알쓸!신잡

지난 한달 돌아보기

한달간 배운것들

- Git (add, commit, push)
- 컴퓨터 개론 (Stack,Queue,진법연산,메모리 구조 …)
- 알고리즘 (윤년구하기, 소수구하기, 하샤드 수, 최대공약수, 최소공배수)
- Type Annotation & Inference
- Function,Closure
- Class, Struct
- 조건문(if, guard, switch, 삼항연산자)
- 반복문(for, while, repeat-while)
- Collection(array, dictoinary, set…)
- Enum
- Optional(Optional Binding, Force Unwrapping,Optional Chaining,nil)
- OOP(Access Control)
- Getter & Setter
- Type Casting & Check
- App Life Cycle, View Life Cycle
- Frame,Bounds
- Story Board, programmatically
- UIKit (UIViewController, UIview,UIlabel…)
- Singleton,UserDefaults
- ARC
- Delegate
- Coding Convension

한달간 배운것들

- Git (add, commit, push)
- 컴퓨터 개론 (Stack,Queue,진법연산,메모리 구조 …)
- **알고리즘 (윤년구하기, 소수구하기, 하샤드 수, 최대공약수, 최소공배수)**
- Type Annotation & Inference
- Function,Closure
- Class, Struct
- **조건문(if, guard, switch, 삼항연산자)**
- 반복문(for, while, repeat-while)
- Collection(array, dictoinary, set…)
- **Enum**
- **Optional(Optional Binding, Force Unwrapping,Optional Chaining,nil)**
- OOP(Access Control)
- **Getter & Setter**
- **Type Casting & Check**
- App Life Cycle, View Life Cycle
- Frame,Bounds
- Story Board, programmatically
- UIKit (UIViewController, UIview,UIlabel…)
- Singleton,UserDefaults
- ARC
- Delegate
- **Coding Convension**

ALGORITHM

자연수를 입력받아 그 숫자보다 작거나 같은 모든 소수와 그 개수를 출력하는 함수를 정의

소수가 무엇인지 찾아보기

1. 구글에 “소수란” 이라고 검색한다.
2. 위키백과를 살펴본다.

소수는 1과 자기자신의 수 이외에 나눌 수 없는 자연수

Ex) 4가 들어오면 2,3으로 나누어본다.

나누어 떨어진다면 소수가 아니다.

자연수를 입력받아 그 숫자보다 작거나 같은 모든 소수와 그 갯수를 출력하는 함수를 정의

문제 쪼개기

1. **소수를 구한다.**
2. 그 소수의 갯수를 구한다.

```
func checkPrimeNumbers(number: Int)
```

> Input : 10

> Output : 10보다 작거나 같은 소수는 [2, 3, 5 ,7]이고 총 4개입니다.

자연수를 입력받아 그 숫자보다 작거나 같은 모든 소수와 그 갯수를 출력하는 함수를 정의

소수를 구한다.

소수는 1과 자기자신 외에 나누어 떨어질 수 없는 수

입력받은 수 전까지 반복하여 나누어 본다.

한번이라도 나누어 떨어진다면 소수가 아님

나누어 떨어진적이 없다면 그것은 소수

```
func isPrime(number: Int) -> Bool
{
    for i in 2..<number {
        if number % i == 0 {
            return false
        }
        return true
}
```

자연수를 입력받아 그 숫자보다 작거나 같은 모든 소수와 그 갯수를 출력하는 함수를 정의

문제 쪼개기

1. ~~소수를 구한다.~~

2. **그 소수의 갯수를 구한다.**

```
func checkPrimeNumbers(number: Int)
```

> Input : 10

> Output : 10보다 작거나 같은 소수는 [2, 3, 5 ,7]이고 총 4개입니다.

자연수를 입력받아 그 숫자보다 작거나 같은 모든 소수와 그 갯수를 출력하는 함수를 정의

소수의 갯수를 구한다.

1. 입력받은 수 까지 반복하여 반복되는 수를 isPrime함수에 넣는다. -> 조건에 따라 true, false가 리턴 됨.
2. 조건을 충족한 경우 primeArray에 추가한다.
3. 반복을 끝낸 후 결과를 출력한다.

```
func checkPrimeNumbers(number: Int) {

    var primeArray: [Int] = []

    for i in 2..<number {

        if isPrime(i) {

            primeArray.append(i)

        }

    }

    print("\(number)보다 작거나 같은 소수는 \
(primeArray)이고 총 \(primeArray.count)개
입니다.")

}
```

CONDITIONAL STATEMENT

IF

IF문

- if
  - 기본 문법
    - if 조건절 {
      // 조건 성립시 실행할 구문
      }

- if-else
  - 기본 문법
    - if 조건절 {
      // 조건 성립시 실행할 구문
      } else {
      // 조건 미성립시 실행할 구문
      }

