Reactive UI

Pontonhíd az RxJava és Data Binding között

Agenda

- Reactive UI programing
- RxJava
- Data Binding
- The “pontoon bridge”
- Example

Reactive UI

Reactive UI programing

- **Reactive principles applied to UI development**
  - Declarative programming paradigm concerned with data streams and the propagation of change
  - Programming with asynchronous data streams
- **Everything is a stream, everything can be observed**

RxJava

RxJava

- **Asynchronous and event-based**
- **Based on Observer design pattern**
  - The stream is the subject (or the “Observable”) being observed
  - “Listening” to the stream is called subscribing
  - Functions defined for listening for data are the observers
- **Compose streams declaratively**

Data Binding

Data Binding

- **What is Data Binding?**
  - **Data binding** is a general technique that binds data sources from the provider and consumer together and synchronizes them. -Wikipedia
- **Android Data Binding**
  - Make findViewById obsolete
  - Encourages the separation of UI logic and business logic
  - Binds data sources to UI elements in a declarative format
  - Uses Observer pattern for synchronization
    - `android.databinding.Observable`
    - `ObservableField`

The “Pontoon bridge”

The “Pontoon bridge”

- Declarative UI
- Data streams in a declarative format
- Observer pattern is implemented in both
- Easy to connect them

The “Pontoon bridge”

- **From RxJava to Data Binding**
  - RxObservableField

```
class RxObservableField<T>(source: Flowable<T>) : ObservableField<T>() {
    private val source: Flowable<T> = source
            .doOnNext { newValue -> set(newValue) }
            .doOnError { e -> Log.e("RxObservableField", "onError in source Flowable", e) }
            .onErrorResumeNext(Flowable.empty())
            .share()

    private val disposables = hashMapOf<OnPropertyChangedCallback, Disposable>()

    override fun addOnPropertyChangedCallback(callback: OnPropertyChangedCallback) {
        super.addOnPropertyChangedCallback(callback)

        disposables[callback] = source.subscribe()
    }

    override fun removeOnPropertyChangedCallback(callback: OnPropertyChangedCallback) {
        super.removeOnPropertyChangedCallback(callback)

        val disposable = disposables.remove(callback)

        if (disposable?.isDisposed == false) {
            disposable.dispose()
        }
    }
}
```

The “Pontoon bridge”

- **From RxJava to Data Binding**
  - RxObservableField
  - toField extension function

