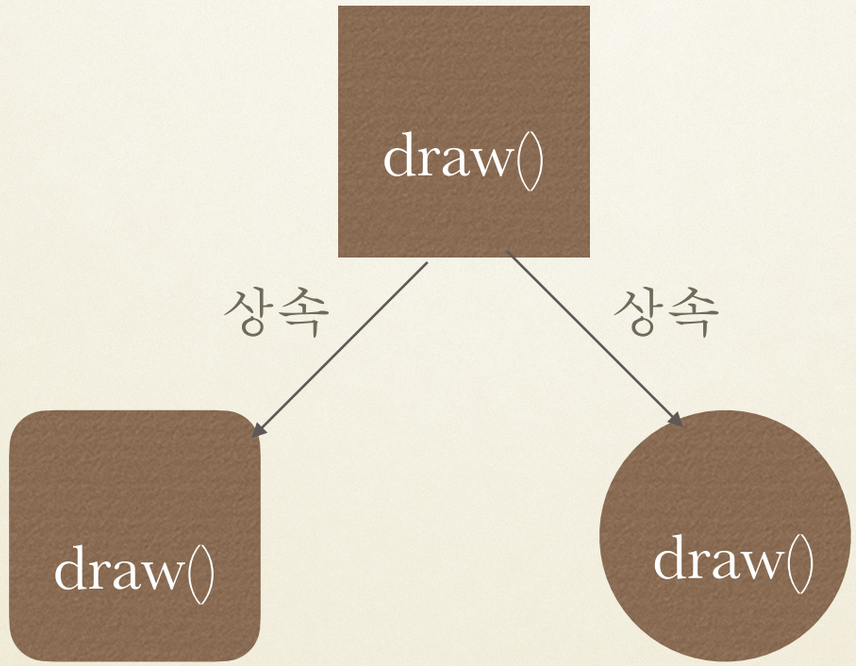
- 상속의 목적 : 코드의 재사용 - 기본 trait구현으로.
- 자식의 타입을 부모의 타입으로 사용하기 원할때
  - trait object를 사용함.
- 상속의 단점 - 코드 공유의 위험, 유연성 저하

Trait object

- 열거형은 몇가지 타입을 컴파일 타임에 알 수 있을 때 좋은 방법이다.
- 코드 작성 당시에 포함되지 않은 타입을 추가하기 위해 trait object를 사용한다.

트레잇 객체에 데이터를 추가 할 수 없다는 점에서 전통적인 객체들과 다릅니다

상속이 있는 경우



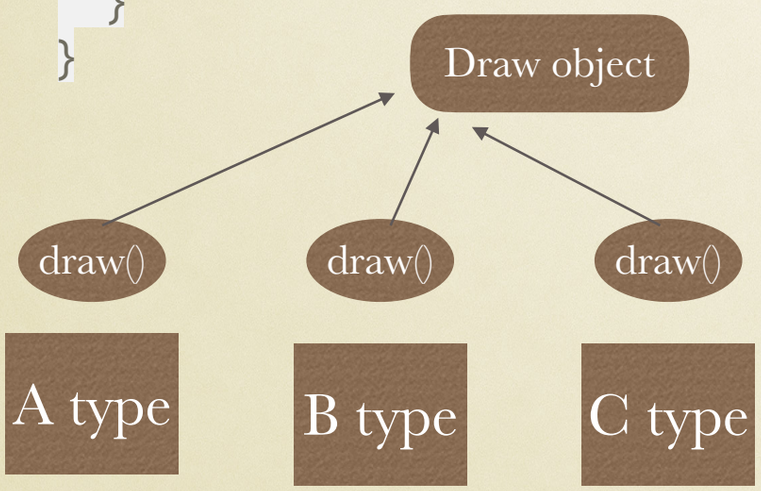
러스트는 상속이 없으므로 다른 방식으로 구현한다.

Draw trait

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

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

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




```
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();
        }
    }
}
```

제네릭 타입 파라미터는 한 번에 하나의 구체 타입으로만 대입될 수 있는 반면, 트레잇 객체를 사용하면 런타임에 여러 구체 타입을 트레잇 객체에 대해 채워넣을 수 있습니다.

하나의 Screen 인스턴스가 Box<Button> 혹은 Box<TextField>도 담을 수 있는 Vec<T>를 보유할 수 있습니다.

Trait 구현

```
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
    }
}
```

```
extern crate gui;
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
    }
}
```

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

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();
}
```

에러 검출

```rust
extern crate gui;
use gui::Screen;

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

    screen.run();
}
```

```
error[E0277]: the trait bound `std::string::String: gui::Draw` is not satisfied
  --> src/main.rs:7:13
   |
 7 |             Box::new(String::from("Hi")),
   |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait gui::Draw is not
    implemented for `std::string::String`
   |
   = note: required for the cast to the object type `gui::Draw`
```

Static & dynamic dispatch

```
struct Circle;
struct Triangle;

trait Figure {
    fn print(&self);
}

impl Figure for Circle {
    fn print(&self) {
        println!("Circle");
    }
}

impl Figure for Triangle {
    fn print(&self) {
        println!("Triangle");
    }
}

// static dispatch with trait bounds
fn log<T: Figure>(figure: &T) {
    figure.print();
}

// dynamic dispatch: function takes a trait object
fn logd(figure: &Figure) {
    figure.print();
}
```

```
fn main() {
// static dispatch
    let circle   = Circle;
    let triangle = Triangle;

    log(&circle);
    log(&triangle);

// dynamic dispatch:
    let mut figures: Vec*Box*Figure** = Vec::new();
    figures.push(Box::new(Circle));
    figures.push(Box::new(Triangle));

    // the precise type of figure can only be known at runtime:
    for figure in figures {
        logd(&*figure);
    }
}
```

객체 안전성

- 반환값의 타입이 Self가 아닙니다.
- 제네릭 타입 매개변수가 없습니다.

- 객체의 소비(사용)으로 원 객체의 타입을 잊으면 Self나 제네릭 타입을 채울 타입을 알 수 없으므로...

- 안전하지 않은 객체의 예 :

```
pub trait Clone {
    fn clone(&self) -> Self;
}

pub struct Screen {
    pub components: Vec<Box<Clone>>,
}

error[E0038]: the trait `std::clone::Clone` cannot be made into an object
 --> src/lib.rs:2:5
  |
2 |     pub components: Vec<Box<Clone>>,
  |            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `std::clone::Clone` cannot be
made into an object
  |
  = note: the trait cannot require that `Self : Sized`
```

Q & A