Reactive Programming In Swift

Professor Larry Heimann
Carnegie Mellon University
Information Systems

What is Reactive Programming?

Reactive
Programming

- Is a declarative programming paradigm that is based on the idea of asynchronous event processing and data stream
    - Declarative: its main focus is on “what to solve” in contrast to an imperative style where the main focus is “how to solve”.
    - Asynchronous processing means that the processing of an event does not block the processing of other events.
- It provides a way to handle and react to data streams as they occur, rather than explicitly programming the steps to execute

Reactive Programming is all about data streams

Reactive programming arose from the problem of how to handle streams of data from a variety of sources.

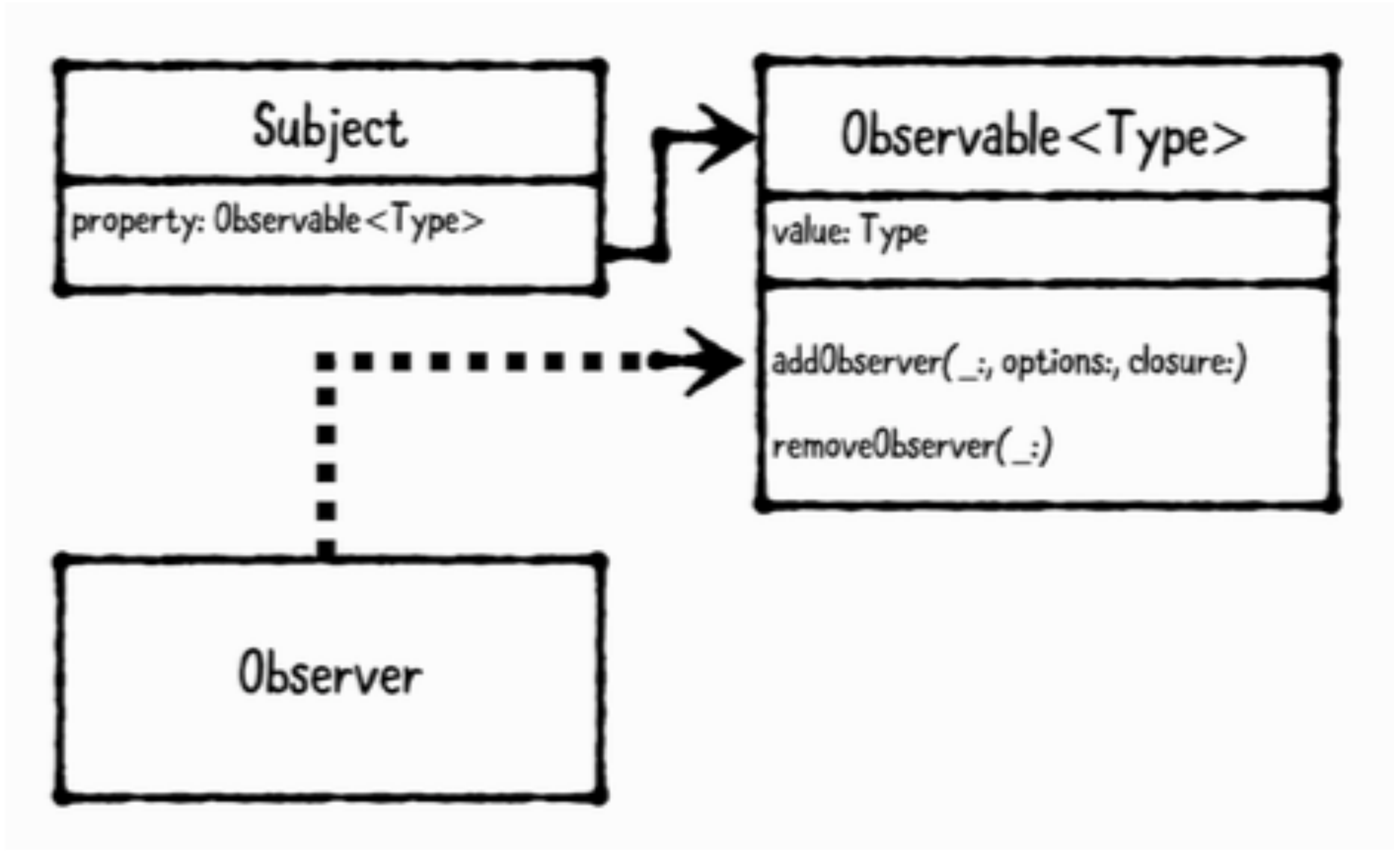
- From external APIs
- From user clicks in a game
- WebSocket communications
- Form inputs and validation
- Timers and calendar alerts

Key concept in reactive programming:

The Observer Pattern

What is the observer pattern?

