Parallel
&
Asynchronous Programming
In
Modern Java

Dilip Sundarraj

About Me

- Dilip
- Building Software’s since 2008
- Teaching in **UDEMY** Since 2016

What's Covered ?

- Need for Parallel and Asynchronous Programming
- Covers the **ParallelStreams and CompletableFuture API**
- Techniques to write **Fast Performing Code using Functional Style Concurrency APIs** in Java
- Covers Best Practices using **ParallelStreams/CompletableFuture API** in your code
- NonBlocking RestFul API Client using **CompletableFuture** API
- Testing ParallelStreams and CompletableFuture API Using **JUnit5**

Targeted Audience

- Experienced Java Developers
- Developers who has the need to write code that executes faster
- Developers who has the need to write code that executes in **Parallel**
- Developer who has the need to write asynchronous/non-blocking code

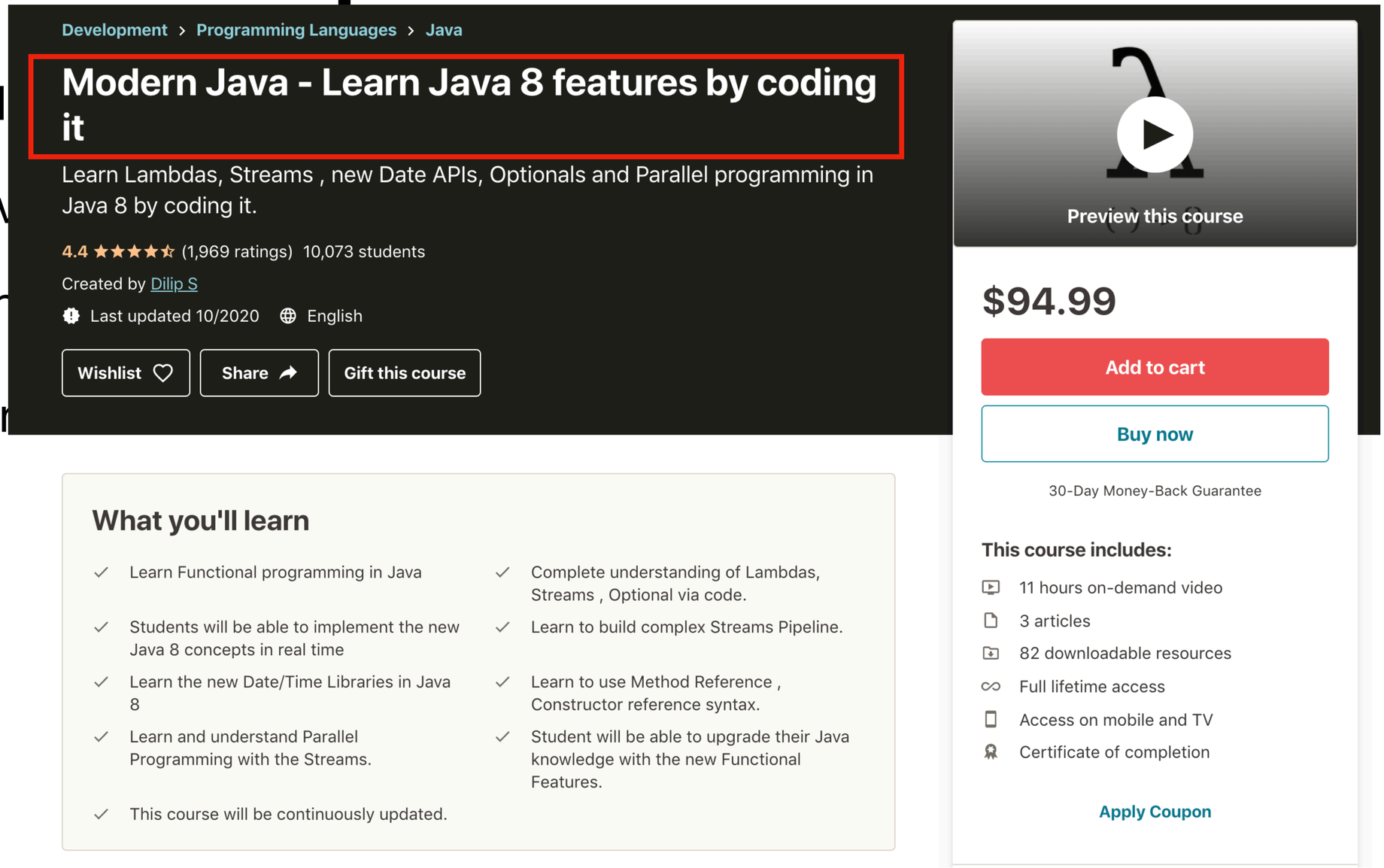
Source Code

Thank You!

Prerequisites

Course Prerequisites

- **Java 11**
- Prior Jav
- Experien
- Experien
- **Intellij ,**



Why Parallel and Asynchronous Programming ?

Why Parallel and Asynchronous Programming?

- We are living in a fast paced environment
- In Software Programming:
  - Code that we write should execute faster
- Goal of Asynchronous and Parallel Programming
  - Provide Techniques to improve the performance of the code

Technology Advancements

Hardware

- Devices or computers comes up with Multiple cores
- Developer needs to learn programming patterns to maximize the use of multiple cores
- Apply the Parallel Programming concepts
- **Parallel Streams**

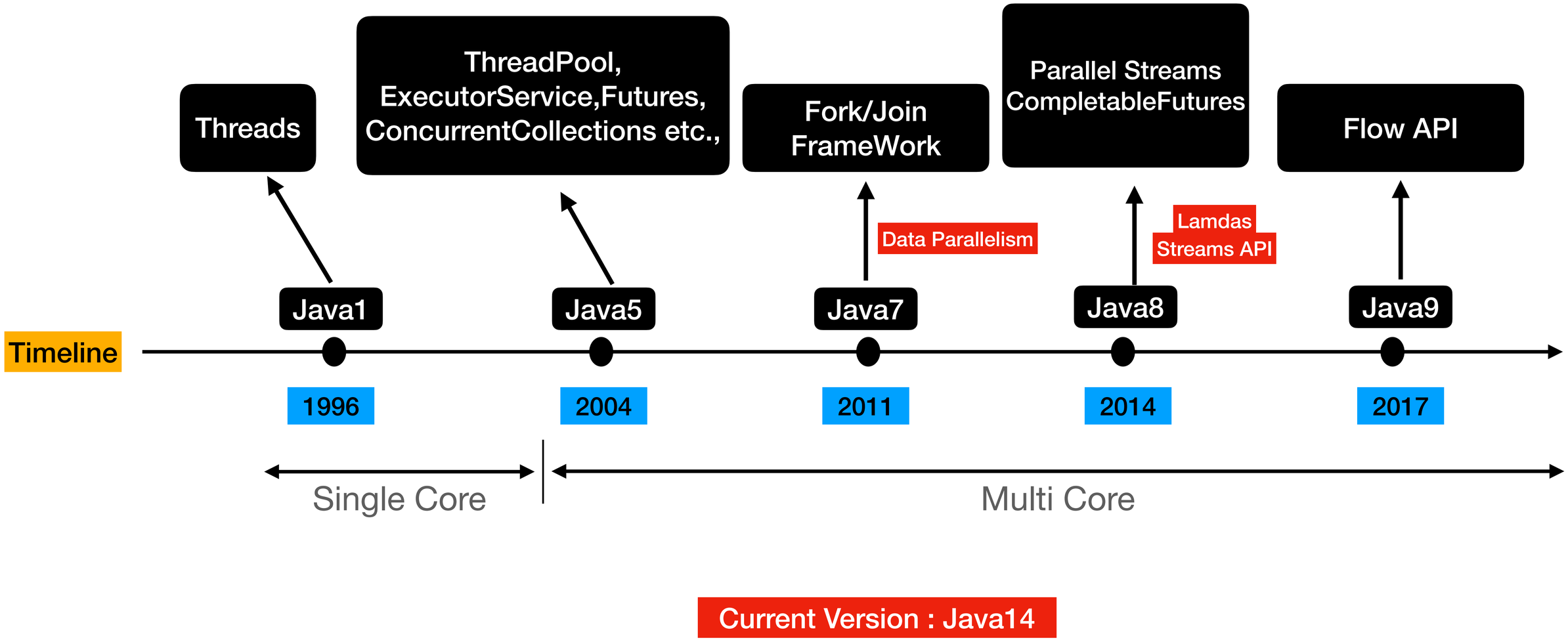
Software

- MicroServices Architecture style
- Blocking I/O calls are common in MicroServices Architecture. This also impacts the latency of the application
- Apply the Asynchronous Programming concepts
- **CompletableFuture**

Threads

Functional Style of Programming

Evolution of Concurrency and Parallelism APIs in Java



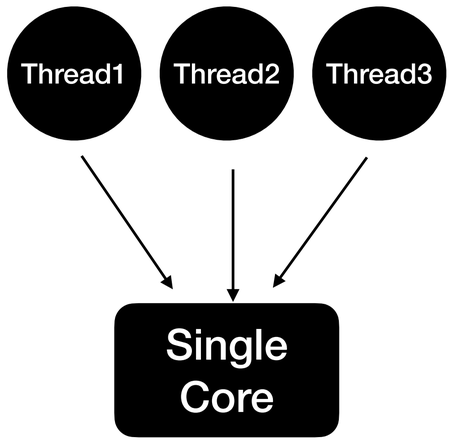
Current Version : Java14

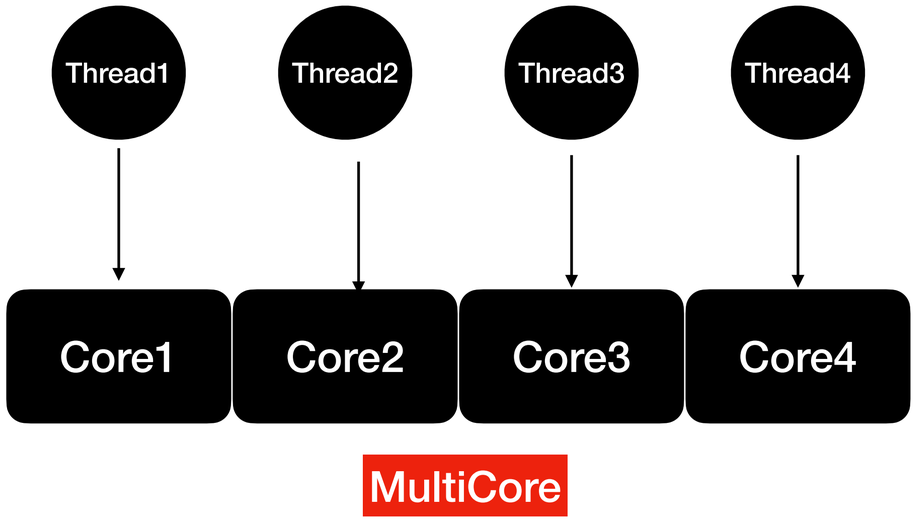
https://en.wikipedia.org/wiki/Java_version_history

Concurrency
vs
Parallelism

Concurrency

- Concurrency is a concept where two or more task can run simultaneously
- In Java, Concurrency is achieved using **Threads**
  - Are the tasks running in interleaved fashion?
  - Are the tasks running simultaneously ?





Concurrency Example

- In a real application, Threads normally need to interact with one another
  - Shared Objects or Messaging Queues
- Issues:
  - Race Condition
  - DeadLock and more
- Tools to handle these issues:
  - Synchronized Statements/Methods
  - Reentrant Locks, Semaphores
  - Concurrent Collections
  - Conditional Objects and More

```
public class HelloWorldThreadExample {
    private static String result="";

    private static void hello(){
        delay(500);
        result = result.concat("Hello");
    }
    private static void world(){
        delay(600);
        result = result.concat(" World");
    }

    public static void main(String[] args) throws InterruptedException {

        Thread helloThread = new Thread(()-> hello());
        Thread worldThread = new Thread(()-> world());

        //Starting the thread
        helloThread.start();
        worldThread.start();

        //Joining the thread (Waiting for the threads to finish)
        helloThread.join();
        worldThread.join();

        System.out.println("Result is : " + result);

    }
}
```

1

2

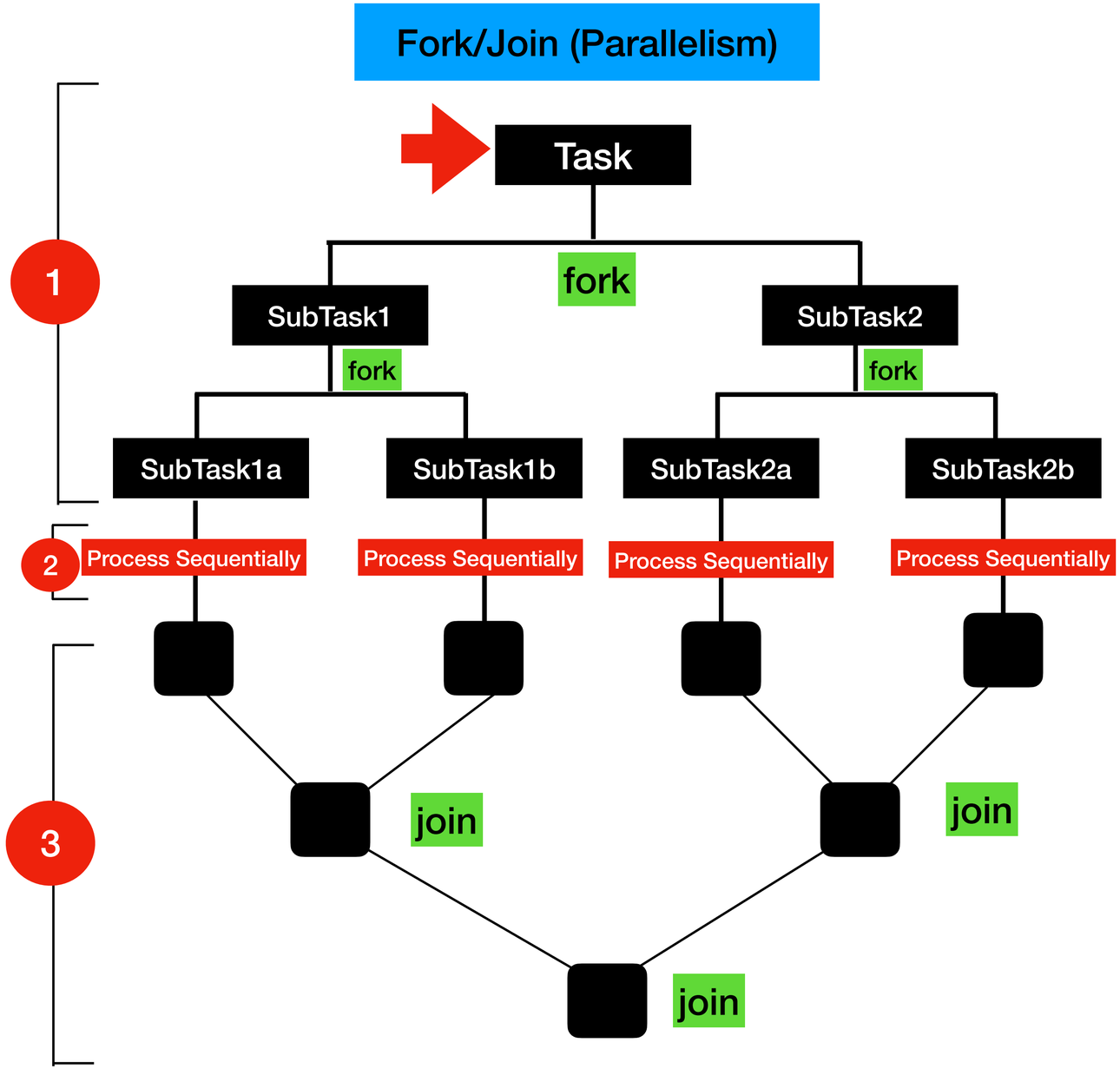
3

Threads

Hello World

Parallelism

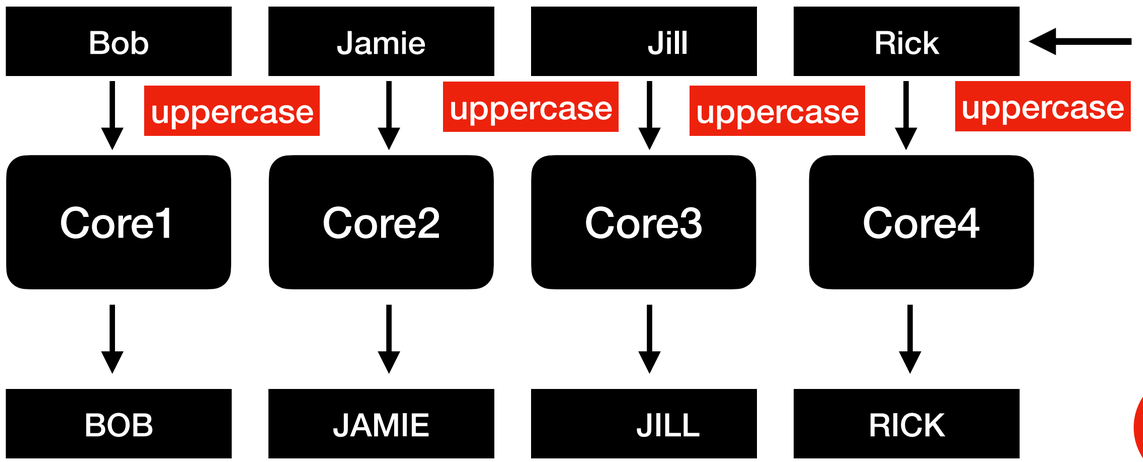
- Parallelism is a concept in which tasks are literally going to run in parallel
- Parallelism involves these steps:
  - Decomposing the tasks in to SubTasks(Forking)
  - Execute the subtasks in sequential
  - Joining the results of the tasks(Join)
- Whole process is also called **Fork/ Join**

