The Java ExecutorService Interface (Part 4)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

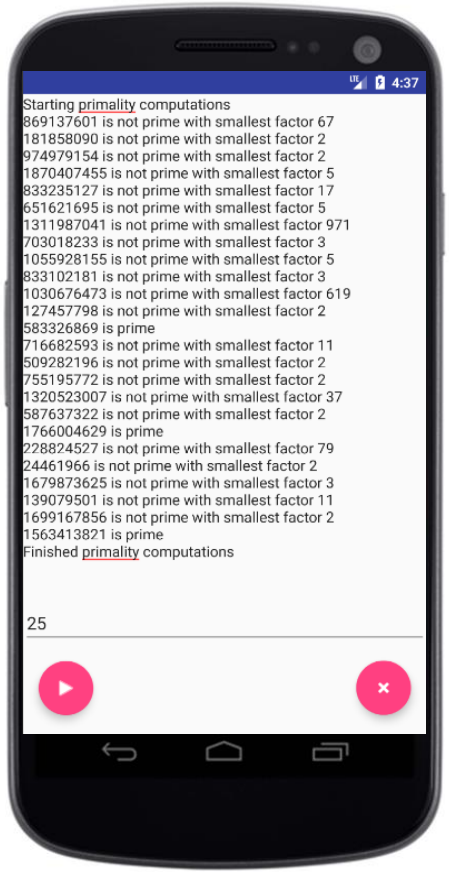
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

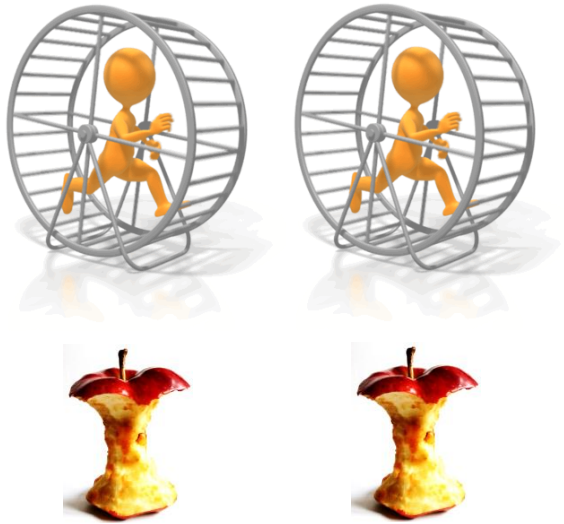
- Recognize the powerful features defined in the Java ExecutorService interface & related interfaces/classes
- Know the key methods provided by the Java ExecutorService
- Understand how ThreadPoolExecutor implements the ExecutorService
- Learn how to program a "PrimeChecker" app using the Java ExecutorService interface

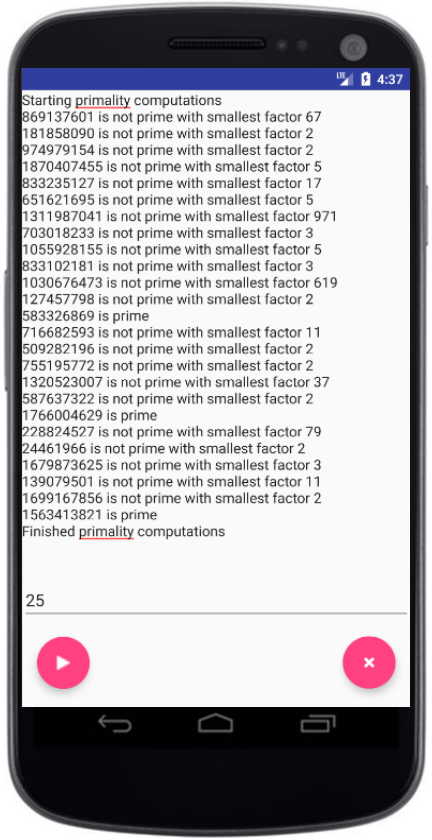


Overview of the PrimeChecker App

Overview of the PrimeChecker App

- This "embarrassingly parallel" & compute-bound app uses the Java ExecutorService to check if *N* random #'s are prime

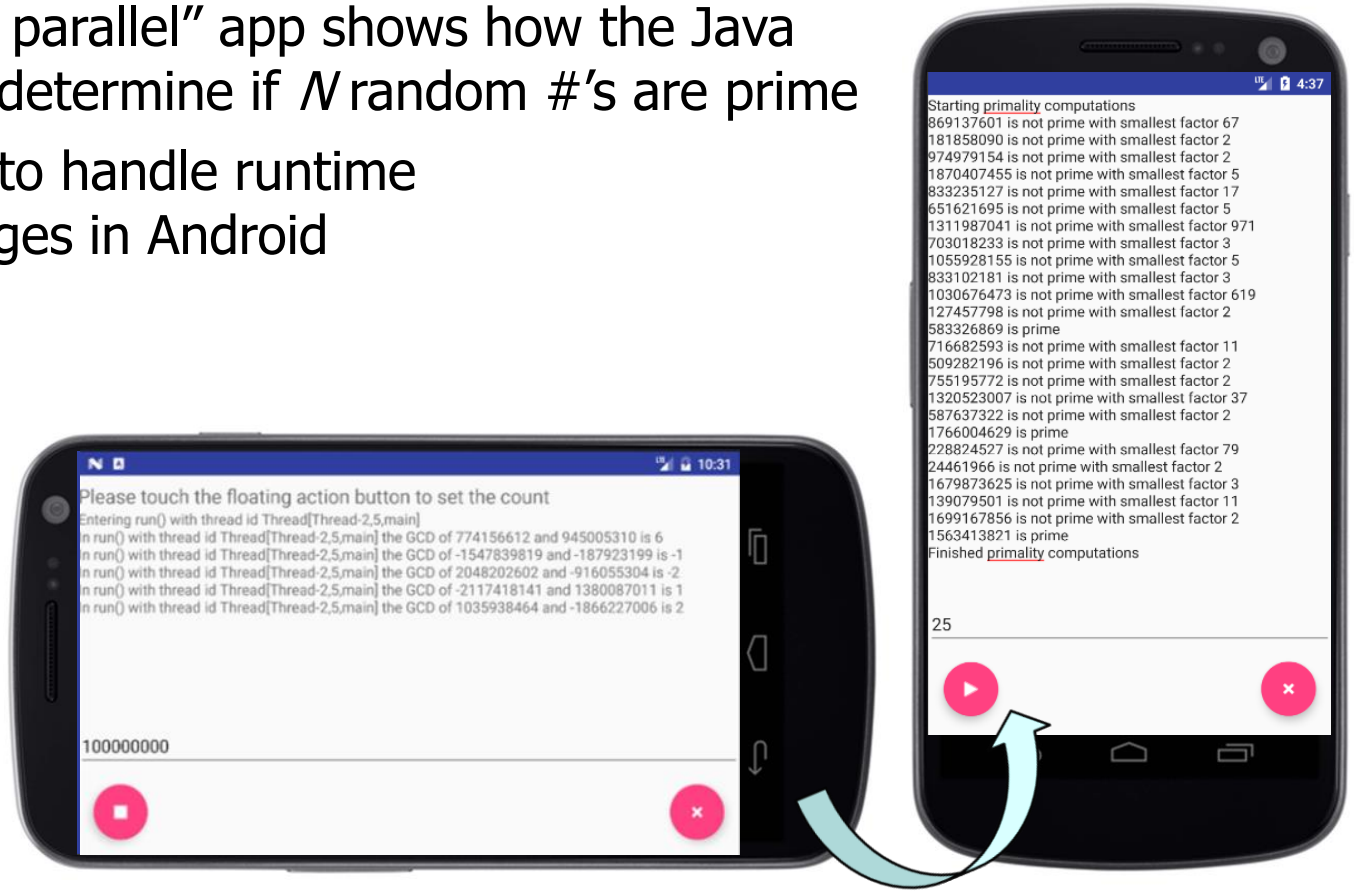




See github.com/douglascraigschmidt/POSA/tree/master/ex/M4/Primes/PrimeExecutorService

Overview of the PrimeChecker App

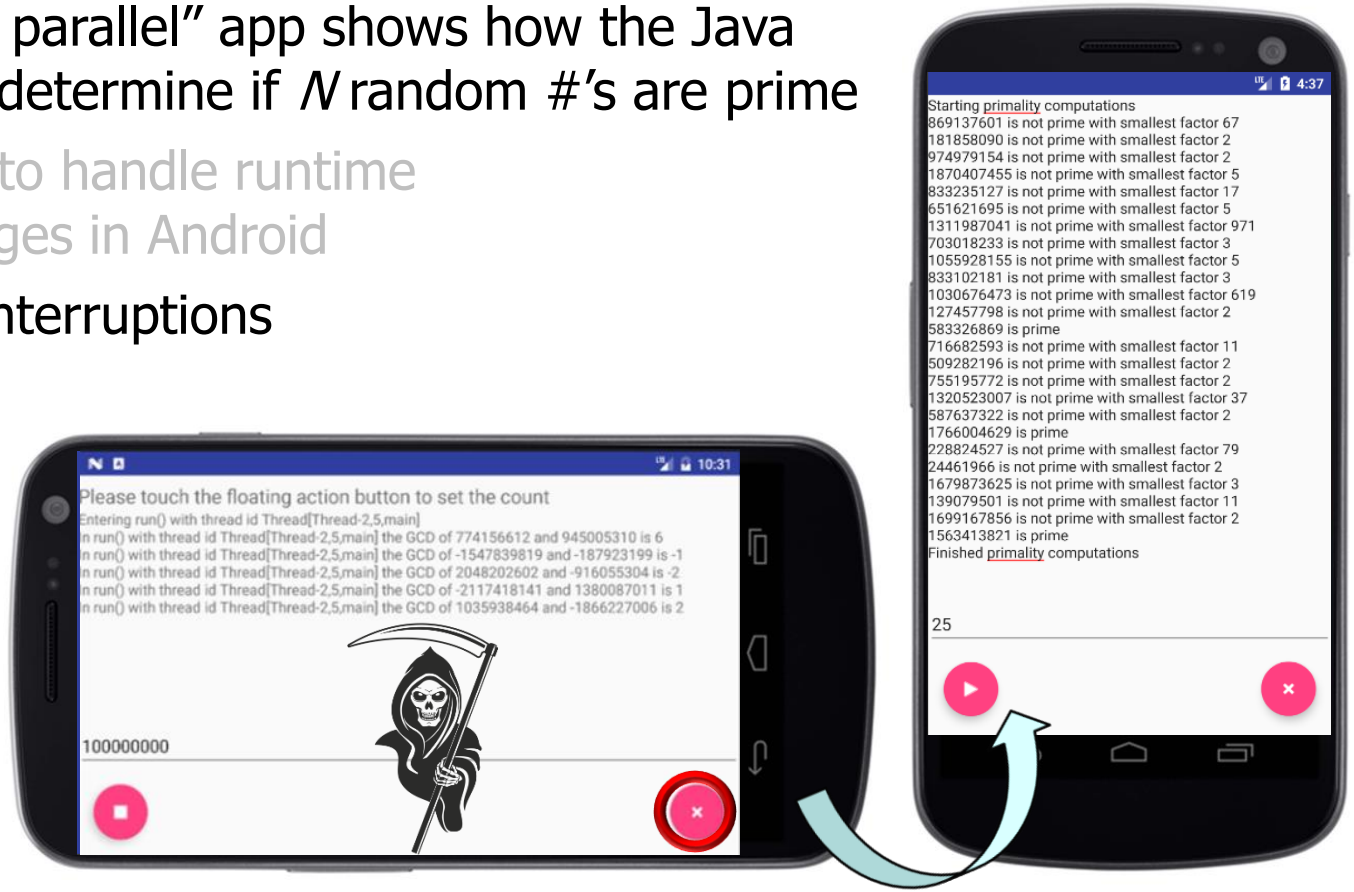
- This "embarrassingly parallel" app shows how the Java ExecutorService can determine if *N* random #'s are prime
  - It also shows how to handle runtime configuration changes in Android



