Reactive-Java

Reactive Manifesto & Reactive Streams

PART A: Reactive Manifesto

Definition

The **Reactive Manifesto** is a **set of principles** for building **modern, responsive, and resilient software systems.**
It focuses on making applications **responsive, resilient, elastic, and message-driven.**

Key Principles

Principle	Meaning
Responsive	System responds **in a timely manner**
Resilient	System stays operational **even when failures occur**
Elastic	System can **scale up or down** under load
Message-Driven	Components communicate **asynchronously** using messages

Purpose

- Handles **high load and concurrency** efficiently
- Improves **user experience** (responsive apps)
- Builds **robust distributed systems**
- Enables **scalable and maintainable architecture**

Real-Life Analogy

- **Banking system**
  - Transactions are **message-driven**
  - System remains **resilient** if a server fails
  - Can **scale** for more users during peak hours
  - Always **responds quickly**

Advantages

- High performance under load
- Better fault tolerance
- Improved scalability
- Clear separation via asynchronous messaging

PART B: Reactive Streams

Definition

Reactive Streams is a **standard for asynchronous stream processing** with **non-blocking backpressure.**

- Helps **control data flow** between producer and consumer in reactive applications.

Key Components

Component	Role
Publisher	Produces data (stream of items)
Subscriber	Consumes data
Subscription	Link between Publisher & Subscriber
Processor	Both consumes and produces data (optional)

How Reactive Streams Work

1. **Subscriber** subscribes to **Publisher**

2. Publisher sends **data asynchronously**
3. Subscriber requests data in **controlled amount** (backpressure)
4. Subscriber processes items at own pace

Code Example (Simplified with Reactor / Project Reactor)

```
Flux<Integer> numbers = Flux.range(1, 5); // Publisher
numbers.subscribe(
    n -> System.out.println("Received: " + n), // onNext
    err -> System.err.println(err),           // onError
    () -> System.out.println("Done!")         // onComplete
);
```

Real-Life Analogy

- **Publisher** → Water tap
- **Subscriber** → Glass receiving water
- **Backpressure** → Glass can only take as much as it can hold

---

Advantages

- Non-blocking I/O → scalable
- Handles **high concurrency** efficiently
- Supports **stream processing** pipelines
- Backpressure prevents **overloading consumers**

Common Interview Questions (Cognizant Level)

Q1. What is the Reactive Manifesto?
A. A set of principles for building responsive, resilient, elastic, message-driven systems.

Q2. What are the 4 key traits of reactive systems?
A. Responsive, Resilient, Elastic, Message-Driven

Q3. What is Reactive Streams?
A. Standard for asynchronous, non-blocking stream processing with backpressure.

Q4. Name the core components of Reactive Streams.
A. Publisher, Subscriber, Subscription, Processor

Q5. Difference between blocking and non-blocking?

- Blocking → waits for response
- Non-blocking → continues processing while waiting

One-Line Summary (Quick Revision)

Reactive Manifesto defines principles for resilient, responsive systems, while Reactive Streams standardizes asynchronous, non-blocking data flow with backpressure.

Reactive Programming in Java

Definition

Reactive Programming in Java is a **programming paradigm** focused on **asynchronous data streams** and **non-blocking communication**.

- It allows the system to **react to data, events, or changes** as they occur, rather than polling or blocking.

Why Reactive Programming is Needed

- Handles **high concurrency** efficiently
- Reduces **blocking and thread overhead**

- Supports **event-driven and responsive applications**
- Ideal for **microservices, real-time apps, and streaming data**

Core Concepts

Concept	**Explanation**
Data Streams	Sequence of data emitted over time
Observer / Subscriber	Consumes the data from the stream
Publisher / Observable	Produces the data asynchronously
Backpressure	Mechanism to **control flow** if consumer is slower than producer
Operators	Functions to **transform, filter, or combine** streams

Reactive Programming Libraries in Java

- **Project Reactor** → Flux (0..N items), Mono (0..1 item)
- **RxJava** → Observable, Flowable, Single
- **Akka Streams** → Actor-based reactive streams

Example Using Project Reactor

```java
import reactor.core.publisher.Flux;

public class ReactiveExample {
    public static void main(String[] args) {
        Flux<Integer> numbers = Flux.range(1, 5); // Publisher
        numbers
            .map(n -> n * 2)                       // Operator
            .filter(n -> n > 5)
            .subscribe(
                n -> System.out.println("Received: " + n), // onNext
                err -> System.err.println(err),           // onError
                () -> System.out.println("Done!")         // onComplete
            );
    }
}
```

