LiveData를 사용한 반응형 프로그래밍

2016010873 박정욱

목차

- 반응형 프로그래밍(Reactive Programming)이란?
- 관찰자(Observer) 패턴이란?
- Android Lifecycle을 활용하여 변화 관찰하기
- 실제 적용 사례

반응형 프로그래밍이란?

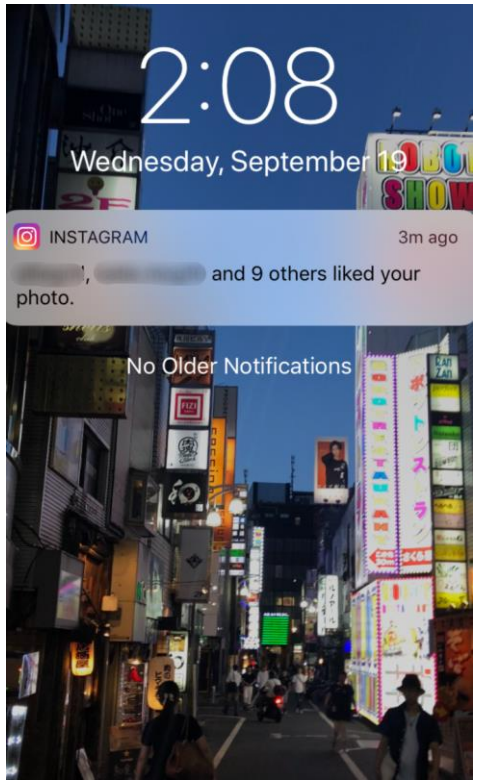
- **변화의 전파, 데이터 스트림**과 관련된 **선언적 프로그래밍**

- 변화의 전파?
  - 데이터가 변경될 때마다 이벤트 발생
  - 변경이 발생된 곳에서 데이터를 보내주는 Push 방식

- 데이터 스트림?
  - 시간에 지남에 따라 사용할 수 있게 되는 일련의 데이터 요소

- 선언형 프로그래밍?
  - ‘어떻게’ 할 것인지가 아닌, ‘무엇을’ 할 것인지에 초점을 맞추는 방식

변화의 전파



Push 알림

Jenkins

Travis CI

지속적 통합/지속적 제공 툴

선언형 프로그래밍

```cpp
int nums[10] = {
    1, 2, 3, 4, 5,
    6, 7, 8, 9, 10
};

for (int i = 0; i < 10; i++) {
    if (nums[i] % 2 == 0) {
        cout << nums[i] << "\n";
    }
}
```

명령형 프로그래밍
(컴퓨터처럼 생각하기)