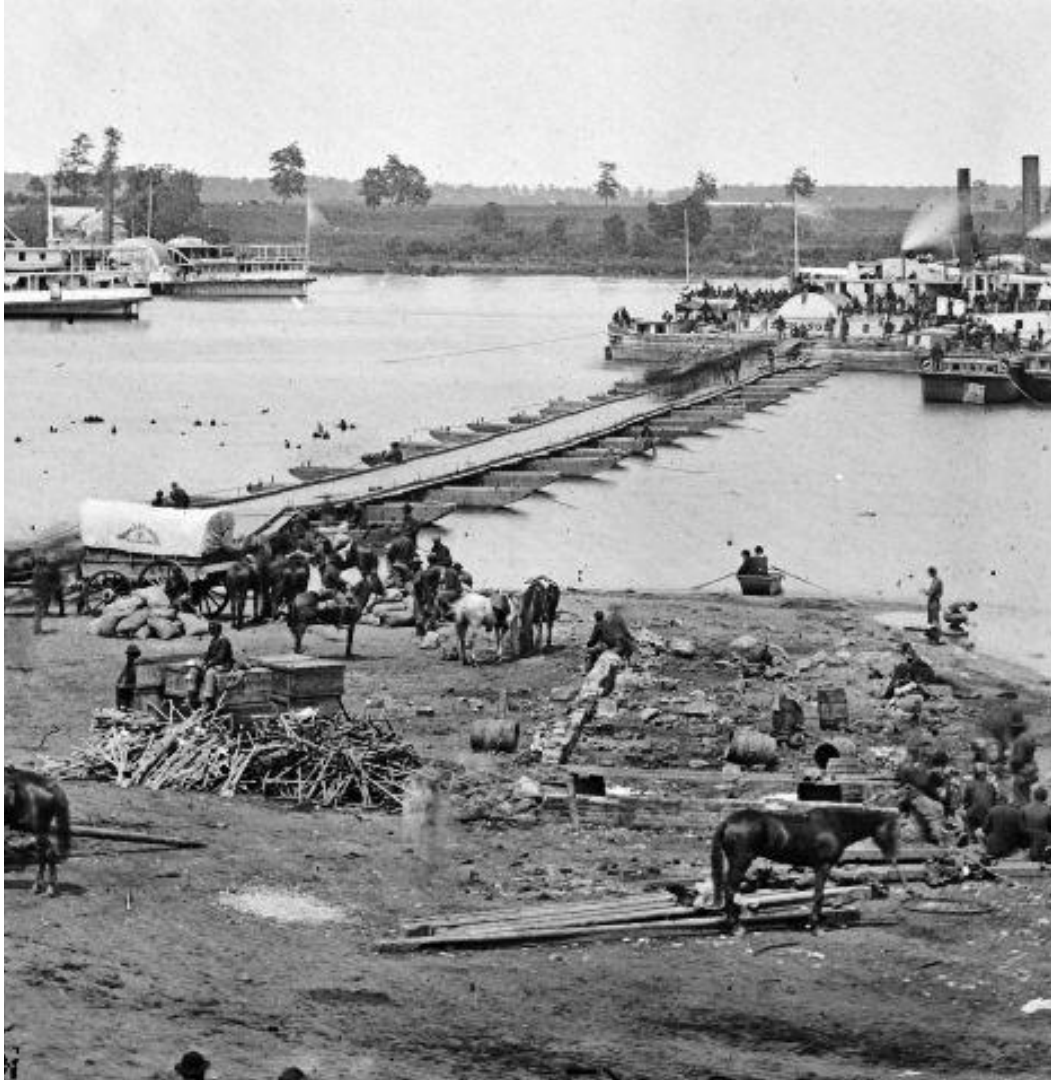
```kotlin
fun <T> Flowable<T>.toField(): ObservableField<T> {
    return RxObservableField(this)
}

fun <T> Single<T>.toField(): ObservableField<T> {
    return RxObservableField(this.toFlowable())
}

fun <T> Maybe<T>.toField(): ObservableField<T> {
    return RxObservableField(this.toFlowable())
}

fun <T> Observable<T>.toField(): ObservableField<T> {
    return RxObservableField(this.toFlowable(BackpressureStrategy.BUFFER))
}
```

The “Pontoon bridge”

- **From RxJava to Data Binding**
  - RxObservableField
  - toField extension function
- **From Data Binding to RxJava**
  - toFlowable extension function

