luxoft | training
A DXC Technology Company

JVA-074

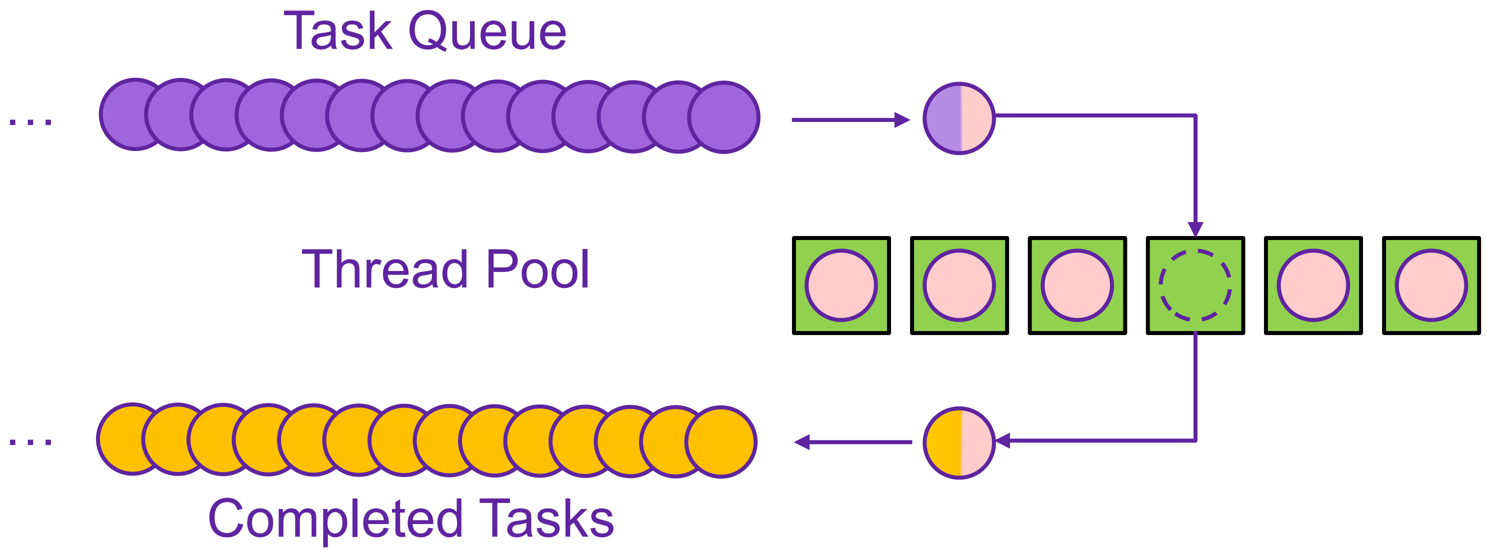
Java Advanced I: Functional, Asynchronous and Reactive Programming

Module 2: Executor Framework, Fork-Join Pool

Executor Framework

Executor Framework

- Manual thread management in a real world application is hard.
- It’s good practice to isolate business from execution logic.
- **Executor Framework** introduces the `Executor` interface that represents some strategies of managing threads.
- There are many `Executor` implementations that represent different strategies.



Using Executors

Class **ThreadPoolExecutor** implements **ExecutorService** and provides the mechanism of thread reusing:

```
ExecutorService executorService1 =
                   Executors.newSingleThreadExecutor();
ExecutorService executorService2 =
                   Executors.newFixedThreadPool(10);
ExecutorService executorService3 =
                   Executors.newScheduledThreadPool(10);
ExecutorService executorService3 =
                   Executors.newCachedThreadPool();
```

Using Executors

```
// Use of execute() method
executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});
executorService.execute(()->System.out.println("Asynchronous task")); // THE SAME WITH LAMBDA

executorService.shutdown();

// Use of submit(): Future
Future future = executorService.submit(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

future.get();  //returns null if the task has finished correctly.
```

Future Interface

- Future interface represents the result of computation.
- Future is abstraction over thread.
  - `isDone` – return true if computation is over,
  - `get` – return result of computation; blocks current thread until computations ends,
  - `get(timeout)` – return result of computation; blocks but not longer than timeout,
  - `cancel(mayInterrupt)` – stop task; if parameter is true then just interrupt thread.

Using of Callable interface

```
Future future = executorService.submit(new Callable(){
    public Object call() throws Exception {
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});

try {
      System.out.println("future.get() = " + future.get());
} catch(CancellationException e) {
    System.out.println(“task was cancelled");
}
```

Luxoft training
A DXC Technology Company

Stopping Tasks

Example:
CallableTutor

- Task stops after reaching `return` from `run/call` method – thread return to pool.
- Task throws exception – in most cases thread returns to pool.
- Call `future.cancel(interrupt)` – stops worker thread via interrupt or wait until end if parameter is `false`.

