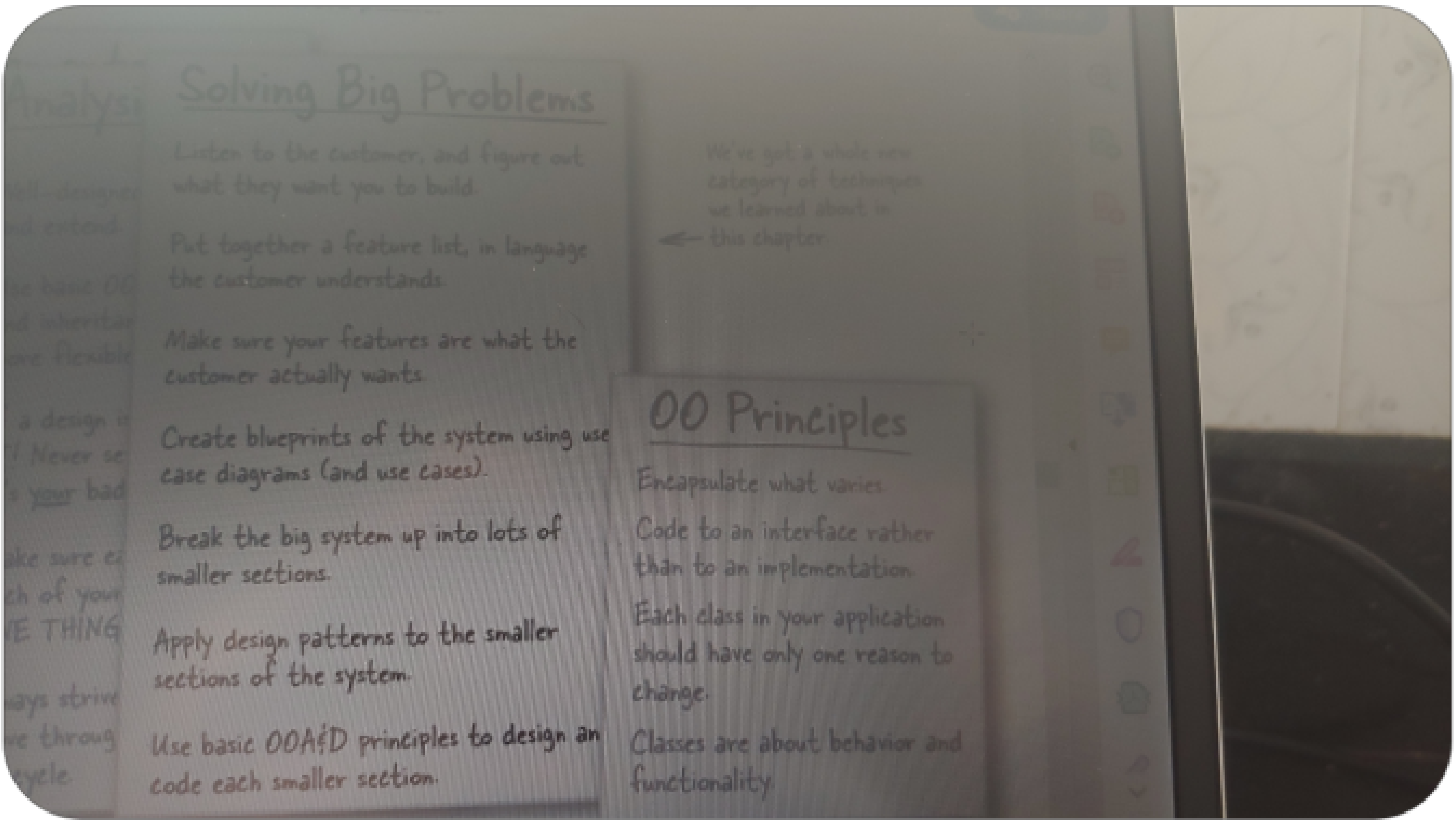
just means that we are describing a problem using terms the customer will understand

- Breaking system into different modules
- Apply some design pattern to architect the system like MVC

Domain analysis -> Features -> Use case diagram -> Requirements -> Use Cases -> Modules -> apply design pattern like MVC



Solving Big Problems

Listen to the customer, and figure out what they want you to build.

Put together a feature list, in language the customer understands.

Make sure your features are what the customer actually wants.

Create blueprints of the system using use case diagrams (and use cases).

Break the big system up into lots of smaller sections.

Apply design patterns to the smaller sections of the system.

Use basic OOA&D principles to design an code each smaller section.

We've got a whole new category of techniques we learned about in this chapter.

OO Principles

Encapsulate what varies.

Code to an interface rather than to an implementation.

Each class in your application should have only one reason to change.

Classes are about behavior and functionality.

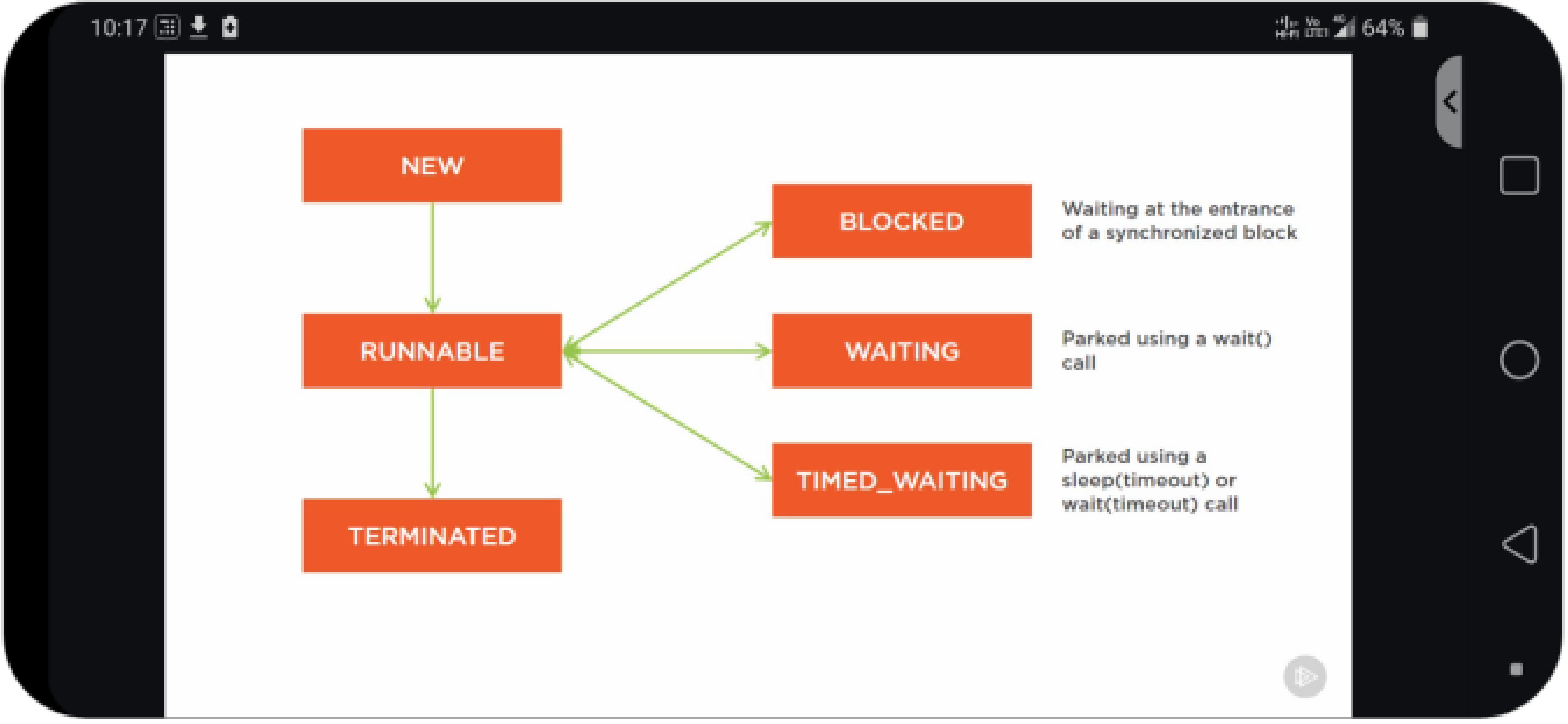
==============================

Java Multithreading and Concurrency

- ○ Single core CPU and multicore CPU
- ○ Thread - help to devide specific opeartion within a single application across multiple threads which can run at same time.
- ○ Threads can be started by usung runnable interface.
- ○ Runnable r = () -> {sysout(Thread.currentThread().getName())}. Thread t = new Thread(r). t.start()
- ○ To stop a thread its not recommended to use its stop method.we should use interrupt method. This will send stop signal to code which is running by thread's run method. Runnable r = () -> {while(! Thread.currentThread().isInterupted()){sysout(...)}}.
- ○ Thread schedulers to schedule

CPU time for each threads

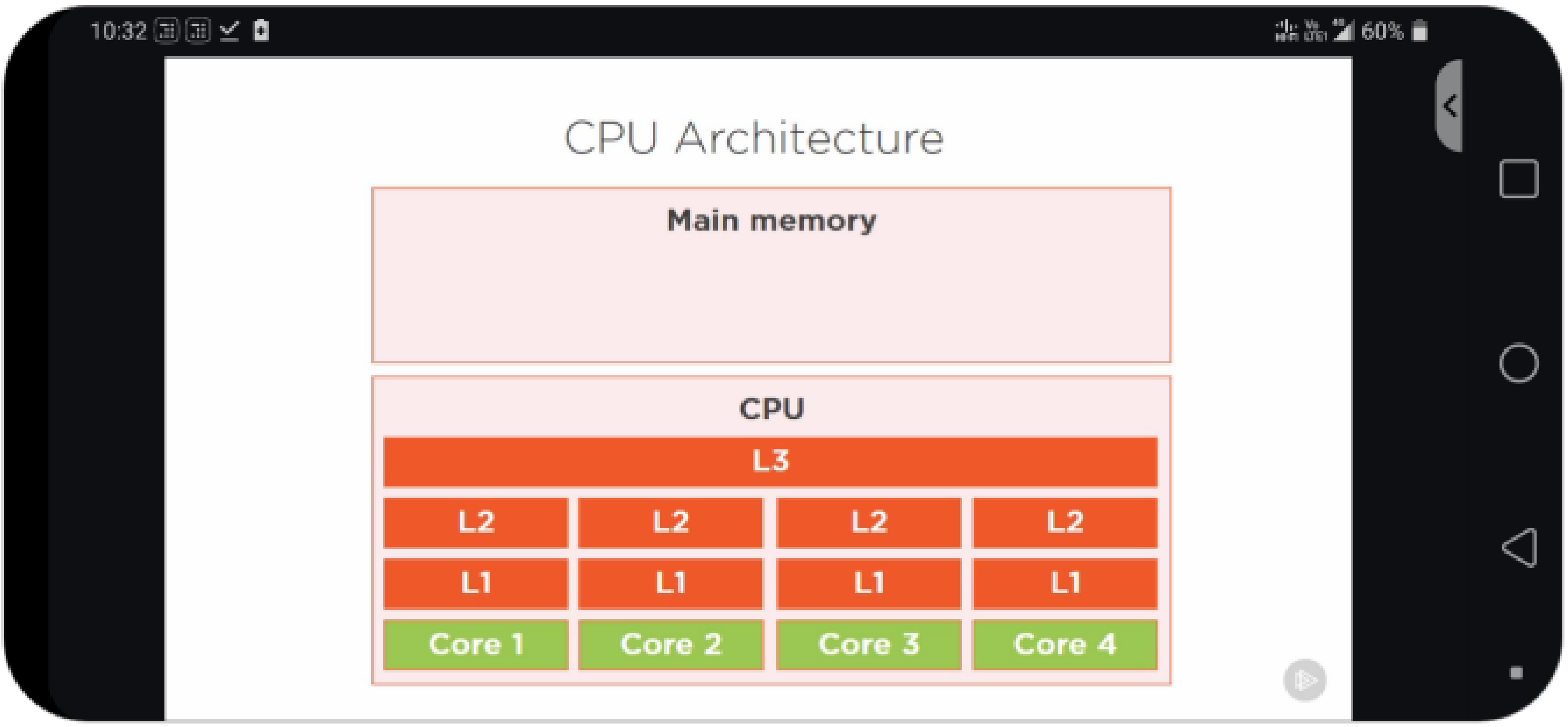
- ◯ Race Conditions - two or more threads sharing common data which results in wrong data in the system. To fix this we can use synchronization
- ◯ Synchronization uses class or specific instance locks and it can be either at method level or block level
- ◯ Reentrate locks - thread with s specific lock can enter any methods which needs the same lock
- ◯ Deadlock - dead situation where each threads are waiting forever for locks which are owned by other thread
- ◯ Producer Consumer pattern and fixing it by using synchronization and wait/notify pattern
- ◯ Thread States


NEW
RUNNABLE
TERMINATED
BLOCKED
Waiting at the entrance of a synchronized block
WAITING
Parked using a wait() call
TIMED_WAITING
Parked using a sleep(timeout) or wait(timeout) call

Synchronization

- Protects block of code. Guarantees this code will be executed by one thread at a time.prevents race condition.
- 20 years ago when CPU had no cache all the variable values were read from main memory. But now a days CPU do have cache, from where all the variable values will be read.
- CPU architecture. It has multi core processors where each processor will have ita own L1 and L2 cache layer and then a common L3 cache layer. Access

to L1 cache is 20 time and access to L2 cache is 15 times faster than the access to main memory. But size of L1 cache is 32kb and L2 cache is 256kb but main memory can have size in several GB.