- Allows other objects to observe events and get notifications when state changes.
- Useful when state is regularly changing and/or many other objects need to know when state has changed.


Subject
property: Observable<Type>
Observable<Type>
value: Type
addObserver(_:, options:, closure:)
removeObserver(_:)
Observer

Review: simple Ruby example of observers

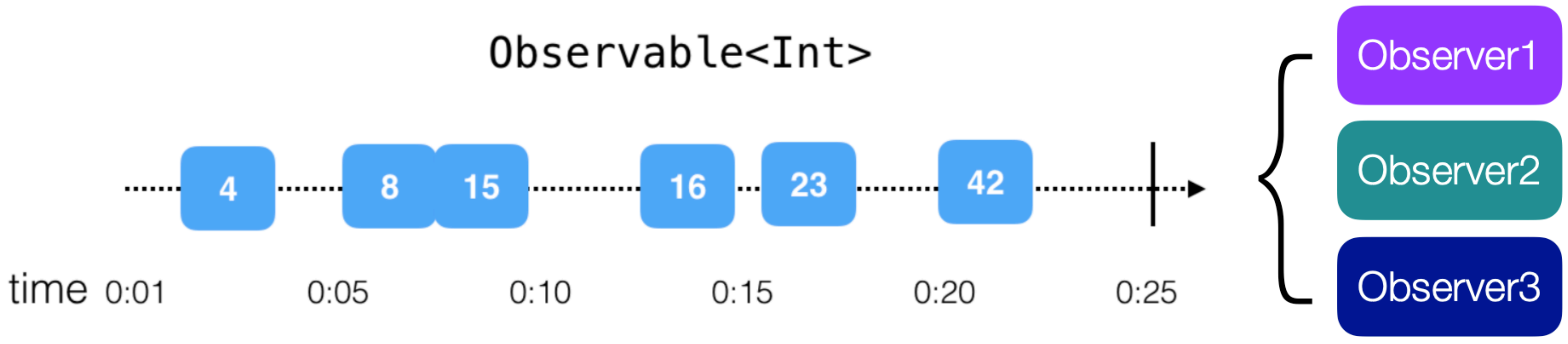
Rx Programming in Swift

- RxSwift can be found on GitHub: https://github.com/ReactiveX/RxSwift
- Materials from lectures drawn heavily from RxSwift book by Pillet, et al.
- Strongly recommended for students wanting to learn more about reactive programming in general and RxSwift in particular

Up to date for iOS
Xcode 9 & RxSwift
RxSwift
Reactive Programming with Swift
SECOND EDITION
By the raywenderlich.com Tutorial Team
Florent Pillet, Junior Bontognali, Marin Todorov & Scott Gardne

