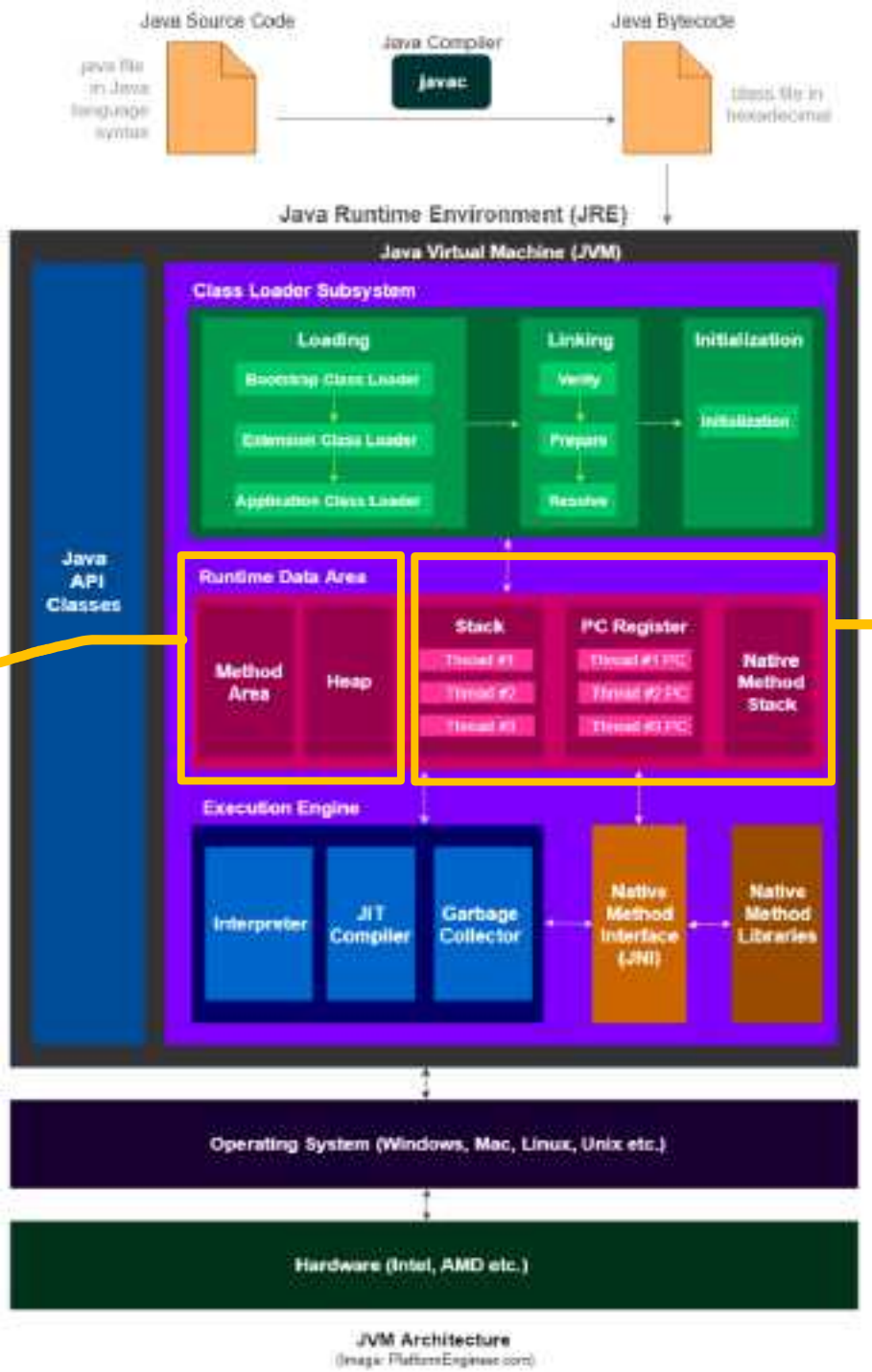
JAVA Multithreading

YAO ZHAO

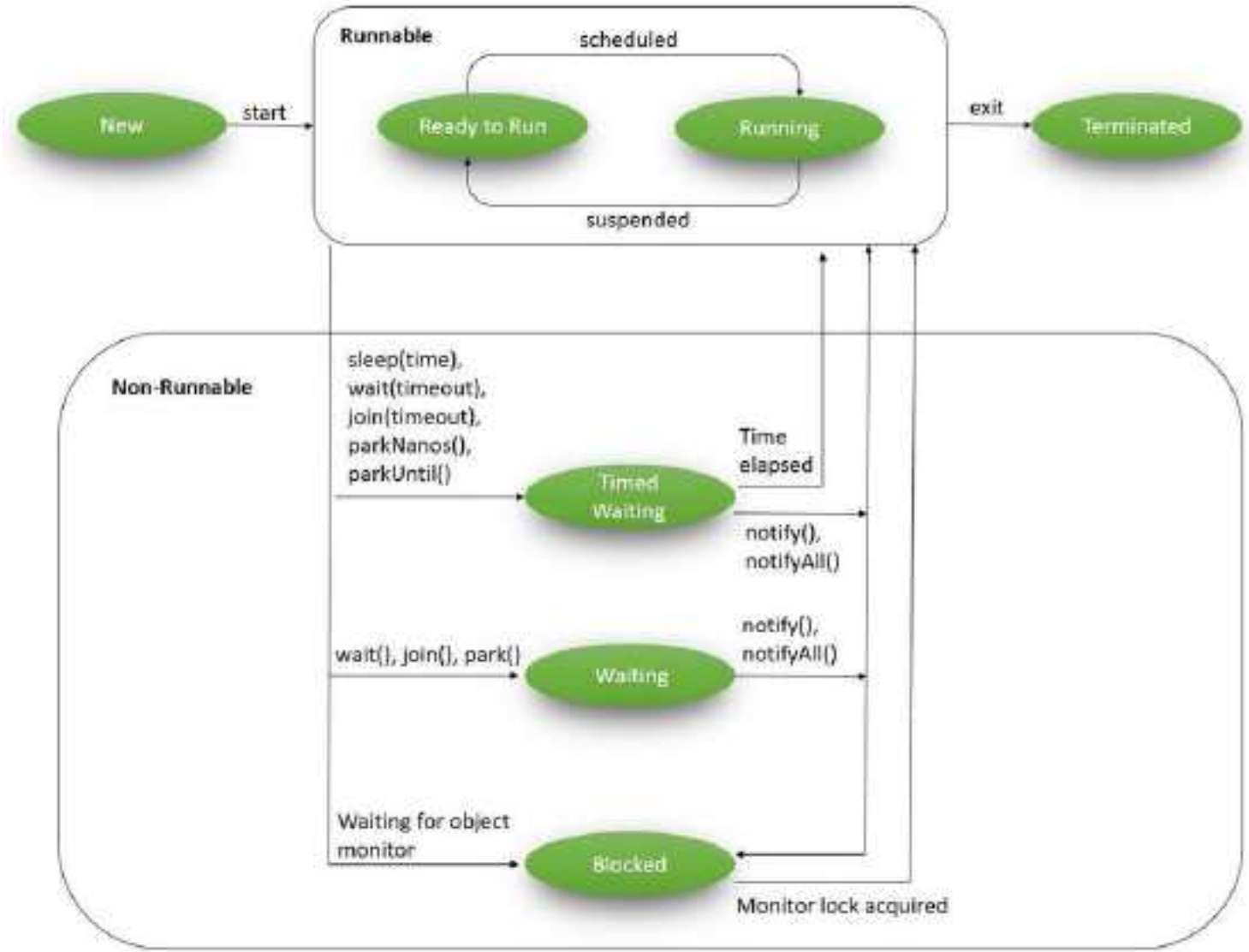
All thread share



JVM Architecture

thread private

Thread states



The examples of Thread State

- ThreadState.java

Thread Safety

- A class is thread safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

-----Brian Goëtz , Java Concurrency,p28

Synchronized vs ReentrantLock

- Benefits of ReentrantLock
  - Ability to lock interruptibly.
  - Ability to timeout while waiting for lock.
  - Power to create fair lock.
  - API to get list of waiting thread for lock.
  - Flexibility to try for lock without blocking.
- Disadvantages of ReentrantLock
  - programmer is responsible for acquiring and releasing lock, which is a power but also opens gate for new subtle bugs, when programmer forget to release the lock in finally block.

The example of Synchronized and ReentrantLock

- Package: counter
- Run Test.java

  Expect: 1000 but 10 or 20
- Make all comments contains "*synchronized*" to runnable code, then run Test.java
- Make all "*synchronized*" as comments
- Make all comments contains "*accountLock*" to runnable code, then run Test.java

Thread Pool

- Each time we create a thread object using **new Thread**, which cannot be reused, and object creation and destruction cost resources.
- The threads **created by new Thread** cannot be configured with Timer or interruption
- There is no unified management for the threads **created by new thread**. If there are too many threads, these threads may compete, or occupy too much CPU resources, and even crash
- **The Thread Pool helps to save resources** in a multithreaded application and to contain the parallelism **in certain predefined limits**.

