1. 콘솔로그
   1. Print = 단순 문자열 출력(ex) let a: Int = 10, print(a))
   2. dump = 인스턴스의 자세한 설명(ex)

**class Person{**

**var name: String = "yagom"**

**var age: Int = 10**

**}**

let yagom : Person = Person()

dump(yagom)

1. 문자열 보간법 = 변수또는 상수의 직접적인 표현을 위한 것
   1. ex)print(“안녕하세요! 저는 \(age)살입니다”
2. 상수,변수의 표현
   1. (상수) let 이름: 타입 = 값 (타입이 명확하다면 let 이름 = 값)
   2. (변수) var 이름: 타입 = 값 (타입이 명확하다면 var 이름 = 값)
   3. 변수(var)는 차후에 다른 값을 할당 가능. 상수(let)는 불가능
3. 기본 데이터 타입
   1. Bool = true,false //0과 1은 정수값이기때문에 안됨
   2. Int = 정수 // 실수는 넣을 수 없다
   3. UInt = unsigned int //-100넣으면 오류. Int형 X
   4. Float = 정수 받기 가능
   5. Double = 정수 받기 가능. Float은 불가능
   6. Character = 한글자를 표현하기 위한 변수형. 큰따옴표 사용. 유니코드 사용하는 모든 문자 가능(이모티콘도)
   7. String = +연산자 사용으로 문자열 합치기 가능. Character형 받아오는건 불가능.
4. Any, AnyObject, nill
   1. Any = Swift의 모든 타입을 지칭하는 키워드. nil은 받지 못함. Double형에 Any형을 할당해 주는 것도 불가능
   2. AnyObject = 모든 클래스 타입을 지칭하는 키워드. nil은 받지 못함. 오직 클래스만 받음
   3. nil = 없음을 의미하는 키워드.
5. 컬렉션타입(Array, Dictionary, Set)
   1. Array = index가 있음(순서가 있다)  
      (선언) var integers: Array<Int> = Array<Int>()  
      Array<int> == [Int]. 빈 배열은 []로 나타냄  
      append, contains, remove, removeLast, removeAl, count
   2. Dictionary = 키와 값의 쌍으로 이뤄짐  
      (선언) var anyDictionary: Dictionary<String,Any> = [String:Any]()  
      (할당) anyDictionary[“someKey”] = “value” //[:] == 빈 Dictionary  
      (삭제) removeValue(forKey: “anotherKey”) == anyDictionary[“anotherKey”] = nil
   3. Set = 순서가 없고 멤버 유일.  
      (선언) Array와 동일  
      insert, contains, remove, removeFirst, count  
      union(합집합), sorted(정렬), intersection(교집합), subtracting(차집합)
6. 함수 기본
   1. (선언) func 함수이름(변수이름: 타입, 변수이름2: 타입2,….) -> 반환타입{

함수 구현  
 return 반환값 }  
 B. 매개변수, 반환타입 생략 가능

1. 함수 고급
   1. func a(b:String, c:String = “Me”){} 처럼 기본 인자 할당 가능.
   2. 전달인자 레이블(중복 선언 가능하게함)  
      func a(to b:String, from c:String = “Me”){} == 위와 다른 함수로 씀.   
      a(to: “haba”, from: “nana”) 처럼 쓴다.
   3. 가변 매개변수  
      함수당 하나만 가질 수 있다.   
      마침표 세개를 마지막에 쓰는것으로 표현  
      ex) print(functionName(me : “haha”, friends: “a”, “b”, “c”))  
      가변 매개변수에 아무것도 할당하기 싫으면 그냥 생략. nil X
   4. 함수를 타입처럼 쓰기 가능  
      var aFunction:(String,String)->void = a (to:from:)  
      func run( function: (String, String)->void) {  
       function(“a”, “b”)} 🡪 함수를 매개변수로 받음
2. 조건문
   1. if{} else if{} else{} 🡪 소괄호 생략 가능. 중괄호 생략 불가능
   2. switch구문  
      범위연산자를 쓴다( 1..<100 == (1<= tmp < 100)) (1…100 == (1<= tmp <= 100))  
      default구문 꼭 필요. break는 안써도댐. fallthrough는 break를 쓰지 않은것처럼 동작.
3. 반복문
   1. for item in items{}
   2. while ~~~(Bool type만 가능) {} //while 1 {} 불가
   3. repeat{} while ~~~ // do아님 repeat임
4. 옵셔널
   1. 값이 있을수도 있고 없을수도 있다. nil의 가능성을 표현
   2. (선언) 열거형 = enum Optional<Wrapped> : ExpressibleByNilLiteral{  
       case none  
       case some(Wrapped) }
   3. (선언) 일반형 = Int! = 100 or Int? = 100
   4. 암시적 추출 옵셔널  
      Int! = 100 + 열거형  
      기존 변수처럼 사용 가능. nil도 사용 가능. nil할당 해준 후 +1을 해주면 오류
   5. 일반적인 옵셔널  
      Int? = 100 + 옵셔널  
      nil할당 가능  
      일반 변수처럼 사용은 불가능
5. 옵셔널 값 추출
   1. 옵셔널 바인딩 == nil 체크 + 안전한 값 추출.   
      if-let 사용  
      쉼표로 여러 조건   
      나열 가능