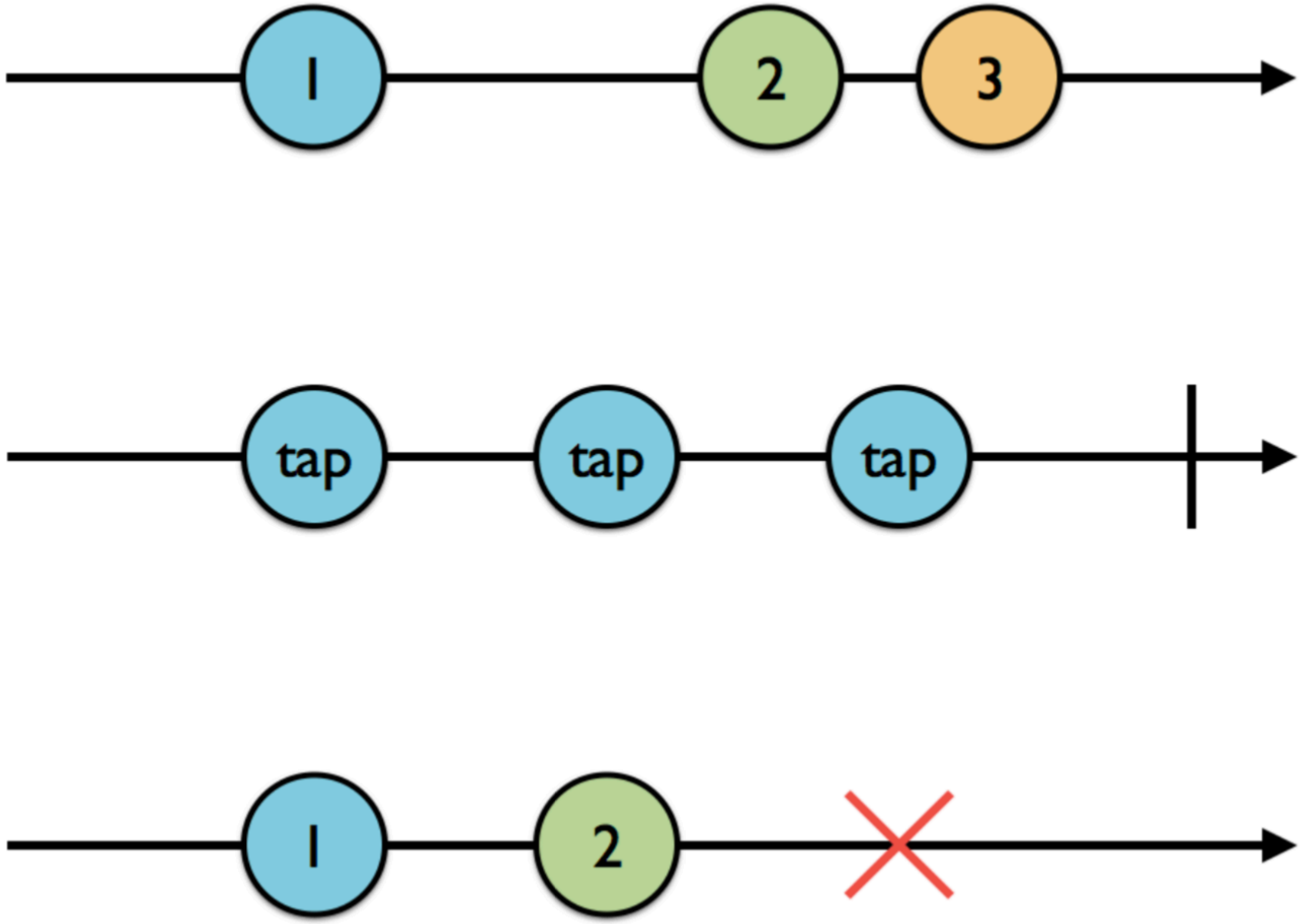
Key to RxSwift: Observables

The **Observable<T>** class provides the foundation of Rx code: the ability to asynchronously produce a sequence of events that can “carry” an immutable snapshot of data **T**. In the simplest words, it allows classes to subscribe for values emitted by another class over time.



At its heart, an observable is just a sequence

Simple examples of observables



Observable traits

- **Singles** will emit either a `.success(value)` or `.error` event. `.success(value)` is actually a combination of the `.next` and `.completed` events. This is useful for one-time processes that will either succeed and yield a value or fail, such as downloading data or loading it from disk.

- A **Completable** will only emit a `.completed` or `.error` event. It doesn't emit any value. You could use a completable when you only care that an operation completed successfully or failed, such as a file write.

- And **Maybe** is a mashup of a Single and Completable. It can either emit a `.success(value)`, `.completed`, or `.error`. If you need to implement an operation that could either succeed or fail, and optionally return a value on success, then Maybe is your ticket.

