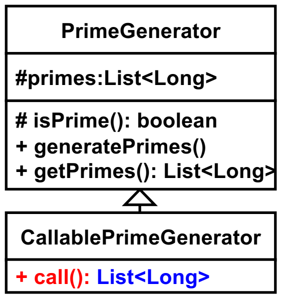
## Tasks

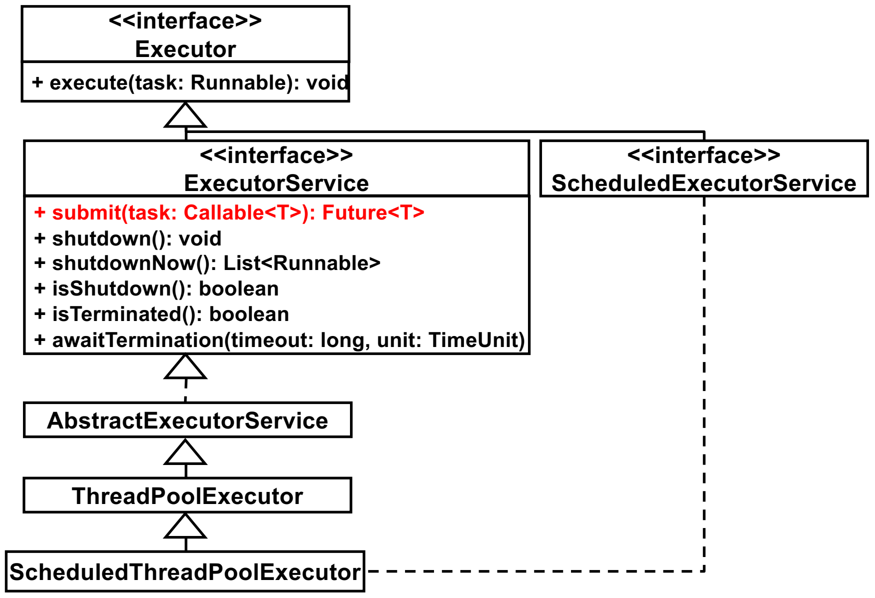
- Runnable task
  - `public interface Runnable{`
    `void run();`
    `}`
  - A **Runnable** class implements a task in **run()**.
    - No parameters. No returned value. No exceptions to be thrown.
  - Passed to an executor with its **execute()**.
- Callable task
  - `public interface Callable<T>{`
    `T call() throws Exception;`
    `}`
  - A **Callable** class implements a task in **call()**.
    - No parameters. Can return a value (**T**) and throw an **Exception**.
  - Passed to an executor with its **submit()**.

## An Example `Callable` Task

```
• class CallablePrimeGenerator
      extends PrimeGenerator
      implements Callable<List<Long>>{

      public List<Long> call() throws Exception {
          generatePrimes();
          return getPrimes();
      }
  }
```

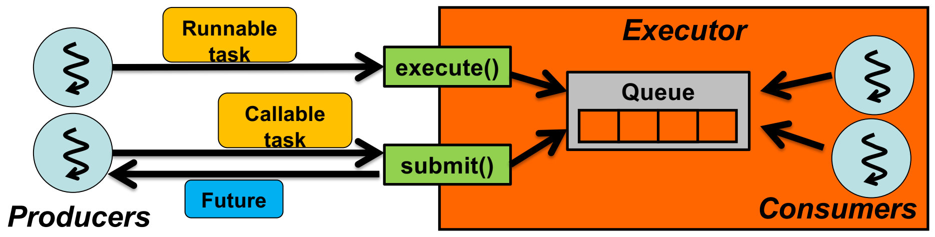


## ExecutorService


<<interface>>
Executor
+ execute(task: Runnable): void
<<interface>>
ExecutorService
+ submit(task: Callable<T>): Future<T>
+ shutdown(): void
+ shutdownNow(): List<Runnable>
+ isShutdown(): boolean
+ isTerminated(): boolean
+ awaitTermination(timeout: long, unit: TimeUnit)
<<interface>>
ScheduledExecutorService
AbstractExecutorService
ThreadPoolExecutor
ScheduledThreadPoolExecutor

