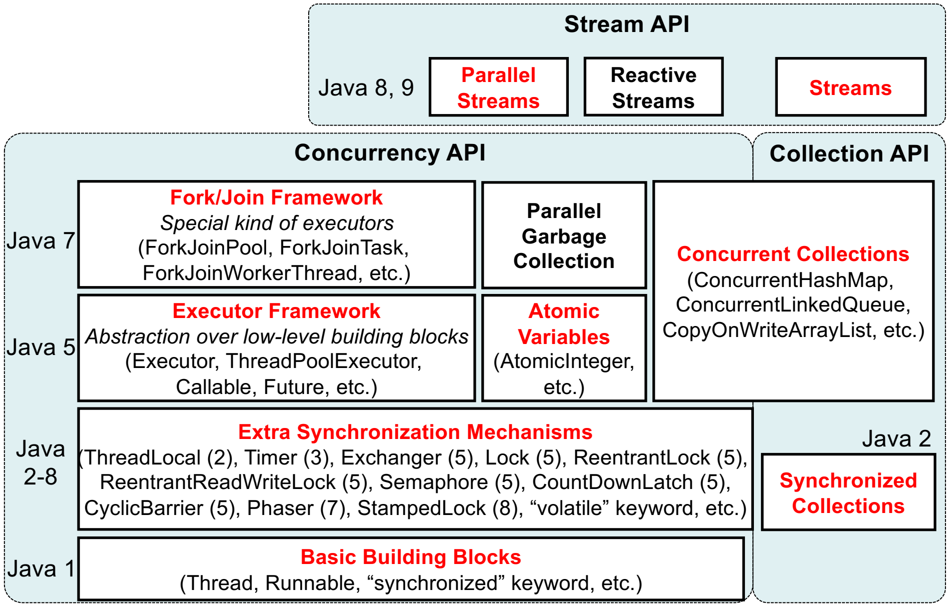
Executor Framework

Concurrency API in Java



Stream API

Java 8, 9

Parallel Streams

Reactive Streams

Streams

Concurrency API

Collection API

Java 7

Fork/Join Framework
Special kind of executors
(ForkJoinPool, ForkJoinTask, ForkJoinWorkerThread, etc.)

Parallel Garbage Collection

Concurrent Collections
(ConcurrentHashMap, ConcurrentLinkedQueue, CopyOnWriteArrayList, etc.)

Java 5

Executor Framework
Abstraction over low-level building blocks
(Executor, ThreadPoolExecutor, Callable, Future, etc.)

Atomic Variables
(AtomicInteger, etc.)

Java 2-8

Extra Synchronization Mechanisms
(ThreadLocal (2), Timer (3), Exchanger (5), Lock (5), ReentrantLock (5), ReentrantReadWriteLock (5), Semaphore (5), CountDownLatch (5), CyclicBarrier (5), Phaser (7), StampedLock (8), "volatile" keyword, etc.)

Java 2

Synchronized Collections

Java 1

Basic Building Blocks
(Thread, Runnable, "synchronized" keyword, etc.)

Executor Framework

- An abstraction layer atop low-level concurrency primitives
  - Focuses on task execution on threads
  - Decouples task execution (on threads) from task submission (to threads) to make task execution configurable.
  - Introduced in Java 5 (2004)
    - Enhanced further in subsequent versions
  - Implemented in `java.util.concurrent`.

Tasks, Threads and Executor

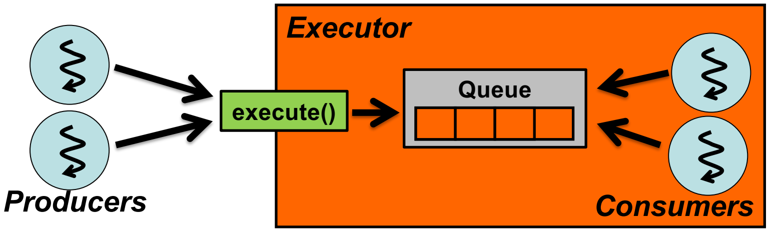
- Tasks
  - Logical units of work
    - e.g., prime number generation, access counting for files, banking (deposit/withdrawal/wire transfer of money), file caching, file crawling, file indexing, etc.
- Threads
  - Mechanism to run tasks *concurrently*.
- Executor
  - Is the primary abstraction for task execution
    - `Thread` is NOT anymore.

Executor

-
  ```
  public interface Executor{
       void execute(Runnable task); }
  ```
- `Runnable`'s `run()` implements a task.