See developer.android.com/guide/topics/resources/runtime-changes.html

Overview of the PrimeChecker App

- This "embarrassingly parallel" app shows how the Java ExecutorService can determine if *N* random #'s are prime
  - It also shows how to handle runtime configuration changes in Android
  - As well as thread interruptions

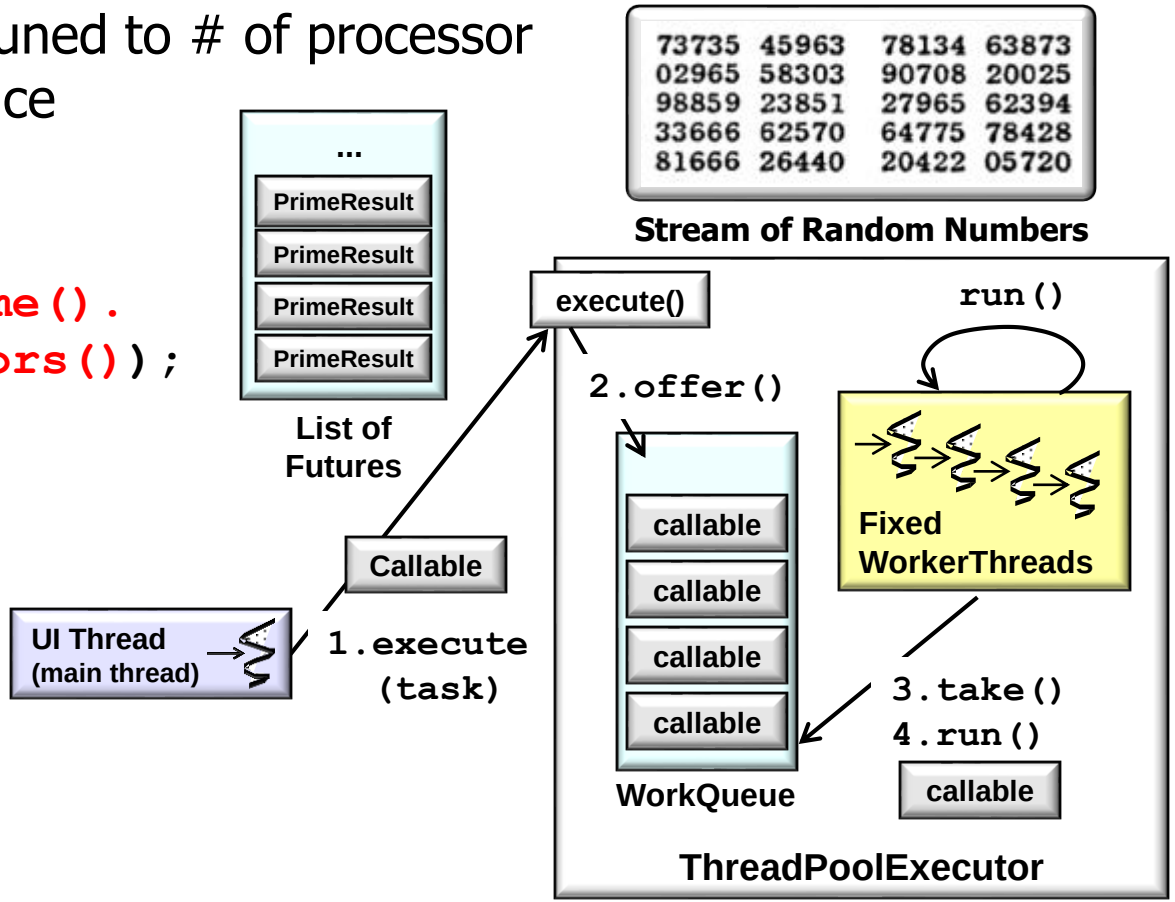


See docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html

Overview of the PrimeChecker App

- A fixed-size thread pool is tuned to # of processor cores in the computing device

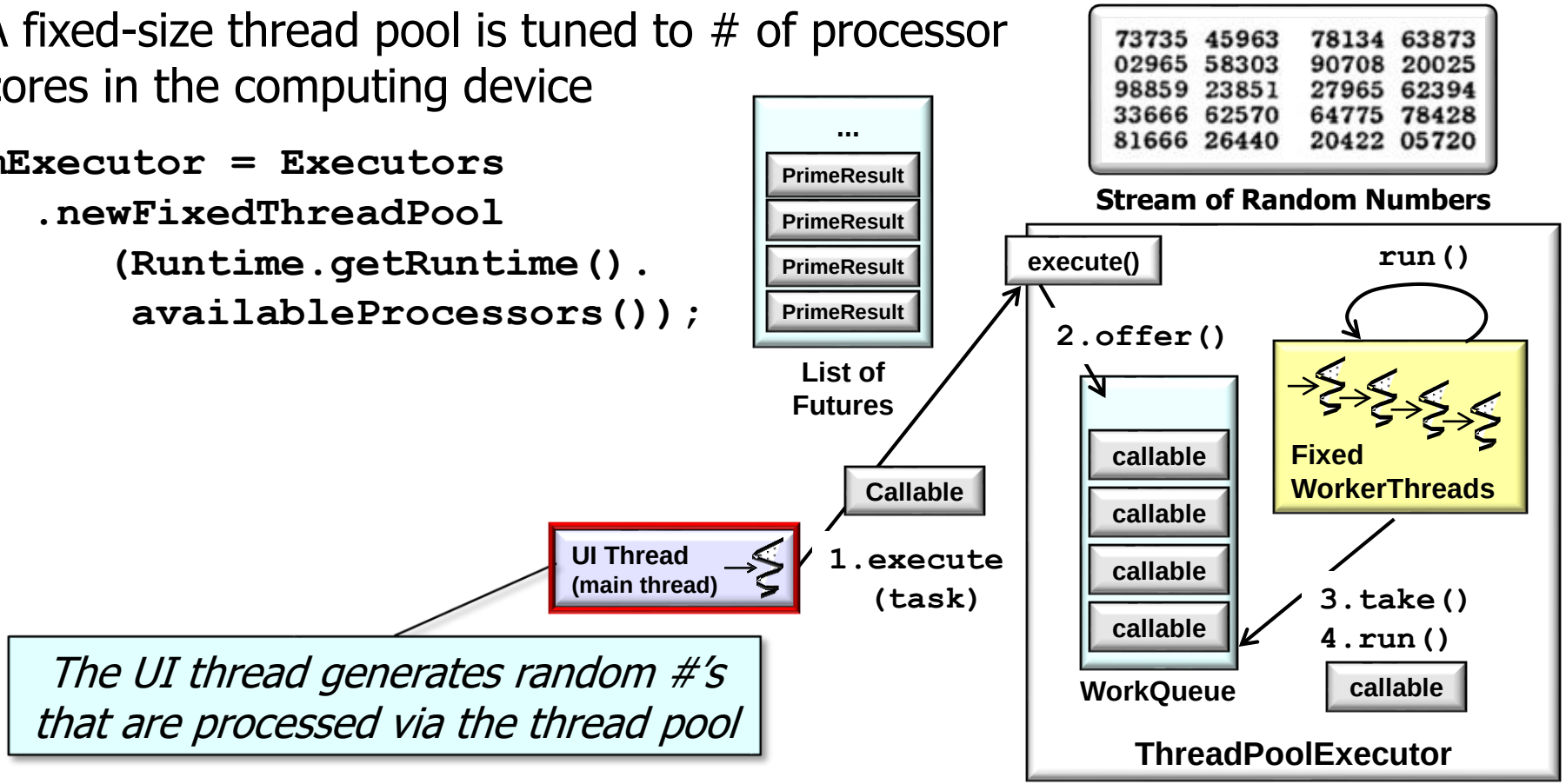
```
mExecutor = Executors
   .newFixedThreadPool
      (Runtime.getRuntime().
       availableProcessors());
```



Overview of the PrimeChecker App

- A fixed-size thread pool is tuned to # of processor cores in the computing device

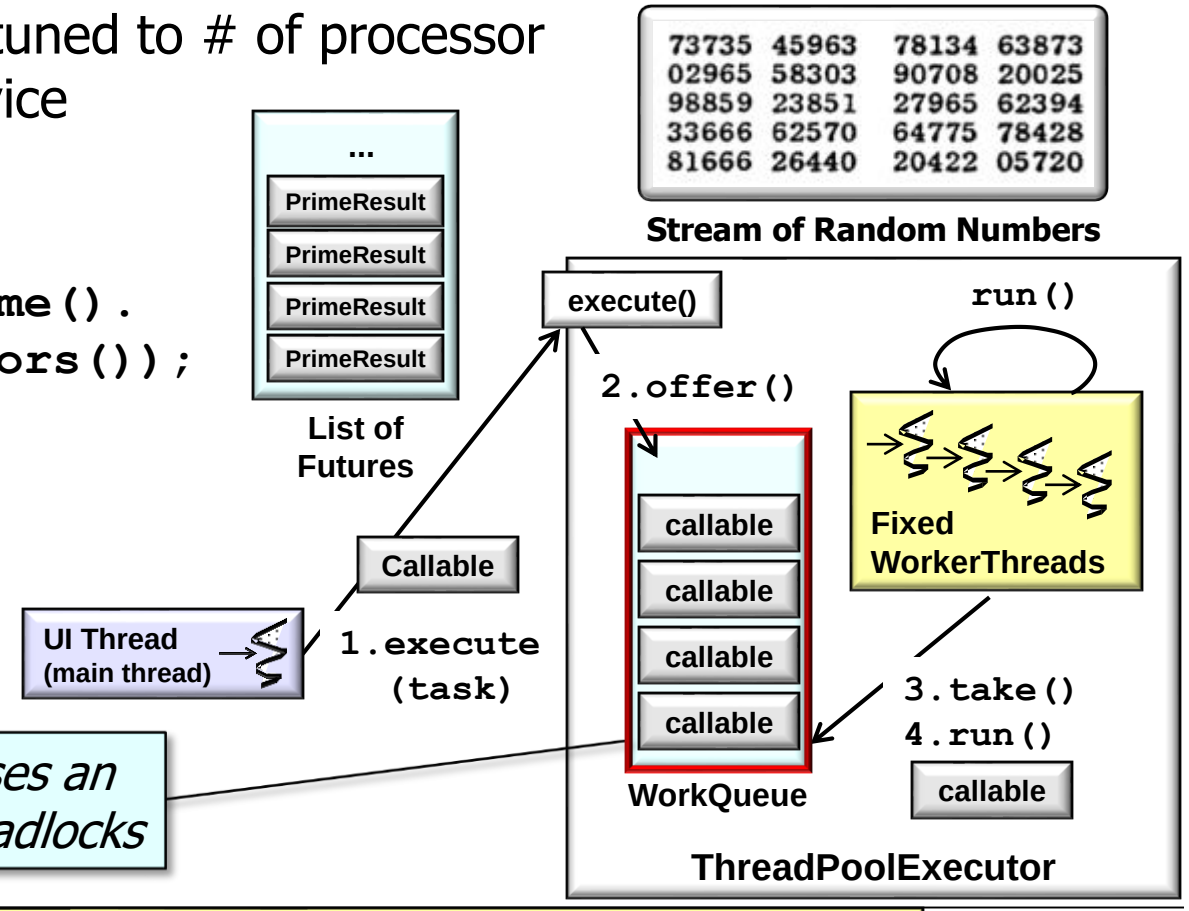
```
mExecutor = Executors
   .newFixedThreadPool
      (Runtime.getRuntime().
       availableProcessors());
```



