모바일앱개발및응용 6주-1강

코틀린의 설계 도구와 Null Safety

학습 내용

1. 설계 도구
2. Null Safety

학습 목표

1. 코틀린의 설계 도구인 패키지, 추상화 클래스, 인터페이스, 접근 제한자, 제너릭을 알고 사용할 수 있다.
2. 개발시 문제가 되는 null 값을 안전하게 처리하는 Null Safety를 알고 사용할 수 있다.

사전학습

클래스 란

- 그룹화할 수 있는 함수와 변수를 한군데 모아놓고 사용하기 쉽게 이름을 붙여놓은 것

클래스의 생성

- 작성된 클래스를 사용하기 위해서는 생성자(constructor)라고 불리는 함수가 호출되어야 함
- 코틀린의 클래스 사용은 함수의 사용과 비슷함

사전학습

클래스의 사용

- 클래스 명 뒤에 ()괄호를 붙이면 클래스의 생성자(constructor)를 호출할 수 있음
- 이때 생성된 것을 인스턴스(Instance)라고 하는데, 인스턴스는 변수에 담아둘 수 있음

사전학습

object

- object를 사용하면 생성자()로 **인스턴스를 만들지 않아도 object 안의 프로퍼티와 메서드**를 호출해서 **바로 사용**할 수 있음

사전학습

data class 생성과 사용

- data class는 한 줄 클래스 등 보통 간단한 변수의 저장을 위해 사용됨
- data class 키워드를 사용하며, 매개변수에는 var 또는 val을 붙여야 함, 한 줄로 모두 표현해야 하므로 var과 val의 생략은 불가

```
data class 클래스명 (val 매개변수1 : 타입, var 매개변수2 : 타입)
```

사전학습

클래스 상속

- 부모 클래스는 **open class 키워드**로 만들어야만 하고 자식 클래스는 **class 키워드**로 만들어야 함
- 상속은 **부모의 클래스를 자식이 갖는 과정**이기 때문에, 인스턴스를 생성하듯 부모 클래스명 뒤에 **()괄호**를 붙여 **부모의 생성자를 호출**해야 함

```
open class 상속될 부모 클래스 {
    // 코드
}
class 자식 클래스: 부모 클래스() {   // 상속
    // 코드
}
```

생성자 호출

사전학습

override(오버라이드, 재정의)

- 부모 클래스의 메서드와 프로퍼티를 상속받았으나, 자식 클래스에서는 **다른 용도로 사용해야 하는 경우** override를 사용하여 **재정의** 할 수 있음
- 부모 클래스(open class) 안의 **메소드와 프로퍼티가 open 키워드**로 정의되어 있어야 있어야, 자식 클래스(class)에서 **override(재정의)**를 할 수 있음

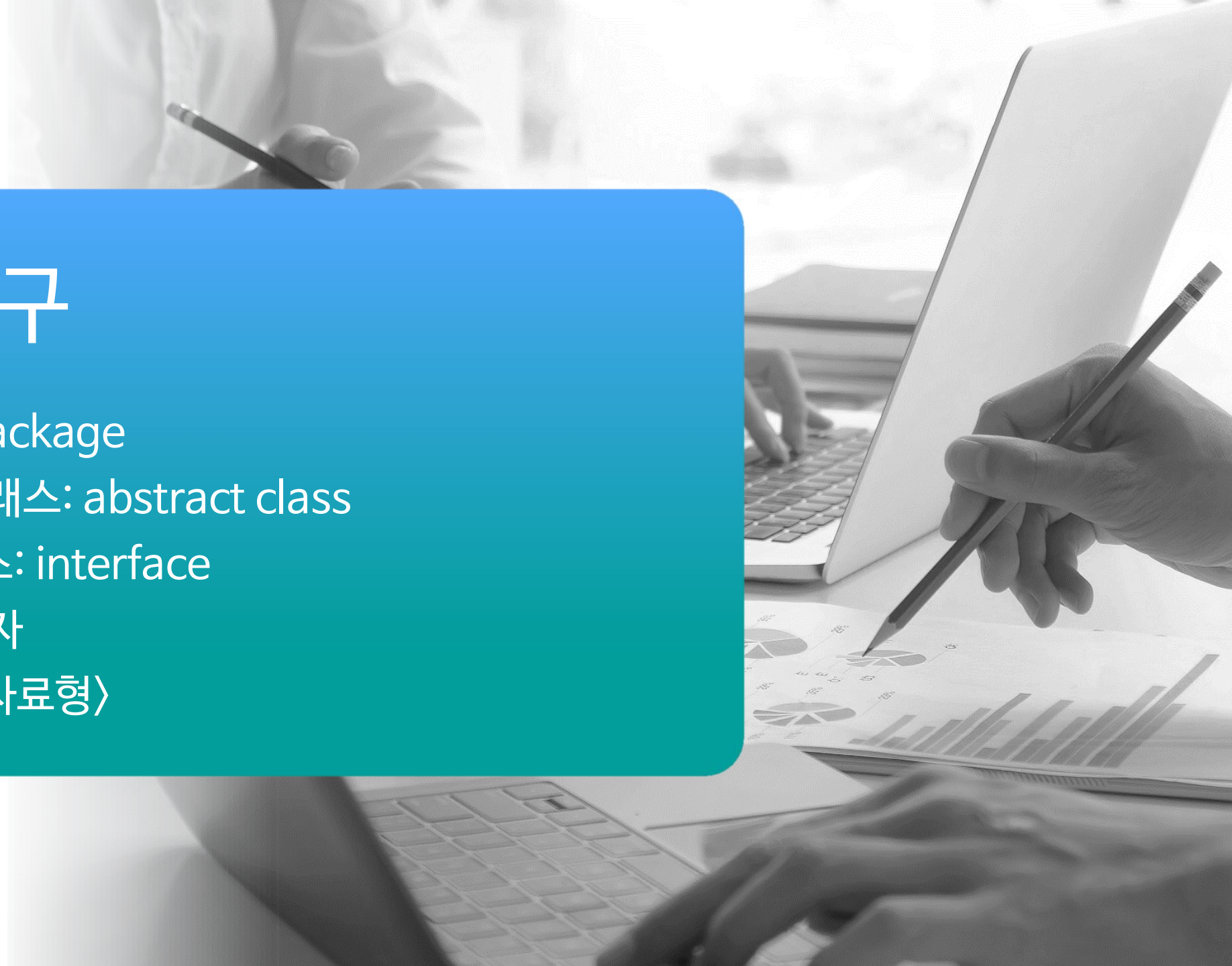
사전학습

익스텐션 (Extensions, 확장)

- 코틀린은 클래스에 **메서드와 프로퍼티의 확장**을 지원함

```
fun 클래스.확장할 메서드() {
    // 코드
}
```

  - **상속**이 미리 만들어져 있는 **클래스를 가져다** 쓰는 개념이라면 **익스텐션**은 미리 만들어져 있는 **클래스에 메서드를 넣는** 개념
  - 익스텐션을 사용한다고 해서 **실제 클래스의 코드가 변경되는 것은 아니며 단지 실행 시에만** (.)도트 연산자로 호출해서 사용하는 것임



1. 설계 도구

1.1 패키지: package
1.2 추상화 클래스: abstract class
1.3 인터페이스: interface
1.4 접근 제한자
1.5 제네릭: 〈자료형〉

1.1 패키지: package

- 패키지는 클래스와 소스 파일을 관리하기 위한 **디렉터리 구조의 저장 공간**
- 다음과 같이 현재 클래스가 어떤 디렉터리에 있는지 표시함
- 디렉터리가 **계층 구조로 만들어져 있으면 온점(.)으로 구분**해서 각 디렉터리를 모두 **나열**해 줌

```
package 메인 디렉터리 . 서브 디렉터리
class 클래스 {

}
```

1.2 추상화 클래스: abstract class

- 프로그래밍을 하기 전 **개념 설계**를 하는 단계에서는 클래스의 이름과 클래스 안에 **있음 직한 기능**을 유추해서 **메서드 이름으로 먼저 나열**함
- 이것을 **추상화**라고하며 **abstract class 키워드**를 사용함
- 명확한 코드는 설계 단계에서 메서드 블록 안에 **직접 코드를 작성하기도** 하지만 / 그렇지 않은 경우에는 구현 단계에서 코드를 작성할 수 있도록 **메서드의 이름**만 작성해 놓음