- Producers: submit tasks
- Consumers: execute tasks

- Makes task execution configurable.



Task Execution Policies

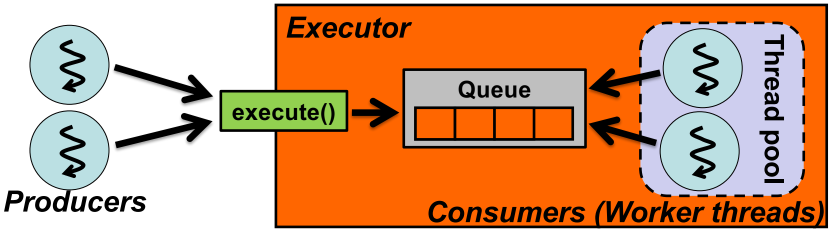
- The Executor framework allows you to specify and customize the *execution policy* for tasks.
  - "What, where, when and how" of task execution.
    - In which thread will tasks be executed?
    - In what order should tasks be executed (FIFO, LIFO, priority-based ordering)?
    - How many tasks may run concurrently.
    - How many tasks may be queued pending execution?
    - If a task has to be rejected because an application is overloaded, which task should be selected as the victim? How should the application be notified?
    - What actions should be taken before or after executing a task?

Thread Pool

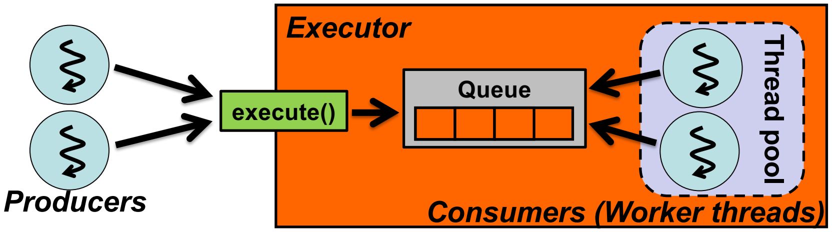
- A key component for task execution.

- A set of pre-created "worker" threads that will be used for future task execution

- Each worker thread
  - Gets and executes a task if it is available in the queue.
  - Goes to the Waiting state, if no tasks are available in the queue, until a producer submits the next task.



- Benefits of using a thread pool
  - Can eliminate runtime overhead to create threads
  - Can bound the maximum number of threads (i.e., the max amount of resource utilization)
    - Running too many threads (i.e., consuming too much resources) will result in a crash of operating system.



Executors

- A utility class for **Executor** objects
  - Defines static factory methods to create an executor with a particular thread pool.

```
– static ExecutorService newFixedThreadPool(int n)
```

- Fixed-size thread pool

```
– ExecutorService executor = Executors.newFixedThreadPool(2);
  executor.execute( new PrimeNumberGenerator(1L, 500000L) );
  executor.execute( new PrimeNumberGenerator(500001L, 1000000L) );

– Thread t1, t2;
  t1 = new Thread( new PrimeNumberGenerator(1L, 500000L) );
  t2 = new Thread( new PrimeNumberGenerator(500001L, 1000000L) );
  t1.start();
  t2.start();
```

- Static factory methods

```
– static ExecutorService newFixedThreadPool(int n)
```

- Fixed-size thread pool.

```
– static ScheduledExecutorService newScheduledThreadPool(int n)
```

- Fixed-size thread pool that supports delayed and periodic task execution.

```
– static ExecutorService newSingleThreadExecutor()
```

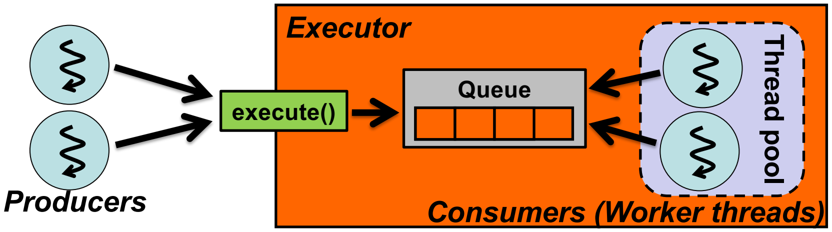
- A pool that operates only one thread.

```
– static ScheduledExecutorService newSingleThreadScheduledExecutor()
```

- A single-threaded pool that supports delayed and periodic task execution.