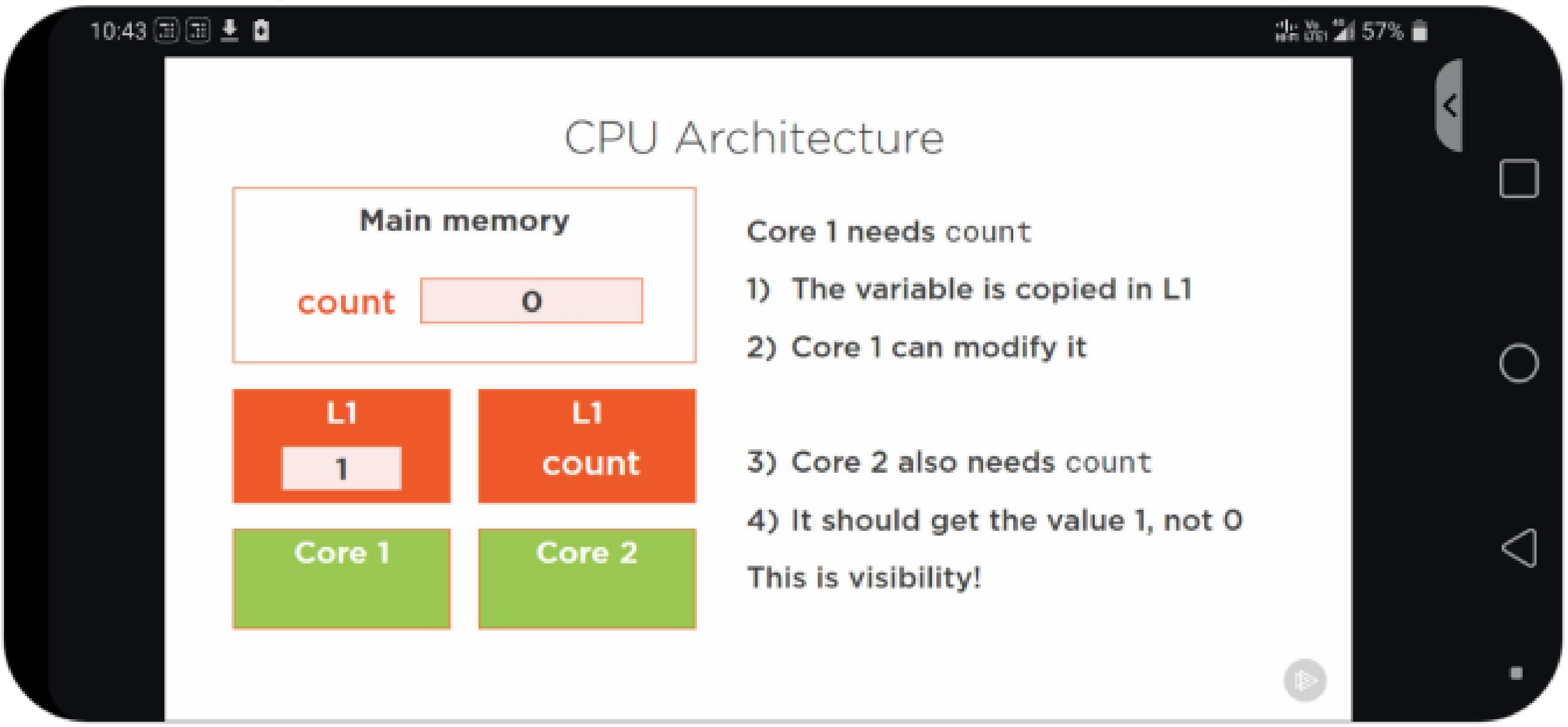

CPU Architecture

Visibility

≡ ○ In case of multi core processor, writing some data into one of the core's L1 cache will be more faster compared to syncing the same value with main memory. Then the other core should read this data from core1 cache memory instead of main memory.

Visibility is caclscading this updated values to other core processors . so a variable is said visible if writes made on it are visible to all other cores. All the synchronized writes are made visible by default.

- This visibility problem exists only in multi core processors
- Visibility can be achieved by synchronization and volatility



CPU Architecture

Main memory

count 0

L1	L1
1	count
Core 1	Core 2

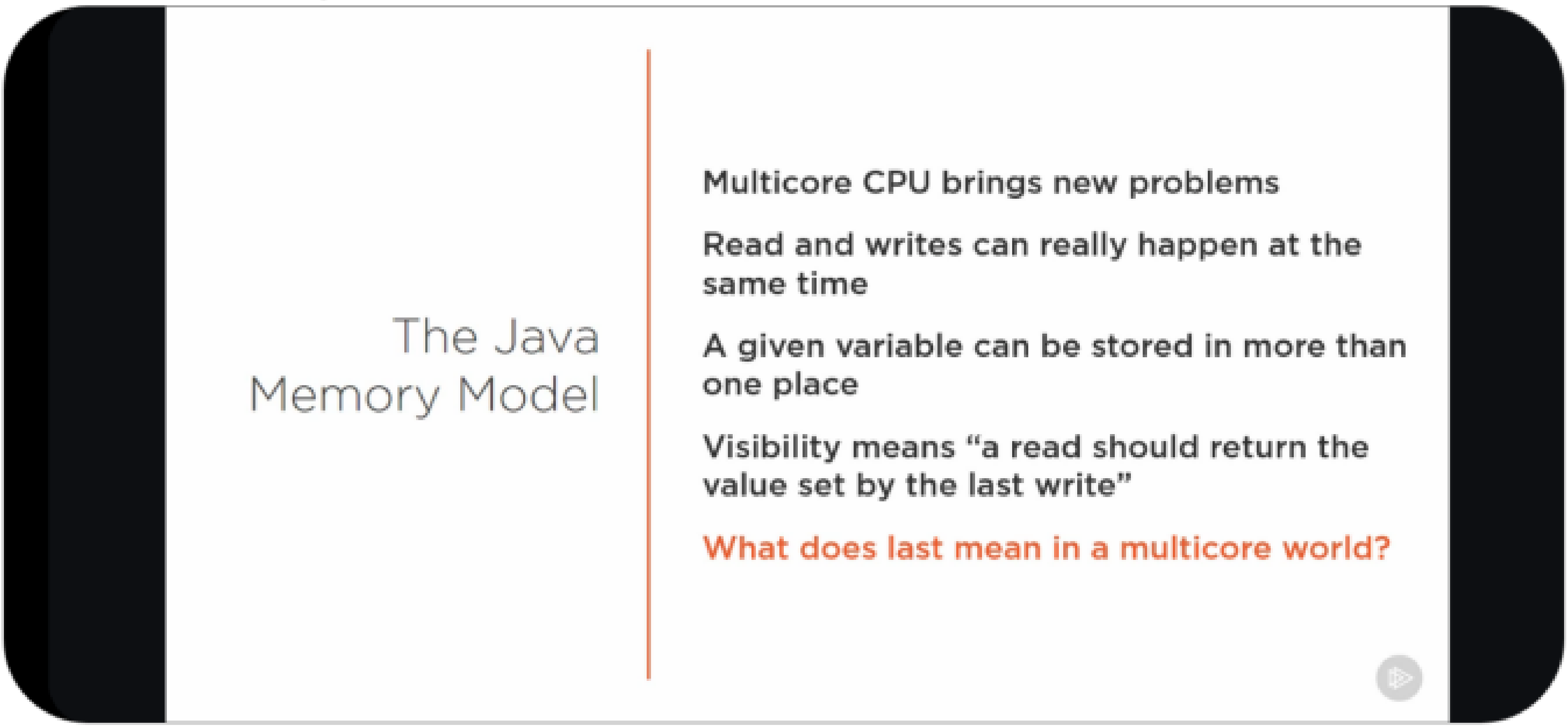
Core 1 needs count

1) The variable is copied in L1
2) Core 1 can modify it
3) Core 2 also needs count
4) It should get the value 1, not 0

This is visibility!

Happens Before link

- Happens before link will help to read and write data values properly. This link must exists between reading and writing values. This link can be created through synchronized or volatile keyword.



The Java Memory Model

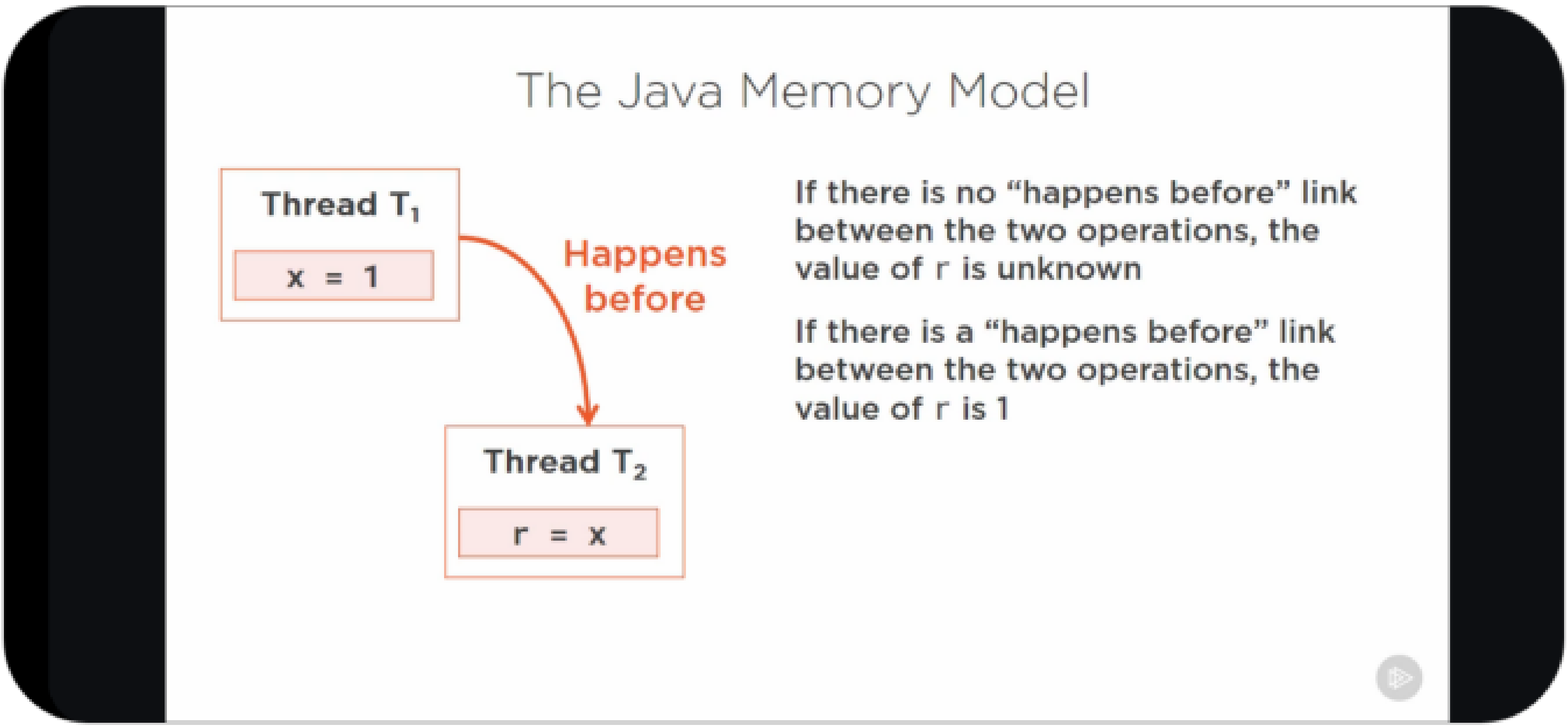
Multicore CPU brings new problems

Read and writes can really happen at the same time

A given variable can be stored in more than one place

Visibility means "a read should return the value set by the last write"

What does last mean in a multicore world?



The Java Memory Model

Thread T_1

x = 1

Happens before

Thread T_2

r = x

If there is no "happens before" link between the two operations, the value of r is unknown

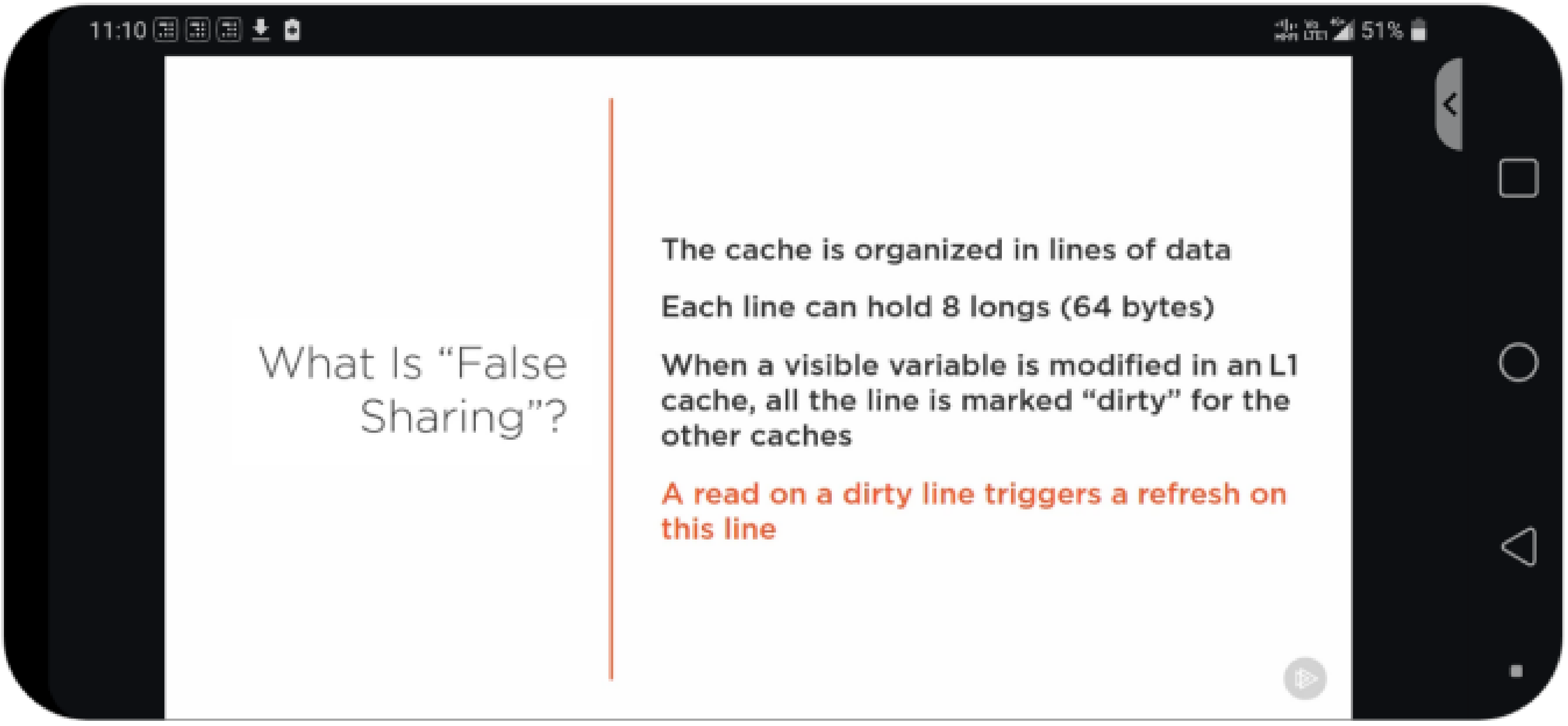
If there is a "happens before" link between the two operations, the value of r is 1

- ○ By default this link exists between all synchronized or volatile read or write operations.
- ○ All shared variables(shared between multiple threads) should

be accessed in a synchronized or volatile way. Else we will have eace condition and bugs in the code.

