

# Mapping Java Reactive Streams Onto Reactive Programming Principles

**Douglas C. Schmidt**

**[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)**

**[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)**

**Professor of Computer Science**

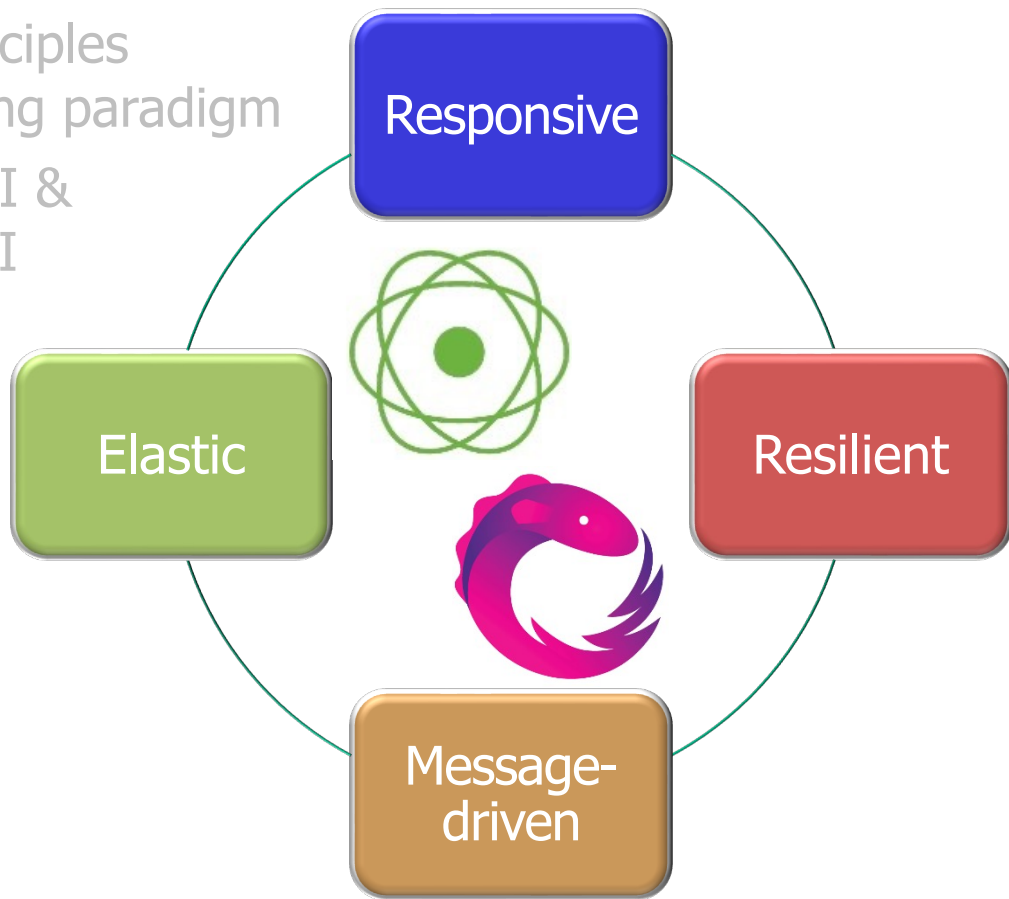
**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



# Learning Objectives in this Part of the Lesson

- Understand the key benefits & principles underlying the reactive programming paradigm
- Know the Java reactive streams API & popular implementations of this API
- Learn how Java reactive streams maps to key reactive programming principles