1.2 추상화 클래스: abstract class

- **구현 단계**에서는 이 **추상화된 클래스를 상속**받아서 아직 **구현되지 않은 부분을 마저 구현**함

```
abstract class Design {
    abstract fun drawText()
    abstract fun draw()
    fun showWindow() {
        // 코드
    }
}
```

이름만

이름만

이미 구현됨

```
class Implements: Design() {
    fun drawText() {
        // 구현 코드
    }
    fun draw() {
        // 구현 코드
    }
}
```

1.2 추상화 클래스: abstract class

- **abstract class 예)**
  - ✓ 실제 구현 클래스는 이 **abstract class**를 **상속**받아서 **아직 구현되지 않은** 추상화되어 있는 기능을 모두 구현해 줌
  - ✓ **abstract class**는 독립적으로 **인스턴스화할 수 없기** 때문에 구현 단계가 따로 하지 않는다면 잘못된 설계가 될 수 있음
  - ✓ **abstract fun**으로 정의된 함수를 **override로 정의**할 수 있음

```
abstract class Animal {
    fun walk() {
        Log.d("abstract", "걷습니다.")
    }
    abstract fun move()
}
```

추상화 클래스

추상화 함수도 abstract fun으로 정의

```
class Bird: Animal() {
    override fun move() {
        Log.d("abstract", "날아서 이동합니다.")
    }
}
```

override fun 으로 move()에 코드작성

1.3 인터페이스: interface

- **interface는** 설계단계에서 **실행코드 없이 메서드 이름만** 나열해 놓은 것
  - ✓ 설계단계에서 클래스에 **코드가 한 줄**이라도 필요하면 **추상화**
    설계단계에서 클래스에 코드 필요없이 **메서드 명만** 필요하면 **인터페이스**

- **interface 형식**
  - ✓ 인터페이스는 **interface 키워드**를 사용해서 정의할 수 있고
    인터페이스에 정의된 메서드를 **override로 구현**할 수 있음
  - ✓ **프로퍼티도** interface 안에 **정의**할 수 있음, **abstract는 생략**함

```
Interface 인터페이스명 {
    var변수: String
    fun 메서드1()
    fun 메서드2()
}
```

1.3 인터페이스: interface

➢ interface 만들기

```
interface InterfaceKotlin {
    var variable: String
    fun get()
    fun set()
}
```

abstract 키워드가 생략되어 있습니다.

interface 로 class 만들기

- 인터페이스로 클래스를 만들 때는 상속과는 다르게
  생성자()를 호출하지 않고 인터페이스 이름만 지정해주면 됨

```
class KotlinImpl: InterfaceKotlin {
    override var variable: String = "init value"
    override fun get() {
        // 코드 구현
    }
    override fun set() {
        // 코드 구현
    }
}
```

1.3 인터페이스: interface

interface는 큰 프로젝트에서 의사소통 방식을 정의하는 것으로 **혼자 개발하거나 소수의 인원이 interface를 사용하는 것은 좋지 않아요.** 인터페이스는 **가독성이 떨어지고, 구현 효율성도 떨어집**니다.

interface를 object로 바로 사용하기

- interface를 class로 만들어 사용하지 않고, object로 가져와 변수에 저장해서 바로 사용할 수 있음. 실제 실무(큰 프로젝트)에서 자주 사용하는 형태
- 인터페이스를 클래스의 상속 형태가 아닌 소스 코드에서 직접 구현할 때 사용

```
var kotlinImpl = object: InterfaceKotlin {
    override var variable: String = "init"
    override fun get() {
        // 코드
    }
    override fun set() {
        // 코드
    }
}
```

1.4 접근 제한자

- 코틀린의 모든 **class, interface, 메서드, 프로퍼티**는
  접근 제한자를 통해 **자신에 대한 접근을 제한**할 수 있음

➢ 접근 제한자의 종류

접근 제한자	제한 범위	외우기 쉽게!
private	다른 파일에서 접근할 수 없음	**접근금지**
internal	같은 모듈에 있는 파일끼리만 접근 가능	**내부모듈만**
protected	private와 같으나 상속 관계라면 자식 클래스는 접근할 수 있음	**자식만**
public	제한 없이 모든 파일에서 접근 가능	**접근허용**

private : 개인
internal : 내부
protected : 보호
public :공용

부모클래스에 접근 제한자 적용

```
open class Parent {
    private val privateVal = 1
    protected open val protectedVal = 2
    internal val internalVal = 3
    val defaultVal = 4
}
```

접근 금지

자식만

내부모듈만

공용(public 생략)

상속받은 자식클래스

```
class Child: Parent() {
    fun callVariables() {
        // privateVal은 호출이 안 됩니다.
        Log.d("Modifier", "protected 변수의 값은 ${protectedVal}")
        Log.d("Modifier", "internal 변수의 값은 ${internalVal}")
        Log.d("Modifier", "기본 제한자 변수 defaultVal의 값은 ${defaultVal}")
    }
}
```

자식이니 가능

자식이니 당연히 같은 모듈

공용이니 접근

부모클래스에 접근 제한자 적용


```
open class Parent {
    private val privateVal = 1
    protected open val protectedVal = 2
    internal val internalVal = 3
    val defaultVal = 4
}
```

접근 금지

자식만

내부모듈만

공용(public 생략)

상속받지 않은 외부클래스에서 사용할 때

```
class Stranger {
    fun callVariables() {
        val parent = Parent()
        Log.d("Modifier", "internal 변수의 값은 ${parent.internalVal}입니다.")
        Log.d("Modifier", "public 변수의 값은 ${parent.defaultVal}입니다.")
    }
}
```

privateVal 은 접근금지여서 접근불가
protectedVal 은 자식이 아니라서 접근 불가

Parent클래스 생성해서 parent 변수에 저장

같은 모듈이니 접근 가능

공용이니 접근

잠깐! 궁금증 해결

Q **모듈**이 뭔가요? 라이브러리 패키지 안에 있는 그 모듈 맞나요? 제가 알고 있는 모듈과 다른 건가요?

A 코틀린에서 모듈이란 한 번에 같이 **컴파일되는 모든 파일**을 말합니다. 안드로이드를 예로 든다면 **하나의 앱**이 하나의 모듈이 될 수 있습니다. 또한 **라이브러리**도 하나의 모듈입니다.

1.5 제네릭: 〈자료형〉

- 제네릭(Generics)은 **입력되는 값의 타입을 자유롭게 사용**하기 위한 도구

➢ 설계 시 제네릭 만들기

- 예) interface 로 MutableList 라는 클래스를 만들어 **제너릭〈자료형〉**을 지정

```
public interface MutableList<E> {
    var list: Array<E>
    ...
}
```

- ✓ 클래스명 뒤에 **〈E〉**라고 되어 있는 부분에 String과 같은 특정 **타입이 지정되면** 클래스 **내부에 선언된 모든 E에 String이 타입으로 지정됨**
- ✓ 즉, 결과적으로 **var list: Array〈E〉**가 **var list: Array〈String〉**으로 변경되는 것임

1.5 제네릭: 〈자료형〉

➤ 구현 시 제네릭의 사용

- 앞에처럼 설계 시 제너릭을 만들어 놓고, 구현 시 컬렉션이나 배열에서 입력되는 값의 타입을 제한하기 위해 다음처럼 사용함

```
var list: MutableList〈제너릭〉 = mutableListOf(“월”, “화”, “수”)
```

컬랙션을 배울 때는 자동으로 “String”이 된다고 배웠어요.