Output:

```
6
8
10
Done!
```

Real-Life Analogy

- **Reactive Programming** → Watching a **live stock market ticker**
- Updates come **asynchronously** and you react instantly
- No need to repeatedly check prices (non-blocking)

Advantages

- Handles **millions of concurrent requests** efficiently
- Simplifies **asynchronous programming**
- Prevents **thread blocking**
- Integrates with **WebFlux, microservices, messaging systems**

Key Operators in Reactive Java

Operator	**Purpose**
map()	Transform data
filter()	Filter data based on condition

Operator	**Purpose**
flatMap()	Transform into another stream
merge()	Combine multiple streams
zip()	Combine streams pairwise

Common Interview Questions (Cognizant Level)

Q1. What is Reactive Programming?
A. A paradigm for asynchronous, non-blocking, event-driven systems.

Q2. Difference between Reactive Programming and Imperative Programming?

- Imperative → Step-by-step, blocking
- Reactive → Event-driven, non-blocking

Q3. What are Mono and Flux in Reactor?

- Mono → 0 or 1 item
- Flux → 0 to N items

Q4. What is backpressure?
A. Mechanism to prevent **overloading the subscriber** when producer is faster.

Q5. Which Java libraries support reactive programming?
A. Project Reactor, RxJava, Akka Streams

One-Line Summary (Quick Revision)
Reactive Programming in Java enables non-blocking, asynchronous, event-driven data processing using streams and operators for high-performance applications.

Building Reactive Applications in Java

Definition:
Reactive Applications are software systems built using **reactive principles**—responsive, resilient, elastic, and message-driven.

- In Java, they leverage **Reactive Streams** and frameworks like **Project Reactor** or **RxJava** for **asynchronous, non-blocking, event-driven behavior**.

Why Build Reactive Applications

- Handle **high concurrency** efficiently
- Maintain **responsiveness under load**
- Provide **resilient behavior** during failures
- Improve **resource utilization** in distributed systems

Core Principles (Reactive Manifesto)

Principle	**How it applies in reactive apps**
Responsive	Fast response to user requests
Resilient	Handles failures gracefully (retry, fallback)
Elastic	Scales up/down dynamically
Message-Driven	Components communicate asynchronously

Steps to Build Reactive Applications in Java

Step 1: Choose a Reactive Library / Framework

- **Project Reactor** → Flux, Mono (Spring WebFlux)
- **RxJava** → Observable, Flowable
- **Akka Streams** → Actor-based reactive streams

Step 2: Define Data Streams

```
Flux<Integer> numbers = Flux.range(1, 10); // Publisher
```

Step 3: Apply Operators

- Transform, filter, or combine streams

```
numbers
    .map(n -> n * 2)
    .filter(n -> n > 10);
```

Step 4: Subscribe to Streams

```
numbers.subscribe(
    n -> System.out.println("Received: " + n),
    err -> System.err.println(err),
    () -> System.out.println("Stream completed!")
);
```

Step 5: Handle Backpressure

```
Flux.range(1, 1000)
    .onBackpressureBuffer(50) // Buffer if subscriber is slow
    .subscribe(System.out::println);
```

Step 6: Integrate with Spring WebFlux

```
@RestController
public class UserController {
    @GetMapping("/users")
    public Flux<User> getUsers() {
        return userRepository.findAll(); // Reactive stream from DB
    }
}
```

Real-Life Analogy

- Live **chat application**
- Messages flow asynchronously
- Users react instantly without delays
- System scales for multiple concurrent users

Advantages

- Non-blocking → better resource usage
- Handles **millions of concurrent users**
- Improves **resilience** and **fault tolerance**
- Integrates well with **microservices and event-driven systems**

Best Practices

- Use **Mono for single result, Flux for multiple**
- Avoid blocking calls inside reactive pipelines
- Handle errors using **onErrorResume / retry**
- Manage backpressure to prevent memory overflow

Common Interview Questions (Cognizant Level)

Q1. What is a reactive application?

A. Software built with reactive principles (responsive, resilient, elastic, message-driven).

Q2. Difference between reactive and traditional applications?

- Reactive → Non-blocking, asynchronous, event-driven

- Traditional → Blocking, thread-per-request model

Q3. Which Java frameworks are used for reactive apps?
A. Project Reactor, RxJava, Akka Streams, Spring WebFlux

Q4. How do you handle errors in reactive streams?
A. Using onErrorResume, onErrorReturn, or retry

Q5. What is backpressure?
A. Mechanism to prevent **overwhelming the subscriber** when producer emits too fast

---

One-Line Summary (Quick Revision)
Reactive applications in Java are non-blocking, asynchronous, and event-driven systems built with reactive streams for scalability, resilience, and responsiveness.