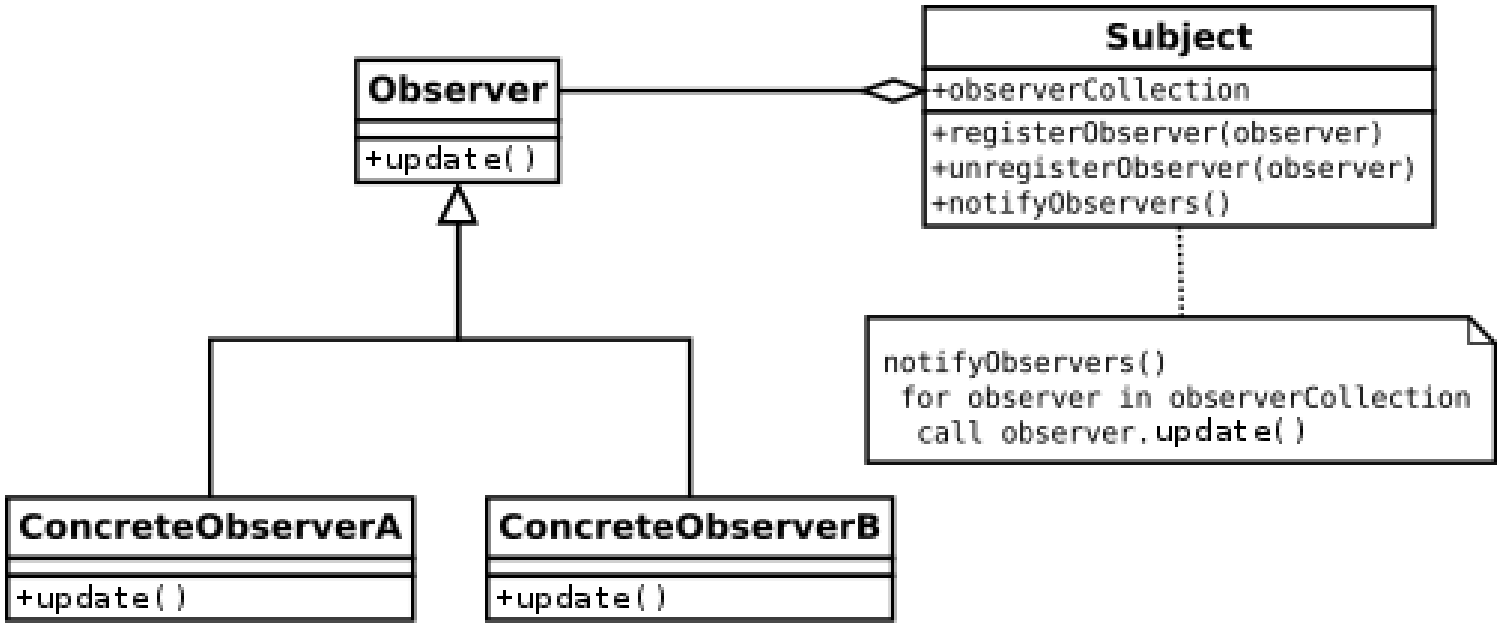
```
>>> nums = list(range(1, 11))
>>> nums
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>>
>>> list(filter(lambda x: x % 2 == 0, nums))
[2, 4, 6, 8, 10]
>>>
```

선언형 프로그래밍
(사람처럼 생각하기)

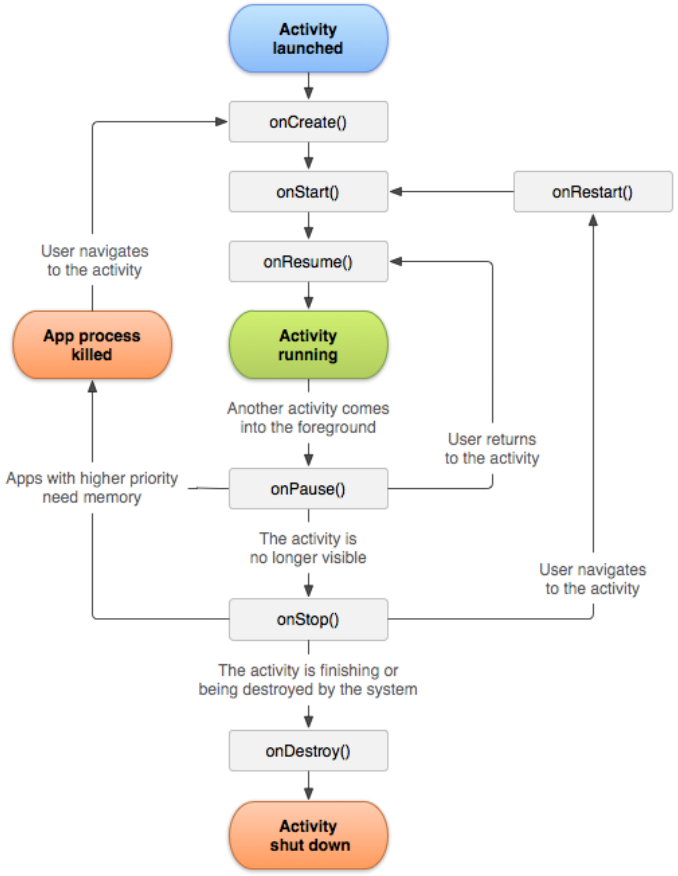
관찰자 패턴이란?