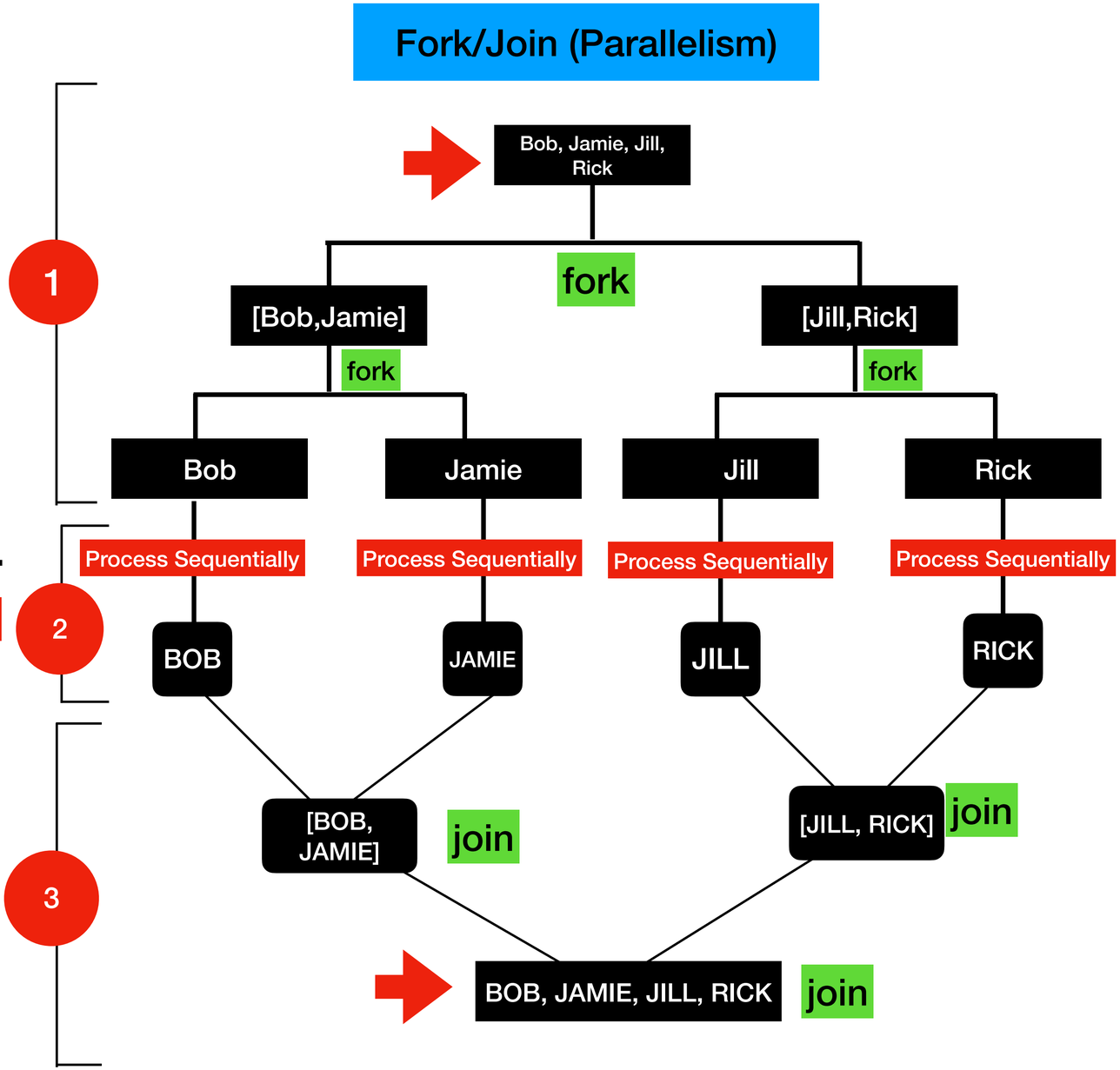


Parallelism Example

UseCase: Transform to UpperCase

[Bob, Jamie, Jill, Rick] -> [BOB, JAMIE, JILL, RICK]





Parallelism Example

```
public class ParallelismExample {

    public static void main(String[] args) {

        List<String> namesList = List.of("Bob", "Jamie", "Jill", "Rick");
        System.out.println("namesList : " + namesList);
        List<String> namesListUpperCase = namesList
                .parallelStream()
                .map(String::toUpperCase)
                .collect(Collectors.toList());

        System.out.println("namesListUpperCase : " + namesListUpperCase);

    }
}
```

Concurrency vs Parallelism

- Concurrency is a concept where two or more tasks can run in simultaneously
- Concurrency can be implemented in single or multiple cores
- Concurrency is about correctly and efficiently controlling access to shared resources

- Parallelism is a concept where two or more tasks are literally running in parallel
- Parallelism can only be implemented in a multi-core machine
- Parallelism is about using more resources to access the result faster

Course Project Setup

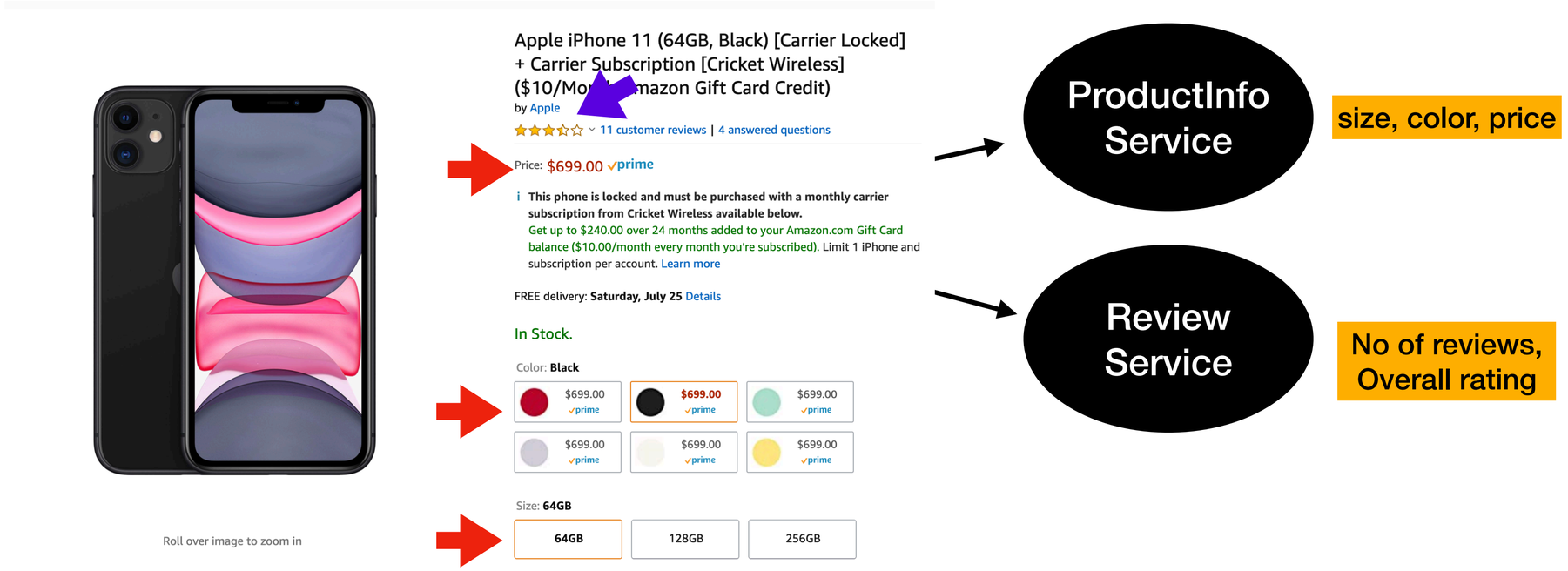
Section Overview

Section Overview

- Covers Asynchronous and Parallel Programming prior Java 8
- Threads, Futures and ForkJoin Framework and its limitations
- Covers Theory and Hands On

Overview of the Product Service

Product Service



Threads

Threads API

- Threads API got introduced in Java1
- Threads are basically used to offload the blocking tasks as **background** tasks
- Threads allowed the developers to write asynchronous style of code

Thread API Limitations

- Requires a lot of code to introduce asynchrony
  - Runnable, Thread
    - Require additional properties in Runnable
    - Start and Join the thread
- Low level
- Easy to introduce complexity in to our code

ThreadPool, ExecutorService & Future

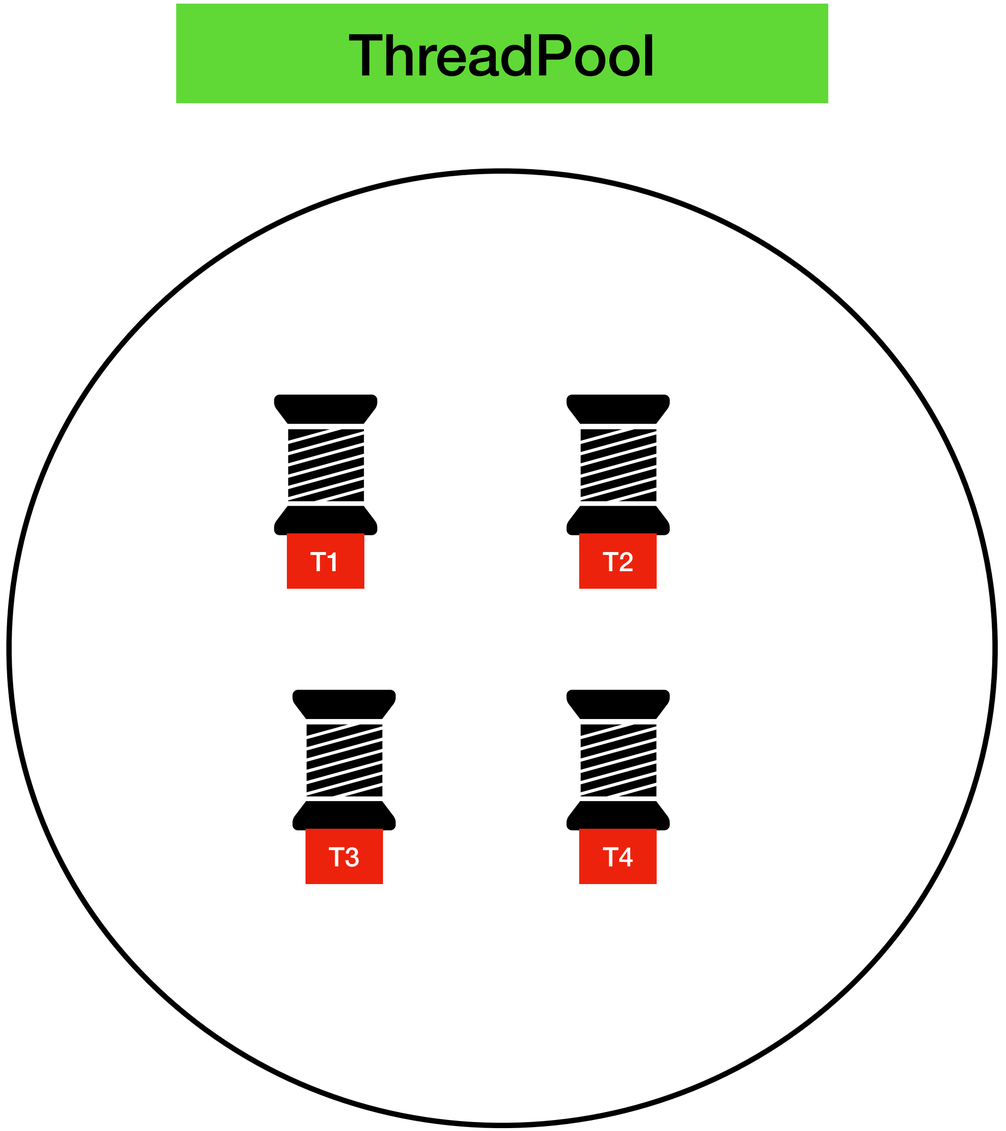
Limitations Of Thread

- Limitations of Thread:
  - Create the thread
  - Start the thread
  - Join the thread
- Threads are expensive
  - Threads have their own runtime-stack, memory, registers and more

Thread Pool was created specifically to solve this problem

Thread Pool

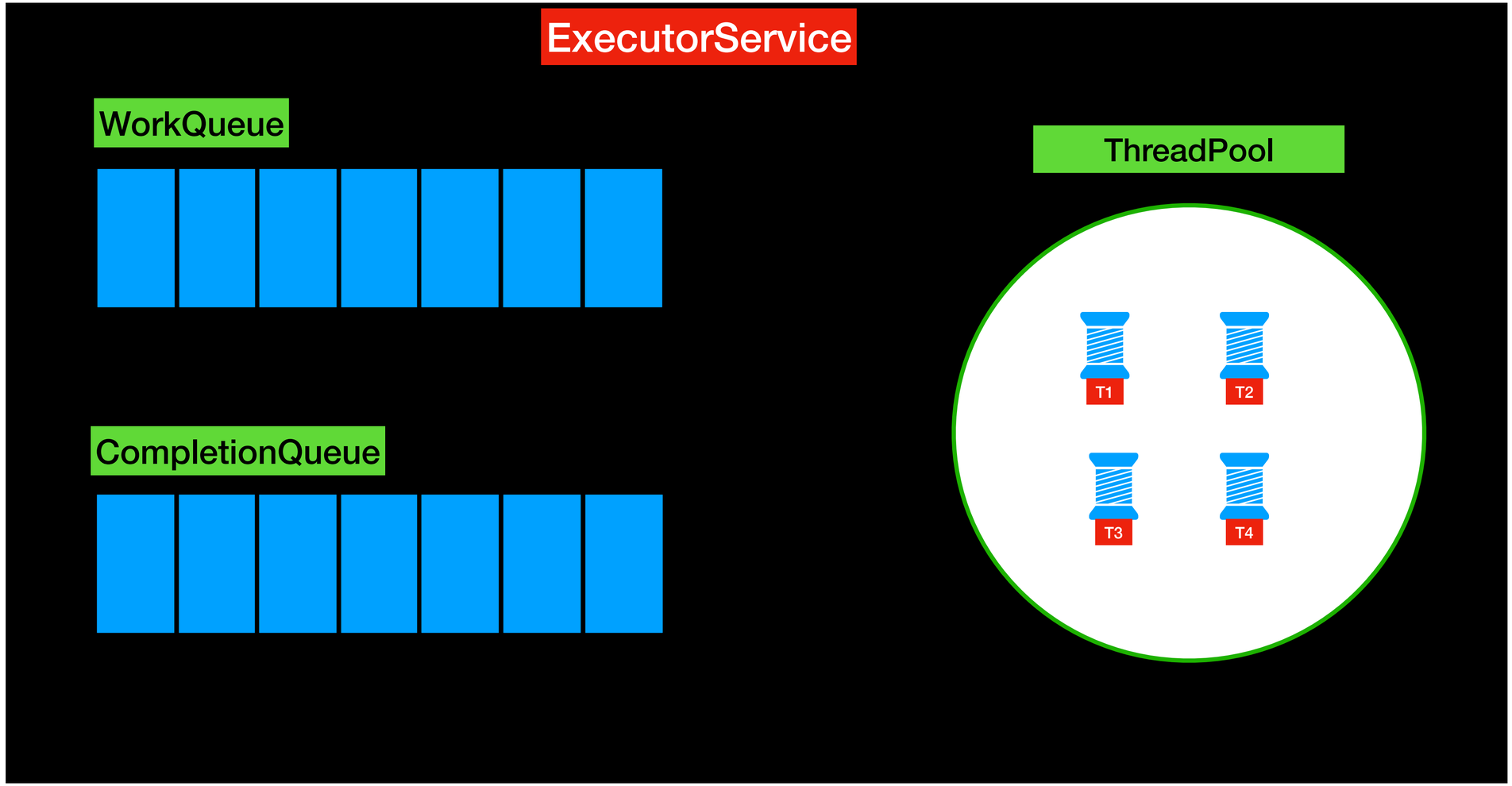
- Thread Pool is a group of threads created and readily available
- CPU Intensive Tasks
  - ThreadPool Size = No of Cores
- I/O task
  - ThreadPool Size > No of Cores
- What are the benefits of thread pool?
  - No need to manually create, start and join the threads
  - Achieving Concurrency in your application



