5주차 쌍쌍바 모둠활동 교육컨텐츠

Protocol Oriented Programming in Swift

Protocol 이란 (기본 개념)

프로토콜은 인터페이스입니다. 최소한으로 가져야 할 프로퍼티나 메서드를 정의합니다. 특정 클래스, 구조체, ENUM 을 만들 때 프로토콜을 적용하여 그 요구사항의 실제 구현을 제공 합니다. 그 결과 프로토콜의 요구사항 대로 메소드나 프로퍼티를 직접 구현해야 합니다. 어떤 타입이 특정 프로토콜의 요구사항을 만족시키면, 그 타입이 프로토콜을 따른다고 말합니다.
필요에 따라 프로토콜은 확장(Extension)을 통해 그것을 따르는 타입에 기능을 제공할 수 있습니다.

- 프로퍼티 제약 사항

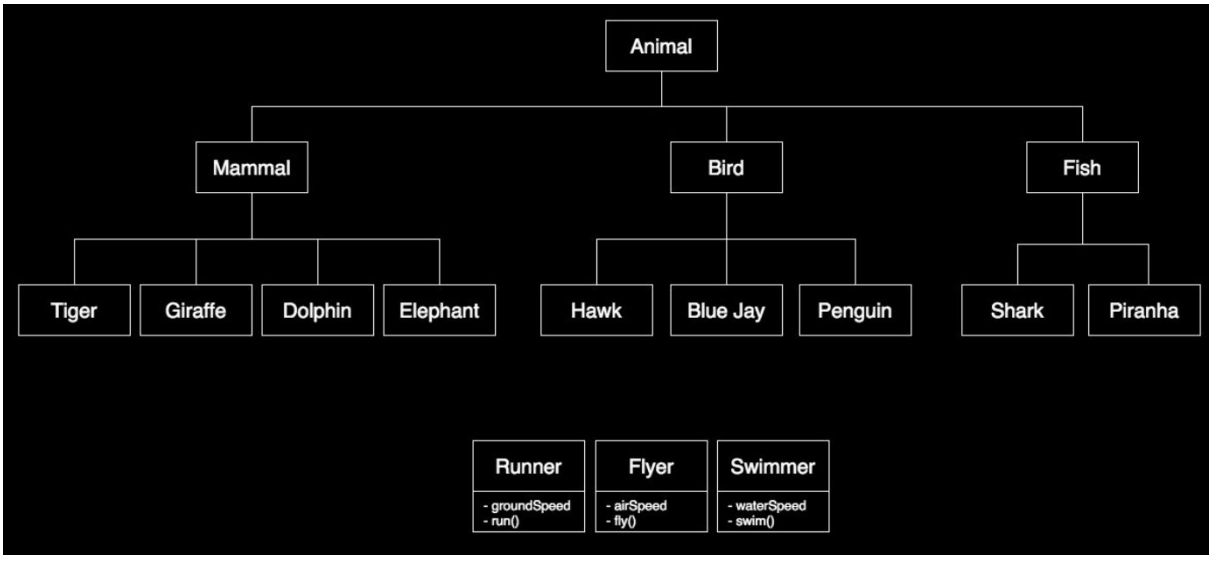
- 프로퍼티가 stored인지 computed인지는 명시하지 않는다.
- 프로퍼티의 이름과 타입을 명시해야 한다.
- 프로퍼티가 읽기/쓰기 속성인지 읽기 전용인지 명시해야 한다.
- 프로토콜이 읽기/쓰기용 프로퍼티를 요구한다면 -> 읽기/쓰기가 가능한 프로퍼티로만 요구사항을 만족시킬 수 있다. (읽기전용 프로퍼티로는 안 된다는 말)
- 프로토콜이 읽기전용 프로퍼티를 요구한다면 -> 어떤 종류의 프로퍼티라도 요구사항을 만족시킬 수 있다. (읽기/쓰기용 프로퍼티도 된다는 말)
- 프로퍼티 요구사항은 항상 변수로 선언되어야 한다. (var 키워드)
- 타입 프로퍼티 요구사항 앞에는 static 키워드를 붙인다. 클래스가 이 프로토콜을 따를 때에는 그 타입 프로퍼티 앞에 class 또는 static 키워드를 붙이도록 한다.