Executor vs ExecutorService

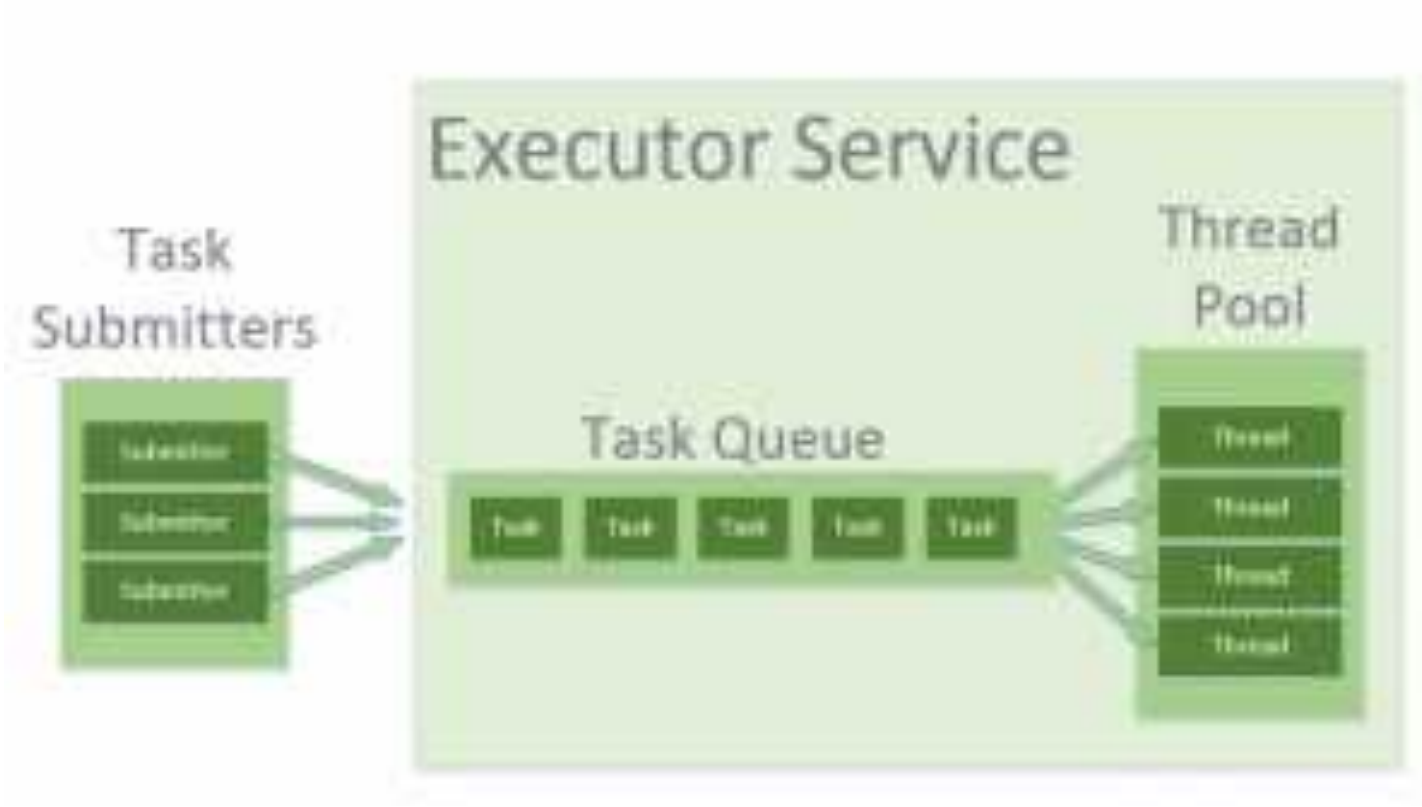
- **ExecutorService** interface extends **Executor** interface
- **ExecutorService** provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.
- **ExecutorService** can be shut down, which will cause it to reject new tasks.

```
public interface Executor {
      void execute(Runnable command);
}
```

```
public interface ExecutorService extends Executor {
       Future<T> submit(Callable<T> task)
       Future<T> submit(Runnable task, T result)
      Future<?> submit(Runnable task)
      …//hava more management methods
      …
}
```

