Motivating the Need for Java Futures

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

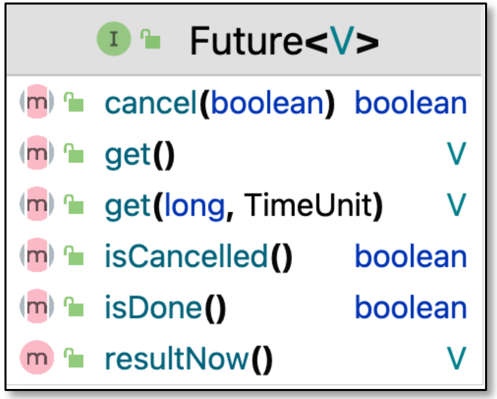
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Lesson

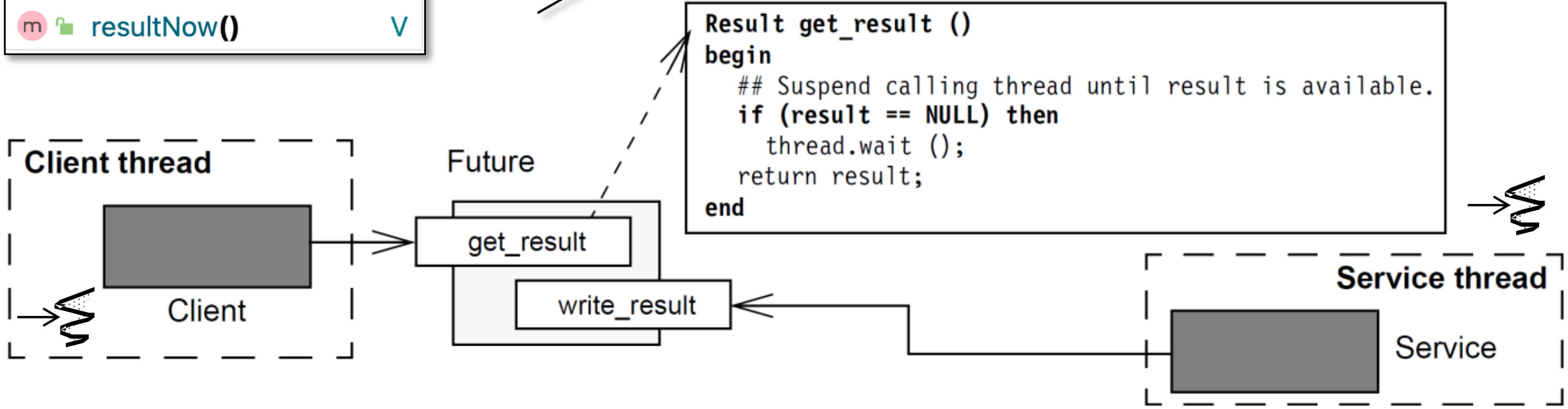
- Understand the need for the *Future* pattern & Java Future interface



Future<V>

cancel(boolean)	boolean
get()	V
get(long, TimeUnit)	V
isCancelled()	boolean
isDone()	boolean
resultNow()	V

A future provides a means to retrieve the result of a computation being executed asynchronously, without indefinitely blocking the client thread



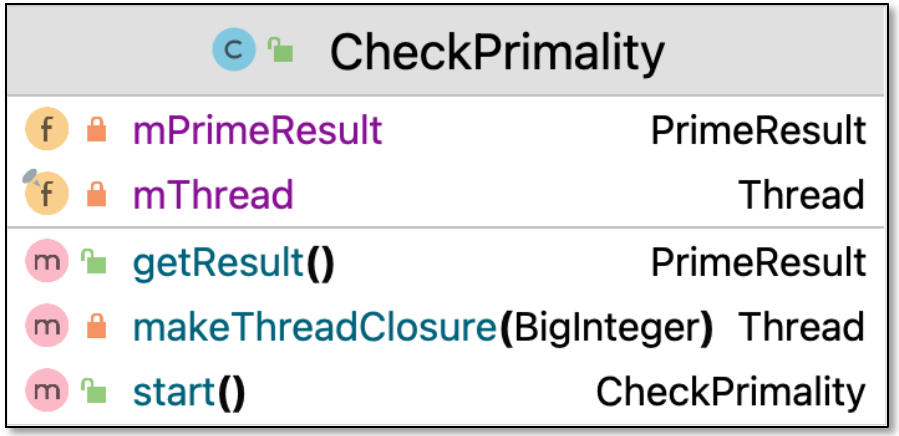
```
Result get_result ()
begin
  ## Suspend calling thread until result is available.
  if (result == NULL) then
    thread.wait ();
  return result;
end
```

See en.wikipedia.org/wiki/Futures_and_promises

Motivating the Need for Java Futures

Motivating the Need for Java Futures

- The CheckPrimality class showed how a closure could store the results of a computation running in a Java Thread