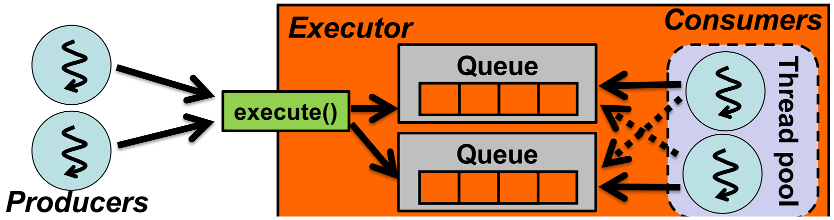
```
– static ExecutorService newCachedThreadPool()
```

- Variable-size (not fixed-size) thread pool.
  - Uses previously created "idle" threads if they are available.
  - Creates a new thread if no idle threads are available.
  - Idle threads are terminated and removed from the pool after they are not used for 60 seconds.
- Pros:
  - Can minimize the number of tasks in the queue.
  - Can minimize the number of threads (resource consumption)
- Cons: No cap for the number of threads in the pool.
- Useful to handle a number of *short-lived* (lightweight) tasks

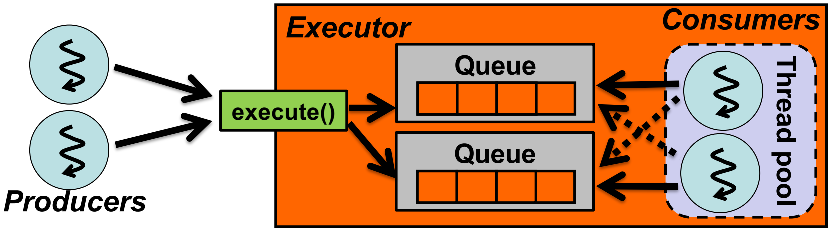


```
– static ExecutorService newWorkStealingPool(int parallelism)
```

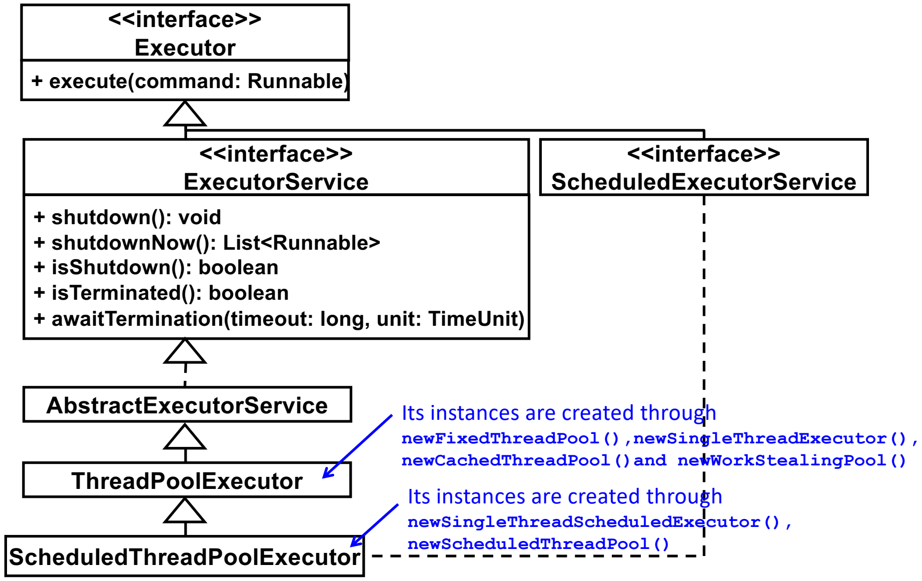
- Variable-size thread pool with a cap for the # of threads.
  - **Parallelism** specifies the cap for the # of threads.
- Each worker thread
  - Has its own "primary" queue and gets the next task from the queue.
  - "Steals" a task from another queue if no tasks are available in its primary queue.
  - Dies after being idle for some time.
- Pros:
  - Each queue requires less thread synchronization.
  - Can minimize the # of tasks in a queue and bound the # of worker threads.
- Cons: No guarantee about the order task execution



- `static ExecutorService newWorkStealingPool()`
  - Obtains the number of available CPU cores by calling `availableProcessors()` and invokes the previous version of `newWorkStealingPool()`



`ExecutorService`

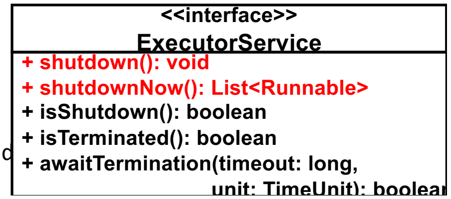


Its instances are created through `newFixedThreadPool()`, `newSingleThreadExecutor()`, `newCachedThreadPool()` and `newWorkStealingPool()`