func printName(\_ name: String) {

print(name)}

var myName: String? = “a”

printName(myName!) // a. 강제 값 추출

myName = nil

print(myName!) // 강제추출 값이 없어 런타임 오류 발생

var yourName: String! = nil

printName(yourName) // !생략. 런타임 오류 발생

func printName(\_ name: String) {

print(name)}

var myName: String! = nil

if let name: String = myName{

printName(name)}

else{

print(“myName == nil”)

* 1. 강제 추출  
     추천 하지 않음

1. 구조체
   1. (선언) struct 이름{구현}
   2. property == 구조체 안에 들어있는 변수  
      static없이 선언된 property == instance property  
      “ 있으면 type property == struct 타입 자체에서 쓸 수 있게됨.
   3. method == 구조체 안에 들어있는 함수. static 붙일수도 있고 아닐수도 있음
   4. let으로 선언되면 값들을 아예 바꿀 수 없음
2. 클래스
   1. 클래스는 참조타입. 구조체는 값 타입
   2. 클래스는 다중 상속이 될 수 없다.
   3. (선언) class 이름 {}
   4. let으로 선언되어도 class안에서 var로 선언된 변수는 바꿀 수 있음
   5. 재정의 불가 타입 메서드-static  
      static func typeMethod(){ print(“type method – static”)
   6. 재정의 가능 타입 메서드-class  
      class func classMethod(){ print(“type method-class”)
3. 열거형(중요!)
   1. (선언) enum Weekday{  
      case mon  
      case tue  
      case wed,thu,fri}
   2. var day: Weekday = Weekday.mon  
      day = .tue
   3. switch문에서는 모든 값을 사용. 그렇지 않으면 dafault값 넣어야함
   4. 열거형에 정수값을 가지게 하고 싶으면 이름뒤에 :Int 명시해줌. 자동으로 1씩 증가
   5. 원시값을 불러올때는 rawValue사용. ex) School.middle.rawValue
   6. let apple: Fruit? = Fruit(rawValue: 0) 이렇게 생성해야한다. case가 없을수도 있기 때문
   7. 메서드 추가 가능
   8. 상속 불가
4. 값 타입과 참조 타입
   1. 클래스 == 참조타입 || 구조체,열거형 == 값 타입
   2. 구조체  
      참조가 아닌 복사를 원할때
   3. 값타입 == 데이터를 전달할 때 값을 복사하여 전달. struct 2개가 있을 때 한 값이 바뀌어도 다른 곳에선 바뀌지 않음  
      참조타입 == 데이터를 전달할 때 값의 메모리 위치를 전달. class 2개가 있을 때 한 값이 바뀌면 다른 곳에서도 바뀜.
5. 클로저
   1. (선언) { (매개변수 목록)-> 반환타입 in   
       실행코드 }
   2. ex) var sum: (Int, Int) -> Int = { (a: Int, b: Int) -> Int in  
       return a+b }
   3. 함수도 클로저의 일종. 클로저에 함수 할당 가능.
6. 클로저 고급
   1. 후행클로저  
      ex) func calculate(a: Int, b: Int, method : 1)-> Int{  
       return method(a,b) } 가 있을때  
       result = calculate(a:1,b:2){ (left: Int, right: Int) (-> Int 생략가능) in return left + right} 처럼 뒤에 전달 가능
   2. 단축 인자 이름(매개변수 생략. 타입을 알고 있으니까 생략 가능)  
      ex) result = calculate(a:10 , b: 10, method: { return $0 + $1 })
   3. 암시적 반환 표현  
      ex) result = calculate(a: 10 , b: 10){ $0 + $1 }
7. 프로퍼티
   1. 열거형, 구조체, 클래스 안의 변수 == 저장 프로퍼티
   2. 연산 프로퍼티  
      ex) var westernAge: Int {  
       get { return KoreanAge – 1}  
       set(inputValue) { KoreanAge = inputValue – 1}}
   3. 타입 저장 프로퍼티  
      ex) static var typeDescription: String = “학생”
   4. 읽기 전용 연산 프로퍼티 == 연산 프로퍼티에서 get만 있는것
   5. 연산 프로퍼티는 set만 있으면 안된다. get or get&set
   6. 전역변수로도 쓰일 수 있다.
8. 프로퍼티 감시자
   1. 상태가 바뀔 때 사용한다
   2. willSet == 상태가 바뀌기 전 사용  
      didSet == 상태가 바뀐 후 사용
   3. ex) var currencyRate: Double = 1100 {  
       willSet(newRate) { print(“환율 바뀔 예정”)}  
       didSet(oldRate) { print(“환율 바뀌었다”)}
   4. 연산 프로퍼티는 프로퍼티 감시자와 같이 쓰일 수 없다.
   5. 전역변수로도 쓰일 수 있다.
9. 상속
   1. 상속은 클래스, 프로토콜 등에서 가능(다중상속은 불가능)
   2. 열거형, 구조체는 상속 불가능
   3. (선언) class 이름: 상속받을 클래스 이름 { 구현 }
   4. final을 함수 앞에 쓰면 자식 클래스에서 수정 불가  
      ex) class Person{   
       final func sayHello() { print(“hello”) }}
   5. static 함수 == 재정의 불가 타입 메서드  
      class 함수 == 재정의 가능 타입 메서드
   6. final class func은 재정의 할 수 없다. (== static)
   7. 자식 클래스에서 재정의를 하고 싶으면 override func이렇게 쓰면 됨.
10. 인스턴스의 생성과 소멸
    1. 인스턴스의 프로퍼티를 초기화 해주지 않으면 오류뜨기때문에 이니셜라이저 씀  
       ex) class a { var b : String var b : Int  
        init(a : String, b : Int){ self.a = a self.b = b}}  
        let hana : a = a(b = “hana”, c = “1218”)
    2. self.init이 이미 만들어져 있음. 그거 init함수에 사용 가능  
       ex) convenience init (~~){self.init(name : name, age : age)}
    3. 일반적 옵셔널을 쓸 땐 생략 가능.   
       암시적 옵셔널 쓸 때도 생략 가능. 값이 초기화해주기 애매하지만 꼭 필요할때 사용
    4. 실패가능한 이니셜라이저  
       init?(~~){if ~~ {return nil} ~~~}   
       + let john: PersonD? = PersonD(~~)
    5. 디이니셜라이저 == 인스턴스가 해제되는 시점에 호출. 클래스타입에만 가능  
       ex)deinit{ if let perName = pet?.name { print(~~) self.pet?.owner = child}}
11. 옵셔널 체이닝과 nil 병합 연산자
    1. 옵셔널 체이닝 == 옵셔널이 nil인지 아닌지 계속 체크해야할때 유용.(옵셔널이 연속적으로 연결되는 경우)  
       ex) class Person{  
        var name:String  
        var job: String?  
        var home: Apartment?(클래스임)  
         