- 메서드 제약 사항

- 메서드 요구사항은 프로토콜 정의부에서 일반적인 메서드와 같은 방식으로 작성하면 된다. 단, 대괄호와 메서드 본문은 뺀다.
- 가변 파라미터Variadic parameters도 허용된다.
- 파라미터의 디폴트 값을 명시할 수 없다.

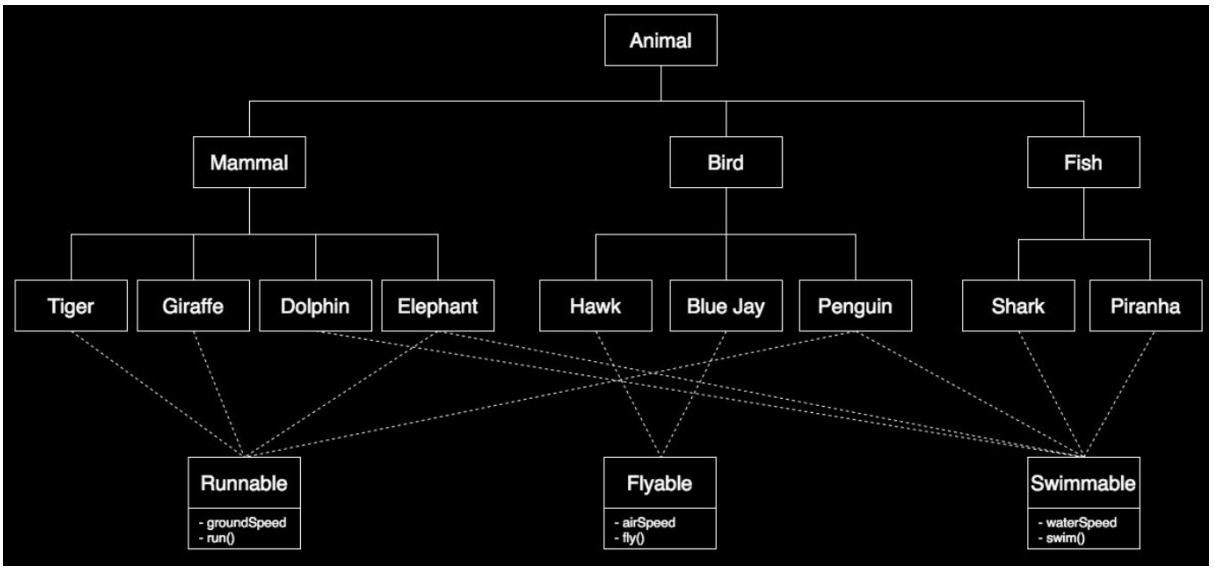
- 타입 메서드 요구사항 앞에는 static 키워드를 붙인다. 클래스가 이 프로토콜을 따를 때에는 그 타입 메서드 앞에 class 또는 static 키워드를 붙이도록 한다.
- 메서드 안에서 자신(self)를 수정할 경우 Mutating 메서드는 오직 값 타입 (구조체, ENUM)에만 적용이 가능하다.

상속과 비교



위와 같은 상속관계를 가진 클래스가 있다. 여기에 달리는 기능, 날 수 있는 기능, 헤엄칠 수 기능을 가진 래스를 따로 정의한다고 한다. 그러면 문제가 발생한다. A라는 클래스는 두가지 이상의 기능을 가질 수 있다. 또는 Bird라는 클래스가 만약 Flyer 클래스를 상속받는다면 Penguin은 Bird를 상속받지만 날지는 못하므로 설계상 어긋나게 된다.