Overview of the PrimeChecker App

- A fixed-size thread pool is tuned to # of processor cores in the computing device

```
mExecutor = Executors
   .newFixedThreadPool
      (Runtime.getRuntime().
       availableProcessors());
```



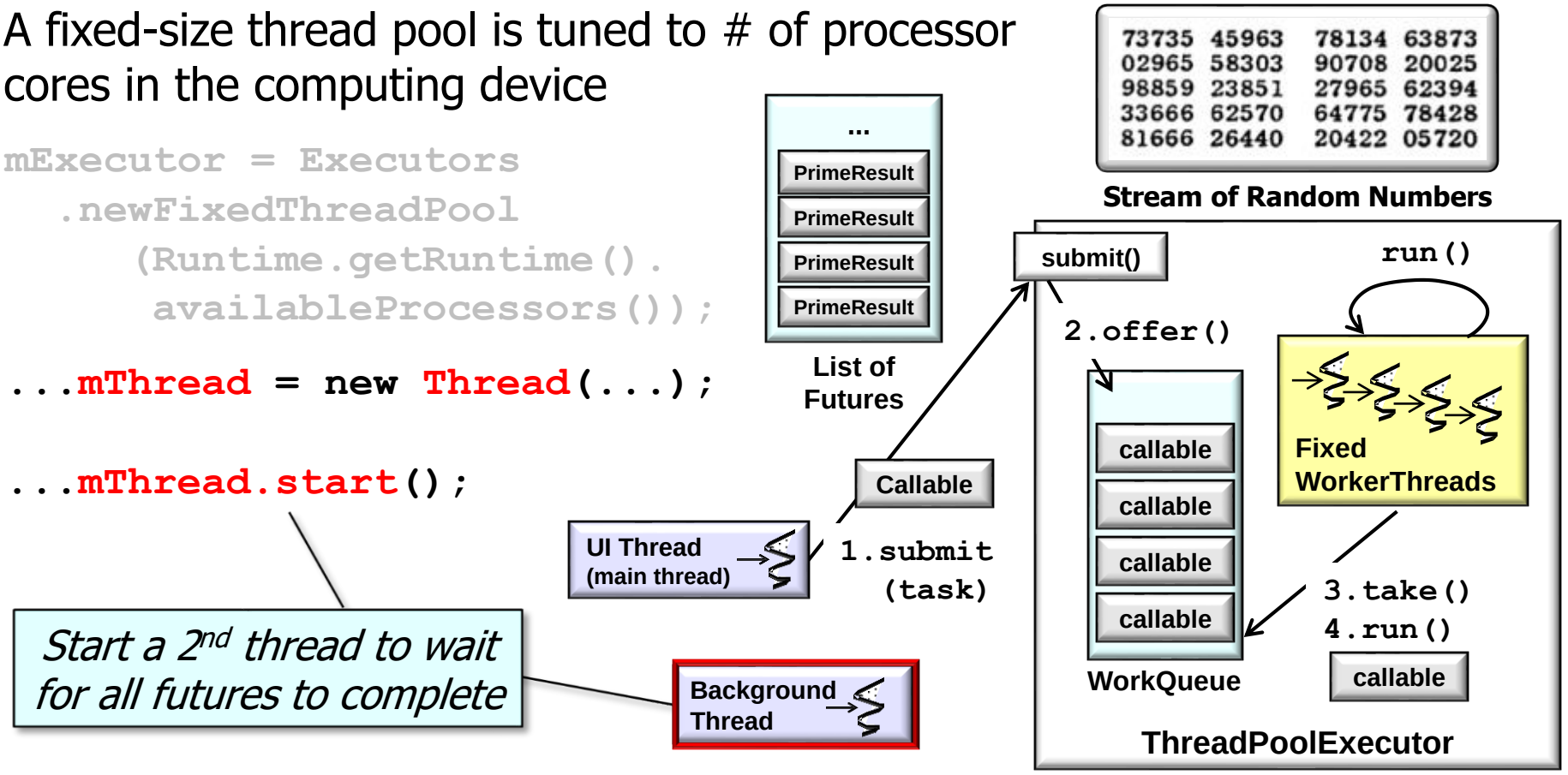
This fixed-size thread pool uses an unbounded queue to avoid deadlocks

See asznajder.github.io/thread-pool-induced-deadlocks

Overview of the PrimeChecker App

- A fixed-size thread pool is tuned to # of processor cores in the computing device

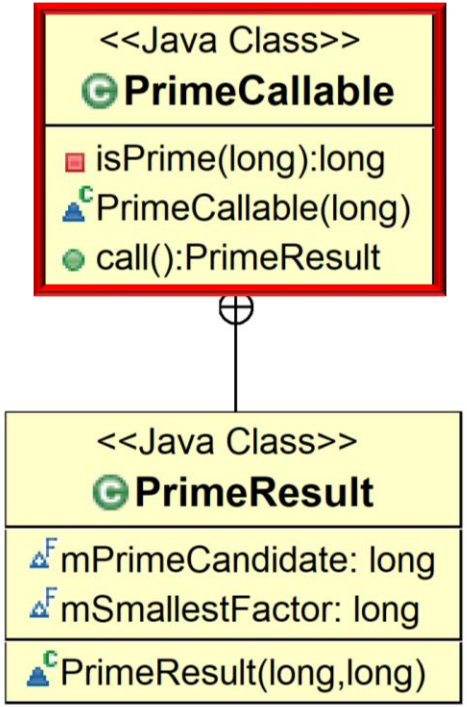
```
mExecutor = Executors
   .newFixedThreadPool
      (Runtime.getRuntime().
       availableProcessors());

...mThread = new Thread(...);

...mThread.start();
```



Overview of the PrimeChecker App

- PrimeCallable defines a two-way means of determining whether a # is prime

```
class PrimeCallable
      implements Callable<PrimeResult> {
  long mPrimeCandidate;
  ...

  PrimeCallable(Long primeCandidate)
  { mPrimeCandidate = primeCandidate; }

  long isPrime(long n) { ... }

  PrimeResult call() {
    return new PrimeResult(mPrimeCandidate,
       isPrime(mPrimeCandidate));
  } ...
```



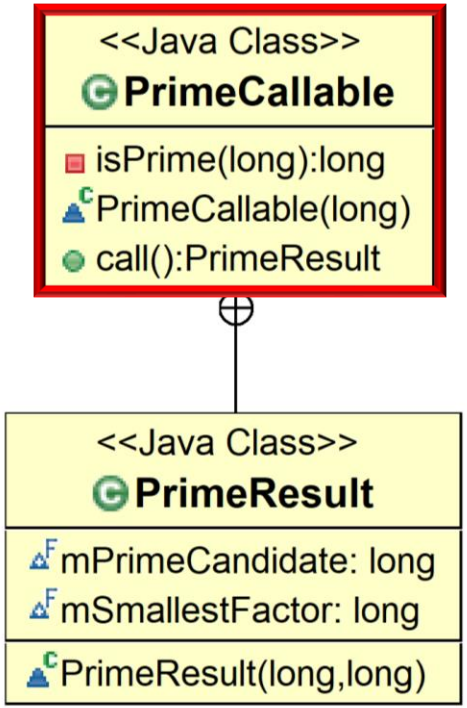
See PrimeExecutorService/app/src/main/java/vandy/mooc/prime/activities/PrimeCallable.java

Overview of the PrimeChecker App

- PrimeCallable defines a two-way means of determining whether a # is prime

```
class PrimeCallable
      implements Callable<PrimeResult> {
  long mPrimeCandidate;
  ...

  PrimeCallable(Long primeCandidate)
  { mPrimeCandidate = primeCandidate; }

  long isPrime(long n) { ... }

  PrimeResult call() {
    return new PrimeResult(mPrimeCandidate,
       isPrime(mPrimeCandidate));
  } ...
```

Implements Callable



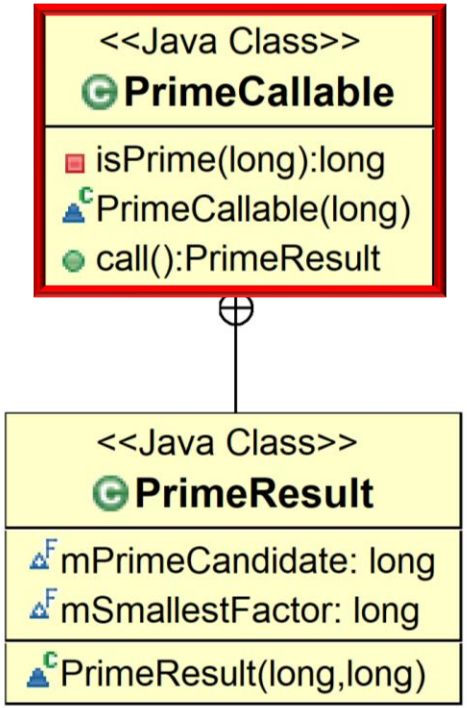
See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html

Overview of the PrimeChecker App

- PrimeCallable defines a two-way means of determining whether a # is prime

```
class PrimeCallable
      implements Callable<PrimeResult> {
  long mPrimeCandidate;
  ...

  PrimeCallable(Long primeCandidate)
  { mPrimeCandidate = primeCandidate; }

  long isPrime(long n) { ... }

  PrimeResult call() {
    return new PrimeResult(mPrimeCandidate,
       isPrime(mPrimeCandidate));
  } ...
```

The constructor stores the prime # candidate



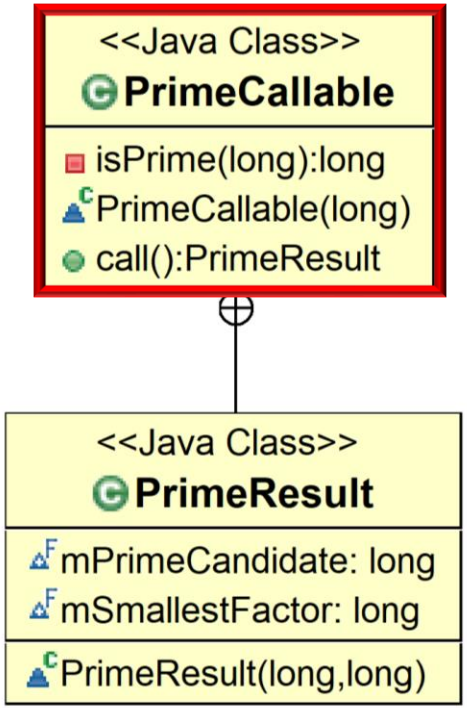
See *"The Java Executor Interface (Part 2)"*

Overview of the PrimeChecker App

- PrimeCallable defines a two-way means of determining whether a # is prime

```
class PrimeCallable
      implements Callable<PrimeResult> {
  long mPrimeCandidate;
  ...

  PrimeCallable(Long primeCandidate)
  { mPrimeCandidate = primeCandidate; }

  long isPrime(long n)

  PrimeResult call() {
    return new PrimeResult(mPrimeCandidate,
       isPrime(mPrimeCandidate));
  } ...
```