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html

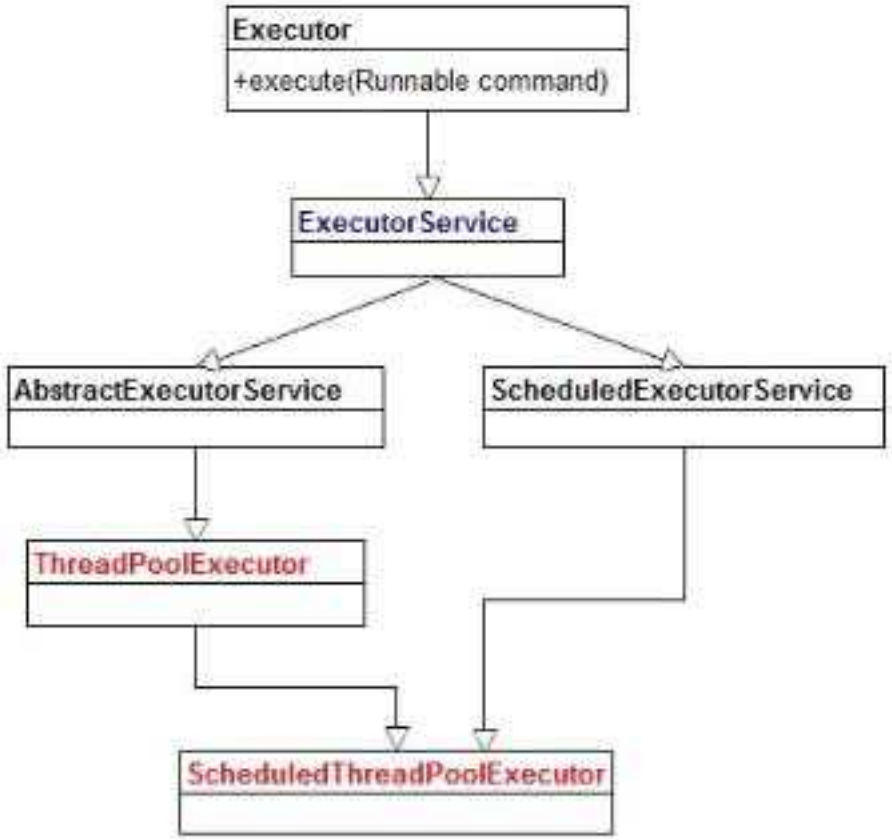
ExecutorService



ExecutorService

- The easiest way to create ***ExecutorService*** is to use one of the factory methods of the ***Executors*** class.
  - the following line of code will create a thread pool with 10 threads:
  - ExecutorService executor = Executors.newFixedThreadPool(10);
  - There are several other factory methods to create a predefined *ExecutorService* that meets specific use cases. To find the best method for your needs, consult Oracle's official documentation.

Executor Inheritance Tree



Four Thread Pool Created by ***Executors***

- **newCachedThreadPool：the cached thread pool may grow without bounds to accommodate any number of submitted tasks.** But when the threads are not needed anymore, they will be disposed of after a specified time of inactivity. A typical use case is when we have a lot of short-living tasks in our application.
- **newFixedThreadPool：** the number of threads in this thread pool is always the same. if the number of simultaneously running tasks is always less than or equal to two, they get executed right away. Otherwise, **some of these tasks may be put into a queue to wait for their turn.**
- **newSingleThreadExecutor：** containing a single thread, Tasks will be run sequentially. **The single thread executor is ideal for creating an event loop.**
- **newScheduledThreadPool：** allows us to run a task once after a specified delay or run a task after a specified initial delay and then run it repeatedly with a certain period.

ForkJoinPool

- **ForkJoinPool** extends AbstractExecutorService
- A ForkJoinPool differs from other kinds of ExecutorService mainly by virtue of employing ***work-stealing***: all threads in the pool attempt to find and execute tasks submitted to the pool and/or created by other active tasks (eventually blocking waiting for work if none exist). This enables efficient processing when most tasks spawn other subtasks (as do most ForkJoinTasks), as well as when many small tasks are submitted to the pool from external clients.
- A static commonPool() is available and appropriate for most applications. The common pool is used by any ForkJoinTask that is not explicitly submitted to a specified pool. Using the common pool normally reduces resource usage (its threads are slowly reclaimed during periods of non-use, and reinstated upon subsequent use).

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html

work stealing

- In a **work stealing scheduler**, each processor in a computer system has a **queue of work items** (computational tasks, threads) to perform. Each work item consists of a series of instructions, to be executed sequentially, but in the course of its execution, **a work item may also *spawn* new work items that can feasibly be executed in parallel with its other work**. These new items **are initially put on the queue of the processor executing the work item**. When **a processor runs out of work**, it looks at the queues of the other processors and **"steals" their work items**. In effect, work stealing distributes the scheduling work over idle processors, and as long as all processors have work to do, no scheduling overhead occurs.

The examples of Thread pool

- ThreadPoolExample.java