Time to look at code…

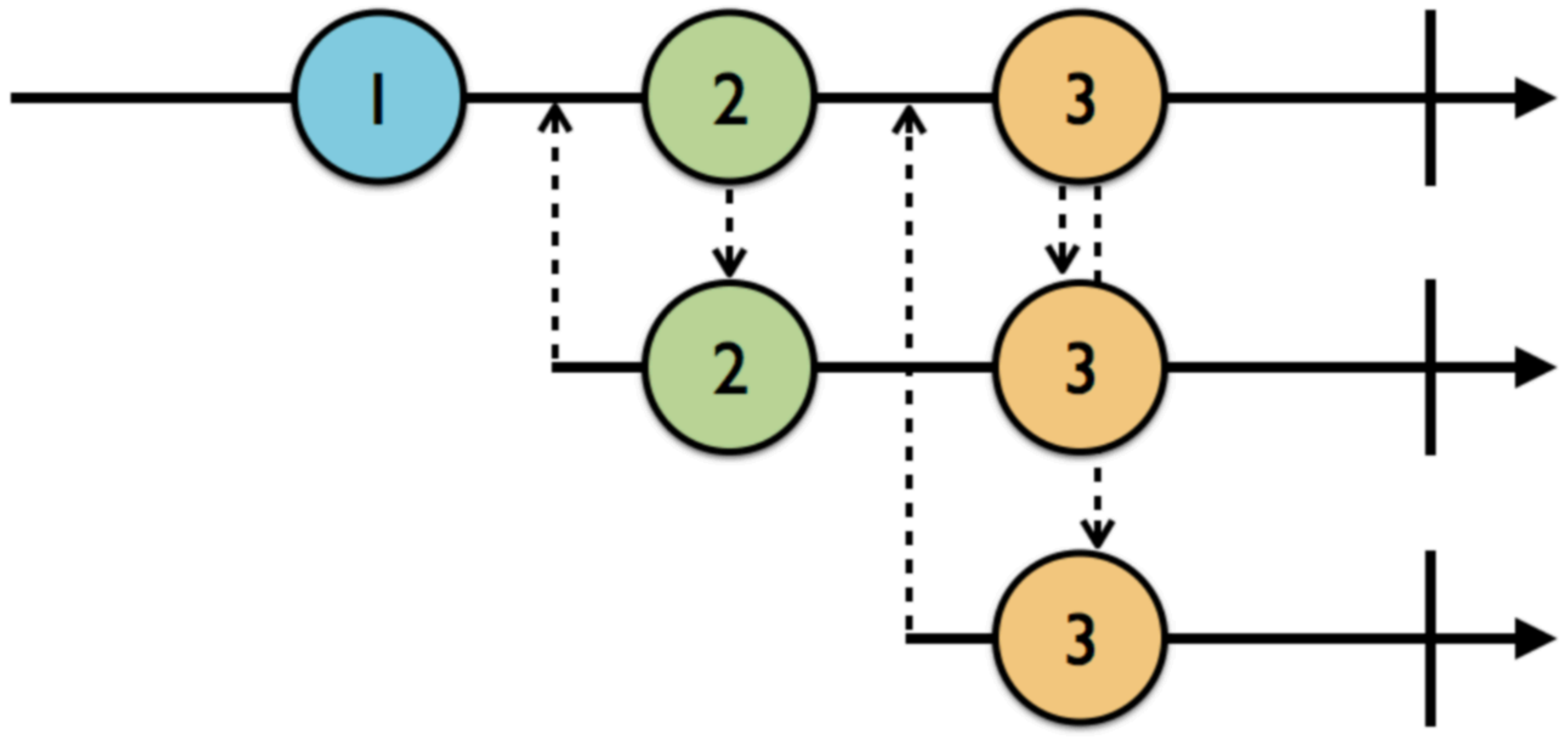
Next key concept: Subjects

A Subject is an object that can be both an observable and an observer. There are four types of subjects:

- **PublishSubject** — starts empty and only emits new elements to subscribers.
- **BehaviorSubject** — starts with an initial value and replays it or the latest element to new subscribers.
- **ReplaySubject** — initialized with a buffer size and will maintain a buffer of elements up to that size and replay it to new subscribers.
- **Variable** — wraps a BehaviorSubject, preserves its current value as state, and replays only the latest/initial value to new subscribers.

