



6부 (실패 방지하기)

1. 예외 처리

- 오류 보고

- 표준 lib 예외로 충분하지 않는 경우 예외 타입을 정의할 수 있다. (Exception 상속)

```
class Exception1(  
    val value: Int  
): Exception("wrong value: $value")  
  
open class Exception2(  
    description: String  
): Exception(description)  
  
class Exception3(  
    description: String  
): Exception2(description)
```

- 복구

- 예외 처리의 큰 목표는 복구이다.
(문제를 해결하고 프로그램을 안정적인 상태로 되돌린 후 실행을 진행한다.)
(복구가 불가능한 경우도 꽤 많다. 복구 할 수 없는 프로그램의 실패, 환경 문제 등)
- java 와 동일하게 `catch` 를 사용해 프로그램 종단을 막을 수 있다.

- 예외 하위 타입

- 문제에 대해 두가지 다른 상황에서 같은 예외를 쓰는 경우가 있으니, 예외에 어울리는 타입을 지정하는 것이 좋다. (추상화 수준에 맞는 예외를 쓴다.)
- 너무 많은 예외 타입을 만들지는 말라.
처리 방식이 달라야 한다면 다른 예외 타입을 사용해 이를 구분하고, 처리 방법이 같은 경우 동일한 예외 타입을 쓰면서 생성자 인자를 다르게 주는 방식으로 전달한다.

- 자원 해제

- 실패를 피할 수 없을 때 자원(IO, Socket 등)을 자동으로 해제하게 만들면 프로그램의 다른 부분이 계속 안전하게 실행되도록 도움 줄 수 있다.
- java 와 동일하게 `finally` 를 사용해 예외를 처리하는 과정에서 자원을 해제할 수 있게 보장한다.
- java 와 동일하게 `try` 안에서 `return` 이 있더라도 `finally` 절은 여전히 실행된다.

- 가이드라인 (코틀린에서 예외를 사용하는 경우)

- 논리 오류
 - 코드에 있는 버그다.

- 애플리케이션의 최상위 수준에서 예외를 잡고 버그 수정해라

```
fun calculateTotalPrice(quantity: Int, price: Int): Int {
    if (quantity < 0 || price < 0) {
        throw IllegalArgumentException("음수 값은 사용할 수 없습니다.")
    }
    return quantity * price
}
```

◦ 데이터 오류

- 프로그래머가 제어할 수 없는 잘못된 데이터에 생긴 오류
- 애플리케이션은 프로그램 로직을 탓하지 않고 어떻게든 이런 오류를 처리해야 한다.

```
fun parseNumber(input: String): Int {
    return input.toInt()
}
```

◦ 검사 명령 (다음 절 `require` 키워드)

- 논리 오류를 검사한다.
- 버그를 찾으면 예외를 발생시키지만, 함수 호출처럼 보이기 때문에 코드가 명시적으로 예외를 던질 필요가 없어진다.

```
fun checkAge(age: Int) {
    require(age >= 0) { "나이는 음수일 수 없습니다." }
}
```

◦ IO 오류

- 제어할 수 없는 외부 조건이지만, 무시해서는 안 되는 오류
- IO 오류로 볼너 복구할 수 있는 경우가 종종 있음...
- 코틀린에서는 IO 예외를 던지므로 애플리케이션은 이런 예외를 처리하고 복구를 시도하는 코드가 포함되어야 한다.

```
import java.io.File

fun readFromFile(fileName: String): String {
    val file = File(fileName)
    return file.readText()
}
```

2. 검사 명령

- 검사 명령은 만족시켜야 하는 제약 조건을 적은 단언문이다.
보통 함수 인자와 결과를 거증할 때 검사 명령을 사용한다.
일반적으로 실패 시 오류를 던진다.

- `require()`
 - 함수 인자를 검증하기 위해 사용됨. 그에 따라 함수 본문 맨 앞에 위치한다. 즉, 컴파일 시점이 아닌 런타임 시점에 사용된다.

```
data class Month(val monthNumber: Int) {
    init {
        require(monthNumber in 1..12) {
            "Month out of range: $monthNumber"
        }
    }
}

fun main() {
    Month(1) eq "Month(monthNumber=1)"
    capture { Month(13) } eq
        "IllegalArgumentException: " +
        "Month out of range: 13"
}
```

- `require` 조건을 만족하지 못하면 `IllegalArgumentException` 을 반환한다. 예외를 던지는 대신 항상 `require` 을 사용할 수 있다.
- `require()` 의 두 번째 파라미터는 `String` 을 만들어내는 람다다.
- `requireNotNull`
 - 첫 번째 인자가 `null` 인지 검사해 널이 아니면 그 값을 돌려준다. 하지만 `null` 이라면 `IllegalArgumentException` 을 발생시킨다. 성공한 경우 인자는 자동으로 널이 아닌 타입으로 스마트 캐스트된다.

```
fun notNull(n: Int?): Int {
    // 직접 메시지를 조작할 수 있는 파라미터 두 개 버전
    requireNotNull(n) {
        "notNull() argument cannot be null"
    }
    // 호출이 null 이 될 수 없는 값으로 스마트 캐스트해줌으로 n 에 대해 null 검사 하지
    return n * 9
}

fun main() {
    val n: Int? = null
    capture {
        notNull(n)
    } eq "IllegalArgumentException: " +
        "notNull() argument cannot be null"
    capture {
        // 디폴트 메시지를 사용하는 파라미터 한 개 버전
    }
```

```

        requireNotNull(n)
    } eq "IllegalArgumentException: " +
        "Required value was null."
    notNull(11) eq 99
}

