- Inbound adapters: server or listener
- Inbound ports: services and actions
- Outbound adapters: client or publisher
- Outbound ports: data repository
- Domain objects: command entity and query entity

==========

Java 8
Functional and Reactive Programming

Functional Programming

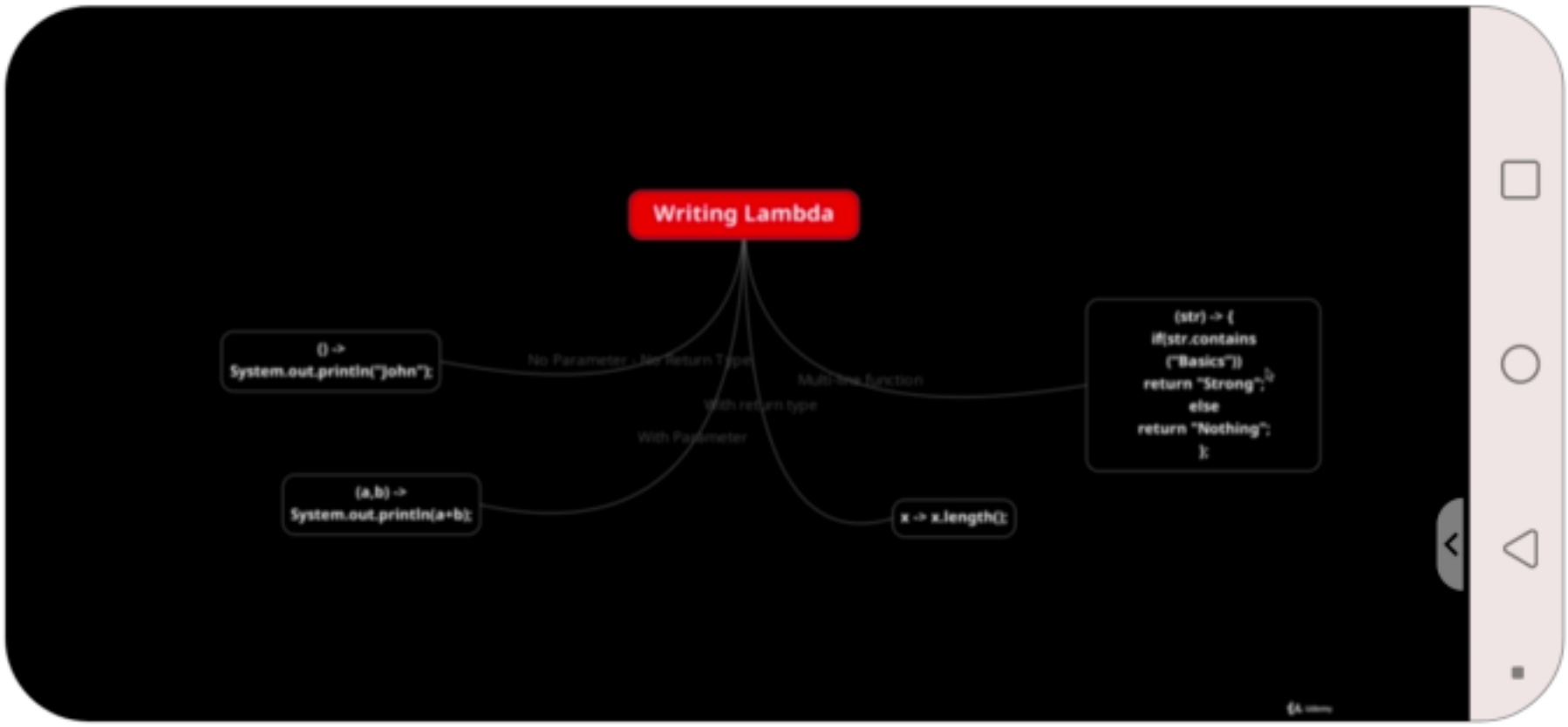
Functional programming paradigm is built upon the idea that everything is a pure function.
Java can uses functions as high class citizen.
Functional interface and lambda

- ≡ ○ @FunctionalInterface: interface with only one public abstract void method.
- ≡ ○ Lambda expression: To create the behaviour of the functional interfaces on the fly instead of using anonymous inner classes
- ≡ ○ Example: Thread t = new Thread(() -> sysout("printed"):
- ≡ ○ We can create behaviour by using inner classes but this ll create separate class files for each inner classes.
- ≡ ○ If we use lamba this ll not create any separate class files and this behaviour ll be executed at run time instead of compile time behaviour check as like inner classes
- ≡ ○ We can pass different behaviors to functions through lambdas. With lambda java ll treat functions as first class objects so

that we can pass functions into functions as parameter

- ○ Oops are basically imperative approach where developer need to write each code about how the logic should work
- ○ In functional programming which is subset of declarative approach developer weite code to say what needs to be done. Functional code ll take care of how the logic should work.
- ○ Functional programming has many benifits over imperative programming as thos ll be error free, always uses the immutable objects, work well in multi thread environments
- ○ We can reuse the same functional interfaces with different behaviour
- ○ 1. Create a lambda or function through functional interface. 2.

Call the lambda or pass the lambda to another function as a parameter



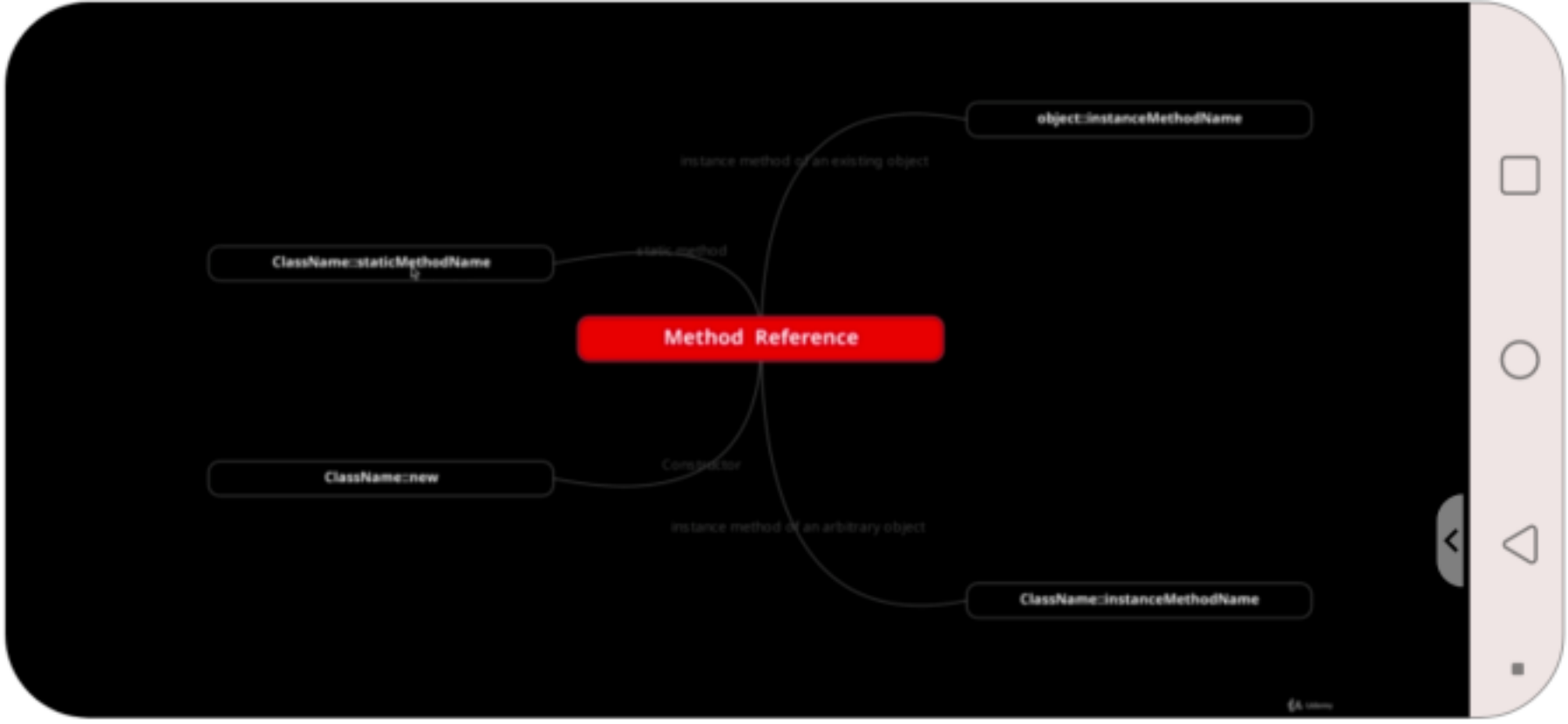
Default functional interfaces

- Java provides default functional interfaces like Predicator, Consumer. Supllier, Function, BiFunction, UnaryOperator, BinaryOperator
- Predicator accept something and returns boolean
- Consumer accepts something returns void
- Supplier accept nothing returns something

- Function accept something return something else
- BiFunction accept 2 arguments and returns something
- Predicator.test(), consumer.accept(), supplier.get(),Function.apply() methods

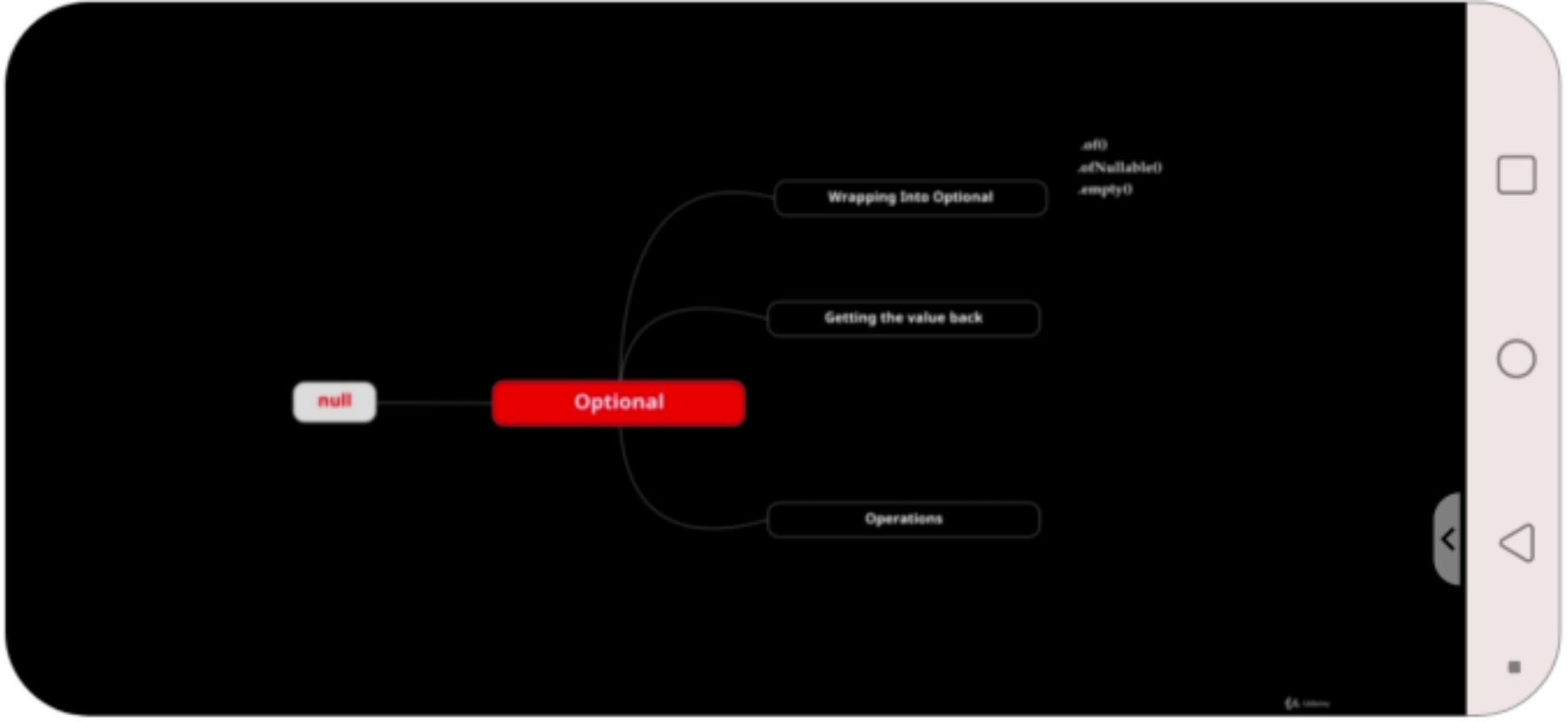
Method Refrences and Constructor Reference

- Instead of using lambda we can use method and constructor refrences
- Method refences like objectname::methodname, classname::static method name, classname::instance method name
- Constructor refernces like classname::new



Java optional operator

- Java provides optional operator to hanlde null pointer exception
- It provides a wrapper around the object to handle null cases
- It provides multiple methods to generates optional from an existinf object, to get value from optional, to hanlde exception etc



Functional programming concepts

- Functional programming as first class citizens. Can pass functions as parameter to a method, return a function from method etc
- Pure mathematical functions: for the inout x always return a value f(x). No other ouput or input functionality.no changing value of parameters.just takes the data perform some logic and return data. So no side effects.
- Higher order functions: functions which accepts multiple functions

in the form of lambda expressions, or a function which returns another function

≡ ○ Referential transperancy: function parametee can be like 2*2*2 or 2+4 or 6

≡ ○ Method chaining: calling functions one by one by passing previous returned value to next function