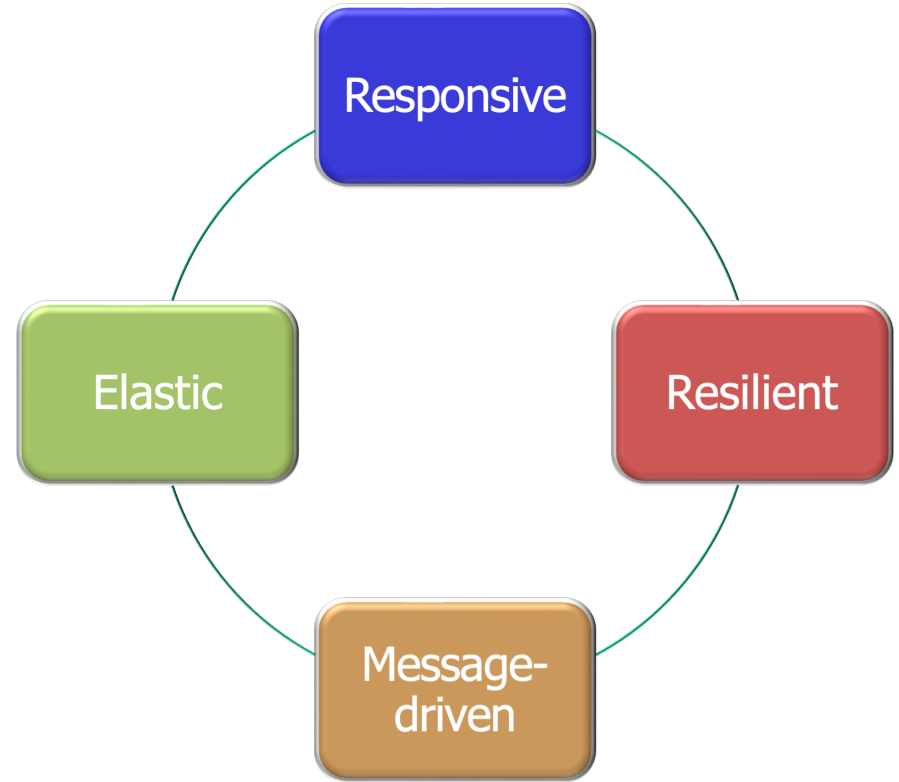
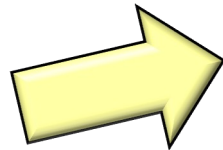


---

# Mapping Reactive Streams to Reactive Programming Principles

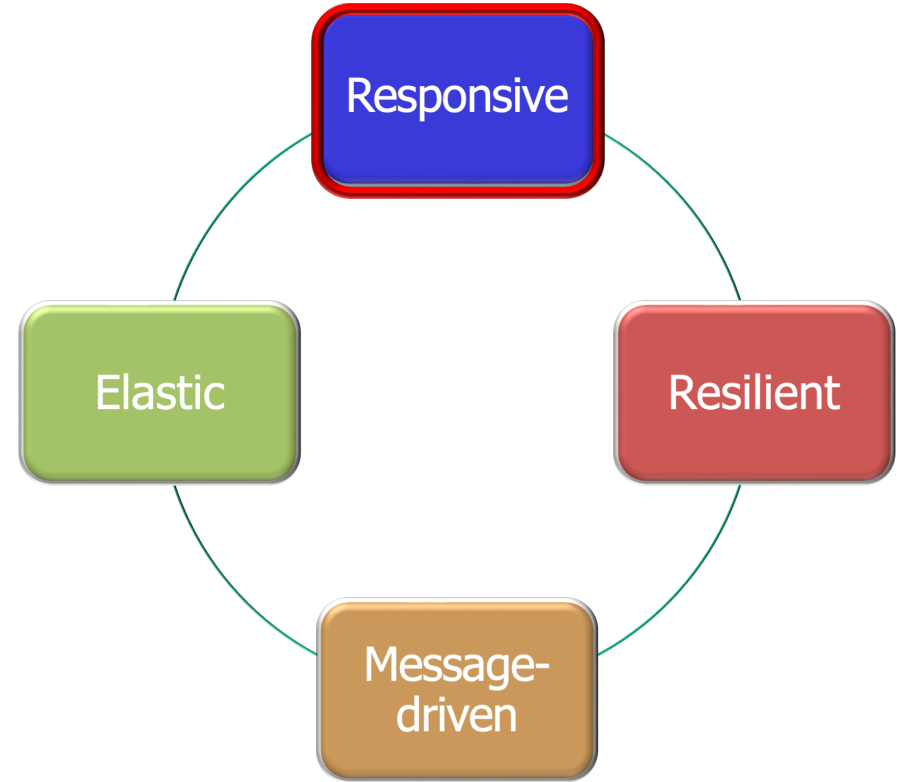
# Mapping Reactive Streams to Reactive Programming Principles

- Mapping reactive programming principles onto reactive streams features



# Mapping Reactive Streams to Reactive Programming Principles

- Mapping reactive programming principles onto reactive streams features, e.g.
  - **Responsive**