Tasks cancellation

Illustration for the example:

CallableTutor1

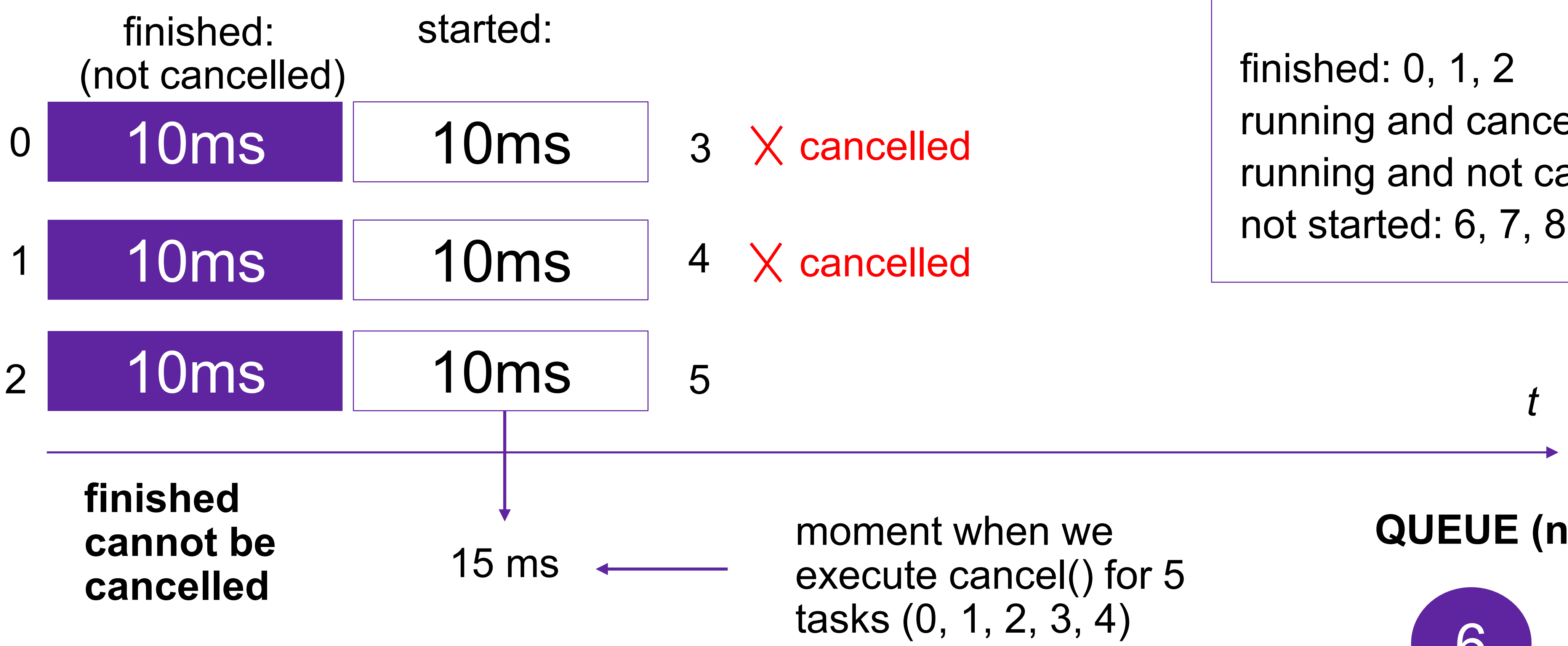
*We executed **10 tasks**, from 0 to 9.*
*One task takes **10 ms** to complete.*
*We have fixed the **thread pool with 3 tasks** running in parallel.*
*Wait for **15 ms**. Try to **cancel() 5 tasks**, 0 to 4.*

*Task state **after canceling**: only **tasks 3 and 4 were canceled**.*
*Tasks **0,1,2 are not canceled**. They are **already finished**.*

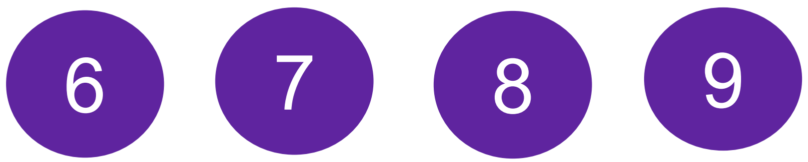
Result:

3 finished, 3 running, 4 not started

finished: 0, 1, 2
running and cancelled: 3, 4
running and not cancelled: 5
not started: 6, 7, 8, 9



QUEUE (not started tasks):



Running Tasks

- There are a few ways to run the task.
- `execute(Runnable)` – fire and forget
- `submit(Runnable)` – returns a **`Future<?>`** that represents task and always returns **`null`**.
- `submit(Callable<T>)` – returns a `Future<T>` that represents task.
- `invokeAll(Collection(Callable<T>))` – returns **`List<Future<T>>`**, all tasks will be executed.
- `invokeAny(Collection(Callable<T>))` – returns result of type T of quickest task; the rest will be **cancelled**.