≡ ○ Method composition: reverse of method chaining

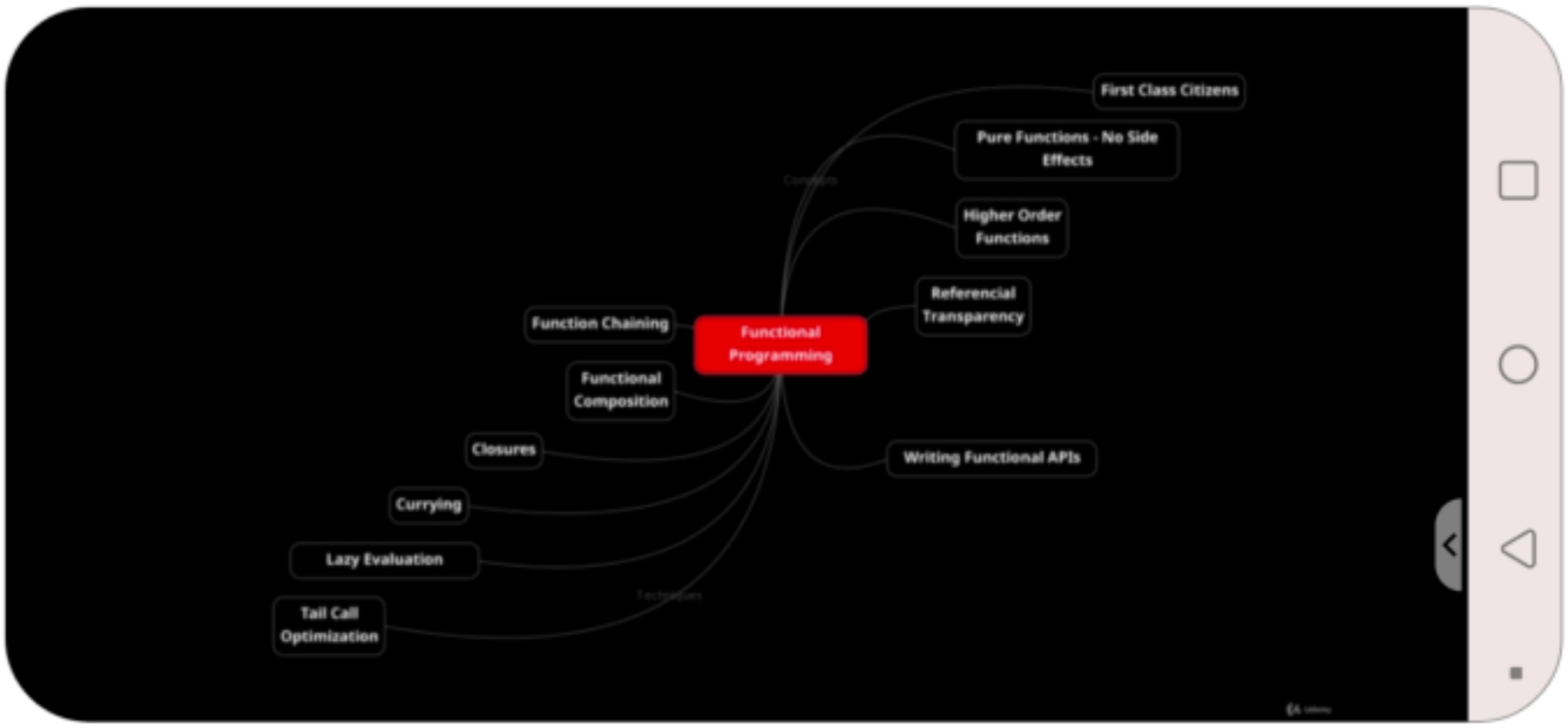
≡ ○ Closure: function which can remember value of it lexical scoped variable even the function is called later.

≡ ○ Currying: converting a function with multiple parameters to multiple functions with each of those single parameter.

≡ ○ Lazy evaluation: lambdas are called lazily

≡ ○ Tail recursion: instead of having

recursion with all previous call stack active use a recursion with accumalator which does not keep call stack



Java streams

Collections are used to store the data and streams are used to perform operations on data. We can not add or update data through streams.

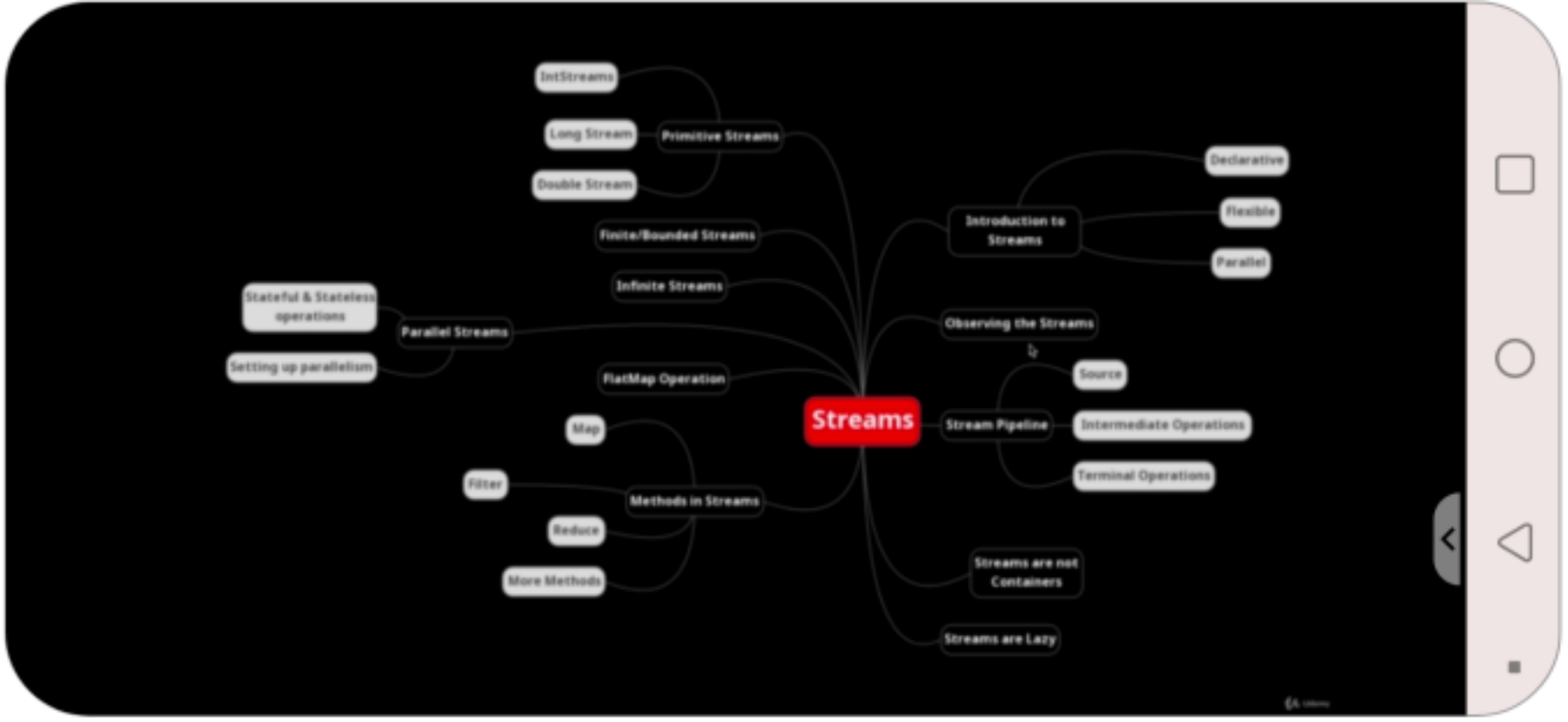
- Takes the collection as stream of data, performs multiple intermediate operations on each of the data which wiil returns new streams of data, amd finally apply the terminating operation which

will collect the final data into containers or collection.

- Streams are basically lazy. All the intermediate operations will not be executed until we call the terminal operation on stream. Terminal operations are eager.
- Stream ll have data source, intermediate operations which takes each previous stream, apply some calculation and returns new stream and finally collector to terminate the stream.
- Ex: books.stream().filter((book) -> book.getRating() > 3).collect(Collectors.toList());
- Bounded stream: There are many stream operators like filter, map, reduce, peek, list.of(),map.of(),stream.of(),stream.build(),builder.add(),builder.add(),lost.stream(),map.entries.stream(),Arrays.stream(array),flatmap(

)

- Infinite stream: stream.iterate(0, i->i+1);
- Parallel Stream: list.parallelStream(), lost.stream().parallel(). To incrrease the performance.
- Numeric Stream Is like IntStream, LongStream and DoubleStream. mapToInt,mapToObj,mapToLong, mapToDouble
- Statefull stream: uses outside informations. Do not use parallelism with statefull streams like skip, limit etc which are statefull stream.
- Stateless stream: do not uses any outside informations, performs one by one



Creating Custom Stream through Spliterator

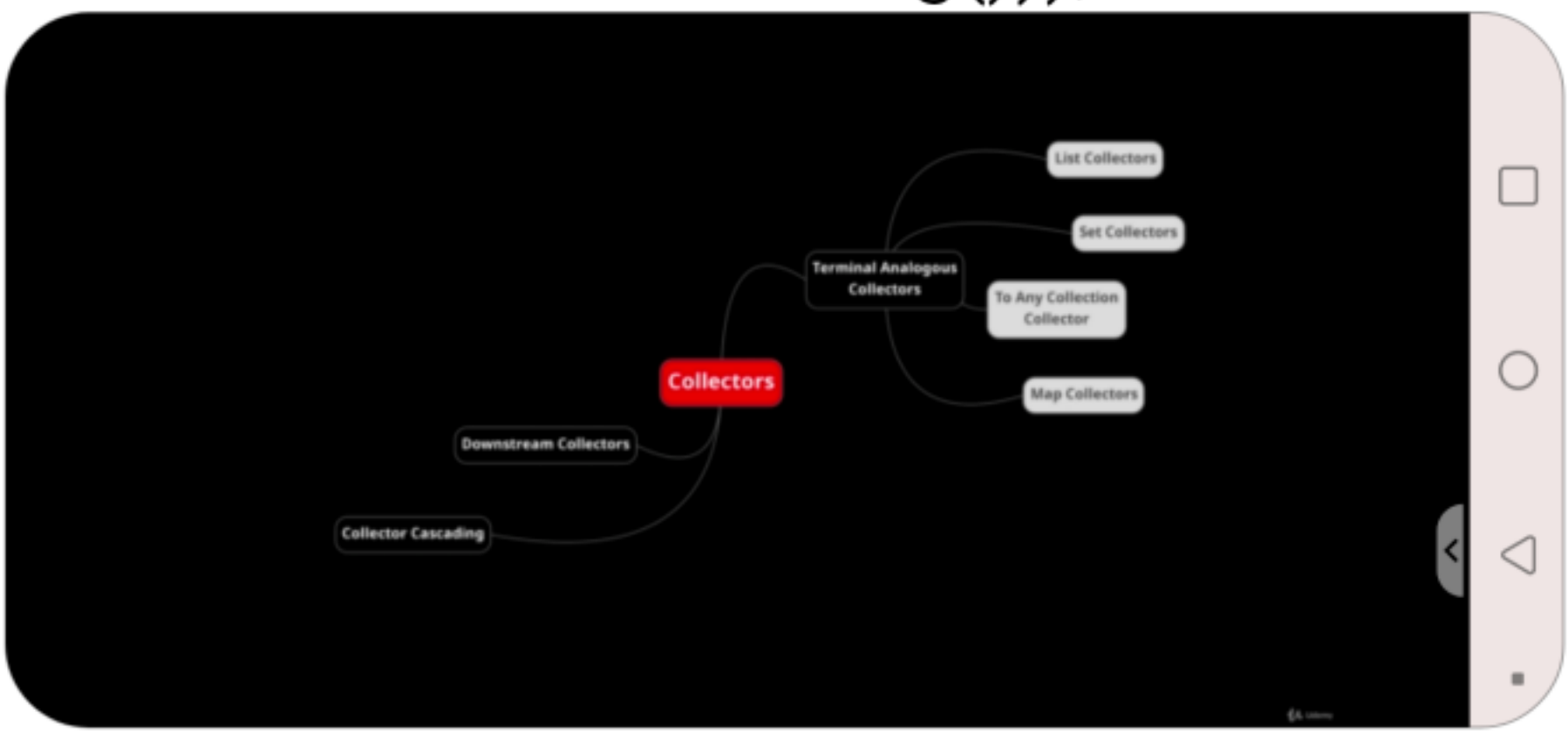
- ≡ ○ When we need to read stream of data from a custom sources like file which is not a collection then we need to create a custom stream through Spliterator

Collectors utility class for data processing

- ≡ ○ stream.collect(Collectors.toList())
- ≡ ○ Stream.collect(Collectors.toSet())
- ≡ ○ stream.collect(Collectors.toCollection(TreeSet::new))

- stream.collect(Collectors.toMap(key, value))
- stream.collect(Collectors.partitioningBy(condition)) - this creates two different list of datas one which passes the condition other one is not.
- stream.collect(Collectors.groupingBy(list of group names)) - this will creates multiple lists of data where each data belongs to a particular group.
- stream.collect(Collectors.joining(join by condition)) - for example we can join stream of empliyee names into a single string with coma separated values.
- Downstream collectors: collector inside collector or nested/ cascaded collectors. This will be usefull for post processing collected data.
- Ex: through stream we can get list

of employees who belongs to each group. Then this data we can pass to another collectors to postprocess this data to get count of number of employees in each group.
stream.collect(Collectors.groupingBy(group by condition, Collectors.counting()));



Playing with java collections in functional style

Basic operations on any collections are: Traversing, sorting, filtering or searching, mapping or transforming, reducing

List

≡ ○ list.forEach()

≡ ○ list.sort((a,b)->b.getInt()-a.getInt())

≡ ○ list.stream().filter(movie->movie.getIndustry("Bollywood"))

≡ ○ list.stream((.map(movie->movie.getName())

≡ ○ list.stream().map(movie -> movie.getName()).reduce((m1,m2->m1 +"|"+m2)) or we can use sum method

Set

≡ ○ set.forEach()

≡ ○ set.stream().sorted()

≡ ○ set.stream().collect(Collectors.toCollection(TreeSet::new))

≡ ○ set.stream().filter(movie->movie.getIndustry("Bollywood"))

≡ ○ set.stream().map(movie->movie.getName())

≡ ○ list.stream().map(movie->movie.g

etInt).sum())

