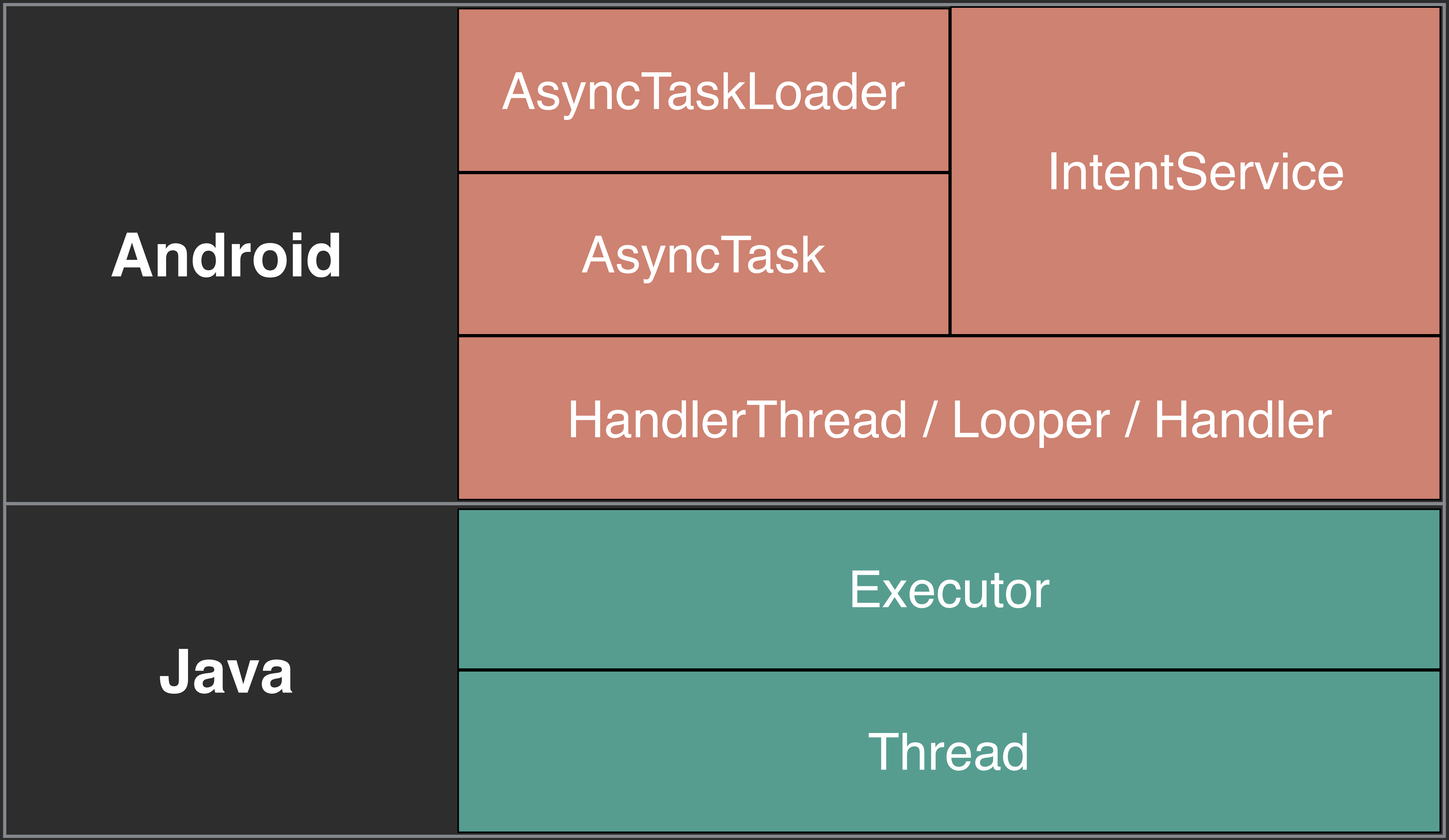
RxJava and Retrolambda

Making Android development more FUNctional

Async Programming

Standard Async Classes



Android

AsyncTaskLoader

AsyncTask

IntentService