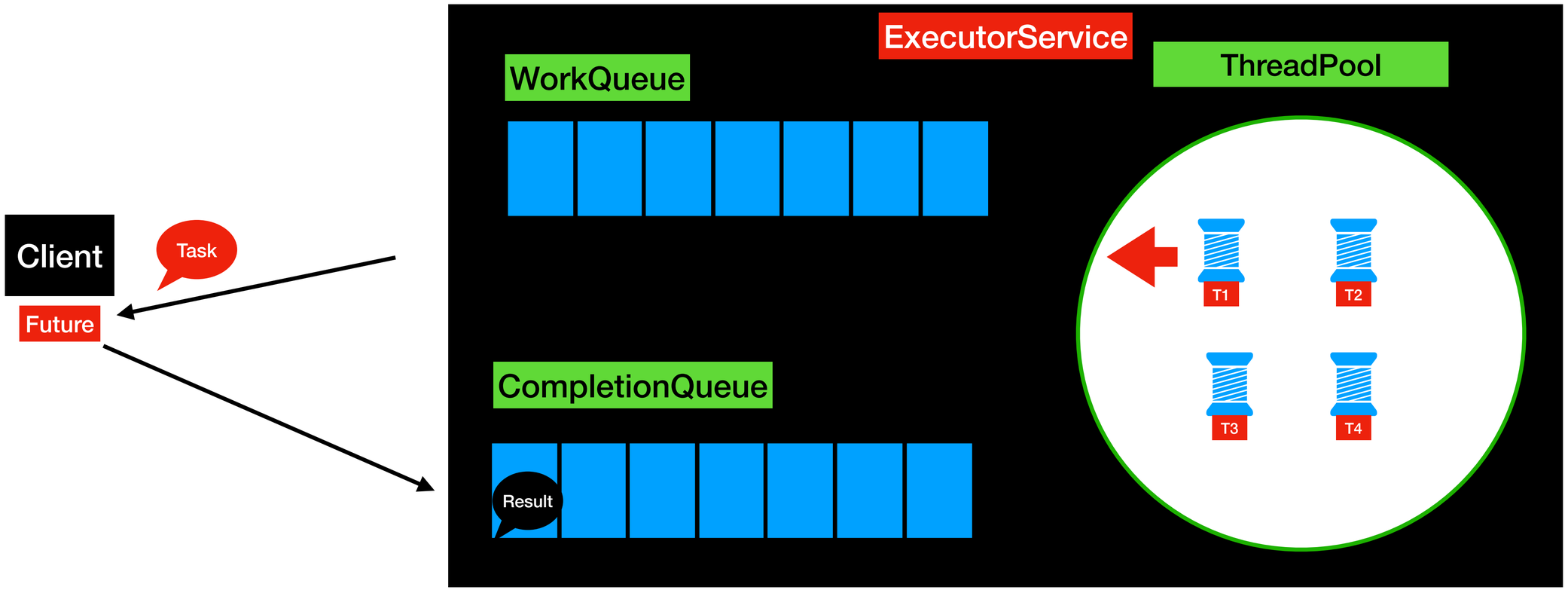
ExecutorService

- Released as part of Java5
- ExecutorService in Java is an **Asynchronous Task Execution Engine**
- It provides a way to asynchronously execute tasks and provides the results in a much simpler way compared to threads
- This enabled coarse-grained task based parallelism in Java

ExecutorService



Working Of ExecutorService



Limitations of ExecutorService

- Designed to Block the Thread

```
ProductInfo productInfo = productInfoFuture.get();
Review review = reviewFuture.get();
```

- No better way to combine futures

```
ProductInfo productInfo = productInfoFuture.get();
Review review = reviewFuture.get();
return new Product(productId, productInfo, review);
```

Fork/Join Framework

Fork/Join Framework

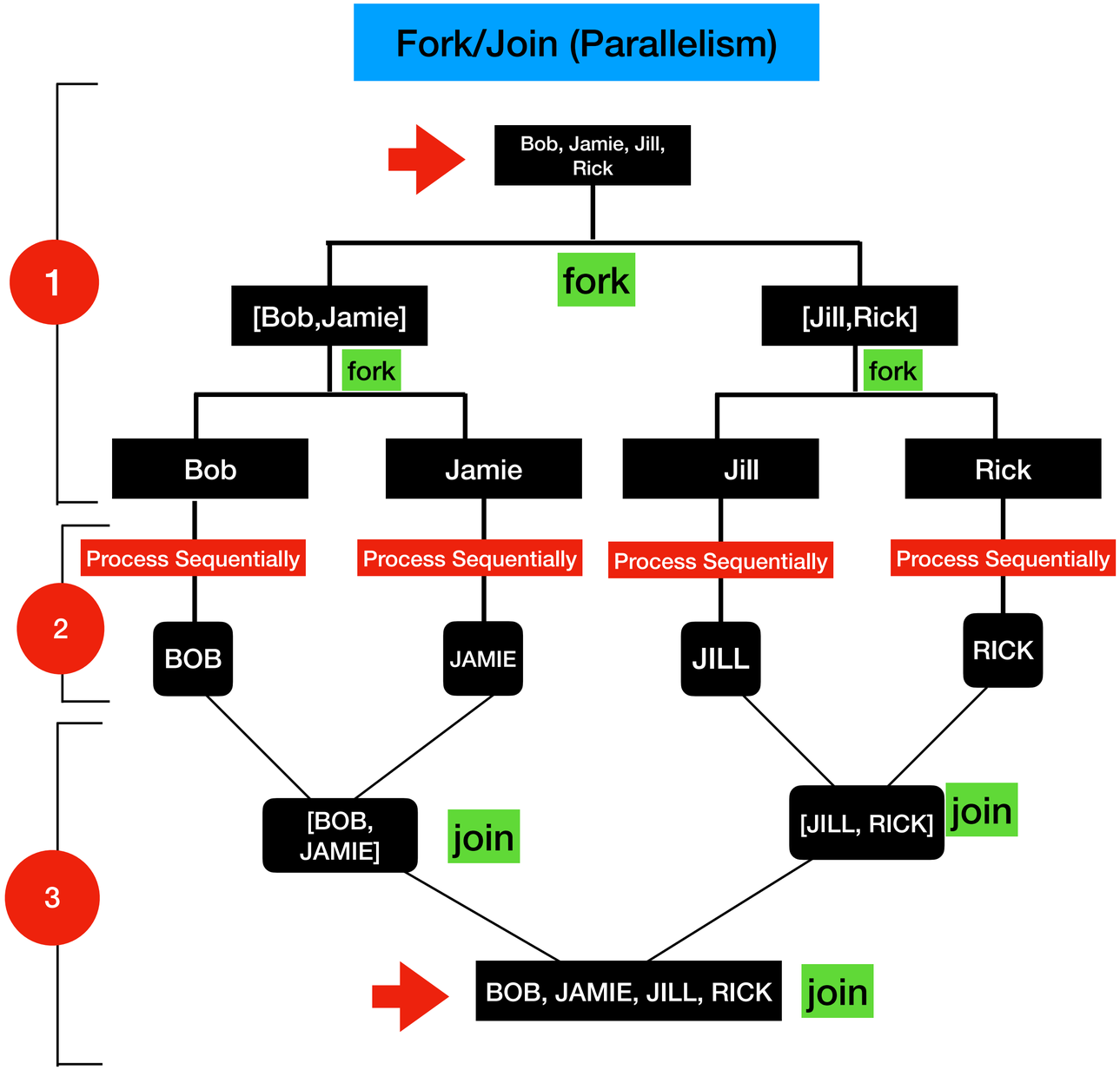
- This got introduced as part of **Java7**
- This is an extension of **ExecutorService**
- Fork/Join framework is designed to achieve **Data Parallelism**
- ExecutorService is designed to achieve **Task Based Parallelism**

```
Future<ProductInfo> productInfoFuture = executorService.submit(() -> productInfoService.retrieveProductInfo(productId));
Future<Review> reviewFuture = executorService.submit(() -> reviewService.retrieveReviews(productId));
```

What is Data Parallelism ?

- Data Parallelism is a concept where a given **Task** is recursively split in to **SubTasks** until it reaches it leaset possible size and execute those tasks in parallel
- Basically it uses the **divide and conquer** approach

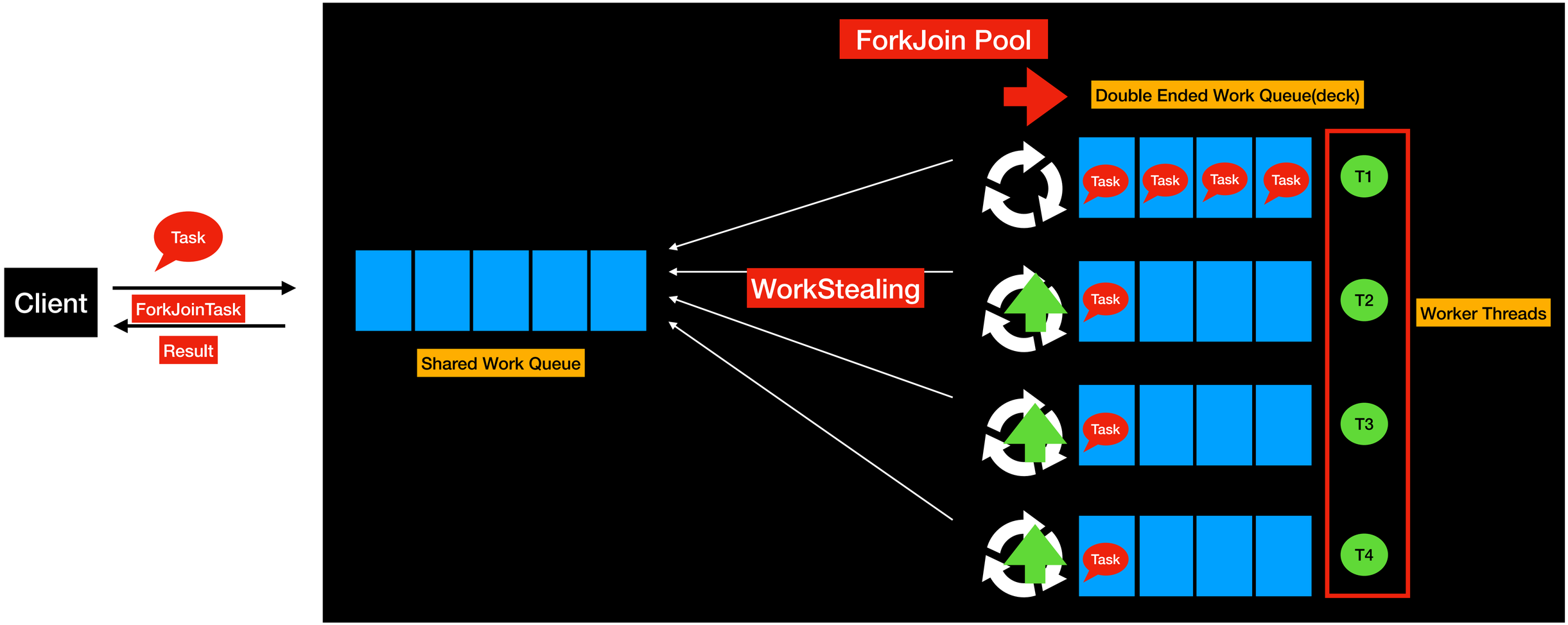
Watch "Concurrency vs Parallelism”



How does Fork/Join Framework Works ?

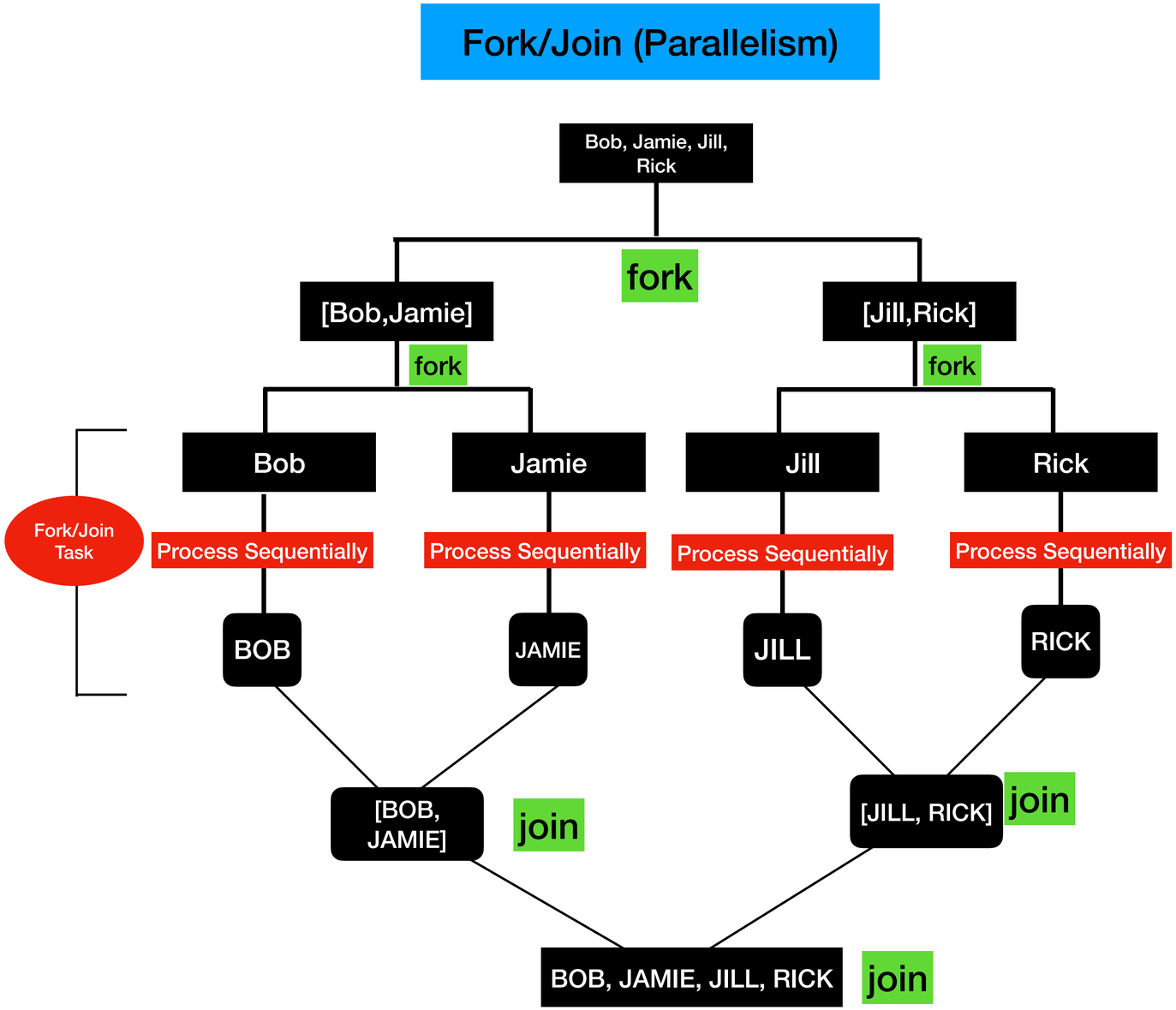
ForkJoin Pool to support Data Parallelism

ForkJoin Pool



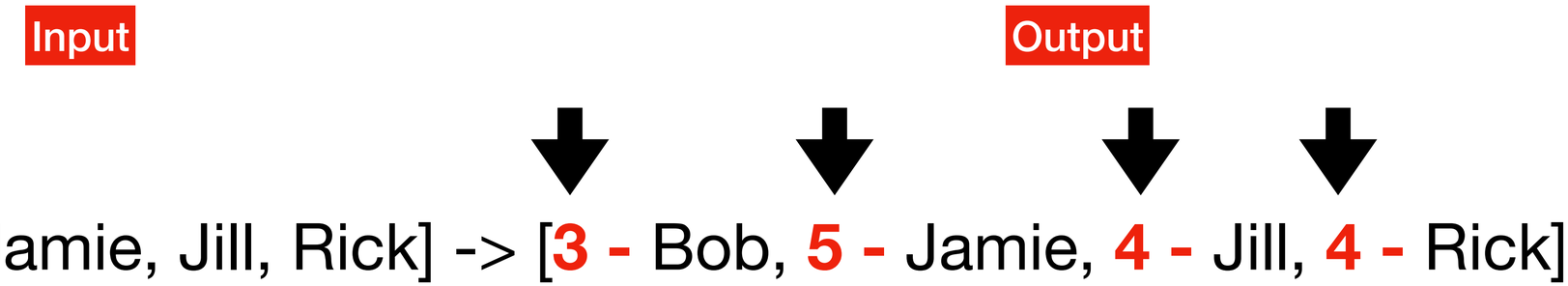
ForkJoin Task

- ForkJoin Task represents part of the data and its computation
- Type of tasks to submit to ForkJoin Pool
  - ForkJoinTask
    - RecursiveTask -> Task that returns a value
    - RecursiveAction -> Task that does not return a value



Fork/Join Example

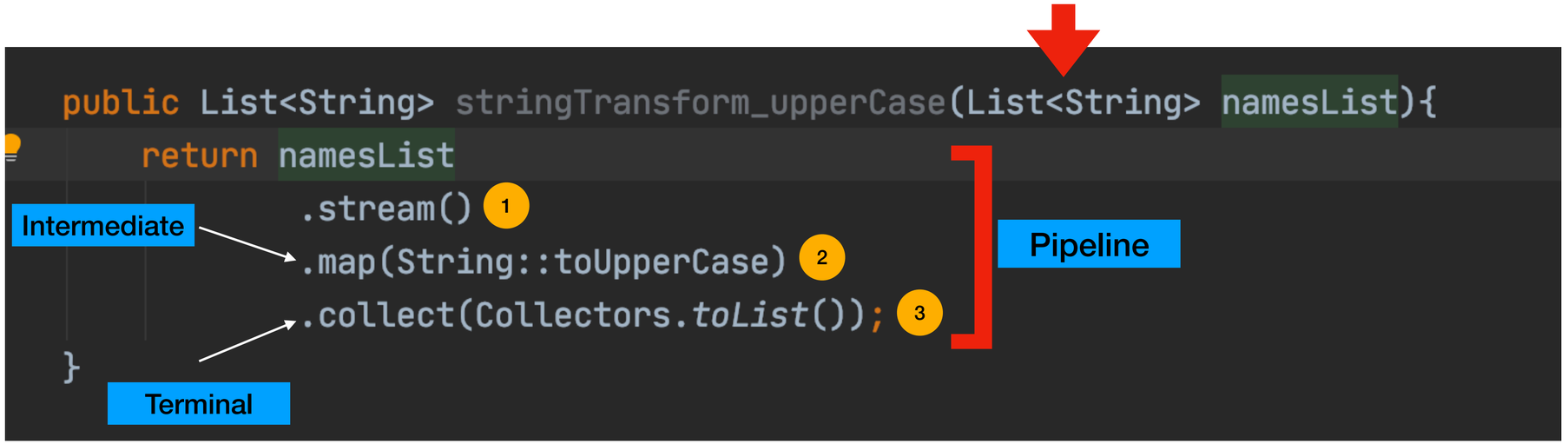
ForkJoin - UseCase



Streams API
&
Parallel Streams

Streams API

- Streams API got introduced in **Java 8**
- Streams API is used to process a collection of Objects
- Streams in Java are created by using the **stream()** method



Development > Programming Languages > Java

Modern Java - Learn Java 8 features by coding it

Learn Lambdas, Streams , new Date APIs, Optionals and Parallel programming in Java 8 by coding it.