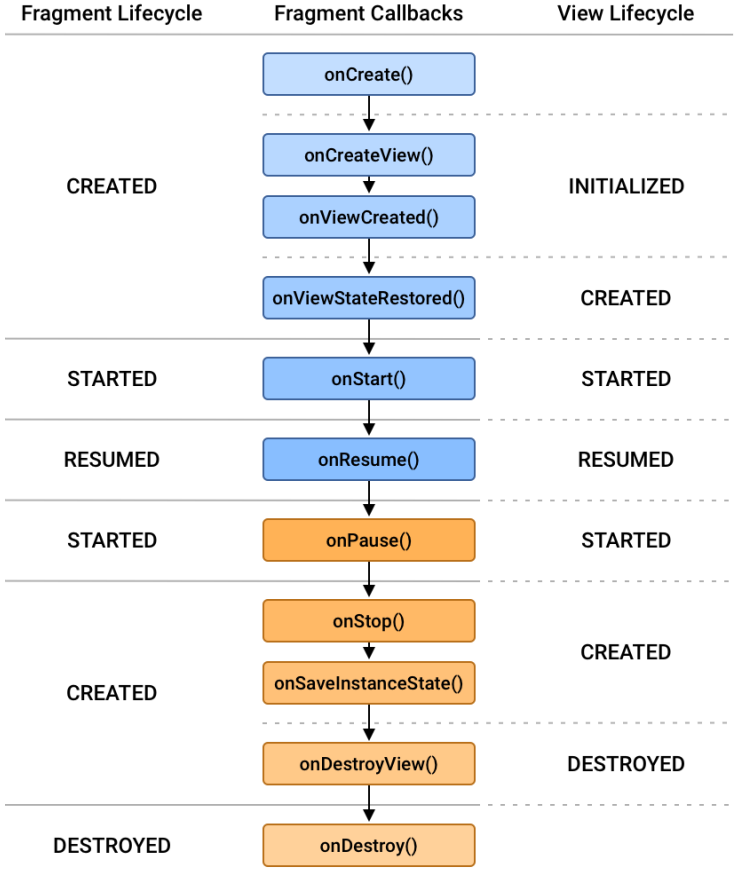
- 객체 사이에 일대다의 의존관계 정의
- 객체 상태 변화 시, 의존 객체들이 변화를 **통지**받아 자동으로 갱신
- MVC 패턴의 기반

관찰자 패턴이란?