Returns 0 if n is prime or smallest factor if it's not



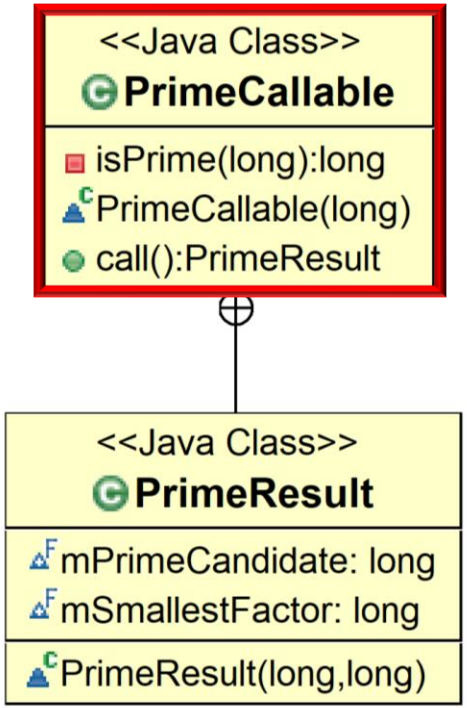
An interruptible version of isPrime() from "*The Java Executor Interface (Part 2)*"

Overview of the PrimeChecker App

- PrimeCallable defines a two-way means of determining whether a # is prime

```
class PrimeCallable
      implements Callable<PrimeResult> {
  long mPrimeCandidate;
  ...

  PrimeCallable(Long primeCandidate)
  { mPrimeCandidate = primeCandidate; }

  long isPrime(long n)

  PrimeResult call() {
    return new PrimeResult(mPrimeCandidate,
       isPrime(mPrimeCandidate));
  } ...
```

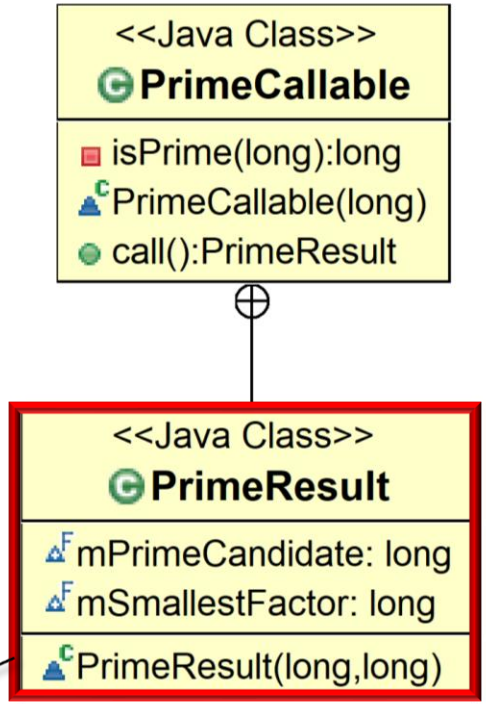
The call() hook method invokes isPrime()



Overview of the PrimeChecker App

- PrimeCallable defines a two-way means of determining whether a # is prime

```
class PrimeCallable
      implements Callable<PrimeResult> {
  long mPrimeCandidate;
  ...

  PrimeCallable(Long primeCandidate)
  { mPrimeCandidate = primeCandidate; }

  long isPrime(long n) { ... }

  PrimeResult call() {
    return new PrimeResult(mPrimeCandidate,
      isPrime(mPrimeCandidate));
  } ...
```

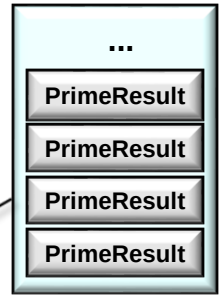


PrimeResult is a tuple that matches the prime # candidate with the result of checking primality

Overview of the PrimeChecker App

- MainActivity creates a list of futures that store results of concurrently checking primality of "count" random #'s within a range

```
List<Future<PrimeResult>>
   futures = ...
```



List of Futures

This list of futures is initialized via a Java 8 sequential stream

<<Java Class>>
MainActivity

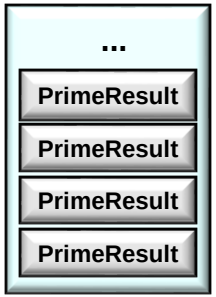
- MainActivity()
- onCreate(Bundle):void
- initializeViews():void
- setCount(View):void
- startOrStopComputations(View):void
- startComputations(int):void
- interruptComputations():void
- done():void
- println(String):void
- onRetainNonConfigurationInstance():Object
- onDestroy():void

See PrimeExecutorService/app/src/main/java/vandy/mooc/prime/activities/MainActivity.java

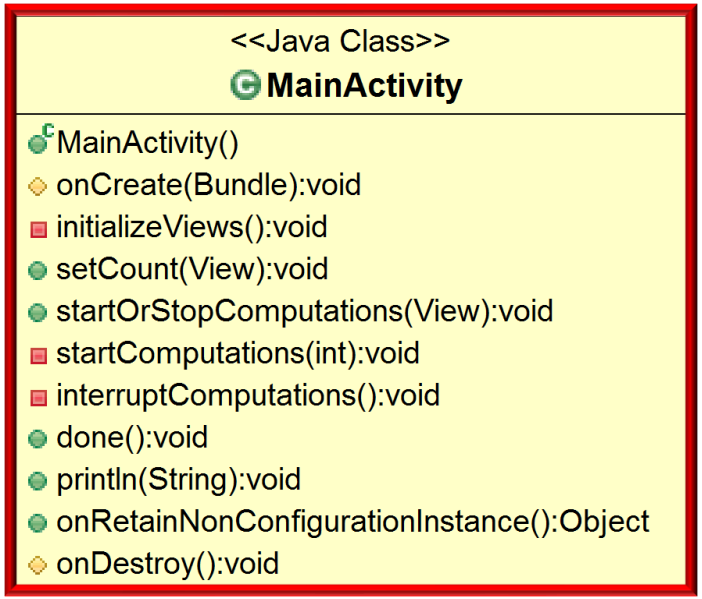
Overview of the PrimeChecker App

- MainActivity creates a list of futures that store results of concurrently checking primality of "count" random #'s within a range

```
List<Future<PrimeResult>>
   futures = new Random()
      .longs(count,
             sMAX_VALUE - count,
             sMAX_VALUE)
```



List of Futures

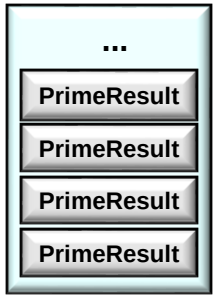


Generates "count" random #'s ranging from sMAX_VALUE – count & sMAX_VALUE

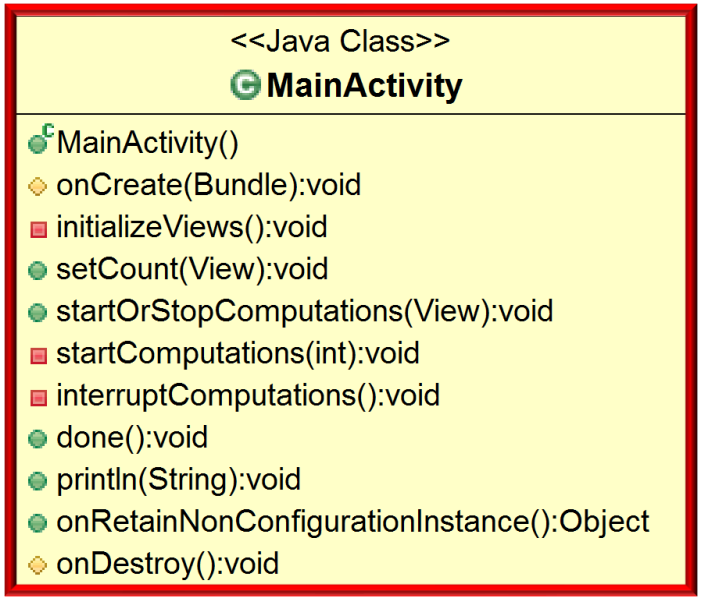
Overview of the PrimeChecker App

- MainActivity creates a list of futures that store results of concurrently checking primality of "count" random #'s within a range

```
List<Future<PrimeResult>>
   futures = new Random()
      .longs(count,
             sMAX_VALUE - count,
             sMAX_VALUE)

      .mapToObj(PrimeCallable::new)
```



List of Futures



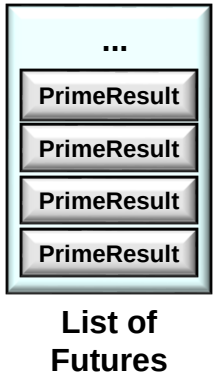
This constructor reference converts random #'s into PrimeCallables

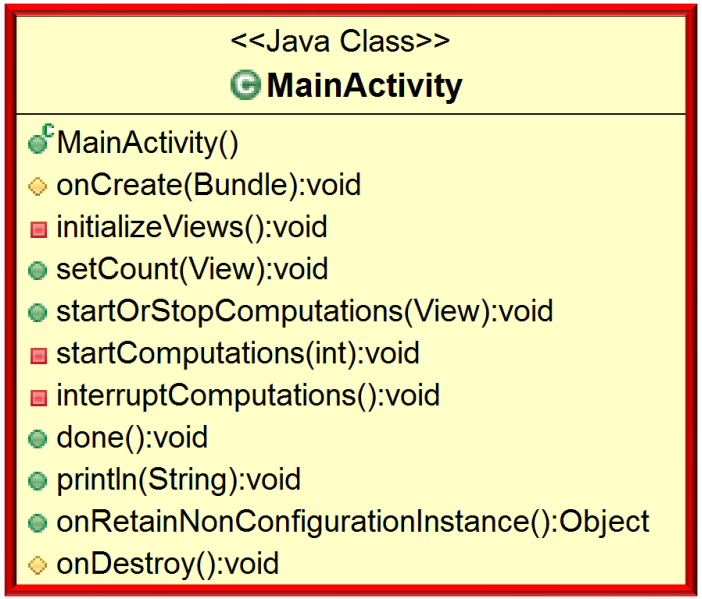
See docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

Overview of the PrimeChecker App

- MainActivity creates a list of futures that store results of concurrently checking primality of "count" random #'s within a range

```
List<Future<PrimeResult>>
   futures = new Random()
      .longs(count,
             sMAX_VALUE - count,
             sMAX_VALUE)

      .mapToObj(PrimeCallable::new)

      .map(mRetainedState.mExecutorService::submit)
```





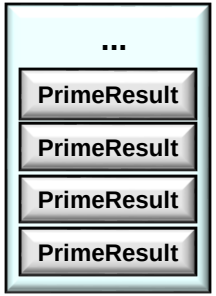
Submit a two-way task for execution & return a future representing pending task results

