Executor framework
Fork-Join pool

Advanced Java I. Functional, Asynchronous, Reactive Java

Module 2

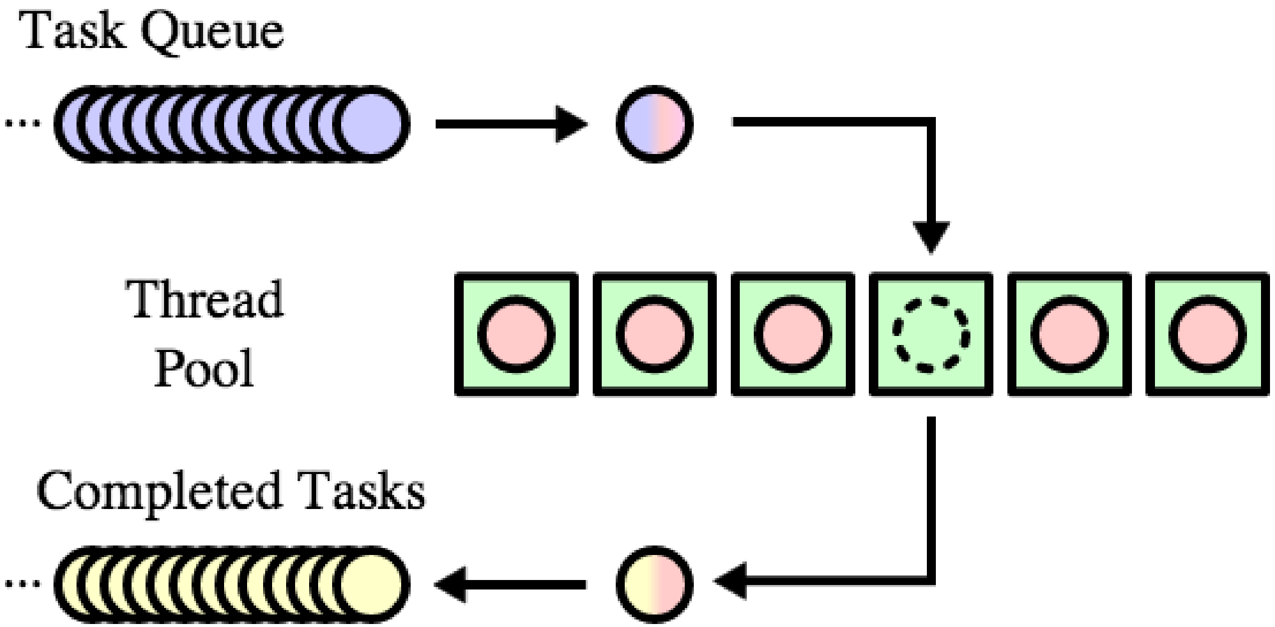
think.
create.
accelerate.

Luxoft | training
A DXC Technology Company

Executor framework

Executor Framework

- Manual thread managment in real world application is hard.
- It is good practice to isolate bussines logic 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.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 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.

Running tasks

- There are few ways to run task.
- `execute(Runnable)` – fire and forget
- `submit(Runnable)` – returns a **`Future<?>`** that represents task and `get` always return **`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 fastest task, rest of tasks will be **cancelled**.

