What is Reactive Programming?

Principles of Reactive Programming

Martin Odersky

Changing Requirements...

	10 years ago	*Now*
Server nodes	10's	1000's
Response times	seconds	milliseconds
Maintenance downtimes	hours	none
Data volume	GBs	TBs $\rightarrow$ PBs.

... Need New Architectures

Previously: Managed servers and containers.

Now: Reactive applications

- event-driven
- scalable
- resilient
- responsive

Reactive

[Merriam Webster] reactive: *"readily responsive to a stimulus"*.

- React to events *(event-driven)*
- React to load *(scalable)*
- React to failures *(resilient)*
- React to users *(responsive)*

Event-Driven

Traditionally: Systems are composed of multiple threads, which communicate with shared, synchronized state.

- Strong coupling, hard to compose.

Now: Systems are composed from loosely coupled event handlers.

- Events can be handled asynchronously, without blocking.

Scalable

An application is *scalable* if it is able to be expanded according to its usage.

- scale up: make use of parallelism in multi-core systems
- scale out: make use of multiple server nodes

Important for scalability: Minimize shared mutable state.

Important for scale out: Location transparency, resilience.

Resilient

An application is *resilient* if it can recover quickly from failures.

Failures can be:

- software failures
- hardware failures, or
- connection failures.

Typically, resilience cannot be added as an afterthought; it needs to be part of the design from the beginning.

Needed:

- loose coupling.
- strong encapsulation of state.
- pervasive supervisor hierarchies.

Responsive

An application is *responsive* if it provides rich, real-time interaction with its users even under load and in the presence of failures.

Responsive applications can be built on an event-driven, scalable, and resilient architecture.

Still need careful attention to algorithms, system design, back-pressure, and many other details.

Call-backs

Handling events is often done using call-backs. E.g. using Java observers:

```
class Counter implements ActionListener {
  private var count = 0
  button.addActionListener(this)

  def actionPerformed(e: ActionEvent): Unit = {
    count += 1
  }
}
```

Call-backs

Handling events is often done using call-backs. E.g. using Java observers:

```
class Counter implements ActionListener {
  private var count = 0
  button.addActionListener(this)

  def actionPerformed(e: ActionEvent): Unit = {
    count += 1
  }
}
```

Problems:

- needs shared mutable state.
- cannot be composed.
- leads quickly to “call-back hell”.

How To Do Better

Use fundamental constructions from functional programming ...

How To Do Better

Use fundamental constructions from functional programming ...

... to get *composable* event abstractions.

How To Do Better

Use fundamental constructions from functional programming ...

... to get *composable* event abstractions.

- Events are first class.
- Events are often represented as messages.
- Handlers of events are also first-class.
- Complex handlers can be composed from primitive ones.

Contents of This Course

- Review of functional programming
- An important class of functional patterns: *monads*
- Functional programs in a stateful world
- Abstracting over events: *futures*
- Abstracting over event streams: *observables*
- Message passing architecture: *actors*
- Handling failures: *supervisors*
- Scaling out: *distributed actors*

Prerequisites

- Need a solid grounding in functional programming.
- Ideally, the “Principles of Functional Programming in Scala” class.
- If you know some other functional language, the switch should be easy.