4.4 ★★★★½ (1,782 ratings) 9,107 students

Created by Dilip S

Last updated 8/2020 English

Wishlist Share Gift this course

$94.99

Add to cart

Buy now

30-Day Money-Back Guarantee

This course includes:

- 11 hours on-demand video
- 3 articles
- 82 downloadable resources
- Full lifetime access
- Access on mobile and TV
- Certificate of completion

Apply Coupon

What you'll learn

- ✓ Learn Functional programming in Java
- ✓ Students will be able to implement the new Java 8 concepts in real time
- ✓ Learn the new Date/Time Libraries in Java 8
- ✓ Learn and understand Parallel Programming with the Streams.
- ✓ This course will be continuously updated.
- ✓ Complete understanding of Lambdas, Streams , Optional via code.
- ✓ Learn to build complex Streams Pipeline.
- ✓ Learn to use Method Reference , Constructor reference syntax.
- ✓ Student will be able to upgrade their Java knowledge with the new Functional Features.

ParallelStreams

- This allows your code to run in parallel
- ParallelStreams are designed to solve **Data Parallelism**

```
public List<String> stringTransform_upperCase(List<String> namesList){
    return namesList
            .parallelStream()
            .map(String::toUpperCase)
            .collect(Collectors.toList());
}
```

Watch “Concurrency vs Parallelism” & “Fork-Join Framework”

Stream/Parallel Stream

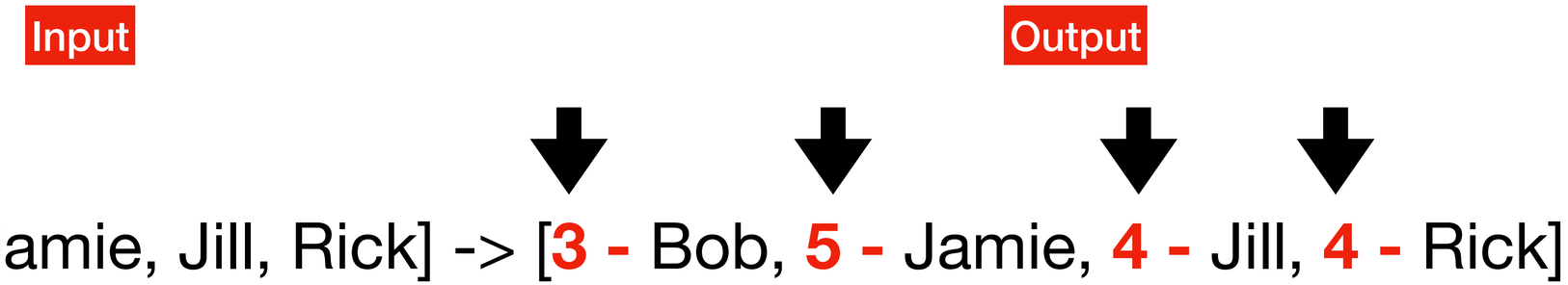
Stream

```
public List<String> stringTransform_upperCase(List<String> namesList){
    return namesList
            .stream()
            .map(String::toUpperCase)
            .collect(Collectors.toList());
}
```

Parallel Stream

```
public List<String> stringTransform_upperCase(List<String> namesList){
    return namesList
            .parallelStream()
            .map(String::toUpperCase)
            .collect(Collectors.toList());
}
```

Parallel Streams - UseCase



Unit Testing Parallel Streams Using JUnit5

Why Unit Tests ?

- Unit Testing allows you to programmatically test your code
- Manual Testing slows down the development and delivery
- Unit Testing allows the developer or the app team to make enhancements to the existing code easily and faster

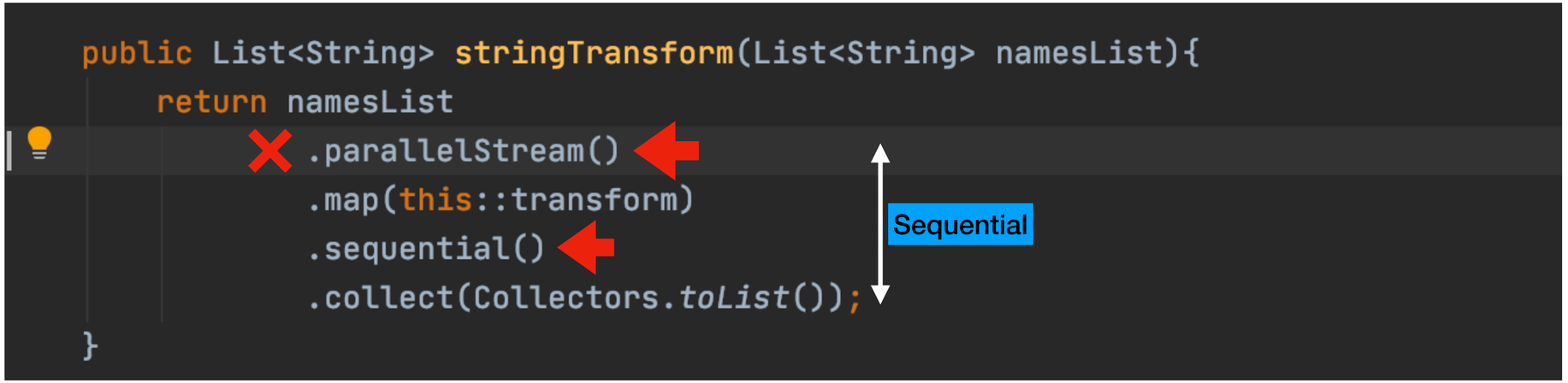
Sequential/Parallel Functions in Streams API

sequential() and parallel()

- Streams API are **sequential** by default
  - sequential() -> Executes the stream in sequential
  - parallel() -> Executes the stream in parallel
  - Both the functions() changes the behavior of the whole pipeline

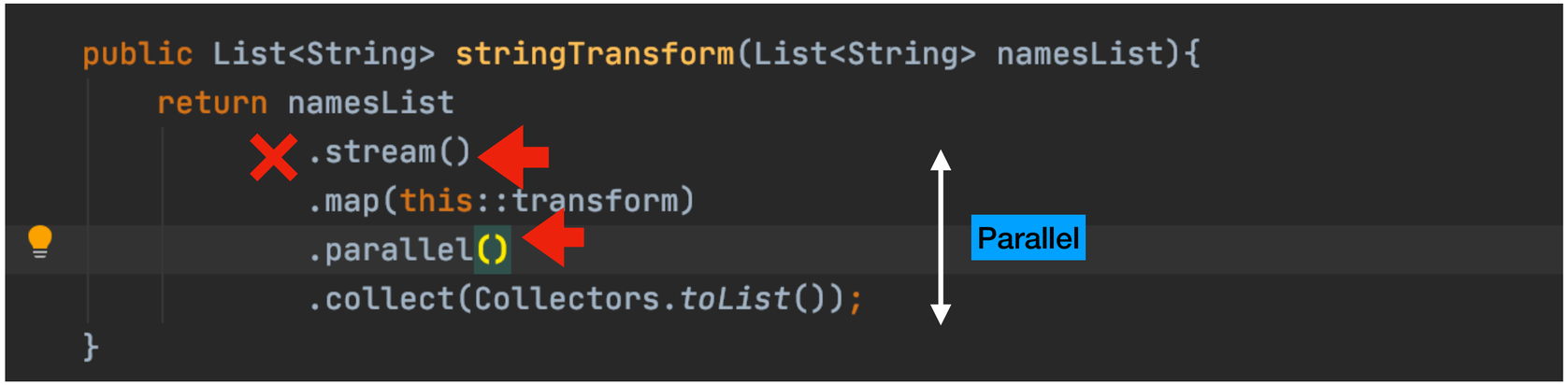
sequential()

- Changing the **parallelStream()** behavior to sequential



parallel()

- Changing the **stream()** behavior to parallel



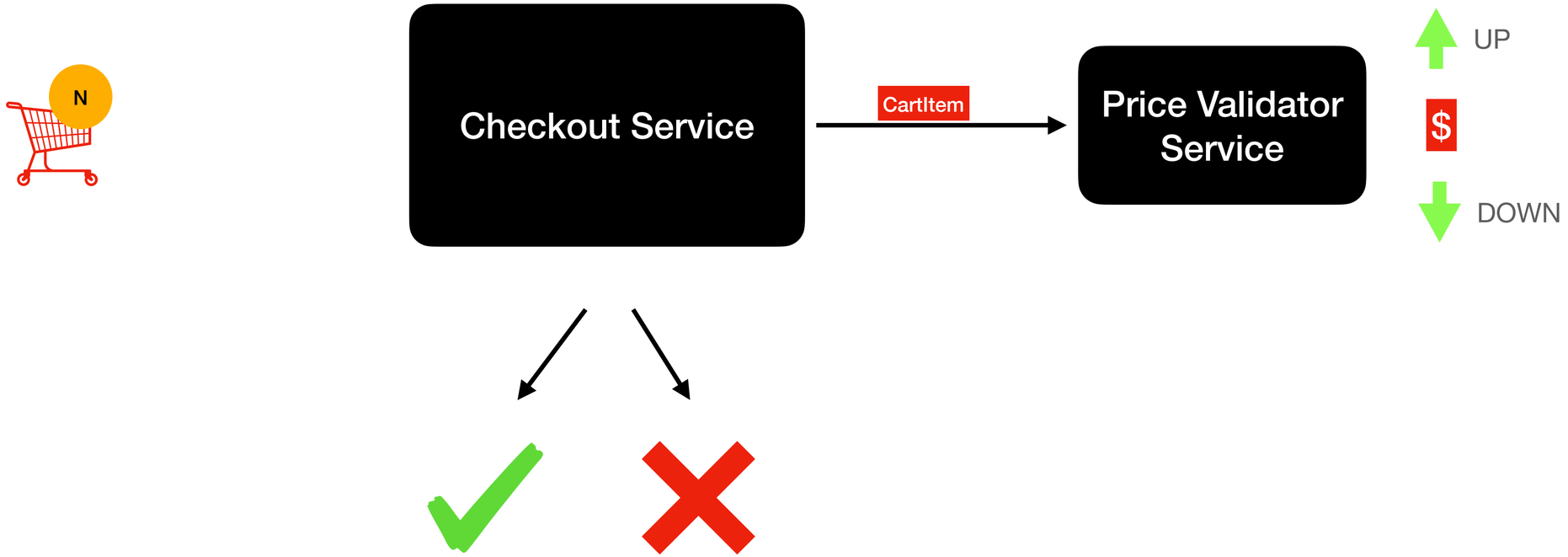
When to use sequential() and parallel() ?

Used these functions when I would like to evaluate between **sequential()** and

parallel()

Overview of the Retail Checkout Service

Checkout Service(BackEnd)



ParallelStreams
How it works ?

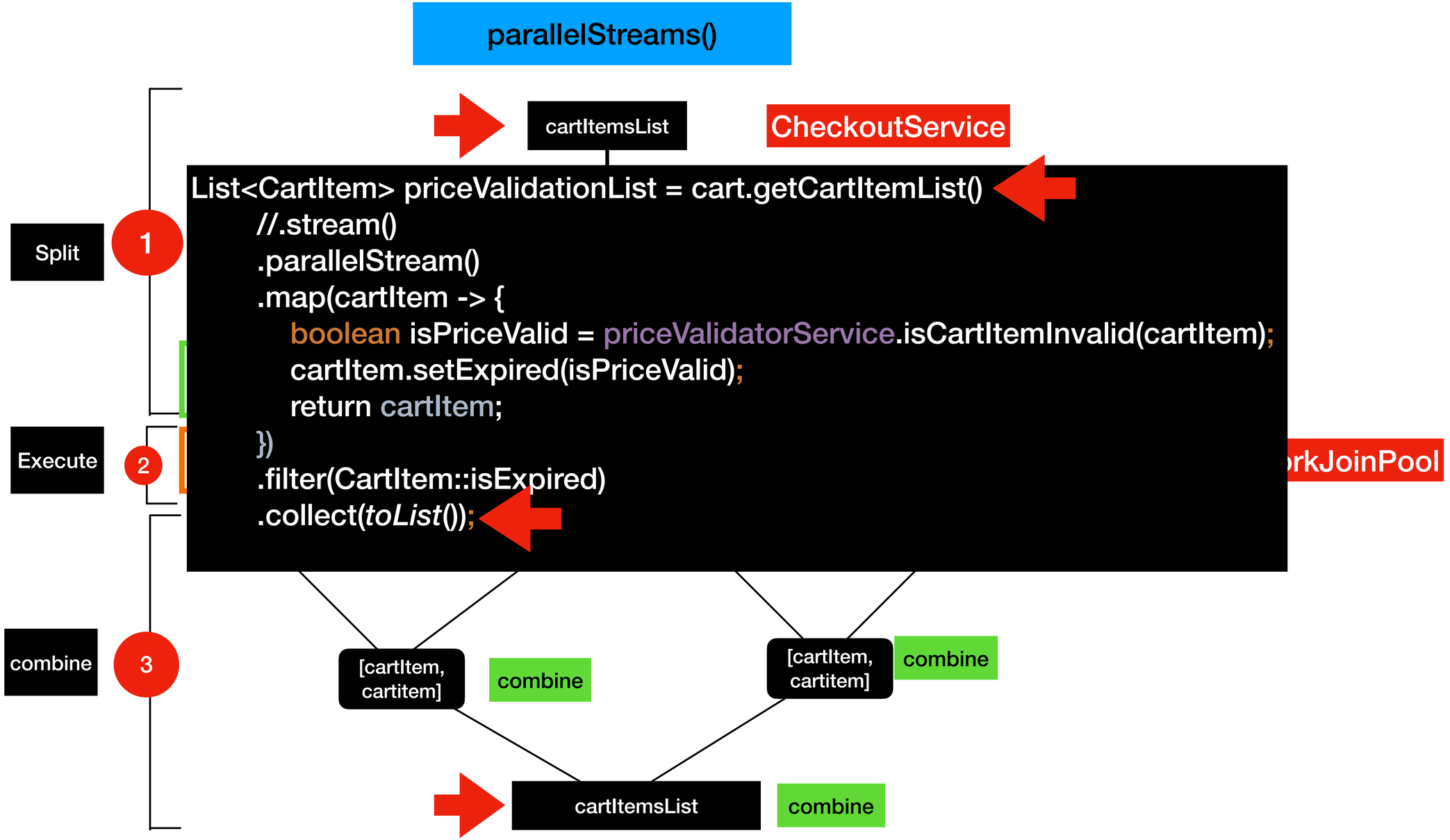
ParallelStreams - How it works ?