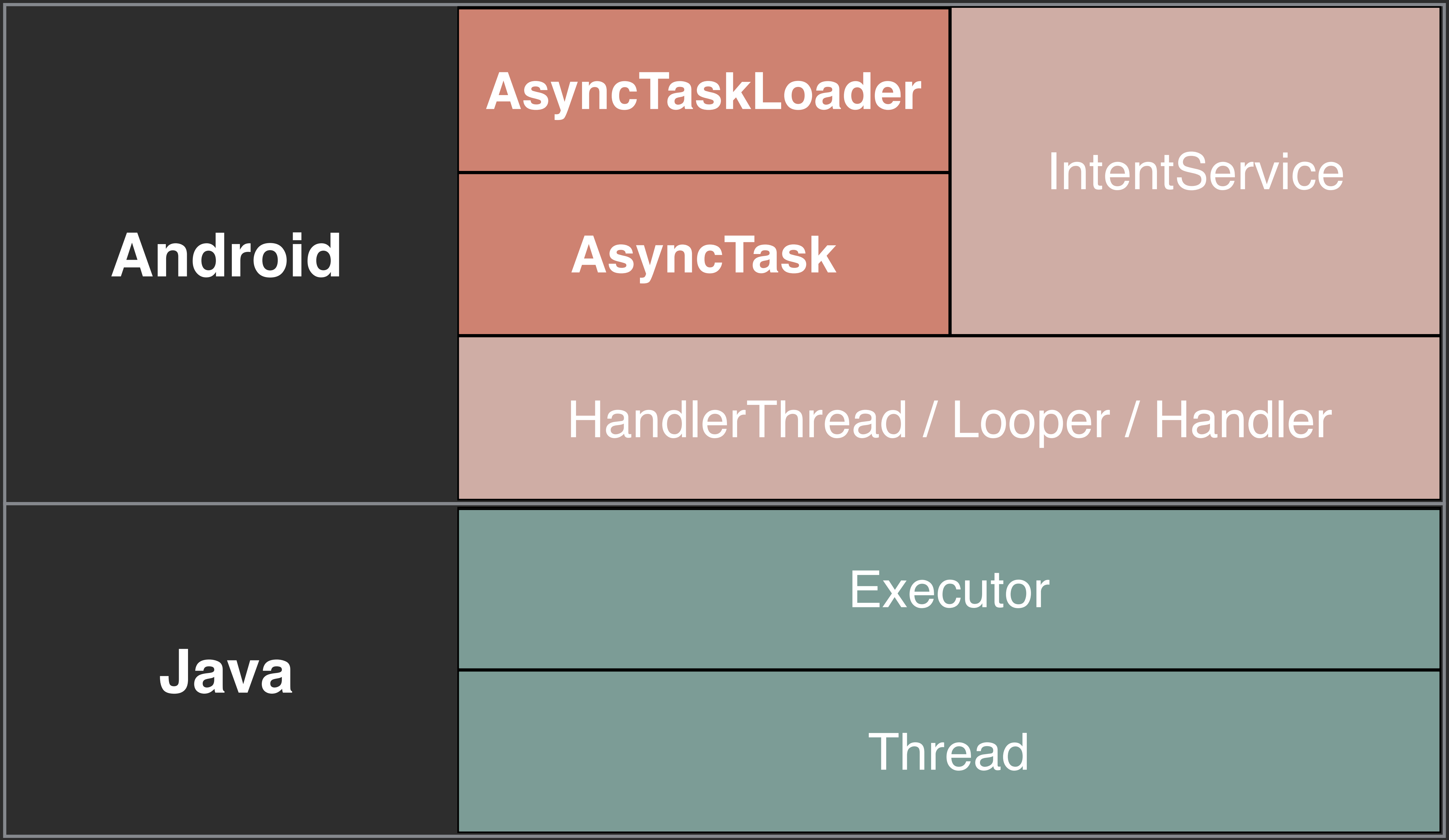
HandlerThread / Looper / Handler

Java

Executor

Thread

Standard Async Classes



Android

AsyncTaskLoader

AsyncTask

IntentService

HandlerThread / Looper / Handler