다시말해, 객체지향의 문제점은 그들이 가지고 다니는 모든 암묵적인 환경을 가지고 있다는 것이다. 자신은 바나나를 원했지만 바나나와 전체 정글을 부모로 들고있는 고릴라를 얻게 되는 상황이 올 수도 있다는 말이다.



그래서 위와같이 상속과 별개인 프로토콜을 생성하여 프로토콜을 따르게 만들어주면 각각의 목적에 맞는 클래스로 활용할 수 있게 된다.

위를 통해 알 수 있듯이, 프로토콜은 상속과 다른점이 몇가지 있다.

기능의 모듈화
필요한 프로토콜에 대해서만 구현을 받음으로써 원하는 기능만을 갖춘 모듈을 만들어낼 수 있다.
상속의 경우 하나의 상속체계에서 벗어난 기능을 사용하려면 코드 중복을 피할 수 없게된다.
하지만 프로토콜을 사용하면 이런 상속의 한계에서 탈피할 수 있다. 원한다면 초기구현도 해놓을 수 있으니 이보다 좋을 수 없다.

클래스 타입이외에도 구현 가능
상속은 구조화된 타입(구조체, 클래스, 열거형) 중에서도 클래스만 상속가능하지만
프로토콜은 그렇지 않다.

Traits and Mixin

일반적으로 프로토콜은 다음과 같이 사용된다.

```
protocol Hello {
  func sayHello() -> String
}

class NoisClass: Hello {
  var name = "nois"
```

```
  func sayHello() -> String {
    return "Hello my name is \(name)"
  }
}
```

NoisClass가 Hello를 따르면, NoisClass에서는 반드시 sayHello 함수를 구현해야한다. (그렇지 않으면 컴파일 에러가 발생한다.)

```
let nois = NoisClass()
nois.sayHello() // "Hello my name is nois"
```

실제로 코드는 잘 동작한다. 그런데, Hello를 따르는 GamjaClass가 sayHello에 대해서 NoisClass와 같은 동작을 해야할 경우를 생각해 보자.

```
class GamjaClass: Hello {
  var name = "gamja"

  func sayHello() -> String {
    return "Hello my name is \(name)"
  }
}
```

이 경우에도 GamjaClass는 sayHello를 반드시 구현해야 한다. 그런데 NoisClass와, GamjaClass의 sayHello는 완벽히 같은 동작을 수행한다. 이 경우, sayHello()의 구현부가 중복된다. 물론 위의 경우에는 불편함을 감수하고 두 번 구현하는 것이 힘들지 않을 수 있지만, 중복된 동작을 구현하는 클래스가 많아지고 구현부가 길어질 경우 이는 비효율적이다.

Swift에서는 이를 해결하기 위해서 protocol의 extension을 지원한다. 아래의 코드를 보자.

```
protocol Hello {
  var name: String { get set }
  func sayHello() -> String
}

extension Hello {
  func sayHello() -> String {
    return "Hello my name is \(name)"
  }
}

class NoisClass: Hello {
  var name = "nois"
}

class GamjaClass: Hello {
  var name = "gamja"
}
```

```
class BluClass: Hello {
  var name = "Blu"

  func sayHello() -> String {
    return "Welcome!"
  }
}

let nois = NoisClass()
nois.sayHello() // "Hello my name is nois"

let gamja = GamjaClass()
gamja.sayHello() // "Hello my name is gamja"

let blu = BluClass()
blu.sayHello() // "Welcome!"
```

Hello protocol과 함께, extension을 사용하고 있다. extension에서는 프로토콜에서 정의한 함수의 기본 정의를 할 수 있다. 만약 BluClass처럼, 재정의가 필요한 경우, 해당 클래스에서 구현부를 수정하면 된다.