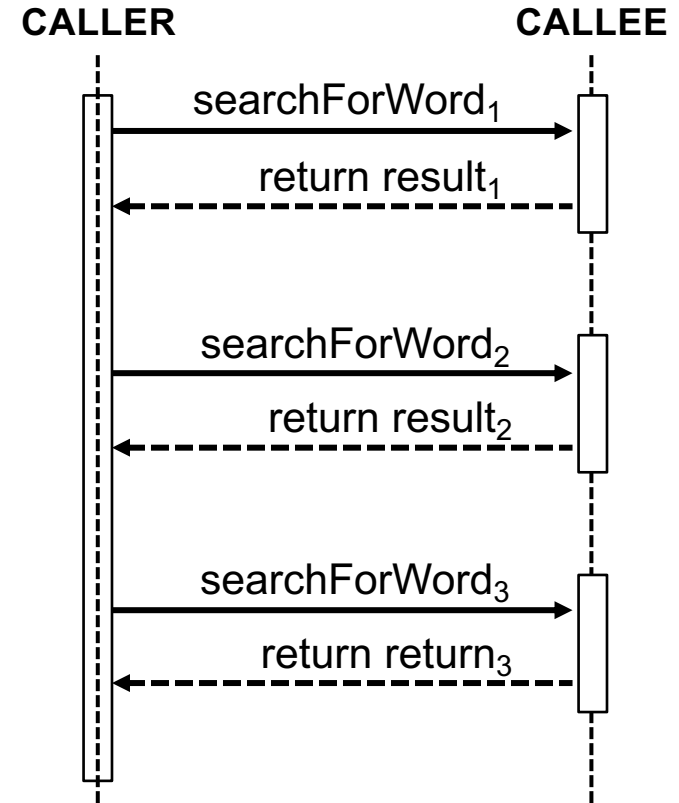
See [en.wikipedia.org/wiki/Responsiveness](https://en.wikipedia.org/wiki/Responsiveness)

# Mapping Reactive Streams to Reactive Programming Principles

- Mapping reactive programming principles onto reactive streams features, e.g.

- **Responsive**

- Avoid blocking caller code
  - Blocking underutilizes cores, impedes inherent parallelism, & complicates program structure



See [www.nastel.com/10-reasons-your-java-apps-are-slow](http://www.nastel.com/10-reasons-your-java-apps-are-slow)

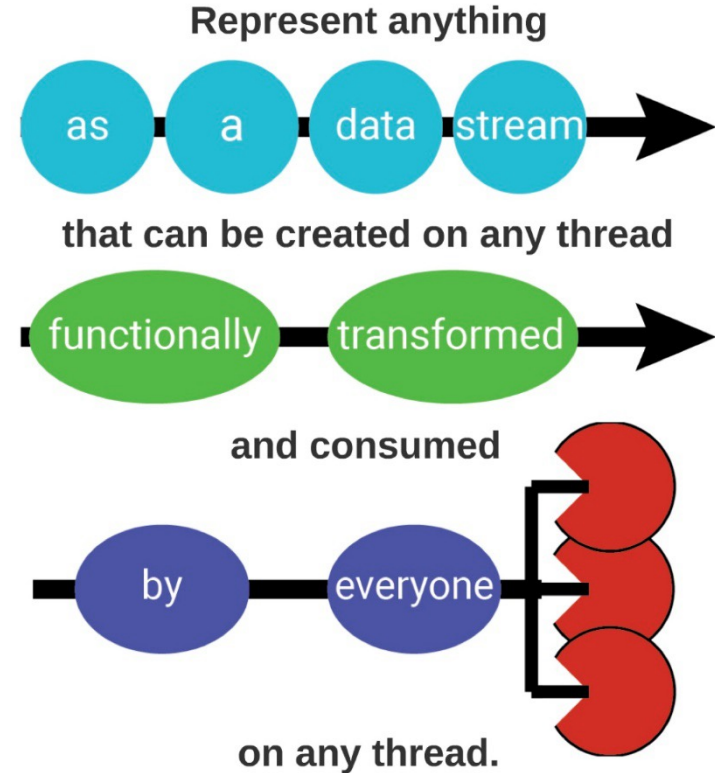
# Mapping Reactive Streams to Reactive Programming Principles

- Mapping reactive programming principles onto reactive streams features, e.g.