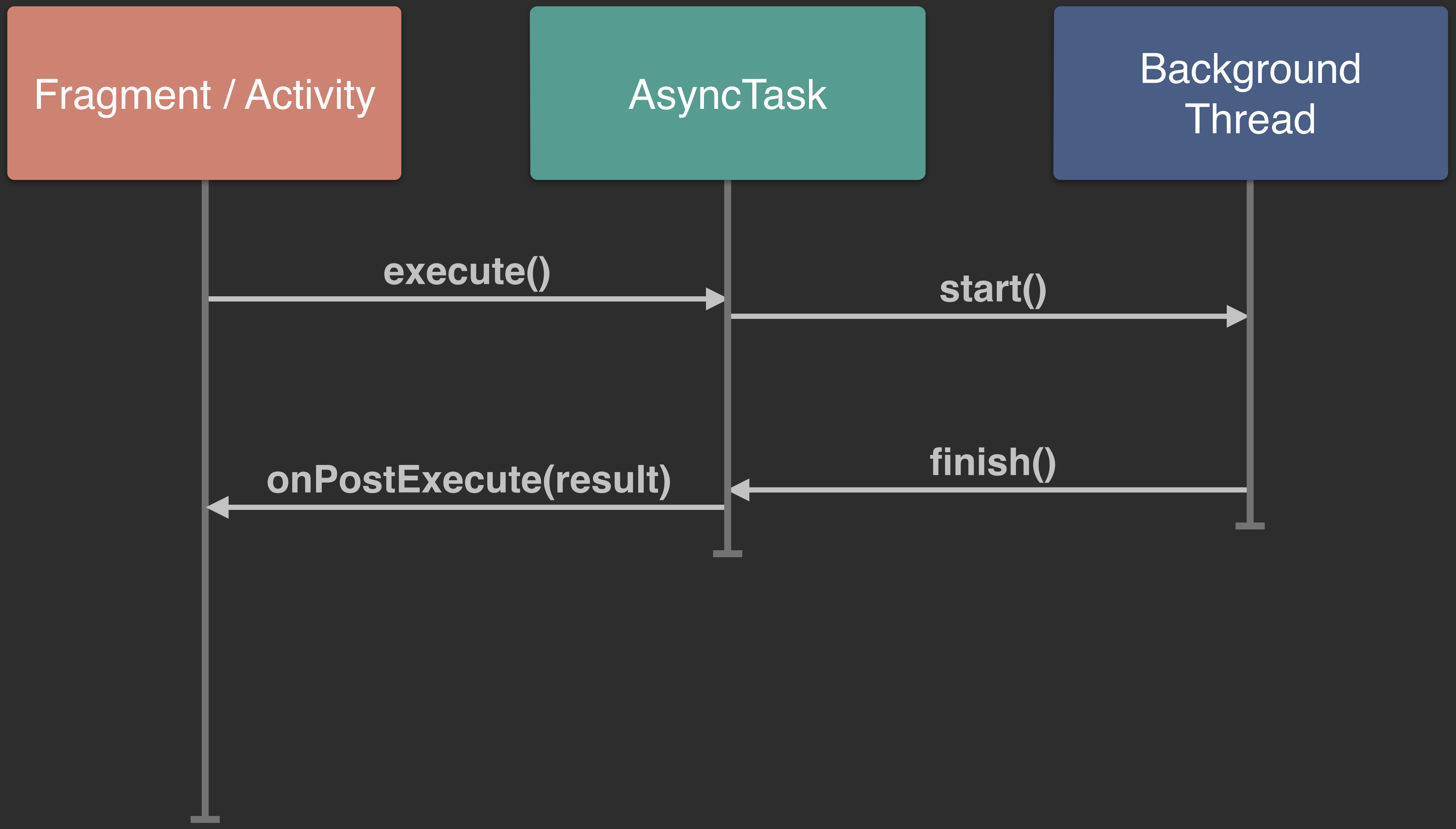
Java

Executor

Thread

AsyncTask

Looks easy



Fragment / Activity

AsyncTask

Background Thread

execute()

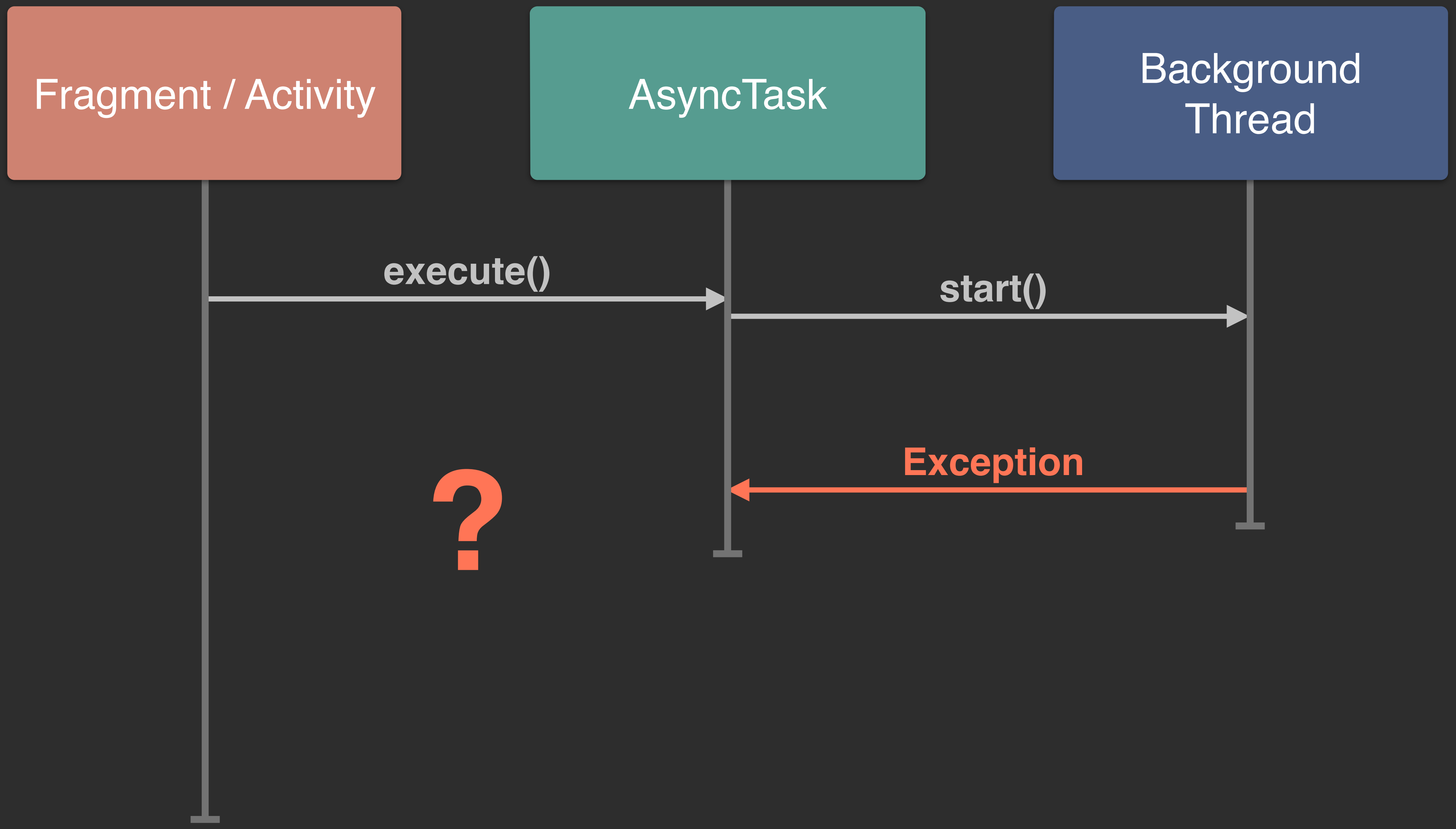
start()

finish()

onPostExecute(result)