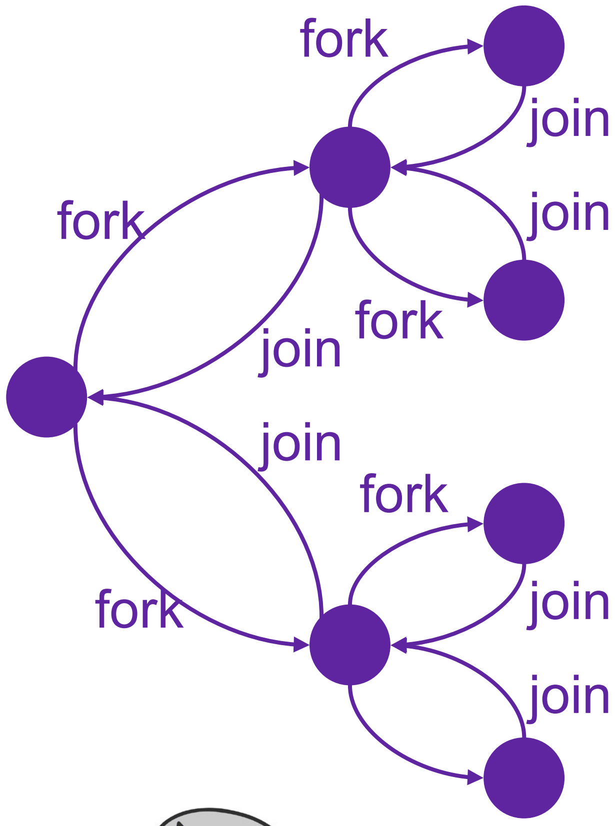
ForkJoin Framework

Why ForkJoin?

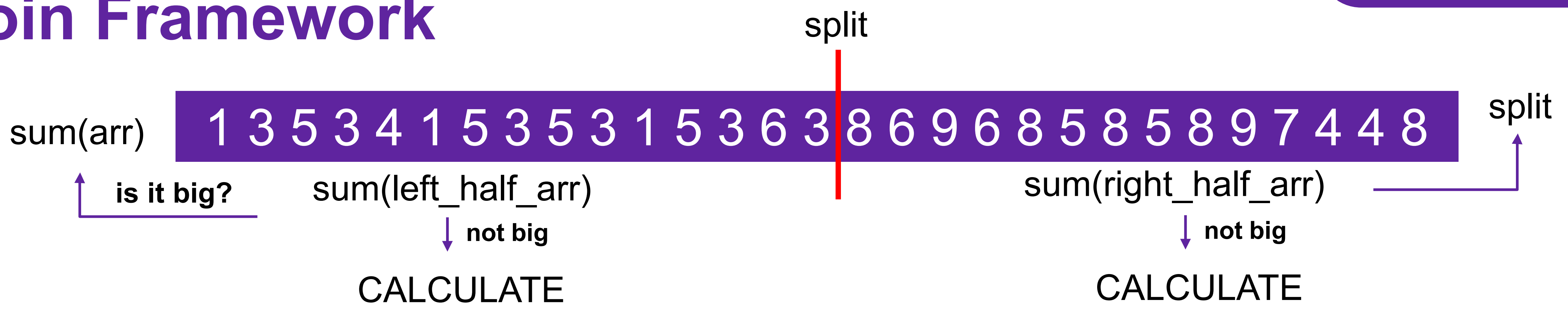
- Working with raw threads is difficult and is a source of strange, hard to locate and fix bugs.
- In Java 5, Sun introduces the Executor Framework to cover most use cases.
- Executor Framework does not solve problem of blocking tasks.
- In Executor Framework, thread waits until the subtask ends its job.
- In Java 7, Oracle introduces the ForkJoin Framework that complements these shortcomings.