- if
  - 기본 문법
    - if 조건절 {
      // 조건 성립시 실행할 구문
      }

- if-else if
  - 기본 문법
    - if 조건절 {
      // 조건 성립시 실행할 구문
      } else if 조건절 {
      // 조건 미성립시 실행할 구문
      }

GUARD

GUARD문

- guard
  - 기본 문법
    - guard 조건절 else { return,throw.. }
      // 조건 성립시 실행할 구문

SWITCH-CASE

SWITCH-CASE문

- Switch
  - 기본 문법
    - switch 체크 값 {
      case 경우 값:
      // 체크값 == 경우 값 인 경우 실행
      case 경우 값2:
      // 체크값 == 경우 값2 인 경우 실행
      default:
      // 경우 값에 안걸린 경우
      }

TERNARY OPERATOR

- 삼항연산자
  - 기본 문법
    - 조건절 ? 참(true)일때 실행 : 거짓(false)일때 실행
  - 예시 (홀수짝수 판별)

```
func oddEven(_ input: Int) -> Bool {
    input % 2 == 0 ? true : false
}
oddEven(5) // false
```

CONDITIONAL ADVANCED

어떠한 차이점이 존재할까요?

- if
  - 기본 문법
    - if 조건절 {
      // 조건 성립시 실행할 구문
      }

      if 조건절 {
      // 조건 성립시 실행할 구문
      }

- if-else if
  - 기본 문법
    - if 조건절 {
      // 조건 성립시 실행할 구문
      } else if 조건절 {
      // 조건 미성립시 실행할 구문
      }

어떠한 차이점이 존재할까요?

- guard
  - 기본 문법
    - guard 조건절 else
      { return,throw.. }
      // 조건 성립시 실행할 구문

- if
  - 기본 문법
    - if 조건절 {
      // 조건 성립시 실행할 구문
      }

출력 결과는?

- Switch
  - Value Binding, where, fallthrough
    - let someTuple = (1, 90)

```
switch someTuple {
    case let (x, y) where y == 90:
        print("x: \(x), y: \(y)")
        fallthrough
    case (1, let x):
        print("x: \(x)")
    default:
        print("default")
}
```

ENUM

임의의 관계를 맺는 값들을 하나의 타입으로 묶어서 사용

- enum
  - 기본문법
  - ```
    enum Site: String {
        case Google
        case Kakao
        case Naver
    }
    ```
    ```

임의의 관계를 맺는 값들을 하나의 타입으로 묶어서 사용

- enum
  - 기본문법 - rawValue
  - ```
    enum Site: String {
        case Google
        case Kakao
        case Naver
    }
    print(Site.Google.rawValue) // “Google“
    ```
    ```

연관값

- enum
  - 기본문법 - 연관값
  - ```
    enum Site {
        case Google(id: String, password: String)
        case Kakao(id: String, password: String)
        case Naver(id: String, password: String)
    }
    print(Site.Google(“hong3", “12345678”)) //
    Google("hong3", "12345678")
    ```
    ```

연관값

- enum
  - 기본문법 - 내부함수
  - enum Site {
    case Google(id: String, password: String)
    case Kakao(id: String, password: String)
    case Naver(id: String, password: String)

    func signIn() {
    // 구현부분
    }
    }
    print(Site.Google("hong3", "12345678"))

OPTIONAL

SWIFT에서 데이터의 부재를 표현하기 위한 방식

- Optional
  - 기본문법
  - ```
    var someVariable: Int?
    print(someVariable) // nil
    someVariable = 1
    print(someVariable) // Optional(1)
    ```
    ```

옵셔널
왜쓰나요?

왜쓰는걸까요?

- 변수에 값이 담길때 nil이 담길가능성을 Optional로 표현함으로써, 추가적인 문서,자료 또는 상황을 설명하지 않아도 코드만으로 데이터 부재의 가능성을 표현할 수 있다.
  -> 문서/주석 작성시간 절약

- 값이 Optional이 아니라면 무조건 데이터가 있다.
  -> 효율적인 코딩, 예외처리에 안전하다.

왜쓰는걸까요?

- 변수에 값이 담길때 nil이 담길가능성을 Optional로 표현함으로써, 추가적인 문서,자료 또는 상황을 설명하지 않아도 코드만으로 데이터 부재의 가능성을 표현할 수 있다.
  -> 문서/주석 작성시간 절약

- 값이 Optional이 아니라면 무조건 데이터가 있다.
  -> 효율적인 코딩, 예외처리에 안전하다.