- **parallelStream()**
  - **Split** the data in to chunks
  - **Execute** the data chunks
  - **Combine** the result

parallelStream() - How it works ?

- **Split**
  - Data Source is split in to small data chunks
    - Example - **List Collection** split into chunks of elements to **size 1**
  - This is done using **Spliterators**
    - For ArrayList, the **Spliterator** is **ArrayListSpliterator**
- **Execute**
  - Data chunks are applied to the Stream Pipeline and the **Intermediate** operations executed in a **Common ForkJoin Pool**
  - Watch the **Fork/Join FrameWork** Lectures
- **Combine**
  - Combine the executed results into a final result
  - Combine phase in Streams API maps to **terminal** operations
  - Uses collect() and reduce() functions
    - **collect(*toList*())**

parallelStream() - How it works ?



Comparing ArrayList vs LinkedList ParallelStreams Performance

Spliterator in ParallelStreams

- Data source is split in to multiple chunks by the Spliterator
- Each and every collection has a different Spliterator Implementation
- Performance differ based on the implementation

Multiply each value in the collection by a user passed value

[1, 2, 3, 4] -> [2, 4, 6, 8]

Value * 2

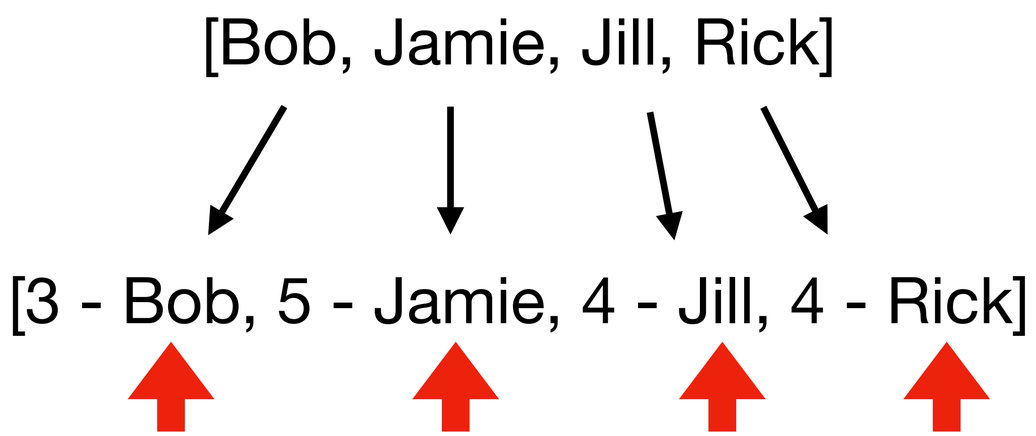
Summary - Spliterator in ParallelStreams

- Invoking **parallelStream()** does not guarantee faster performance of your code
  - Need to perform additional steps compared to sequential
  - Splitting , Executing and Combining

Recommendation - Always compare the performance before you use parallelStream()

Parallel Streams
Final Computation Result Order

Parallel Streams - Final Computation Result Order



Parallel Streams - Final Computation Result Order

- The order of the collection depends on:
  - Type of Collection
  - Spliterator Implementation of the collection
- Example : ArrayList
  - Type of Collection - **Ordered**
  - Spliterator Implementation - Ordered Spliterator Implementation
- Example : Set
  - Type of Collection - **UnOrdered**
  - Spliterator Implementation - UnOrdered Spliterator Implementation

Collect
&
Reduce

Collect( ) vs Reduce( )

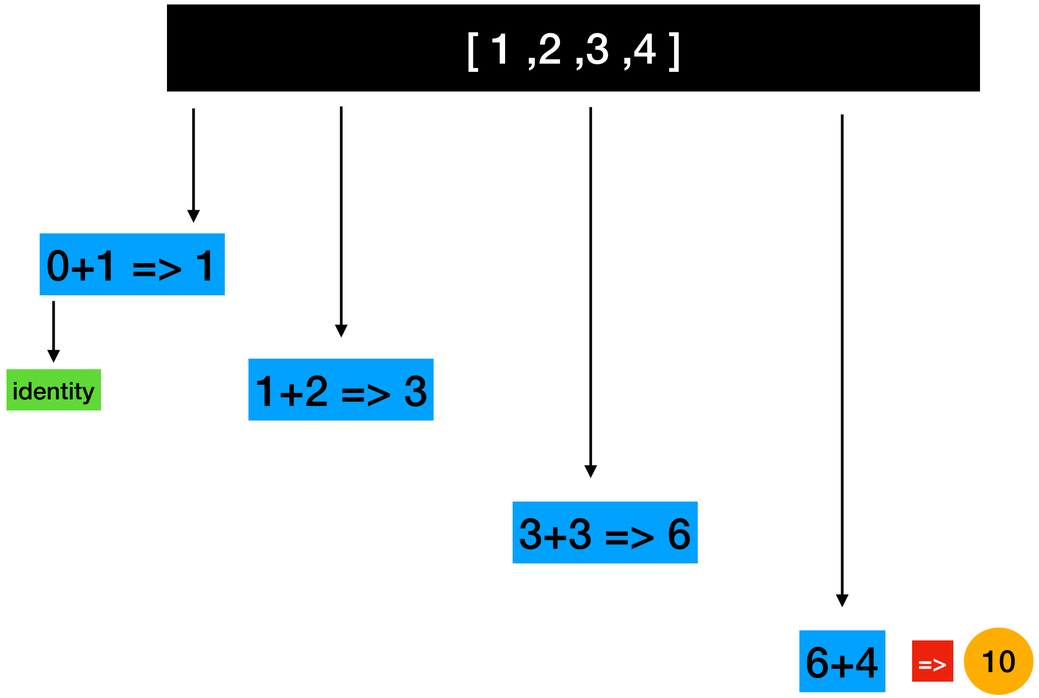
Collect

- Part of Streams API
- Used as a terminal operation in **Streams API**
- Produces a single result
- Result is produced in a mutable fashion
- Feature rich and used for many different use cases
- Example
  - collect(*toList*()), collect(*toSet*())
  - collect(*summingDouble*(Double::doubleValue));

Reduce

- Part of Streams API
- Used as a terminal operation in **Streams API**
- Produces a single result
- Result is produced in a immutable fashion
- Reduce the computation into a single value
  - Sum, Multiplication
  - Example
    - Sum -> reduce(0.0, **(x , y)->x+y**)
    - Multiply -> reduce(1.0, **(x , y)->x * y**)

How reduce() works ?




```java
public static int reduce(){
    int sum= List.of(1,2,3,4)
            .stream()
            .reduce( identity: 0, (x,y)->x+y);
    return sum;
}
```

10

The reduce( ) function performs an immutable computation throughout in each and every step.

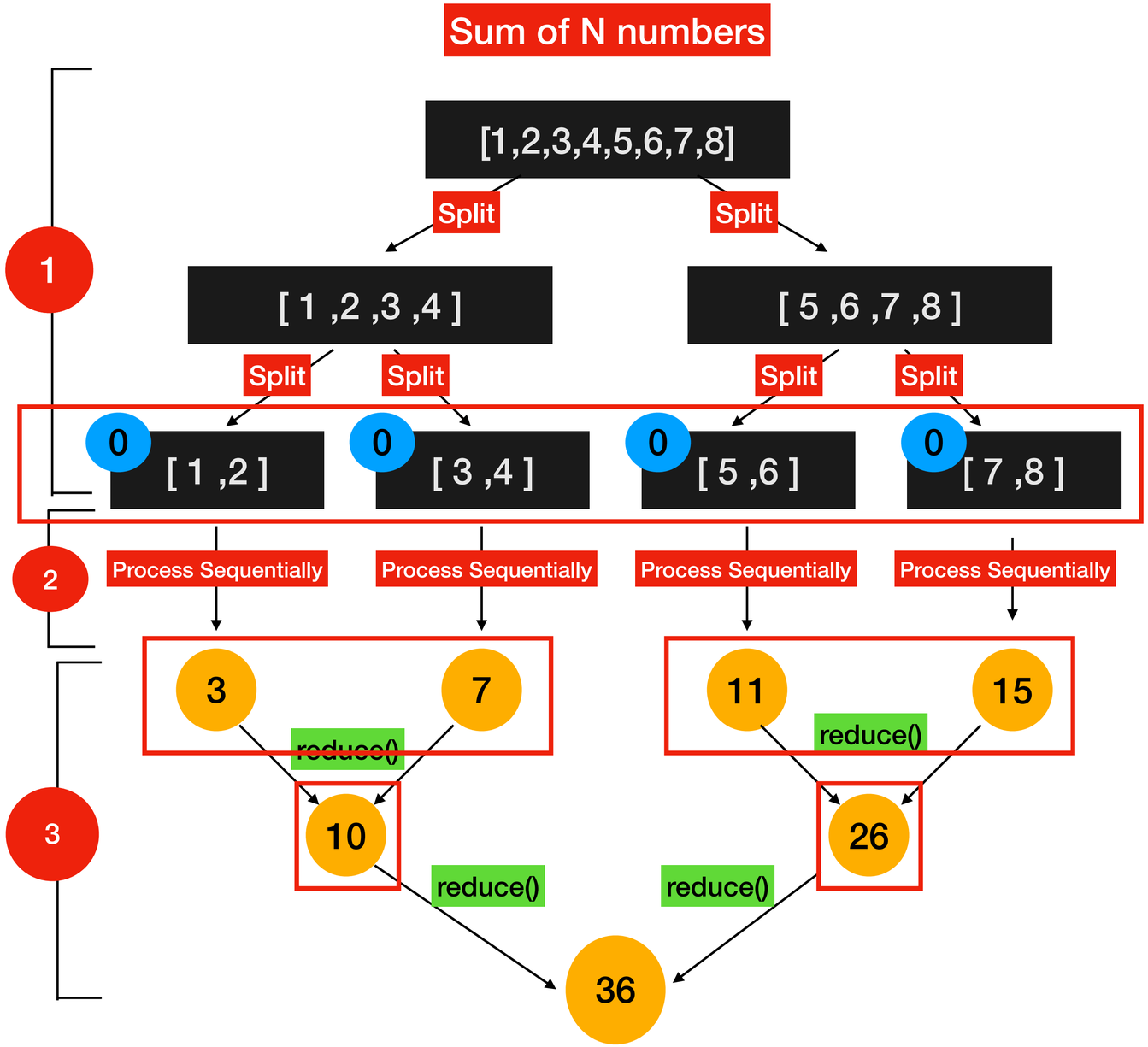
How reduce() with ParallelStream works ?

Sum of N numbers - reduce() and parallelStream()

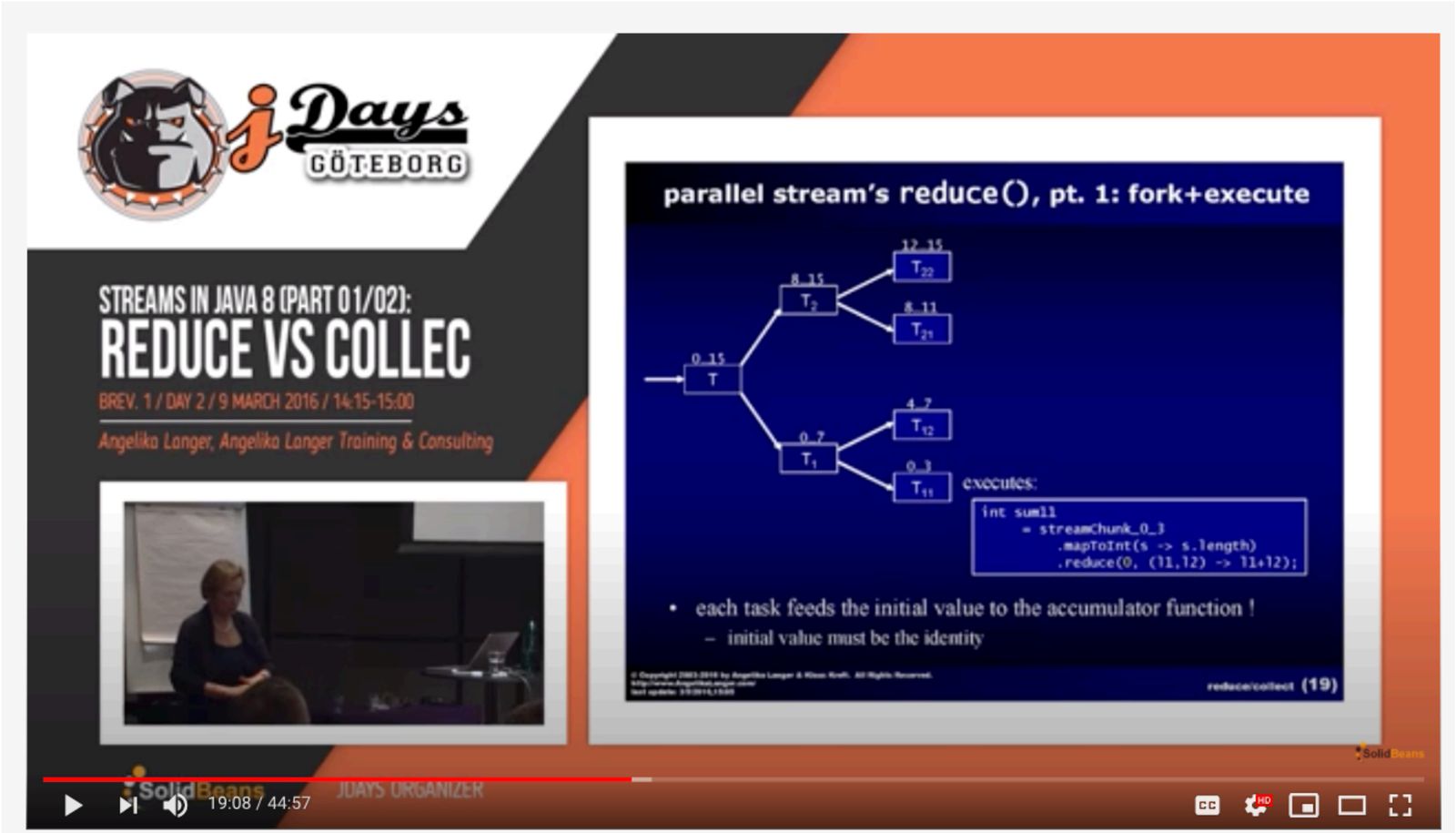
```
public static int reduce_ParallelStream(){

    int sum= List.of(1,2,3,4,5,6,7, 8)
            .parallelStream()
            .reduce( identity: 0, (x,y)->x+y);
    return sum; 36
}
```

1. Spliterator
2. ForkJoinPool
3. Reduce



Want to learn more ?