거기에 이것만 추가됨

```
fun testGenerics() {
    // String을 제네릭으로 사용했기 때문에 list 변수에는 문자열만 담을 수 있습니다.
    var list: MutableList<String> = mutableListOf()
    list.add("월")
    list.add("화")
    list.add("수")
    // list.add(35) // <- 입력 오류가 발생합니다.
    // String 타입의 item 변수로 꺼내서 사용할 수 있습니다.
    for (item in list) {
        Log.d("Generic", "list에 입력된 값은 ${item}입니다.")
    }
}
```

String 제네릭이 적용된 빈 뮤터블리스트 생성

월, 화, 수 차례대로

설계 도구 전체 소스


```
package com.example.designtool

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log

class MainActivity: AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // 접근 제한자 테스트
        var child = Child()
        child.callVariables()

        // 부모 클래스 직접 호출해보기
        var parent = Parent()
        Log.d("Visibility", "Parent: 기본 제한자 defaultVal의 값은 ${parent.defaultVal}")
        Log.d("Visibility", "Parent: internalVal의 값은 ${parent.internalVal}")
    }
}
```

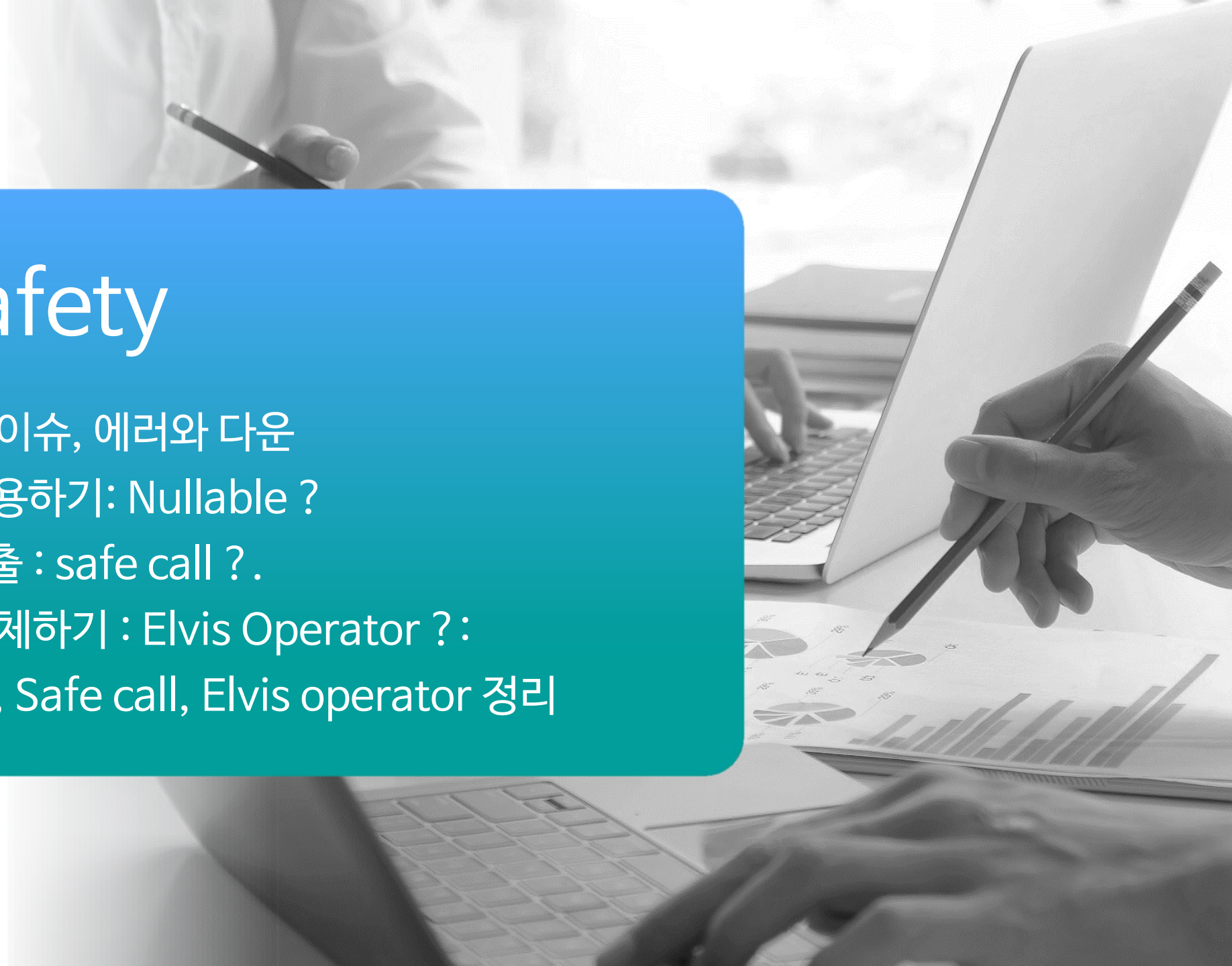
```
// 추상 클래스 설계
abstract class Animal {
    fun walk() {
        Log.d("abstract", "걷습니다.")
    }
    abstract fun move()
}
// 구현
class Bird: Animal() {
    override fun move() {
        Log.d("abstract", "날아서 이동합니다.")
    }
}
// 인터페이스 설계
interface InterfaceKotlin {
    var variable: String
    fun get()
    fun set()
}
// 구현
class KotlinImpl: InterfaceKotlin {
    override var variable: String = "init value"
    override fun get() {
        // 코드 구현
    }
    override fun set() {
        // 코드 구현
    }
}
```

코드의 동작과는 무관하지만 실습 예제로 적어둔 코드입니다.

```
// 접근 제한자 테스트를 위한 부모 클래스
open class Parent {
    private val privateVal = 1
    protected open val protectedVal = 2
    internal val internalVal = 3
    val defaultVal = 4
}
// 자식 클래스
class Child: Parent() {
    fun callVariables() {
        // privateVal은 호출이 안 됩니다.
        Log.d("Visibility", "Child: protectedVal의 값은 ${protectedVal}")
        Log.d("Visibility", "Child: internalVal의 값은 ${internalVal}")
        Log.d("Visibility", "Child: 기본 제한자 defaultVal의 값은 ${defaultVal}")
    }
}
```

결과

Child: protectedVal의 값은 2
Child: internalVal의 값은 3
Child: 기본 제한자 defaultVal의 값은 4
Parent: 기본 제한자 defaultVal의 값은 4
Parent: 기본 제한자 internalVal의 값은 3



2. Null Safety

2.1 Null 값의 이슈, 에러와 다운
2.2 null 값 허용하기: Nullable ?
2.3 안전한 호출 : safe call ? .
2.4 Null 값 대체하기 : Elvis Operator ? :
2.5 Nullable, Safe call, Elvis operator 정리

2.1 Null 값의 이슈, 에러와 다운

- null은 개발시 항상 이슈의 중심이었는데, null로 인해 프로그램 전체가 멈출 수 있기 때문

➢ null 값으로 프로그램이 멈추는 예)

```
class One {
    fun print() { // print() 메서드를 갖고있는 One 클래스
        Log.d("null_safety", "can you call me?")
    }
}

            …중간 생략…

var one: One    // One 클래스 타입의 one 변수 선언
if (1 > 2) {
    one = One() // One 클래스를 생성()해서 one 변수에 저장
}
one.print() // one 변수(=one클래스)의 print()메서드 실행
```

이 코드의 문제점: if 문이 거짓이므로
one 변수는 초기값이 들어가지 못하고 null상태인데
print()메서드를 호출했기 때문에
프로그램이 다운됨

2.1 Null 값의 이슈, 에러와 다운

- 앞에 코드는 물론 **오류**를 발생시켜 **컴파일**되지 **않음**
- 하지만 코드의 양이 많아 지면 **오류 발생없이** 컴파일되다가 프로그램이 **멈춰버리는** 경우가 많음
  → 이럴 경우 오류를 찾는 건 매우 어려워짐
- 이런 null 값 에러를 해결하기 위해 코틀린은 많은 공을 들였음
  → 이것을 **Null Safety**라고 함

2.2 null 값 허용하기: Nullable ?

- 코틀린의 기본 변수는 모두 null이 입력되지 않음
- null값을 입력하기 위해서는 변수를 선언할 때 타입 뒤에 ? (Nullable, 물음표)를 입력함

```
var variable: String?        //String 타입의 nullable 변수 variable
```

변수명 / 형 / null 값 허용

2.2 null 값 허용하기: Nullable ?