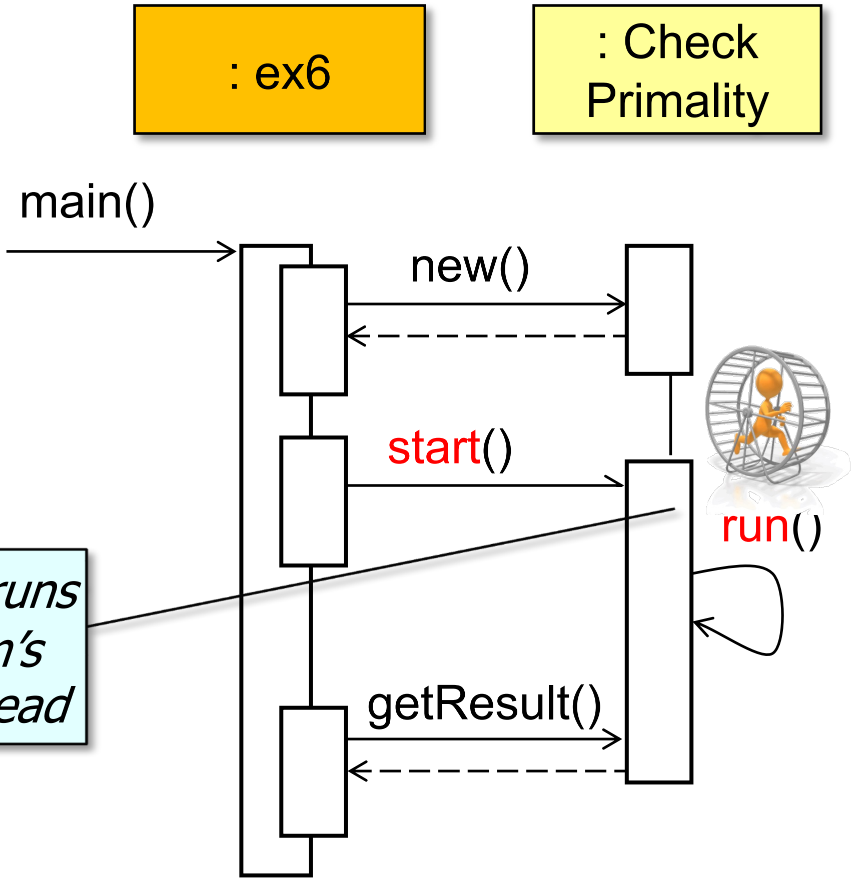


```
CheckPrimality
mPrimeResult                         PrimeResult
mThread                              Thread
getResult()                          PrimeResult
makeThreadClosure(BigInteger)        Thread
start()                              CheckPrimality
```

See earlier lessons on "*Implementing Closures with Java Lambda Expressions*"

Motivating the Need for Java Futures

- The CheckPrimality class showed how a closure could store the results of a computation running in a Java Thread

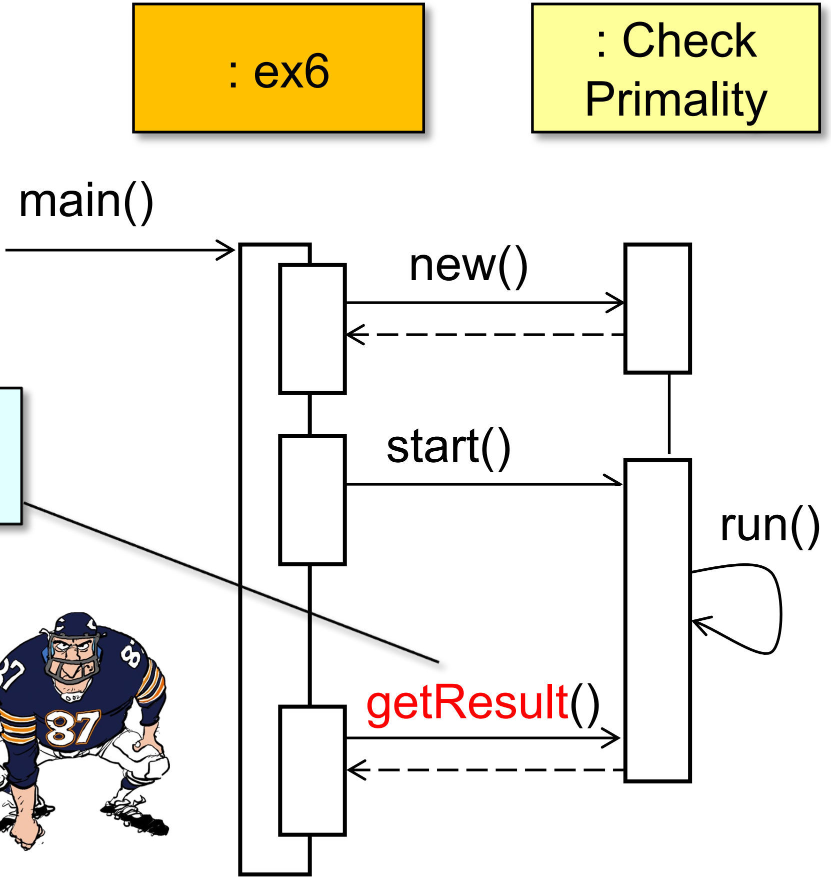


The computation for primality checking runs asynchronously after the main program's client Thread starts the background Thread

See ModernJava/blob/main/FP/ex6/src/main/java/CheckPrimality.java

Motivating the Need for Java Futures

- The CheckPrimality class showed how a closure could store the results of a computation running in a Java Thread