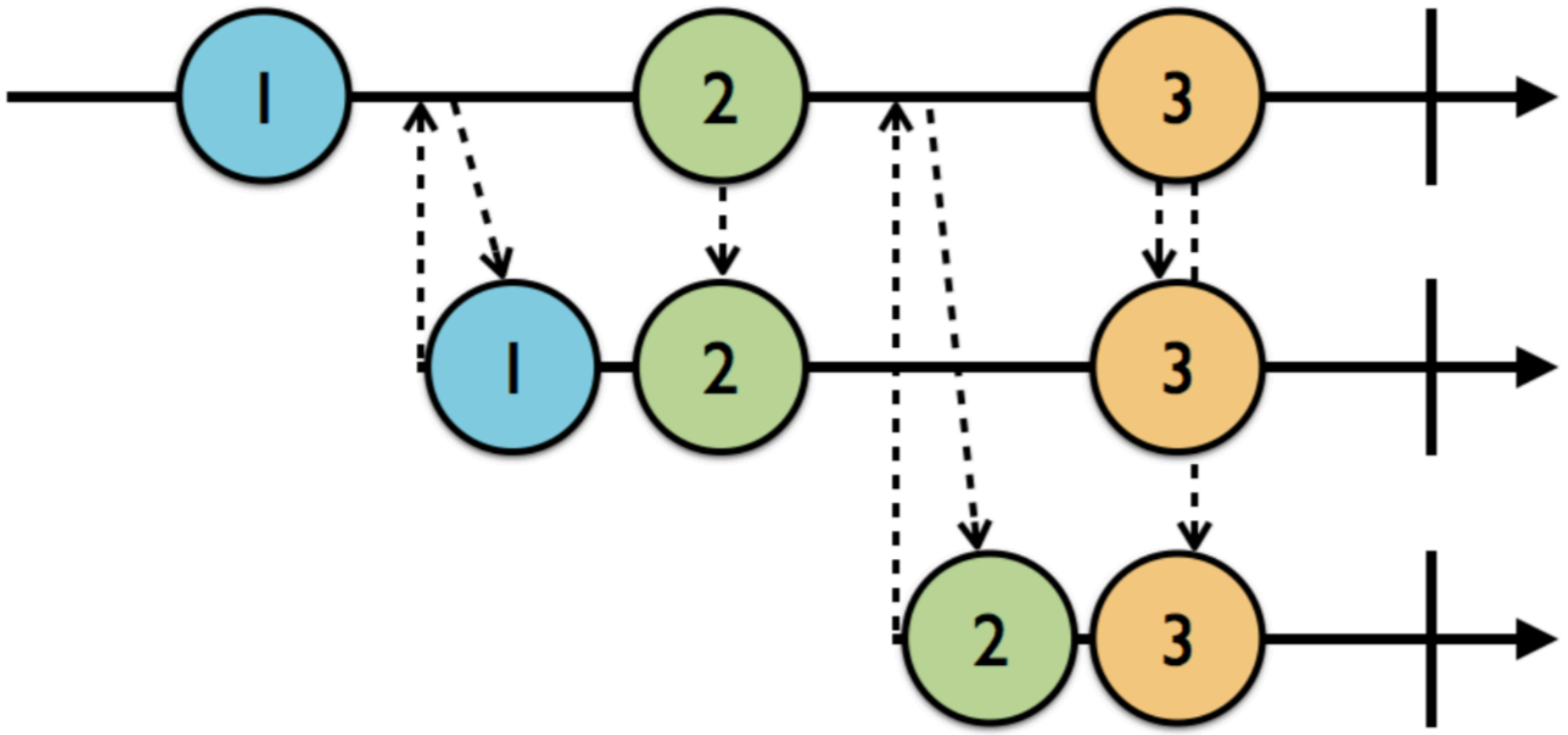
PublishSubjects

- The first subject emits three events and completes
- The second subject subscribes after first event, but gets the other two
- The third subject subscribes after the second event. First subject notifies both observers of the last event