➢ 변수에 null 허용 설정하기

- 변수의 타입 뒤에 물음표를 붙이지 않으면
  null 값을 입력할 수 없음
- null 예외를 발생시키고 싶지 않다면 기본형으로 선언함

```
var nullable: String? // 타입 다음에 물음표를 붙여서 null 값을 입력할 수 있습니다.
nullable = null  //nullable 변수는 널 값을 허용했으니 에러 안 남, 대신 주의해서 사용해야 함

var notNullable: String
notNullable = null // 일반 변수에는 null을 입력할 수 없습니다.
                   // 그런데 null을 입력했으니 에러
```

2.2 null 값 허용하기: Nullable ?

➢ 함수의 파라미터에 null 허용 설정하기

- 함수의 파라미터에도 null 허용 여부를 설정할 수 있음
- 단, **함수의 파라미터가 null을 허용하려면**
  코드 내부에서 해당 파라미터의 **null 체크를 먼저 해야만 사용**할 수 있음

조금 복잡하죠~

```
fun nullParameter(str: String?) {
    if (str != null) {
        var length2 = str.length
    }
}
```

파라미터 str에 null이 허용되었기 때문에 함수 내부에서 null 체크를 하기 전에는 str을 사용할 수 없습니다.

✔ 이 코드에서처럼 조건문 if 를 통해 **str 파라미터**를 **null인지 아닌지 체크해야지만 사용**할 수 있음

2.2 null 값 허용하기: Nullable ?

➢ 함수의 리턴 타입에 null 허용 설정하기

- 함수의 리턴 타입에도 물음표를 붙여서 null 허용 여부를 설정할 수 있음

함수의 리턴타입

```
fun nullReturn(): String? {
    return null
}
```

✔ 함수의 리턴 타입에 Nullable이 지정되어 있지 않으면 null 값을 리턴할 수 없음

2.3 안전한 호출 : safe call ?.

- Safe Call (?.)을 사용해서 null 체크를 좀 더 간결하게 할 수 있음
- Nullable인 변수 뒤에 ?.을 붙이면
  해당 Nullable 변수가 null일 경우 ?. 뒤의 메소드를 실행하지 않고 null을 반환함

String형 Nullable 변수

함수를 Int 형으로 Nullable 리턴

Int 형 Nullable 변수

```
fun testSafeCall(str: String?): Int? {
    // str이 null이면 length를 체크하지 않고 null을 반환합니다.
    var resultNull: Int? = str?.length  // str이 null이면 length(길이) 프로퍼티를 실행하지 않고
                                        // null값을 resultNull 변수에 저장
    return resultNull;
}           // 저장된 null값을 반환
```

만약 (?.)을 사용하지 않고 (.)를 사용했는데,
str 변수가 null이라면 프로그램은 다운됨.
왜요?
null의 length를 구하라고 하니까 다운되죠~

2.4 Null 값 대체하기 : Elvis Operator ? :

- Elvis Operator (? :) 를 사용해서 **Nullable 변수가 null일 때 넘겨줄 기본값**을 설정할 수 있음

잠깐! 궁금증 해결과 쉽게 외우기!!

Q '? :'를 이용해서 null 대신 넘겨줄 값을 정한다는 건데 왜 **Elvis Operator라고 하죠?**

A 책에서는 가수 엘비스 프레슬리의 이모티콘에서 유래되어 Elvis Operator라고 하지만, **Elvis는** 신조어로 **(엘비스가 빌딩에서 투신한 것처럼) 갑자기 자취를 감추었다는 뜻**으로 사용됨

→ 즉, **null 값을 사라지게 하고 다른 값으로 대체**하겠다는 뜻

주의 클래스 이후 다른 문법들도 마찬가지지만, **Elvis Operator**라는 용어는 C++, PHP, C#, SQL 등 **다른 프로그램에서도 사용하는 용어**입니다. **하지만 용도가 다르니** 비교하면서 공부하면 안 되고, 새로 배운다고 생각하세요.

2.4 Null 값 대체하기 : Elvis Operator ? :

- 예) Safe call(? .) 뒤에 오는 프로퍼티에 다시 Elvis operator(? :)를 붙이고
  '0' 이라는 값을 표시함
  이렇게 호출하면 str 변수가 null일 경우 가장 뒤에 표시한 0을 반환

빠짐 빠짐

```
fun testElvis(str: String?): Int {
    // length 오른쪽에 ?:을 사용하면 null일 경우 ?: 오른쪽의 값이 반환됩니다.
    var resultNonNull: Int = str?.length?:0 // str이 null이면 null값을 넘기지 않고, 0을 넘김
    return resultNonNull
}
```

2.5 Nullable, Safe call, Elvis operator 정리

Nullable	표기법	선언하는 변수의 타입 다음에 ? 표기
	사용 목적	null을 입력받기 위해 사용
	사용 예	**var nullable : 타입?**
Safe call	표기법	선언된 변수의 이름 다음에?. 표기
	사용 목적	null일 때 ?. 다음에 나오는 속성이나 명령어를 처리하지 않고 결과를 null로 처리하기 위해 사용
	사용 예	**var result = 변수?.length** **변수?.프로퍼티?.something**
Elvis operator	표기법	선언된 변수의 이름 다음에 ?: 표기
	사용 목적	null일 때, null 대신 ?: 다음에 나오는 값으로 대체하기 위해 사용
	사용 예	**var result = 변수?:0** **변수?.프로퍼티?:0**

학습정리

패키지: package

- 패키지는 클래스와 소스 파일을 관리하기 위한 디렉터리 구조의 저장 공간
- 디렉터리가 **계층 구조로 만들어져 있으면 온점(.)으로 구분**해서 각 디렉터리를 모두 **나열**해 줌

학습정리

추상화 클래스: abstract class

- 프로그래밍을 하기 전 **개념 설계**를 하는 단계에서는 클래스의 이름과 클래스 안에 **있음 직한 기능**을 유추해서 **메서드 이름으로 먼저 나열**함
- 이것을 추상화라고하며 **abstract class 키워드**를 사용함

학습정리

인터페이스: interface

- interface는 설계단계에서 **실행코드 없이 메서드 이름만** 나열해 놓은 것
  - 설계단계에서 클래스에 **코드가 한 줄**이라도 필요하면 **추상화**
  - 설계단계에서 클래스에 코드 필요없이 **메서드 명만** 필요하면 **인터페이스**

학습정리

접근 제한자의 종류

접근 제한자	제한 범위	외우기 쉽게!
private	다른 파일에서 접근할 수 없음	접근금지
internal	같은 모듈에 있는 파일끼리만 접근 가능	내부모듈만
protected	private와 같으나 상속 관계라면 자식 클래스는 접근할 수 있음	자식만
public	제한 없이 모든 파일에서 접근 가능	접근허용

학습정리

제네릭: 〈자료형〉

- 제네릭(Generics)은 **입력되는 값의 타입을 자유롭게 사용**하기 위한 도구

학습정리

제네릭: 〈자료형〉

- 예) interface 로 MutableList 라는 클래스를 만들어 **제너릭〈자료형〉**을 지정

```
public interface MutableList<E> {
    var list: Array<E>
    ...
}
```

- 클래스명 뒤에 **〈E〉**라고 되어 있는 부분에 String과 같은 특정 **타입이 지정되면** 클래스 **내부에 선언된 모든 E에 String이 타입으로 지정됨**
- 즉, 결과적으로 var list: Array**〈E〉**가 var list: Array**〈String〉**으로 변경되는 것임

학습정리

Nullable, Safe call, Elvis operator 정리

Nullable	표기법	선언하는 변수의 타입 다음에 ? 표기
	사용 목적	null을 입력받기 위해 사용
	사용 예	**var nullable : 타입?**
Safe call	표기법	선언된 변수의 이름 다음에?. 표기
	사용 목적	null일 때 ?. 다음에 나오는 속성이나 명령어를 처리하지 않고 결과를 null로 처리하기 위해 사용
	사용 예	**var result = 변수?.length** **변수?.프로퍼티?.something**
Elvis operator	표기법	선언된 변수의 이름 다음에 ?: 표기
	사용 목적	null일 때, null 대신 ?: 다음에 나오는 값으로 대체하기 위해 사용
	사용 예	**var result = 변수?:0** **변수?.프로퍼티?:0**

