# Paradigmas

* Paradigmas de programação são diferentes formas ou estilos pela qual um programa ou linguagem de programação pode ser organizada. Cada paradigma consiste de certas estruturas, features e opiniões sobre como um problema pode ser resolvido
* Linguagens de programação não precisam seguir um paradigma a risca. Elas podem ser criadas com um paradigma em mente e fornecer features de outros paradigmas ou suportar múltiplos paradigmas ao mesmo tempo

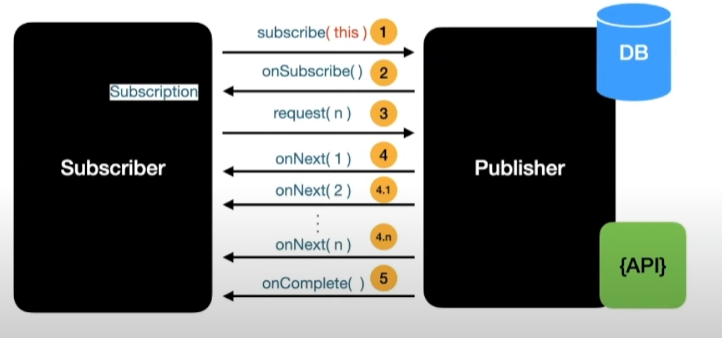
# Imperativo vs Reativo

* No **paradigma imperativo**, as tarefas são executadas sequencialmente, uma de cada vez, sendo necessário que a tarefa anterior seja concluída antes que a próxima possa começar. Os dados geralmente são processados em bulk (todos de uma vez), e só podem ser disponibilizados para a próxima etapa após o término do processamento atual.
* No **paradigma reativo**, há um conjunto de tarefas que podem processar partes dos dados de forma paralela.
  + Uma tarefa (função) realiza operações em uma fração dos dados e, em seguida, passa essa fração para a próxima tarefa enquanto continua processando o restante dos dados, utilizando fluxos contínuos (streams).
* O paradigma reativo foi criado como uma alternativa ao imperativo, com o objetivo de superar algumas de suas limitações, mas não é um substituto, ambos tem seus casos de uso
* **Imperativo** é mais intuitivo para muitos desenvolvedores, que frequentemente escrevem nesse estilo por padrão, uma vez que sua lógica é direta: basta definir um conjunto de instruções que resolvem o problema em uma sequência lógica.

## Blocking

* O maior problema do **código imperativo** é que, quando uma instrução que demora para ser completada (como operações de I/O) é executada, a **thread** fica bloqueada até que essa operação seja finalizada, desperdiçando recursos.
  + Mesmo que outra thread seja criada para realizar a operação de I/O, ela também ficará bloqueada eventualmente. Além disso, gerenciar concorrência entre várias threads é um processo complexo, já que mais threads implicam maior complexidade.
* A **programação reativa** adota um estilo funcional e declarativo. Em vez de definir um conjunto de etapas para serem executadas sequencialmente, ela descreve um pipeline ou um stream por onde os dados fluem gradualmente. Isso permite processar os dados à medida que ficam disponíveis, sem a necessidade de esperar por todos eles (async por natureza).
* A **JVM** oferece alternativas ao código bloqueante, como:
  + Future<T>: Fornece uma forma de representar o resultado de uma computação assíncrona, mas possui limitações, como: tratamento de erros mais restrito, computação não-lazy (a execução começa imediatamente) e risco de block ao chamar o método get().
  + **Listeners de **eventos** (callbacks)**: São usados para reagir a eventos assíncronos, mas podem se tornar difíceis de ler e gerenciar à medida que crescem em complexidade, resultando no chamado callback hell. Além disso, eles não oferecem funcionalidades nativas como timeout.