OPTIONAL UNWRAPPING

옵셔널 타입에서 값을 추출해내는 행위

- Optional Unwrapping
  - 1. Optional Binding
  - 2. Force Unwrapping

옵셔널 바인딩

- Optional Binding
  - nil체크 + 안전한 값 추출이 가능
- var someVariable: Int?
  someVariable = 1
  - 1. if-let
    ```
    if let x = someVariable {
        print(x) // 1
    }
    ```
  - 2. guard-let
    ```
    guard let x = someVariable else
    { return OR throw 등등 }
    print(x) // 1
    ```
- let이 아니라 var도 사용이 가능하다.

- var someVariable: Int?
  someVariable = 1
  - 3. Nil-Coalescing Operator
    someVariable ?? 0
  - 4. Implicitly Unwrapped Optional
    someVariable!


CAUTION

암시적 추출 옵셔널

- var someVariable: Int?
  someVariable!

- 위와 같은 Force Unwrapping을 진행시 해당변수에 값이 없다면 (nil이 있다면)
  런타임 에러가 발생

- 런타임 에러 발생 -> Application Crash -> 사용자 불만 -> 혼남

OPTIONAL CHAINING

하위 프로퍼티의 값들을 연속적으로 살펴보는 방식

```
Class Person {
    var checkCard: CheckCard?
}
Class CheckCard {
    var balance: Money?
    var sign: Bool = true
}
Class Money {
    var won = 123
}
```

```
let hong3 = Person()

print(hong3.checkCard?.balance?.won) // nil

let S20 = CheckCard()

hong3.checkCard = S20

print(hong3.checkCard?.sign) // Optional(true)
```

GETTER
&
SETTER

프로퍼티

프로퍼티의 종류

- Stored Property
- Computed Property
- Property Observers

저장,연산 프로퍼티

- ```
  var _balance = 0 // 저장 프로퍼티
  var balance: Int // 연산 프로퍼티
      get {
          return _balance
      }
      set (원하는 변수명) {
          _balance = newValue // 기본값은 newValue
      }
  }
  ```

- 연산 프로퍼티의 경우 set을 쓰지 않는다면 get은 암시적 무시가 가능
  ```

프로퍼티 옵저버

```
var balance: Int = 0 // 저장 프로퍼티
    didSet (원하는 변수명) {
        print("balance를 \(oldValue)로 설정합니다.")
    }
    WillSet (원하는 변수명) {
        print("balance를 \(newValue)로 설정합니다.")
    }
}
```

TYPE CASTING
&
TYPE CHECK

타입 체크

- type(of:) 내장함수를 사용
  - 기본문법

    ```
    var someArray: [Int]
    type(of: someArray)
    ```

  - is를 사용하여 비교가 가능

    ```
    type(of: someArray) is Array<Int>.Type // true
    ```

형 변환

- 인스턴스의 타입을 검사하는 방법
- Swift는 암시적 형변환을 지원하지 않는다.
- 다운캐스팅 (as?, as!)
  부모클래스에서 파생된 각종 서브 클래스로의 타입 변환
- 업캐스팅 (as)
  자식클래스의 타입을 부모클래스의 타입으로 변환

형 변환

- as : 타입변환이 확실하게 가능한 경우 사용, 실패시 컴파일 에러
- as?: 강제 타입 변환 시도, 성공시 Optional값, 실패시 nil
- as!: 강제 타입 변환 시도, 성공시 Unwrapping 값, 실패시 런타임 에러

형 변환

```
class Shape {

}

class Rectangle: Shape {

}

class Triangle: Shape {

}
```

```
let shapeTriangle: Shape
= Triangle()
```

```
shapeTriangle as?
Triangle //
Optional(Triangle)
```

```
shapeTriangle as?
Rectangle // nil
```

CODING CONVENSIONS

CODING RULE

- 코드 들여쓰기는 스페이스 2칸 or 4칸
- self는 필요할때만 사용한다.
- 빈배열,딕셔너리 선언시 타입추론을 사용
- Array<String> 보다 [String]
- if문이 중첩된다면 guard문을 고려해보자.
- 세미콜론을 쓰지말자

기호

- 소괄호 () - Parenthesis 퍼렌퍼시스
- 중괄호 {} - Brace 브레이스
- 대괄호 [] - Bracket - 브래킷
- 꺽쇠괄호 <> - Angle Bracket 앵글 브래킷
- 골뱅이 @ - At sign 앳
- 앤드기호 & - Ampersand 앰퍼샌드
- 빼기 - - Hyphen or Dash 하이픈, 대시
- 밑줄 _ - UnderScore,UnderLine 언더스코어, 언더라인

기호

- 별표 * - Asterisk 애스터리스크
- 물음표 ? - Question Mark 퀘스천 마크
- OR사인 | - Pipe 파이프
- 윗꺽쇠 ^ - Caret 캐럿
- 따옴표 ` - Grave 그레이브
- 큰따옴표 " - Quotation Mark 쿼테이션 마크
- 작은 따옴표 ' - Apostrophe 어퍼스트로피

검수 : 성현


:WQ