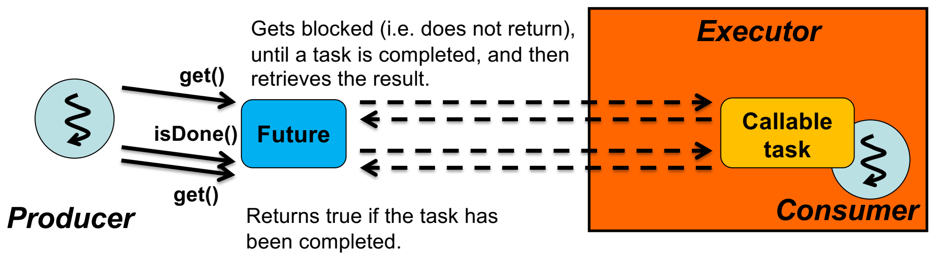
```
• CallablePrimeGenerator gen = new CallablePrimeGenerator(...);
  ExecutorService executor = Executors.newFixedThreadPool(2);
  Future<List<Long>> future = executor.submit(gen);
  List<Long> primes = future.get();
```

- **submit()** returns a **Future**, which represents the result of a task.
- An **Executor** can receive **Runnable** and **Callable** tasks simultaneously.
  - Note: A task cannot implement both **Runnable** and **Callable**.

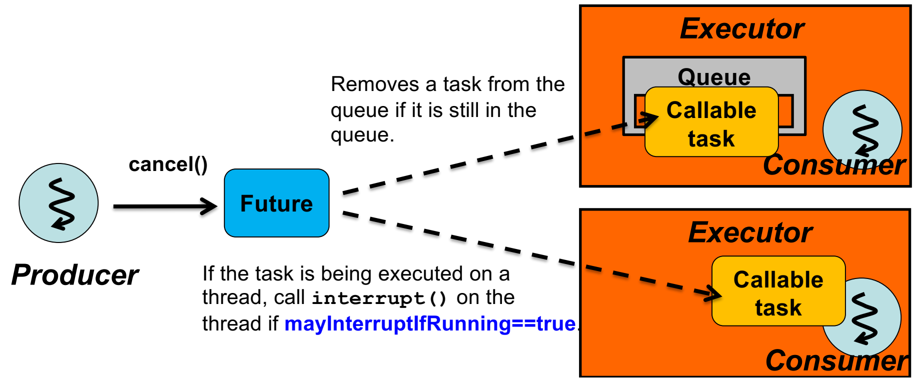


## Future

-
  ```
  public interface Future<T>{
      T get() throws ...;
      T get(long timeout, TimeUnit unit) throws ...;

      boolean isDone();
      boolean cancel(boolean mayInterruptIfRunning);
      boolean isCanceled(); }
  ```


Gets blocked (i.e. does not return), until a task is completed, and then retrieves the result.
get()
isDone()
get()
Future
Executor
Callable task
Consumer
Producer
Returns true if the task has been completed.

-
  ```
  public interface Future<T>{
      T get() throws ...;
      T get(long timeout, TimeUnit unit) throws ...;

      boolean cancel(boolean mayInterruptIfRunning);
      boolean isCanceled();
      boolean isDone();  }
  ```


Executor
Queue
Callable task
Consumer
Removes a task from the queue if it is still in the queue.
cancel()
Future
Producer
Executor
Callable task
Consumer
If the task is being executed on a thread, call interrupt() on the thread if mayInterruptIfRunning==true