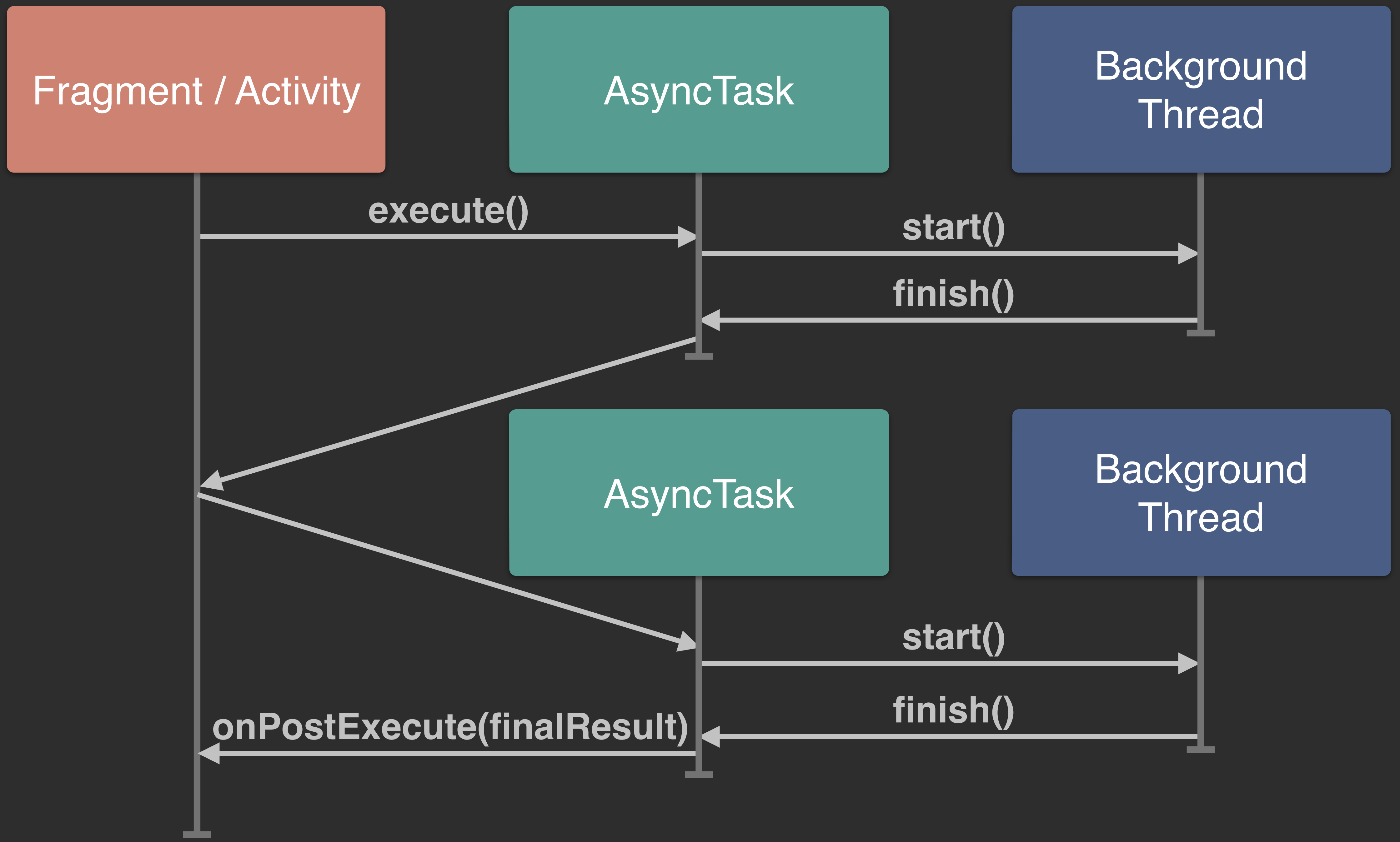
AsyncTask

No exception handling

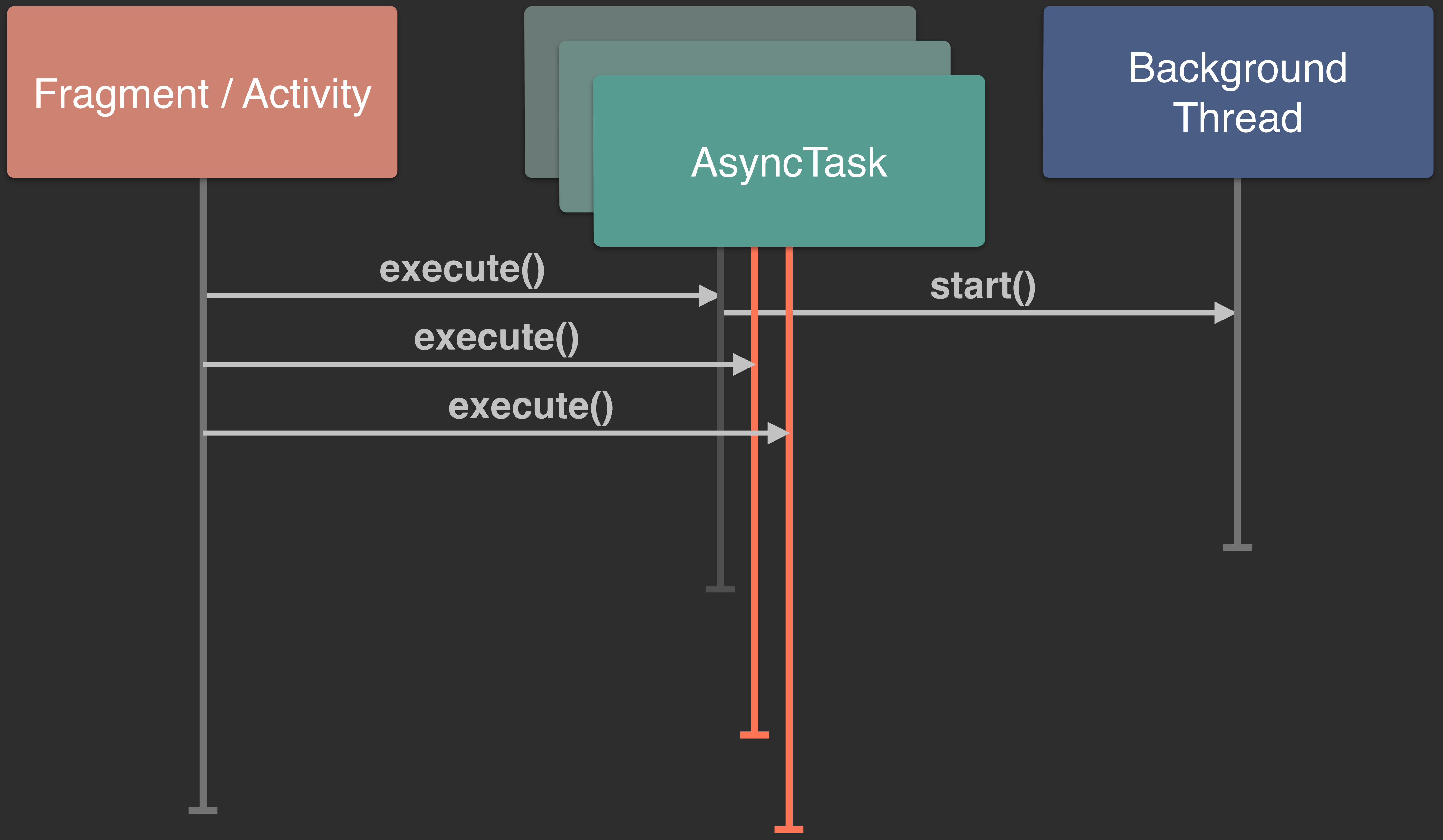


AsyncTask

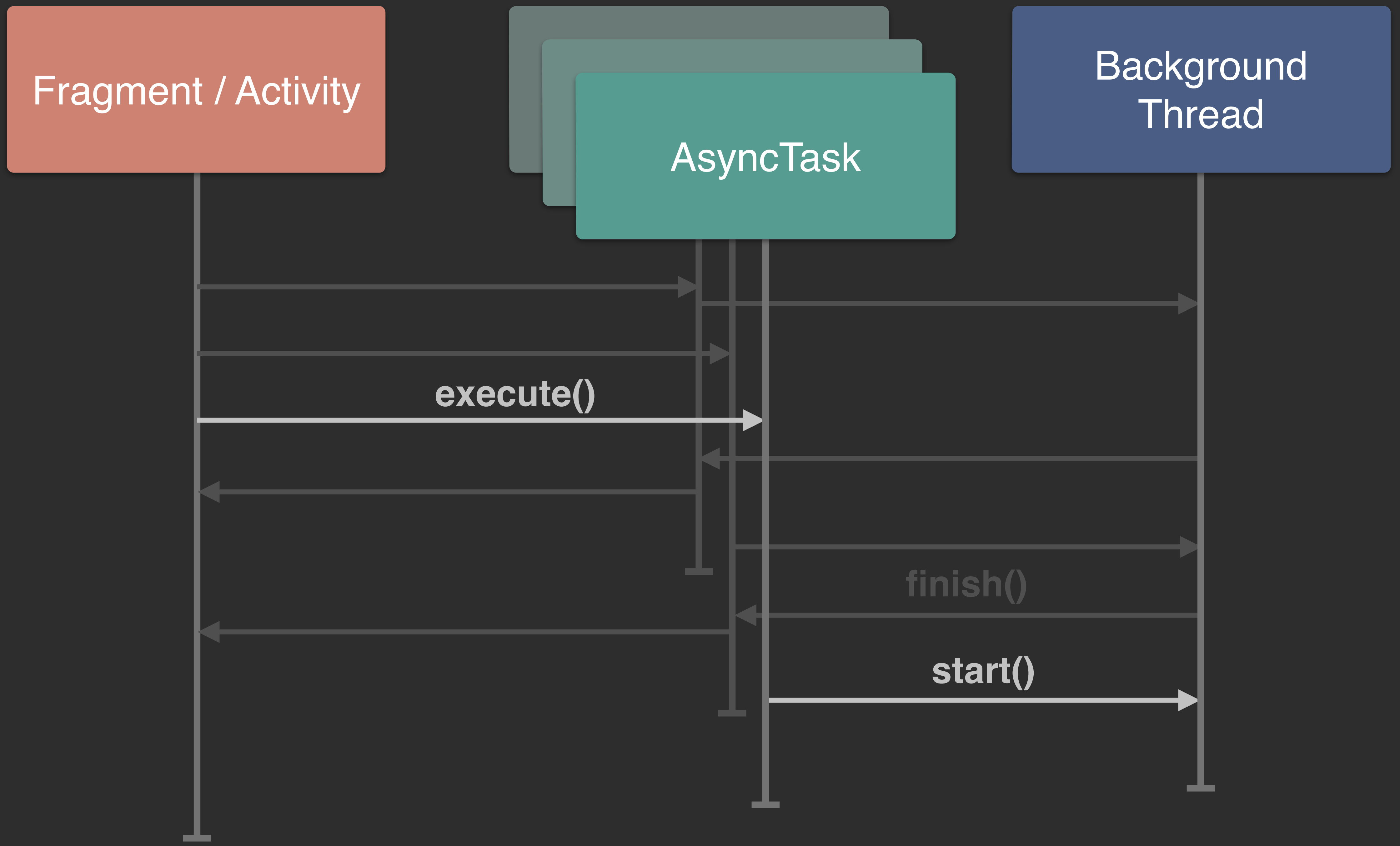
Chaining tasks



AsyncTask Serialized thread processing