Map

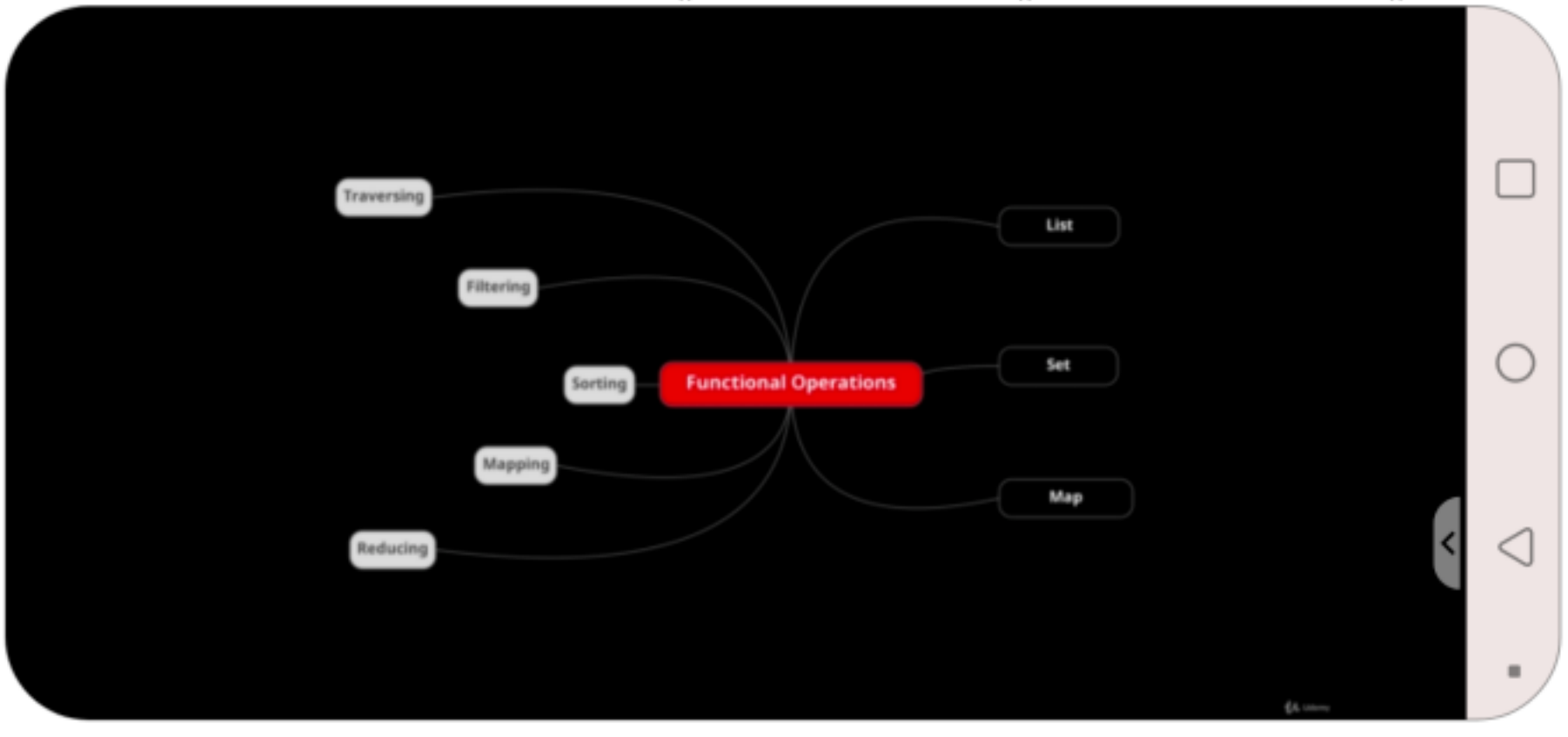
≡ ○ map.forEach((k,v)->sysout(k+" "+v))

≡ ○ map.entrySet().stream().sorted(Entry.comparingByValue())

≡ ○ map.entrySet().stream().filter(e->e.getName.equals("true"))

≡ ○ map.entrySet().stream().map(movie->movie.getName())

≡ ○ map.values().stream().average()



Reactive Programming

Reactive programming paradigm is built upon the idea that everything is a

stream observer and observable philosophy. Reactive means one who reacts.

Reactive Manifesto

- Responsive: Application should be responsive. There should be upper limit before which application should response to user
- Resilient: Application should be resilient for error or failures. Errors and faikures should be considered as first class citizens like data.
- Elastic: Handle user loads as per incoming requests. Auto scale up or scale down the application instances
- Message Driven: Applications are communicated between each other asynchronously through messages

Fundamental behind reactive

programming

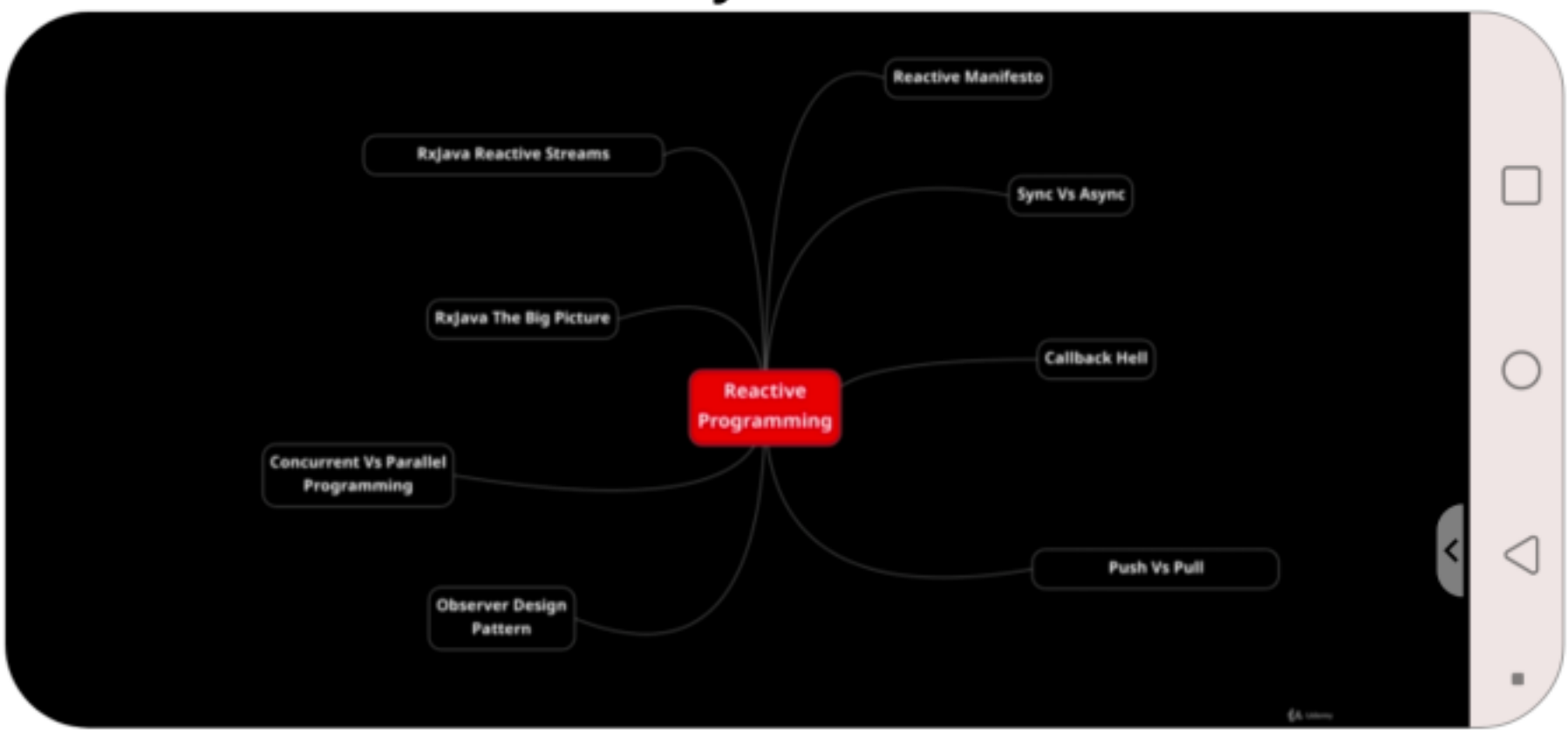
- Sync vs Async: occurences of events independent of main program flow
- Callback and callback hells: we can run some functions on a separate thread, by passing some callback method. So the main thread continuous to run and once another thread completes it calls the passed callback method automatically so the callback method runs. If we have many such callbacks then maintaining those callbacks will be an issue.
- Push vs Pull:request response model uses pull mechanism while RxJava uses push mechanism
- Obseever design pattern: RxJava is implemented by using Observer design pattern. List of observers which are interested in knowing

data changes of a subject are subscribed to a subjet. Then whenver data of subject is changed, subject automatically notifies each of the registered observers. Then observers will request subjects and gets the changed data. In RxJava subjects are called observables.

- ○ Subjects are usually has methods to subscribe, unsubscribe and notify Observer. And observer usually has a method update to update itself with the chnaged data.
- ○ Concurrency/multithreading vs parallel programming: concurrency means in a sigle CPU how run multiple tasks in muliple threads. Parallelism means how to run multiple tasks across available multiple cpu's. RxJava supports both

concurrency and parallelism

- Reactive streams: As reactive programming become popular java come up with a standard called reactive streams. This provides set off interfaces witch uses flowable instead odlf observables and api's which implements these interfaces are reactive ready



RxJava

- Include the RxJava library
- Create the observables which can emit either series of data, trigger an complete event or emit an

error.

- Observable<String> source = Observable.create(e -> { e.onNext("Hello"); e.onNext("Welcome");}
- List of obseevers who can subscribe to this observable.
- Observer obs1 = source.subscribe(data->sysout(data));
- Observer obs2 = source.subscribe(data -> sysout(data));

Observable and Observer

- Observables are similar to java stream but in observables we can get the data asynchronously
- Observables are also lazy similar to java stream, until the observer subscribe on the observables none of the data are emitted

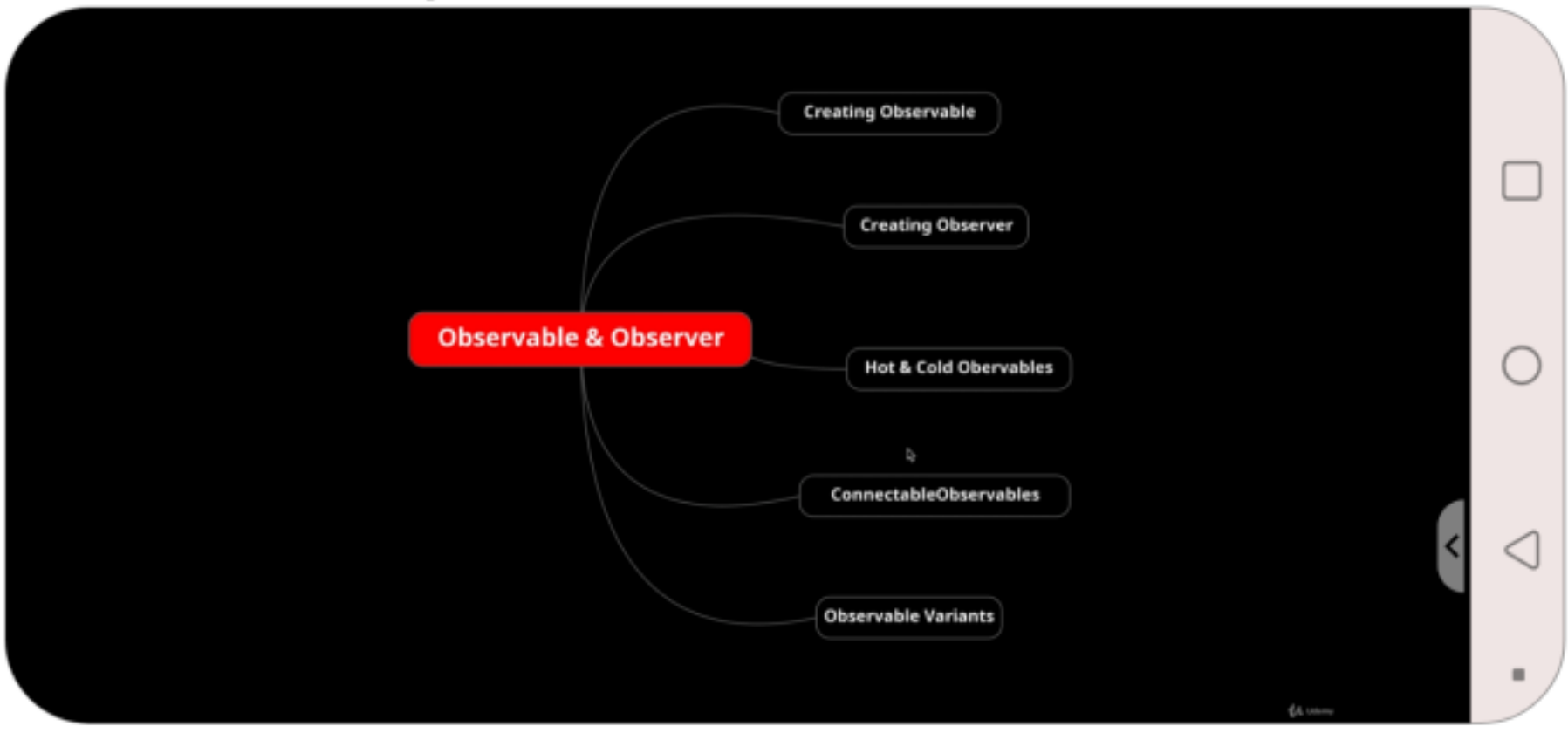
- Observables or subjects has 3 channels. One is to keep emit the data another one is to send complete signal and the last one is to send error signal and error informations to observers.
- Observer can keep listen the emitted data or completed signal or handling the the error.
- There are many ways to create the observables in java by using RxJava plugin. Like Observable.create(e->e.onNext(1)), Observable.just(1,2.3), Observable.fromIterable(any java collection),range?,interval,empty,never
- Any Observer then can subscribe to this observables and gets the data. Observables.subscribe(sysout(data))
- Hot and cold observables: Hot

observables are eager while cold observables are lazy. All the observer will get data from beginning whenver it subscribes from cold observables. In case of hot observables all observer ll receive the data which are triggered only after it subscribed. Example for hot observable is playing a movie in theatre. Ex for cold obseevable is playing a movie at home.

- Connectable observable: this can be used to convert cold to hot observable.
  ConnectableObservable c = Observable.interval(1,TimeUnit.seconds).publish(); c.conncet(); c.subscribe(data->sysout(data));
- Observable varients: observable with zero emission called completable, observable with either 0 or 1 called maybe,

observable with 1 emission called single.

- Dispose(): this method is used to unsubscribe from observable so that it ll get disposed. Disposable d = source.subscribe(e->sysout(e)); d.dispose(); we can use CompositeDisposable to dispose multiple observers.



RxJava Operators

- RxJava provides multiple operators which are used to perfoem transform, filter, merge and other kinds of operations on

the data.

- Each of these operators takes the data from the source and send the data to next oerators. This ll continue until it reaches terminal operator.