학습평가

Q1 Q2 Q3

Q1 아래 코드의 결괏값은 무엇인가?

```
var nullable: String? = null
var size = nullable.length
Log.d("Nullable", "문자열의 길이 = $size")
```

학습평가

Q1 Q2 Q3

Q1 아래 코드의 결괏값은 무엇인가?

```
var nullable: String? = null
var size = nullable.length
Log.d("Nullable", "문자열의 길이 = $size")
```

정 답 동작하지 않음, 다운됨

해 설 nullable변수에 null이 있는데, length를 구하라고 했기 때문임

학습평가

Q1 Q2 Q3

Q2 아래 코드의 예상되는 결괏값은 무엇인가?

```
var nullable: String? = null
var size = nullable?.length
Log.d("Nullable", "문자열의 길이 = $size")
```

학습평가

Q2 아래 코드의 예상되는 결괏값은 무엇인가?

```
var nullable: String? = null
var size = nullable?.length
Log.d("Nullable", "문자열의 길이 = $size")
```

정 답 문자열의 길이 = null

해 설 nullable변수에 null이 있어서, length를 구하지 말고, null을 유지한채 null을 size 변수에 넘겨줌 (safe call)

학습평가

Q3 아래 코드의 예상되는 결괏값은 무엇인가?

```
var nullable: String? = null
var size = nullable?.length?:33
Log.d("Nullable", "문자열의 길이 = $size")
```

학습평가

Q1 Q2 Q3

Q3 아래 코드의 예상되는 결괏값은 무엇인가?

```
var nullable: String? = null
var size = nullable?.length?:33
Log.d("Nullable", "문자열의 길이 = $size")
```

정 답 문자열의 길이 = 33

해 설 nullable변수에 null이 있어서, length를 구하지 말고,
대체값 33 을 size에 넘겨줌 (Elvis Operator)

다음 시간

지연 초기화와 스코프 함수

모바일앱개발및응용 6주-2강

지연 초기화와 스코프 함수

학습 내용

1. 지연초기화
2. 스코프 함수

학습 목표

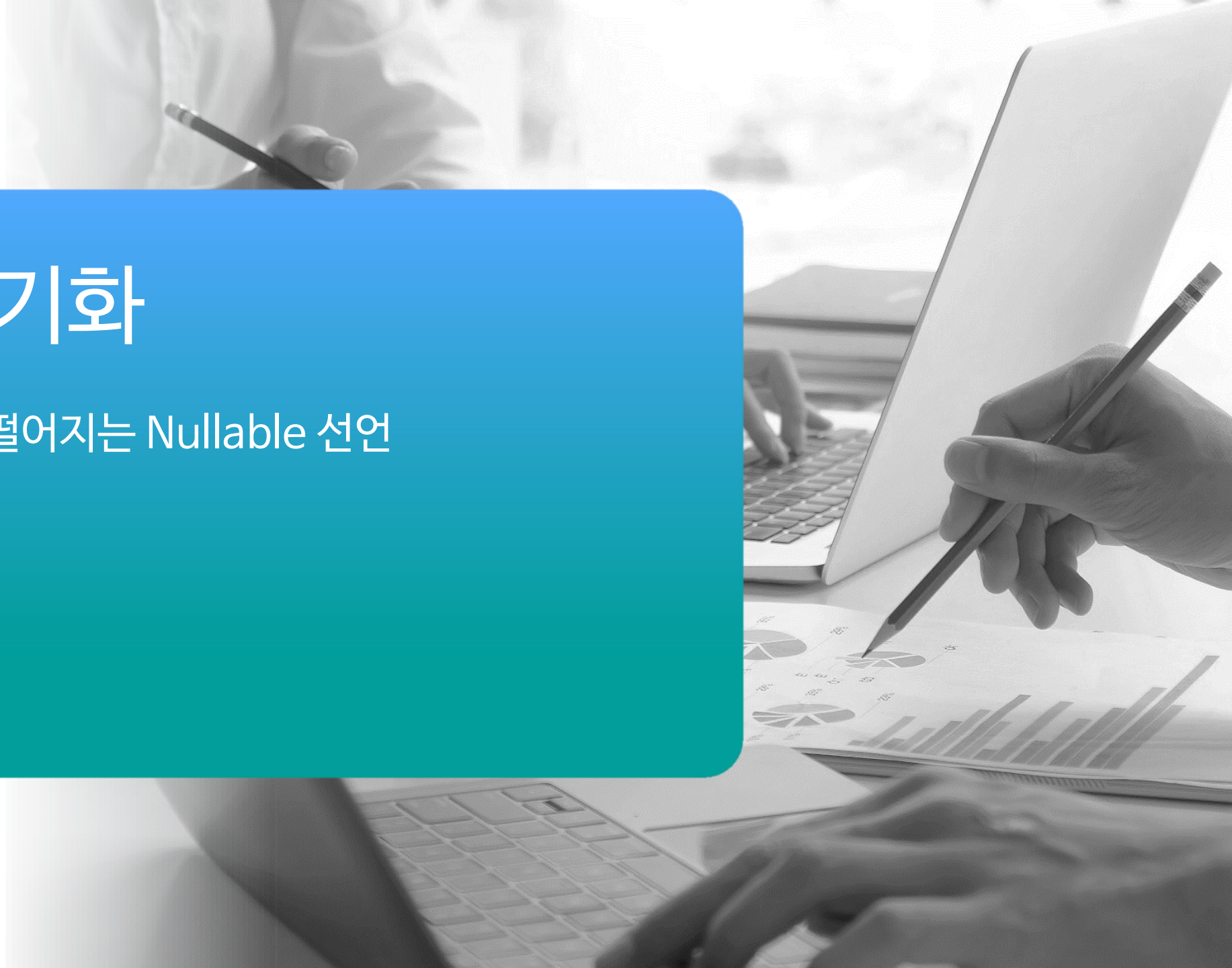
1. 초기화를 지연시키는 lateinit 와 by lazy 를 알고 적절히 사용할 수 있다.
2. run, apply, with(), let, also 스코프 함수를 이용하여 코드를 축약할 수 있고, 상황에 따라 적절한 함수를 선택해서 사용할 수 있다

1. 지연 초기화

1.1 가독성이 떨어지는 Nullable 선언

1.2 lateinit

1.3 by lazy



1.1 가독성이 떨어지는 Nullable 선언

- 클래스 안에서 **Nullable로 변수만 미리 선언**하고,
  초기화는 나중에(생성자 호출 후에)해야 하는 경우가 있음
- 예) 다음은 class 안 **name 변수에 초기값으로 null 값**을 입력해두고,
  class 안 **process() 메서드에서 값을 입력**하는 **Nullable의 일반적인 선언 방법임**

Nullable 변수 name

initial: 첫글자
initiate:
착수하다.
개시하다.

```
class Person {
    var name: String? = null
    init {
        name = "Lionel"
    }
    fun process() {
        name?.plus(" Messi") // name이 null일 때 (plus()를 실행하면 다운되므로 실행시키지 않고) null로 존재
        print("이름의 길이 = ${name?.length}") // 마찬가지로 null 로 존재
        print("이름의 첫 글자 = ${name?.substring(0,1)}") // 마찬가지로 null 로 존재
    }
}
```

→ 하지만 Safe Call(?.)이 남용되어 가독성이 떨어진다는 문제가 발생함

1.1 가독성이 떨어지는 Nullable 선언

- 앞에처럼 Nullable(?) 처리가 남용되는 것을 방지하기 위해 **코틀린은 초기화를 지연시킬 수 있음**
- **var**로 선언된 변수의 지연초기화는 **lateinit 키워드**를 사용
- **val**로 선언된 변수의 지연초기화는 **by lazy 키워드**를 사용