False sharing:

- ○ False sharing can cause serious performance issues where data value can be read from main memory because of dirty line refresh



What Is "False Sharing"?

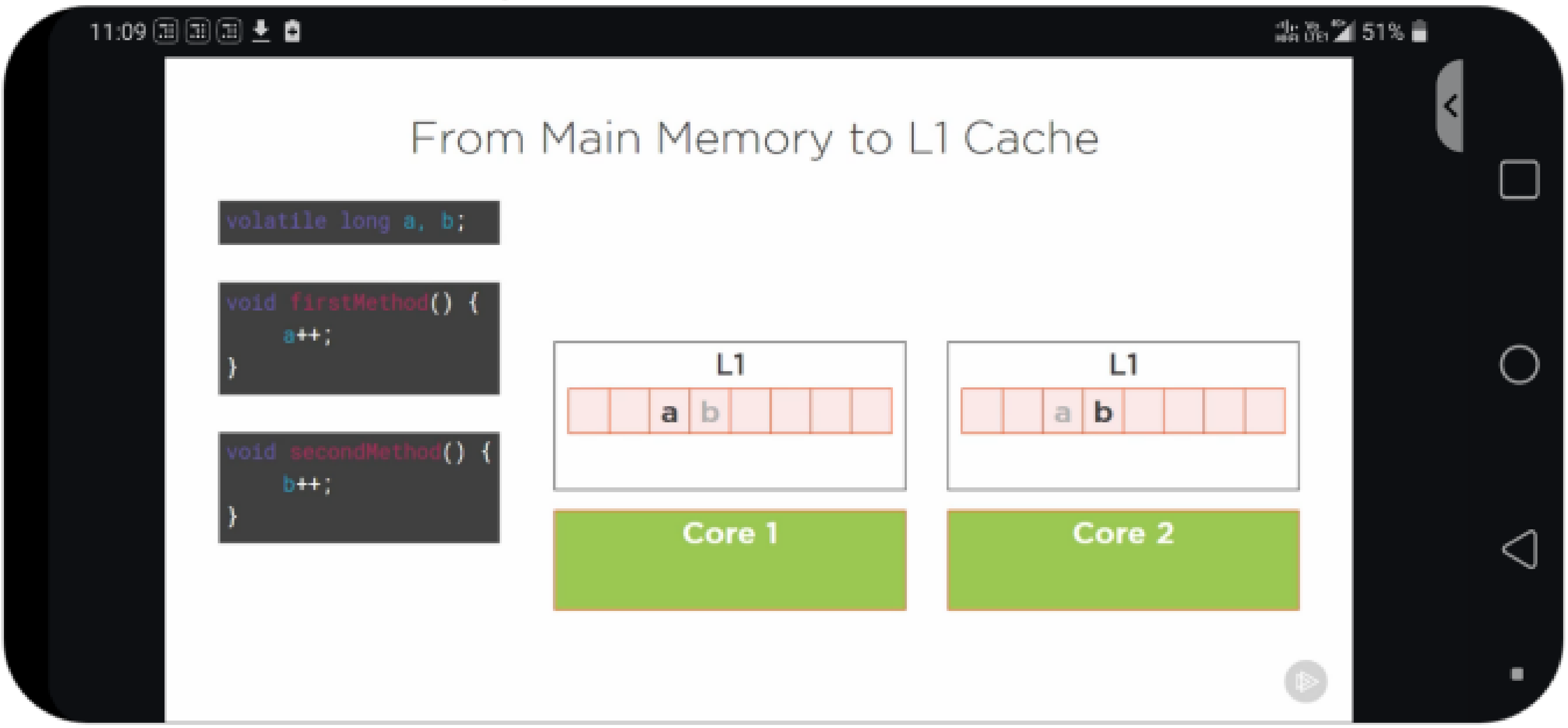
The cache is organized in lines of data

Each line can hold 8 longs (64 bytes)

When a visible variable is modified in an L1 cache, all the line is marked "dirty" for the other caches

A read on a dirty line triggers a refresh on this line

- ○ First thread loads a value from main memory and second thread loads b value. Because of the way memory and compiler organized, both a and b are written in 2 contiguous area of main memory hence both threads will copy a and b line of cache from main memory.



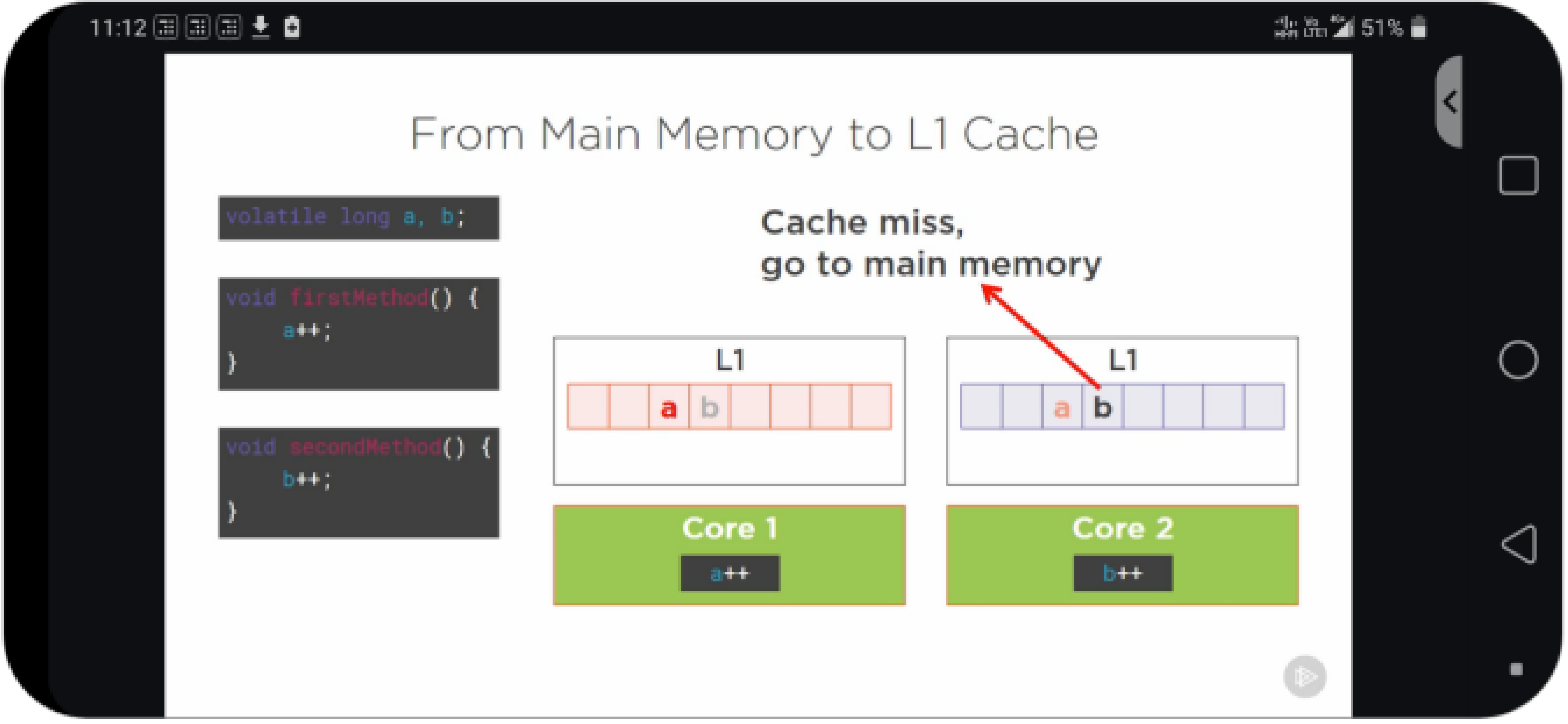
From Main Memory to L1 Cache

```
volatile long a, b;
```

```
void firstMethod() {
    a++;
}
```

```
void secondMethod() {
    b++;
}
```

When thread 1 increaments the value of a it marks the entire line of cache as dirty which also brodcasted to other caches. So when thread 2 tries to increment b since the line of cache is dirty it will fetch the latest value from main memory even though first thread didn't incrrment the value b which can cause serious performance problem because of unnecessary call to main memory.



From Main Memory to L1 Cache

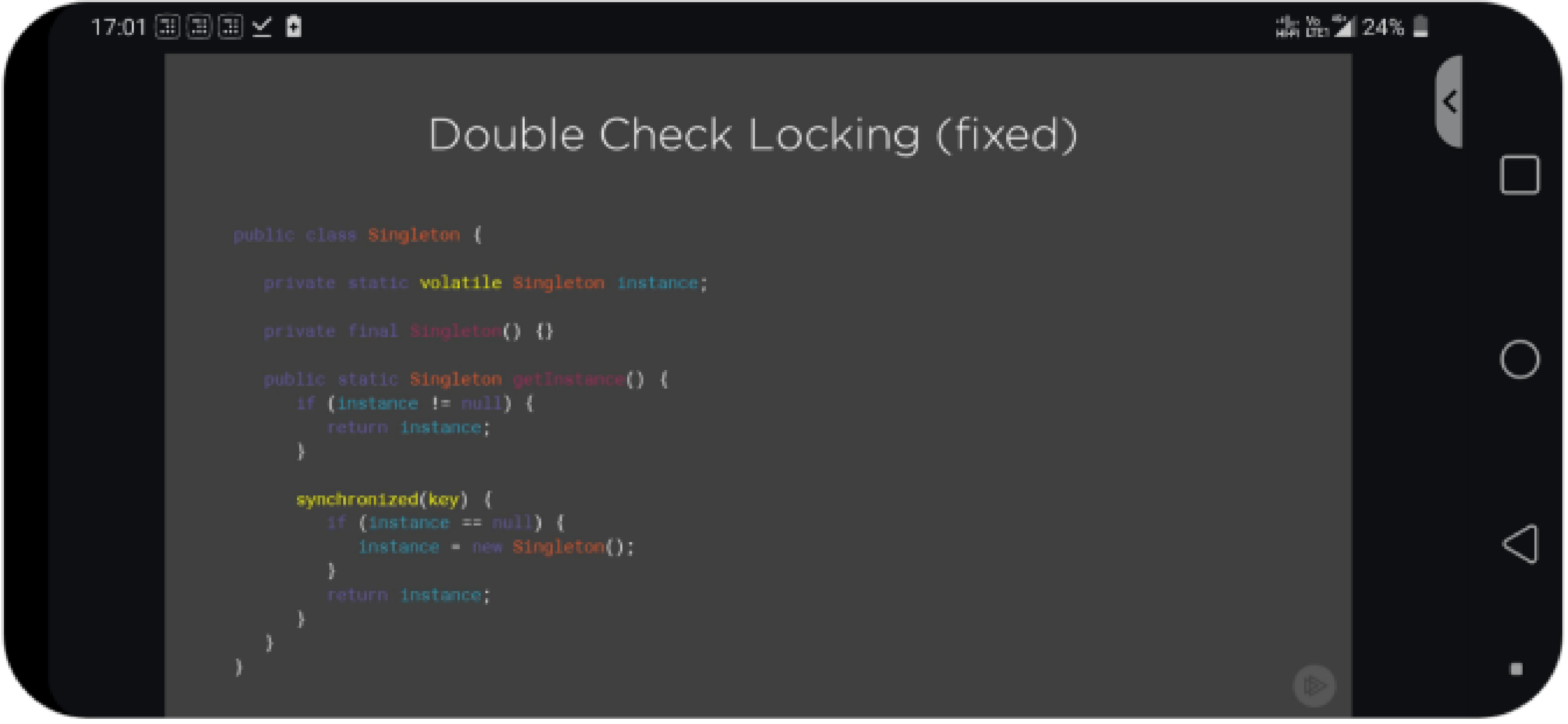
```
volatile long a, b;
```

```
void firstMethod() {
    a++;
}
```

```
void secondMethod() {
    b++;
}
```

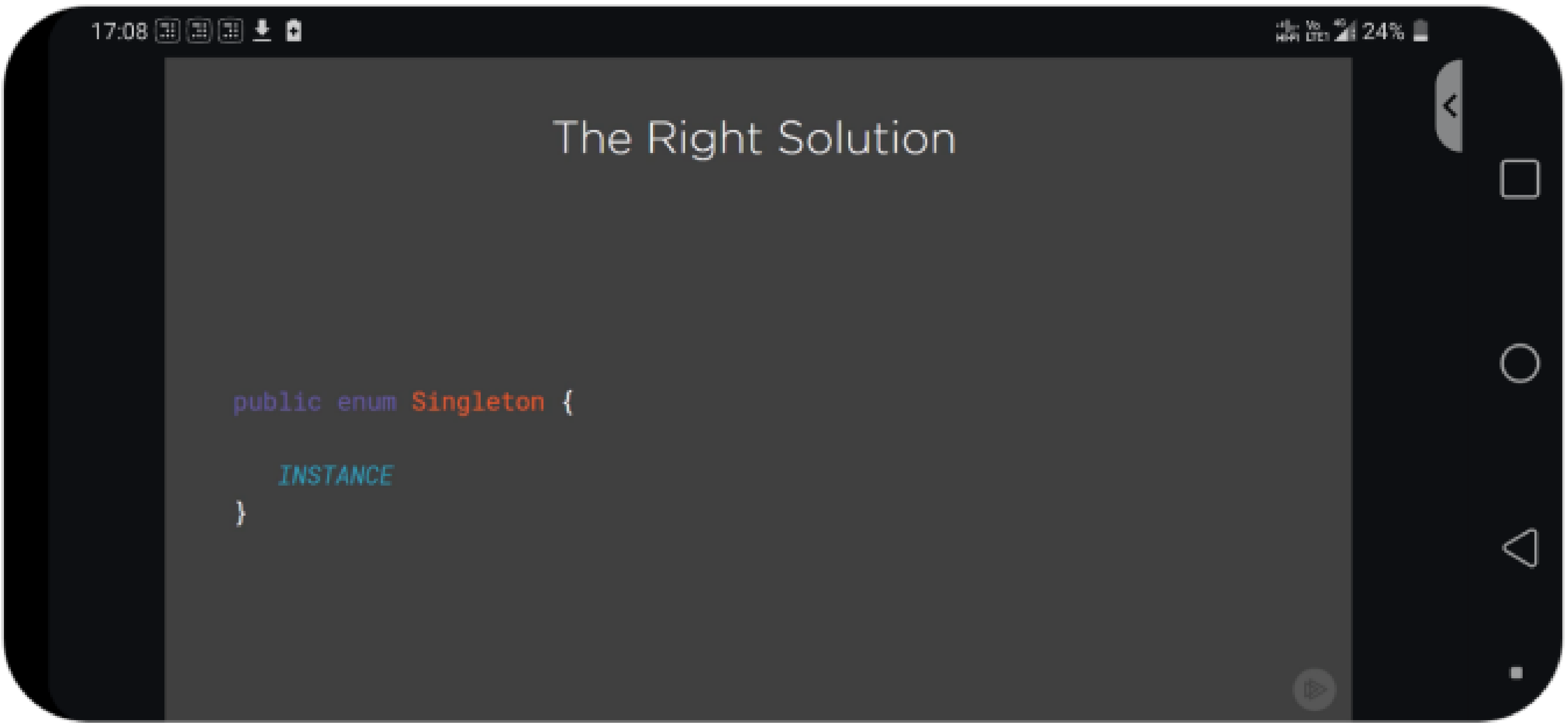
Singleton pattern with double check locking:

≡ ○ We make the instance field as volatile so there won't be visibility issues in reading this instance in multicore CPU and writing happens inside synchronization. Constructor should be final to prevent object extension. But there will be performance issues because of this synchronization logic.