Types of operators:

- Suppressing operaotr: this ll supress the data from observables to observer. Example: filter(),take(5),skip(5),distinct(),elementAt
- Transforming operator: this ll transform the data from observable to observer. Example: map(),cast(),delay(),scan(),sorted(),repeat()
- Reducing operators: this ll takes all the finite data and convert to single data. Example: reduce(),count(),contains(),all(),any()

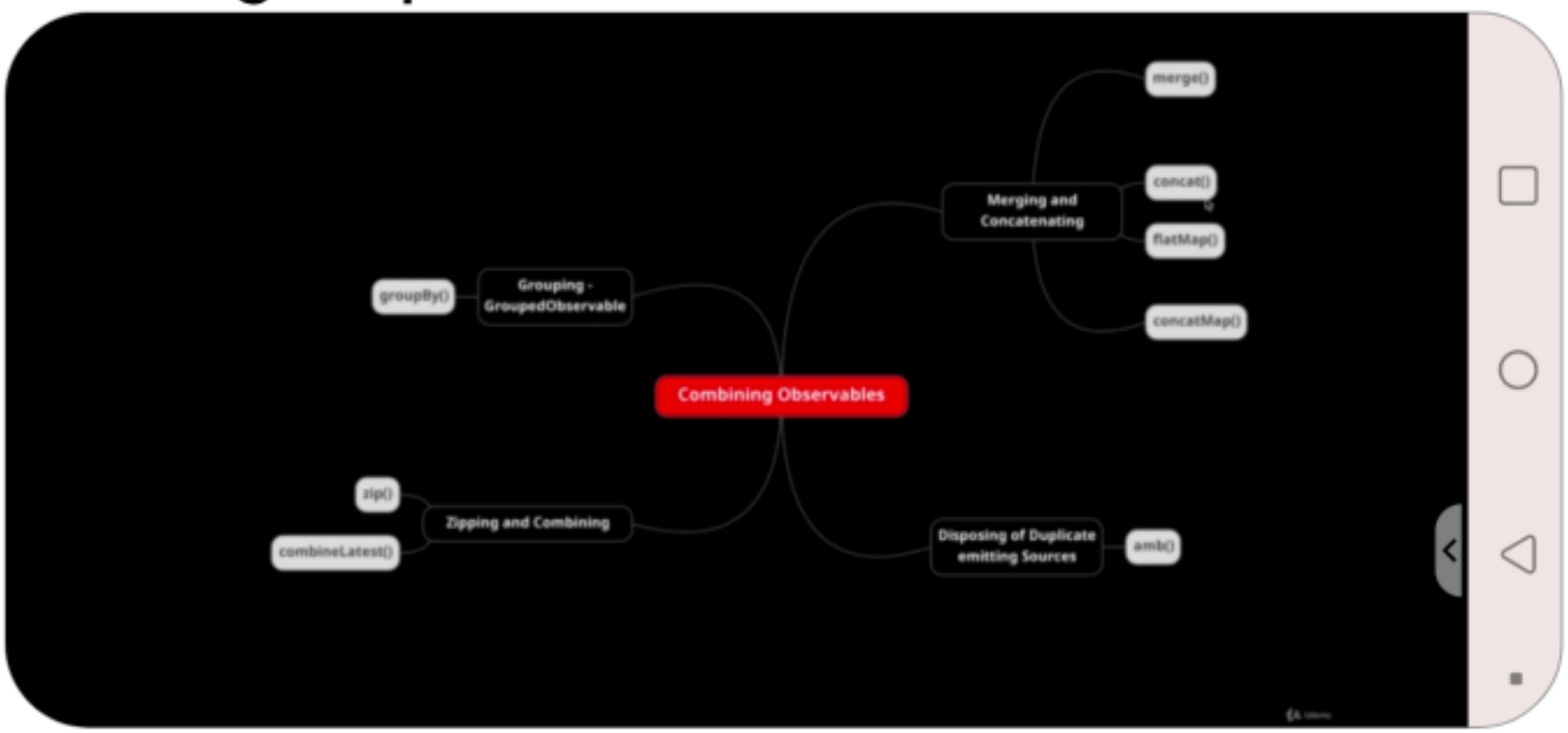
- Collection operator: combines all the upstream emissions into some collection. Example: toList(),toMap(),collect(),toSortedList()
- Error recovery operator: used to handle the errors and recover from them. Example: onError(),onErrorReturn(),onErrorReturnItem(), retry()
- Action operator: used to debussing in the observable chain. Example: doOnNext(),doOnError(), doOnSubscribe(), doOnComplete(), peek

Combining observables:
Combining the emissions of multiple observables to have a single observable source

- Merge: merge emissions from 2 observables in the order they

triggerred

- Concat: merge emission from first observable and once it is completed merge emission from second observable
- Switch: switch emission from first observable to second observable
- Groupby: single observable emission passed to grouping functions which emits multiple grouped observables



Concurrency and parallelism

- We can run the observables emission in a thread, so each

observer ll invoke and execite it in a separate thread

Schedulers: RxJava peovides schedulers to support concurrent and parallelism.

- Computation schedulers: use this when there is a lots of calculation. This is the default scheduler to use. Creates pool of threads and reuses threads.
  Observable<Integer> src = Observable.just(1,2,3,4).**subscribeOn**(**Schedulers.computation**());
  src.subscribe(e->compute());
- IOschedulers (Schedulers.io()): use this when we have io operation or http call operation. Creates pool of threads and reuses threads
- Single thread (Schedulers.single()): creates only one thread to run multiple tasks sequentially for all

observers. This scheduler is required when we have a sensitive information which needs to executed one by one each obseever

- ○ New Thread Schedulers (Schedulers.newThread()): creates new thread for each observer and destroys it when its done.
- ○ From scheduler (Schedulers.from()): to create custom scheduler by using java's executor service. Schedulers.from(Executors.newFixedThreadPool(2))

subscribeOn

- ○ Tells observable about which type of scheduler to use.
- ○ If there are multiple subscribeOn then the one which is near to source will be selected by default

observeOn

- If there are multiple subscribeOn in a chain and we want to move from one subsrcibeOn to another we use observeOn()Schedulers.io())

Using flatmap for concurrency

- We can not run emission concurrently on a single observable. It should emit sequentially. But we can create multiple observables from those emission and run them parallely by using flatmap

Replay and Cache

- Replay operator holds/caches all or preciously emitted values and re-emit them from beginning when new observer subscribed later. It returns connectable observable. Obseevable.interval(1,Times.millisec).replay(1).autoConnect();

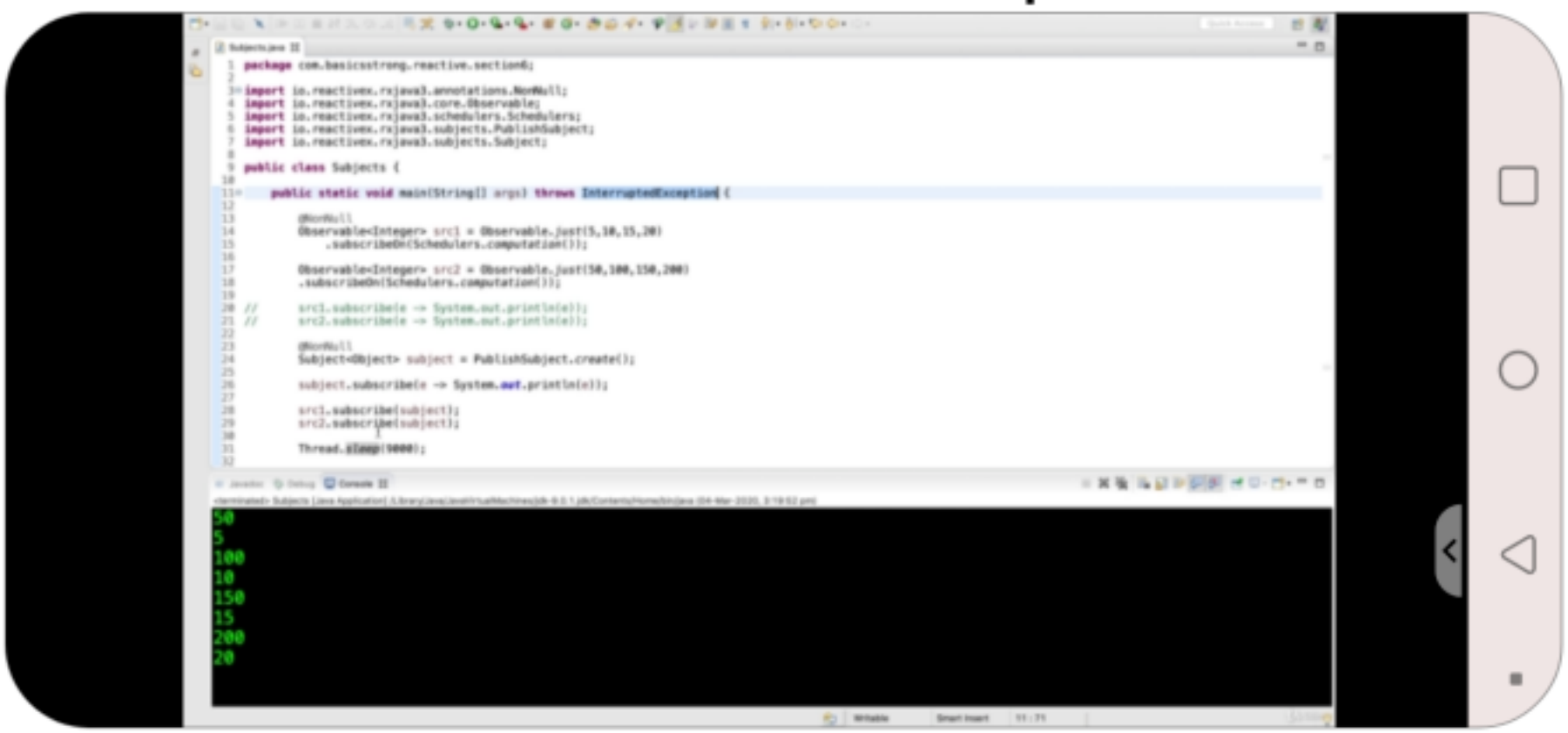
- Cache ooerator: similar to replay where it caches all emissions and when observer subscribes it returns cacjed values. Only difference between replay and cache is cache does not returns connected observables. It just returns observables.

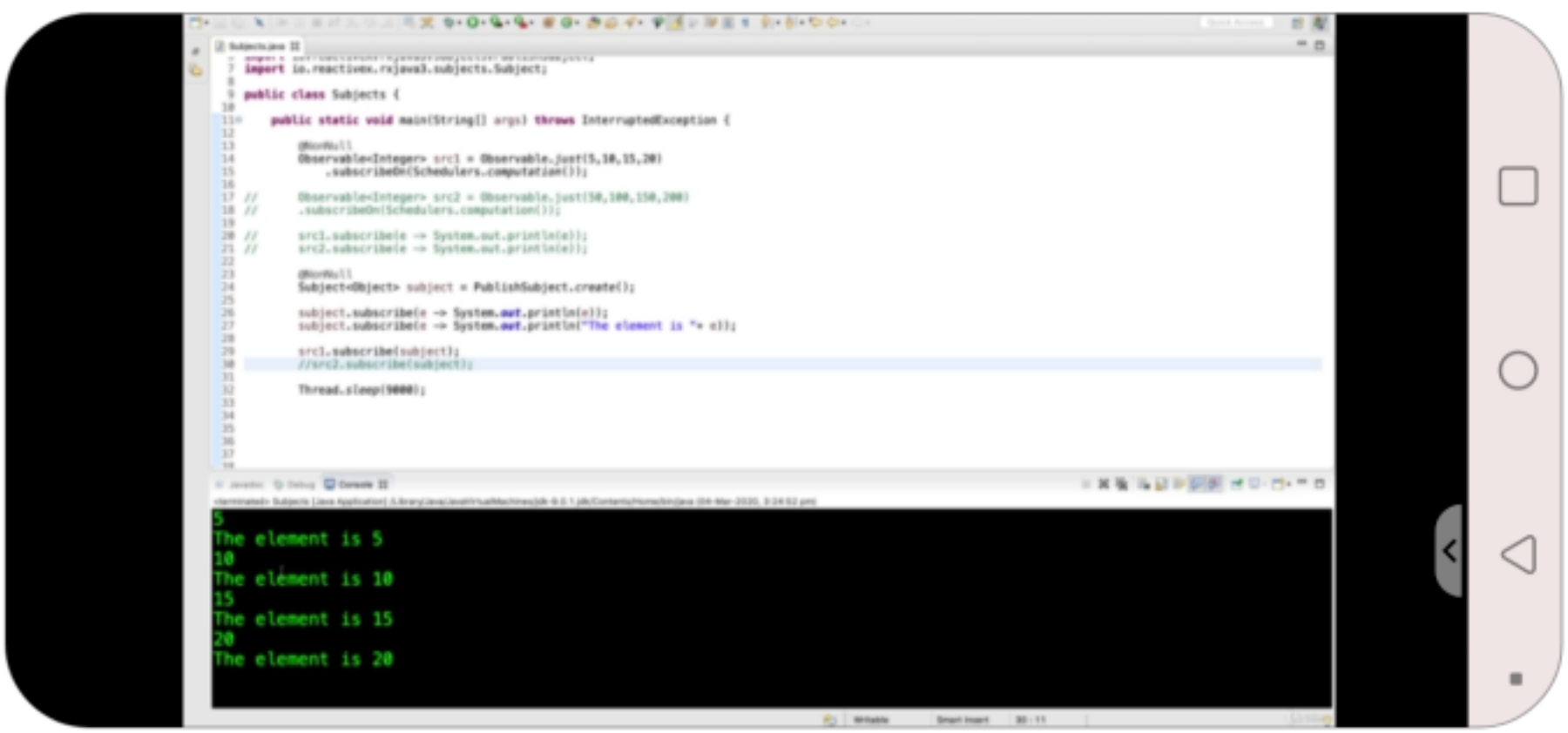
Subject

- Can act as both observables and observers
- Subject can be a observer for an observable and at the same time it can be an observable for other observer. Its like a bridge between observable and Observer.
- Observable src = Observable.just(1,3,5);Subject<Object> sub = PublishSubject.create(); sub.subscribe(e->sysout(e));

sex.subscribe(sub);

- Subjects are multicast or hot observables. As soon as it subscribes it pass values to observers.
- Subjects can use merge the emissions from multiple sources.