```

- **check**

- 계약에 의한 설계의 사후 조건은 함수의 결과를 검사한다.
결과를 확신할 수 없는 복잡하고 긴 함수에서 사후 조건이 유용하다.

require 와 동일하지만 **IllegalStateException** 을 던지는 차이가 있다.
일반적으로 함수의 맨 끝에서 함수 결과가 올바른지 검증하기 위해 사용된다.

```

val resultFile = DataFile("Results.txt")

fun createResultFile(create: Boolean) {
    if (create)
        resultFile.writeText("Results\n# ok")
    // ... 다른 실행 경로들
    check(resultFile.exists()) {
        "${resultFile.name} doesn't exist!"
    }
}

fun main() {
    resultFile.erase()
    capture {
        createResultFile(false)
    } eq "IllegalArgumentException: " +
        "Result.txt doesn't exist!"
    createResultFile(true)
}

```

- 사전 조건이 정상적이라면 사후 조건은 거의 문제 없다. (문제가 있다면 프로그래밍 실수)

- **assert**

- java 에서 넘겨온 예약어로 사용가능하다. 하지만 **require**, **check** 를 사용하는 것이 좋다.

```

// 인자로 boolean 으로 평가되는 표현식 또는 값을 받아서 지나가고, 거짓이면 AssertionError
assert expression1;
// 표현식1 이 거짓인 경우 두 번째 표현식의 값이 예외 메시지로 보여지게된다.
assert expression1: expression2;

```

3. Nothing 타입

- 항상 예외를 던지는 함수의 반환 타입이다.

- 함수가 결코 반환되지 않는다는 뜻의 반환 타입
- 코틀린 내장 타입으로 아무 인스턴스가 없다.
- `Nothing` 은 모든 타입의 서브 클래스

```
fun infinite(): Nothing {
    while(true) {}
}

fun fail(i: Int): Nothing = throw Exception("fail($i)")
```

```
// ver1
class BadData(m: String): Exception(m)

fun checkObject(obj: Any?): String =
    if (obj is String)
        obj
    else
        throw BadData("Needs String, got $obj")

// ver2 엘비스 연산자, 안전한 캐스트
fun failWithBadData(obj: Any?): Nothing =
    throw BadData("Needs String, got $obj")

fun checkObject2(obj: Any?): String =
    (obj as? String) ?: failWithBadData(obj)

fun test(checkObj: (obj: Any?) -> String) {
    checkObj("abc") eq "abc"
    capture {
        checkObj(null)
    } eq "BadData: Need String, got null"
    capture {
        checkObj(123)
    } eq "BadData: Need String, got 123"
}

fun main() {
    test(::checkObject)
    test(::checkObject2)
}
```

- `checkObject()` 반환 타입은 `if` 식의 타입이다. `if/else` 나 `if/else` 식의 타입은 모든 가지의 식이 만들어내는 값의 타입을 만족하는 최소한의 공통 타입이다.
- 코틀린은 `throw` 를 `Nothing` 타입으로 취급하고, `Nothing` 은 임의의 타입의 하위 타입으로 취급될 수 있다.

```

fun main() {
    val none: Nothing? = null

    var nullableString: String? = null
    nullableString = "abc"
    nullableString = none
    nullableString eq null

    val nullableInt: Int? = none
    nullableInt eq null

    val listNone: List<Nothing?> = listOf(null)
    val ints: List<Int?> = listOf(null)
    ints eq listNone
}

```

- 추가적인 타입 정보가 없는 상태로 그냥 `null` 이 주어지면, 컴파일러가 널이 될 수 있는 `Nothing` 을 타입으로 추론한다.
- `null` 과 `none` 모두 `nullableString` 이나 `nullableInt` 같이 널이 될 수 있는 타입의 `var` 나 `val` 에 대입할 수 있다.
- `null` 과 `none` 타입이 모두 `Nothing?` (널이 될 수 있는 `Nothing`) 이므로 이런 대입이 가능하다.
- `Nothing` 타입의 식을 '모든 타입'으로 해석할 수 있고.

`null` 같은 `Nothing?` 타입의 식을 '널이 될 수 있는 모든 타입'으로 해석할 수 있다.

4. 자원해제 (`use`)

- 자바의 `try-resource` 라고 보면 될 것 같다.
- `AutoCloseable` 인터페이스를 구현하는 모든 객체에 `use()` 를 사용할 수 있다.
- `use()` 는 인자로 받은 코드 블록을 실행하고, 그 블록을 어떻게 빠져나왔는지에 관계없이 객체의 `close` 를 호출한다.

```

fun main() {
    DataFile("Results.txt")
        .bufferedReader()
        .use { it.readLines().first() } eq
        "Results"

    // useLines 함수
    // 파일의 모든 줄을 읽어 처리한 후 자동으로 파일을 닫고, 대상 함수에 모든 줄을 전달한다.
    DataFile("Results.txt").useLines() {
        it.filter { "#" in it }.first()
    } eq "# ok"

    // forEachLine 함수
}

```

```
// 전달된 람다는 Unit 을 반환한다.  
// 파일을 열고 각 줄을 순차적으로 처리한 후, 작업이 완료되면 파일을 자동으로 닫습니다.  
DataFile("Results.txt").forEachLine {  
    if (it.startsWith("#"))  
        trace("$it")  
}  
trace eq "# ok"  
}
```