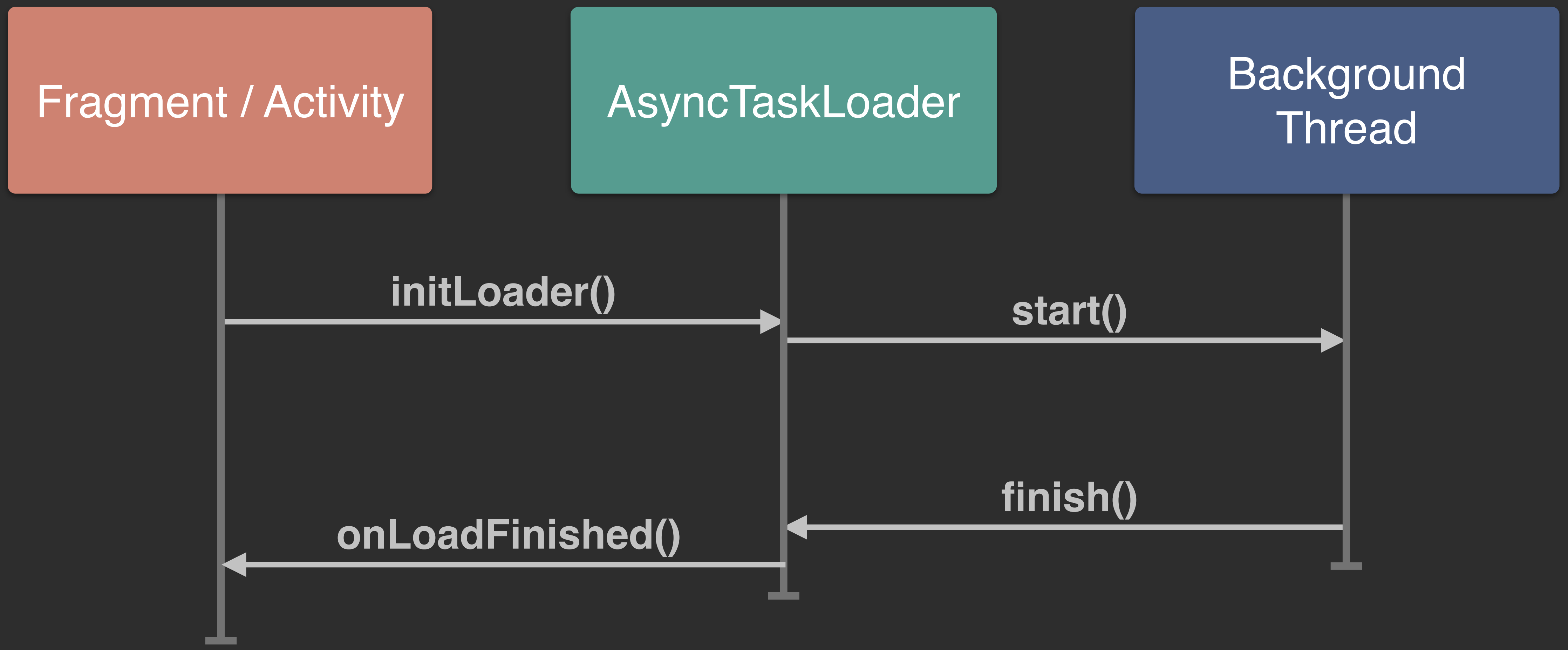


AsyncTask Serialized thread processing



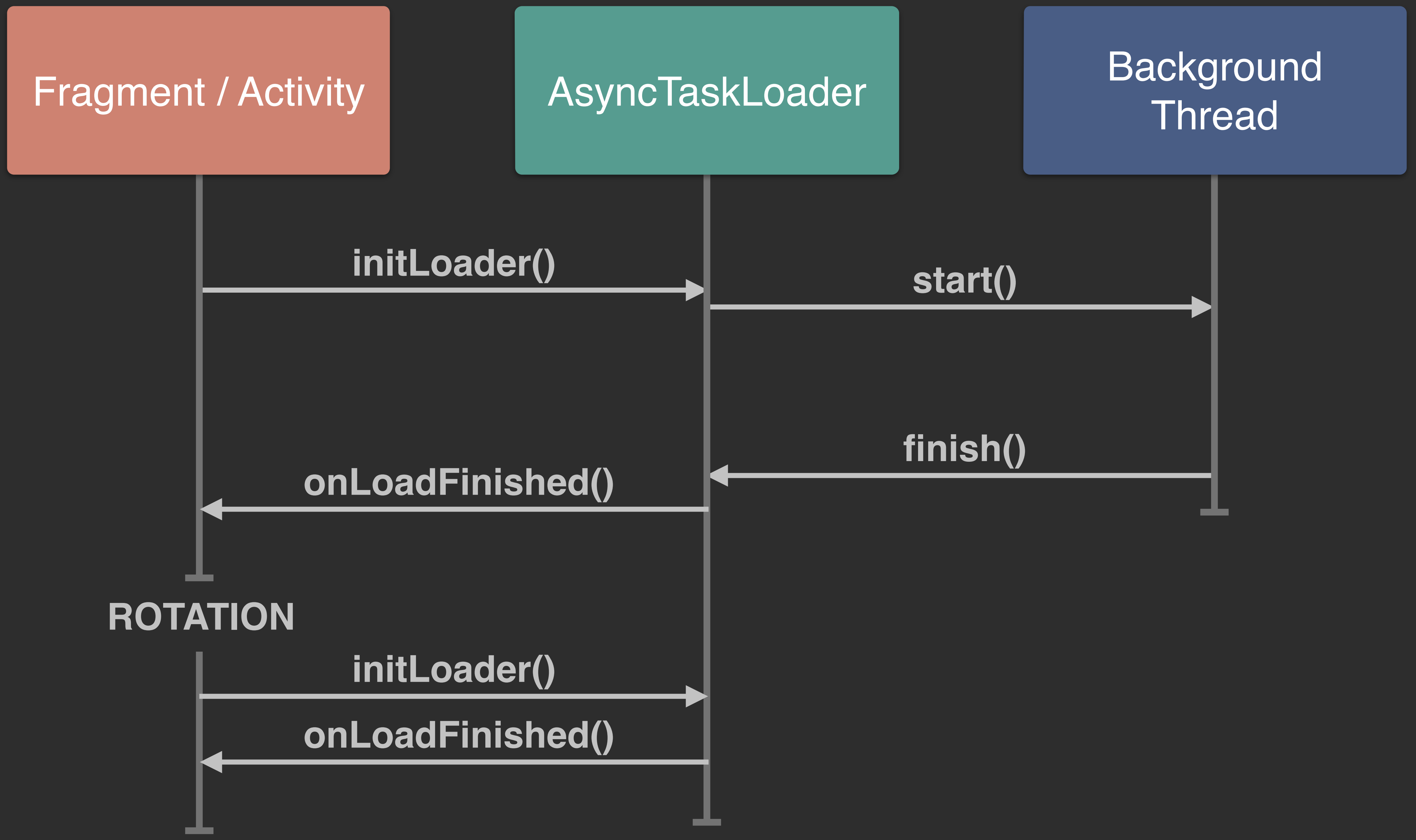
Loader

Caching



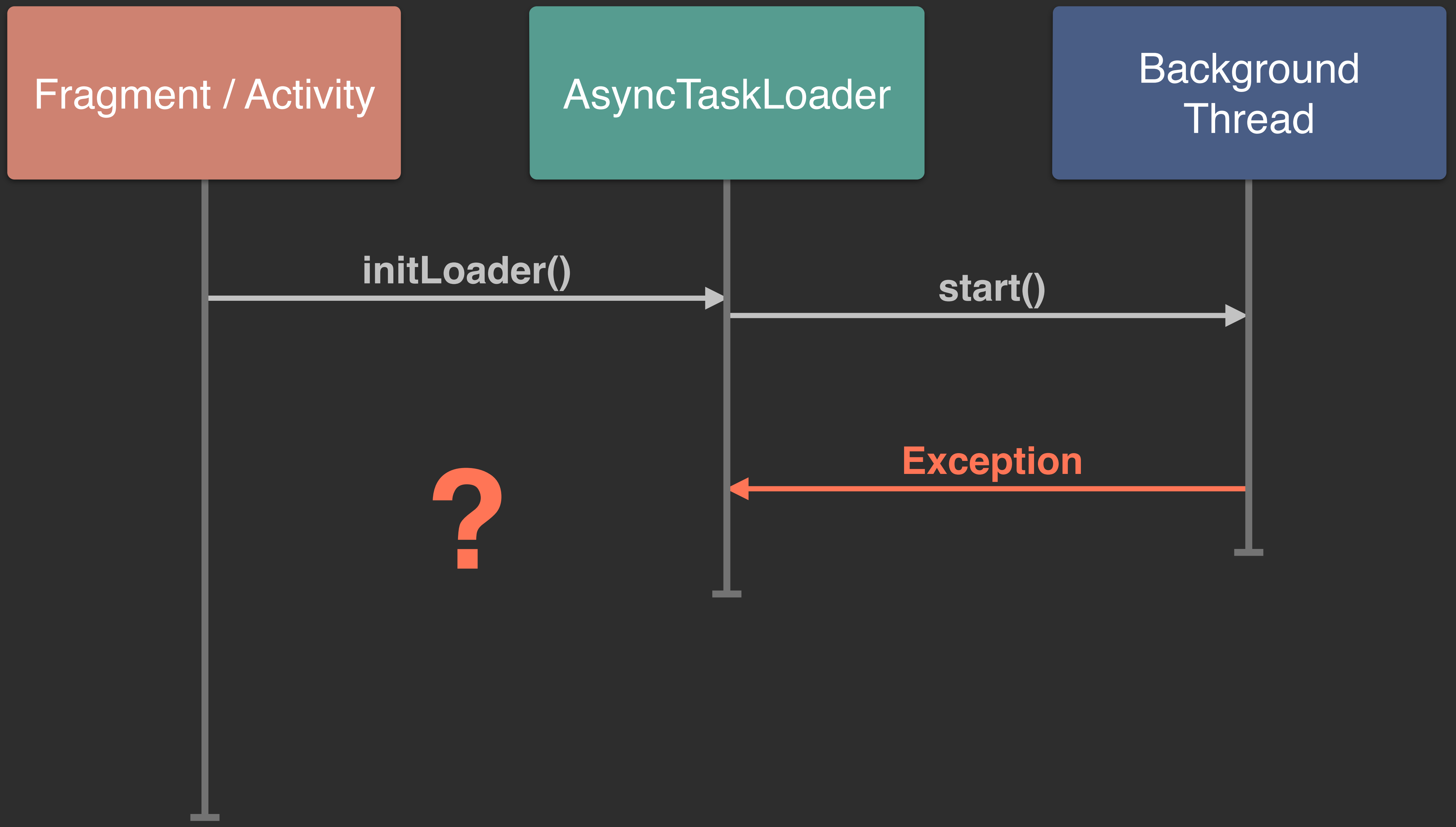
Loader

Caching



Loader

Exceptions