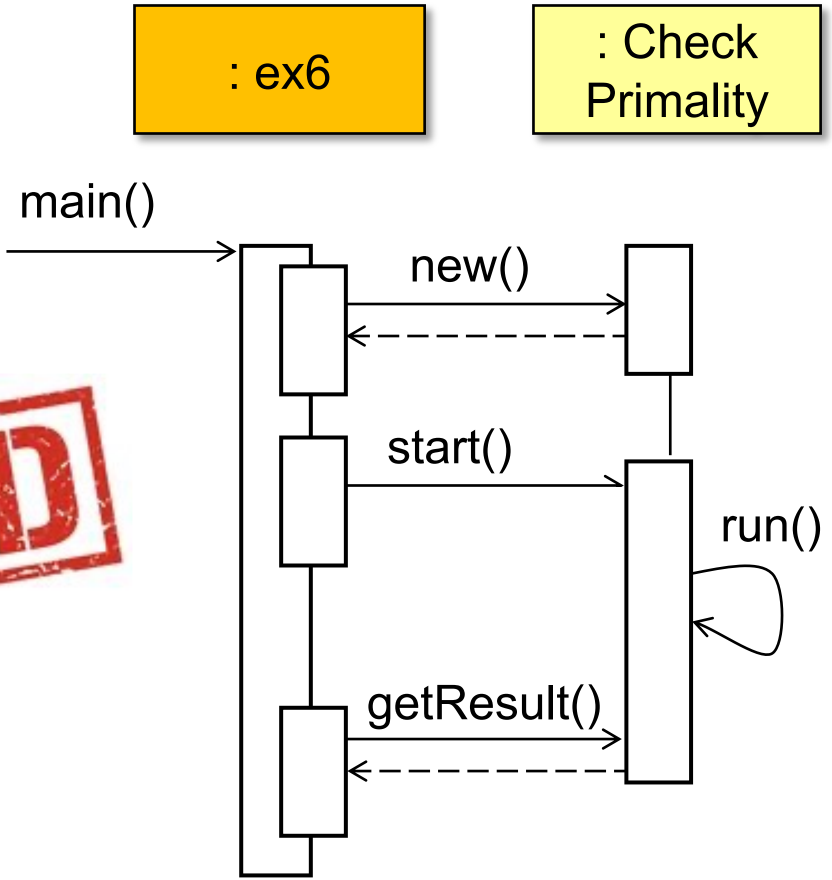


The client Thread blocks until the result is available

See www.geeksforgeeks.org/blocking-methods-in-java

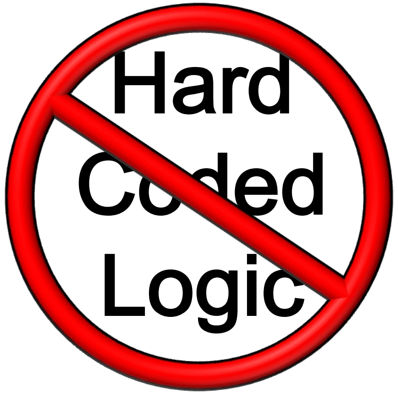
Motivating the Need for Java Futures

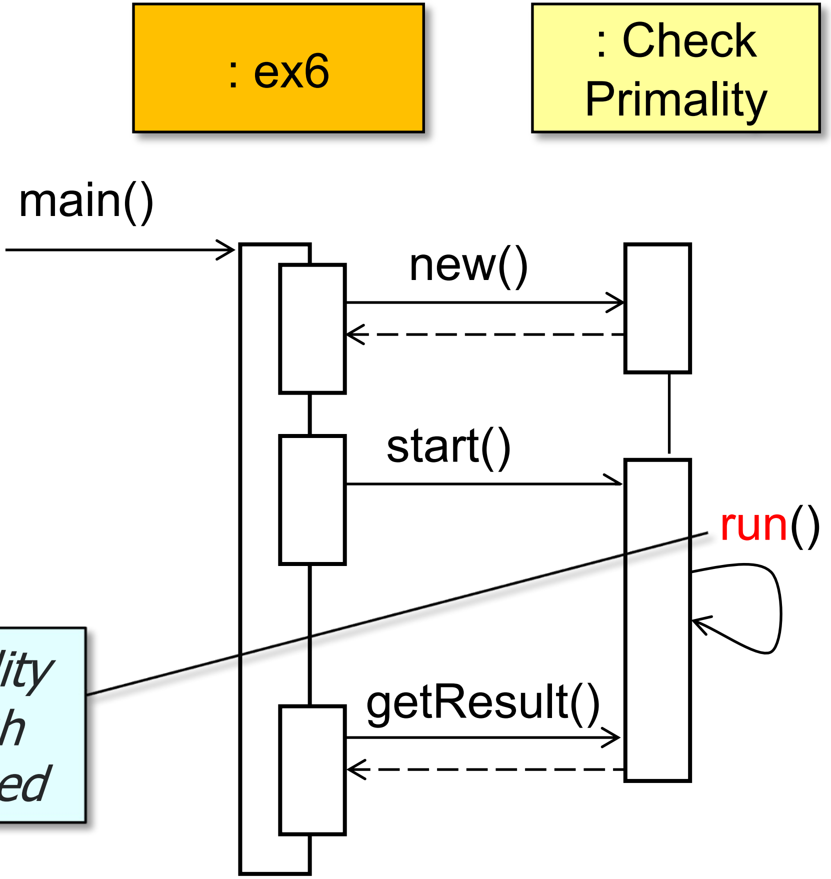
- Although CheckPrimality provides some useful features, there are two limitations



Motivating the Need for Java Futures

- Although CheckPrimality provides some useful features, there are two limitations
  - Its behavior is "hard-coded"