- **Responsive**

- Avoid blocking caller code
  - Blocking underutilizes cores, impedes inherent parallelism, & complicates program structure

*Operators like `subscribeOn()`, `publishOn()`, & `observeOn()` can be used to avoid blocking caller code*



# Mapping Reactive Streams to Reactive Programming Principles

- Mapping reactive programming principles onto reactive streams features, e.g.

- **Responsive**

- Avoid blocking caller code
- Avoid changing threads
  - Incurs excessive overhead wrt synchronization, context switching, & memory/cache management





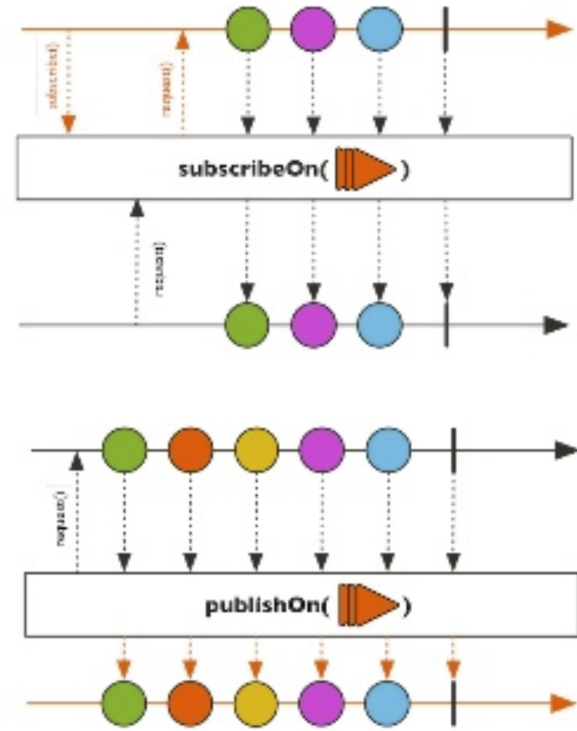
# Mapping Reactive Streams to Reactive Programming Principles

- Mapping reactive programming principles onto reactive streams features, e.g.

- **Responsive**

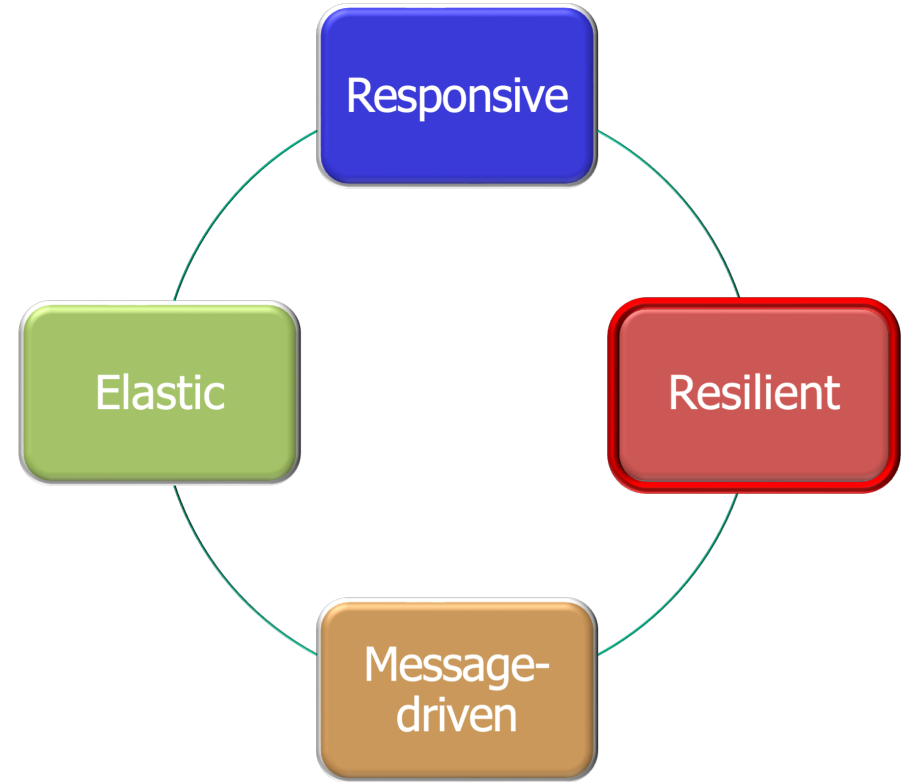
- Avoid blocking caller code
- Avoid changing threads
  - Incurs excessive overhead wrt synchronization, context switching, & memory/cache management