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#submit

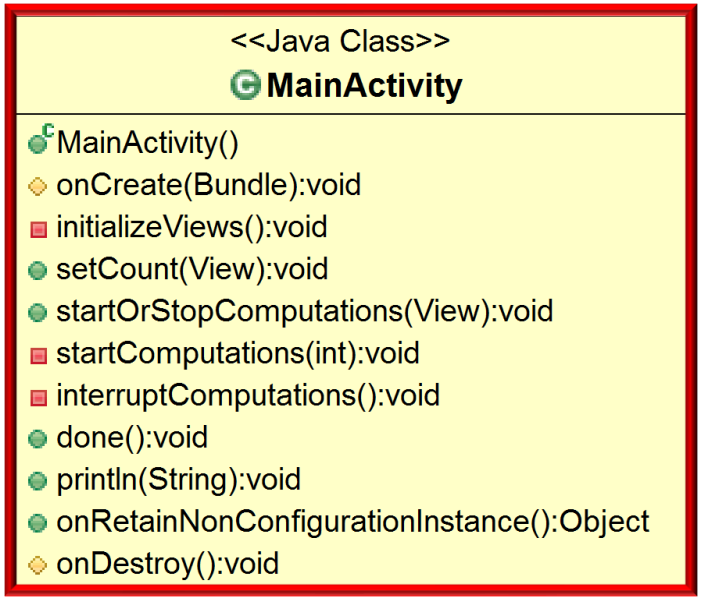
Overview of the PrimeChecker App

- MainActivity creates a list of futures that store results of concurrently checking primality of "count" random #'s within a range

```
List<Future<PrimeResult>>
   futures = new Random()
      .longs(count,
             sMAX_VALUE - count,
             sMAX_VALUE)

      .mapToObj(PrimeCallable::new)

      .map(mRetainedState.mExecutorService::submit)

      .collect(toList());
```



List of Futures



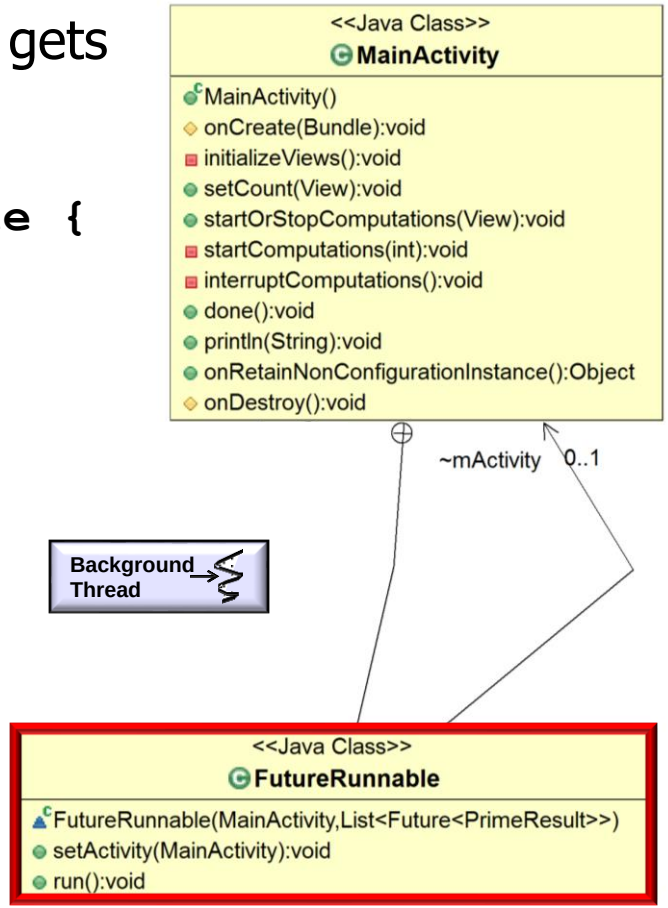
Collect results into a list of futures to PrimeResults

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#collect

Overview of the PrimeChecker App

- FutureRunnable runs in a background thread & gets the results of all futures as they complete

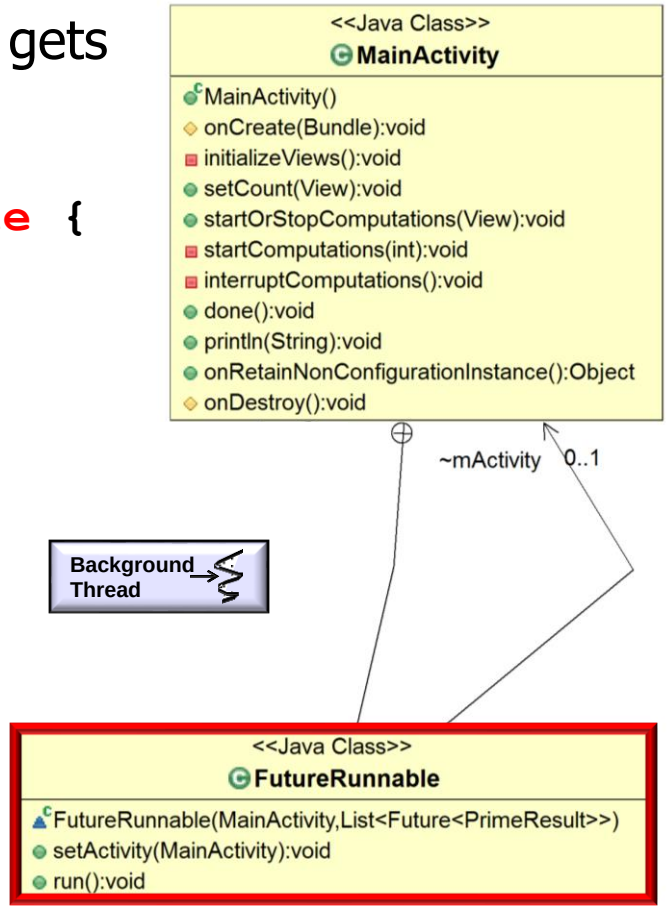
```
...
class FutureRunnable implements Runnable {
  List<Future<PrimeResult>>
    mFutures;


  MainActivity mActivity;
  ...

  FutureRunnable(MainActivity a,
                 List<Future<PrimeResult>> f)
  { mActivity = a; mFutures = f; }
  ...
```



Overview of the PrimeChecker App

- FutureRunnable runs in a background thread & gets the results of all futures as they complete

```
...
class FutureRunnable implements Runnable {
  List<Future<PrimeResult>>
    mFutures;


  MainActivity mActivity;
  ...

  FutureRunnable(MainActivity a,
                 List<Future<PrimeResult>> f)
  { mActivity = a; mFutures = f; }
  ...
```

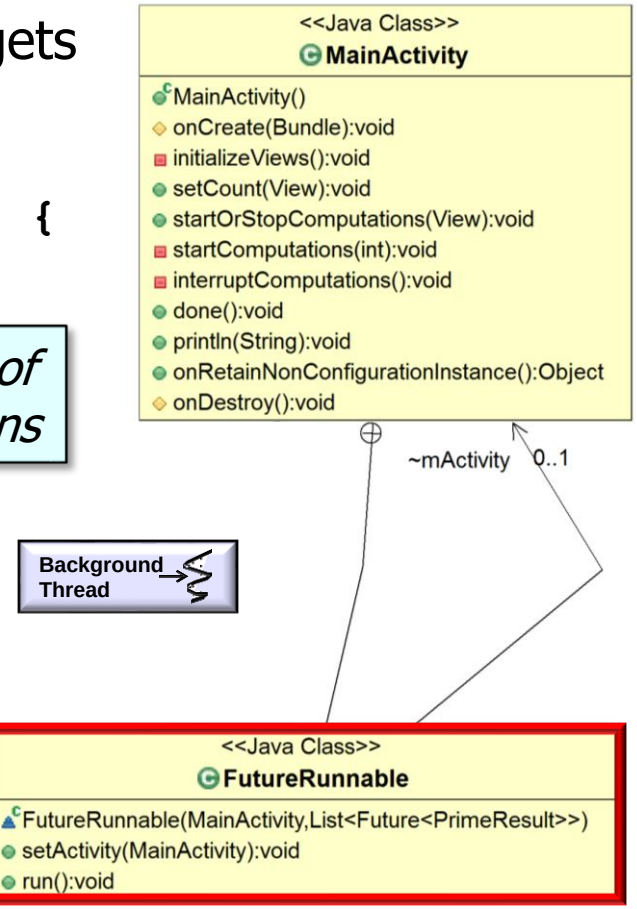


Overview of the PrimeChecker App

- FutureRunnable runs in a background thread & gets the results of all futures as they complete

```
...
class FutureRunnable implements Runnable {
  List<Future<PrimeResult>>
    mFutures;


  MainActivity mActivity;
  ...

  FutureRunnable(MainActivity a,
                 List<Future<PrimeResult>> f)
  { mActivity = a; mFutures = f; }
  ...
```

List of futures to results of PrimeCallable computations

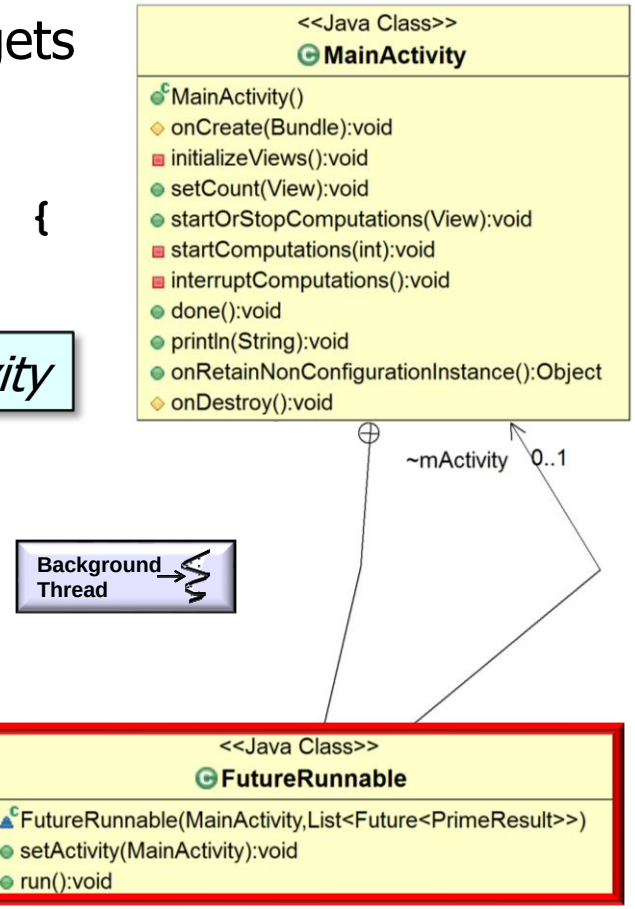


Overview of the PrimeChecker App

- FutureRunnable runs in a background thread & gets the results of all futures as they complete

```
...
class FutureRunnable implements Runnable {
  List<Future<PrimeResult>>
    mFutures;

  MainActivity mActivity;
  ...

  FutureRunnable(MainActivity a,
                 List<Future<PrimeResult>> f)
  { mActivity = a; mFutures = f; }
  ...
```