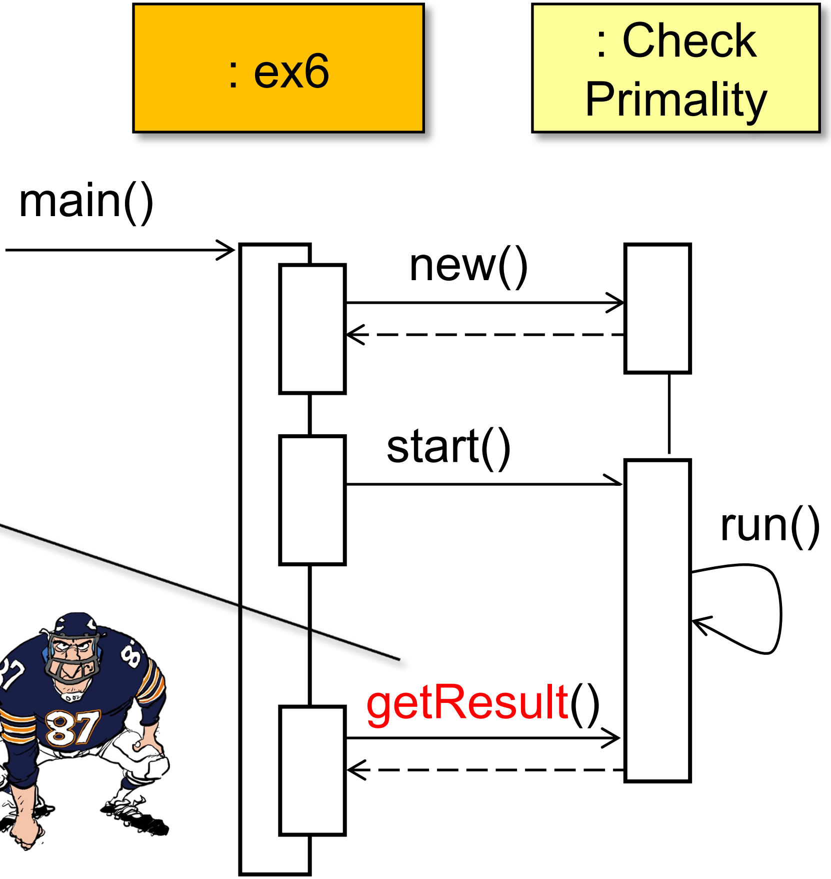
i.e., it only checks the primality of a BigInteger, even though its design could be generalized

See en.wikipedia.org/wiki/Hard_coding

Motivating the Need for Java Futures

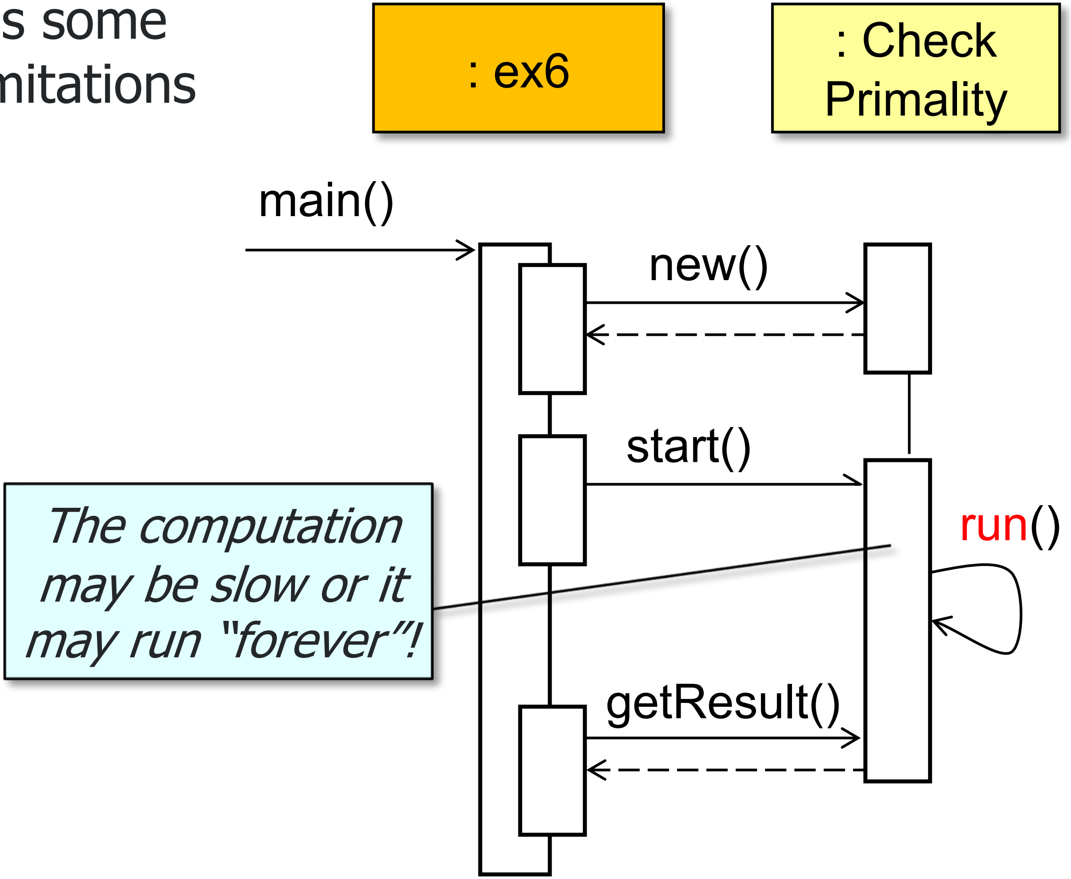
- Although CheckPrimality provides some useful features, there are two limitations
  - Its behavior is "hard-coded"
  - getResult() blocks indefinitely

