Swift 치트시트

기본: 상수, 변수, 타입

- 상수와 변수: `let` 키워드로 상수를, `var` 키워드로 변수를 선언합니다. Swift는 강력한 타입 추론 기능을 제공하여 초기값이 있을 경우 타입을 생략할 수 있습니다.
- 타입 명시 (Type Annotation): 변수나 상수의 타입을 직접 명시할 때 사용합니다. (예: `var welcomeMessage: String`)
- 기본 데이터 타입: `Int` (정수형), `Double`/`Float` (실수형), `Bool` (논리형), `String` (문자열), `Character` (단일 문자).
- 튜플 (Tuples): 여러 값을 하나의 복합 값으로 그룹화합니다. (예: `let http404Error = (404, "Not Found")`)
- 타입 별칭 (Type Alias): 기존 타입에 새로운 이름을 부여하여 코드의 가독성을 높입니다. (예: `typealias AudioSample = UInt16`)

옵셔널 (Optionals)

값이 존재하지 않을 수 있는 상황(`nil`)을 안전하고 명확하게 처리하기 위한 Swift의 핵심 기능입니다.

- 옵셔널 선언: `var optionalString: String? = "Hello"` (타입 뒤에 `?`를 붙입니다)
- 강제 언래핑 (Forced Unwrapping): `optionalString!`를 사용하여 값을 강제로 꺼냅니다. (주의: 값이 `nil`일 경우 런타임 에러가 발생하므로 사용에 주의해야 합니다)
- 옵셔널 바인딩 (Optional Binding): 안전하게 값을 확인하고 추출하는 권장 방식입니다.

```
if let constantName = someOptional {
    // someOptional에 유효한 값이 있을 때만 실행됩니다.
}
```

- 가드 구문 (Guard Statement): 조건이 충족되지 않으면 조기에 종료(return/throw)하여 코드의 가독성을 높입니다.

```
guard let constantName = someOptional else {
    // someOptional이 nil일 때 실행되며, 반드시 현재 스코프를 벗어나야 합니다.
    return
}
// 이후 스코프에서 constantName을 옵셔널이 아닌 타입으로 자유롭게 사용 가능합니다.
```

- Nil-Coalescing 연산자 (`??`): `a ?? b` 형식으로 사용하며, `a`가 `nil`이 아니면 언래핑된 값을, `nil`이면 기본값 `b`를 반환합니다.
- 옵셔널 체이닝 (Optional Chaining): 옵셔널의 속성이나 메서드에 접근할 때, 중간에 `nil`이 있으면 전체 결과로 `nil`을 반환하여 에러를 방지합니다.
- 암시적 언래핑 옵셔널 (Implicitly Unwrapped Optionals): `var assumedString: String!`와 같이 선언하며, 선언 후 즉시 값이 할당될 것이 확실한 상황에서 편의상 사용합니다.

컬렉션 타입

데이터를 그룹화하여 저장하기 위한 세 가지 기본 컬렉션 타입을 제공합니다.

- 배열 (Array): 순서가 있는 값의 리스트입니다. (예: `var shoppingList: [String] = ["Eggs", "Milk"]`)
- 집합 (Set): 순서가 없고 중복을 허용하지 않는 유일한 값들의 집합입니다. (예: `var favoriteGenres: Set<String> = ["Rock", "Classical"]`)
- 딕셔너리 (Dictionary): 키(Key)와 값(Value)의 쌍으로 이루어진 컬렉션입니다. (예: `var airports: [String: String] = ["YYZ": "Toronto Pearson"]`)

제어 흐름

- `for-in` 루프: 배열, 범위, 문자열 등 시퀀스의 각 항목을 순회합니다.
- `while` 루프: 조건이 거짓이 될 때까지 반복 실행합니다.
- `repeat-while` 루프: 구문을 먼저 실행한 후 조건을 확인합니다. (다른 언어의 `do-while`과 유사)
- `if-else` 조건문: 조건에 따라 코드 블록을 실행합니다.
- `switch` 선택문: Swift의 `switch`는 매우 강력하며, `break`를 명시하지 않아도 다음 케이스로 넘어가지 않습니다. 모든 가능한 경우를 처리해야 합니다.

```
switch someValue {
case 1:
    // 특정 값 매칭
case 2...5:
    // 범위 매칭
case let (x, 0):
    // 튜플 및 값 바인딩 활용
case let (x, y) where x == y:
    // where 절을 사용한 추가 조건 검사
default:
    // 일치하는 케이스가 없는 경우 (필수)
}
```