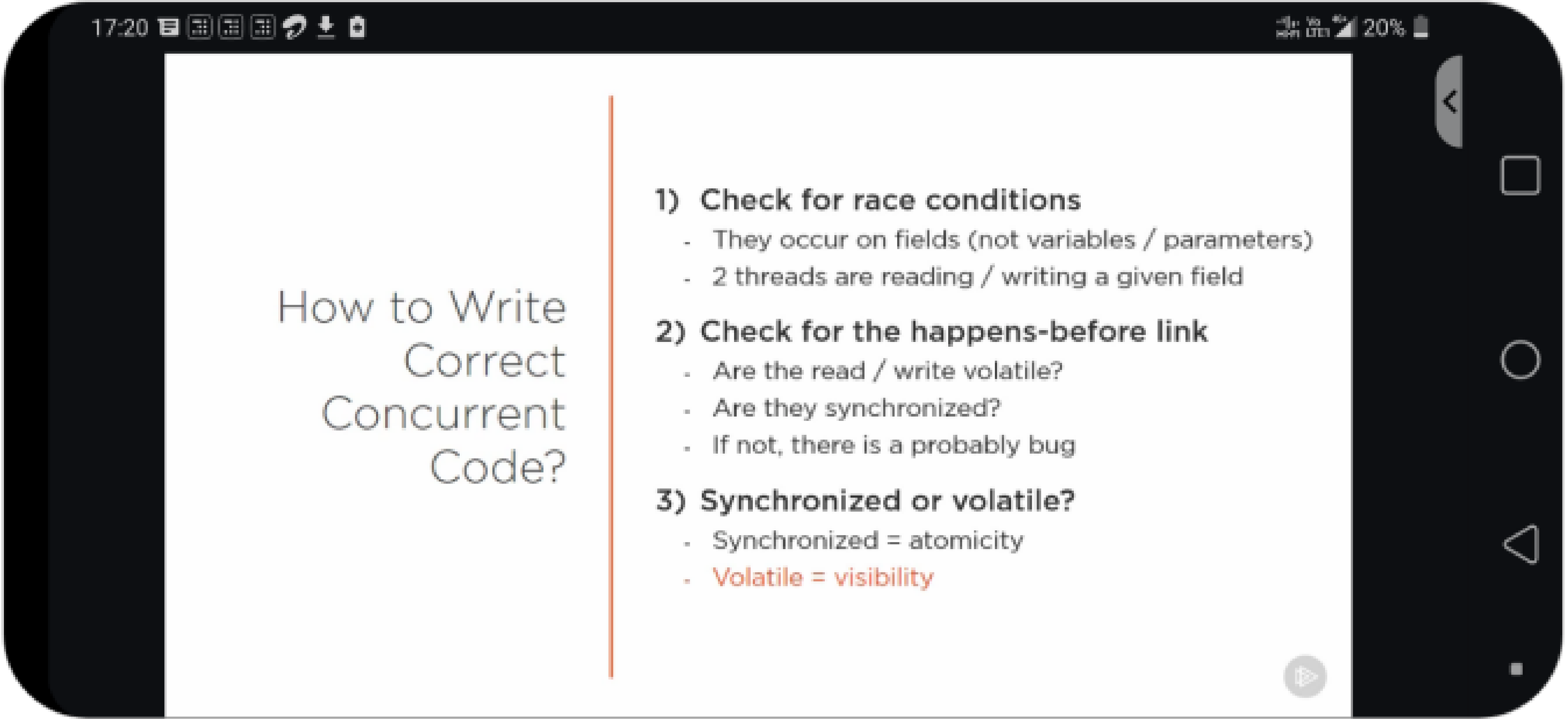
Double Check Locking (fixed)

```
public class Singleton {

   private static volatile Singleton instance;

   private final Singleton() {}

   public static Singleton getInstance() {
      if (instance != null) {
         return instance;
      }

      synchronized(key) {
         if (instance == null) {
            instance = new Singleton();
         }
         return instance;
      }
   }
}
```



The Right Solution

```
public enum Singleton {

   INSTANCE
}
```


How to Write Correct Concurrent Code?

1) Check for race conditions
   - They occur on fields (not variables / parameters)
   - 2 threads are reading / writing a given field
2) Check for the happens-before link
   - Are the read / write volatile?
   - Are they synchronized?
   - If not, there is a probably bug
3) Synchronized or volatile?
   - Synchronized = atomicity
   - Volatile = visibility

==============================

Advanced concepts

≡ ○ Concurrent or parallel programming is about running multiple threads

≡ ○ Task is some functionalities which may or may not take some input and may or may not produce any output.

≡ ○ Synchronous programming: thread will be block until the synchronous call is executed. The called functionality will be executed right away

≡ ○ Asynchronous programming: thread may be blocked or may not be blocked. Called task will be executer sometime in the future.

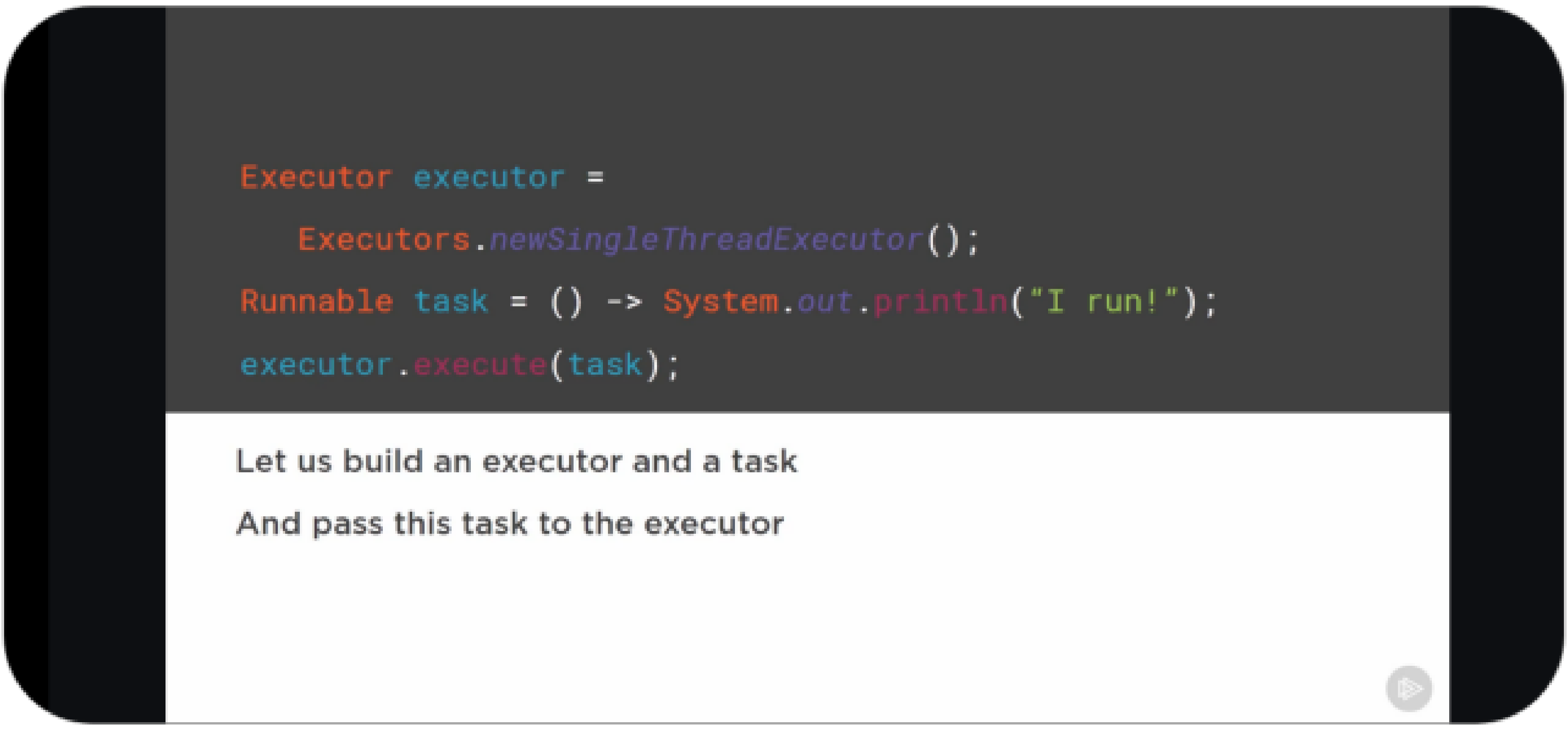
- ≡ ○ In concurrent programming its always good to call functionalities asynchronously without blocking current thread.
- ≡ ○ Creating threads by using Runnable interface is not the best approach in java, as it allows users to create hundreds of threads and creating and stopping each thread will cause performance issues.

Executor interface

- ≡ ○ Instead of creating and stopping threads, it is better to create pool of threads and then create a task and provide these tasks to pool of threads. One of the thread from this pool takes the task and executes it.
- ≡ ○ By creating an instance of an executor interface we can create

such pool of threads in java. This pool of thread will be available until the executor or executor service instance is availablen

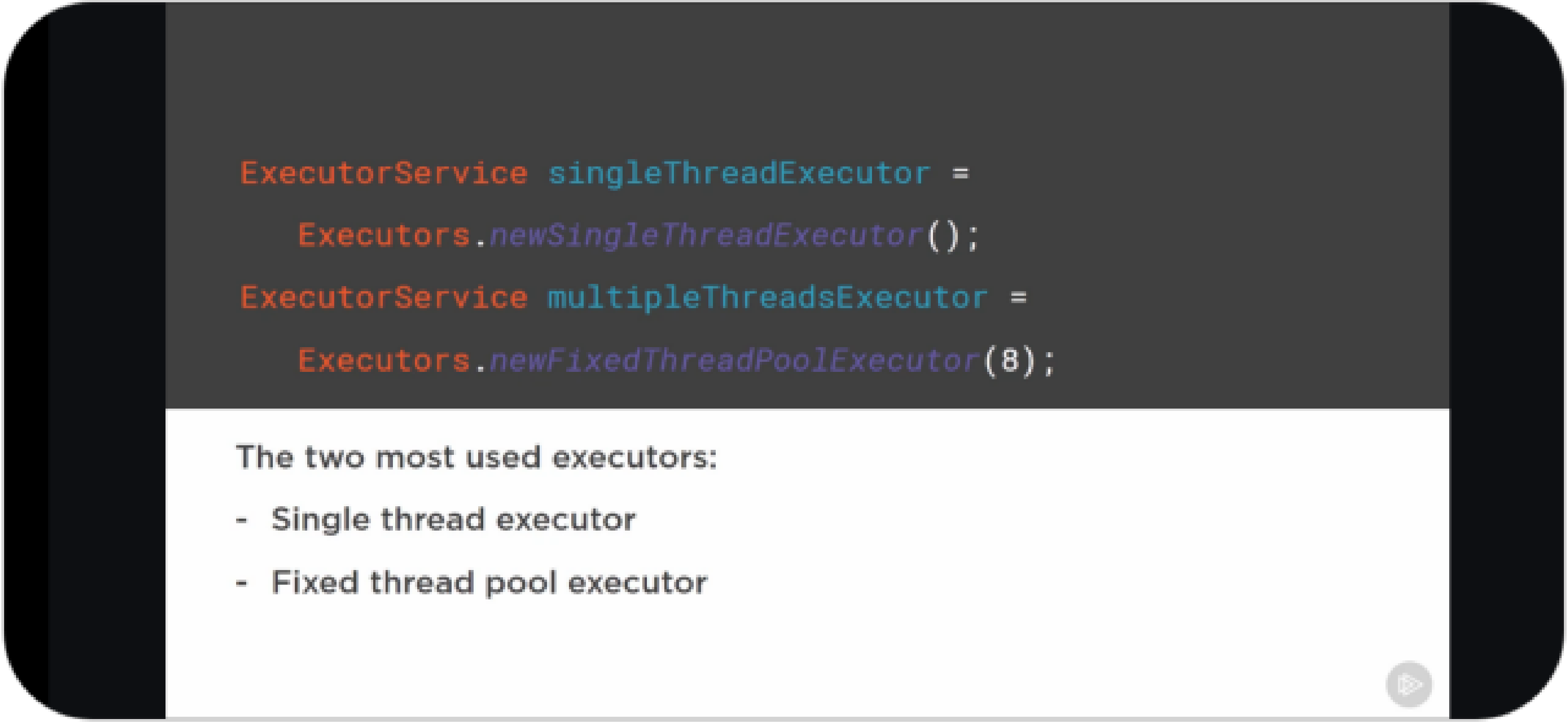
- ○ public interface executor { void execute(Runnable task) }
- ○ Jdk by default provides many implemented classes of this interface. We can uses methods of 'Executors' factory class to create executor instance.



```
Executor executor =
   Executors.newSingleThreadExecutor();
Runnable task = () -> System.out.println("I run!");
executor.execute(task);
```

