

# Serializable과 Parcelable의 차이점은 무엇인가요?

## 목차

1. 직렬화가 필요한 이유
2. Serializable
3. Parcelable
4. Parcel
5. 질문

# 직렬화가 필요한 이유

각각의 프로세스는 별도의 메모리를 갖기 때문에 프로세스 간 데이터 공유를 위해 별도의 기법이 필요하다.

IPC를 통해 우리는 다른 프로세스에 byte-stream을 전달할 수 있고, byte-stream 형태로 만들기 위해서는 직렬화가 필요하다.

cf. 별도의 기법 : IPC(Inter-Process Communication)

# Serializable

java.io.Serializable 인터페이스를 구현하여 직렬화, 역직렬화를 지원한다.

Android 의존성을 갖지 않는 표준 Java 인터페이스

별도로 구현해야 하는 메서드가 없는 마커 인터페이스, JVM 내부에서 직렬화, 역직렬화 해준다.  
JVM 내부에서 처리하기에 자바에서 제공하는 타입과 Serializable 타입 외의 커스텀 타입은 처리해주지 않는다.

리플렉션 기반으로 구현되어 있고 임시 객체들이 생성되며 GC 호출로 인해 성능에 영향을 줄 수 있다.

writeObject, readObject, readObjectNoData 메서드를 구현하여 커스텀할 수 있다.

DeepDive Point : 공식문서에서는 Effective Java의 직렬화 관련 내용을 확인하는 것을 제안한다.

# Serializable

사용 예시



```
class Person(val firstName: String, val lastName: String, val age: Int) : Serializable
```

# Parcelable

Android 의존성을 가지고 있으며 직렬화, 역직렬화를 지원한다.

리플렉션을 사용하지 않는다.

안드로이드 컴포넌트 내에서 프로세스 간 통신(IPC)에서 사용하는 것을 권장한다.

# Parcelable

## 사용 예시

```
class Person(  
    val firstName: String, val lastName: String, val age: Int  
) : Parcelable {  
    constructor(parcel: Parcel) : this(  
        parcel.readString().toString(),  
        parcel.readString().toString(),  
        parcel.readInt()  
    ) {  
    }  
  
    // 직렬화  
    override fun writeToParcel(parcel: Parcel, flags: Int) {  
        parcel.writeString(firstName)  
        parcel.writeString(lastName)  
        parcel.writeInt(age)  
    }  
  
    // 특별한 객체 타입이 포함되어 있는 지(특별한 객체가 있을 경우 그에 맞는 플래그를 반환)  
    override fun describeContents(): Int {  
        return 0  
    }  
  
    // 복원하거나, 배열을 생성하는 역할(역직렬화)  
    companion object CREATOR : Parcelable.Creator<Person> {  
        override fun createFromParcel(parcel: Parcel): Person {  
            return Person(parcel)  
        }  
  
        override fun newArray(size: Int): Array<Person?> {  
            return arrayOfNulls(size)  
        }  
    }  
}
```

# Parcelable

## parcelize

Parcelable 사용 시 직렬화, 역직렬화 메서드 구현에 대한 보일러 플레이트 코드를 줄이도록 도움을 준다.

### 사용하기 위한 설정



```
plugins {  
    id("kotlin-parcelize")  
}
```


build.gradle에 해당 플러그인을 추가해야 한다.



# Parcelable

## parcelize

### 사용 예시



```
import kotlinx.parcelize.Parcelize

@Parcelize
class User(val firstName: String, val lastName: String, val age: Int): Parcelable
```

@Parcelize 어노테이션을 기본 생성자에 사용하면 직접 구현 메서드를 작성하지 않아도 된다.

기본 생성자 외의 필드에는 적용되지 않는다.

기본 생성자에서 직렬화를 제외하고 싶다면 @IgnoredOnParcel 어노테이션을 붙인다.

커스텀 직렬화 로직이 필요하다면 companion class 내부에 Parceler<T> 를 구현하여 커스텀 할 수 있다.

# Parcel

안드로이드에서 컴포넌트간의 고성능 IPC 전송을 가능하게 하는 컨테이너 클래스

마샬링(직렬화), 언마샬링(역직렬화)하여 안드로이드의 IPC 통신에서 데이터를 전송할 수 있게 한다.

Parcelable은 객체를 직렬화하여 Parcel을 통해 전달할 수 있도록 한다.

Parcelable을 구현한 객체는 Parcel에 쓰고 복원할 수 있어 컴포넌트 간 복잡한 데이터 전달에 적합하다.

# 질문

? 안드로이드에서 Serializable과 Parcelable의 차이점은 무엇이며, 일반적으로 컴포넌트 간 데이터 전달에 Parcelable이 선호되는 이유는 무엇인가요?

Serializable은 Java 표준 직렬화 인터페이스고, Parcelable은 안드로이드 전용 인터페이스다.

Serializable은 리플렉션 기술을 사용하여 직렬화, 역직렬화 과정에서 임시 객체들을 생성하고, 이로 인해 GC 부담이 커져 성능이 저하될 수 있는 반면, Parcelable은 직렬화, 역직렬화 과정을 개발자가 작성하고 리플렉션을 사용하지 않아 불필요한 객체 생성이 되지 않기에 성능에 미치는 영향이 비교적 적다. 그리고, 구현이 불편하다는 단점이 있었지만 플러그인을 제공하며 어노테이션을 활용해 단점을 보완했다.

Parcelable이 안드로이드 전용 인터페이스고, 리플렉션을 사용하지 않아 속도가 빠르다는 장점이 컴포넌트 간 데이터 전달에 선호하는 이유가 될 수 있다.

Serializable은 Java 표준이므로 Android 외의 환경과 호환성이 필요한 경우 선호할 수 있을 것 같다.

# 참고

<https://developer.android.com/reference/java/io/Serializable>

<https://developer.android.com/kotlin/parcelize#kts>

<https://medium.com/jaesung-dev/android-%EC%A7%81%EB%A0%AC%ED%99%94%EC%99%80-%EC%97%AD%EC%A7%81%EB%A0%AC%ED%99%94-18fd04f1c0ed>