i.e., waiting for the completion of the primality computation



Motivating the Need for Java Futures

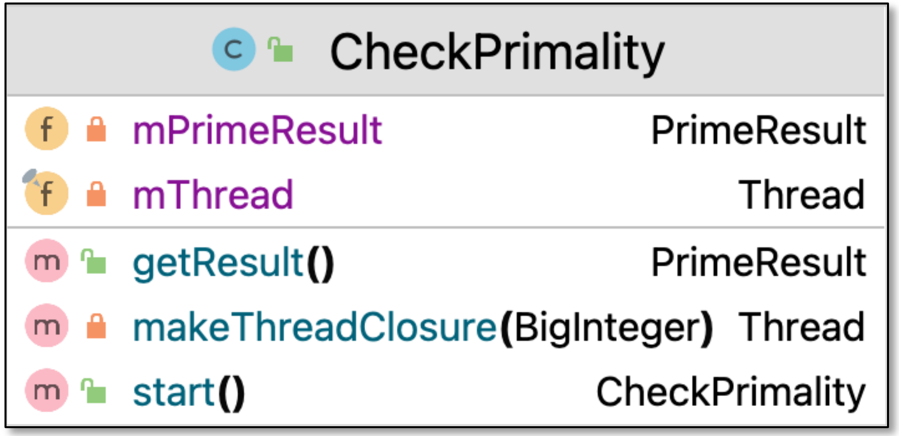
- Although CheckPrimality provides some useful features, there are two limitations
  - Its behavior is "hard-coded"
  - getResult() blocks indefinitely



Applying Java Futures to Address These Limitations

Applying Java Futures to Address These Limitations

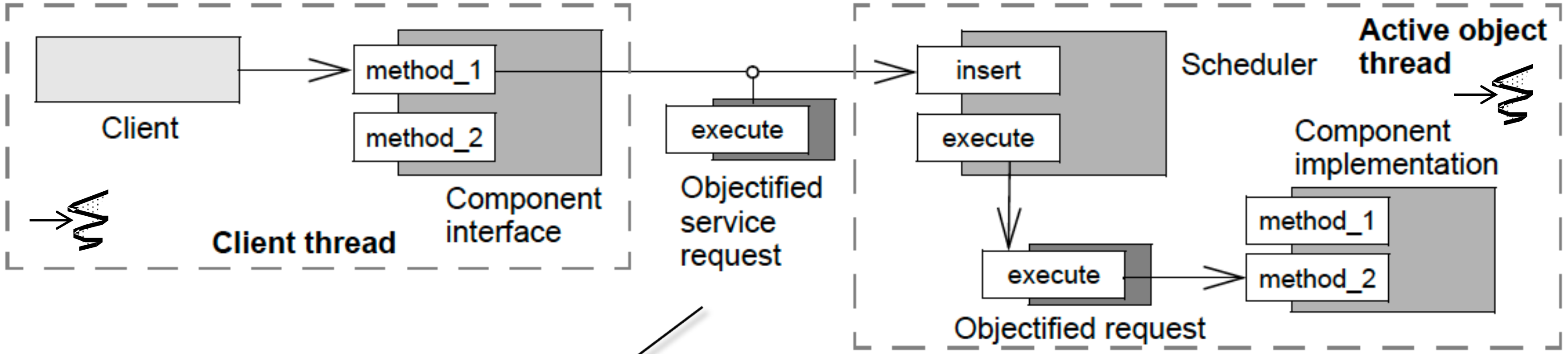
- We address these limitations in 2 ways



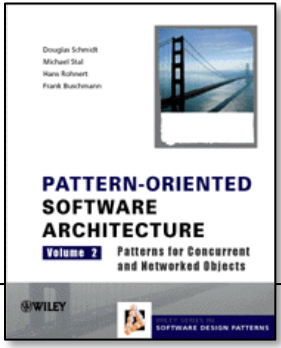
CheckPrimality	
mPrimeResult	PrimeResult
mThread	Thread
getResult()	PrimeResult
makeThreadClosure(BigInteger)	Thread
start()	CheckPrimality

Applying Java Futures to Address These Limitations

- We address these limitations in 2 ways
  - Apply the *Active Object* pattern to create generic concurrent objects



This pattern decouples method execution from method invocation for objects residing in their own thread of control