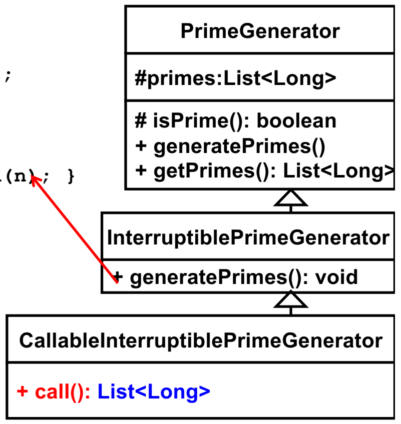
## Sample Code: CallableInterruptiblePrimeGenerator.java

-
  ```
  CallableInterruptiblePrimeGenerator gen
      = new CallableInterruptiblePrimeGenerator(1L, 500000L);

  ExecutorService executor = Executors.newFixedThreadPool(2);
  Future<List<Long>> future = executor.submit(gen);

  future.cancel(true);
  if(future.isCancelled()){
      ...
  }
  ```

## CallableInterruptiblePrimeGenerator

- 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); }
    }
    ```



## `Runnable` and `Callable` as Functional Interfaces

- `Runnable` (functional interface)

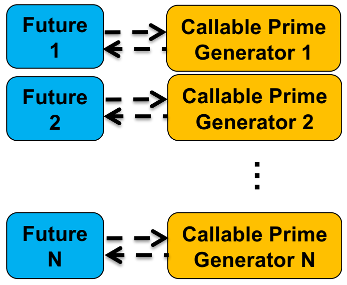
  -
    ```
    public interface Runnable{
        void run();
    }
    ```
  - Can implement the body of `run()` as a lambda expression and pass it to an executor's `execute()`.
- `Callable` (functional interface)

  -
    ```
    public interface Callable<T>{
        T call() throws Exception;
    }
    ```
  - Can implement the body of `call()` as a lambda expression and pass it to an executor's `submit()`.

## If You have a Batch of Tasks...

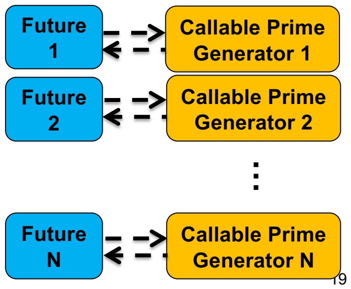
CallablePrimeGeneratorBatchTest.java

-
  ```
  List<Future<List<Long>>> futures;

  for(int i=0; i<N; i++){
    futures.add(
      executor.submit(
        new CallablePrimeGenerator(...)) );
  }

  for(Future<List<Long>> future: futures){
    allPrimes = future.get();
  }
  ```


Future 1
Callable Prime Generator 1
Future 2
Callable Prime Generator 2
⋮
Future N
Callable Prime Generator N

- A `Future's get()` gets blocked (i.e. does not return) if its associated task is not completed yet.
  - A `get()` call on `Future` #1 is blocked if Prime Generator #1 is still generating primes,
    - even if Prime Generator #2 has completed its task.
  - The order of collecting results (from generators) follow the order of generators.

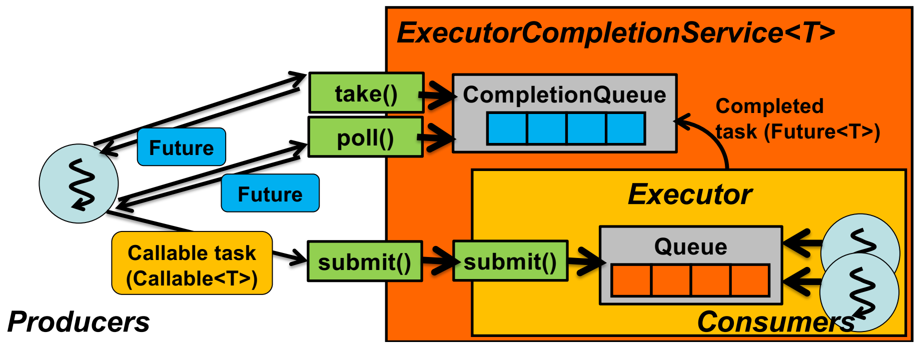
- `Executors` have no mechanisms to return completed tasks as they complete.
  - Need to repeatedly check if each task is completed, if you want to retrieve results as they become available.
    - Call `isDone()` and `get()` with a timeout of zero. A bit tedious.