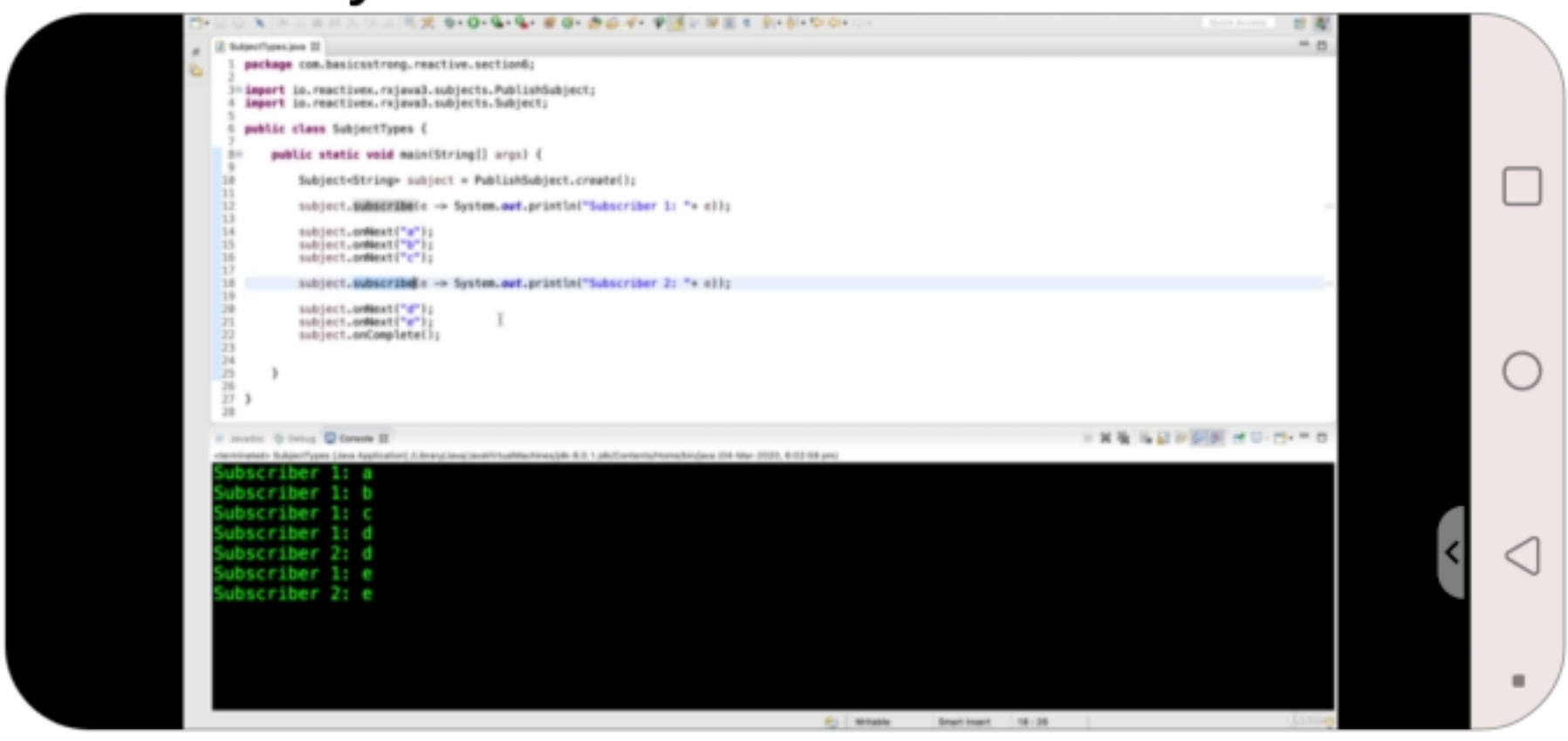
OnNext, onComplete, onError:

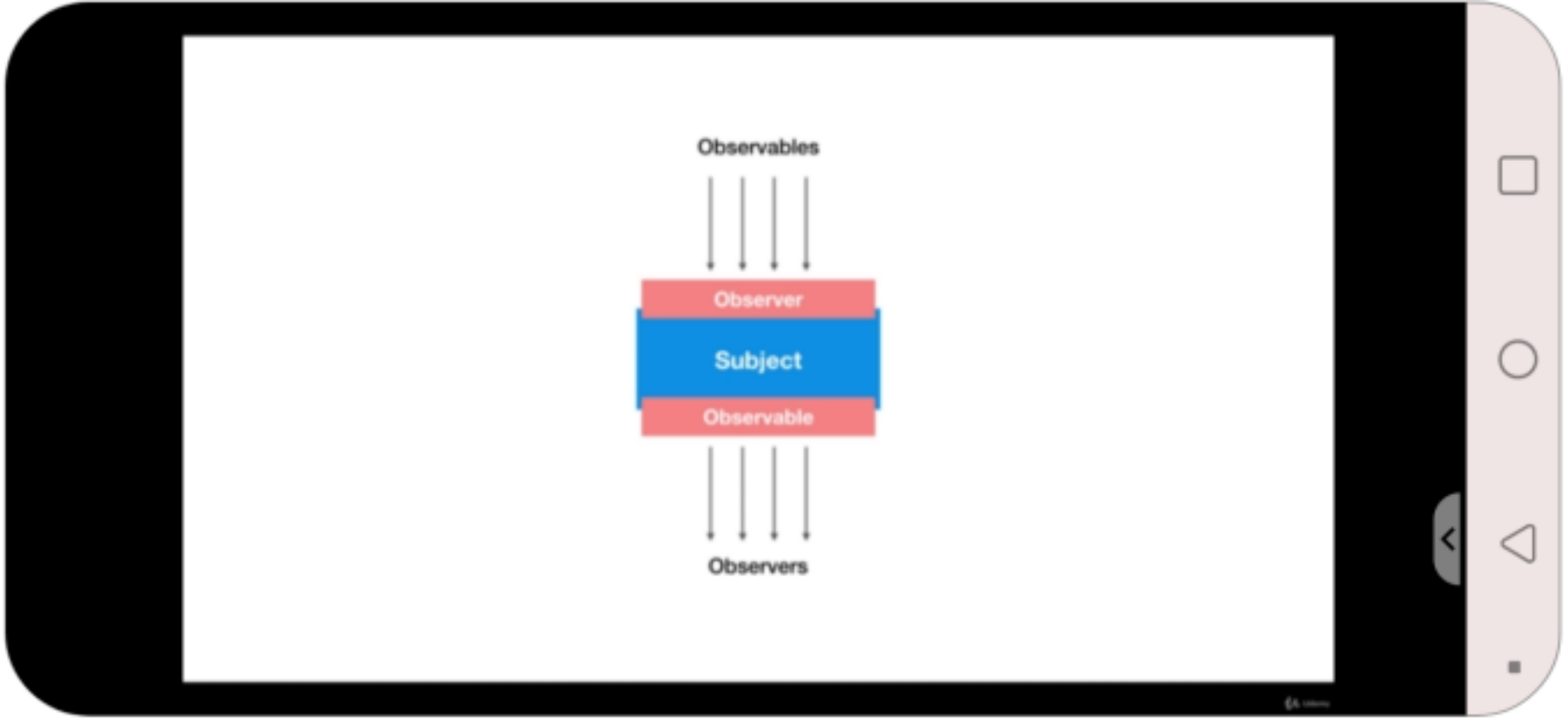
- Create a subject
- Subscribe multiple observer on this subject
- Emit data from subject using onNext()
- Complete the emission through onComplete()
- Handle the error through onError()

Types of subject:

- PublishSubject: starts emission from the moment observer subscribe to it
- ReplaySubject: emits all the items regardless when observer subscribes. Its basically PublishSubject with cache operator
- BehaviourSubject: emit most recent item with subseequent items from the point of subscription

- AsyncSubject: emits only the last value of the observable after it completes
- UnicastSubject: buffers all emissions until observer subscribes to it. Once observer subscribes it releases all buffered emissions and clears it. Allows only one observer





Buffering throttling and switching

- Observables emits data at a higher rate than observer. So to handle this situation we use below operators
- Buffer(): gather emissions in batch and emits each batch as collection
- Window(): buffer into other observables rather than collections
- Throttle(): skip some of the emissions which are emitted rapidly

- Switchmap(): similar to flatmap but difference is it only subscribes to last emitted observable, and dispose the previous one

Flowable and Backpressure

- Producer-Consumer problem or Backpressure: producer emits the data at a faster rate than the consumer consume it because of May be concurrency or consumer's operations. So all emissions are queued up in the memory which may cause memory issue
- Flowable - Subscriber: to handle backpressure, we have flowable which can emits data and subscriber which can listem data. Similar to observable and subscribe. Flowable is the replacement for observable

where we need backpressure.

≡ ○ With flowable we dont have producer consumer issues. Producer ll produce some set of data which consumer consumes immediately and then producer continues

≡ ○ While creating flowable we need to define backpressure strategies

≡ ○ Flowable is required when source emits large amount of data and when application uses concurrency

==========

JAVA9:

Java 9 provides additional features on top of java8.

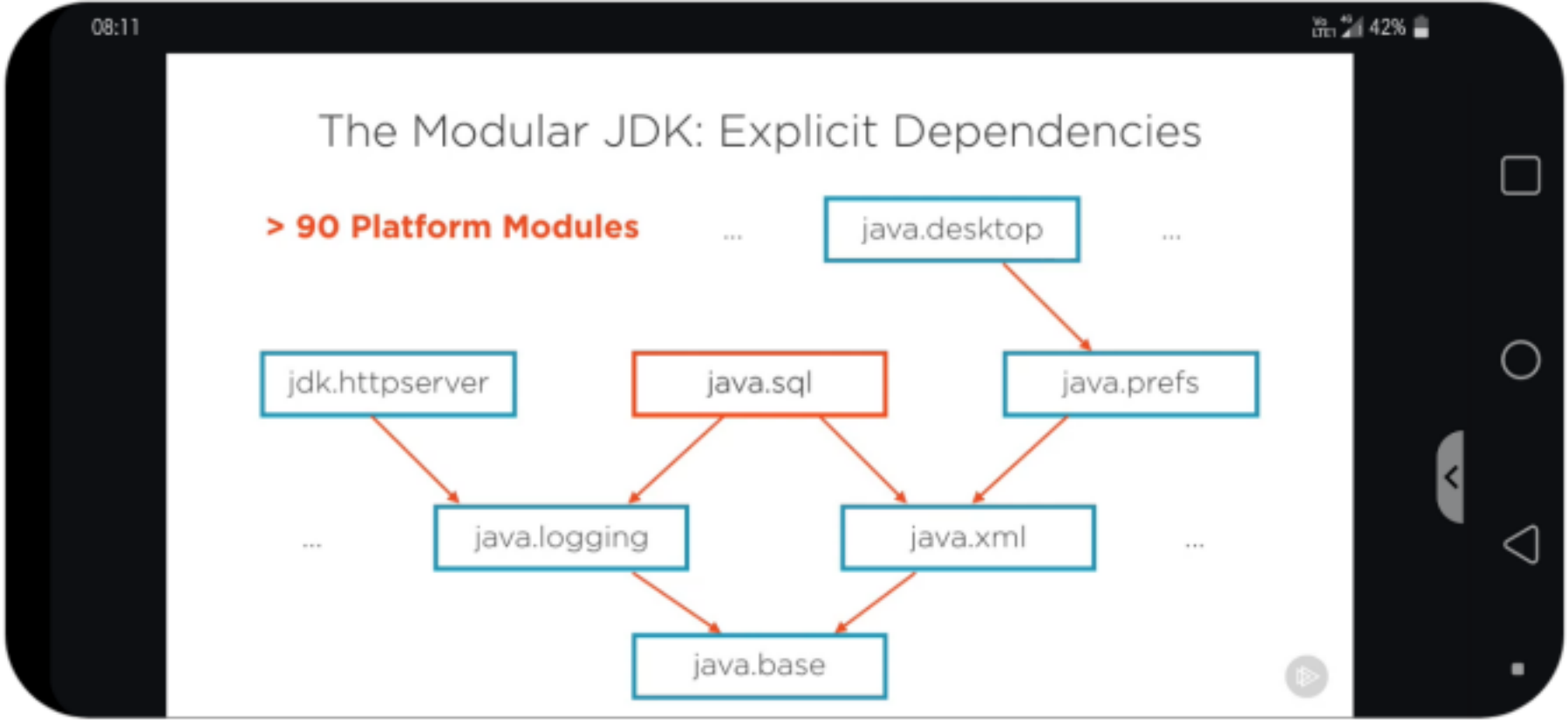
Modularity:

Provides too strong encapsulation. Modules has a name, it groups related codes, and is self contained.

≡ ○ Java9 put jdk packages with

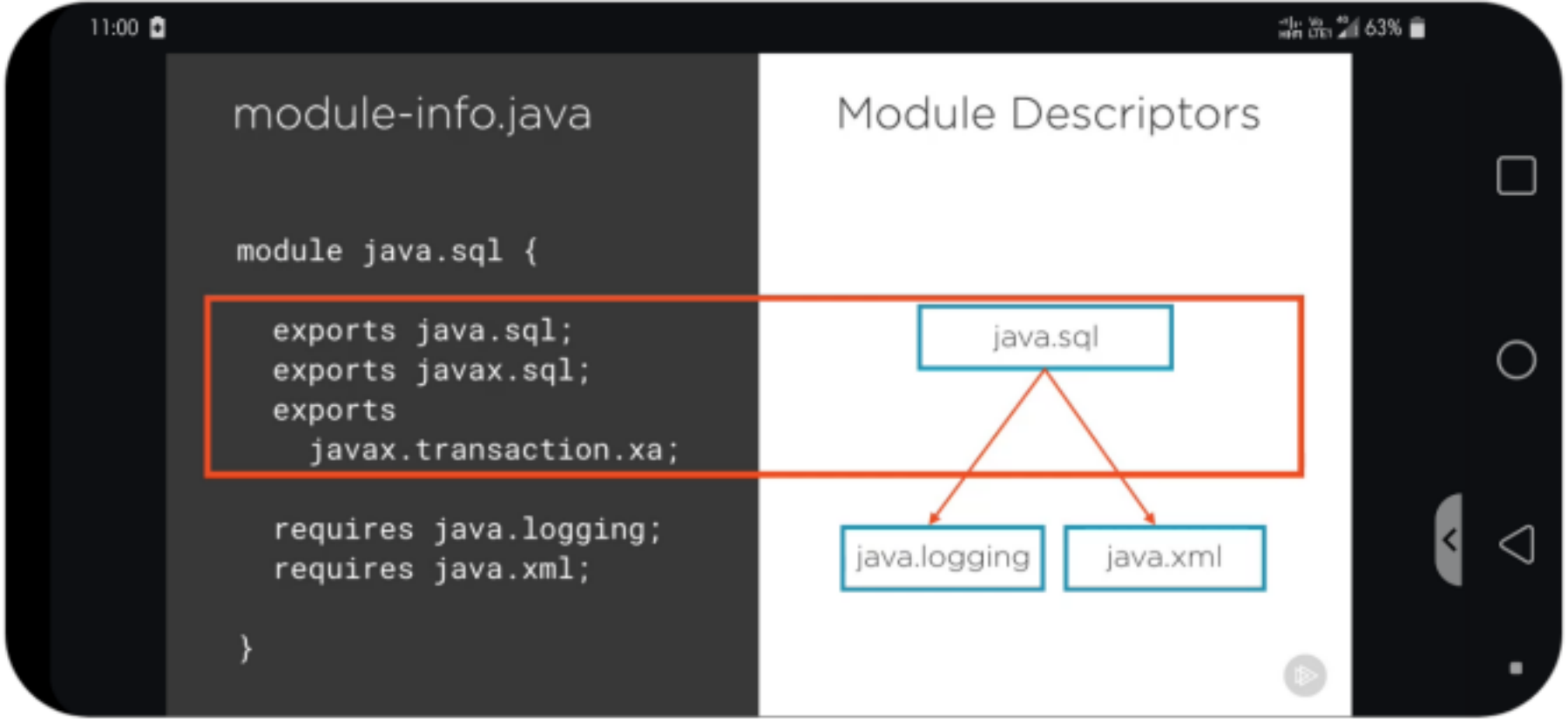
classes into multiple modules Also it enables developers to introduce modules in their java applications

- ≡ ○ Java modules helps java to evolve easily and move forward
- ≡ ○ Java9 provides Modules are like java.base, java.logging, java.xml, java.sql, java.httpserver, java.prefs, java.desktop
- ≡ ○ Advantages of this modularity is increased security, use only those required modules so class load time will be increased, easy to depricate legacy codes in jdk
- ≡ ○ Also if a module can not find the required modules, at the start up time of the jvm it throws error instead of during runtime as of with old jar based class path loading, where during runtime it throws error by saying required class does not exists