## Reactive Streams Spec



* É uma especificação que define um padrão para processar streams de maneira assíncrona e com controle de backpressure não bloqueante para data sources rápidos.
* A API é composta por quatro interfaces principais: Publisher, Subscriber, Subscription e Processor.
* O Publisher declara um único método, subscribe(), que é usado pelo Subscriber para se inscrever e receber eventos. O Publisher atua como a fonte de dados, e pode ser uma API, um banco de dados, etc.
* O Subscriber é uma interface com métodos chamados pelo Publisher para enviar dados e erros. Os métodos disponíveis são onNext, onError e onComplete.
  + Quando o Publisher chama onError, nenhum outro evento é enviado para a stream.
* A Subscription é uma interface intermediária entre o Subscriber e o Publisher. E. Subscription compõe Subscriber, que chama os métodos request(n) para solicitar n itens ao Publisher e cancel para fechar a stream.
  + Quando request é chamado o Publisher chama onNext de Publisher para cada evento enviado ou onError em caso de erro, quando ele não possui nenhum dado para produzir onComplete é chamado
  + Quando request é chamado, o Publisher chama o método onNext do Subscriber para cada evento enviado ou onError em caso de erro. Quando não há mais dados para produzir, o método onComplete é chamado.
* O Processor é uma interface que herda de Publisher e Subscriber. Ela é usada para produzir e processar dados, permitindo atuar como intermediária entre fontes de dados e assinantes.

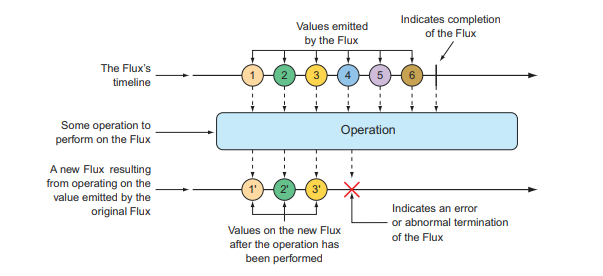
# Project Reactor

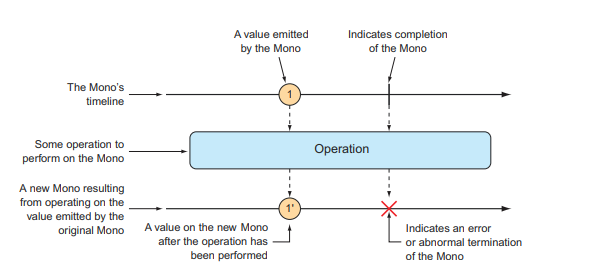
* Implementação de reactive streams, também chamada de reactor library

## Mono e Flux

* Ambos são implementações da interface Publisher:
  + **Flux** representa um Publisher que pode emitir de 0 a n eventos, onde n pode ser infinito.
  + **Mono** representa um Publisher que emite no máximo um evento (0 ou 1).
* Oferecem APIs semelhantes às streams de Collections (Java), permitindo o encadeamento de chamadas de funções lambda (as funções intermediárias são chamadas de operators) para construir um pipeline de processamento de eventos

**\*\*\* Ler os diagramas de cima para baixo**





## Características de RP

* **Composability:** Orquestra tarefas async, seja aplicando operações sequenciais sobre os dados para construir um pipeline, seja realizando várias chamadas assíncronas em paralelo e unindo seus resultados (fork-join).
* **Readability**: Facilidade de dividir tarefas complexas em unidades menores, tornando o código mais simples de entender e manter.
* **Assembly Line:** Dados podem ser comparados a uma linha de montagem. O material bruto sai da fonte (Publisher), é transformado por uma sequência de operações (Pipeline) e chega ao consumidor (Subscriber) já processado
* **Operators**: Cada operador processa os dados recebidos de um Publisher, retornando uma nova instância dos dados transformados. Esse processo se repete até que os dados alcancem sua forma final e sejam entregues ao Subscriber.
* **Nada acontece até que subscribe seja chamado**: Os dados não são processados pelo pipeline até que subscribe seja chamado, até que isso seja feito a pipeline é uma descrição de um processo async e o Publisher acumula os eventos
* **Backpressure**: O Subscriber controla quantos dados e quando deseja recebê-los, implementando um mecanismo de backpressure. Além disso, operadores podem ser usados para aplicar estratégias como buffering.
* Hot vs Cold:
  + **Cold**: Representa uma sequência de dados que começa do zero para cada novo Subscriber. Por exemplo, uma nova requisição HTTP é iniciada quando um Subscriber chama subscribe.
  + **Hot**: Não reinicia a sequência de dados. Novos Subscribers começam a receber os dados emitidos a partir do momento em que se inscrevem, permitindo que dados sejam emitidos mesmo sem Subscribers presentes.