Reference back to enclosing activity

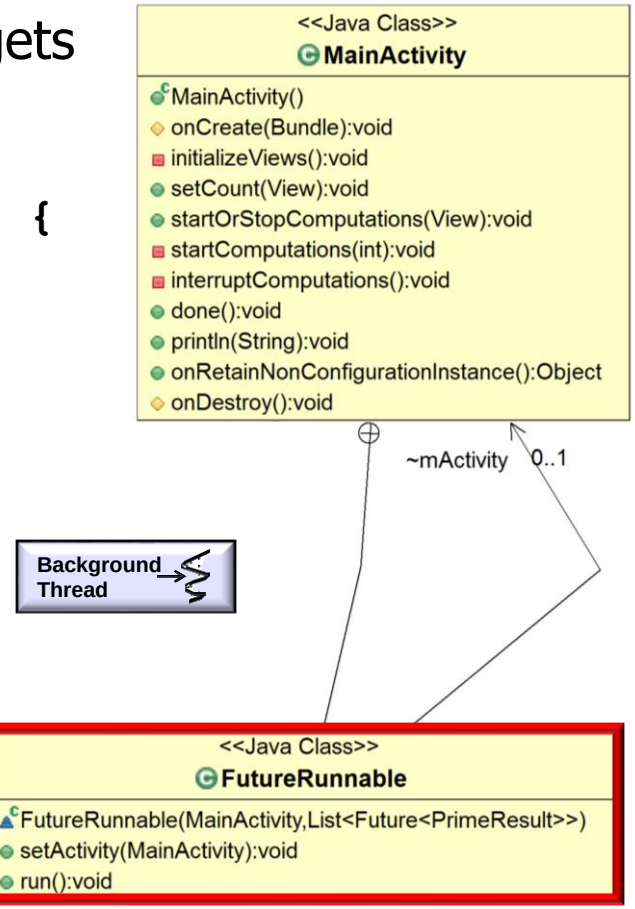


Overview of the PrimeChecker App

- FutureRunnable runs in a background thread & gets the results of all futures as they complete

```
...
class FutureRunnable implements Runnable {
  List<Future<PrimeResult>>
    mFutures;


  MainActivity mActivity;
  ...

  FutureRunnable(MainActivity a,
                 List<Future<PrimeResult>> f)
  { mActivity = a; mFutures = f; }
  ...
```

Constructor initializes the fields

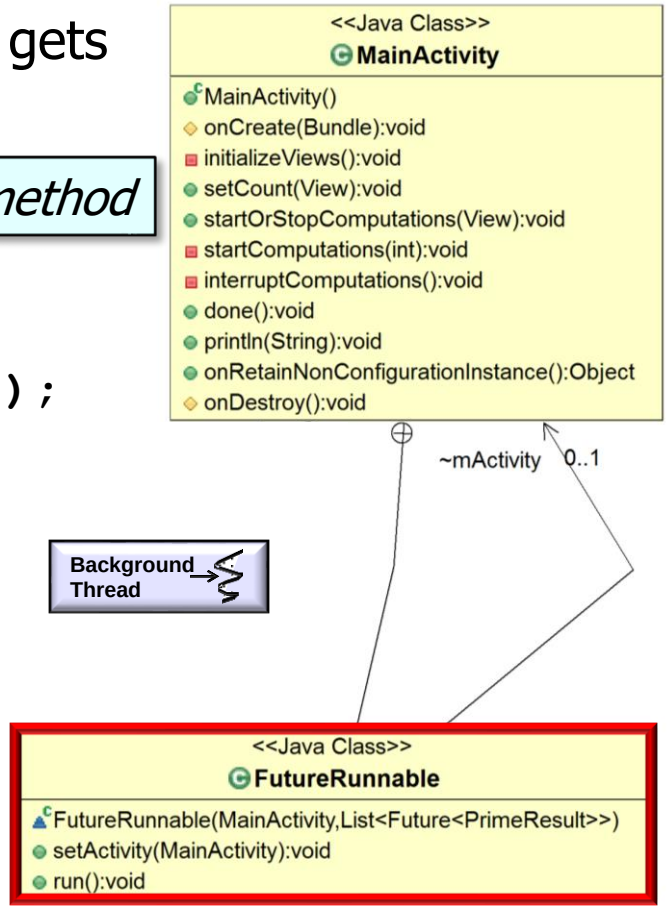


Overview of the PrimeChecker App

- FutureRunnable runs in a background thread & gets the results of all futures as they complete

```
...
public void run() {
  mFutures.forEach(future -> {
    PrimeCallable.PrimeResult pr =
      rethrowSupplier(future::get).get();

    if (pr.mSmallestFactor != 0)
      ...
    else ...});


  mActivity.done(); ...
```

Runnable hook method

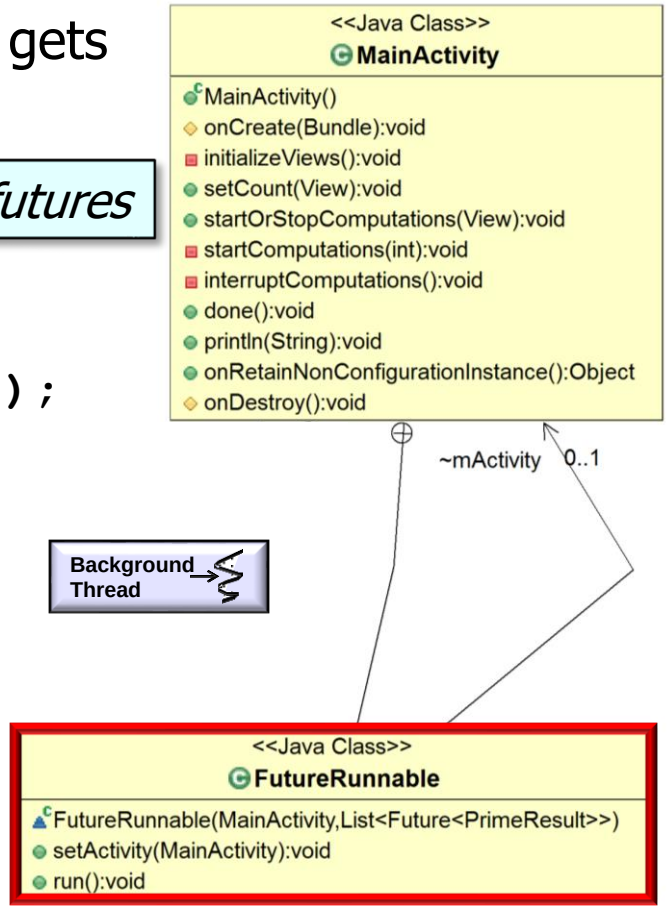


Overview of the PrimeChecker App

- FutureRunnable runs in a background thread & gets the results of all futures as they complete

```
...
public void run() {
  mFutures.forEach(future -> {
    PrimeCallable.PrimeResult pr =
      rethrowSupplier(future::get).get();

    if (pr.mSmallestFactor != 0)
      ...
    else ...});


  mActivity.done(); ...
```

Iterate thru all futures



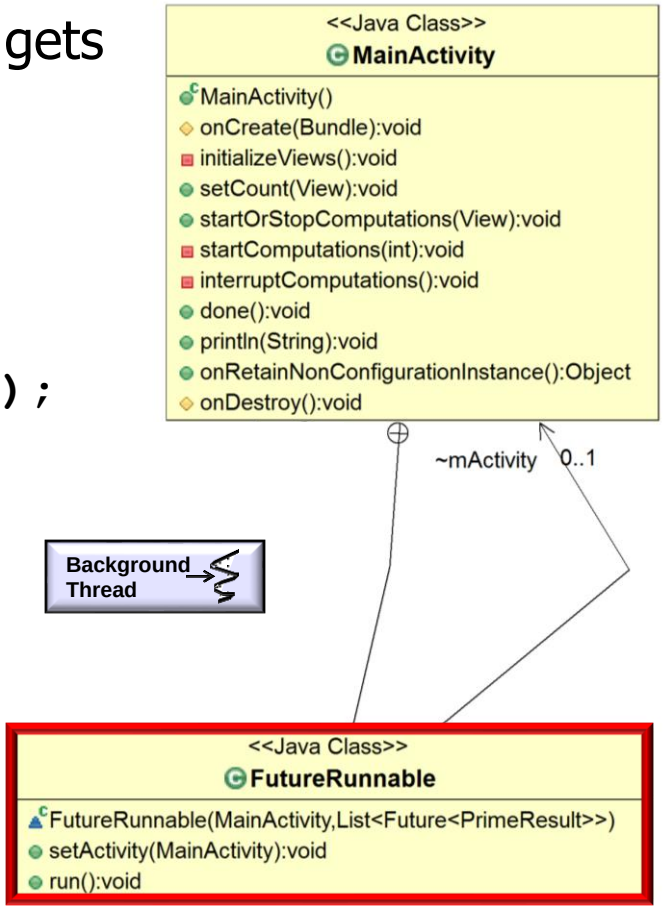
Overview of the PrimeChecker App

- FutureRunnable runs in a background thread & gets the results of all futures as they complete

```
...
public void run() {
  mFutures.forEach(future -> {
    PrimeCallable.PrimeResult pr =
      rethrowSupplier(future::get).get();

    if (pr.mSmallestFactor != 0)
      ...
    else ...});
```

future::get blocks if async processing associated with future hasn't completed

```
  mActivity.done(); ...
```



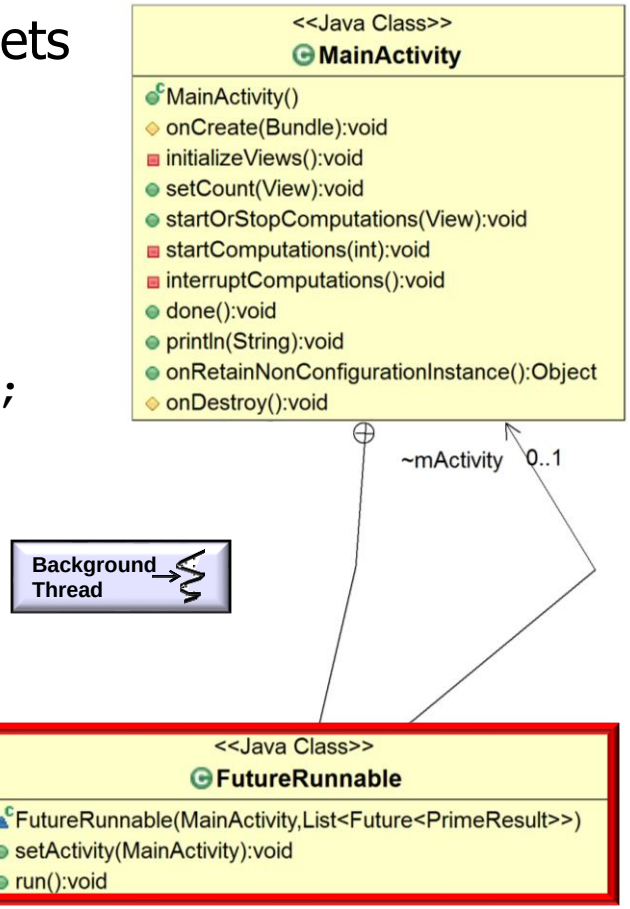
This is an example of the "synchronous future" processing model

Overview of the PrimeChecker App

- FutureRunnable runs in a background thread & gets the results of all futures as they complete

```
...
public void run() {
  mFutures.forEach(future -> {
    PrimeCallable.PrimeResult pr =
      rethrowSupplier(future::get).get();

    if (pr.mSmallestFactor != 0)
      ...
    else ...});


  mActivity.done(); ...
```