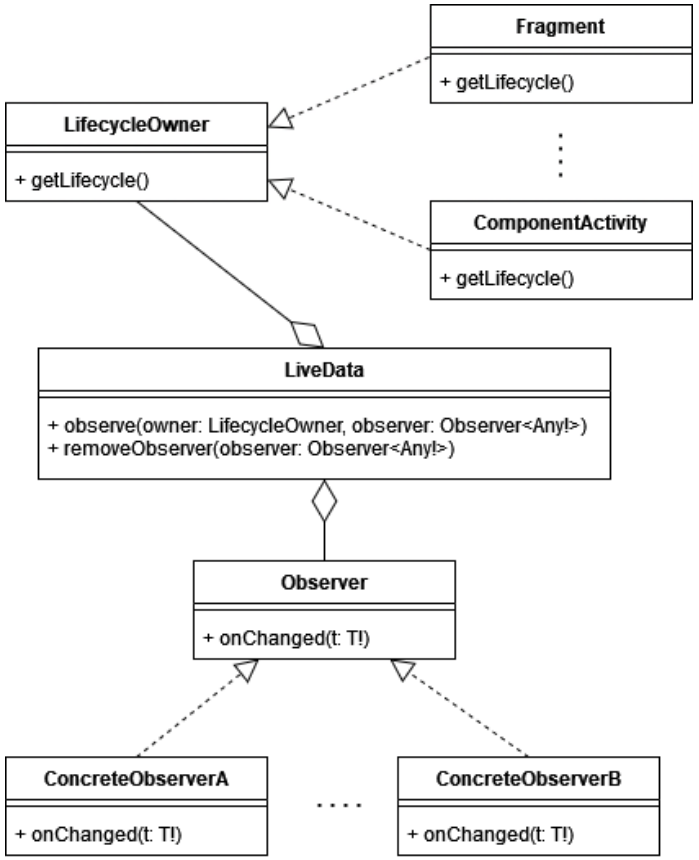

Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
Observer
+update()
notifyObservers()
for observer in observerCollection
call observer.update()
ConcreteObserverA
+update()
ConcreteObserverB
+update()

Android Lifecycle을 활용하여 변화 관찰하기





Android Lifecycle을 활용하여 변화 관찰하기



- LifecycleOwner
  - 구성요소의 생명주기를 제공
- Observer
  - 상태 변경 시 취할 동작 정의
- LiveData<T>
  - LifecyclerOwner, Observer를 제공받아 변화 관찰 가능

실제 적용 사례

```kotlin
@HiltViewModel
class RoutineDetailViewModel @Inject constructor(
    getUsedRoutineListUseCase: GetUsedRoutineListUseCase,
    private val insertRoutineUseCase: InsertRoutineUseCase
) : ViewModel() {
    private val _usedRoutines = getUsedRoutineListUseCase().onStart { emptyList<Routine>() }.asLiveData()
    val usedRoutines: LiveData<List<Routine>> get() = _usedRoutines

    fun insertNewRoutine(routine: Routine) {
```

ViewModel에서 LiveData 관리

Activity, Fragment 등에서 관찰

Kotlin Single Abstract Method 형식으로 정의

```kotlin
with(viewModel) {  this: RoutineDetailViewModel
    usedRoutines.observe(viewLifecycleOwner) { list ->
        adapters = list.map { ExpandableRoutineAdapter(it) }
    }
}
```

실제 적용 사례

```
viewModel.posts.observe(viewLifecycleOwner) {  it: List<Post>!
    val adapter = binding.communityMainList.adapter as CommunityMainAdapter
    adapter.setData(it)
}
```

ViewModel에서 LiveData 관리

Activity, Fragment 등에서 관찰

Kotlin Single Abstract Method 형식으로 정의

```
@HiltViewModel
class CommunityViewModel @Inject constructor(
    getPostListUseCase: GetPostListUseCase,
) : ViewModel() {
    private val _posts = getPostListUseCase().asLiveData()
    val posts: LiveData<List<Post>> get() = _posts
}
```

실제 적용 사례

```
viewModel.user.observe( owner: this) {  it: User?
    if (it != null) {
        loginSuccess()
    }
}
```

ViewModel에서 LiveData 관리

Activity, Fragment 등에서 관찰

Kotlin Single Abstract Method 형식으로 정의

```
@HiltViewModel
class LoginViewModel @Inject constructor(
    private val authWithFirebaseUseCase: AuthWithFirebaseUseCase,
    private val fetchUserUseCase: FetchUserUseCase,
    private val insertUserUseCase: InsertUserUseCase
) : ViewModel() {
    private val _user = MutableLiveData<User?>()
    val user: LiveData<User?> get() = _user

    fun signIn(idToken: String) {
```

정리

- 변화의 전파 + 데이터 스트림 + 선언적 프로그래밍 = Reactive!
- 데이터 변화에 실시간으로 반응하여 상태를 변경하는 데 쓰임
- LiveData를 이용하여 Android에서 Reactive 프로그래밍 가능
- LiveData를 이해하기 위해 관찰자 패턴, 수명 주기 확인