ForkJoin Framework – Basics

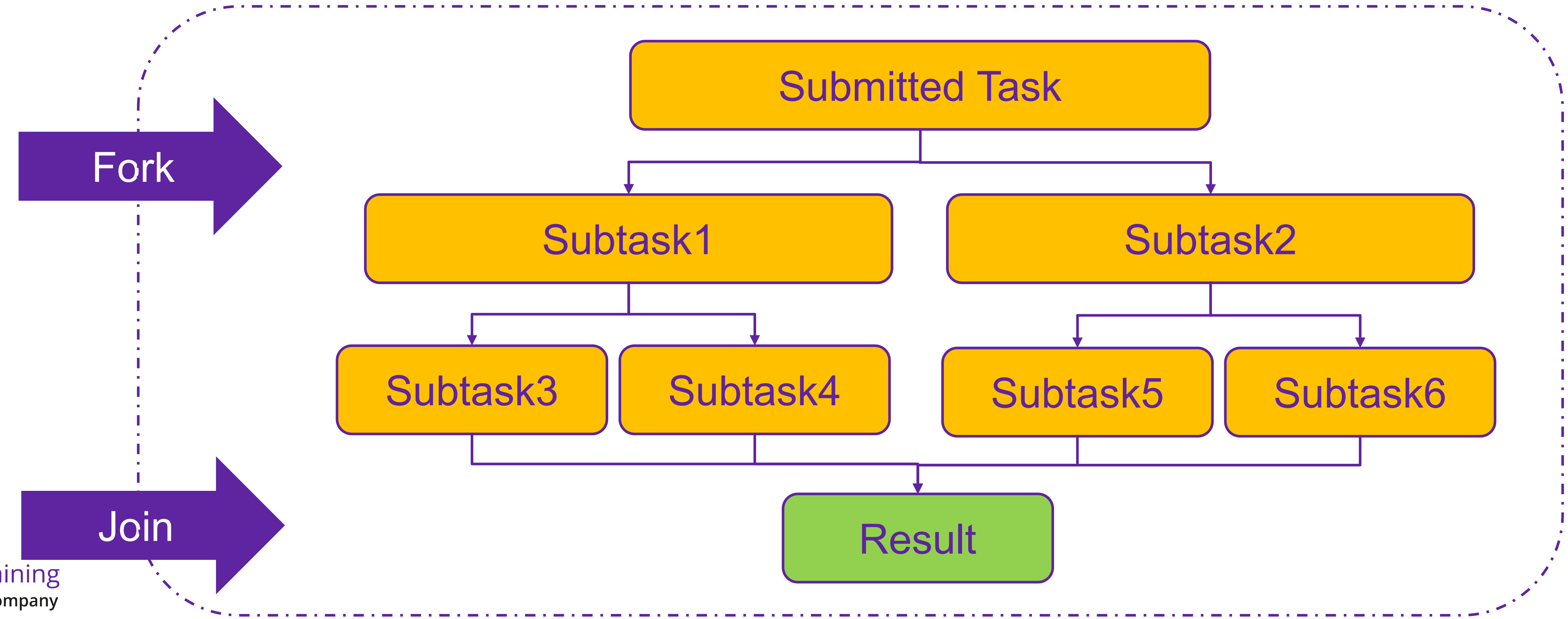
- **ForkJoin Framework** is an implementation of `ExecutorService`.
- It implements a **work-stealing** algorithm:
  - Task - needs to wait for finalization of subtask created by join operation;
  - Executor Framework – worker thread will be waiting;
  - ForkJoin – worker thread will be utilized by executing the next task which is not executed yet.
- ForkJoin framework is based on two operations:
  - `fork` – divide the problem into smaller parts and solve it using framework;
  - `join` – waits for the finalization of created tasks.
- **The Divide and conquer pattern.**