적용사례 : 17장 과제에 대해 Protocol 및 Traits 적용

1. Strokable.swift

```
import UIKit
protocol Strokable {
   var color: UIColor { get set }                  /// 원 혹은 직선의 색상

   init(locations: [CGPoint], color: UIColor)      /// 터치 위치 정보만을 이용해 초기화

   func stroke(thick: CGFloat)                     /// 화면에 실제로 그림 그리기
   func move(locations: [CGPoint])                 /// 손가락 위치 변경 시 그림 모양 변경
   func finish(locations: [CGPoint]) -> Strokable  /// 손가락을 화면에서 뗄 경우 그림 모양
확정
}
/// Traits 예제
extension Strokable {
   /// 그림에 알파 값을 조절 해서 적용하는 것은 원이든, 직선이든 상관없음
   func apply(alpha: CGFloat) -> UIColor {

       /// 적당한 예제가 없어서..., 핵심은 Protocol에 정의된 변수로 기능이 공통인 것들을 Traits로
구현
       return color.withAlphaComponent(alpha)
   }
}
```

2. StrokableFactory.swift

```
import UIKit
enum StrokableType {
   case circle
   case line
}
class StrokableFactory {

   /// 터치된 손가락의 개수에 따라 결정된 타입의 객체를 생성해 반환
   static func makeSegments(type: StrokableType, with locations:[CGPoint]) -> Strokable {

       switch type {
           case .line:
               return Line(locations: locations, color: UIColor.black)
           case .circle:
               return Circle(locations: locations, color: UIColor.red)
       }
   }

}
```

3. Line.swift

```
import UIKit
struct LineSegment {
   var begin: CGPoint
   var end: CGPoint
   var color: UIColor
}
class Line : Strokable{

   var segments:[LineSegment] = []
   var color: UIColor

   required init(locations: [CGPoint], color: UIColor) {

       for location in locations {
           self.segments.append(LineSegment(begin: location, end: location, color:
UIColor.black))
       }
       self.color = color
   }

   /// 실제로 DrawView에서 그림을 화면에 그릴 때 실행 됨
   func stroke(thick: CGFloat) {

       for segment in segments {
           segment.color.setStroke()
```

```
            let path = UIBezierPath()

            path.lineWidth = thick
            path.lineCapStyle = .round

            path.move(to: segment.begin)
            path.addLine(to: segment.end)

            path.stroke()
        }
    }

    /// touchesMoved에서 손가락을 움직여 그림의 모양을 변경
    func move(locations: [CGPoint]) {

        for (idx, location) in locations.enumerated() {

            if idx < segments.count {
                segments[idx].end = location

                /// Strokable Protocol Extension의 Traits로 구현된 apply
                segments[idx].color = apply(alpha: CGFloat(arc4random_uniform(100)) / 100)
            } else {
                break
            }
        }
    }

    /// touchesEnded에서 화면에서 손가락을 떼서 그림의 모양을 확정
    func finish(locations: [CGPoint]) -> Strokable {

        for (idx, location) in locations.enumerated() {

            if idx < segments.count {
                segments[idx].end = location

                /// Strokable Protocol Extension의 Traits로 구현된 apply
                segments[idx].color = apply(alpha: CGFloat(arc4random_uniform(100)) / 100)
            } else {
                break
            }
        }

        return self
    }
}
```