*Operators like `subscribeOn()`, `publishOn()`, & `observeOn()` provide fine-grained control over mapping events to threads*



# Mapping Reactive Streams to Reactive Programming Principles

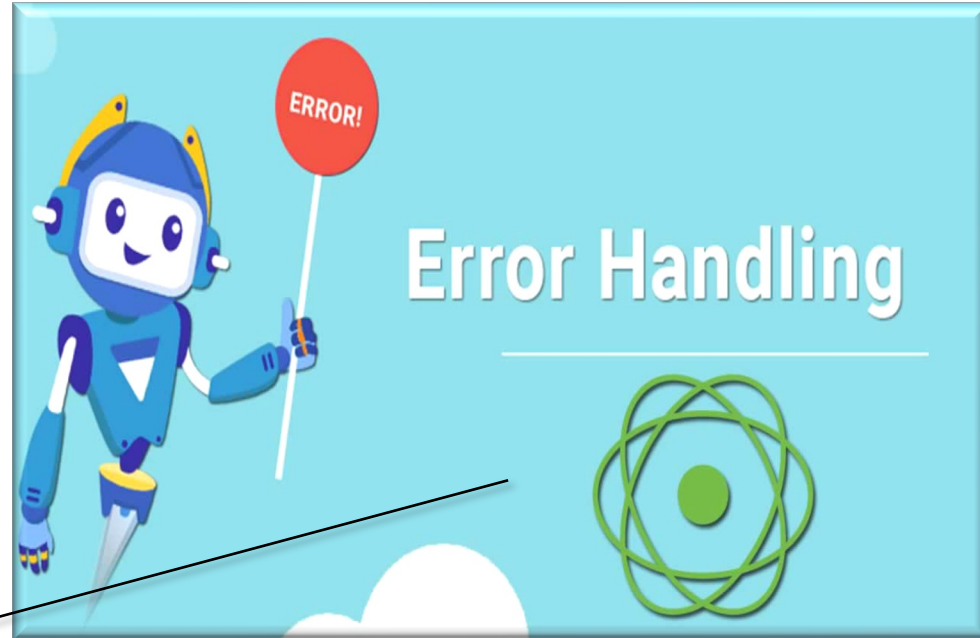
- Mapping reactive programming principles onto reactive streams features, e.g.
  - **Resilient**



See [en.wikipedia.org/wiki/Resilience\\_\(network\)](https://en.wikipedia.org/wiki/Resilience_(network))

# Mapping Reactive Streams to Reactive Programming Principles

- Mapping reactive programming principles onto reactive streams features, e.g.
  - **Resilient**
    - Exception methods make more programs resilient to failures