Convert checked exception to a runtime exception



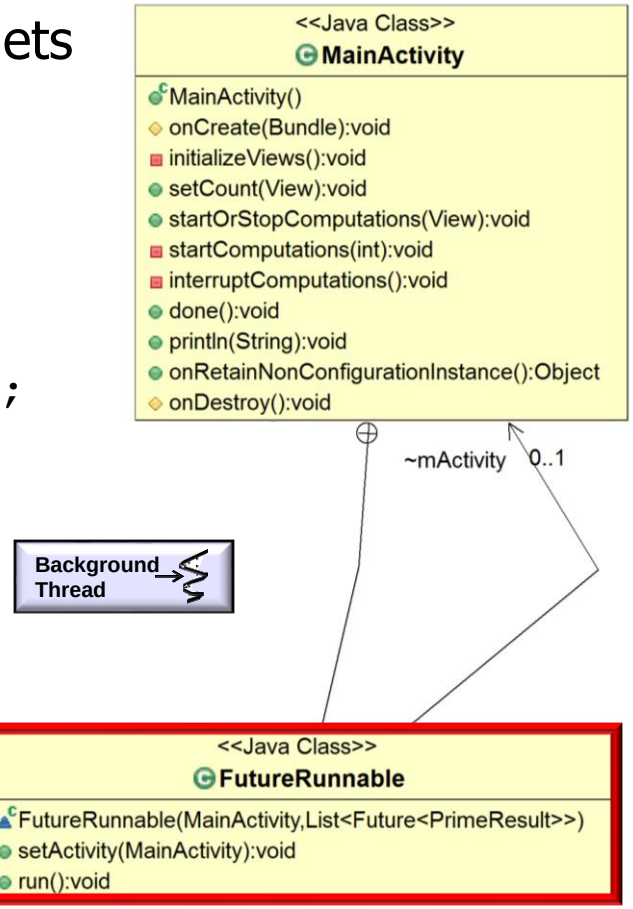
See stackoverflow.com/a/27644392/3312330

Overview of the PrimeChecker App

- FutureRunnable runs in a background thread & gets the results of all futures as they complete

```
...
public void run() {
  mFutures.forEach(future -> {
    PrimeCallable.PrimeResult pr =
      rethrowSupplier(future::get).get();

    if (pr.mSmallestFactor != 0)
      ...
    else ...});

  mActivity.done(); ...
```

Get the result from the supplier



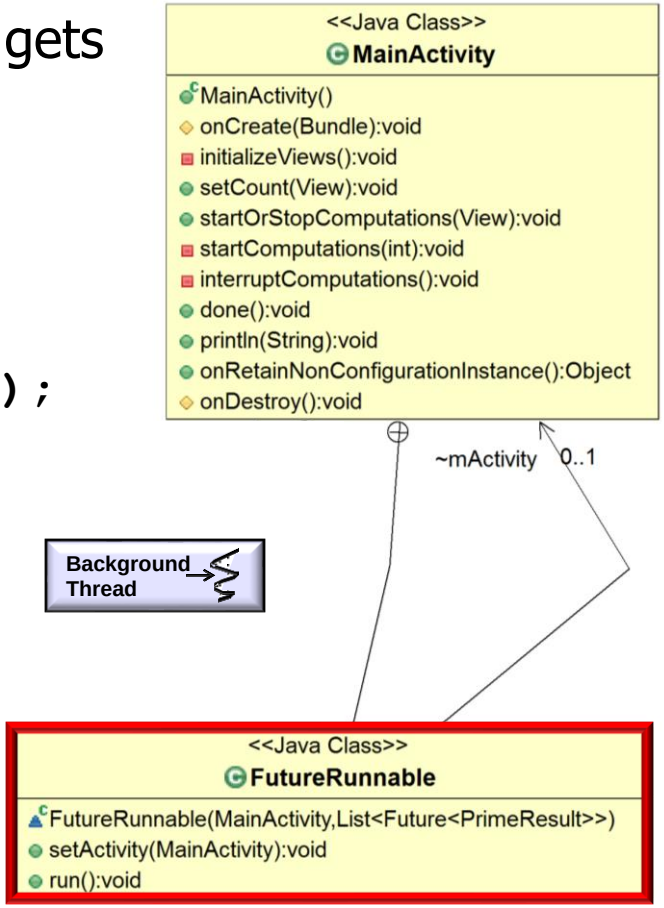
See docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html#get

Overview of the PrimeChecker App

- FutureRunnable runs in a background thread & gets the results of all futures as they complete

```
...
public void run() {
  mFutures.forEach(future -> {
    PrimeCallable.PrimeResult pr =
      rethrowSupplier(future::get).get();

    if (pr.mSmallestFactor != 0)
      ...
    else ...});

  mActivity.done(); ...
```

Process each result & produce output



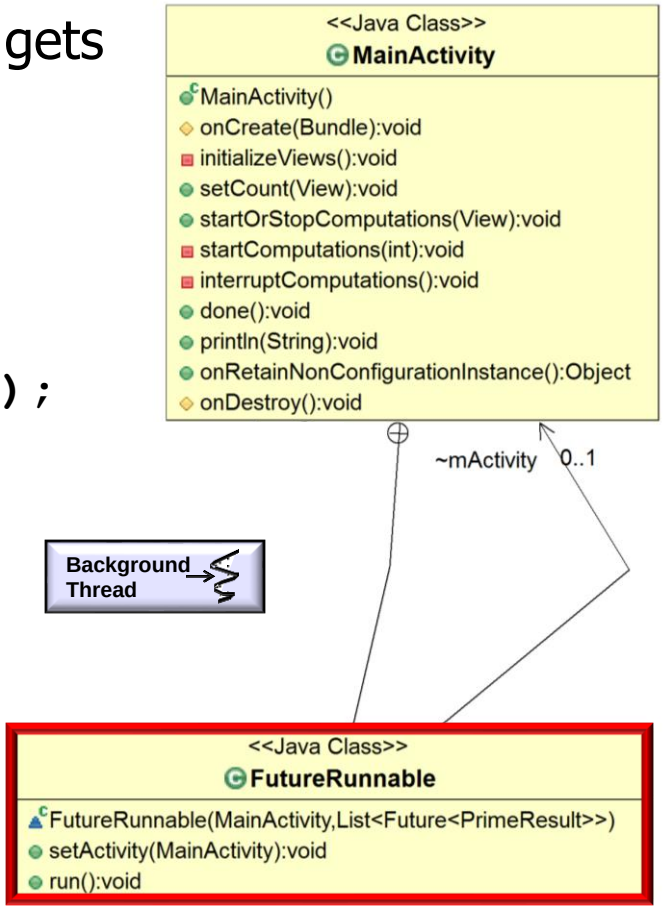
Overview of the PrimeChecker App

- FutureRunnable runs in a background thread & gets the results of all futures as they complete

```
...
public void run() {
  mFutures.forEach(future -> {
    PrimeCallable.PrimeResult pr =
      rethrowSupplier(future::get).get();

    if (pr.mSmallestFactor != 0)
      ...
    else ...});
```

Inform MainActivity that we're all done

```
  mActivity.done(); ...
```

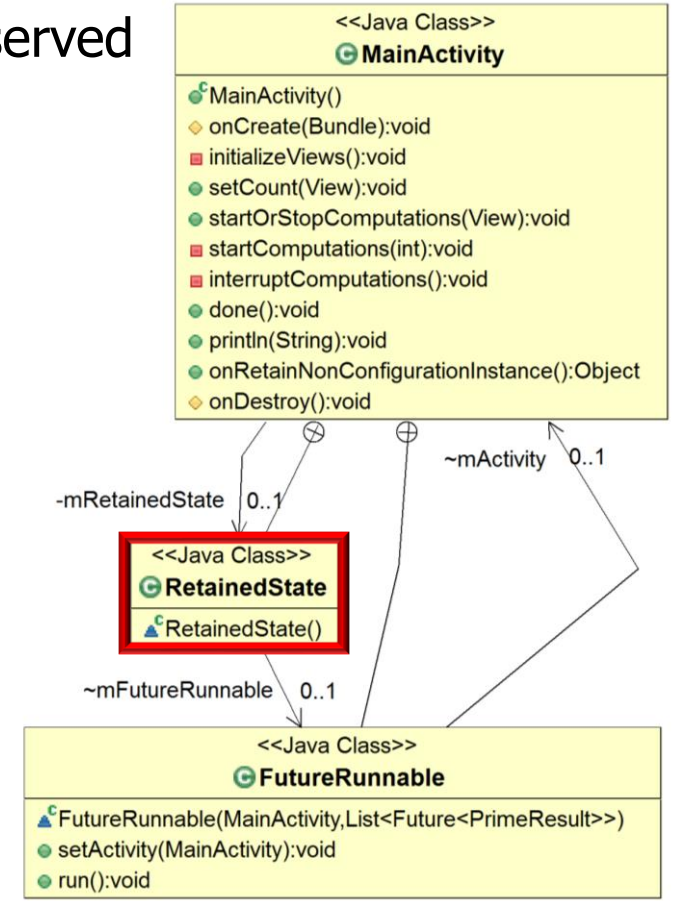


Overview of the PrimeChecker App

- RetainedState contains fields that must be preserved across runtime configuration changes

```
class RetainedState {
   ExecutorService mExecutorService;
   FutureRunnable mFutureRunnable;
   Thread mThread;
}
```

These fields store concurrency-related objects



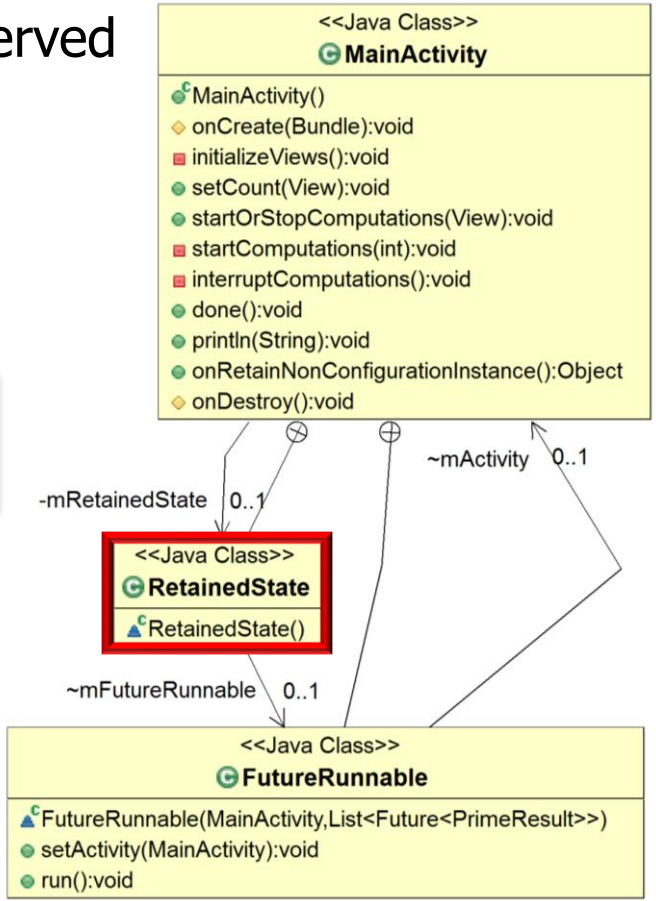
Overview of the PrimeChecker App

- RetainedState contains fields that must be preserved across runtime configuration changes

```
...
mRetainedState.mFutureRunnable =
   new FutureRunnable(this, futures);
```

FutureRunnable is stored in a field so its state can be updated during a runtime configuration change

```
mRetainedState.mThread =
   new Thread(mRetainedState
              .mFutureRunnable);

mRetainedState.mThread.start();
```



