2부

1. 테스트

```
import atomictest.*

fun main() {
    val a = 'a'
    // 'a'
    a eq 'a'
    // 'a'
    // [Error]: a != b
    a neq 'b'

    trace("a")
    trace(48)
    trace("B")
    trace eq """
    a
    48
    B
    """
}
```

- 중위 표기법으로 eq, neq 을 넣어서 테스트 가능
- trace 를 사용하면 println 를 대치할 수 있다.
  trace 한 내용을 기록하고 eq 로 기록된 데이터를 비교 가능

2. 객체는 모든 곳에 존재한다

- 코틀린은 하이브리드 객체-함수형 언어 (객체지향, 함수형 프로그래밍 지원)
- 멤버 함수, 프로퍼티, 객체 생성(인스턴스 생성) - 간단 설명

3. 클래스 만들기

```
class NoBody

class SomeBody {
    // 멤버 변수
    val name = "Jonghyun Park"

    // 멤버 함수
    fun getFirstName () = name.split(" ")[1]
```

```
}

class EveryBody {
    val all = list(SomeBody(), SomeBody(), SomeBody())
}

fun main() {
    val nb = NoBody()
    val sb = SomeBody()
    val eb = EveryBody()

    // Park
    sb.getFirstName() eq "Park"
}
```

- java 와 다르게 객체 생성에 new 가 안 들어간다. (함수 실행 처럼 들어감)

4. 생성자

```
// 생성자에 멤버 변수 init 이 필요한 경우

// id, years 파라미터는 생성자 본문에만 쓸 수 있다.
class Badger (id: String, years: Int) {
    val name = id
    val age = years
    override fun toString(): String {
        return "Badger: $name, $age"
    }
}

// 파라미터가 프로퍼티로 변경되고 클래스 내부에서도 이 프로퍼티를 사용가능하다.
class Snake (
    val type: String,
    val length: Double
) {
    override fun toString(): String {
        return "Snake: $type, $length"
    }
}

fun main() {
    Badger("Bob", 11) eq "Badger: Bob, 11"
    Snake("Garden", 2.4) eq "Snake: Grden, 2.4"
}
```

5. 가시성 제한하기

- 코틀린 접근 제한자 public, private, protected, internal
  지정을 하지 않으면 public 이 들어감 (java 는 package-private)
  internal 은 모듈 내부에서만 볼 수 있다. (모듈이 더 큰 범위, 패키지가 보다 작은 범위)
- 접근 제한자 파일의 클래스, 함수, 프로퍼티, 클래스 맴버(함수, 프로퍼티)

6. 예외

```
import atomictest.*

fun divide(num, num2) {
    if (num == 0 || num2 == 0)
        throw IllegalArgumentException("num is zero")
    return num / num2
}

fun main() {
    capture {
        "$1.9".toDouble()
    } eq "NumberFormatException: " +
        """For input string: "$1.9""""

    capture {
        divide(0, 0)
    } eq "IllegalArgumentException: " +
        "num is zero"
}
```

- `capture` 함수는 책에서 생성한 함수, 예외가 발생하는 모습을 보여주고, 빌드 시스템이 예외 출력을 검사할 수 있도록 구현됨
- 예외를 던지려면 `throw` 키워드 사용

7. List

- 자바 컬렉션 - 코틀린 컨테이너
- `listOf` 읽기 전용 리스트 - 변경 함수 작동 X
- `mutableListOf` 가변 리스트 - 변경 함수 작동 O

```
import atomictest.*

fun main() {
    val first = mutableListOf(1)
    val second: List<Int> = first
    second eq listOf(1)
    first += 2
    // second 에서도 변경된 내용을 볼 수 있다.
    second eq listOf(1, 2)
}
```

- second 와 first 가 한 객체에 대해 참조 여러 개를 유지 하는 경우 **에일리어싱** 이라 부른다.

8. 파라미터화한 타입

- 타입 파라미터(<>)
  ex)
  `val nums: List<Int>, val strs: List<String>`
- 주로 컨테이너에서 자주 사용됨

9. 가변 인자 목록

```
fun varargs(s: String, vararg ints: Int {
    for (i in ints) {
        trace("$i")
    }
    trace(s)
}

fun main() {
    varargs("primes", 5, 7, 11, 13, 17, 19, 23)
    trace eq "5 7 11 13 17 19 23 primes"
}
```

- `vararg` 키워드 - 가변 인자 목록
  함수가 임의의 개수만큼 같은 타입의 인자를 받게 해줌 (길이 0도 포함!)
  보통 사용법은 마지막 파라미터를 vararg 로 지정한다.
- vararg 는 Array 가 된다.
  Array 는 항상 가변이다!
  Array 타입이 아닌 여러 값으로 이뤄진 시퀀스로 취급하려면
  스프레드 연산자(*) (Array 에서만 가능) 를 써야 한다.

10. Set (집합)

- Set은 중복을 자동으로 제거해준다.
- `setOf` 읽기 전용 집합 - 변경 함수 작동 X
- `mutableSetOf` 가변 집합 - 변경 함수 작동 O
- in, contains() 를 사용해 집합의 원소인지 검사 가능

11. Map (맵)

- Map은 key, value 를 연결시키고 키가 주어지면 값을 찾아준다.
- `map.keys` 키 전부를 List 로 보여줌
- `map.values` 값 전부를 List 로 보여줌
- `mapOf` 읽기 전용 집합 - 변경 함수 작동 X
- `mutableMapOf` 가변 집합 - 변경 함수 작동 O

12. 프로퍼티 접근자

```
class Holder(var i: Int)

fun main() {
    val holder = Holder(10)
    holder.i eq 10  // i 프로퍼티를 읽음
    holder.i = 20   // i 프로퍼티에 값을 씀
}
```

- 프로퍼티 값을 얻기 위한 사용하는 접근자 **게터 (getter)**
  프로퍼티 값을 설정 위한 사용하는 접근자
  세터 (setter)

```
// trace 를 추가한 커스텀 get, set
class Default {
    var i: Int = 0
        get() {
            trace("get()")
            return field // get 기본 행동 박식
        }
        set(value) {
            trace("set($value)")
            field = value // set 기본 행동 박식
        }
}
```

- `field` 는 게터와 세터안에서만 접근할 수 있는 이름
- 커스텀 설정 방법은 프로퍼티 설정 후 바로 다음에 `get(), set()` 을 지정하면 된다.
- `private`로 프로퍼티를 선언하면 두 접근자 모두 `private`가 된다.

p. 연습문제 기록용