

# Achieving Fully Reactive Android Apps

# What is reactive programming?

- Programming with asynchronous data streams
- Propagating change and reacting to it

# Why should we care about reactive programming?

- Keeps your data up to date
- Unifies the asynchronous and the synchronous
- Represents a more honest modeling of our applications

# You don't need Rx to do reactive programming

```
myButton.setOnClickListener {  
    // react to the click event  
}
```

**But it helps**

# How does RxJava fit into the picture?

- Provides a way to interact with and manipulate that stream of data
- **Can** help facilitate writing reactive programs.

**“Can”**

# How have we been using RxJava?

- Making asynchronous network calls
- Making asynchronous database calls
- Using it as a replacement for AsyncTasks and other tools



# Non reactive Rx example

```
interface MyNetworkService {  
    fun fetchMyModelObjects(): Single<List<MyModelObject>>  
}  
  
class MyViewModel(val networkService: MyNetworkService) {  
    fun fetchTheStuff() {  
        networkService.fetchMyModelObjects()  
            .subscribeOn(Schedulers.io())  
            .observeOn(AndroidSchedulers.mainThread())  
            .subscribe { modelObjects -> updateMyUi(modelObjects) }  
    }  
}
```

How do we make our  
apps reactive?

Introducing:

Todo or not todo

## Todo or not todo

Figure out what reactive programming is



Panic



Ask myself why I would ever choose to engage in public speaking



Pretend deadlines don't exist



Oh my god I really need to write this talk



Figure out how slides work



# Lots of familiar code

```
data class Todo(  
    val text: String,  
    val addedDate: Date,  
    val isDone: Boolean  
)
```

# Lots of familiar code

```
interface TodoService {  
    fun fetchTodos(): Single<List<Todo>>  
    fun saveUpdatedTodo(todo: Todo): Completable  
}
```

# Lots of familiar code

```
interface TodoListView {  
    fun setListItems(todos: List<Todo>)  
}
```

# Lots of familiar code

```
class TodoListViewModel(private val view: TodoListView): ViewModel() {  
    private val repo = TodoServiceImpl()  
    private val disposables = CompositeDisposable()  
    private var todos = mutableListOf<Todo>()  
}
```



# Lots of familiar code

```
class TodoListViewModel(private val view: TodoListView): ViewModel() {  
    private val repo = TodoServiceImpl()  
    private val disposables = CompositeDisposable()  
    private var todos = mutableListOf<Todo>()  
  
    fun start() {  
        repo.fetchTodos()  
            .map { todos -> todos.filter { !it.isDone } }  
            .subscribeOn(Schedulers.io())  
            .observeOn(AndroidSchedulers.mainThread())  
            .subscribe { todos ->  
                this.todos = todos.toMutableList()  
                view.setListItems(todos)  
            }  
            .addTo(disposables)  
    }  
}
```

# Lots of familiar code

```
class TodoListViewModel(private val view: TodoListView): ViewModel() {
    private val repo = TodoServiceImpl()
    private val disposables = CompositeDisposable()
    private var todos = mutableListOf<Todo>()

    fun start() {
        repo.fetchTodos()
            .map { todos -> todos.filter { !it.isDone } }
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe { todos ->
                this.todos = todos.toMutableList()
                view.setListItems(todos)
            }
        .addTo(disposables)
    }

    fun todoCompleted(todo: Todo) {
        repo.saveUpdatedTodo(todo.copy(isDone = true))
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe {
                this.todos.remove(todo)
                view.setListItems(todos.toMutableList())
            }
        .addTo(disposables)
    }
}
```

**Woof, we're done  
right?**

# Actually...

```
class TodoListViewModel(private val view: TodoListView): ViewModel() {
    private val repo = TodoServiceImpl()
    private val disposables = CompositeDisposable()
    private var todos = mutableListOf<Todo>()

    fun start() {
        repo.fetchTodos()
            .map { todos -> todos.filter { !it.isDone } }
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe { todos ->
                this.todos = todos.toMutableList()
                view.setListItems(todos)
            }
        .addTo(disposables)
    }

    fun todoCompleted(todo: Todo) {
        repo.saveUpdatedTodo(todo.copy(isDone = true))
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe {
                this.todos.remove(todo)
                view.setListItems(todos.toMutableList())
            }
        .addTo(disposables)
    }

    fun todoUpdated(updatedTodo: Todo) {
        if (!todos.any { it.text == updatedTodo.text }) {
            todos.add(updatedTodo)
            view.setListItems(todos.toMutableList())
        }
    }
}
```

# And don't forget about...

```
class TodoListFragment: Fragment(), TodoListView, TodoToggledCallback, TodoUpdatedCallback {
    private val adapter = TodoAdapter(this)
    private lateinit var viewModel: TodoListViewModel

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_characters_list, container, false)
        view.list.adapter = adapter
        view.list.layoutManager = LinearLayoutManager(view.context)
        return view
    }

    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        viewModel = buildViewModel {
            TodoListViewModel(this)
        }
    }

    override fun onStart() {
        super.onStart()
        viewModel.start()
    }

    override fun todoUpdated(todo: Todo) {
        viewModel.todoUpdated(todo)
    }

    override fun todoToggled(todo: Todo) {
        viewModel.todoCompleted(todo)
        (activity as? TodoUpdatedCallback)?.todoUpdated(todo.copy(isDone = !todo.isDone))
    }

    override fun setListItems(todos: List<Todo>) {
        adapter.submitList(todos)
    }
}
```