Let us build an executor and a task

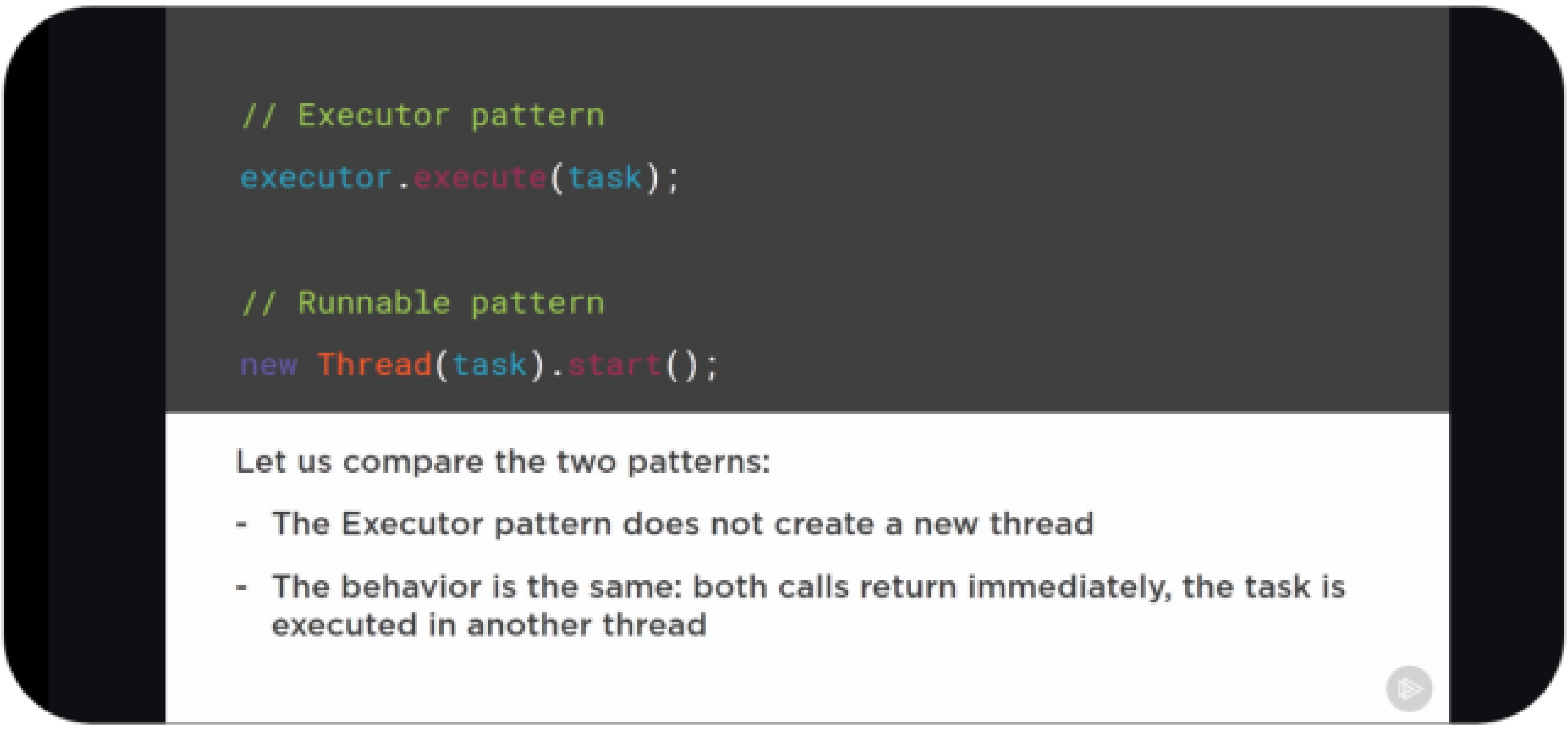
And pass this task to the executor



```
ExecutorService singleThreadExecutor =
    Executors.newSingleThreadExecutor();
ExecutorService multipleThreadsExecutor =
    Executors.newFixedThreadPoolExecutor(8);
```

The two most used executors:

- Single thread executor
- Fixed thread pool executor



```
// Executor pattern
executor.execute(task);

// Runnable pattern
new Thread(task).start();
```

Let us compare the two patterns:

- The Executor pattern does not create a new thread
- The behavior is the same: both calls return immediately, the task is executed in another thread

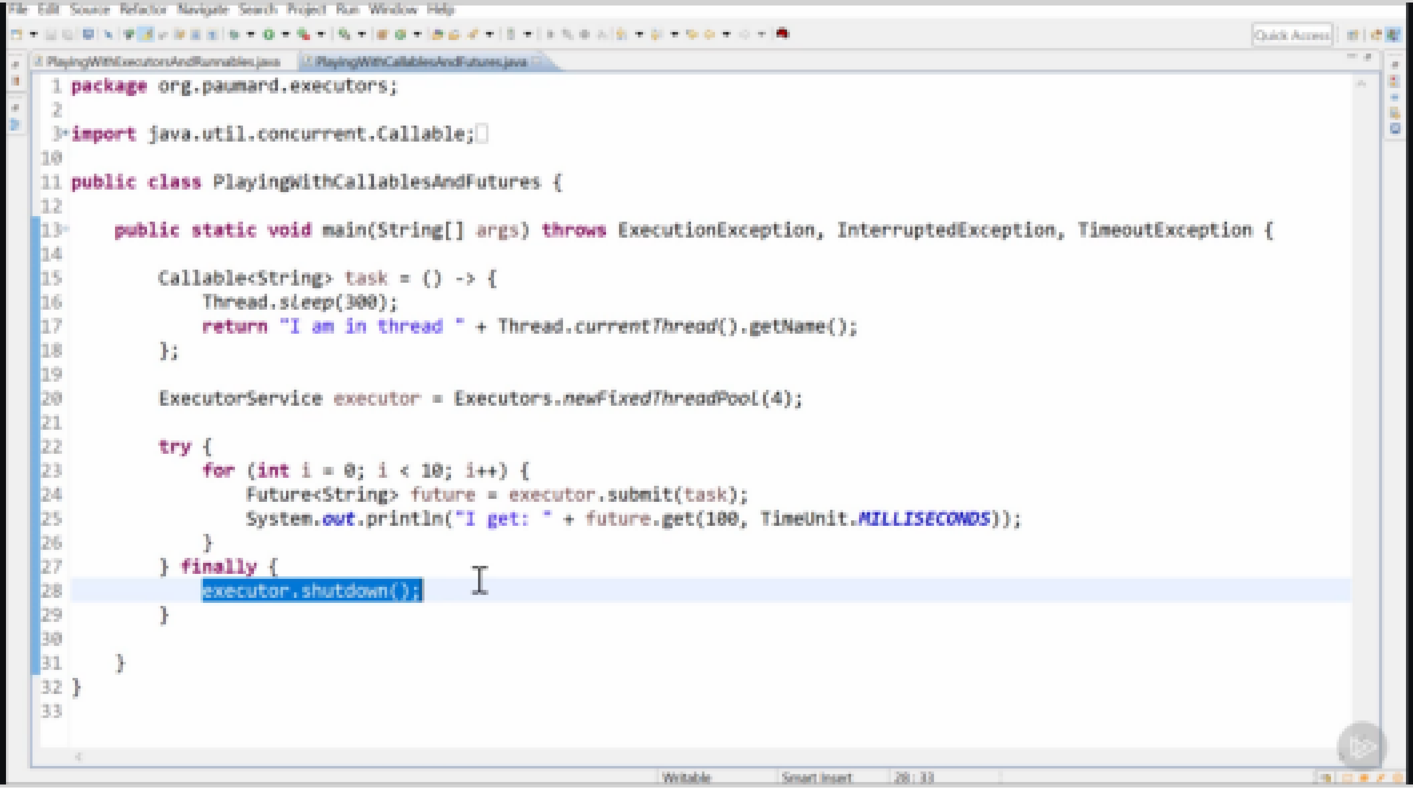
- ○ When no threads are available tasks are added to waiting queue and all tasks are executed in the order they have submitted.
- ○ We can ask executor to remove a task from waiting queue.
- ○ Runnable interfaces does not

return any object or any exception

- So how can a task return a value and how can we get the exception raised by a task or how can this value or exception to from one thread to another. The answer is Callable interface

Callable Interface

- @FunctionalInterface public interface Callable<V> { V call() throws Exception }
- The Executor interface does not have method to handle Callable. But the ExecutorService interface has a submit method which takes callable tasks. <T> Future<T> submit(Callable<T> task)
- This Future object is a wrapper for the end value or execption, and this will be returned from executor service to main thread.



```java
package org.paumard.executors;

import java.util.concurrent.Callable;

public class PlayingWithCallablesAndFutures {

    public static void main(String[] args) throws ExecutionException, InterruptedException, TimeoutException {

        Callable<String> task = () -> {
            Thread.sleep(300);
            return "I am in thread " + Thread.currentThread().getName();
        };

        ExecutorService executor = Executors.newFixedThreadPool(4);

        try {
            for (int i = 0; i < 10; i++) {
                Future<String> future = executor.submit(task);
                System.out.println("I get: " + future.get(100, TimeUnit.MILLISECONDS));
            }
        } finally {
            executor.shutdown();
        }

    }
}
```

PlayingWithExecutorsAndRunnables.java
PlayingWithCallablesAndFutures.java
Writable
Smart Insert
28 : 33

- ○ Future.get() behaviour: it returns the value immediately if the executor has one such else it will block the thread until the value is available. If there are any exception then get method also throws such exception in a ExecutionException. We can also provide timeout to avoid blocking the the thread forever.
- ○ Fixed thread pool vs cached thread pool: cached thread pool creates requested amount of threads on pool and if its not used for lets say 60s it terminates

all those 4 threads. Fixed thread pool create threads immediately and keep them alive.

- ○ Scheduled executor service: to run task in a scheduled way
- ○ Shutting down an executor service: many ways to do it. shutdown(), shutdownNow(), awaitTermination(timeout)

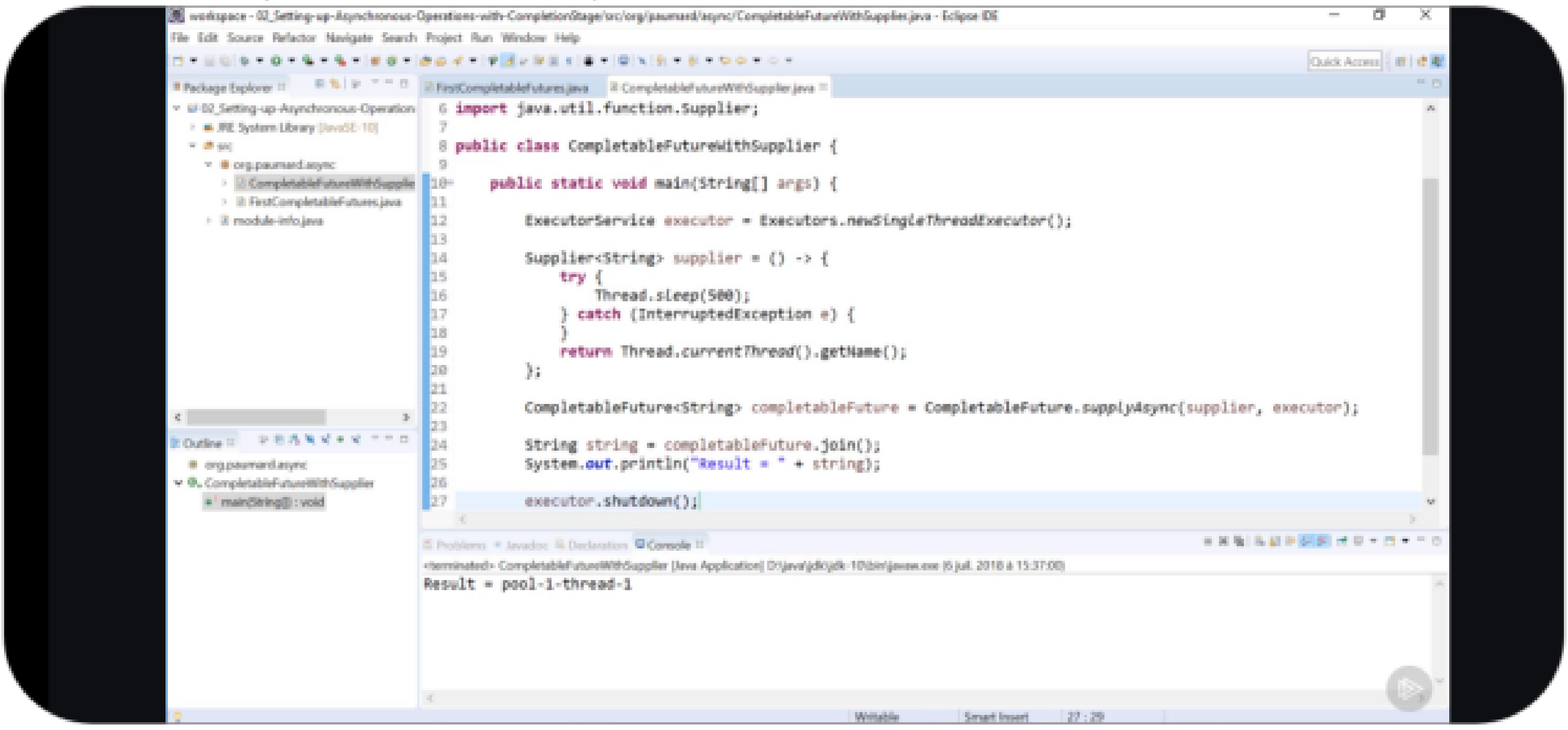
Java functional Interfaces

- ○ Supplier: Java supplier function which takes nothing and return something. Data supply().
- ○ Consumer: java consumer function which takes something and return nothing. Void accept(data)
- ○ Function: takes something and return something. Data apply(data)
- ○ Runnable: takes nothing and returns nothing. Void run()

- Callable: similar to supplier but can handle exceptions as well. Data call() throws exception.