1.2 lateinit

- var로 선언된 변수에 lateinit을 사용하면 Safe Call(?.)을 쓰지 않을 수 있기 때문에 코드에서 발생할 수 있는 수많은 Nullable(?)을 방지하여 코드가 간단해짐
- 예) 앞에 코드에 lateinit 를 추가하고, ?. 과 ?을 제거하여 가독성이 좋아짐

```
class Person {
    lateinit var name: String // var로 선언된 name변수의 초기화는 나중에 할게
    init {
        name = "Lionel"
    }
    fun process() {
        name.plus(" Messi") // name변수가 초기화 된 후 plus()를 실행하면 됨
        print("이름의 길이 = ${name.length}")
        print("이름의 첫 글자 = ${name.substring(0,1)}")
    }
}
```

1.2 lateinit

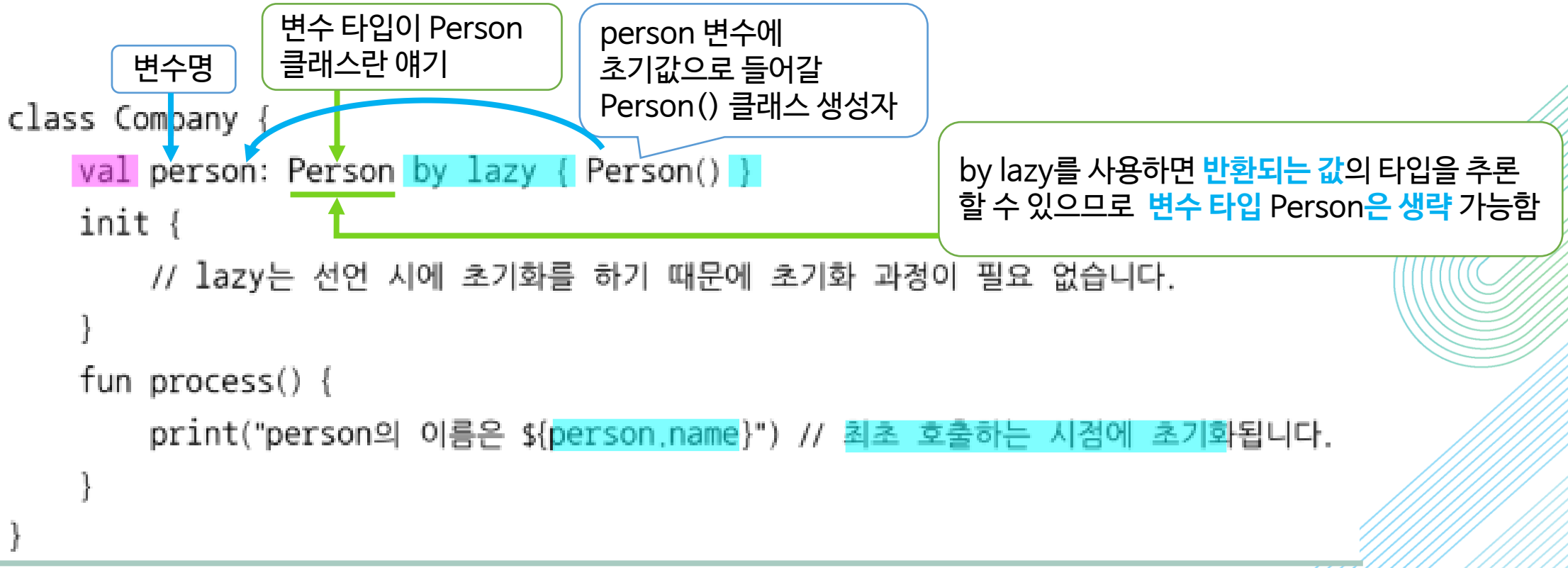
➢ lateinit 특징

- var로 선언된 클래스의 변수에만 사용할 수 있음
- null은 허용되지 않음
- 기본 자료형 Int, Long, Double, Float 등은 사용할 수 없음

lateinit으로 초기화를 지연시킨 변수는 **미리 선언만 해 놓은** 방식이므로,
초기화되지 **않은** 상태에서 메서드를 사용하면 **앱이 종료**됨
따라서 혹시라도 초기화되지 **않을** 상황이 발생할 수 있다면 **사용하지 않는 것이 좋음**

1.3 by lazy

- 읽기 전용 변수 **val**로 선언된 변수에 **by lazy** 키워드를 사용해서 초기화를 지연시킬 수 있음
- (lateinit와 형식이 다름) 먼저 **val로 변수를 선언**한 후 **코드 뒤쪽에 by lazy { 초기화할 값 }**을 씀




```
class Company {
    val person: Person by lazy { Person() }
    init {
        // lazy는 선언 시에 초기화를 하기 때문에 초기화 과정이 필요 없습니다.
    }
    fun process() {
        print("person의 이름은 ${person.name}") // 최초 호출하는 시점에 초기화됩니다.
    }
}
```

1.3 by lazy

➢ by lazy의 특징

- 초기화 코드를 함께 작성하는 val 변수에만 사용
- **by lazy{초기값}**은 해당 변수가 **최초로 호출되는 시점**에 초기화 됨
- 앞에 코드에서 Company 클래스가 초기화되더라도
  val person 변수는 초기화되지 않고,
  fun process()가 호출되어 **${person.name} 코드가 실행되는 순간 초기화**됨

지연 초기화는 말 그대로 **최초 호출되는 시점에 초기화** 작업이 일어나기 때문에
초기화하는 데 사용하는 **리소스가 너무 크면**(메모리를 많이 쓰거나 코드가 복잡한 경우)
전체 처리 속도에 나쁜 영향을 미칠 수 있음

미니퀴즈

Q1 다음 코드에서 ( ) 안에 들어가야 하는 키워드는 무엇인가?

```
lateinit ( ) school: School
```

정답 var

해설 lateinit 는 var 로 선언된 변수에만 사용할 수 있다.

미니퀴즈

Q2 다음 코드에서 ( ) 안에 들어가야 하는 키워드는 무엇인가?

```
( ) apple: Apple by lazy { Apple() }
```

정답 val

해설 by lazy는 val로 선언된 변수에만 사용할 수 있다.