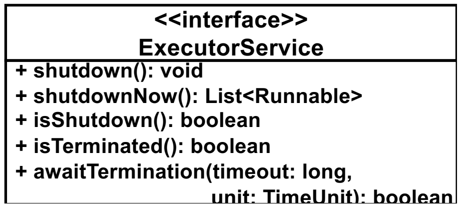
Its instances are created through `newSingleThreadScheduledExecutor()`, `newScheduledThreadPool()`

Termination of `Executor`

- Methods to terminate an executor
  - `shutdown()`
    - Rejects new tasks to get in
      - Throws a `RejectedExecutionException`
    - Allows previously submitted tasks to complete
      - Tasks being executed and tasks in the queue
  - `shutdownNow()`
    - Rejects new tasks to get in
    - Removes all tasks from the queue and returns them
    - Tries to stop the tasks that are being executed.
      - Call `interrupt()` on each worker thread
    - A task can be stopped if it checks `Thread.interrupted()` or catches `InterruptedException` to exit `run()`.
      - Otherwise, it may not be stopped



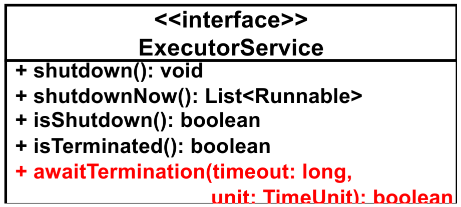
- 3 states of an executor
  - Running
  - Shutting down
    - Once `shutdown()` or `shutdownNow()` is called.
    - `isShutdown()` returns true.
  - Terminated
    - Once all tasks have been completed or stopped.
    - `isTerminated()` returns true.



- Use `awaitTermination()` if you wait for an executor to be terminated.
  - It blocks until the executor is terminated or the timeout occurs.
  - It returns true if the executor is terminated or false otherwise.

```
– executor.shutdown();
  executor.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
  doSomething();
```

```
– executor.shutdown();
  if(!executor.awaitTermination(60, TimeUnit.SECONDS)){
      shutdownNow();
      if(!executor.awaitTermination(60, TimeUnit.SECONDS)){
          doErrorHandling();
      }
  }
```



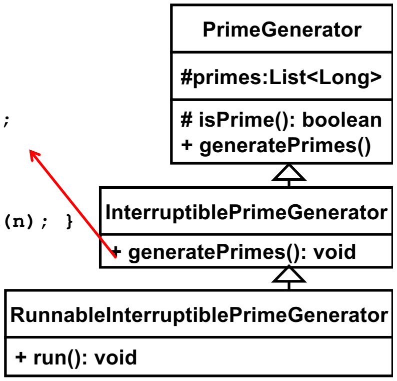
Sample Code: RunnableInterruptiblePrimeGenExecutorTest.java

```
• RunnableInterruptiblePrimeGenerator r1, r2;
  r1 = new RunnableInterruptiblePrimeGenerator(1L, 500000L);
  r2 = new RunnableInterruptiblePrimeGenerator(500001L, 1000000L);

  ExecutorService executor = Executors.newFixedThreadPool(2);

  executor.execute(r1);
  executor.execute(r2);

  executor.shutdown();

  //executor.shutdownNow();
      // Calls interrupt() on each prime gen thread. An
      // interruption is caught by Thread.interrupted() in
      // RunnableInterruptiblePrimeGenerator's run().

  executor.awaitTermination(...);

  r1.getPrimes().forEach(...);
  r2.getPrimes().forEach(...);
```

RunnableInterruptiblePrimeGenerator

- Detect an interruption from another thread to stop generating prime numbers.

```
– for (long n = from; n <= to; n++){
      if(Thread.interrupted()){
          System.out.println("Stopped");
          this.primes.clear();
          break;
      }
      if( isPrime(n) ){ this.primes.add(n); }
  }
```



- If you use `shutdown()`,
  - Two runnable tasks generate all primes.
    - gen1 generated 41538 prime numbers.
    - gen2 generated 36960 prime numbers.

- If you use `shutdownNow()`,
  - Two runnable tasks can cancel prime generation
    - Stopped generating prime numbers due to a thread interruption.
    - Stopped generating prime numbers due to a thread interruption.
    - gen1 generated 0 prime numbers.
    - gen2 generated 0 prime numbers.