## Async

* Por padrão, o código é executado na mesma thread que chama o método subscribe, a menos que especificado explicitamente com o uso de Schedulers.
* Os métodos publishOn e subscribeOn são utilizados executar um publisher/subscriber em threads específicas fornecidas pelos Schedulers. Além dos schedulers padrão, é possível criar implementações customizadas.
* Alguns schedulers disponíveis incluem:
  + **Bounded elastic**: Usa várias threads para processar eventos da stream em paralelo, pode aumentar o número de threads conforme necessário, até um limite máximo e é otimizado para tarefas de execução mais longa.
  + **Parallel**: Baseado em um pool de workers implementado com ExecutorService, é ideal para processar non-blocking I/O
  + **Single**: Possui um único ExecutorService com uma thread, é otimizado para operações de baixa latência e com execução única
  + **Immediate**: Executa operações imediatamente na mesma thread que chamou subscribe, sem agendar a tarefa em um scheduler. Embora seja semelhante ao comportamento padrão, ele deve ser explicitamente configurado

# Exemplos

// Cold stream: stream com dados e tamanho estáticos  
List<Integer> result = new ArrayList<>();  
Flux<Integer> flux = Flux.*just*(1, 2, 3, 4, 5)  
  
 // Operação intermediaria que sempre retorna uma nova stream  
 .map(it -> {  
 System.*out*.println(it + 1);  
 return it;  
 })  
 .map(it -> it + 1)  
  
 // fallback que será executada quando ocorrer um erro para recuperar e seguir o fluxo da stream  
 .onErrorResume(error -> {  
 System.*out*.println(error);  
 return Mono.*just*(0); // valor de fallback emitido quando há erro  
 })  
  
 // zip combina dois flux em um só, é como reduce  
 .zipWith(Flux.*range*(5, 10), (one, two) -> {  
 System.*out*.printf("First Flux: %d, Second Flux: %d%n \n", one, two);  
 return one + two;  
 });  
  
  
// Os dados só vão para a pipeline quando subscribe é chamado, subscribe chama Subscriber.request (sem backpressure nesse caso)  
flux.subscribe(result::add);  
assert result.size() == 5 : "size should be 5";  
  
// Hot stream: são streams que estão sempre ativas que precisam ser lidas constantemente e não precisam ser lidas do início  
ConnectableFlux<Object> publish = Flux.*create*(fluxSink -> {  
 for (int i = 0; i < 5; i++) {  
 fluxSink.next(System.*currentTimeMillis*());  
 }  
 })  
 // Sample é o delay entre um elemento ser enviado para a pipeline  
 .sample(Duration.*ofSeconds*(1))  
 // publish começa a executar o código no lambda acima, sem iniciar o pipeline de eventos  
 .publish();  
// Apenas indica que essa função lambda tem interesse em ouvir os eventos, mas eles não são passados para a pipeline  
publish.subscribe(System.*out*::println);  
publish.subscribe(it -> System.*out*.println((Long) it \* 2));  
// Agora os eventos são emitidos para o pipeline  
publish.connect();  
  
// Concurrency com Scheduler  
// Scheduler é uma interface que permite fazer operações de maneira concorrente  
List<Integer> resultX2 = new ArrayList<>();  
Flux.*just*(1, 2, 3, 4)  
 .map(i -> i \* 2)  
  
 // os subscribers vão ser executados em um thread secundária  
 .subscribeOn(Schedulers.*parallel*())  
 .subscribe(resultX2::add);  
System.*out*.println(resultX2);  
  
// Concurrency com parallel  
Flux.*range*(1, 10)  
 .retry(2) // número de retentativas para cada elemento  
 .parallel()// divide os dados para serem executados em paralelo  
 .runOn(Schedulers.*parallel*()) // Executa em threads paralelas  
 .map(number -> number \* 2) // Processa cada item  
  
 .subscribe(value ->  
 System.*out*.println("Thread: " + Thread.*currentThread*().getName() + " Valor: " + value)  
 );  
  
// Async: A ideia é criar os dois publishers com a chamada Async em outra thread, juntar com zip e depois bloquear com block  
Mono<Integer> firstInt = Mono.*fromCallable*(() -> *returnIntDelayed*(1))  
 .subscribeOn(Schedulers.*boundedElastic*());  
Mono<Integer> secondInt = Mono.*fromCallable*(() -> *returnIntDelayed*(2))  
 .subscribeOn(Schedulers.*boundedElastic*());  
  
Integer resSum = Mono.*zip*(firstInt, secondInt) // zip combina o resultado dos dois monos quando o valor emitido  
 .map(tuple -> tuple.getT1() + tuple.getT2())  
 .block(); // block vai bloquear até que o valor seja emitido por esse mono e também chama request  
System.*out*.println(resSum);