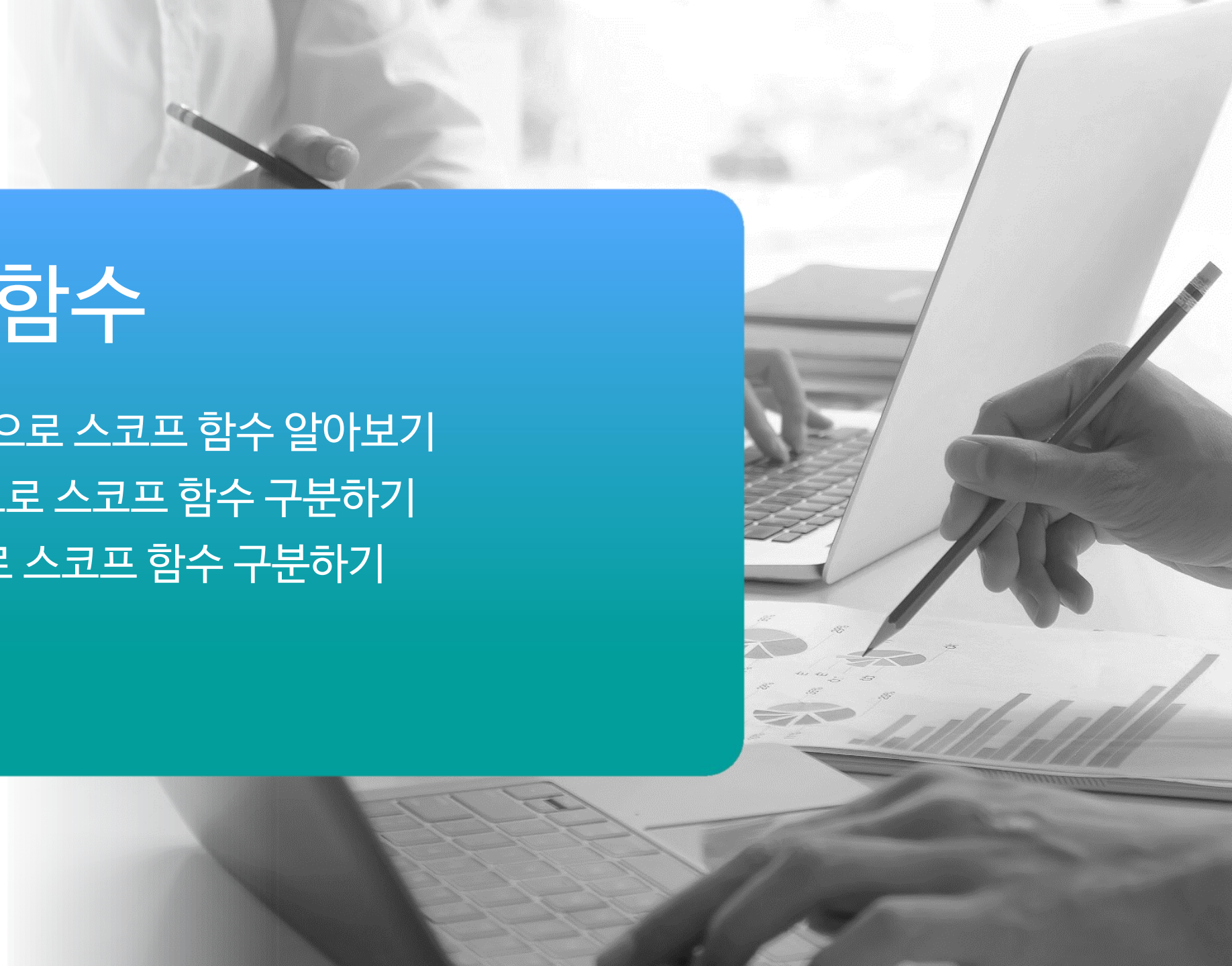
미니퀴즈

Q3 다음 코드를 실행하면 오류가 발생하는 이유는 무엇인가?

```
class Market {
    lateinit var candy: Candy
    init {
      Log.d("Candy", "사탕의 이름은 ${candy.name} 입니다.")
    }
}
```

정답 candy의 초기화 문제

해설 아직 candy 변수에 초기값이 없는데 candy.name를 실행하라고 하기 때문이다.



2. 스코프 함수

2.1 run과 let으로 스코프 함수 알아보기
2.2 this와 it으로 스코프 함수 구분하기
2.3 반환값으로 스코프 함수 구분하기

2. 스코프 함수

- 스코프 함수(Scope functions)는 **코드를 축약해서 표현할 수 있도록 도와주는 함수**로, 영역 함수 또는 범위지정 함수라고도 함
- 사용법은 ()괄호를 쓰지 않고 run, let 처럼 일종의 키워드같이 사용할 수 있음
- lateinit 과 함께 Safe Call 남용을 막아주는 역할도 함
- 스코프 함수에는 **run, let, apply, also, with()**가 있음

2. 스코프 함수

- 스코프 함수를 사용하지 않았을 때

```
val clark = Person()
clark.name = "Clark"
clark.age = 18
```

- 위 코드를 스코프 함수(apply)로 변경하면 코드가 간단해 짐

```
val peter = Person().apply {
    name = "Peter"
    age = 18
}
```

2.1 run과 let으로 보는 스코프 함수

- run과 let 함수는 **자신의 함수 블록({}, 스코프) 안**에서
  자신을 사용한 대상을 **this와 it로 대체**해서 사용 할 수 있음
- **run 함수**는 **this**를 사용하며 **생략**이 가능함
- **let 함수**는 **it**을 사용하며 **생략은 불가능**하고,
  it 대신 **다른 별명** 사용이 가능함

2.1 run과 let으로 보는 스코프 함수

run

- 자신의 블록 안에서 **자신을 사용할 대상**을 **this로** 사용할 수 있음
- **this를 생략**하고 메서드나 프로퍼티만 바로써서 사용할 수 있음

```
var list = mutableListOf("Scope", "Function")
list.run {
    val listSize = size
    println("리스트의 길이 run = $listSize")
}
```

this.size 대신에 this를 생략한 채로 도트 연산자(.) 없이 바로 사용할 수 있습니다.

= list.size
= this.size

2.1 run과 let으로 보는 스코프 함수

let

- 자신의 블록 안에서 **자신을 사용할 대상**을 **it으로** 사용할 수 있음
- **it을** 생략할 수는 **없지만**, target 등 **다른 이름으로** 바꿀 수 있음

```
var list = mutableListOf("Scope", "Function")
list.let { // it -> 생략된 형태. it -> 대신에 target -> 등으로 변경 가능합니다.
    val listSize = it.size // 모든 속성과 함수를 it.멤버로 사용할 수 있습니다.
    println("리스트의 길이 let = $listSize")
}
```

자신을 사용할 대상을
it이 아닌 다른 이름으로 변경하고
싶다면 "**별명 ->**" 을 쓰면 됨,
생략하면 그냥 it을 사용함

여기서 생략할 수 없다는 뜻,
사용 시 생략할 수 없다는 뜻
헷갈리지 말기!!

2.2 this와 it 으로 스코프 함수 구분하기

- 코틀린에서 제공하는 스코프 함수는
  run, apply, let, also with() 함수가 있음
- 이 함수들은 인자와 작동방식, 결과가 매우 비슷하기
  때문에 서로를 대체해서 사용이 가능함

2.2 this와 it 으로 스코프 함수 구분하기

this를 사용하는 run, apply, with() 스코프 함수

- run은 앞에서 했음
- 다음은 apply와 with()의 사용 예)
  - ✓ 스코프 함수 안에서 this로 사용되기 때문에 this는 생략이 가능하며, size 프로퍼티만 써서 사용할 수 있음
  - ✓ with는 다른 스코프 함수와 달리 "." 대신 함수처럼 ( )를 사용함

```
var list = mutableListOf("Scope", "Function")
list.apply {
    val listSize = size       // = this.size = list.size
    println("리스트의 길이 apply = $listSize")
}
with (list) {
    val listSize = size       // = this.size = list.size
    println("리스트의 길이 with = $listSize")
}
```

2.2 this와 it 으로 스코프 함수 구분하기

➢ it 을 사용하는 let, also 스코프 함수

```
var list = mutableListOf("Scope", "Function")
list.let { target ->
        // it을 target 등과 같이 다른 이름으로 변경 가능함// 생략하면 it-> 의 의미
    val listSize = target.size
        // target으로 변경했기 때문에 멤버 접근은 target.속성으로 함
    println("리스트의 길이 let = $listSize")
}
list.also {
    val listSize = it.size
    println("리스트의 길이 also = $listSize")
}
```

2.3 반환값으로 스코프 함수 구분하기

- 같은 this를 사용하는 run과 apply 함수라도 **반환되는 값이 다름**
- 같은 it을 사용하는 let과 also 함수라도 **반환되는 값이 다름**
- 결과값을 반환할 경우,
  스코프 함수가 종료되는 시점에(={ }이 끝나는 시점에)
  반환값이 다르기 때문에 서로 다른 역할을 하는 스코프 함수가 필요함

→ 즉, 동일하게 this를 사용하는 run과 apply함수라도
대입 연산자(=)를 사용해서 값을 반환할 경우에는
값이 달라지므로 상황에 맞게 사용해야 함

2.3 반환값으로 스코프 함수 구분하기

➢ 자기 자신을 반환하는 apply, also 스코프 함수

- apply를 사용하면 스코프 함수 안에서 코드가 모두 완료된 후 자기 자신을 되돌려줌

```
var list = mutableListOf("Scope", "Function")
val afterApply = list.apply {
    add("Apply")  // = list.add() = this.add() // 리스트에 'Apply' 추가됨
    count()       // = list.count() = this.count() // 개수 '3'을 갖고 있는 코드
}  // .apply 의 블록 끝, afterApply의 블록 아님 주의!
println("반환값 apply = $afterApply") // 자기자신인 리스트의 값이 출력됨

val afterAlso = list.also {
    it.add("Also") // it 은 생략이 안됨 // 리스트에 'Also' 추가
    it.count()                          // 개수 '4'를 갖고 있는 코드
} // .also 의 블록 끝, afterAlso의 블록 아님 주의!
println("반환값 also = $afterAlso")
```