Future 1
Callable Prime Generator 1
Future 2
Callable Prime Generator 2
⋮
Future N
Callable Prime Generator N

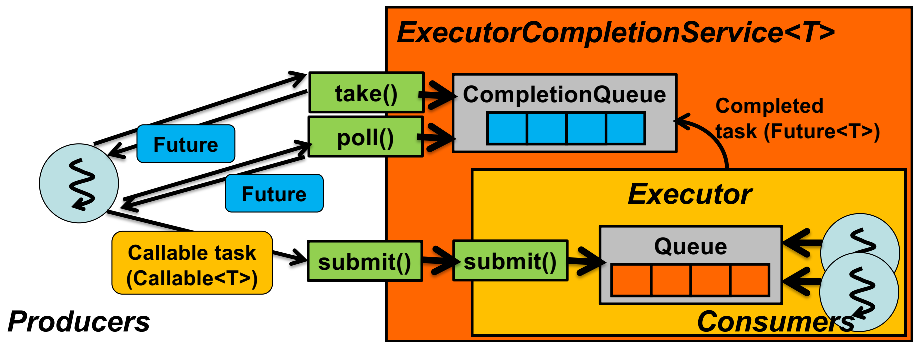
## An Extra Type of Executors: `ExecutorCompletionService`

## ExecutorCompletionService<T>

- A wrapper of an **`Executor`**
  - Introduces a *completion queue* atop an **`Executor`**
    - A queue that contains completed tasks.
- Can return completed tasks as they complete.
- T: Type of a result generated by a task.



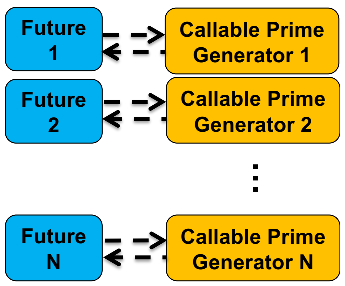
- **`take()`**
  - Retrieves and removes the **`Future`** object that represents the next completed task, waiting (i.e., getting blocked) if none are present.
- **`poll()`**
  - Retrieves and removes the **`Future`** object that represents the next completed task, or returns null if none are present.



## If You have a Batch of Tasks…

CallablePrimeGeneratorBatchTestCompletionService.java

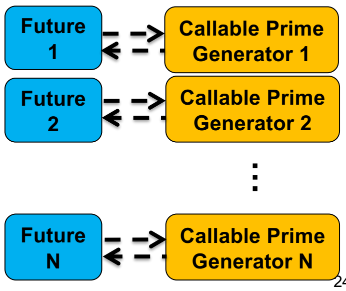
```
• ExecutorService executor =
    Executors.newFixedThreadPool(2);

  ExecutorCompletionService<List<Long>>
  completionService =
    new ExecutorCompletionService<>(executor);

  ArrayList<Future<List<Long>>> futures =
    new ArrayList<>;

  for(int i=0; i<N; i++){
      futures.add(
          completionService.submit(
            new CallablePrimeGenerator(...)) );
```



Future 1
Callable Prime Generator 1
Future 2
Callable Prime Generator 2
⋮
Future N
Callable Prime Generator N

```
• for(int completedTaskNum=0, taskNum=futures.size();
      compl<taskNum; completedTaskNum++){

          Future<List<Long>> future = completionService.take();
          List<Long> primes = future.get();
          ... // do something with primes.
  }
```

- **`ExecutorCompletionService`** returns completed tasks as they complete.
  - **`take()`** returns one of the results that have been generated by Prime Generators.
  - The order of collecting results (from generators) DOES NOT follow the order of generators.



Future 1
Callable Prime Generator 1
Future 2
Callable Prime Generator 2
⋮
Future N
Callable Prime Generator N