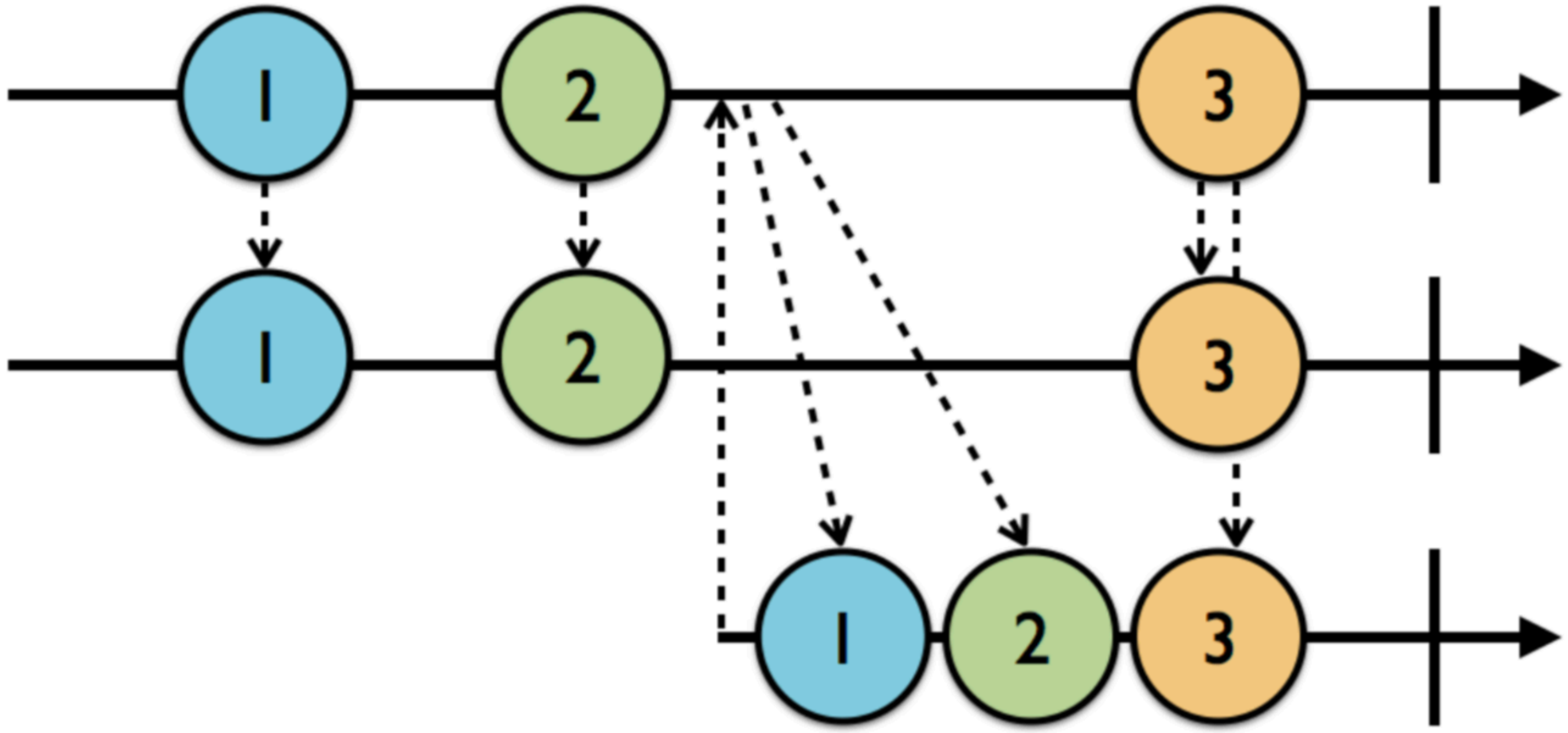
BehaviorSubjects

- The first subject emits three events and completes
- The second subject subscribes after first event, but gets the first event immediately and notified of the other events when they occur
- The third subject subscribes after the second event. It gets the prior event immediately (but not the original) and other events when they occur



ReplaySubjects

- The first subject emits three events and completes and notifies the second subject as they occur
- The third subject subscribes after the second event. It gets all the prior events immediately and is notified of other events when they occur



Code (and quiz) time …