See developer.android.com/guide/topics/resources/runtime-changes.html

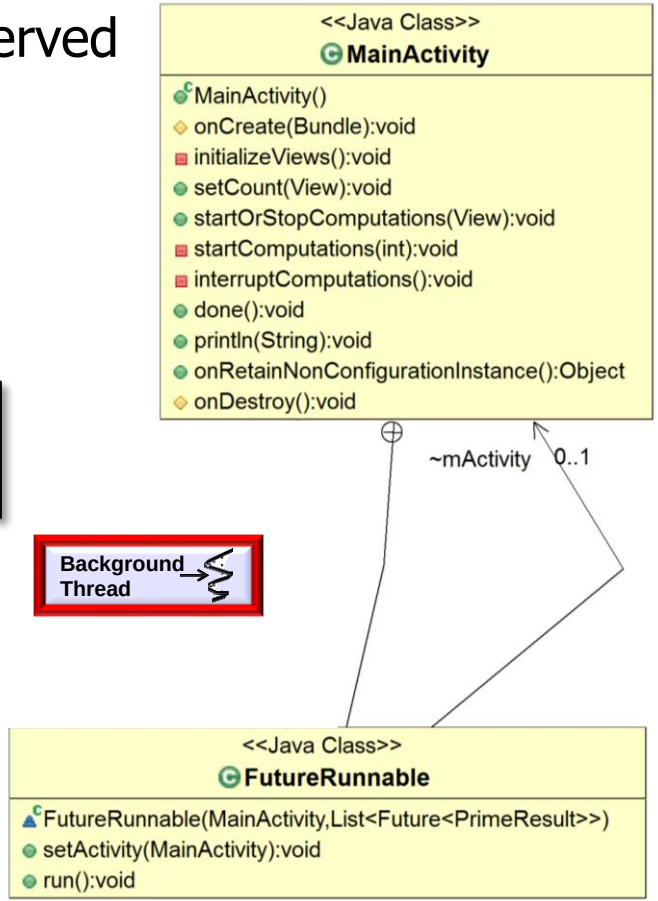
Overview of the PrimeChecker App

- RetainedState contains fields that must be preserved across runtime configuration changes

```
...
mRetainedState.mFutureRunnable =
   new FutureRunnable(this, futures);
```

A background thread is started to wait for all future results to avoid blocking the UI thread

```
mRetainedState.mThread =
   new Thread(mRetainedState
              .mFutureRunnable);

mRetainedState.mThread.start();
```



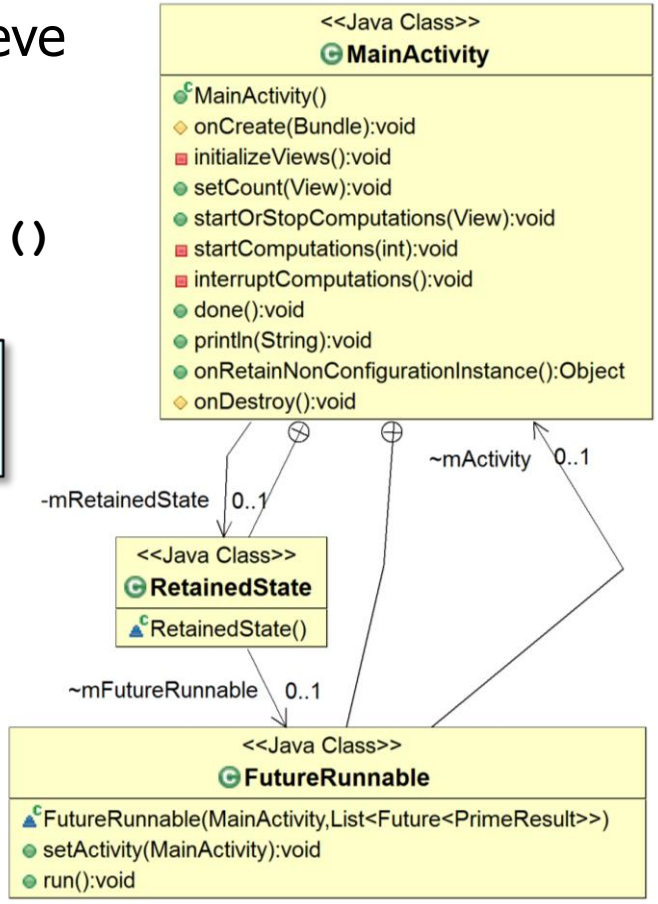
See developer.android.com/training/articles/perf-anr.html

Overview of the PrimeChecker App

- Android provides hook methods to store & retrieve app state across runtime configuration changes

```
...
Object onRetainNonConfigurationInstance()
{ return mRetainedState; }
...


void onCreate(...) {
  mRetainedState = (RetainedState)
   getLastNonConfigurationInstance();

 if (mRetainedState != null) {
   ...
```

Retained state is loaded/stored via Android hook methods



See developer.android.com/reference/android/app/Activity.html#onRetainNonConfigurationInstance()

Evaluating this PrimeChecker App

Evaluating this PrimeChecker App

- ExecutorService version of PrimeChecker app fixes problems with earlier Executor PrimeChecker

Evaluating this PrimeChecker App

- ExecutorService version of PrimeChecker app fixes problems with earlier Executor PrimeChecker, e.g.
  - Two-way semantics of Java callables decouple PrimeCallable & MainActivity


```
public class PrimeCallable
       implements Callable<PrimeResult> {
   ...

   public PrimeCallable(long PrimeCandidate) { ... }

   public PrimeResult call() {
     return new PrimeResult(mPrimeCandidate,
                            isPrime(mPrimeCandidate));
   } ...
```

MainActivity appears nowhere in PrimeCallable class..

This decoupling simplifies runtime configuration changes

Evaluating this PrimeChecker App

- ExecutorService version of PrimeChecker app fixes problems with earlier Executor PrimeChecker, e.g.
  - Two-way semantics of Java callables decouple PrimeCallable & MainActivity
  - Lifecycle operations enable task interruptions

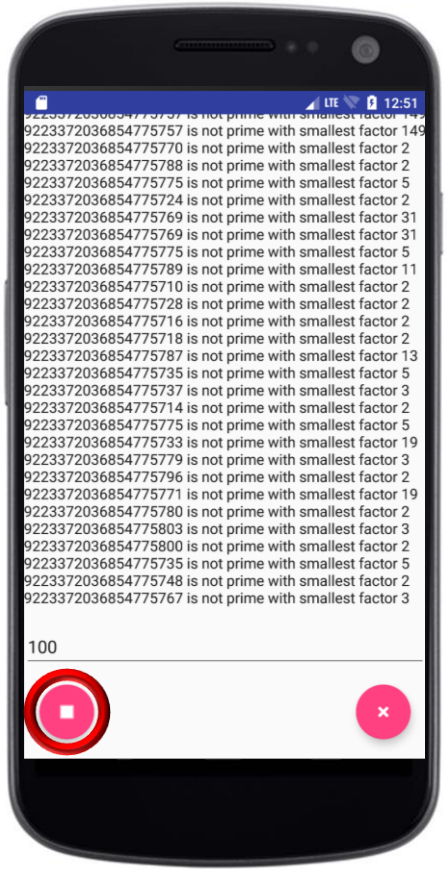
```
void interruptComputations() {
  mRetainedState.mExecutorService
                .shutdownNow();

  mRetainedState.mThread.interrupt();
  ...
  mRetainedState
    .mExecutorService.awaitTermination
      (500, TimeUnit.MILLISECONDS);
```

Shutting down an executor service interrupts *all* threads running tasks

Evaluating this PrimeChecker App

- ExecutorService version of PrimeChecker app fixes problems with earlier Executor PrimeChecker, e.g.
  - Two-way semantics of Java callables decouple PrimeCallable & MainActivity
  - Lifecycle operations enable task interruptions

```
long isPrime(long n) {
  if (n > 3)
    for (long factor = 2;
         factor <= n / 2; ++factor)
      if (Thread.interrupted()) break;
      else if (n / factor * factor == n)
        return factor;
  return 0L;
}
```



The isPrime() method repeatedly checks to see if it's been interrupted

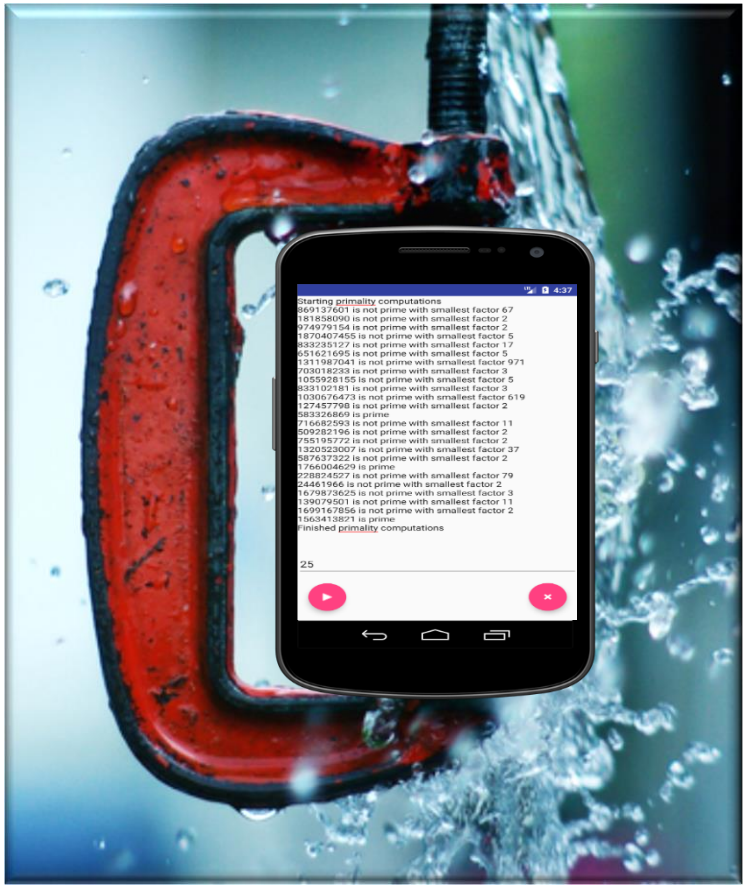
Evaluating this PrimeChecker App

- ExecutorService version of PrimeChecker app fixes problems with earlier Executor PrimeChecker, e.g.
  - Two-way semantics of Java callables decouple PrimeCallable & MainActivity
  - Lifecycle operations enable task interruptions
  - Runtime configuration changes handled gracefully

Running tasks execute & update the GUI until they finish or are interrupted

Evaluating this PrimeChecker App

- However, there are still some limitations



Evaluating this PrimeChecker App

- However, there are still some limitations, e.g.
  - future::get blocks the thread, even if other futures may have completed

```
private class FutureRunnable
  implements Runnable {
  MainActivity mActivity; ...

 public void run() {
   mFutures.forEach(future -> {
     PrimeCallable.PrimeResult pr =
       rethrowSupplier(future::get).get();

   if (pr.mSmallestFactor != 0) ...
   else ...
   mActivity.done(); ...
```

This problem is inherent with the "synchronous future" processing model

We fix this problem in an upcoming lesson on "*Java ExecutorCompletionService*"!

Evaluating this PrimeChecker App

- However, there are still some limitations, e.g.
  - future::get blocks the thread, even if other futures may have completed
  - isPrime() tightly coupled with PrimeCallable

```
public class PrimeCallable ... {
  long isPrime(long n) {
    if (n > 3)
      for (long factor = 2; factor <= n / 2; ++factor)
        if (Thread.interrupted())
          break;
        else if (n / factor * factor == n)
          return factor;

    return 0L;
  } ...
```

Primality checking always runs, even if results were computed previously

This problem is fixed by Memoizer in an upcoming lesson on "*Java FutureTask*"!

End of Overview of Java ExecutorService (Part 4)