Streams in Java 8: Reduce vs. Collect

12,839 views • Apr 28, 2016

243 6 SHARE SAVE

Collect() & Reduce()
Hands-On

Identity in reduce()

Identity in reduce()

- Identity gives you the same value when its used in the computation
  - Addition: **Identity = 0**
    - 0 + 1 => 1
    - 0 + 20 => 20
  - Multiplication : **Identity = 1**
    - 1 * 1 => **1**
    - 1 * 20 => **20**

Sum of N numbers - reduce() and parallelStream()

```
public static int reduce_ParallelStream(){

    int sum= List.of(1,2,3,4,5,6,7, 8)
            .parallelStream()
            .reduce( identity: 0, (x,y)->x+y);
    return sum;
}
```

reduce() is recommended for computations that are associative

Parallel Stream Operations
&
Poor Performance

Parallel Stream Operations & Poor Performance

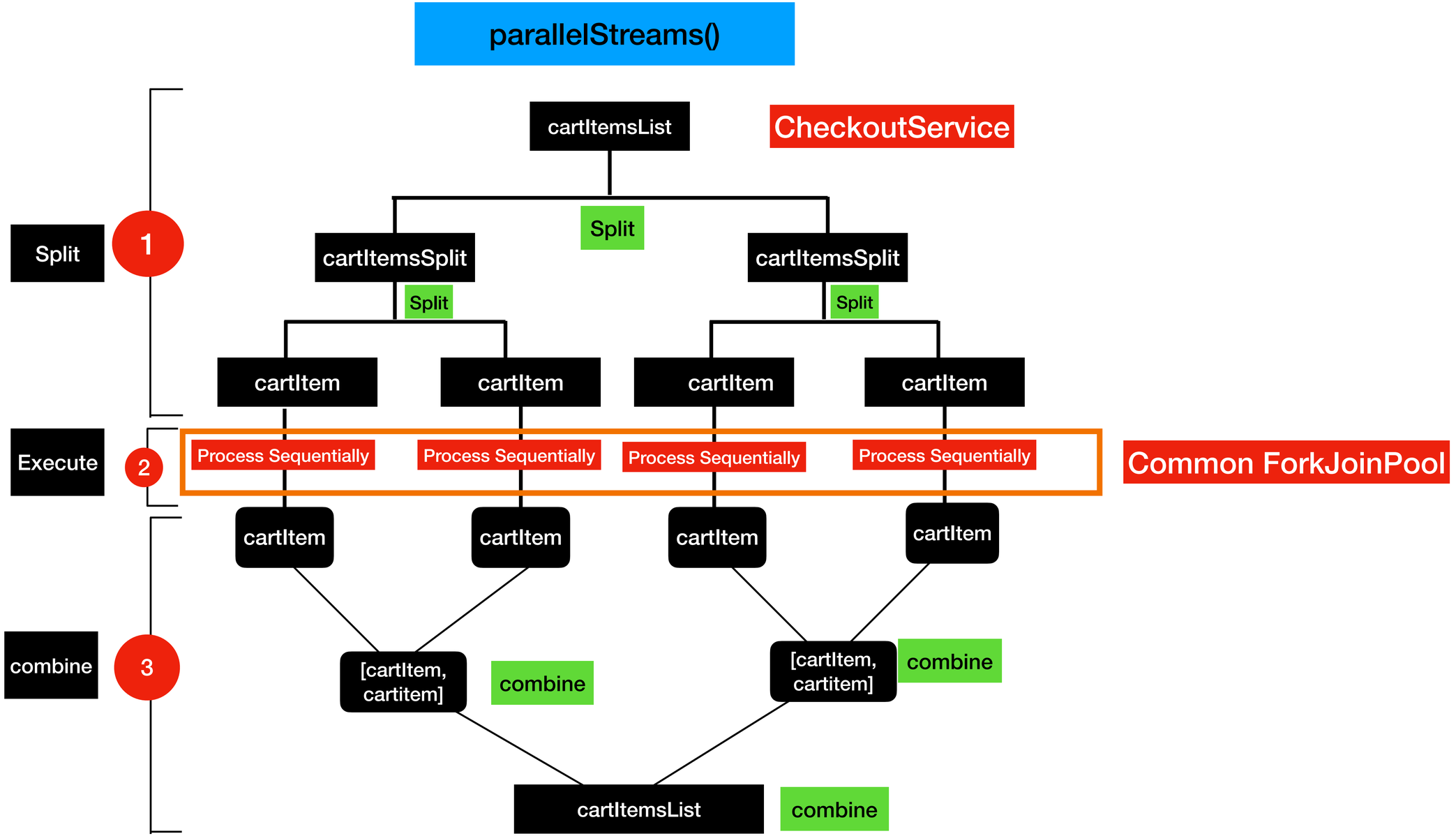
- Stream Operations that perform poor
- Impact of **Boxing** and **UnBoxing** when it comes to parallel Streams
  - **Boxing** -> Converting a Primitive Type to Wrapper class equivalent
    - **1** -> **new Integer(1)** 
  - **UnBoxing** -> Converting a Wrapper class to Primitive equivalent
    - **new Integer(1)** -> **1** 

Common ForkJoin Pool

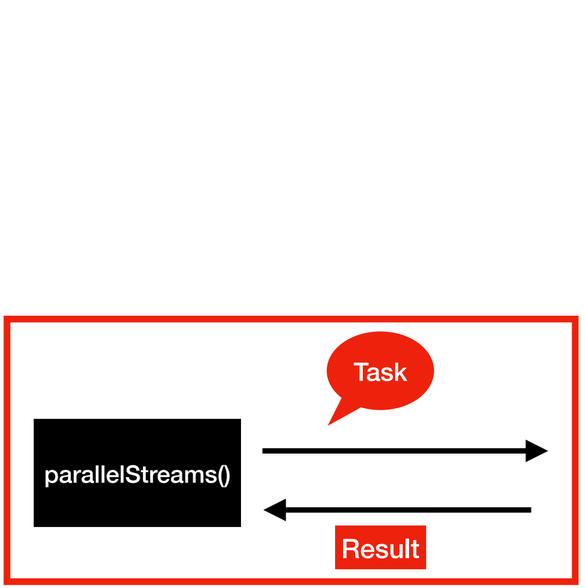
Common ForkJoin Pool

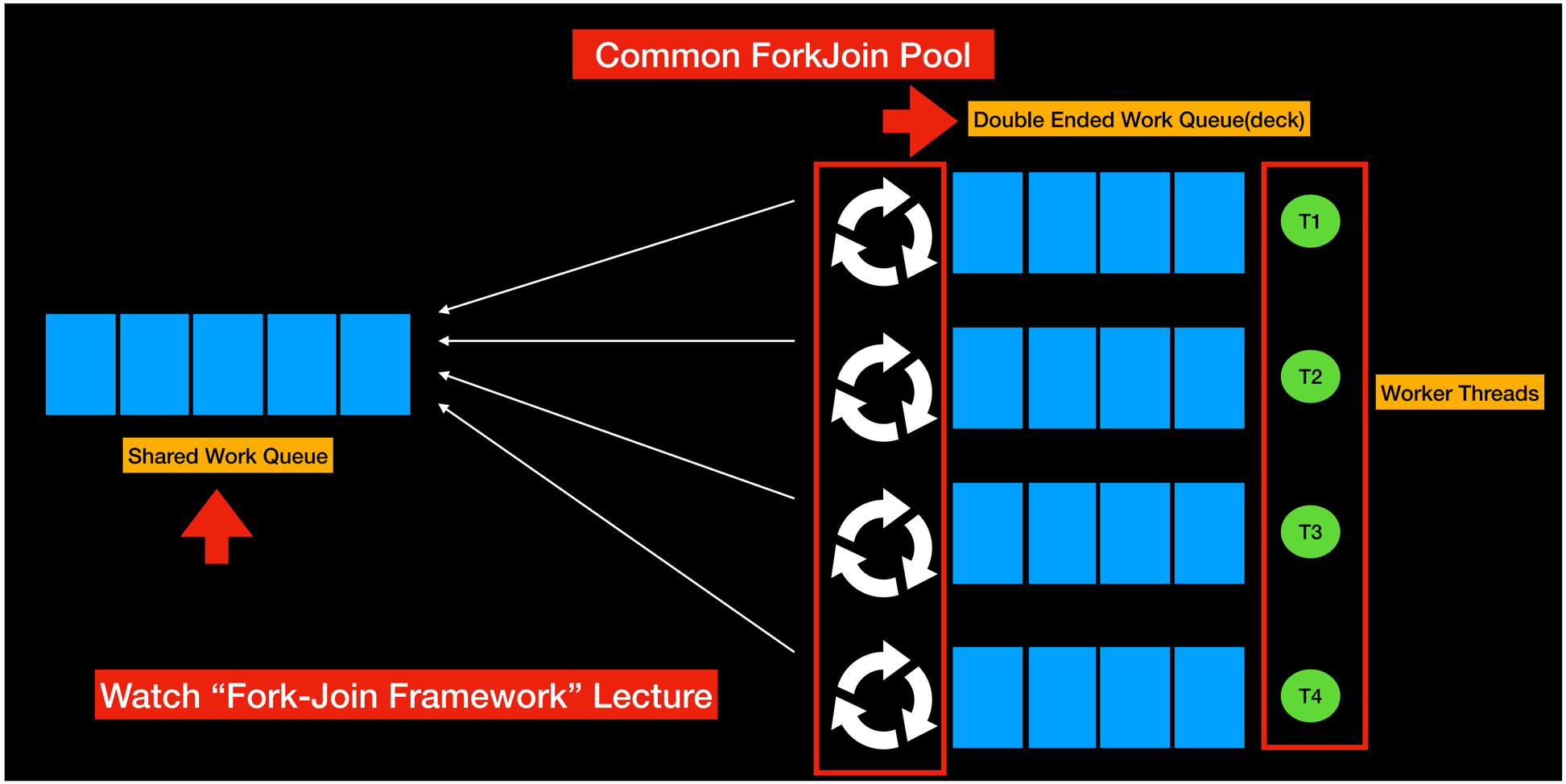
Execution Engine for Parallel Streams

parallelStream() - How it works ?



Common ForkJoin Pool





Common ForkJoin Pool

- Common ForkJoin Pool is used by:
  - ParallelStreams
  - CompletableFuture
    - Completable Future have options to use a User-defined ThreadPools
  - Common ForkJoin Pool is shared by the whole process

Parallelism
&
Threads in
Common ForkJoinPool

Parallelism & Threads in Common ForkJoinPool

- parallelStreams()
  - Runs your code in parallel
  - Improves the performance of the code
- Is there a way to look in to parallelism and threads involved ?
  - **Yes**

Modifying Default parallelism in Parallel Streams

Modifying Default parallelism

```
System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "100");
```

OR

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=100
```

Parallel Streams - Summary

Parallel Streams - When to use them ?

Parallel Streams - When to use them ?

- Parallel Streams do a lot compared to sequential(default) Streams
- Parallel Streams
  - Split 
  - Execute
  - Combine 

Parallel Streams - When to use them ?

- Computation takes a longer time to complete
- Lots of data
- More cores in your machine

Always compare the performance between sequential and parallel streams

Parallel Streams - When to use them ?

Parallel Streams - When not to use them ?

- Parallel Streams
  - Split ⬅
  - Execute
  - Combine ⬅
- Data set is small
- Auto Boxing and Unboxing doesn’t perform better
- Stream API operators -> **iterate**(), **limit**()

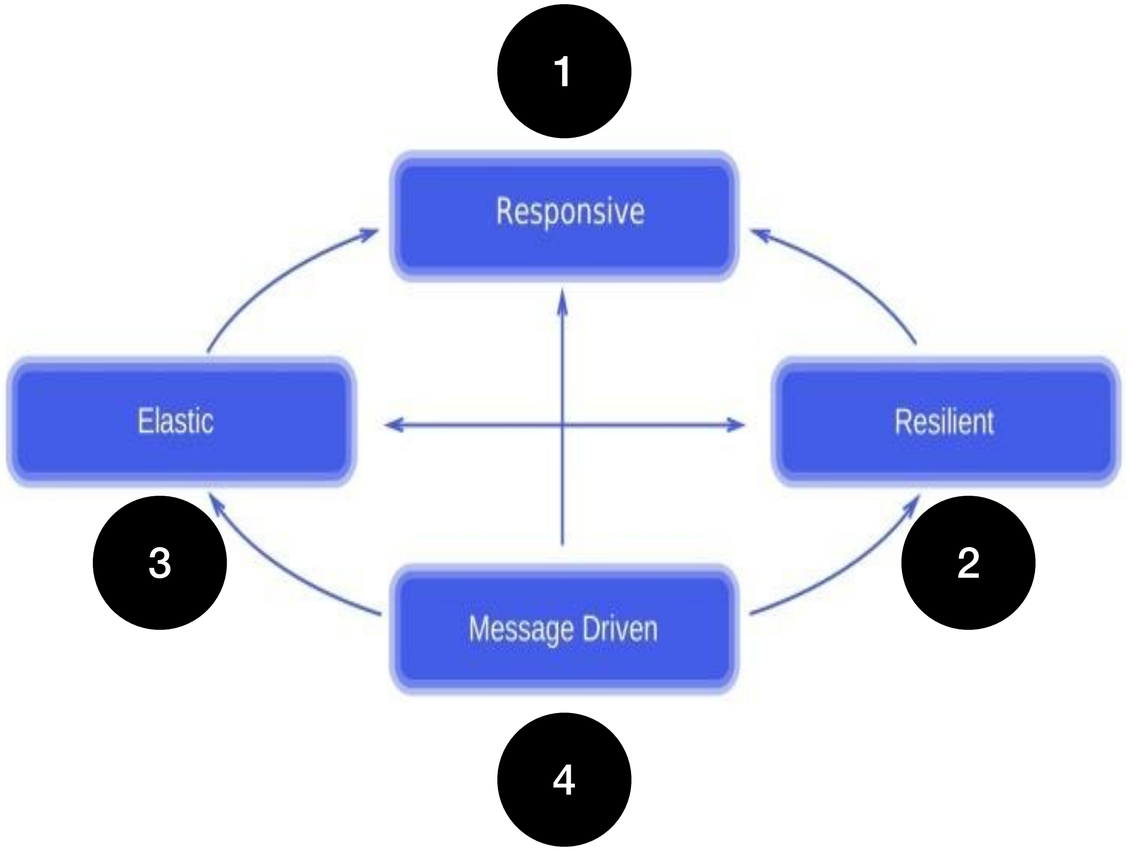
CompletableFuture

CompletableFuture

- Introduced in **Java 8**
- CompletableFuture is an **Asynchronous Reactive Functional Programming API**
- Asynchronous Computations in a functional Style
- CompletableFutures API is created to solve the limitations of Future API

CompletableFuture and Reactive Programming

- **Responsive**:
  - Fundamentally Asynchronous
  - Call returns immediately and the response will be sent when its available
- **Resilient**:
  - Exception or error won’t crash the app or code
- **Elastic**:
  - Asynchronous Computations normally run in a pool of threads
  - No of threads can go up or down based on the need
- **Message Driven:**
  - Asynchronous computations interact with each through messages in a event-driven style



CompletableFuture API

- **Factory Methods** 
  - Initiate asynchronous computation 
- **Completion Stage Methods** 
  - Chain asynchronous computation 
- **Exception Methods** 
  - Handle Exceptions in an Asynchronous Computation 

Lets Write
our
First CompletableFuture

CompletableFuture

supplyAsync()

- FactoryMethod
- Initiate Asynchronous computation
- Input is **Supplier** Functional Interface
- Returns CompletableFuture<**T**>()

thenAccept()

- CompletionStage Method
- Chain Asynchronous Computation
- Input is **Consumer** Functional Interface
  - Consumes the result of the previous
- Returns CompletableFuture<**Void**>
- Use it at the end of the Asynchronous computation

thenApply()

thenApply()

- Completion Stage method
- Transform the data from one form to another
- Input is **Function** Functional Interface
- Returns CompletableFuture<**T**>

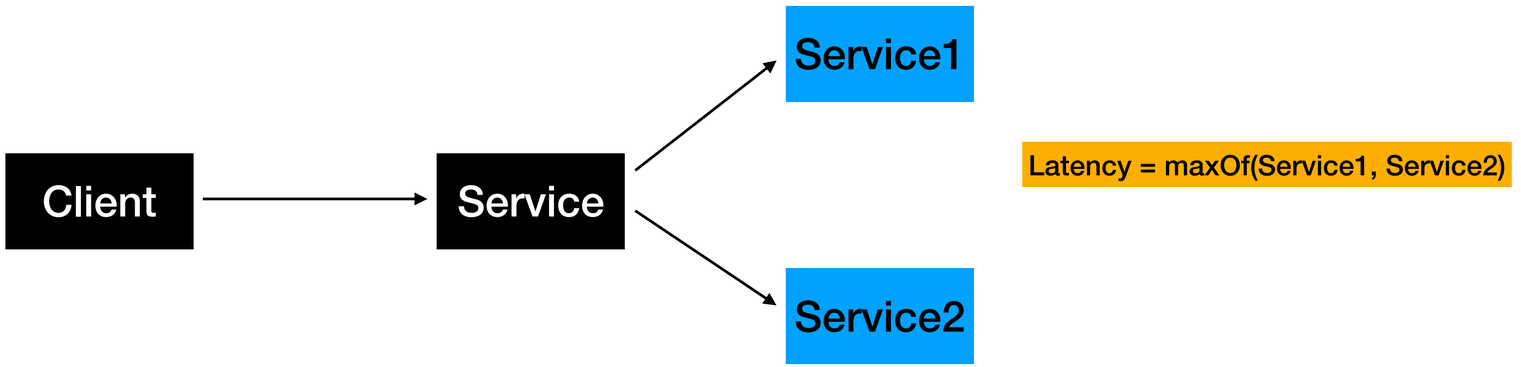
CompletableFuture

Unit Testing CompletableFuture

Combing independent Async Tasks using “thenCombine"

thenCombine()

- This is a Completion Stage Method
- Used to Combine Independent Completable Futures



- Takes two arguments
  - CompletionStage , BiFunction
- Returns a CompletableFuture

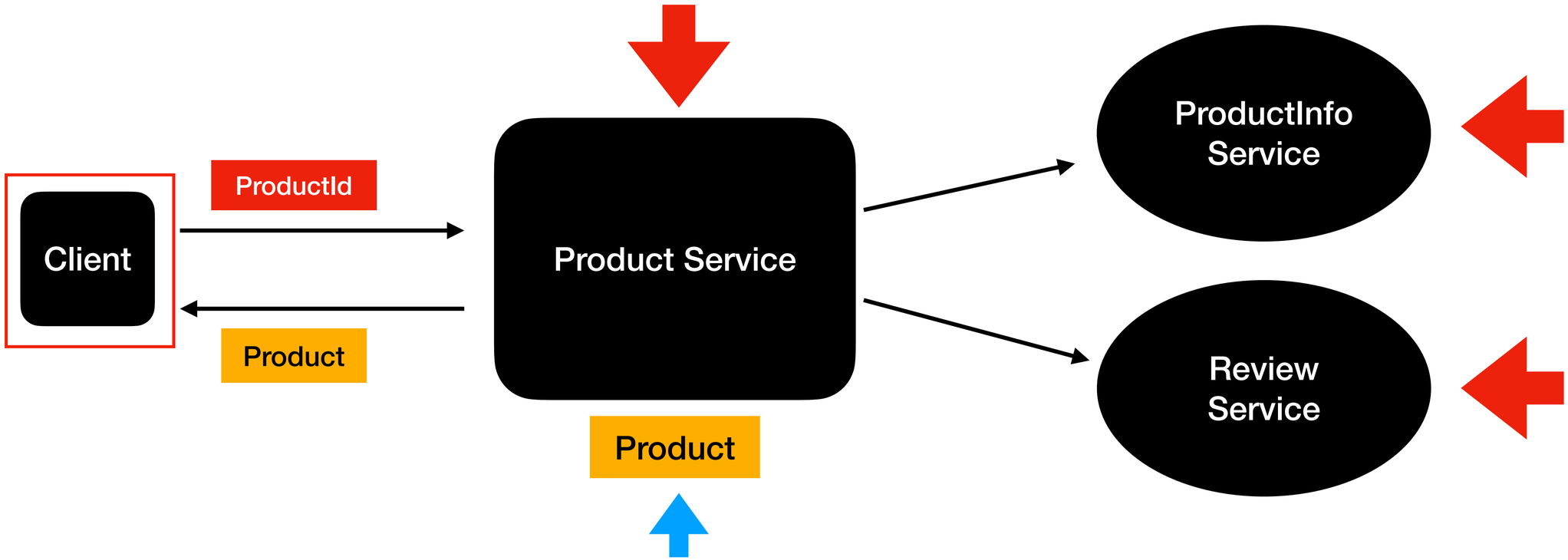
thenCompose

thenCompose()

- Completion Stage method
- Transform the data from one form to another
- Input is **Function** Functional Interface
- Deals with functions that return CompletableFuture
  - thenApply deals with Function that returns a value
- Returns CompletableFuture<**T**>

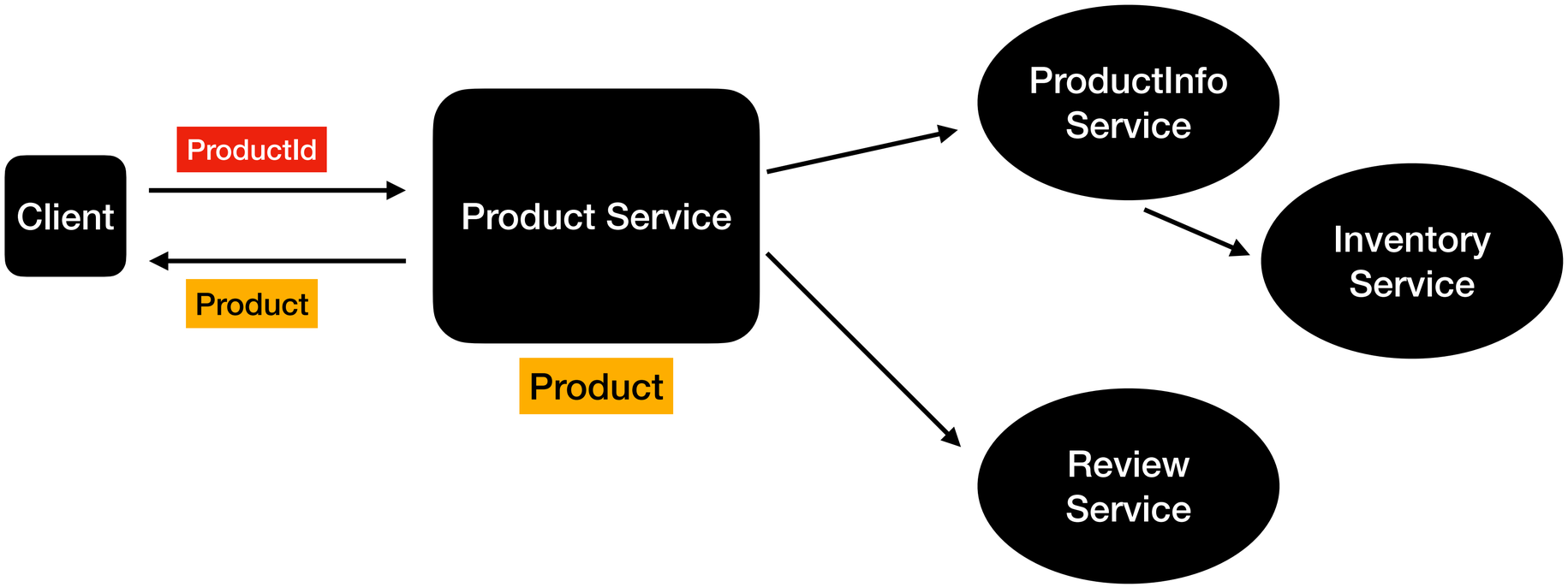
```
public CompletableFuture<String> worldFuture(String input)
{
    return CompletableFuture.supplyAsync(()->{
        delay(1000);
        return input+" world!";
    });
}
```

Product Service



Combining Streams & CompletableFuture

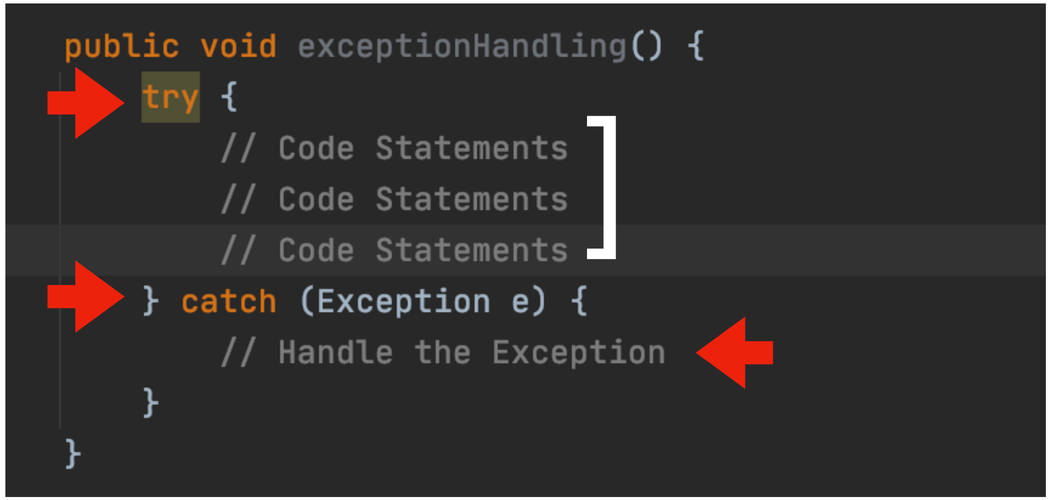
Product Service with Inventory



Exception Handling In CompletableFuture

Exception Handling in Java

- Exception Handling in Java is available since the inception of **Java**



Exception Handling in CompletableFuture

- CompletableFuture is a functional style API