        init(name: String){ self.name = name}}  
          
        class Apartment{  
        var buildingNumber: String  
        var roomNumber: String  
        var ‘guard’: Person?  
        var owner: Person?  
         
        init(dong: String, ho: String){ buildingNumber = dong roomNumber = ho}}  
         
        //옵셔널 체이닝 사용  
        guardJobWithOptionalChaining(owner: Person?){  
        if let guardJob = owner?.home?.guard?.job{//사람이 있니? 집이 있니? 경비원이 있니? 직업은 있니?  
        print(“우리집 경비원의 직업은\(guadJob)입니다”)}  
        else { print((“우리집 경비원은 직업이 없어요”))}
    2. nil 병합 연산자  
       ex) guardJob = yagom?.home?.guard?.job ?? “슈퍼맨”  
        ??뒤에 나오는 것은 할당해주는 값. 앞의 값이 nil이면 할당. nil이 아니면 할당X
12. 타입캐스팅
    1. 인스턴스의 타입을 확인하는 용도. 클래스를 부모 혹은 자식 클래스 타입으로 사용할 수 있는지 확인하는 용도. is,as사용
    2. Person, Student(Person이 부모 클래스), UniversityStudent(Student가 부모 클래스)  
       result = hana is Person // true   
       result = hana(Student) is Student // true  
       result = hana is UniversityStudent // false
    3. 업 캐스팅(많이 사용하진 않음)  
       ex) var mike: Person = UniversityStudent() as Person
    4. 다운 캐스팅(as?,as!) // 자식 클래스의 인스턴스로 사용할 수 있도록 해줌
    5. 조건부 다운 캐스팅 (as?)  
       var optionalCasted: Student?  
       optionalCasted = mike(Person) as? UniversityStudent //마이크는 실질적 대학생인데 Person으로 업 캐스팅 한거니까 가능  
       optionalCasted = jenny(Student) as? UniversityStudent // nil //제니는 학생이었기에 대학생이 될 수 없음.
    6. 강제 다운 캐스팅  
       var forcedCasted: Student?  
       forcedCasted = mike(Person) as? UniversityStudent //가능  
       forcedCasted = jenny(Student) as? UniversityStudent // 런타임 오류
13. assert와 guard
    1. 동작 도중 생성하는 다양한 결과값 동적으로 확인&처리
    2. assert   
       ex) var someInt: Int = 0  
        assert (someInt == 0, “someInt != 0”)  
        만약 앞의 조건문이 맞다면 그냥 지나치고 아니면 뒤의 메세지 출력&동작중지  
        메세지는 생략 가능. 저 경우는 통과
    3. guard // early exit  
       return,break같은 코드블럭 종료 지시어가 꼭 있어야함.   
       else도 꼭 따라붙음  
       guard문의 조건을 충족시키면 그 밑의 코드로 내려감. 계속 실행 가능  
       ex) guard count < 3 else { break }  
        ~~~~~  
       함수, 반복문에도 사용 가능
14. 프로토콜
    1. 타입에 이 기능이 꼭 필요하다고 강요하는것.
    2. (선언) protocol 이름 {정의}
    3. ex) protocol Talkable{   
        var topic: String { get set }  
        var language: String { get }  
        func talk()  
        init(topic: String, language: String)}  
         