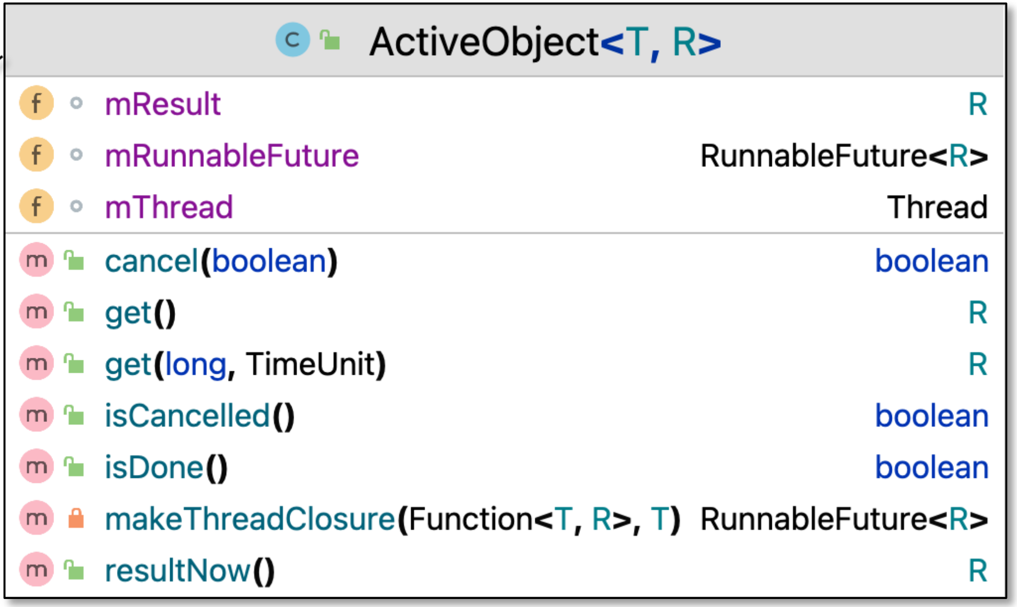


Douglas Schmidt
Michael Stal
Hans Rohnert
Frank Buschmann
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
Volume 2
Patterns for Concurrent and Networked Objects
WILEY

See en.wikipedia.org/wiki/Active_object

Applying Java Futures to Address These Limitations

- We address these limitations in 2 ways
  - Apply the *Active Object* pattern to create generic concurrent objects



ActiveObject<T, R>

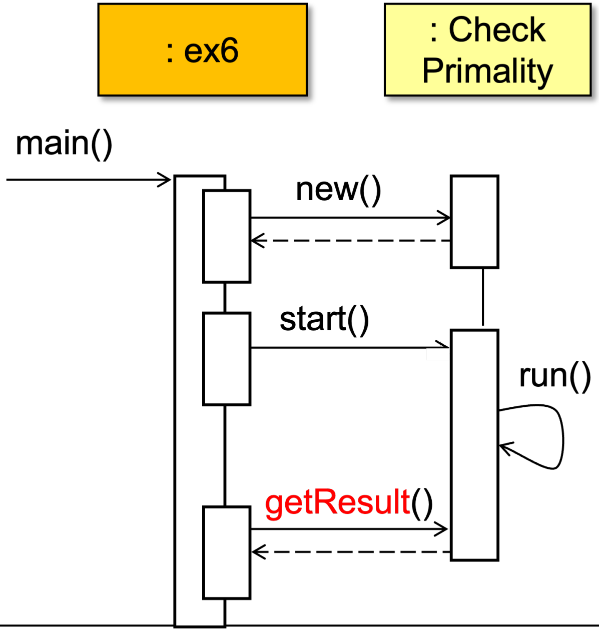
mResult	R
mRunnableFuture	RunnableFuture<R>
mThread	Thread
cancel(boolean)	boolean
get()	R
get(long, TimeUnit)	R
isCancelled()	boolean
isDone()	boolean
makeThreadClosure(Function<T, R>, T)	RunnableFuture<R>
resultNow()	R

This class implements a variant of the Active Object pattern using modern Java features (e.g., a virtual Thread & the Function functional interface)

See ModernJava/blob/main/FP/ex16/src/main/java/utils/ActiveObject.java

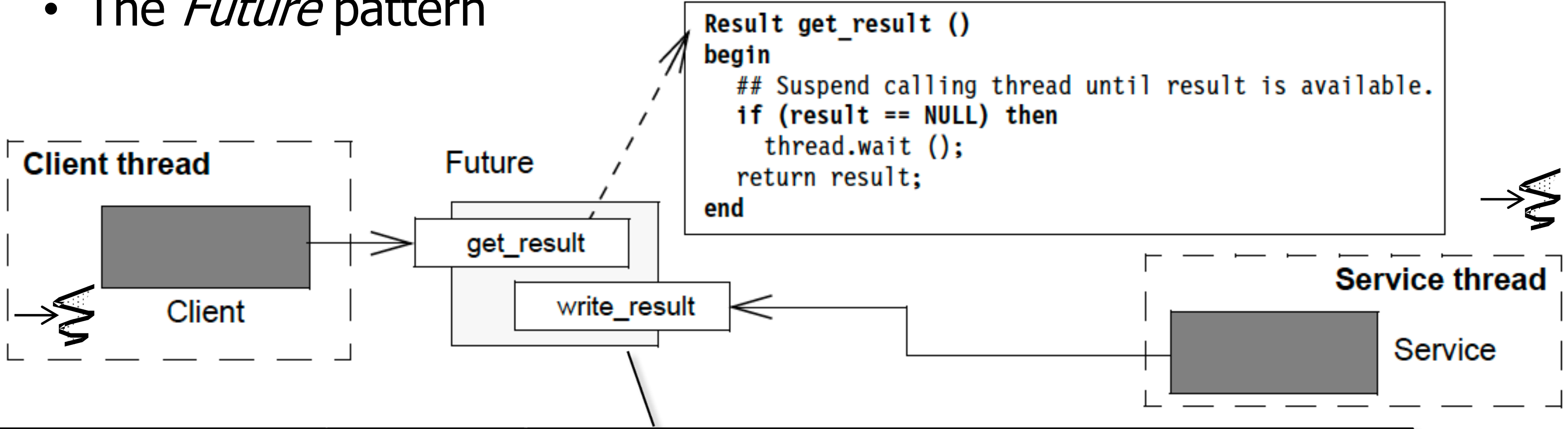
Applying Java Futures to Address These Limitations