```
public String helloWorld_3_async_calls() {
    CompletableFuture<String> hello = CompletableFuture.supplyAsync(() -> this.hws.hello());
    CompletableFuture<String> world = CompletableFuture.supplyAsync(() -> this.hws.world());
    CompletableFuture<String> hiCompletableFuture = CompletableFuture.supplyAsync(() -> {
        delay( delayMilliSeconds: 1000);
        return " HI CompletableFuture!";
    });

    String hw = hello
            .thenCombine(world, (h, w) -> h + w) // (first,second)
            .thenCombine(hiCompletableFuture, (previous, current) -> previous + current)
            .thenApply(String::toUpperCase)
            .join();
    return hw;
}
```

Exception Handling in CompletableFuture

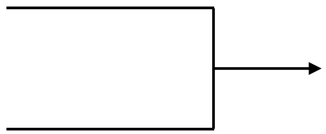
try/catch

```
public String helloWorld_3_async_calls1() {
    try{
        CompletableFuture<String> hello = CompletableFuture.supplyAsync(() -> this.hws.hello());
        CompletableFuture<String> world = CompletableFuture.supplyAsync(() -> this.hws.world());
        CompletableFuture<String> hiCompletableFuture = CompletableFuture.supplyAsync(() -> {
            delay( delayMilliSeconds: 1000);
            return " HI CompletableFuture!";
        });