CompletableFuture:

- Similar to java Future object but provides additional methods to handle tasks asynchronously
- We can create comoletablefuture either from runnable or supplier
- CompletableFuture.runAsync(Runnable) or CompletableaFuture.supplyAsync(Supplier)

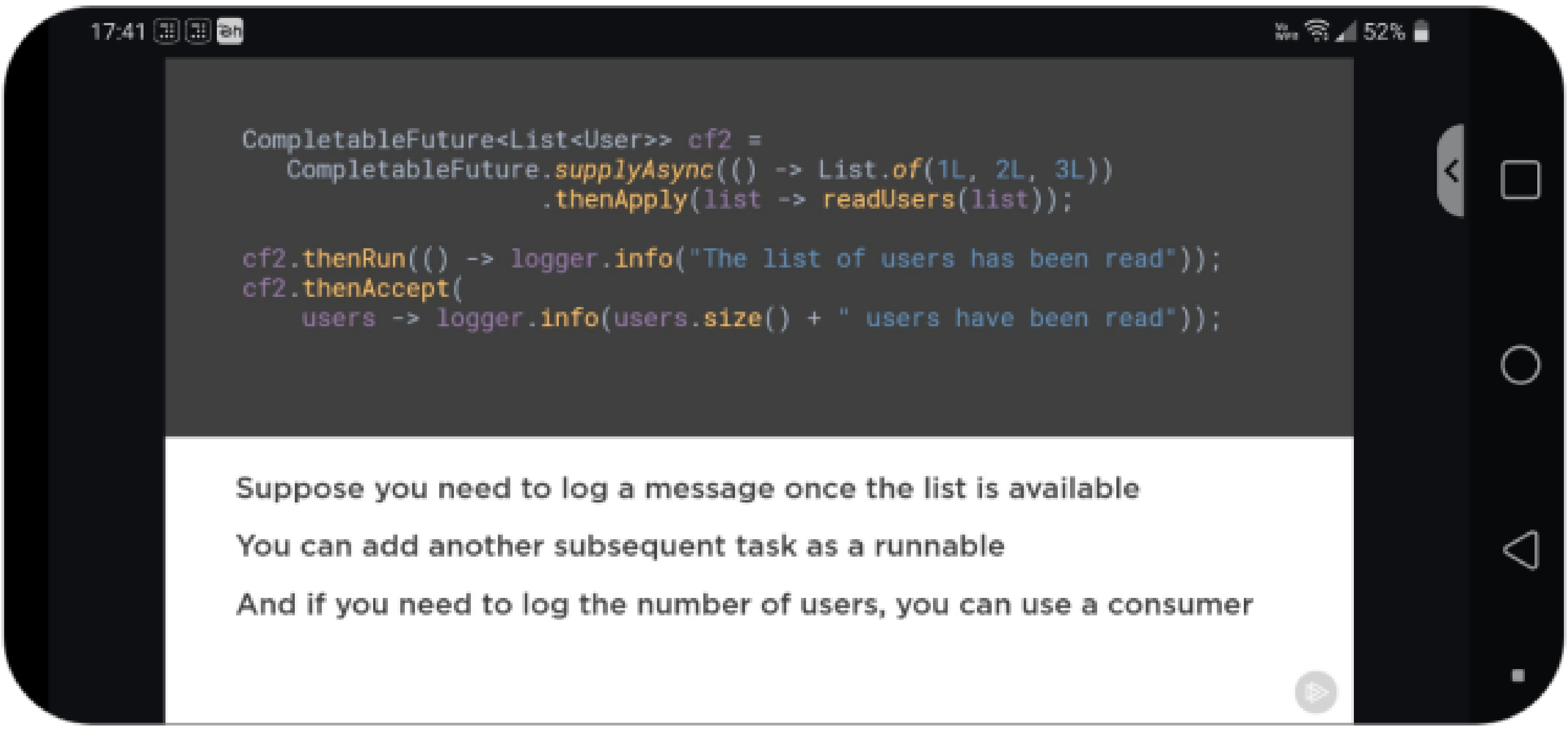


```java
import java.util.function.Supplier;

public class CompletableFutureWithSupplier {

    public static void main(String[] args) {

        ExecutorService executor = Executors.newSingleThreadExecutor();

        Supplier<String> supplier = () -> {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
            }
            return Thread.currentThread().getName();
        };

        CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(supplier, executor);

        String string = completableFuture.join();
        System.out.println("Result = " + string);

        executor.shutdown();
```

```
Result = pool-1-thread-1
```

Requirement:

- We need to first get list of user id's and then for each user get the corresponding user object and map these user id's to user list and finally reduce this entire users to list. First functionality is to use map and then reduce

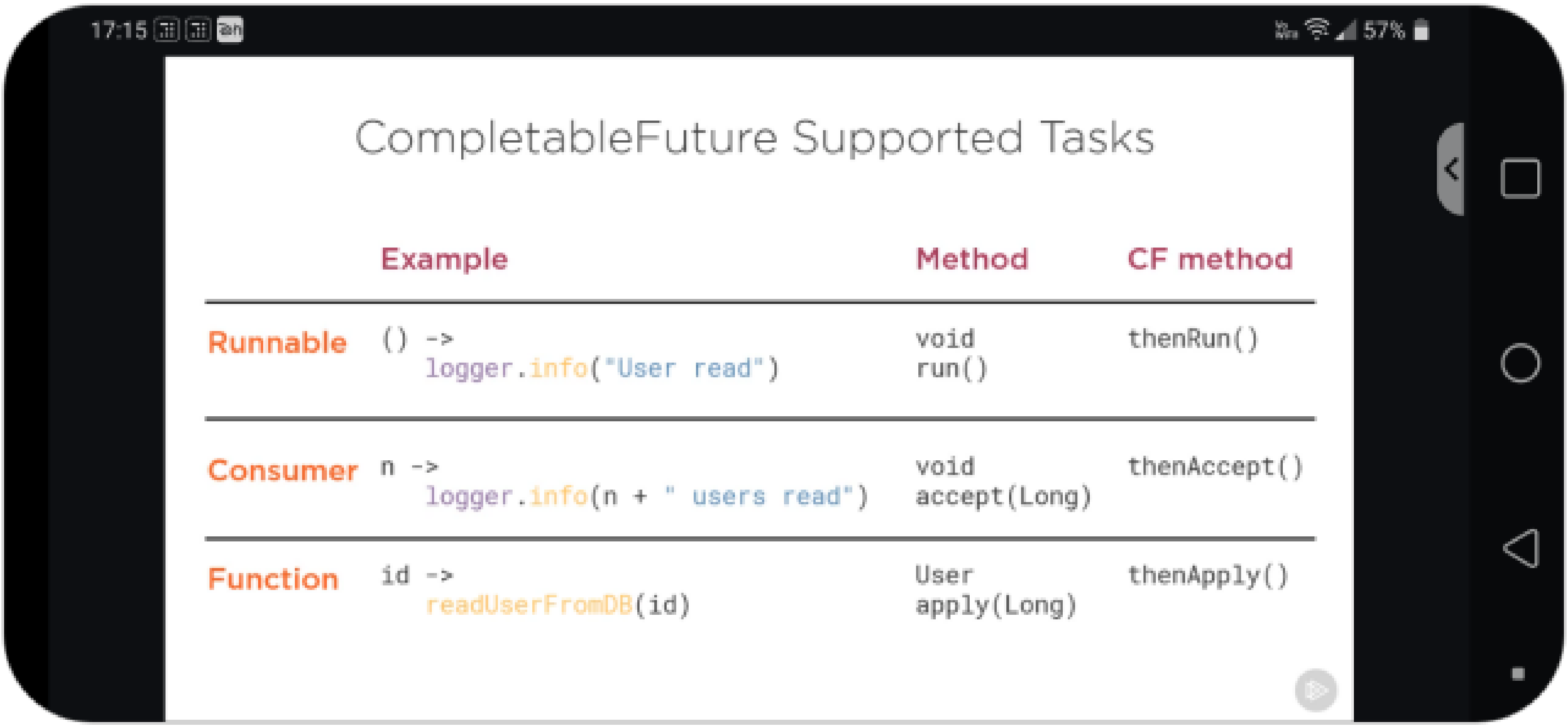


```java
CompletableFuture<List<User>> cf2 =
   CompletableFuture.supplyAsync(() -> List.of(1L, 2L, 3L))
                    .thenApply(list -> readUsers(list));

cf2.thenRun(() -> logger.info("The list of users has been read"));
cf2.thenAccept(
    users -> logger.info(users.size() + " users have been read"));
```

Suppose you need to log a message once the list is available

You can add another subsequent task as a runnable

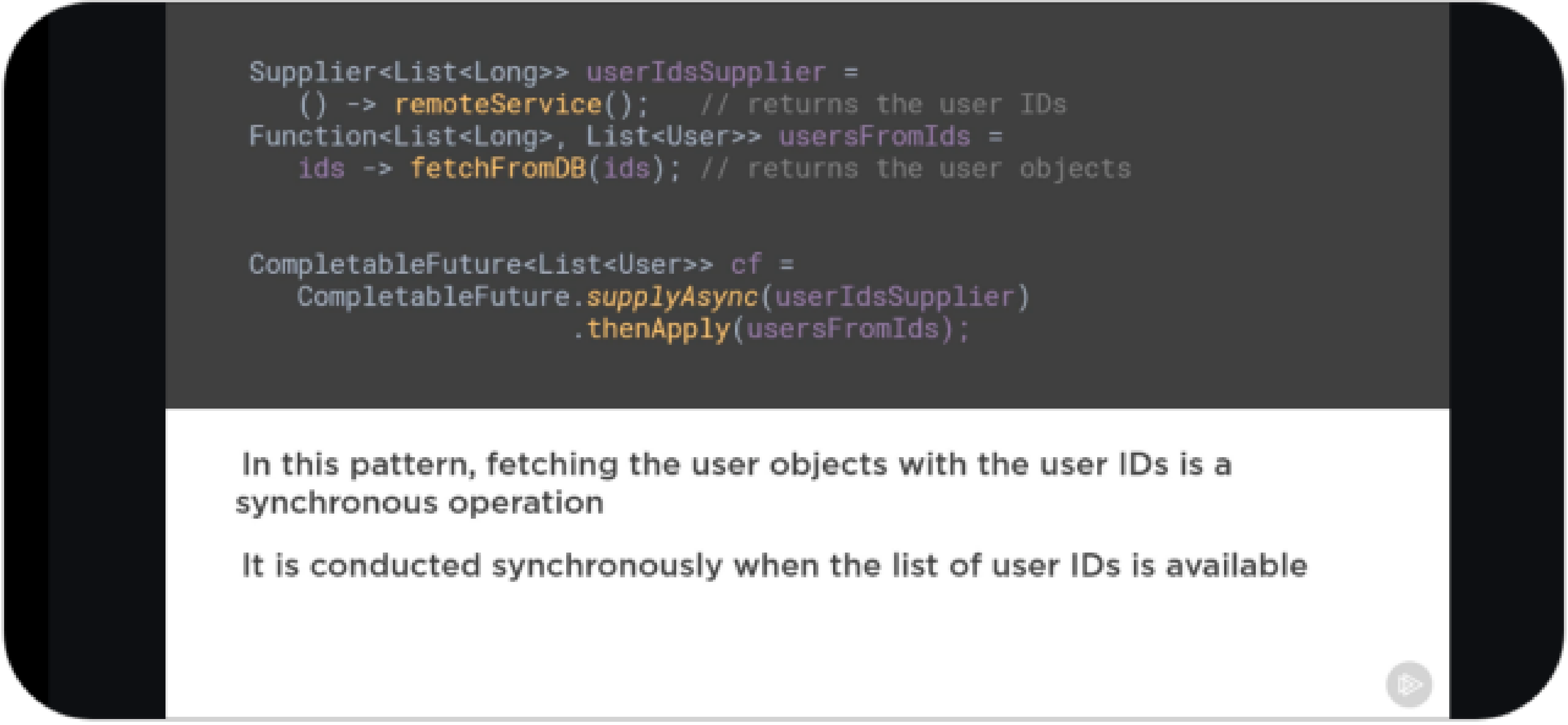
And if you need to log the number of users, you can use a consumer



CompletableFuture Supported Tasks

	Example	Method	CF method
Runnable	`() -> logger.info("User read")`	`void run()`	`thenRun()`
Consumer	`n -> logger.info(n + " users read")`	`void accept(Long)`	`thenAccept()`
Function	`id -> readUserFromDB(id)`	`User apply(Long)`	`thenApply()`

○ The below code gets user list synchronously once the user ids available asynchronously.

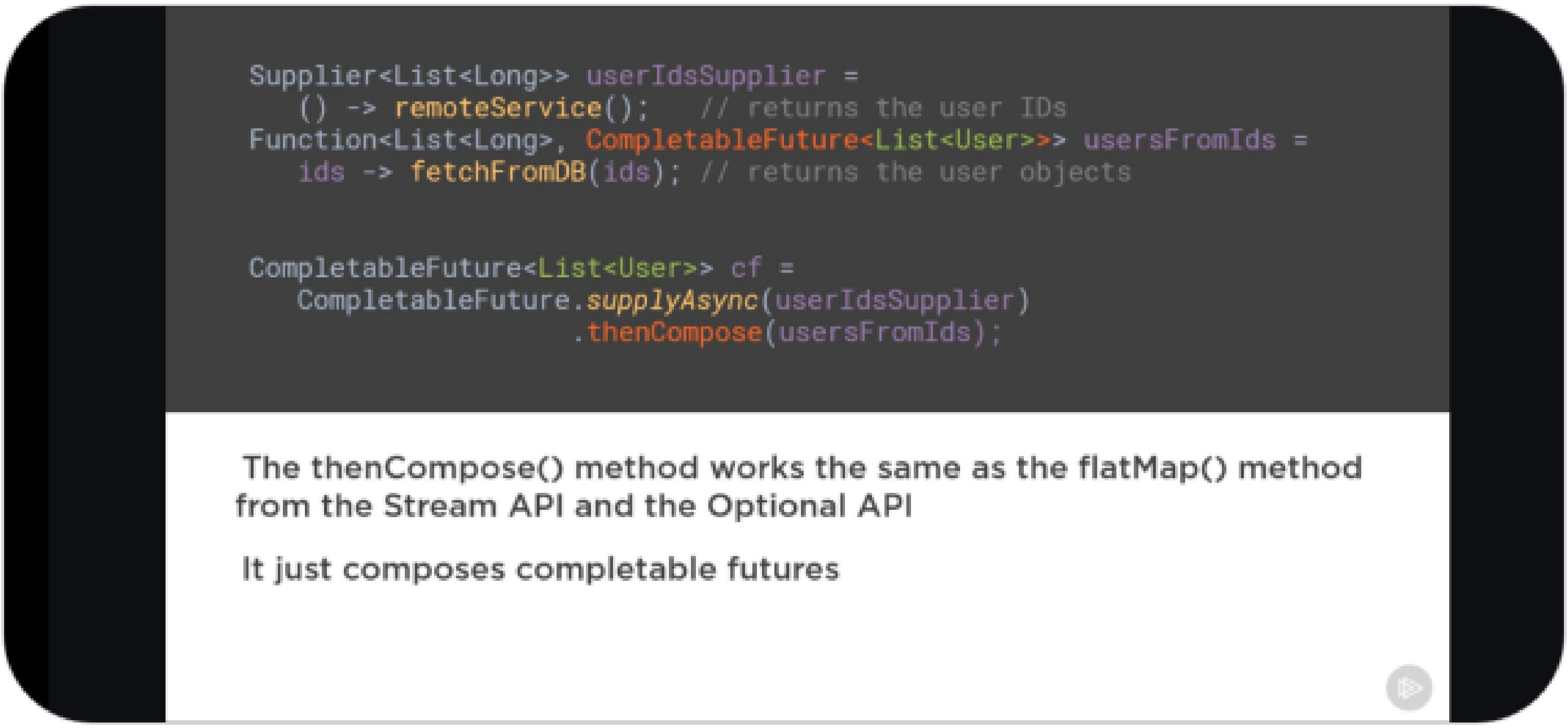


```
Supplier<List<Long>> userIdsSupplier =
   () -> remoteService();   // returns the user IDs
Function<List<Long>, List<User>> usersFromIds =
   ids -> fetchFromDB(ids); // returns the user objects


CompletableFuture<List<User>> cf =
   CompletableFuture.supplyAsync(userIdsSupplier)
                    .thenApply(usersFromIds);
```