결과

반환값 apply = [Scope, Function, Apply]
반환값 also = [Scope, Function, Apply, Also]

2.3 반환값으로 스코프 함수 구분하기

➢ 마지막 코드를 반환하는 let, run, with() 스코프 함수

```
var list = mutableListOf("Scope", "Function")
val lastCount = list.let {
    it.add("Run")
    it.count()
}
println("반환값 let = $lastCount")  // 마지막 it.count() 값인 '3' 출력

val lastItem = list.run {
    add("Run")
    get(size-1)
}
println("반환값 run = $lastItem") // 마지막 get(size-1) 값인 'Run' 출력

val lastItemWith = with(list) {
    add("With")
    get(size-1)
}
println("반환값 with = $lastItemWith") // 마지막 get(size-1) 값인 'With' 출력
```

결과

```
반환값 let = 3
반환값 run = Run
반환값 with = With
```

학습정리

가독성이 떨어지는 Nullable 선언

- 클래스 안에서 **Nullable로 변수만 미리 선언**하고,
  초기화는 나중에(생성자 호출 후에)해야 하는 경우가 있음

학습정리

가독성이 떨어지는 Nullable 선언

- 예) 다음은 class 안 **name 변수에 초기값으로 null 값**을 입력해두고, class 안 **process() 메서드에서 값을 입력**하는 **Nullable의 일반적인 선언 방법임**

Nullable 변수 name

```
class Person {
    var name: String? = null
    init {
        name = "Lionel"
    }
    fun process() {
        name?.plus(" Messi") // null일 때 (plus()를 실행하면 다운되므로 실행시키지 않고) null로 존재
        print("이름의 길이 = ${name?.length}") // 마찬가지로 null 로 존재
        print("이름의 첫 글자 = ${name?.substring(0,1)}") // 마찬가지로 null 로 존재
    }
}
```

→ 하지만 Safe Call(?.)이 남용되어 가독성이 떨어진다는 문제가 발생함

학습정리

lateinit

- var로 선언된 변수에 lateinit을 사용하면
  Safe Call(?.)을 쓰지 않을 수 있기 때문에 코드에서
  발생할 수 있는 수많은 Nullable (?)을 방지할 수 있음

학습정리

lateinit

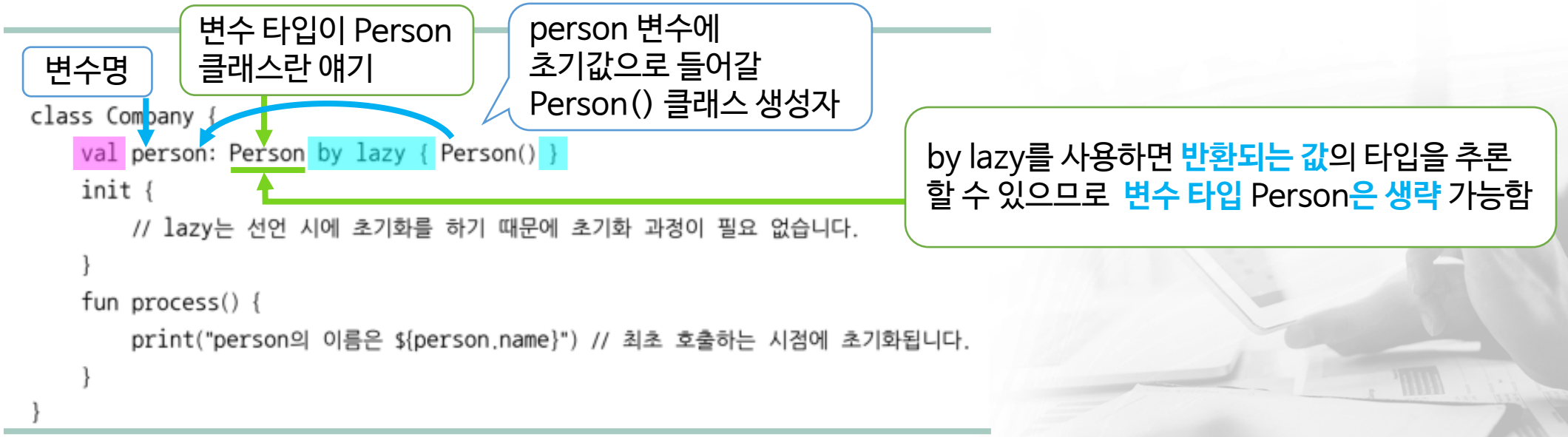
- 예) 코드 앞에 lateinit 를 추가하고, ? . 과 ? 을 제거하여 가독성이 좋아짐

```
class Person {
    lateinit var name: String // var로 선언된 name변수의 초기화는 나중에 할게
    init {
        name = "Lionel"
    }
    fun process() {
        name.plus(" Messi") // name변수가 초기화 된 후 plus()를 실행하면 됨
        print("이름의 길이 = ${name.length}")
        print("이름의 첫 글자 = ${name.substring(0,1)}")
    }
}
```

학습정리

by lazy

- 읽기 전용 변수 **val**로 선언된 변수에 **by lazy** 키워드를 사용해서 초기화를 지연시킬 수 있음
- (lateinit와 형식이 다름)먼저 **val로 변수를 선언**한 후 **코드 뒤쪽에 by lazy{ 초기화할 값 }**을 씀




```
class Company {
    val person: Person by lazy { Person() }
    init {
        // lazy는 선언 시에 초기화를 하기 때문에 초기화 과정이 필요 없습니다.
    }
    fun process() {
        print("person의 이름은 ${person.name}") // 최초 호출하는 시점에 초기화됩니다.
    }
}
```

학습정리

스코프 함수

- 스코프 함수(Scope functions)는 **코드를 축약해서 표현할 수 있도록 도와주는 함수**로, 영역 함수 또는 범위지정 함수라고도 함
- 스코프 함수에는 **run, let, apply, also, with()**가 있음

학습정리

스코프 함수 구분하기

- this 를 사용하는 run, apply, with() 스코프 함수
- it 을 사용하는 let, also 스코프 함수
- 자기 자신을 반환하는 apply, also 스코프 함수
- 마지막 코드를 반환하는 let, run, with() 스코프 함수

학습평가

Q1 Q2 Q3

Q1 마지막 실행 코드를 반환하는 스코프 함수는 무엇인가?

학습평가

Q1 Q2 Q3

Q1 마지막 실행 코드를 반환하는 스코프 함수는 무엇인가?

정 답 run, let, with()

해 설 run, let, with() 스코프 함수는 {}의 맨 마지막 코드를 반환함

학습평가

Q1 Q2 Q3

Q2 스코프 함수 안에서 it으로 사용되는 것 2개는 무엇인가?

학습평가

Q1 Q2 Q3

Q2 스코프 함수 안에서 it으로 사용되는 것 2개는 무엇인가?

정 답 let, also

해 설 let, also 스코프 함수는 자신을 사용할 대상을 it으로 표기하며, 사용시 it은 생략할 수 없으며, it 대신 다른 명칭으로 변경할 수 있다(별명 중 target이 많이 사용됨)

학습평가

Q1 Q2 Q3

Q3 다음 코드의 예상되는 결과는 무엇인가?

```
var fruits = mutableListOf("Apple", "Banana")

val afterFruits = fruits.let {
    it.add(“Melon”)
    it.count()
}
Log.d("결괏값: ", "$afterFruits")
```

학습평가

Q1 Q2 Q3

Q3 다음 코드의 예상되는 결과는 무엇인가?

```
var fruits = mutableListOf("Apple", "Banana")

val afterFruits = fruits.let {
    it.add(“Melon”)
    it.count()
}
Log.d("결괏값: ", "$afterFruits")
```

정 답 결괏값: 3

해 설 let은 블록의 마지막 코드를 반환하는 스코프 함수이다.

다음 시간

배치를 담당하는 레이아웃