4. Circle.swift

```
import UIKit
struct CircleSegment {
    var center: CGPoint
```

```
    var radius: CGFloat
    var color: UIColor
}
class Circle : Strokable {

    var segment: CircleSegment
    var color: UIColor

    required init(locations: [CGPoint], color: UIColor) {

        var center: CGPoint = CGPoint()
        center.x = (locations[0].x + locations[1].x) / 2
        center.y = (locations[0].y + locations[1].y) / 2

        let dx: CGFloat = locations[1].x - locations[0].x
        let dy: CGFloat = locations[1].y - locations[0].y

        let radius = sqrt(dx * dx + dy * dy) / 2

        self.segment = CircleSegment(center: center, radius: radius, color: UIColor.black)
        self.color = color
    }

    /// 실제로 DrawView에서 그림을 화면에 그릴 때 실행 됨
    func stroke(thick: CGFloat) {
        segment.color.setStroke()

        let circlePath = UIBezierPath(
            arcCenter: segment.center,
            radius: segment.radius,
            startAngle: CGFloat(0),
            endAngle:CGFloat(Double.pi * 2),
            clockwise: true)

        circlePath.lineWidth = thick

        circlePath.stroke()
    }

    /// touchesMoved에서 손가락을 움직여 그림의 모양을 변경
    func move(locations: [CGPoint]) {

        if locations.count > 1 {
            var center: CGPoint = CGPoint()
            center.x = (locations[0].x + locations[1].x) / 2
            center.y = (locations[0].y + locations[1].y) / 2

            let dx: CGFloat = locations[1].x - locations[0].x
            let dy: CGFloat = locations[1].y - locations[0].y

            let radius = sqrt(dx * dx + dy * dy) / 2

            /// Strokable Protocol Extension의 Traits로 구현된 apply
            let colorWithAlpha = apply(alpha: dx / radius)
```

```
            segment = CircleSegment(center: center, radius: radius, color: colorWithAlpha)
        }
    }

    /// touchesEnded에서 화면에서 손가락을 떼서 그림의 모양을 확정
    func finish(locations: [CGPoint]) -> Strokable {

        if locations.count > 1 {
            var center: CGPoint = CGPoint()
            center.x = (locations[0].x + locations[1].x) / 2
            center.y = (locations[0].y + locations[1].y) / 2

            let dx: CGFloat = locations[1].x - locations[0].x
            let dy: CGFloat = locations[1].y - locations[0].y

            let radius = sqrt(dx * dx + dy * dy) / 2

            /// Strokable Protocol Extension의 Traits로 구현된 apply
            let colorWithAlpha = apply(alpha: dx / radius)

            segment = CircleSegment(center: center, radius: radius, color: colorWithAlpha)
        }

        return self
    }
}
```

5. DrawView.swift

```
import UIKit
class DrawView: UIView {

    var currentSegment: Strokable?          /// 현재 그려지고 있는 그림
    var drawedSegments: [Strokable] = []    /// 이전에 완성되어 저장된 그림

    /// setNeedsDisplay가 호출되면, 실행 됨
    override func draw(_ rect: CGRect) {

        for segment in drawedSegments {
            segment.stroke(thick: 1.0)
        }

        currentSegment?.stroke(thick: 1.0)  /// 직선이든 원이든 상관없이 stroke가 가능하므로
실행
    }

    /// 손가락으로 그림을 그리기 시작
    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {

        var locations: [CGPoint] = []
        for touch in touches {
```

```
            locations.append(touch.location(in: self))
        }

        /// 터치된 손가락의 개수에 따라 직선인지, 원인지가 결정 되고, Factory에서 알맞은 객체를 만들어 반환
        let type: StrokableType = (touches.count == 2) ? .circle : .line
        currentSegment = StrokableFactory.makeSegments(type: type, with: locations)

        setNeedsDisplay()
    }

    /// 손가락을 움직이는 경우
    override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {

        var locations: [CGPoint] = []
        for touch in touches {
            locations.append(touch.location(in: self))
        }

        currentSegment?.move(locations: locations) /// 직선이든 원이든 상관없이 move가 가능하므로 실행

        setNeedsDisplay()
    }

    /// 손가락을 뗌
    override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {

        var locations: [CGPoint] = []
        for touch in touches {
            locations.append(touch.location(in: self))
        }

        /// 직선이든 원이든 상관없이 finish가 가능하므로 실행
        guard let completeSegment = currentSegment?.finish(locations: locations) else {
            return
        }

        drawedSegments.append(completeSegment)

        setNeedsDisplay()
    }

    /// 그림 그리기가 취소 되는 경우(Ex. 전화가 오는 경우)
    override func touchesCancelled(_ touches: Set<UITouch>, with event: UIEvent?) {

        currentSegment = nil

        setNeedsDisplay()
    }
}
```