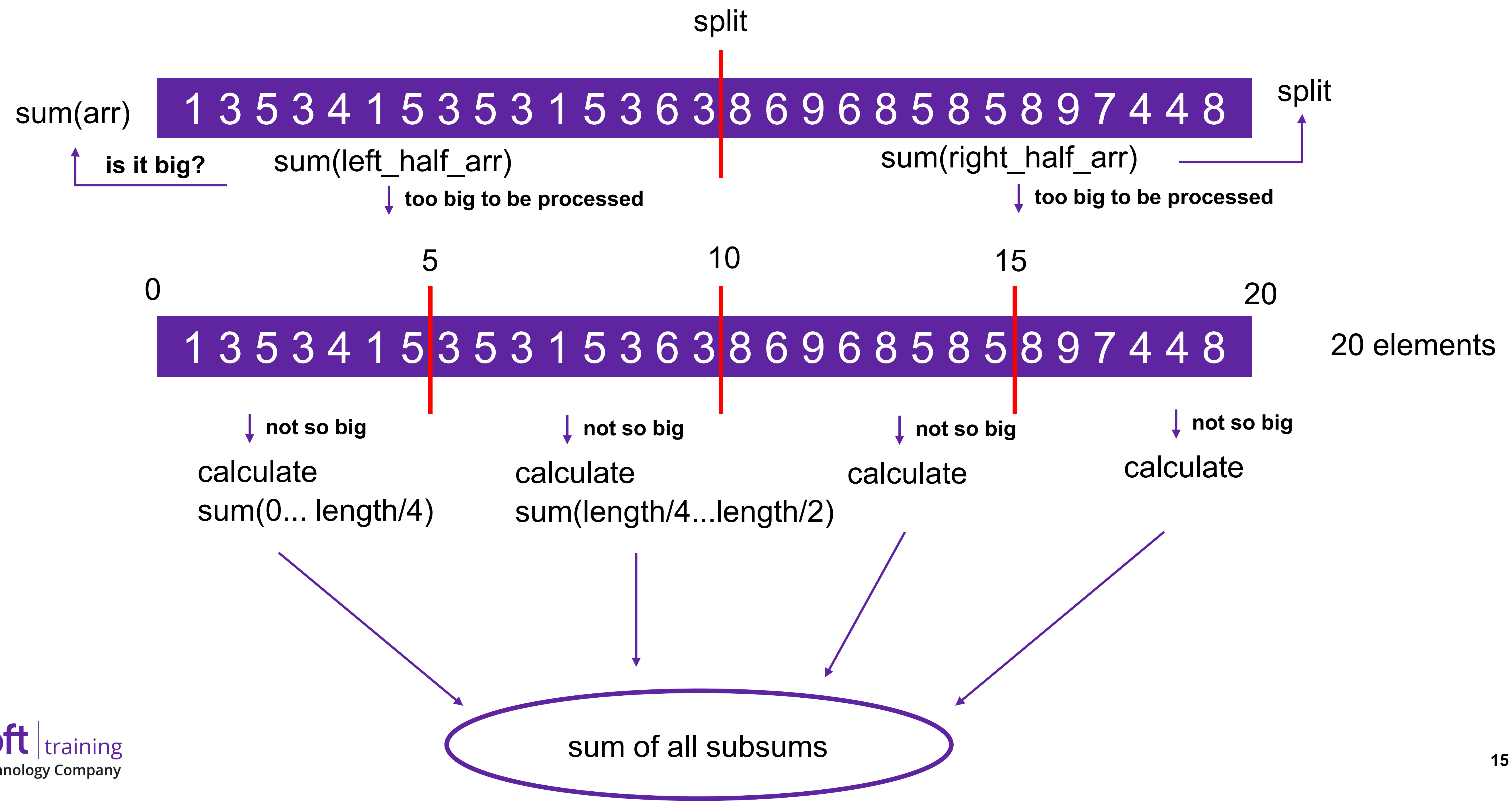
ForkJoin Framework



Fork/Join thread pool



ForkJoin Framework



ForkJoin Framework – Limitations

- Task can only use `fork()` and `join()` operations as synchronization mechanisms.
- Tasks could not perform I/O operations.
- Task can’t throw checked exceptions.

ForkJoin Framework – Elements

- ForkJoin Framework is formed by two classes.
- **`ForkJoinPool`** – is the `ExecutorService` implementation with work-stealing algorythm.
- **`ForkJoinTask`** – base class for tasks executed in `ForkJoinPool`.

Creating Pool and Task

Examples:
ForkJoinUpdatePriceTutor
ForkJoinSearchTutor

- ForkJoin is designed for solving problems by divide them into smaller parts.
- Mechanics of creating pool and tasks is quite similar to common `Executors`.

```
ForkJoinPool pool = new ForkJoinPool();
PriceUpdateTask task = new PriceUpdateTask(// ... );
// in task
protected void compute() {
   if (isSmallEnough()) {
      conquer();
   } else {
      divide();
   }
}
```

Processing array data: performance comparison

Calculating the sum of arrays with a different number of elements (time in microseconds):

Number of elements	1000	100_000	1_000_000	10_000_000
sequential adding	**9**	**85**	673	6732
stream.sum()	22	**87**	347	3395
stream.parallel().sum()	158	155	**251**	**876**
ForkJoin	29	120	676	1575

ForkJoinSum

Variance

Imperative version done in: 56 msecs
Parallel streams version done in : 41 msecs
ForkJoin version done in : 8 msecs

Thank you!

Please share your feedback.
Your opinion is important to us!

Luxoft training
A DXC Technology Company