In this pattern, fetching the user objects with the user IDs is a synchronous operation

It is conducted synchronously when the list of user IDs is available

- ○ Below code gets both data asynchronoulsy



```
Supplier<List<Long>> userIdsSupplier =
   () -> remoteService();   // returns the user IDs
Function<List<Long>, CompletableFuture<List<User>>> usersFromIds =
   ids -> fetchFromDB(ids); // returns the user objects


CompletableFuture<List<User>> cf =
   CompletableFuture.supplyAsync(userIdsSupplier)
                    .thenCompose(usersFromIds);
```

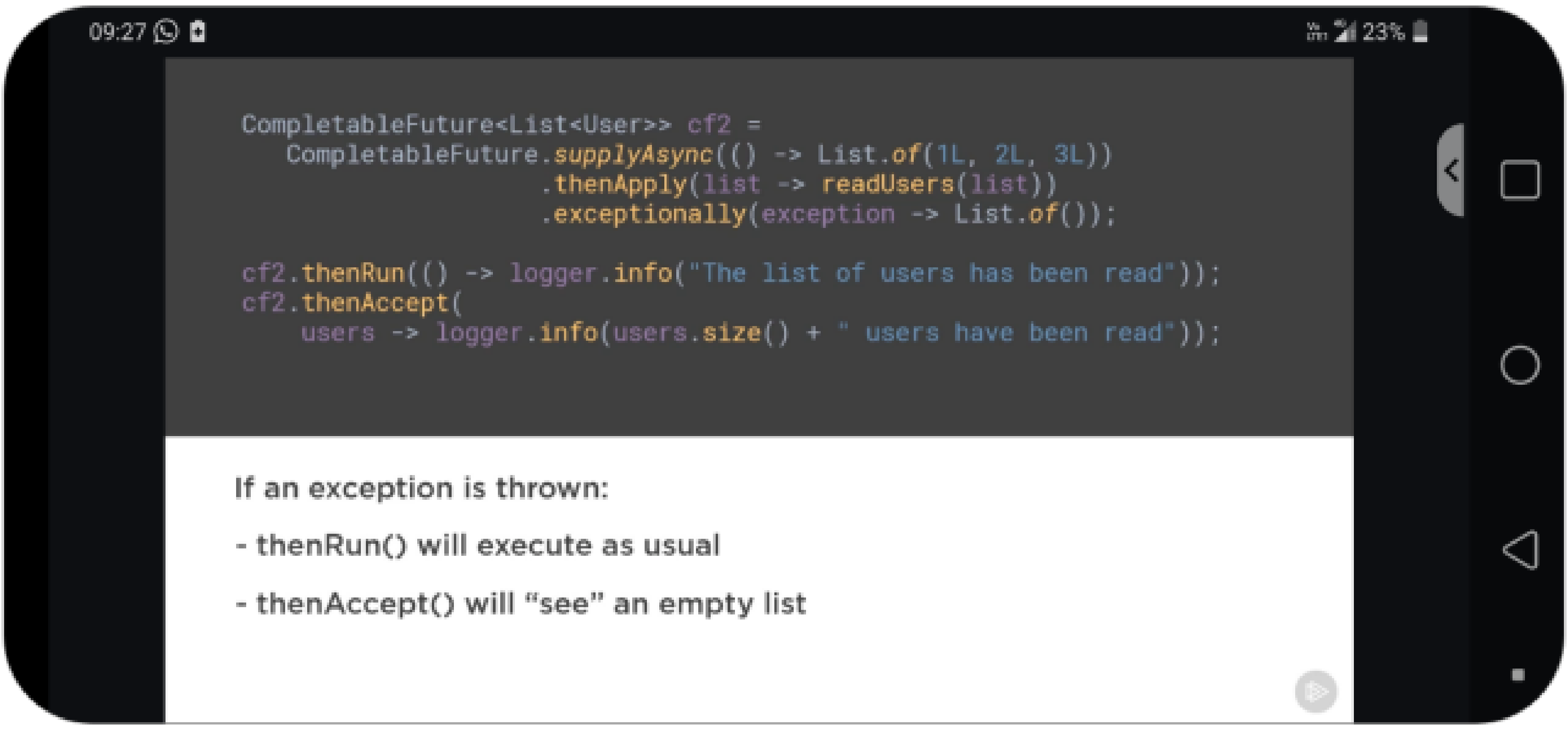
The thenCompose() method works the same as the flatMap() method from the Stream API and the Optional API

It just composes completable futures

- ○ By default all the task will run in the common fork join pool. If we provide executor service then those task ll run in pool of thread

Exception Handler

- ○ Chain with **exceptionally** method which takes the exception object in case any and return default result which will send further to downstream chains

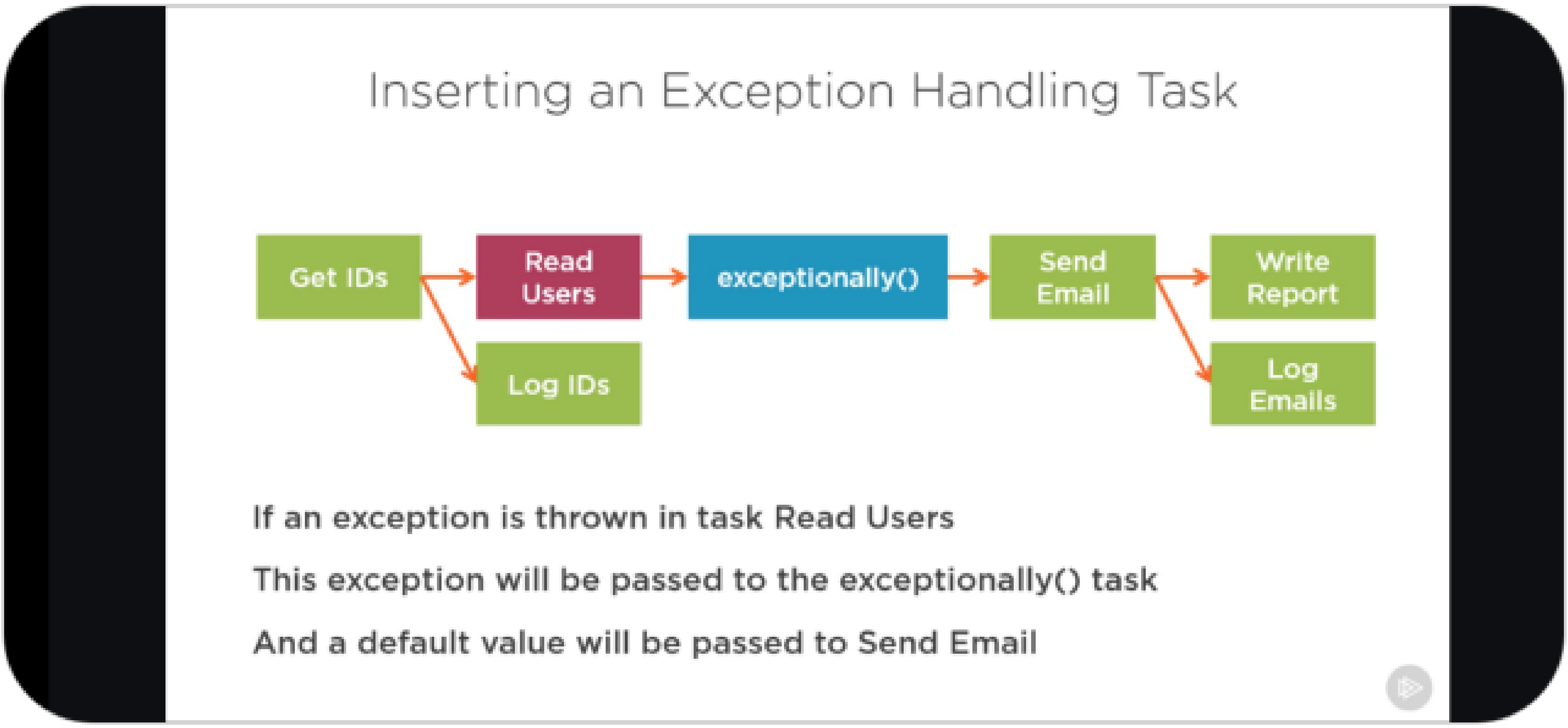


```
CompletableFuture<List<User>> cf2 =
   CompletableFuture.supplyAsync(() -> List.of(1L, 2L, 3L))
                    .thenApply(list -> readUsers(list))
                    .exceptionally(exception -> List.of());

cf2.thenRun(() -> logger.info("The list of users has been read"));
cf2.thenAccept(
    users -> logger.info(users.size() + " users have been read"));
```

If an exception is thrown:

- thenRun() will execute as usual

- thenAccept() will "see" an empty list



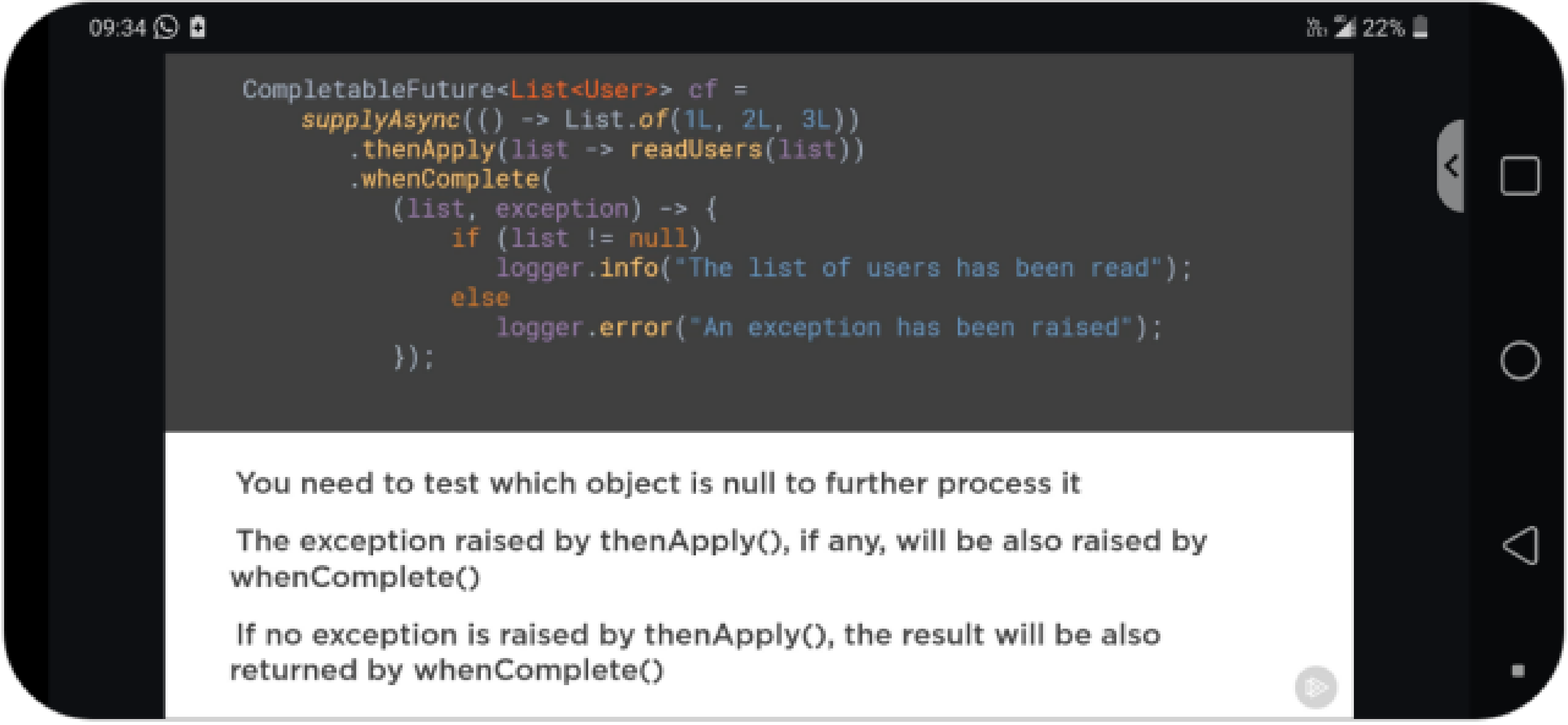
Inserting an Exception Handling Task

If an exception is thrown in task Read Users

This exception will be passed to the exceptionally() task

And a default value will be passed to Send Email

≡ ○ Chain with **whenComoplete** method

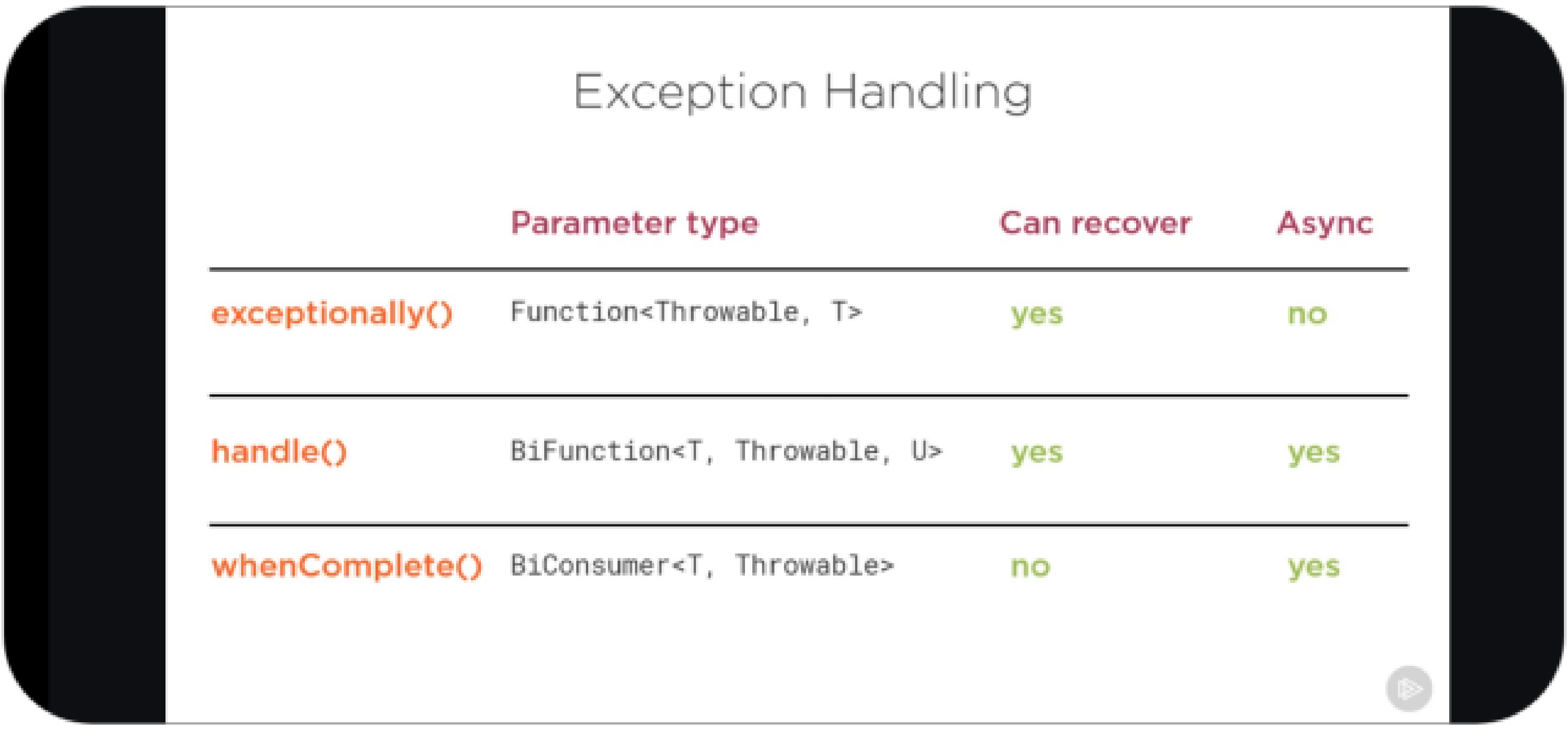


```
CompletableFuture<List<User>> cf =
    supplyAsync(() -> List.of(1L, 2L, 3L))
        .thenApply(list -> readUsers(list))
        .whenComplete(
           (list, exception) -> {
               if (list != null)
                  logger.info("The list of users has been read");
               else
                  logger.error("An exception has been raised");
           });
```