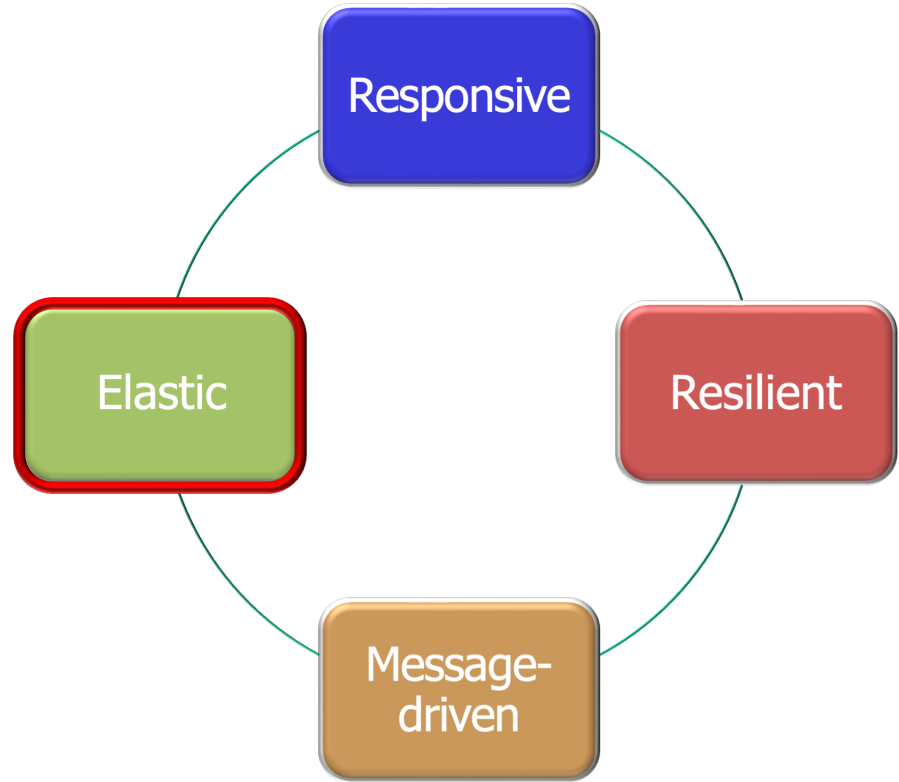


*Exception handling operators  
decouple error processing  
from normal operations*

Reactive streams are localized to a single process, *not* a cluster!

# Mapping Reactive Streams to Reactive Programming Principles

- Mapping reactive programming principles onto reactive streams features, e.g.
  - **Elastic**



See [en.wikipedia.org/wiki/Autoscaling](https://en.wikipedia.org/wiki/Autoscaling)

# Mapping Reactive Streams to Reactive Programming Principles

- Mapping reactive programming principles onto reactive streams features, e.g.

- **Elastic**

- Async computations can run scalably in a pool of threads atop a set of cores

Name	Description
<code>Schedulers.computation()</code>	Schedules computation bound work (ScheduledExecutorService with pool size = N CPU, LRU worker select strategy)
<code>Schedulers.immediate()</code>	Schedules work on current thread
<code>Schedulers.io()</code>	I/O bound work (ScheduledExecutorService with growing thread pool)
<code>Schedulers.trampoline()</code>	Queues work on the current thread
<code>Schedulers.newThread()</code>	Creates new thread for every unit of work
<code>Schedulers.test()</code>	Schedules work on scheduler supporting virtual time
<code>Schedulers.from(Executor e)</code>	Schedules work to be executed on provided executor

*RxJava schedulers support many types of threads and/or thread pools*

See [www.baeldung.com/rxjava-schedulers](http://www.baeldung.com/rxjava-schedulers)

# Mapping Reactive Streams to Reactive Programming Principles

- Mapping reactive programming principles onto reactive streams features, e.g.
  - **Elastic**
    - Async computations can run scalably in a pool of threads atop a set of cores

Reactor, like RxJava, can be considered to be **concurrency-agnostic**. That is, it does not enforce a concurrency model. Rather, it leaves you, the developer, in command. However, that does not prevent the library from helping you with concurrency.

Obtaining a **Flux** or a **Mono** does not necessarily mean that it runs in a dedicated **Thread**. Instead, most operators continue working in the **Thread** on which the previous operator executed. Unless specified, the topmost operator (the source) itself runs on the **Thread** in which the **subscribe()** call was made.

The following example runs a **Mono** in a new thread:

```
public static void main(String[] args) throws InterruptedException {
    final Mono<String> mono = Mono.just("hello "); 1

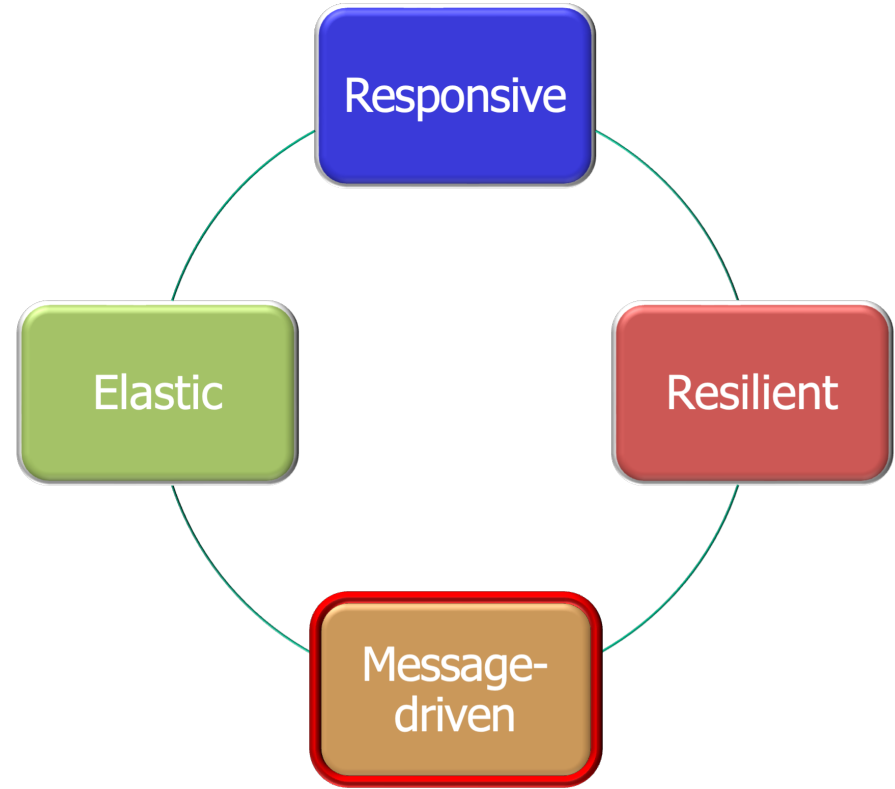
    Thread t = new Thread(() -> mono
        .map(msg -> msg + "thread ")
        .subscribe(v -> 2
            System.out.println(v + Thread.currentThread().getName()) 3
        )
    );
    t.start();
    t.join();
}
```