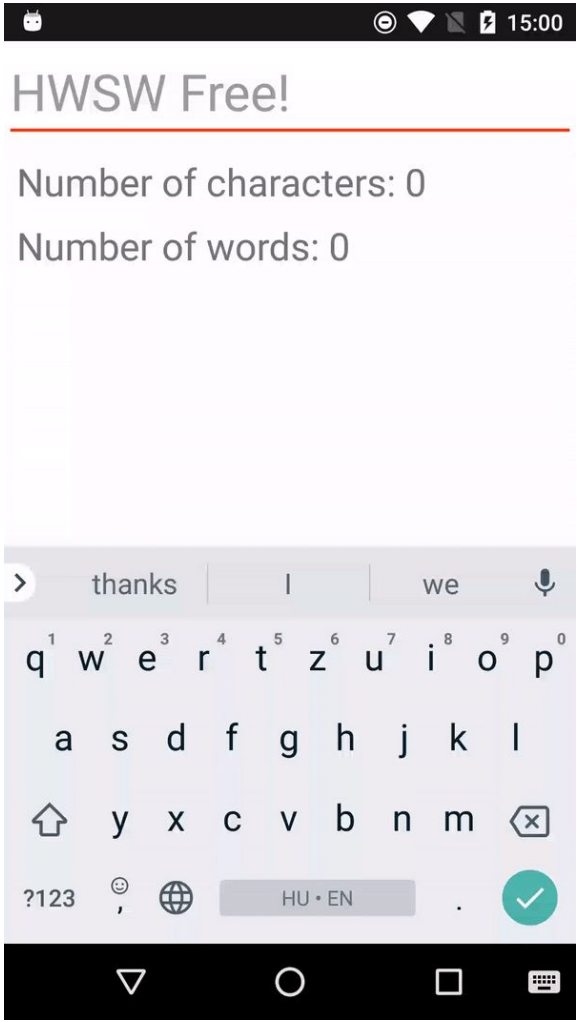
```
fun <T> ObservableField<T>.toFlowable(): Flowable<T> {
   return Flowable.create({ emitter ->
       get()?.let { nextValue -> emitter.onNext(nextValue) }

       val callback = object : OnPropertyChangedCallback() {

           override fun onPropertyChanged(sender: Observable?, propertyId: Int) {
               get()?.let { nextValue -> emitter.onNext(nextValue) }
           }
       }

       addOnPropertyChangedCallback(callback)
       emitter.setCancellable { removeOnPropertyChangedCallback(callback) }
   }, BUFFER)
}
```

Example



```
class ExampleViewModel : ViewModel() {

    val query = ObservableField<String>("")

    val charCount = query.toFlowable()
            .map { it.length }
            .toField()

    val wordCount = query.toFlowable()
            .subscribeOn(Schedulers.computation())
            .map { it.toLowerCase() }
            .map { it.replace(Regex("[^\\W]'"), " ") }
            .map { it.trim() }
            .map { it.split(Regex("\\s+")) }
            .map { it.size }
            .toField()
}
```

```
<EditText
    ...
    android:text="@={vm.query}"/>

<TextView
    ...
    android:text="@{@string/character_count_format(vm.charCount)}"/>

<TextView
    ...
    android:text="@{@string/word_count_format(vm.wordCount)}"/>
```



LiveData

LiveData

- **Data Binding support since 3.1.0-alpha06**

```
val binding: ActivityExampleBinding = DataBindingUtil
        .setContentView(this, R.layout.activity_example)
binding.setLifecycleOwner(this)
```

- **From RxJava to Data Binding**
  - RxLiveData
  - toLiveData extension function
- **From Data Binding to RxJava**
  - toFlowable extension function

```
class RxLiveData<T>(source: Flowable<T>) : LiveData<T>() {
    private val source: Flowable<T> = source
            .doOnError { e -> Log.e("RxLiveData", "onError in source Flowable", e) }
            .onErrorResumeNext(Flowable.empty())

    private var disposable: Disposable? = null

    override fun onActive() {
        disposable = source.subscribe { newValue -> postValue(newValue) }
    }

    override fun onInactive() {
        disposable?.dispose()
    }
}
```

```kotlin
fun <T> Flowable<T>.toLiveData(): LiveData<T> {
   return RxLiveData(this)
}

fun <T> Single<T>.toLiveData(): LiveData<T> {
   return RxLiveData(this.toFlowable())
}

fun <T> Maybe<T>.toLiveData(): LiveData<T> {
   return RxLiveData(this.toFlowable())
}

fun <T> Observable<T>.toLiveData(): LiveData<T> {
   return RxLiveData(this.toFlowable(BackpressureStrategy.BUFFER))
}
```

```kotlin
fun <T> LiveData<T>.toFlowable(): Flowable<T> {
    return Flowable.create({ emitter ->
        val observer = Observer<T> { newValue ->
            newValue?.let { emitter.onNext(it) }
        }

        observeForever(observer)
        emitter.setCancellable { removeObserver(observer) }
    }, BUFFER)
}
```

Thank you for your attention.

Attila Polacsek
Android Expert | Supercharge

attila.polacsek@supercharge.io


SUPER
CHARGE