        String hw = hello
                .thenCombine(world, (h, w) -> h + w) // (first,second)
                .thenCombine(hiCompletableFuture, (previous, current) -> previous + current)
                .thenApply(String::toUpperCase)
                .join();
        return hw;
    }catch (Exception e){
        log("Exception is " + e);
        throw e;
    }
}
```

Exception Handling in CompletableFuture

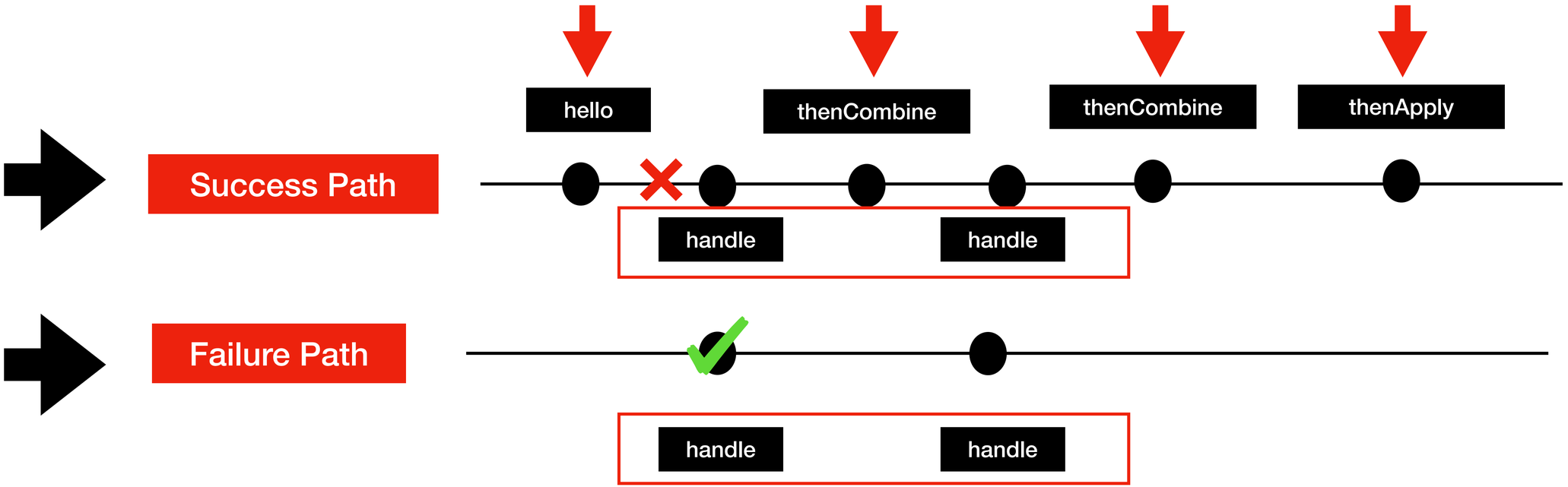
- CompletableFuture API has functional style of handling exceptions
- Three options available:
  - handle()
  - exceptionally()
  - whenComplete()



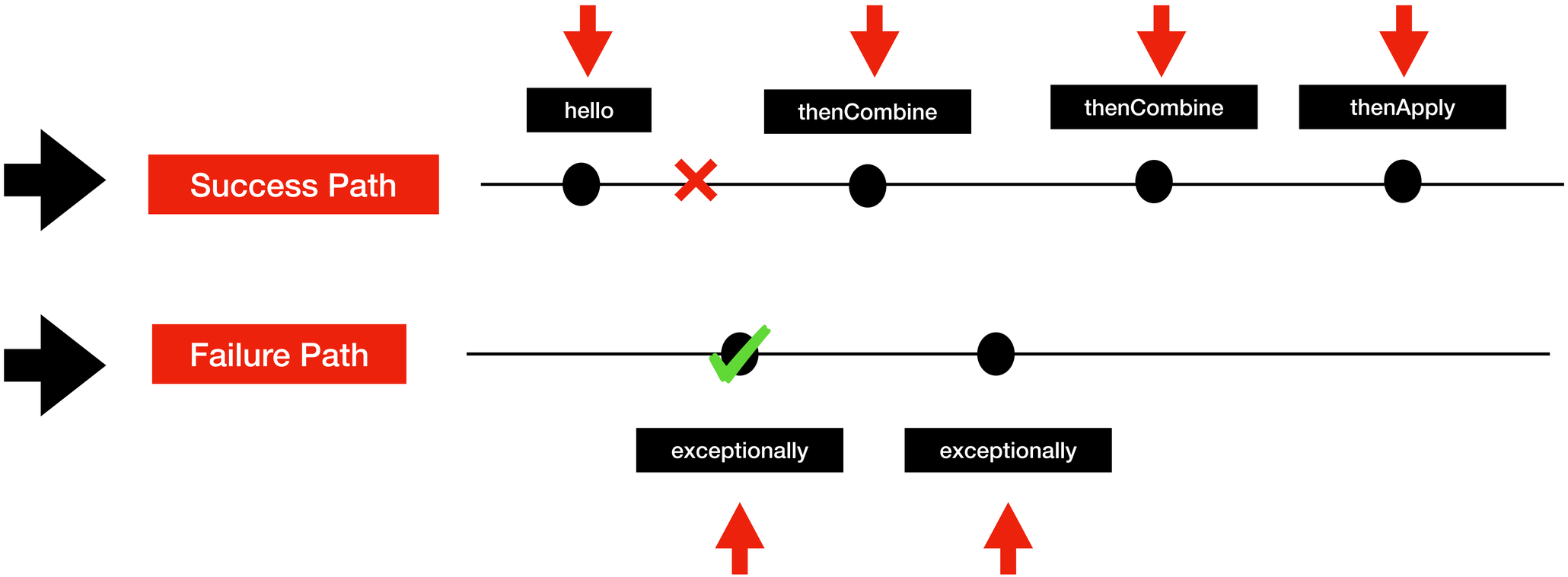
Catch Exception and Recover

Catch Exception and Does not Recover

Exception Handling using handle()



Exception Handling using exceptionally()



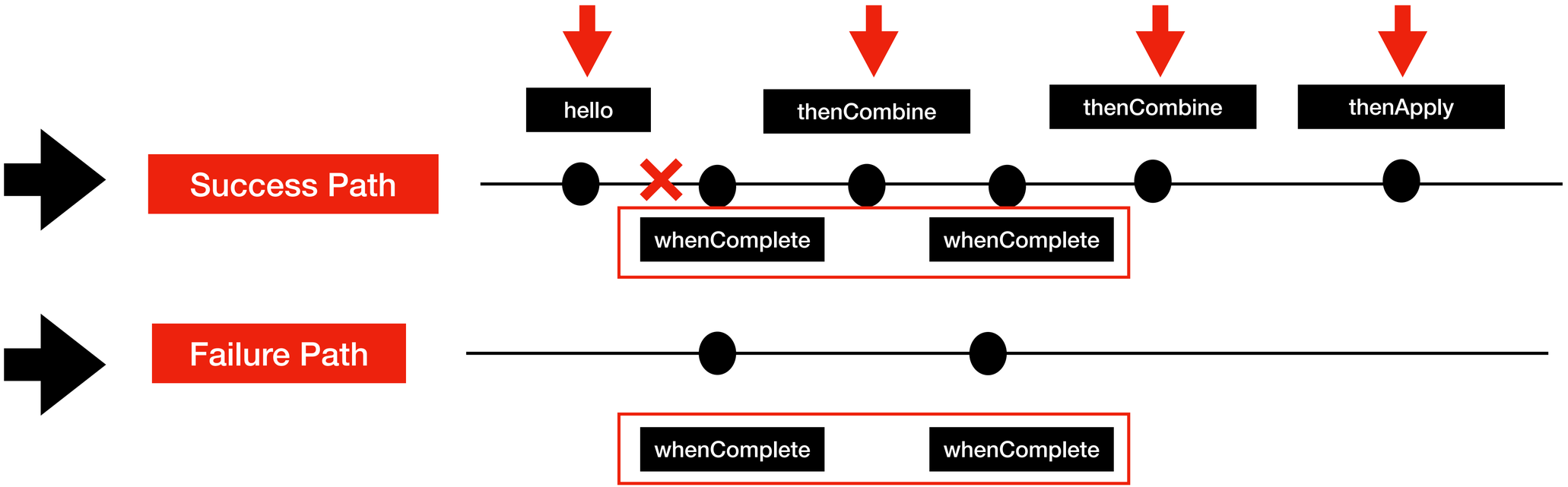
whenHandle()

whenHandle()

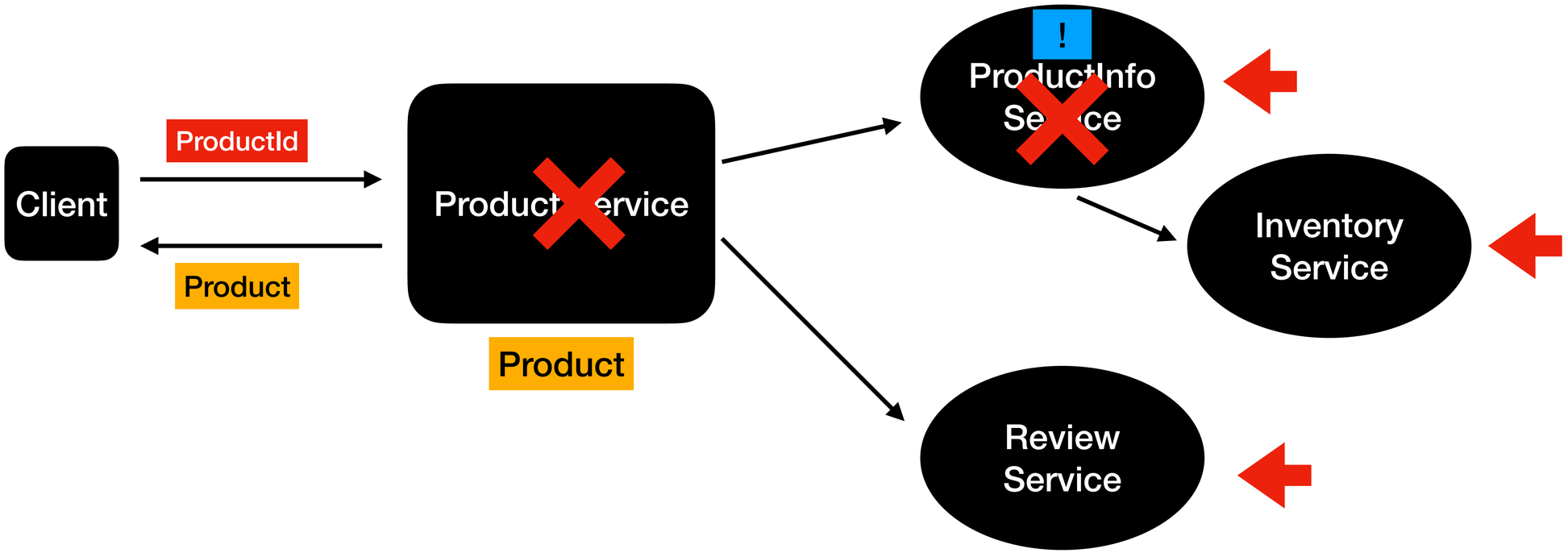
- Exception handler in CompletableFuture API

- Catches the Exception but does not recover from the exception

Exception Handling using whenComplete()



Product Service with Inventory

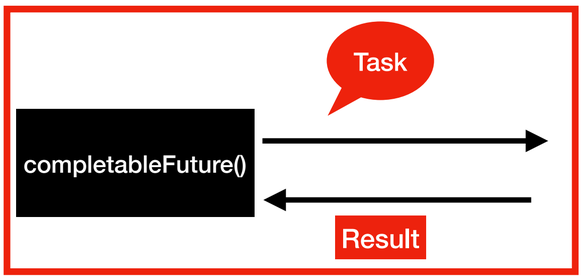


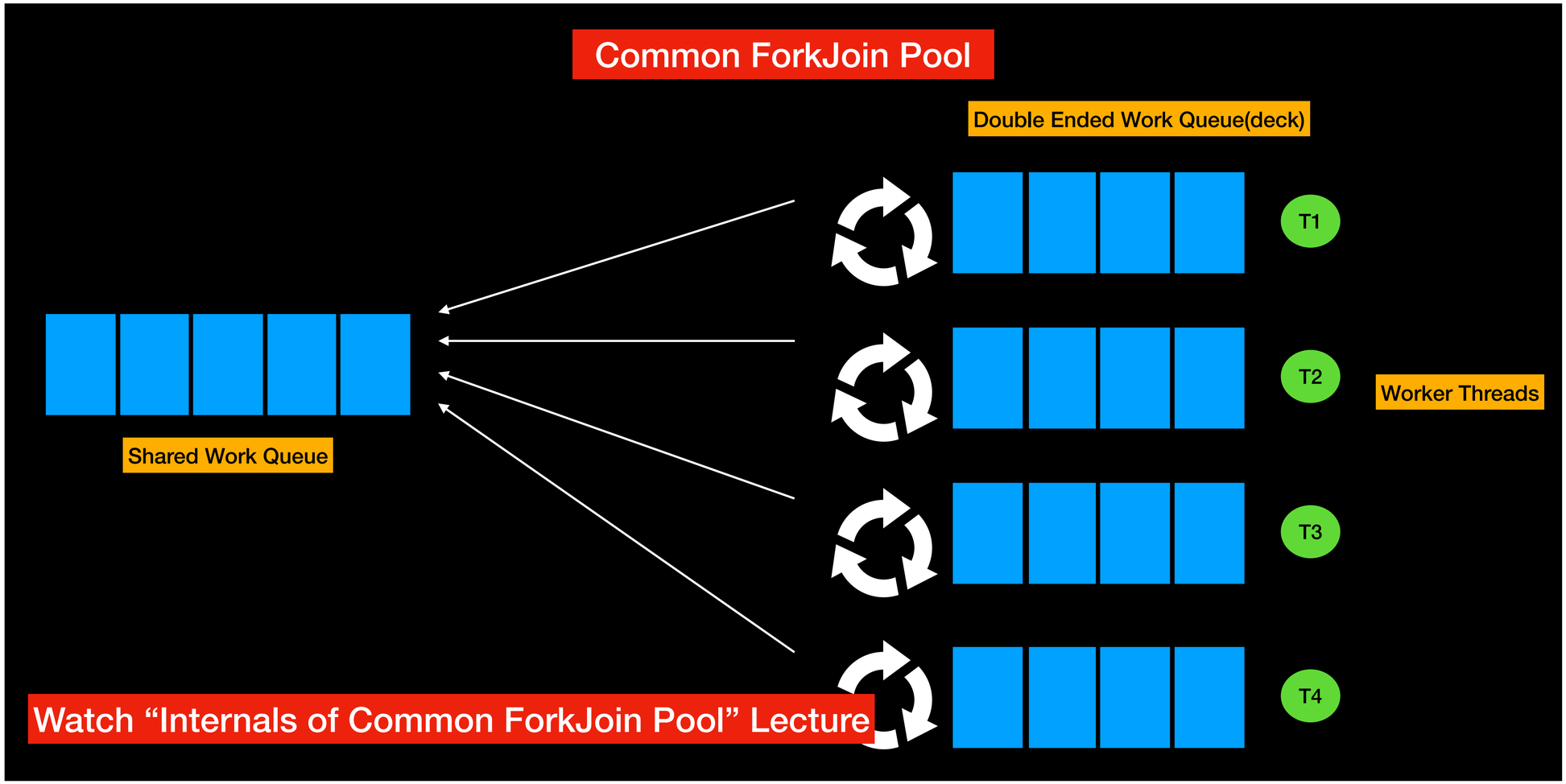
CompletableFuture
Default ThreadPool

CompletableFuture - ThreadPool

- By default, CompletableFuture uses the **Common ForkJoinPool**
  - The no of threads in the pool == number of cores

Common ForkJoin Pool





CompletableFuture
&
User Defined ThreadPool
using
ExecutorService

Why use a different ThreadPool ?

- Common ForkJoinPool is shared by
  - ParallelStreams
  - CompletableFuture
- Its common for applications to use **ParallelStreams** and **CompletableFuture** together
  - The following issues may occur:
    - Thread being blocked by a time consuming task
    - Thread not available

Creating a User-Defined ThreadPool

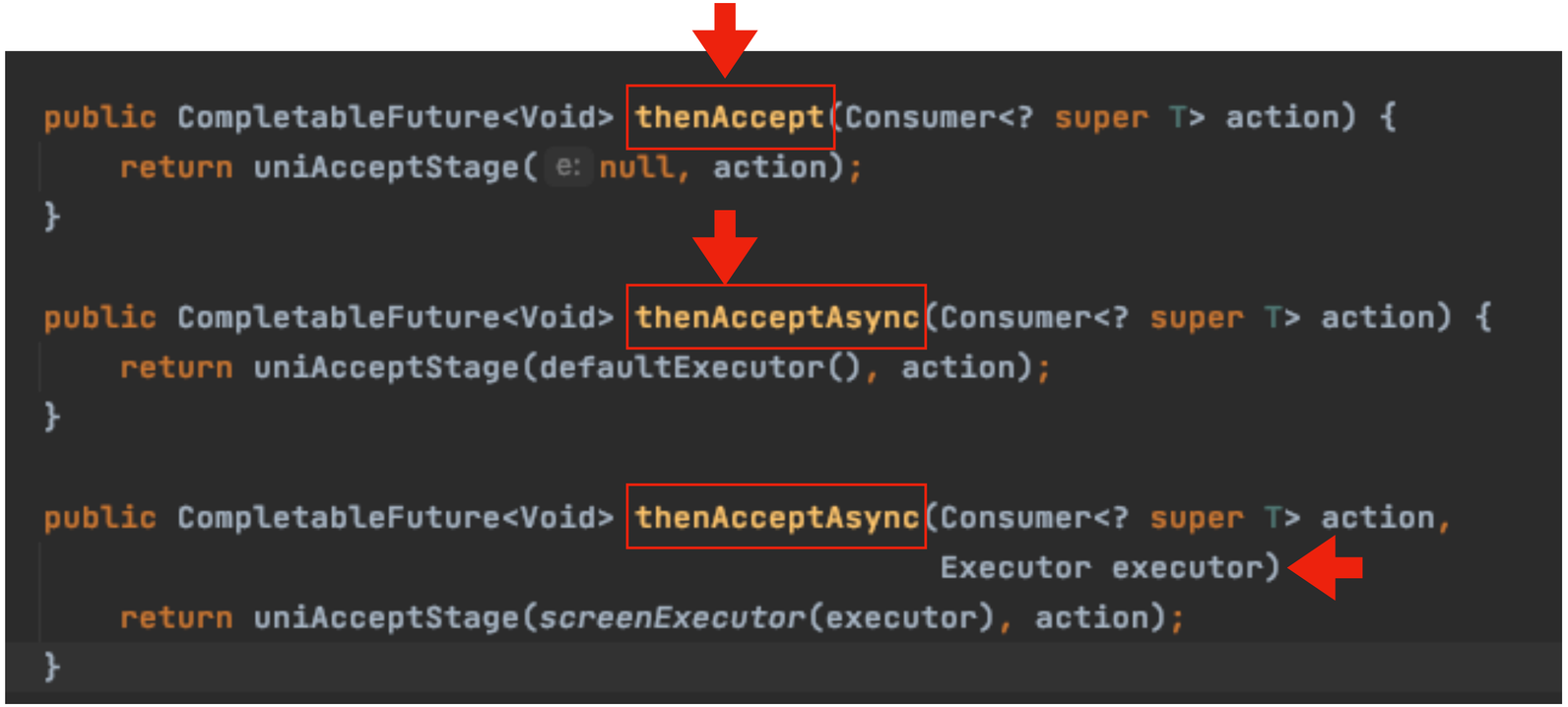
```
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
```

Threads
In
CompletableFuture

Async()
Overloaded Functions
In
CompletableFuture

Async Overloaded Functions

- thenAccept()



```
public CompletableFuture<Void> thenAccept(Consumer<? super T> action) {
    return uniAcceptStage( e: null, action);
}

public CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action) {
    return uniAcceptStage(defaultExecutor(), action);
}

public CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action,
                                               Executor executor)
    return uniAcceptStage(screenExecutor(executor), action);
}
```

Async() Overloaded Functions

Regular Functions

- thenCombine()
- thenApply()
- thenCompose()
- thenAccept()

Async() overloaded Functions

- thenCombineAsync()
- thenApplyAsync()
- thenComposeAsync()
- thenAcceptAsync()

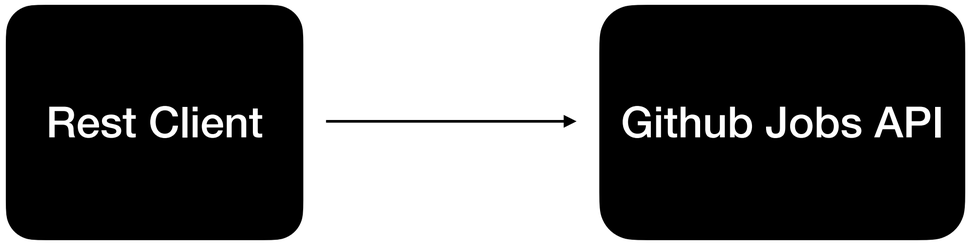
Async() Overloaded Functions

- Using async() functions allows you to change the thread of execution

- Use this when you have blocking operations in your Completablefuture pipeline

Introduction to Spring WebClient and Overview of the GitHub Jobs API

About this section



Build Restful API Using Spring WebClient

Why Spring WebClient?

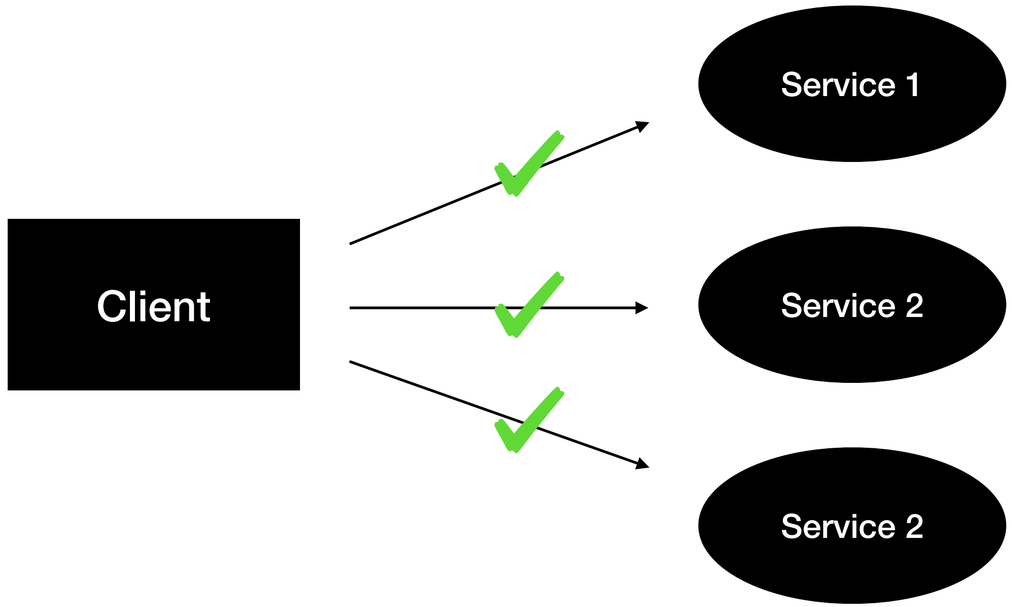
- Spring is one of the popular framework in the Java Community
- **Spring WebClient** is a rest client library that’s got released as part of **Spring 5**
- **Spring WebClient** is a functional style RestClient
- **Spring WebClient** can be used as a blocking or non blocking Rest Client

Github Jobs API

allOf()

allOf() - Dealing with Multiple CompletableFutures

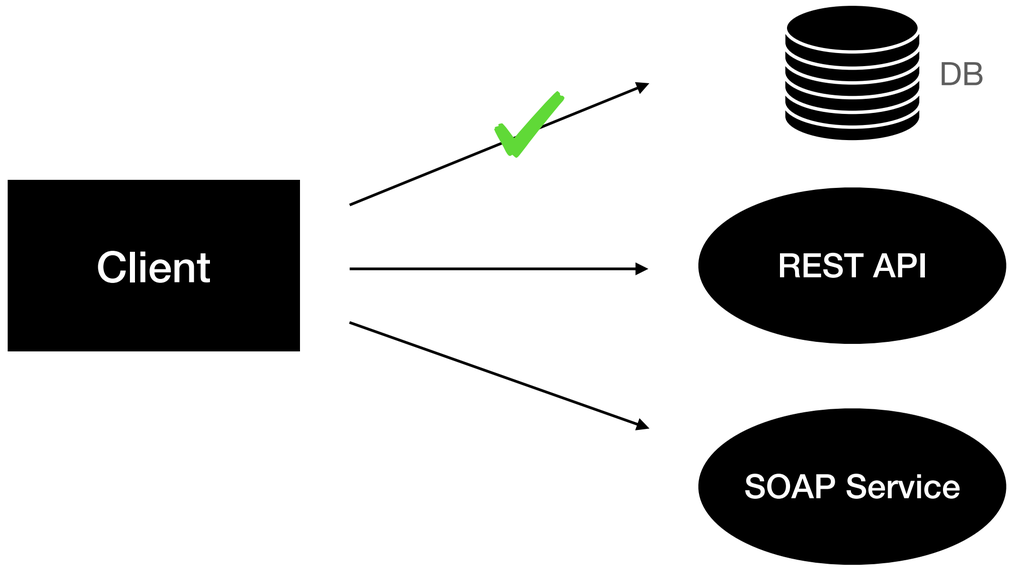
- static method that's part of CompletableFuture API
- Use allOf() when you are dealing with Multiple CompletableFuture



anyOf()

anyOf() - Dealing with Multiple CompletableFutures

- static method that's part of CompletableFuture API
- Use anyOf() when you are dealing with retrieving data from multiple Data Sources



TimeOuts
In
CompletableFuture

Timeouts in CompletableFuture

- Asynchronous tasks may run indefinitely
- Used to timeout a task in CompletableFuture
- **orTimeout()** in CompletableFutureAPI