- We address these limitations in 2 ways
  - Apply the *Active Object* pattern to create generic concurrent objects
  - Apply the *Future* pattern & Java Future interface to avoid indefinite blocking



Applying Java Futures to Address These Limitations

- We address these limitations in 2 ways
  - Apply the *Active Object* pattern to create generic concurrent objects
  - Apply the *Future* pattern & Java Future interface to avoid indefinite blocking
    - The *Future* pattern



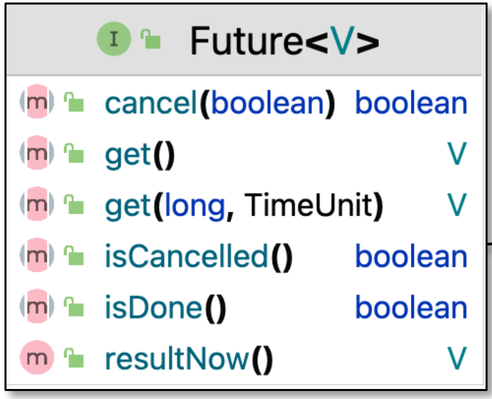
```
Result get_result ()
begin
   ## Suspend calling thread until result is available.
   if (result == NULL) then
     thread.wait ();
   return result;
end
```

Provides a 'virtual' data object that blocks (or do not block) clients when they try to get its contents before its concurrent computation completes

See en.wikipedia.org/wiki/Futures_and_promises

Applying Java Futures to Address These Limitations

- We address these limitations in 2 ways
  - Apply the *Active Object* pattern to create generic concurrent objects
  - Apply the *Future* pattern & Java Future interface to avoid indefinite blocking
    - The *Future* pattern
    - The Future interface



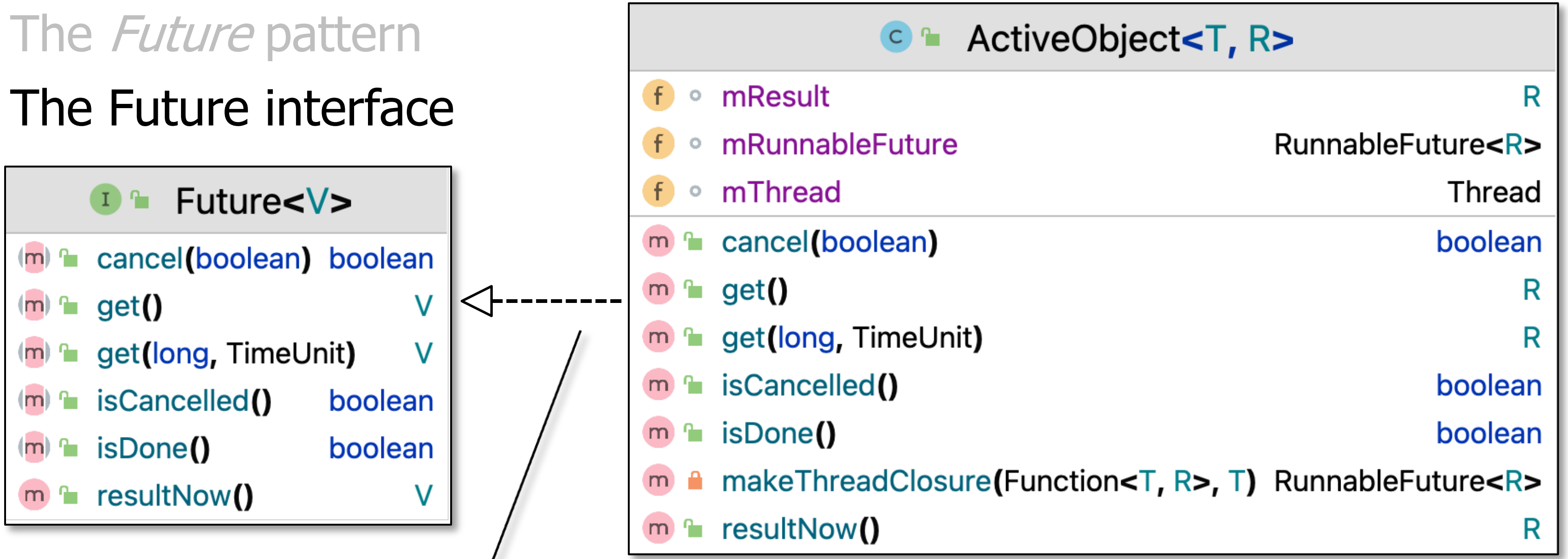
```
Future<V>
cancel(boolean)       boolean
get()                 V
get(long, TimeUnit)   V
isCancelled()         boolean
isDone()              boolean
resultNow()           V
```