함수와 클로저

- 함수 정의: `func` 키워드를 사용하며 매개변수와 반환 타입을 명시합니다.
  - 인자 레이블 생략: `_`를 사용하여 호출 시 인자 이름을 생략할 수 있습니다.
- 가변 매개변수 (Variadic Parameters): 타입 뒤에 `...`을 붙여 0개 이상의 인자를 받을 수 있습니다.
- 입출력 매개변수 (In-Out Parameters): `inout` 키워드를 사용하여 함수 내부에서 인자의 값을 직접 수정할 수 있게 합니다.
- 함수 타입: 함수 자체를 변수에 저장하거나 다른 함수의 인자로 전달할 수 있습니다. (예: `(Int, Int) -> Int`)
- 클로저 (Closures): 코드에서 전달되거나 사용될 수 있는 이름 없는 독립적인 코드 블록입니다.
  - 문맥을 통한 타입 추론: 매개변수와 반환 타입을 생략하여 간결하게 작성할 수 있습니다.
  - 단축 인수 이름: `$0`, `$1` 등 시스템에서 제공하는 이름을 사용합니다.
  - 연산자 메서드: `names.sorted(by: >)`와 같이 연산자만으로 클로저를 대체할 수 있습니다.
- 후행 클로저 (Trailing Closure): 함수의 마지막 인자가 클로저일 때, 함수 호출 괄호 외부에 클로저 블록을 작성할 수 있습니다.
- 탈출 클로저 (`@escaping`): 함수가 종료된 이후에도 호출될 가능성이 있는 클로저에 명시합니다.

구조체와 클래스 (Structures and Classes)

- 공통점: 프로퍼티와 메서드 정의, 초기화 구문(`init`), 서브스크립트 접근 등을 지원합니다.
- 차이점:
  - 클래스: 상속이 가능하며, 타입 캐스팅, 소멸자(`deinit`), 참조 카운팅을 지원합니다.
  - 구조체: 상속이 불가능하며 간단한 데이터 묶음에 적합합니다.
  - 핵심 차이: 클래스는 **참조 타입(Reference Type)**이며, 구조체와 열거형은 **값 타입(Value Type)**입니다. 값 타입은 전달되거나 할당될 때 복사본이 생성됩니다.

프로퍼티와 메서드

- 저장 프로퍼티 (Stored Properties): 인스턴스의 일부로 값을 저장합니다.
- 연산 프로퍼티 (Computed Properties): 값을 직접 저장하지 않고, `get`과 `set`을 통해 값을 계산하여 반환하거나 설정합니다.
- 프로퍼티 옵저버 (Property Observers): 값이 변경될 때 반응합니다. (`willSet`, `didSet`)
- 타입 프로퍼티/메서드: `static` 키워드를 사용하여 인스턴스가 아닌 타입 자체에 속하는 프로퍼티와 메서드를 정의합니다.

프로토콜 (Protocols)

- 특정 작업이나 기능에 필요한 메서드, 프로퍼티 등의 요구사항을 정의하는 청사진입니다.
- 프로토콜 채택: 구조체나 클래스 이름 뒤에 `:`를 붙여 하나 이상의 프로토콜을 준수하도록 선언합니다.
- 프로토콜 합성 (Protocol Composition): 여러 프로토콜을 하나로 결합하여 요구사항을 표현합니다. (예: `First & Another`)
- 프로토콜 확장 (Protocol Extension): 프로토콜 자체를 확장하여 메서드나 연산 프로퍼티의 기본 구현을 제공할 수 있습니다.

에러 처리

- 에러 타입 정의: `Error` 프로토콜을 준수하는 열거형 등을 통해 에러 상황을 정의합니다.
- 에러 던지기: `throws` 키워드를 사용하여 에러가 발생할 수 있는 함수임을 나타냅니다.
- 에러 처리 방법:
  1. `do-catch`: 에러를 잡아내어 적절히 처리합니다.
  2. `try?`: 결과를 옵셔널로 변환하며, 에러 발생 시 `nil`을 반환합니다.
  3. `try!`: 에러가 절대 발생하지 않을 것이라 확신할 때 사용합니다. (에러 발생 시 앱 종료)
- `defer` 구문: 현재 코드 블록이 종료되기 직전에 반드시 실행되어야 하는 정리 작업을 수행합니다.

비동기 처리 (Concurrency)

- `async/await`: 비동기 코드를 마치 동기 코드처럼 순차적이고 직관적으로 작성하게 해줍니다.
- 비동기 시퀀스: `for await ... in` 루프를 사용하여 비동기적으로 생성되는 데이터 흐름을 순회합니다.
- Task: 비동기 작업 단위를 생성하고 관리하며 취소 기능을 지원합니다.

- Actor: 참조 타입이지만 한 번에 하나의 작업만 데이터에 접근할 수 있도록 보장하여 데이터 경쟁(Race Condition)을 방지합니다.

자동 참조 카운팅 (ARC)

Swift는 자동 참조 카운팅(ARC) 시스템을 통해 애플리케이션의 메모리 사용량을 자동으로 추적하고 효율적으로 관리합니다.

- 강한 참조 사이클 (Strong Reference Cycles): 두 인스턴스가 서로를 강하게 참조하여 메모리에서 영구히 해제되지 않는 문제를 말합니다.
- 해결 방안:
  - `weak` 참조: 참조하는 인스턴스의 수명이 더 짧을 때 사용하며, 인스턴스가 해제되면 자동으로 `nil`이 됩니다. (항상 옵셔널)
  - `unowned` 참조: 참조하는 인스턴스의 수명이 같거나 더 길 때 사용하며, 옵셔널이 아니어야 하는 상황에 적합합니다.
  - 클로저의 캡처 리스트: 클로저 내부에서 `self` 등을 참조할 때 `[weak self]` 또는 `[unowned self]`를 사용하여 순환 참조를 방지합니다.