When people talk about “reactive” in the context of software development and design, they generally mean one of three things:

* Reactive systems (architecture and design)
* Reactive programming (declarative event-based)

**Reactive systems**

A *reactive system* is an architectural style that allows multiple individual applications to coalesce as a single unit, reacting to its surroundings, while remaining aware of each other—this could manifest as being able to scale up/down, load balancing, and even taking some of these steps proactively.

It’s possible to write a single application in a reactive style (i.e. using reactive programming); however, that’s merely one piece of the puzzle. Though each of the above aspects may seem to qualify as “reactive,” in and of themselves they do not make a *system* reactive.

**Reactive programming**

Reactive programming is a subset of asynchronous programming and a paradigm where the availability of new information drives the logic forward rather than having control flow driven by a thread-of-execution.

It supports decomposing the problem into multiple discrete steps where each can be executed in an asynchronous and non-blocking fashion, and then be composed to produce a workflow—possibly unbounded in its inputs or outputs.

Reactive programming is generally event-driven, in contrast to reactive systems, which are message-driven—the distinction between event-driven and message-driven is clarified later.

The application program interface (API) for reactive programming libraries are generally either:

* Callback-based—where anonymous side-effecting callbacks are attached to event sources, and are being invoked when events pass through the dataflow chain. (loop of hell)
* Declarative—through functional composition, usually using well-established combinators like *map*, *filter*, *fold* etc.

Most libraries provide a mix of these two styles, often with the addition of stream-based operators like windowing, counts, triggers, etc.

It would be reasonable to claim that reactive programming is related to dataflow programming, since the emphasis is on the flow of data rather than the flow of control.

Examples of programming abstractions that support this programming technique are:

* Futures/Promises—containers of a single value, many-read/single-write semantics where asynchronous transformations of the value can be added even if it is not yet available.
* Streams—as in reactive streams: unbounded flows of data processing, enabling asynchronous, non-blocking, back-pressured transformation pipelines between a multitude of sources and destinations.
* Dataflow variables—single assignment variables (memory-cells) which can depend on input, procedures and other cells, so that they are automatically updated on change. A practical example is spreadsheets—where the change of the value in a cell ripples through all dependent functions, producing new values downstream.

**Event-Driven vs Message-Driven**

A message is an item of data that is sent to a specific destination. An event is a signal emitted by a component upon reaching a given state.

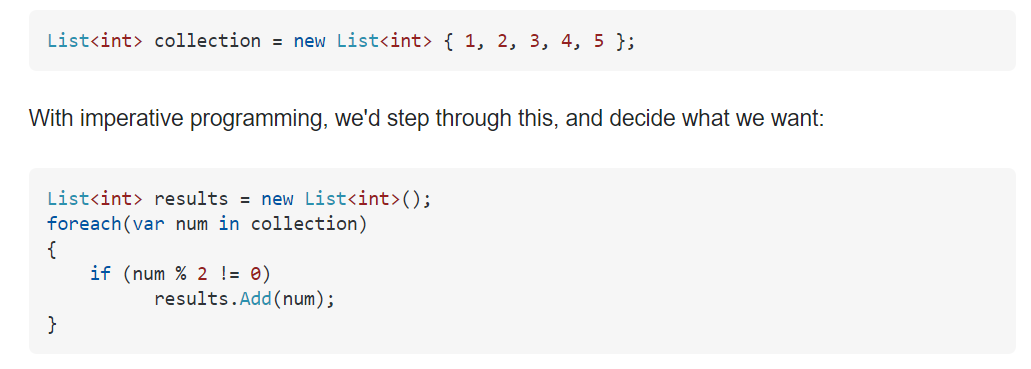
In a message-driven system addressable recipients await the arrival of messages and react to them, otherwise lying dormant.

In an event-driven system notification listeners are attached to the sources of events such that they are invoked when the event is emitted. This means that an event-driven system focuses on addressable event sources while a message-driven system concentrates on addressable recipients.

**Imperative Programming vs Declarative Programming**

With imperative programming, you tell the compiler what you want to happen, step by step.

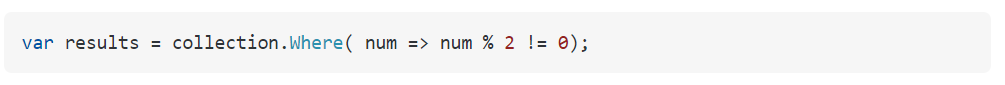
For example, let's start with this collection, and choose the odd numbers:



Here, we're saying:

* Create a result collection
* Step through each number in the collection
* Check the number, if it's odd, add it to the results

With declarative programming, on the other hand, you write code that describes what you want, but not necessarily how to get it (declare your desired results, but not the step-by-step):

 Here, we're saying "Give us everything where it's odd", not "Step through the collection. Check this item, if it's odd, add it to a result collection."