A proxy that represents the result of an asynchronous computation

See 20/docs/api/java.base/java/util/concurrent/Future.html

Applying Java Futures to Address These Limitations

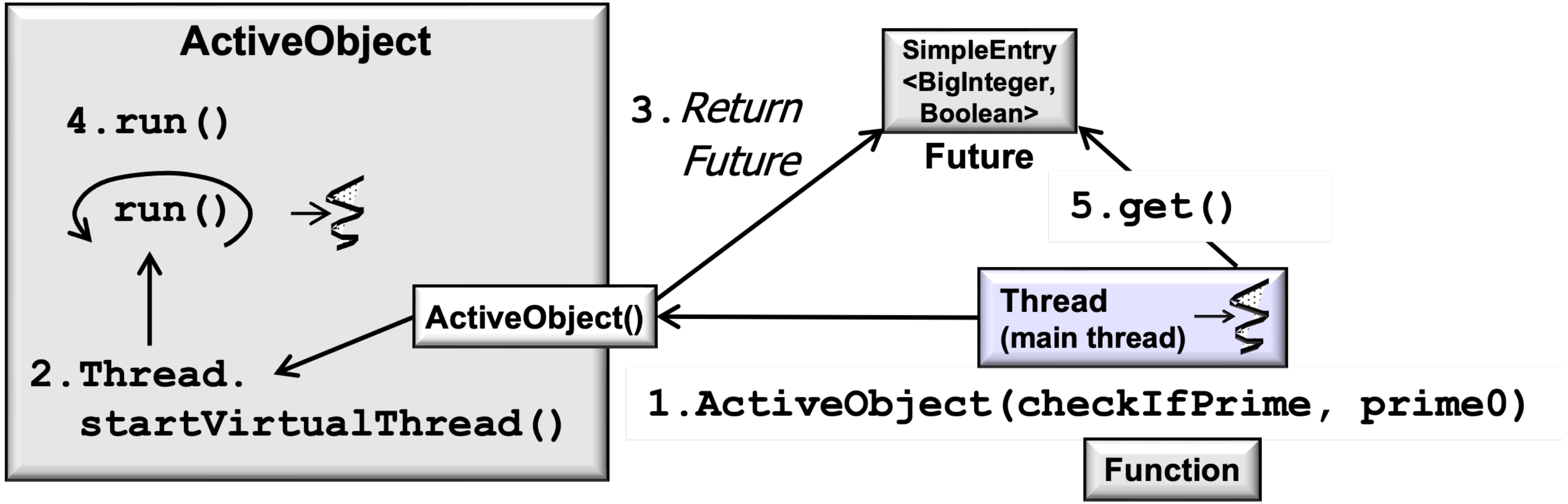
- We address these limitations in 2 ways
  - Apply the *Active Object* pattern to create generic concurrent objects
  - Apply the *Future* pattern & Java Future interface to avoid indefinite blocking
    - The *Future* pattern
    - The Future interface



The ActiveObject class implements the Java Future interface, so a caller can obtain its results without blocking indefinitely

Applying Java Futures to Address These Limitations

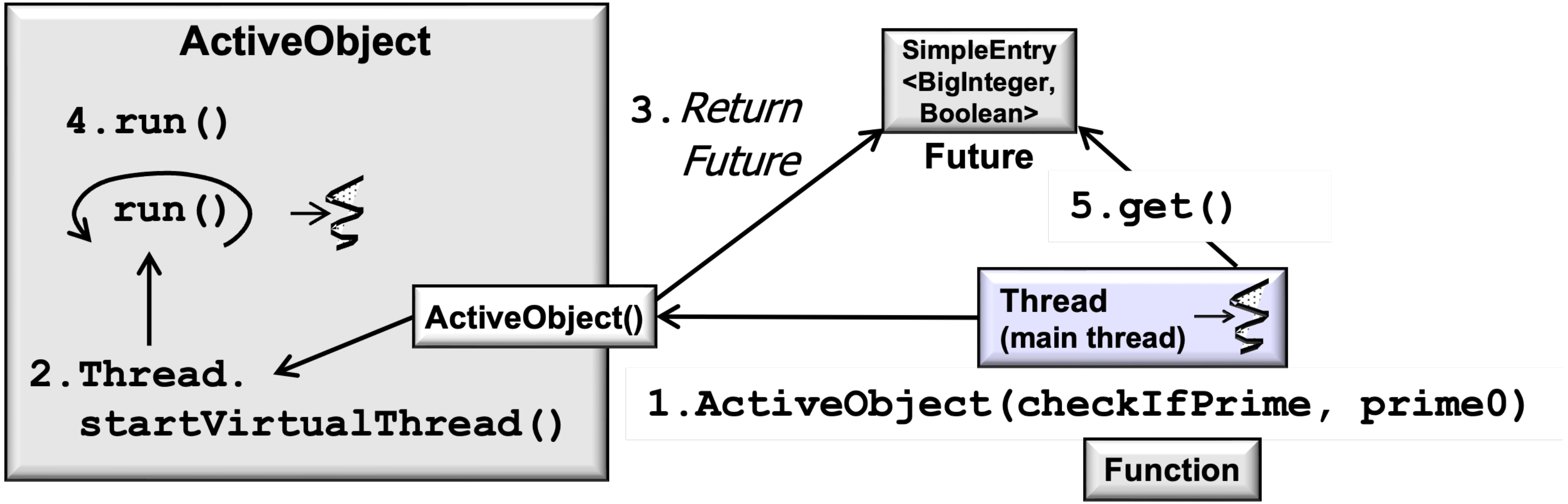
- We demonstrate the *Active Object* & *Future* patterns in conjunction with the Java Future interface in an upcoming case study



See upcoming lesson on "*Applying Java Futures in Case Study ex16*"

Applying Java Futures to Address These Limitations

- We demonstrate the *Active Object* & *Future* patterns in conjunction with the Java Future interface in an upcoming case study



  - This case study generalizes case study ex6 that checked the primality of BigInteger objects when computing RSA public & private keys

See earlier lessons on "*Implementing Closures with Java Lambda Expressions*"

End of Motivating the Need for Java Futures