**And again for the  
completed list screen**

**Transitioning to a  
more reactive flow**

# A reactive repository

```
class TodoRepository(private val service: TodoService) {  
}
```



# A reactive repository

```
class TodoRepository(private val service: TodoService) {  
    fun fetchTodos(): Observable<List<Todo>> {  
        return service.fetchTodos().toObservable()  
    }  
}
```

# A reactive repository

```
class TodoRepository(private val service: TodoService) {  
    private val disposables = CompositeDisposable()  
    private val todoSubject =  
BehaviorSubject.create<List<Todo>>()  
    val todoObservable = todoSubject.hide()  
  
    fun fetchTodos() {  
        service.fetchTodos()  
            .subscribeOn(Schedulers.io())  
            .observeOn(AndroidSchedulers.mainThread())  
            .subscribe(todoSubject::onNext)  
            .addTo(disposables)  
    }  
}
```

# A reactive repository

```
class TodoRepository(private val service: TodoService) {  
    private val disposables = CompositeDisposable()  
    private val todoSubject =  
BehaviorSubject.create<List<Todo>>()  
    val todoObservable = todoSubject.hide()  
  
    fun fetchTodos() {  
        service.fetchTodos()  
            .subscribeOn(Schedulers.io())  
            .observeOn(AndroidSchedulers.mainThread())  
            .subscribe(todoSubject::onNext)  
            .addTo(disposables)  
    }  
  
    fun saveUpdatedTodo(todo: Todo): ??? {  
        ???  
    }  
}
```

# A reactive repository

```
class TodoRepository(private val service: TodoService) {  
  
    ...  
  
    fun saveUpdatedTodo(todo: Todo) {  
        service.saveUpdatedTodo(todo)  
            .subscribeOn(Schedulers.io())  
            .subscribe {  
                val updatedTodo = todo.copy(isDone = !todo.isDone)  
                val todos = todoSubject.value ?: emptyList()  
                val list = todos.map { it.id == todo.id  
                                     updatedTodo else it }  
                todoSubject.onNext(list)  
            }  
            .addTo(disposables)  
    }  
}
```

**Don't forget to kick  
the stream off!**

# A reactive repository

```
class TodoRepository(private val service: TodoService) {  
  
    ...  
  
    init {  
        fetchTodos()  
    }  
  
    private fun fetchTodos() {  
        service.fetchTodos()  
            .subscribeOn(Schedulers.io())  
            .observeOn(AndroidSchedulers.mainThread())  
            .subscribe(todoSubject::onNext)  
            .addTo(disposables)  
    }  
  
    ...  
}
```

# Consuming the reactive API

# A reactive ViewModel

```
class TodoListViewModel(todoRepository: TodoRepository) :  
    ViewModel() {  
    private val disposables = CompositeDisposable()  
  
    override fun onCleared() {  
        disposables.dispose()  
    }  
}
```



# A reactive ViewModel

```
class TodoListViewModel(todoRepository: TodoRepository) :  
    ViewModel() {  
    private val disposables = CompositeDisposable()  
  
    init {  
        todoRepository  
            .todoObservable  
            .map { todos -> todos.filter { !it.isDone } }  
            .subscribeOn(Schedulers.io())  
            .subscribe { ??? }  
            .addTo(disposables)  
    }  
  
    override fun onCleared() {  
        disposables.dispose()  
    }  
}
```

# Communicating with the UI

# A reactive ViewModel

```
class TodoListViewModel(todoRepository: TodoRepository) :  
    ViewModel() {  
    private val disposables = CompositeDisposable()  
    val listItemsLiveData = MutableLiveData<List<Todo>>()  
  
    init {  
        todoRepository  
            .todoObservable  
            .map { todos -> todos.filter { !it.isDone } }  
            .subscribeOn(Schedulers.io())  
            .subscribe(listItemsLiveData::postValue)  
            .addTo(disposables)  
    }  
  
    override fun onCleared() {  
        disposables.dispose()  
    }  
}
```

**What about user  
input?**

# A reactive ViewModel

```
class TodoListViewModel(
    todoRepository: TodoRepository,
    todoToggled: Observable<Todo>
) : ViewModel() {
    ...
    init {
        todoRepository
            .todoObservable
            .map { todos -> todos.filter { !it.isDone } }
            .subscribeOn(Schedulers.io())
            .subscribe(listItemsLiveData::postValue)
            .addTo(disposables)

        todoToggled
            .subscribeOn(Schedulers.io())
            .subscribe(repository::saveUpdatedTodo)
            .addTo(disposables)
    }
    ...
}
```

# Wrapping up the UI

```
class TodoListFragment: Fragment() {  
    private val adapter = TodoAdapter()  
  
    override fun onActivityCreated(savedInstanceState: Bundle?)  
    {  
        super.onActivityCreated(savedInstanceState)  
        val viewModel = buildViewModel {  
            TodoListViewModel(  
                repository,  
                adapter.todoToggledObservable  
            )  
        }  
  
        viewModel.listItemsLiveData.observe(  
            this,  
            observer(adapter::submitList)  
        )  
    }  
}
```

# Key takeaways

- Building reactive android applications is fun in a bun
- It's up to you to design reactive APIs within your application
- Use Subjects to transition from imperative to reactive programs
- Aim to setup your objects such that all logic could be determined in the initializer