You need to test which object is null to further process it

The exception raised by thenApply(), if any, will be also raised by whenComplete()

If no exception is raised by thenApply(), the result will be also returned by whenComplete()

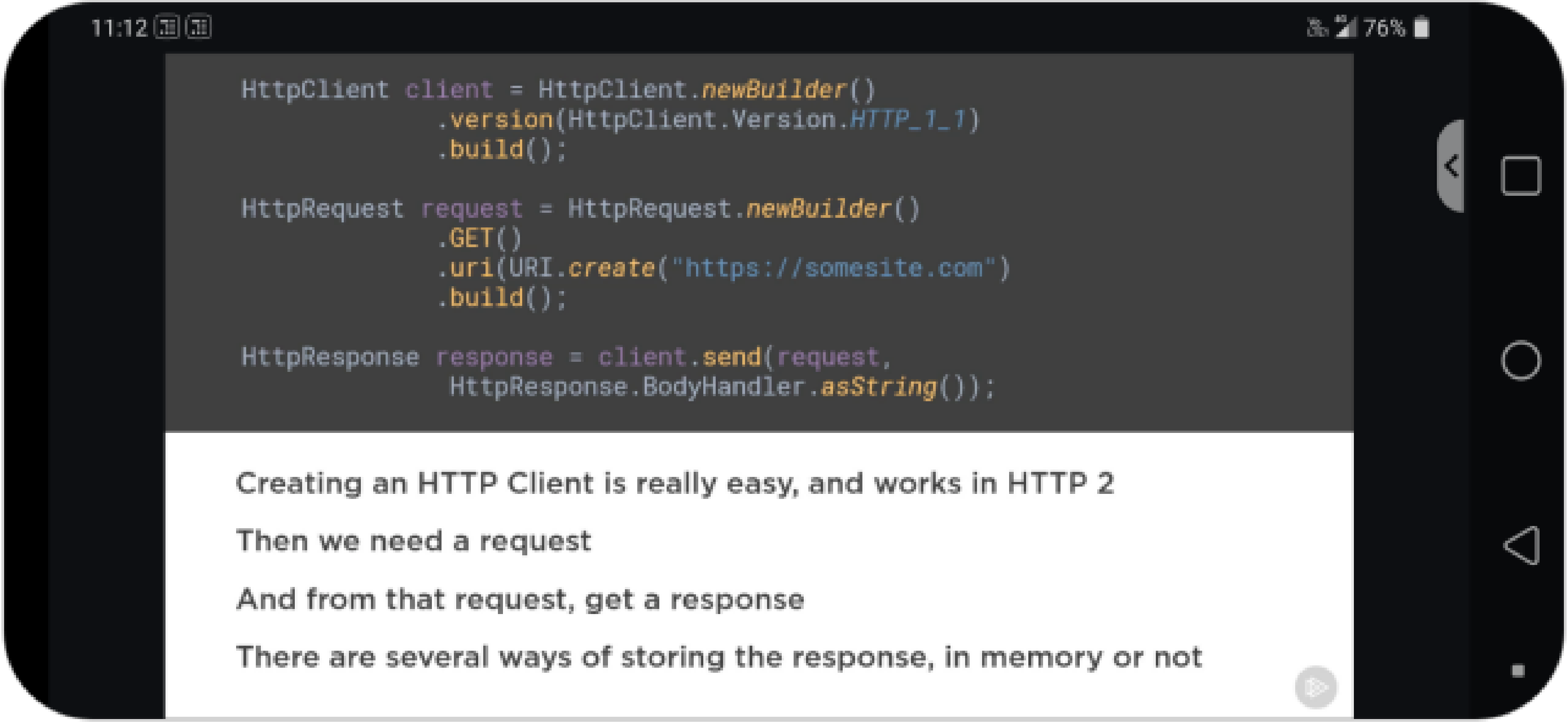


Exception Handling

	Parameter type	Can recover	Async
exceptionally()	`Function<Throwable, T>`	yes	no
handle()	`BiFunction<T, Throwable, U>`	yes	yes
whenComplete()	`BiConsumer<T, Throwable>`	no	yes

HttpClinet

- From java 10 we have HttpClient to handle asynchronous request



```
HttpClient client = HttpClient.newBuilder()
                .version(HttpClient.Version.HTTP_1_1)
                .build();

HttpRequest request = HttpRequest.newBuilder()
                .GET()
                .uri(URI.create("https://somesite.com")
                .build();

HttpResponse response = client.send(request,
                 HttpResponse.BodyHandler.asString());
```

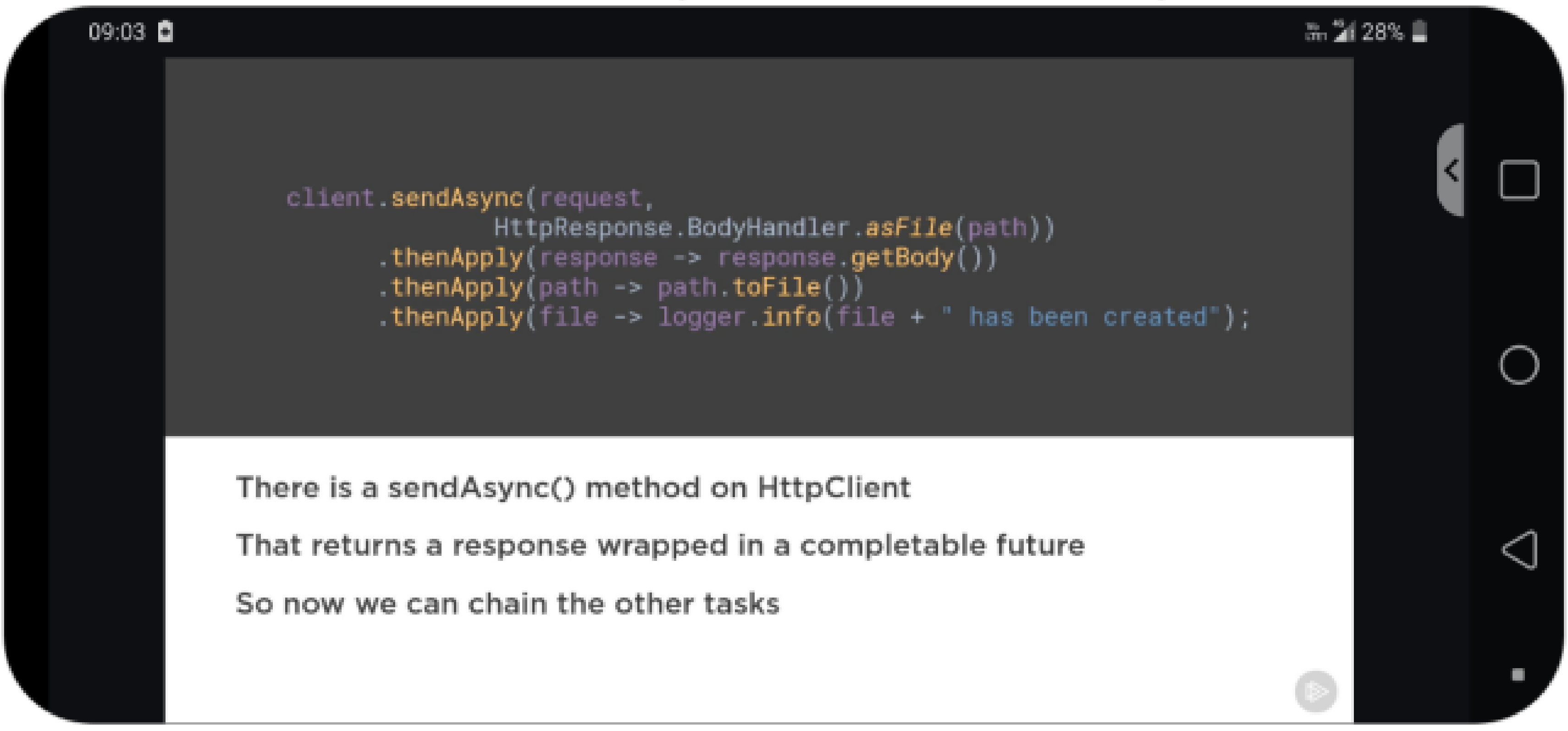
Creating an HTTP Client is really easy, and works in HTTP 2

Then we need a request

And from that request, get a response

There are several ways of storing the response, in memory or not

Sending asynchronously

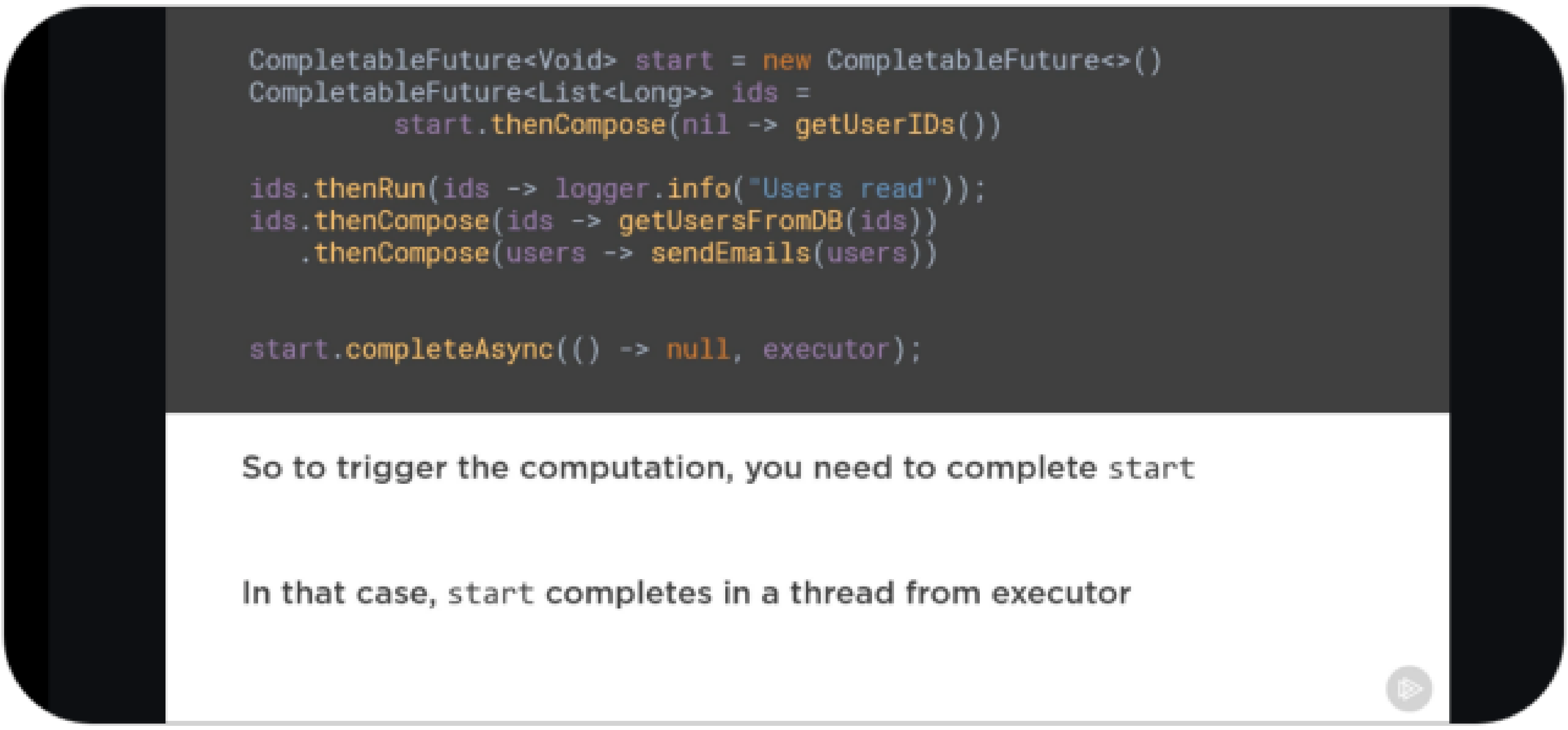


```
client.sendAsync(request,
                 HttpResponse.BodyHandler.asFile(path))
        .thenApply(response -> response.getBody())
        .thenApply(path -> path.toFile())
        .thenApply(file -> logger.info(file + " has been created");
```

There is a sendAsync() method on HttpClient

That returns a response wrapped in a completable future

So now we can chain the other tasks



```
CompletableFuture<Void> start = new CompletableFuture<>()
CompletableFuture<List<Long>> ids =
         start.thenCompose(nil -> getUserIDs())

ids.thenRun(ids -> logger.info("Users read"));
ids.thenCompose(ids -> getUsersFromDB(ids))
   .thenCompose(users -> sendEmails(users))


start.completeAsync(() -> null, executor);
```

So to trigger the computation, you need to complete `start`

In that case, `start` completes in a thread from executor