Module descriptor

- Each module will have its name, list of exported packages and list of required other modules. All other packages which are not exported are like private to the same module. We can export a module to qualified modules only like 'exports java.sql to javafx.graphics'

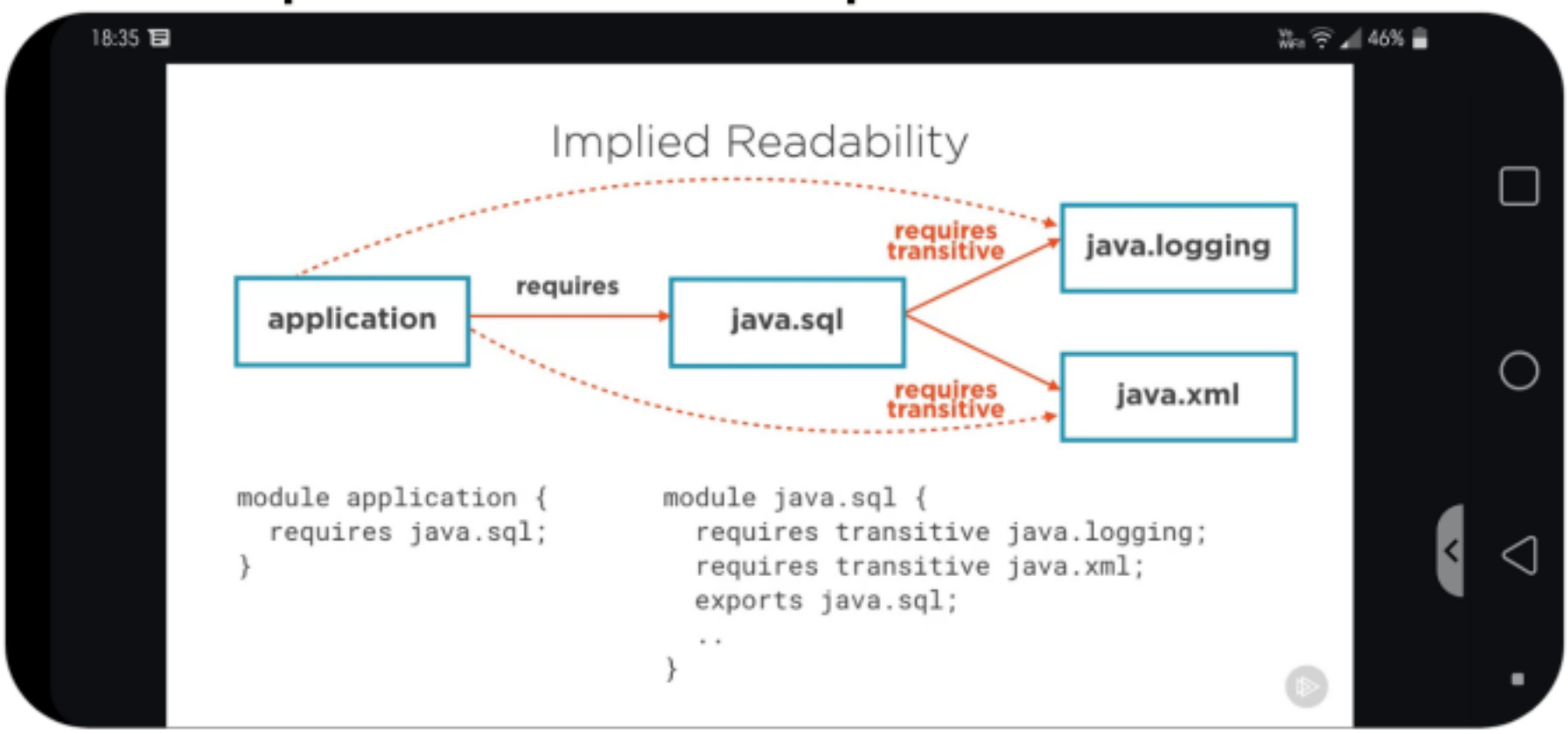


- Java 9 provides few command line queries like java --list-modules, java --describe-module java.sql
- There will be issues while migrating from java 8 classpath based app to java 9 module based app, related to usage of encapsulated packages as this wont be available for the existing java 8 code. There are few ways to overcone this
- We can identify such issues by using a tool called jdeps

Transitive module

- By default a module can not use

other modules which are used by its required module. In that case we can have transitive require optuon in the required module



JShell and REPL

- JShell is the REPL for java language.
- REPL: Read(type in code), evaluate(execute the code), Print(see result), Loop(interactively refine)
- Why Jshell: no need of main method for testing, direct feedback without compilation, easy to explore new java api's

- JShell provides lots of command to type and execute java codes

Collection factory method

- List.of(1,2,3) - provides immutable collection
- Set.of("value1")
- Map.of("key","value")
- Map.ofEntries(Map.entry("key","value")
- Iteration order will be guranteed only in list

Stream API improvements

- New stream methods are added like takeWhile, dropWhile, ofNullable
- New improvements on Collectors and Optionals API

Language improvements

- Improved try with resources
- Better generic type for anonymous class
- Private interface methods for

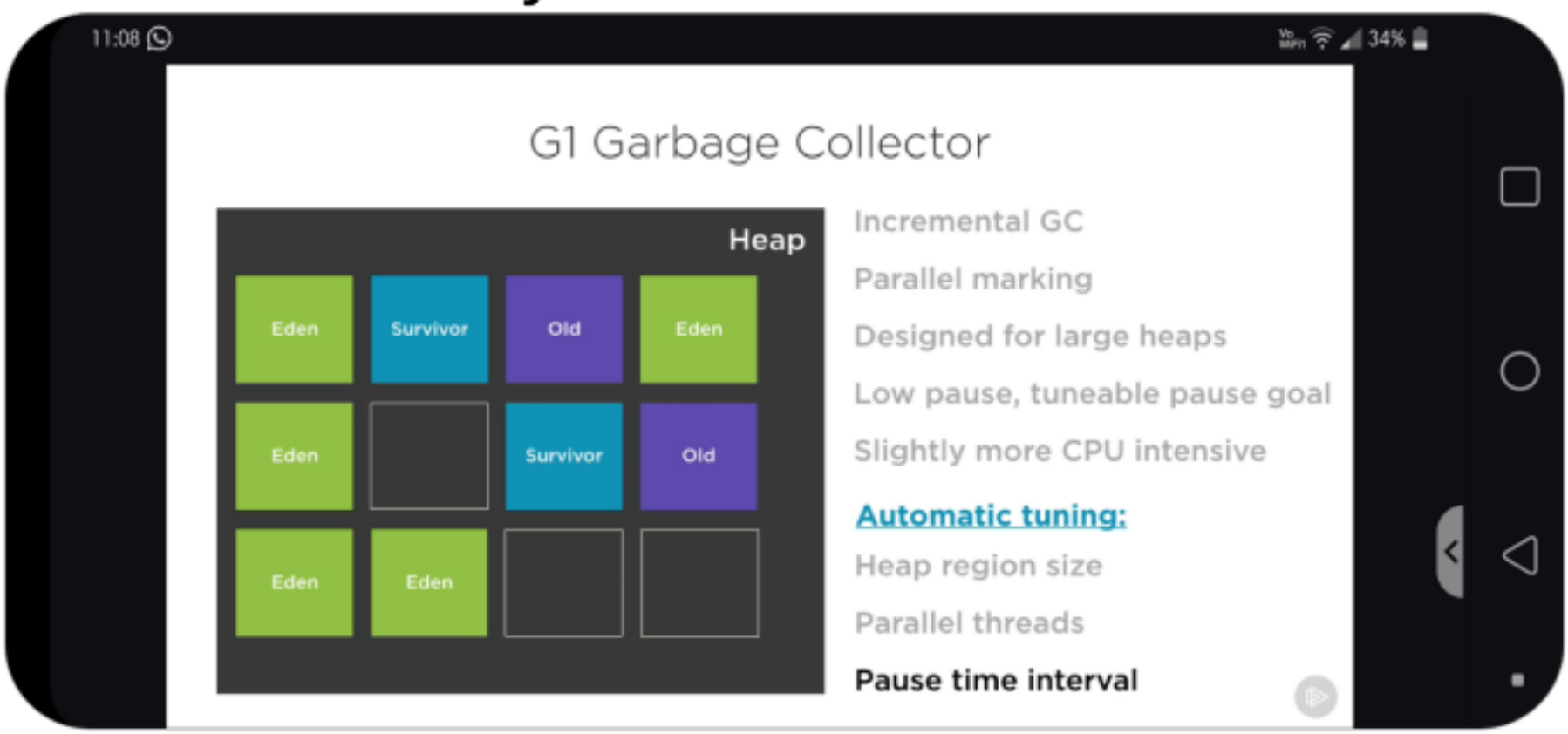
default methods introduced in java 8

==========

Java 10

- Java relaeses Long Term Support(LTS) version for every 3 years and non LTS version for every 6 months
- So in production its better to use LTS releases and in local development we can experimemt with LTS relaeases
- Currently java 8, java 11 are the LTS releases
- Java 10 provides Local Variable Type Inference support, where we can use the keyword var instead of actual type. Ex: var authors = books.getAuthors();
- This has some limitations also where we can not use var like in aninymous class ext

- G1 parallel full garbage collector in java 10 instead of serial full GC as of in java 9



Java 11

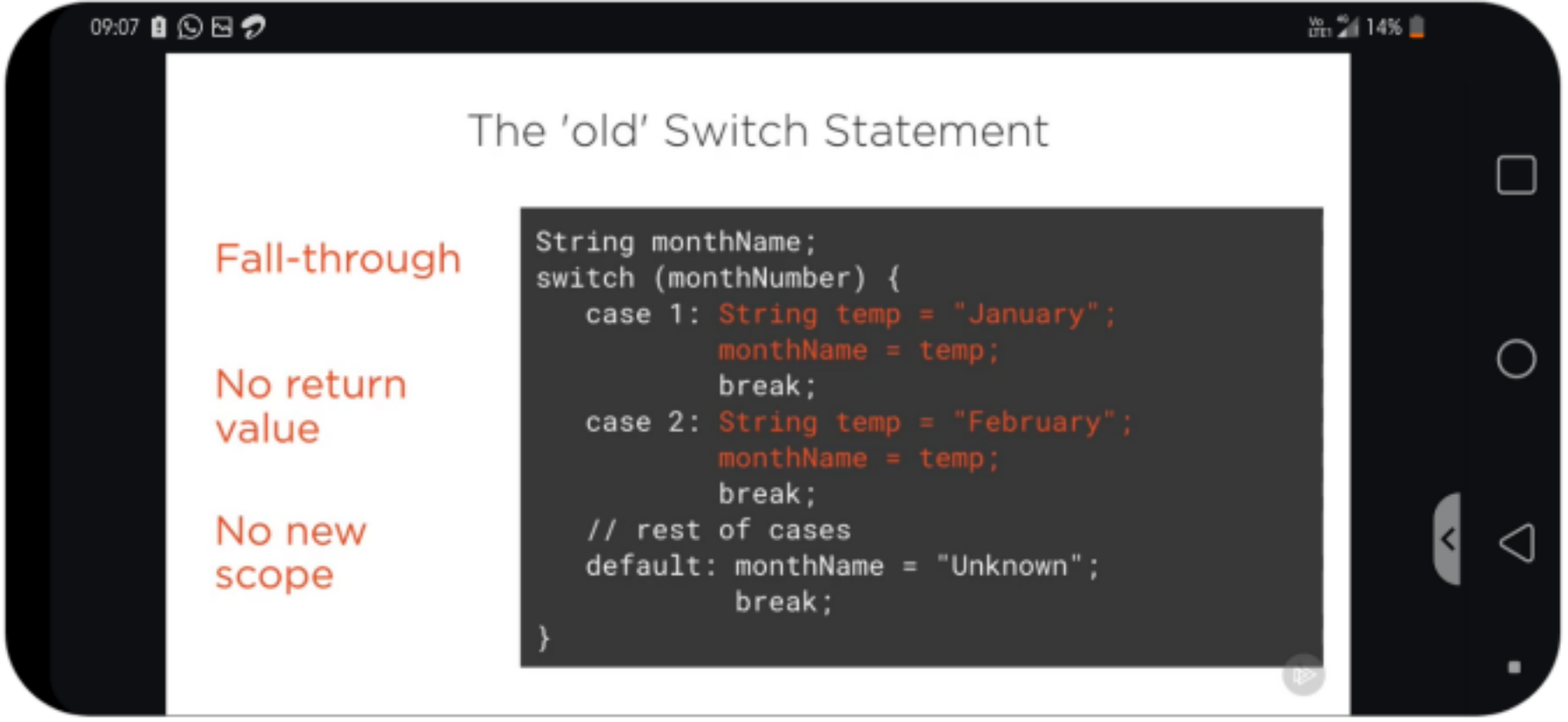
- In java 11 most of the old technoligies are cleaned up and deprecater like applets, javaFx which is moved out to openJFx and new things are added like new HttpClient API, running source code directly without compiling and 2 new GC
- From java11, we can not use oracle jdk in production, as its no

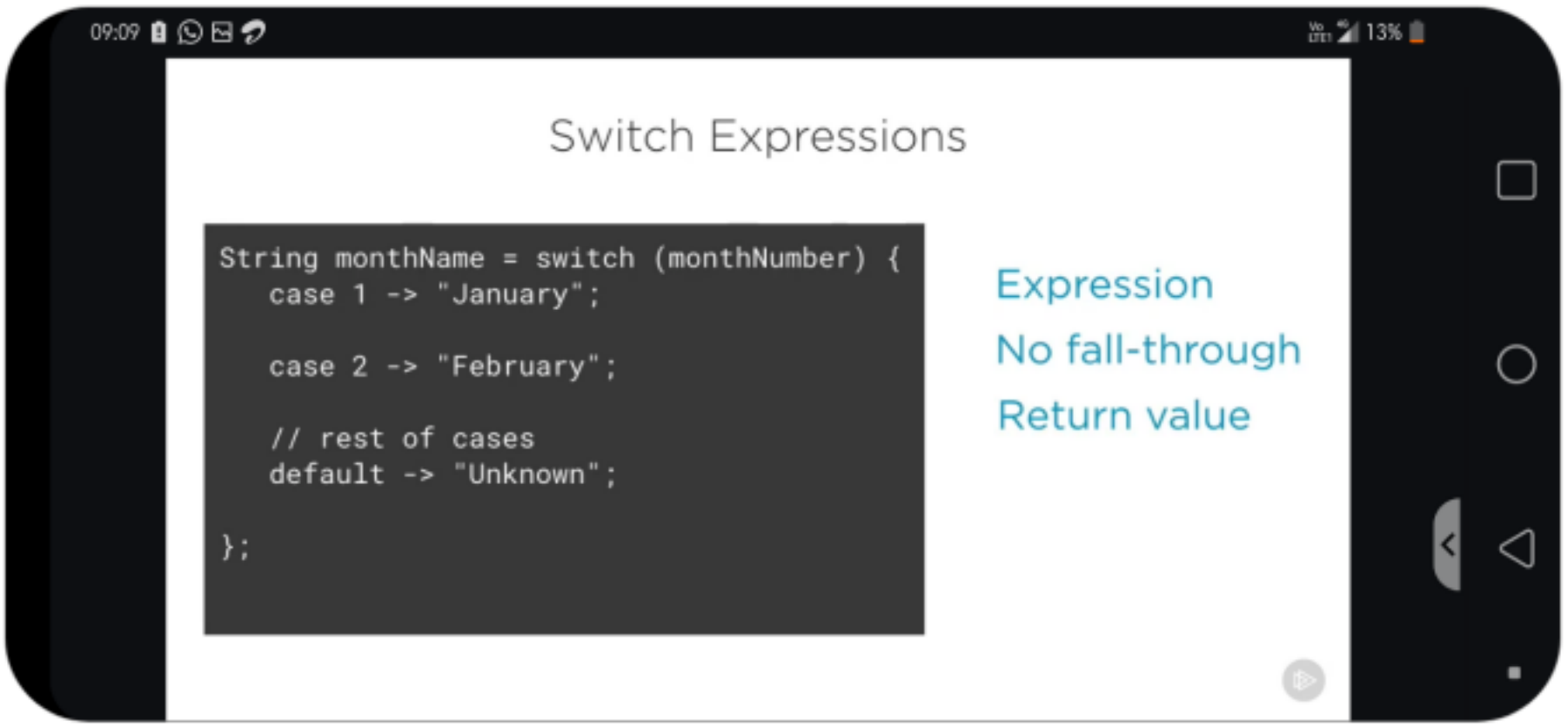
more free licence. We need to use open jdk instead