*Project Reactor's schedulers also support threads and/or thread pools*

See [projectreactor.io/docs/core/release/reference/#schedulers](https://projectreactor.io/docs/core/release/reference/#schedulers)

# Mapping Reactive Streams to Reactive Programming Principles

- Mapping reactive programming principles onto reactive streams features, e.g.
  - **Message-driven**

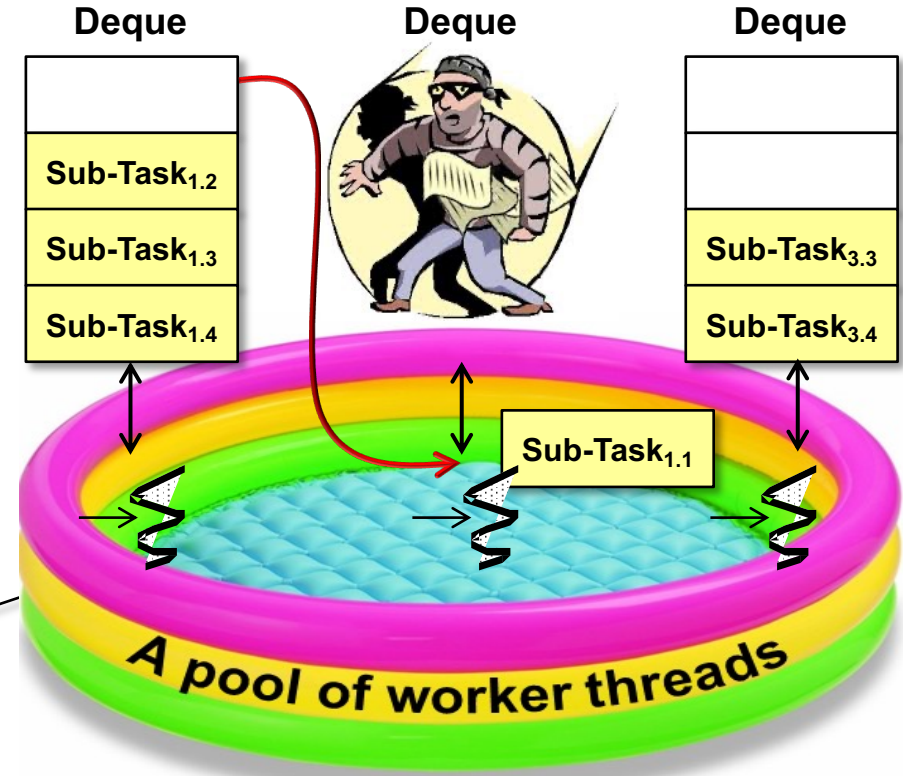


See [en.wikipedia.org/wiki/Message-oriented\\_middleware](https://en.wikipedia.org/wiki/Message-oriented_middleware)



# Mapping Reactive Streams to Reactive Programming Principles

- Mapping reactive programming principles onto reactive streams features, e.g.
  - **Message-driven**
    - Implementations of reactive streams & Java-based thread pools pass messages internally



*e.g., Java's fork-join pool supports "work-stealing" between deques*



---

# End of Mapping Java Reactive Streams onto Reactive Programming Principles