       struct Person: Talkable{~~~~} // 프로토콜을 채택 했다.
    4. 프로토콜은 다중상속이 가능하다  
       (선언) protocol 이름:부모\_프로토콜\_이름\_목록 {정의}
    5. 클래스가 프로토콜&부모클래스를 상속 받으려면 부모클래스를 먼저 쓰고 프로토콜들 씀
15. 익스텐션
    1. 구조체,클래스,열거형,프로토콜 타입에 새로운 기능 추가하는 기능
    2. 존재하는 기능을 재정의하는것은 불가.
    3. (선언) extension 확장할 타입 이름{ 타입에 추가될 새로운 기능 구현 }  
        extension 확장할 타입 이름: 프로토콜1, 프로토콜2, 프로토콜3….{  
        프로토콜 요구사항 구현 }
    4. 타입이름에 자료형(ex) Int)들어갈 수 있다. 새로운 기능은 메서드,이니셜라이저 가능
16. 오류처리
    1. error 프로토콜과 (주로) 열거형을 통해 오류를 표현
    2. ex) enum 오류종류이름: Error{ case 종류1 case 종류2…}
    3. 메서드에서 발생한 오류 던지기 🡪 throws 사용  
       ex) func receiveMoney(\_ money: Int) throws{//오류 발생할 수 있다고 표시한것  
        guard money > 0 else { throw VendingMachineError.invalidInput }//오류 발생
    4. 오류발생 여지 있는 throws함수는 try를 사용해 호출해야함. 🡪 try, try?, try!  
       + do-catch구문  
       ex) do{ try machine.receiveMoney(0)}  
        catch VendingMachineError.invalidInput{ print(“입력이 잘못되었습니다.”)}  
        catch ~~~  
       or  
       ex) do{ try machine.receiveMoney(0)}  
        catch { switch error{ case ~~:~~ case ~~:~~…}}
    5. try? == 별도의 오류처리 결과를 통보받지 않음. 오류 발생하면 결과값을 nil로 받음.  
       ex) result = try? machine.vend(numberOfItems:2)  
        result
    6. try! == 오류가 발생하지 않을 것이라는 강력한 확신을 가질때. 오류 발생하면 런타임 오류가 발생e  
       ex) result = try! machine.vend(numberOfItems:1)  
        result
17. 고차함수
    1. map, filter, reduce
    2. map == 컨테이너 내부의 기존 데이터를 변형하여 새로운 컨테이너 생성  
       ex) let numbers: [Int] = [0,1,2,3,4]  
        doubledNumbers = numbers.map( { (number:Int) -> Int in   
        return number \* 2 })  
        == (doubledNumbers = numbers.map { $0 \* 2 } )
    3. filter == 컨테이너 내부의 값을 걸러서 새로운 컨테이너로 추출  
       ex) let evenNumbers: [Int] = numbers.filter{  
        (number: Int) -> Bool in   
        return number % 2 == 0}   
       == (let evenNumbers: [Int] = numbers.filter{ $0 % 2 != 0 } )
    4. reduce == 컨테이너 내부의 콘텐츠를 하나로 통합  
       ex) 초기값이 0이고 someNumbers내부의 모든 값을 뺀다.  
        var subtract: Int = someNumbers.reduce(0, {   
        (first: Int, second: Int) -> Int in  
        return first – second } )  
        == ( var subtract: Int = someNumbers.reduce(0) { $0 - $1 } )