Java 12

- Teeing collector: we can apply two collectors to a single stream of data, and then combine results from both collectors to a single result.
- Ex: Collectors.teeing(c1,c2,combine)

- Preview features: Switch Expression

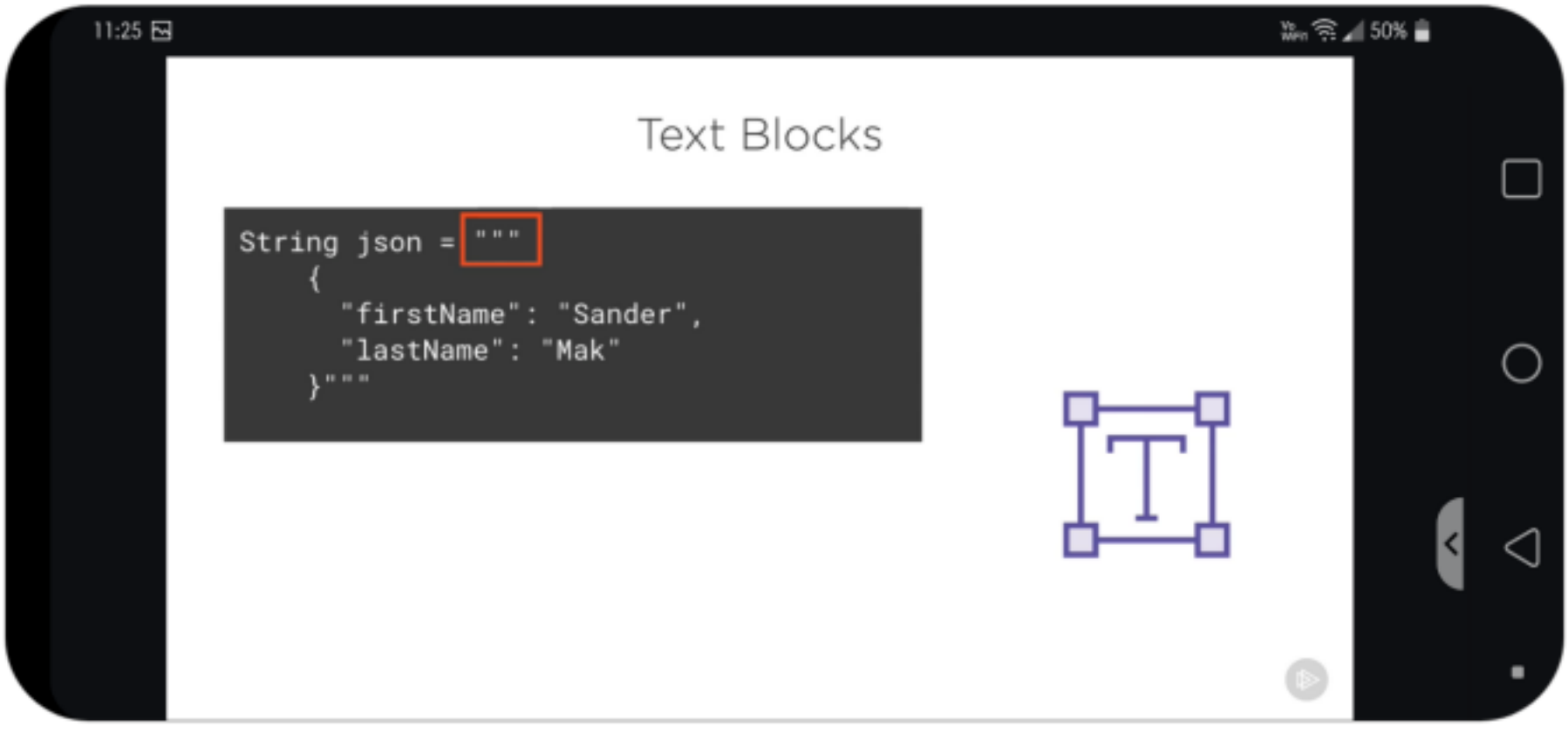




- Support for Micro benchmark in java 12 through JMH(Java Micro-Benchmarking Harness) tool. This tool helps in finding performance of each piece of code

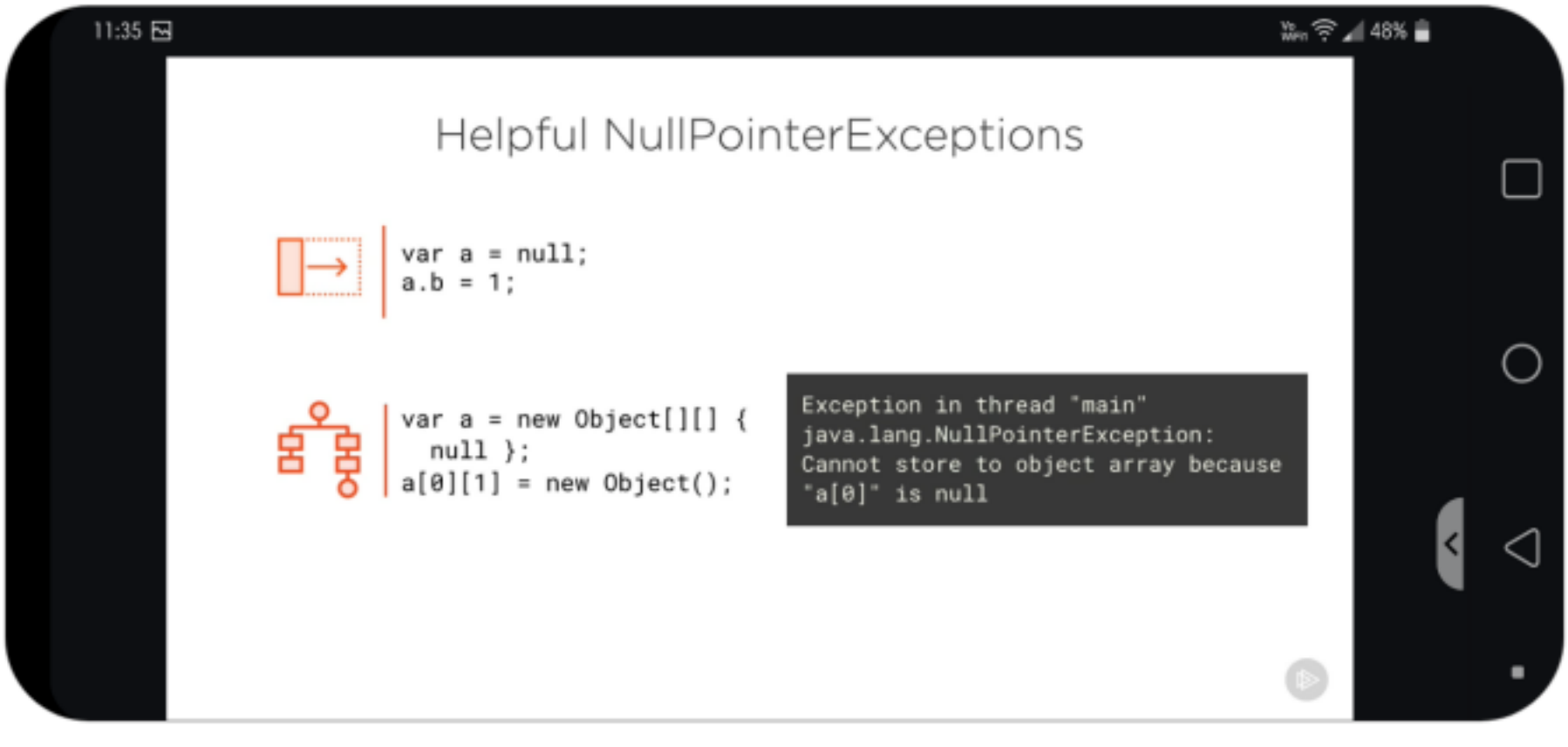
Java 13

- Switch expression update
- Text block preview feature: to have multi lines string data

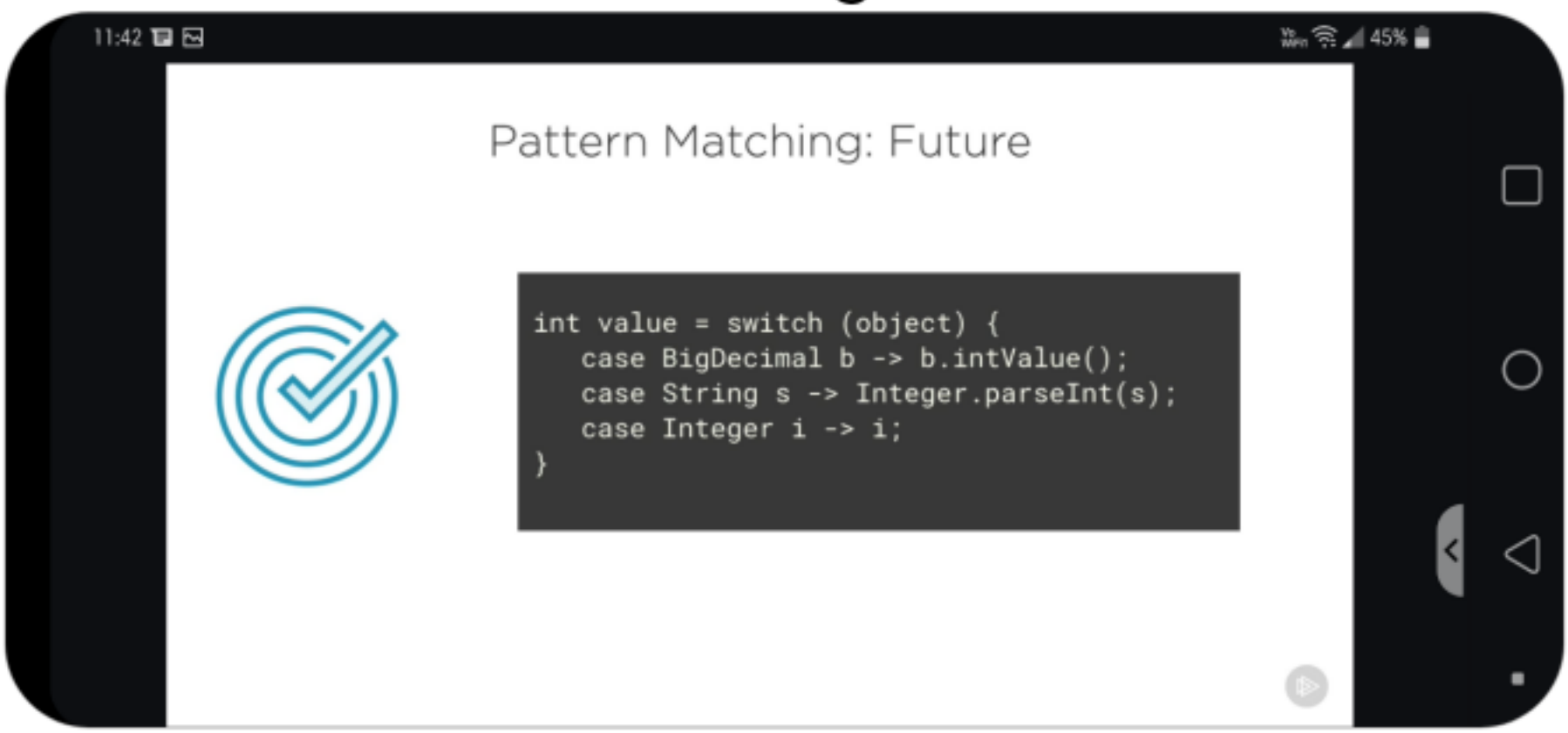


Java 14

- Helpfull Null pointer Exception: which provides more details on this exception.



- Many preview features are updated or introduced like switch expressions, text blocks, pattern matching, packaging tool
- Pattern Matching



- Records: in java 14 creats dto classes in one line.