Using of Callable interface

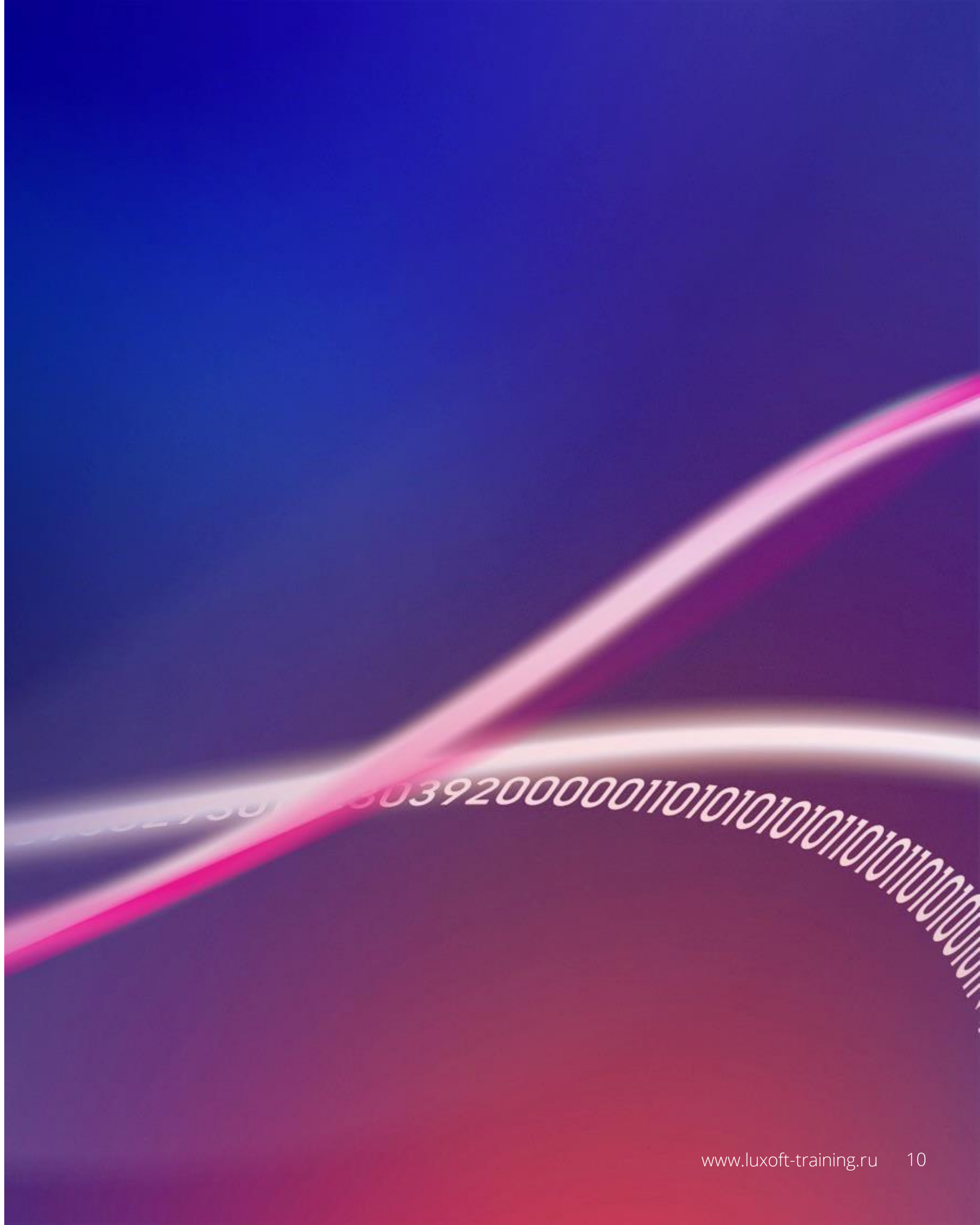
```
Future future = executorService.submit(new Callable(){
    public Object call() throws Exception {
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});

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

Stoping tasks

- 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`.

Example:
CallableTutor

ForkJoin Framework

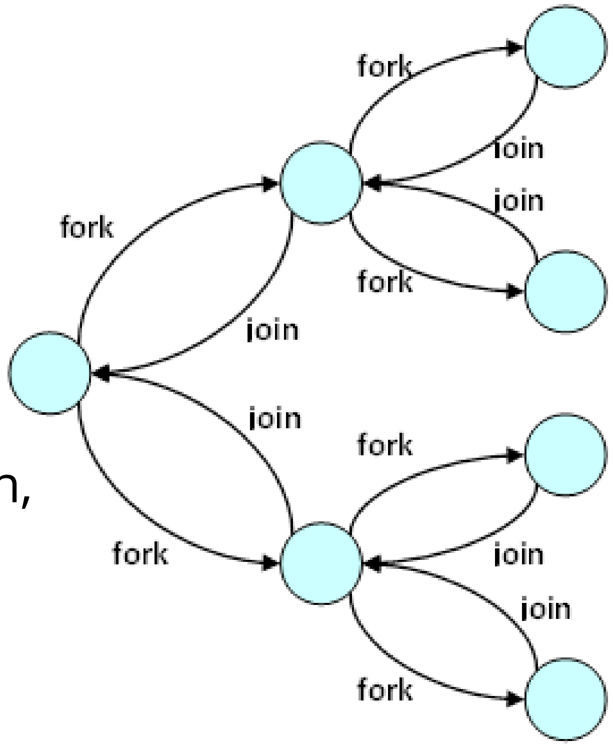


Why ForkJoin?

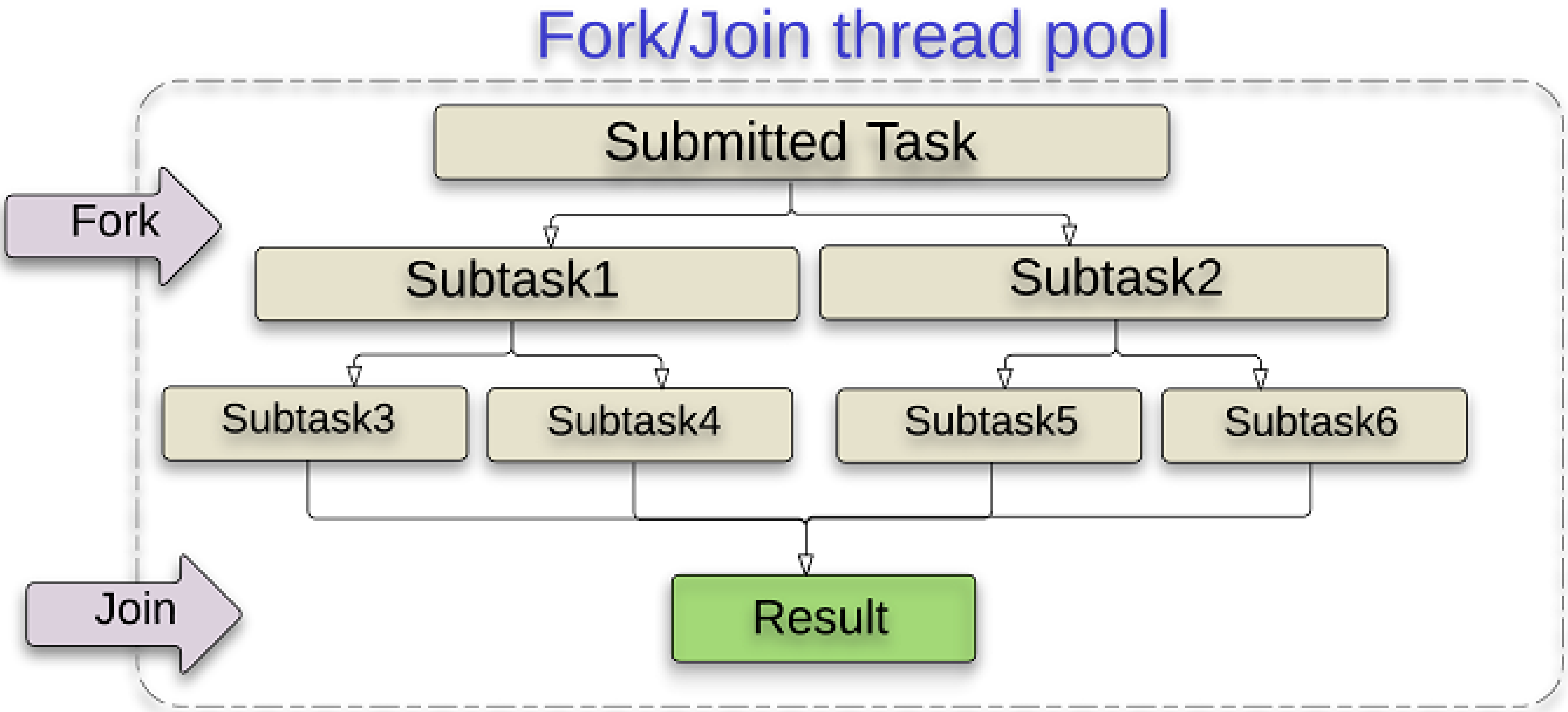
- Work with raw threads is difficult and is a source of strange, hard to locate and fix bugs.
- In Java 5, Sun introduces the Executor Framework to cover most of use cases.
- Executor Framework does not solve problem of blocking tasks.
- In Executor Framework thread wait until sub task end they job.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

ForkJoin Framework – basics

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



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 alghorytm.
- **`ForkJoinTask`** – base class for tasks executed in `ForkJoinPool`.

Creating pool and task

- ForkJoin is designed for solving problems by divide it 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();
   }
}
```

Examples:
ForkJoinUpdatePriceTutor
ForkJoinSearchTutor

Thank You!

think.
create.
accelerate.

luxoft | training
A DXC Technology Company