Public record Person(String name, int age, String hobby);

==========

Whats new in Spring 5

- Support for few of the things are discontinued like velocity, Guava etc And Tiles 3 is default now

- In spring 5, JDK 8 is the minimum requirement and it also supports JDK 9 modular system. Java EE7, JUnit5 and Hibernete 5 are the default one

Spring core changes

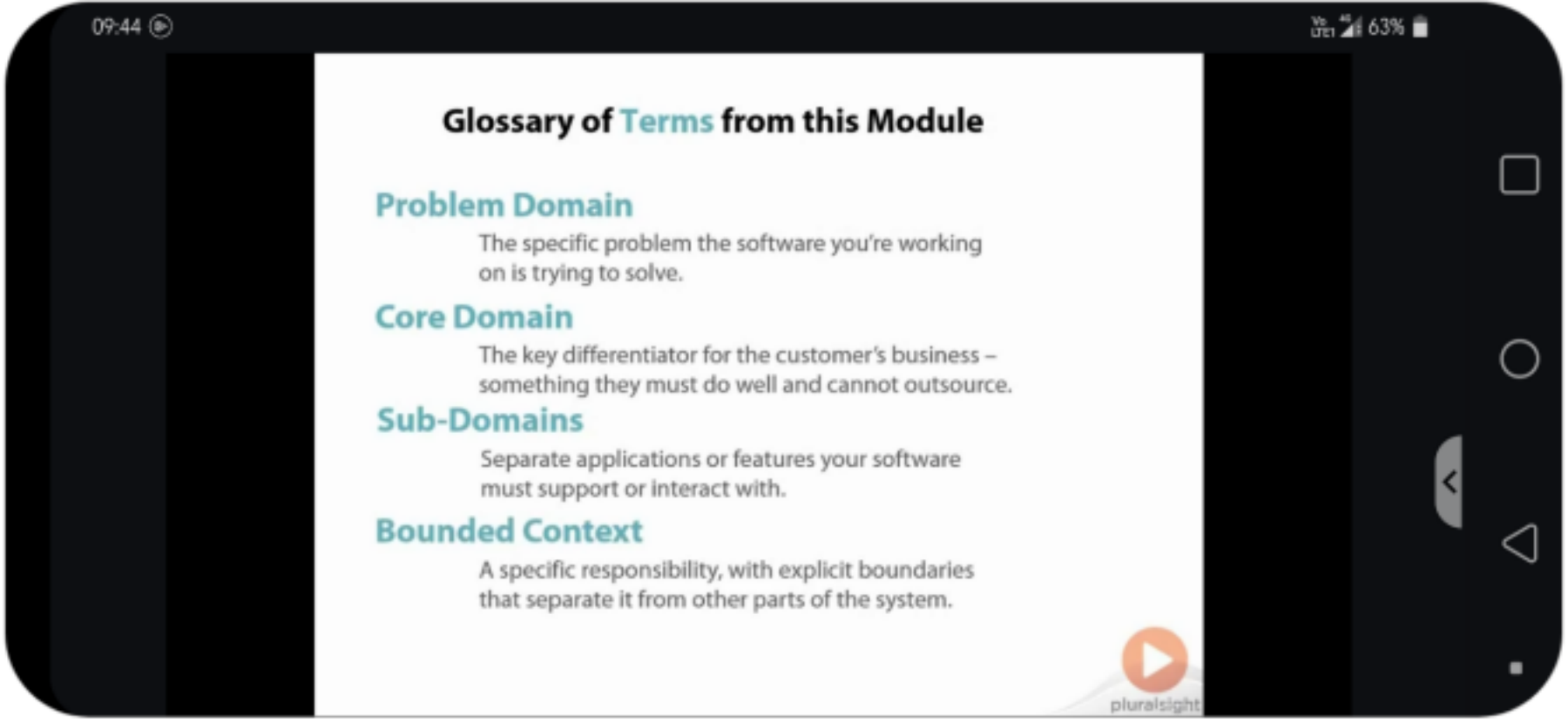
- Nullable arguments: @NotNull, @Nullable, @NonNullApi
- Default methods: PostConstruct default methods in interfaces
- Logging enhancements

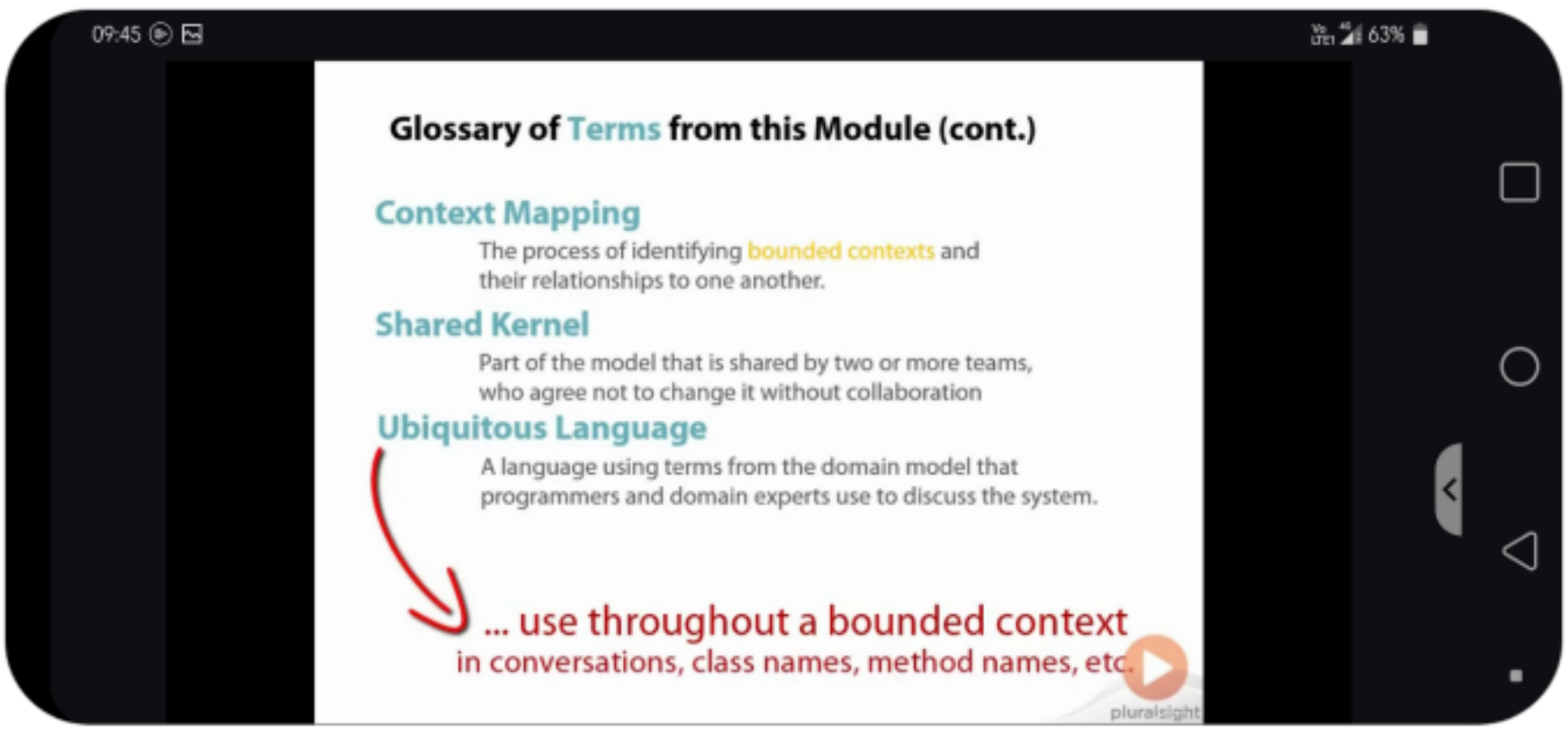
==========

Domain Driven Design

Domain driven design fundamentals - a pluralsight course

Modelling problems in software:





Elements of a domain model:

Glossary of Terms from this Module

Anemic Domain Model

Model with classes focused on state management. Good for CRUD.

Rich Domain Model

Model with logic focused on behavior, not just state. Preferred for DDD.

Entity

A mutable class with an identity (not tied to it's property values) used for tracking and persistence.

Immutable

Refers to a type whose state cannot be changed once the object has been instantiated.

Glossary of Terms from this Module

Value Object

An immutable class whose identity is dependent on the combination of its values

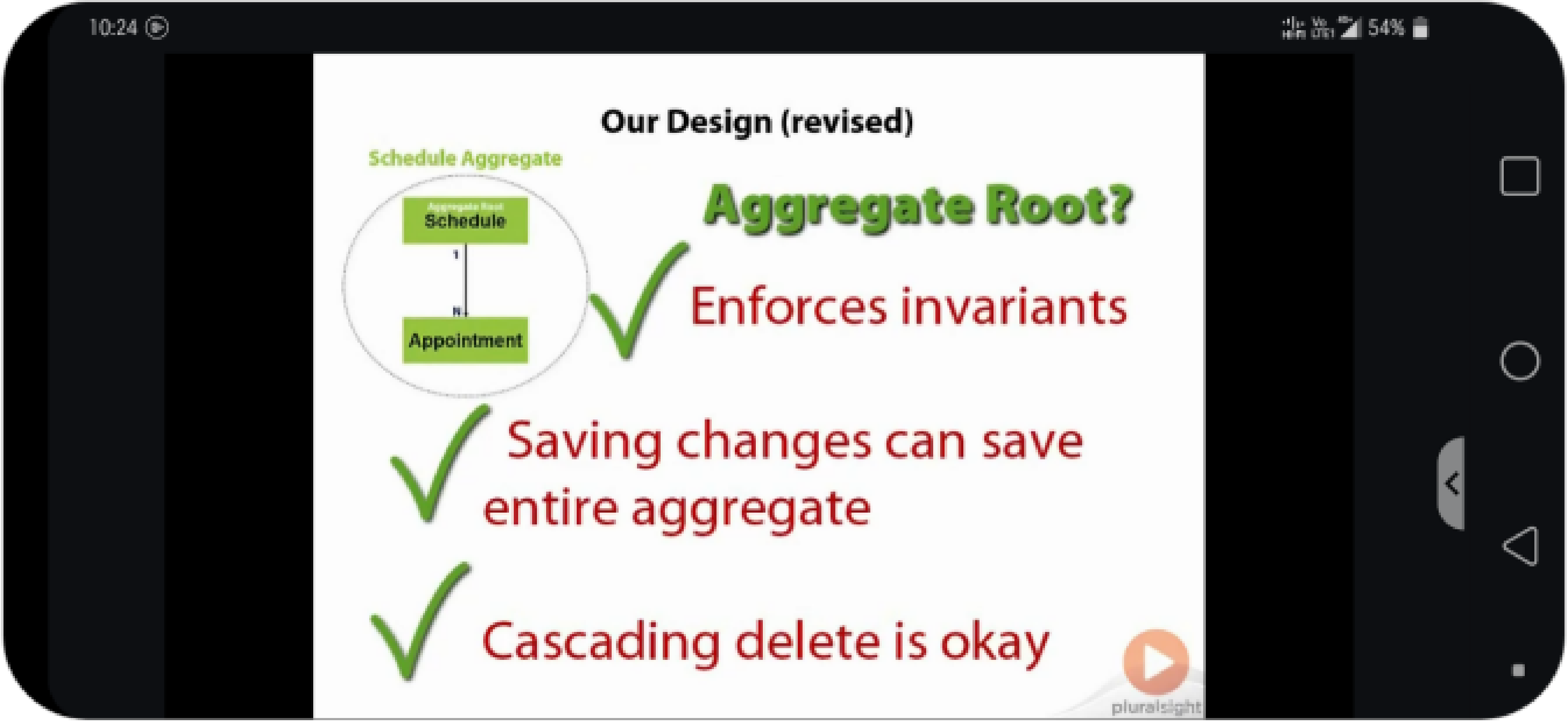
Services

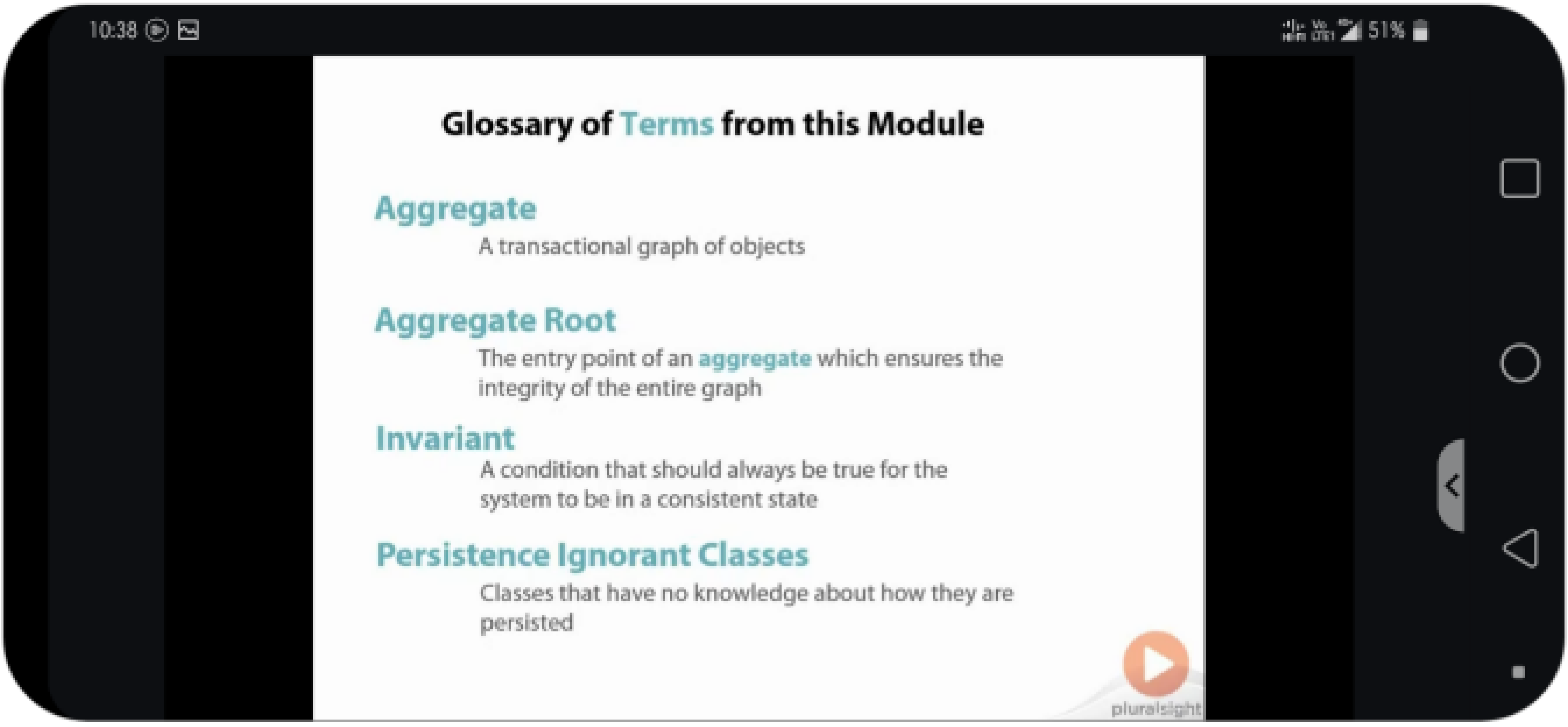
Provide a place in the model to hold behavior that doesn't belong elsewhere in the domain

Side Effects

Changes in the state of the application or interaction with the outside world (e.g. infrastructure)

Aggregates in DDD:
Aggregates are group of related entities or value objects that work together in a transaction.


10:24
54%
Our Design (revised)
Schedule Aggregate
Aggregate Root
Schedule
1
N
Appointment
Aggregate Root?
Enforces invariants
Saving changes can save entire aggregate
Cascading delete is okay
pluralsight


10:38
51%
Glossary of Terms from this Module
Aggregate
A transactional graph of objects
Aggregate Root
The entry point of an aggregate which ensures the integrity of the entire graph
Invariant
A condition that should always be true for the system to be in a consistent state
Persistence Ignorant Classes
Classes that have no knowledge about how they are persisted
pluralsight