Fragment / Activity

AsyncTaskLoader

Background Thread

initLoader()

start()

Exception

?

Alternative: RxJava

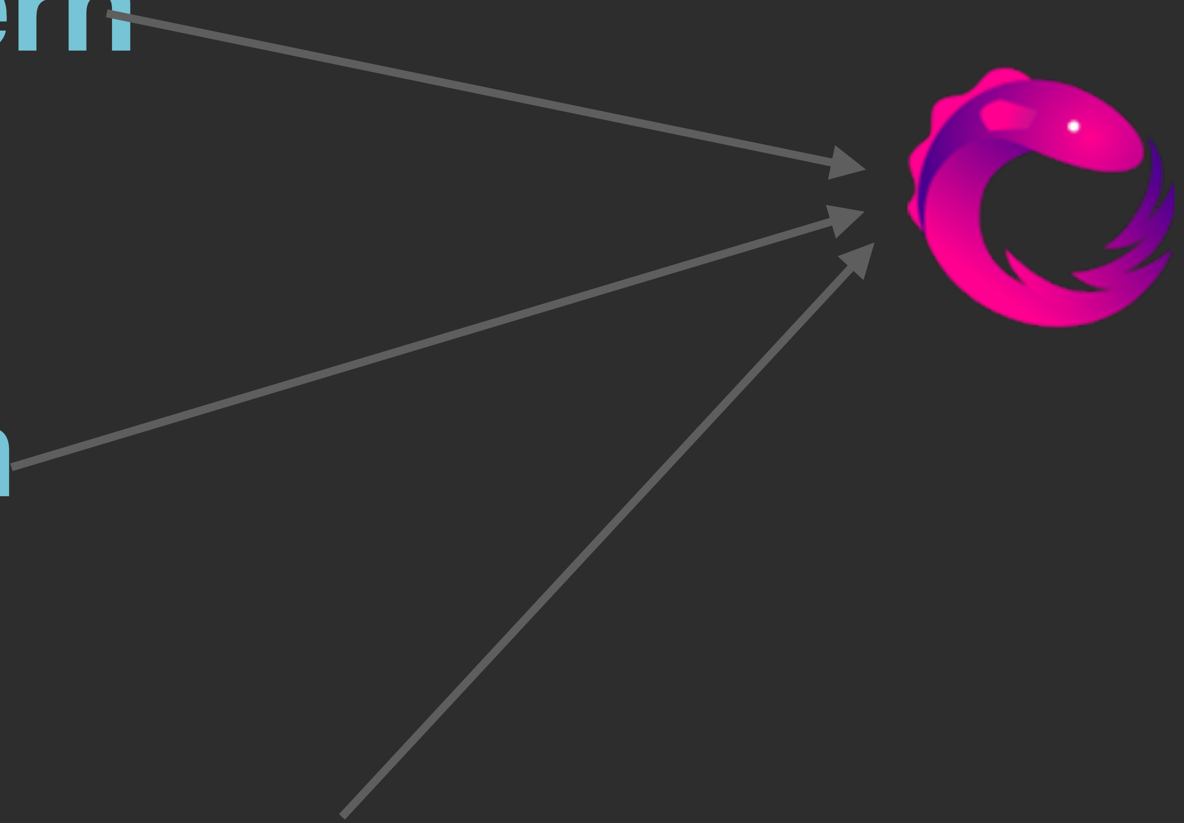
Functional Reactive Programming

Reactive Extensions

- Created by Erik Meijer at Microsoft
- Ported to Java by Netflix
- It’s everywhere
  - .NET, Java, Groovy, Scala, JS, Cocoa, etc.

Rx Concepts

- **Observer Pattern**
  - Reactive
- **Iterator Pattern**
  - Collections
- **Functional Programming**
  - Transformations



Reactive Programming

Not Reactive

Imperative

```
x = 2;
y = 3;
sum = x + y;
// sum == 5
```

Not Reactive

Imperative

```
x = 2;
y = 3;
sum = x + y;
// sum == 5

x = 4;
// sum == ?
```

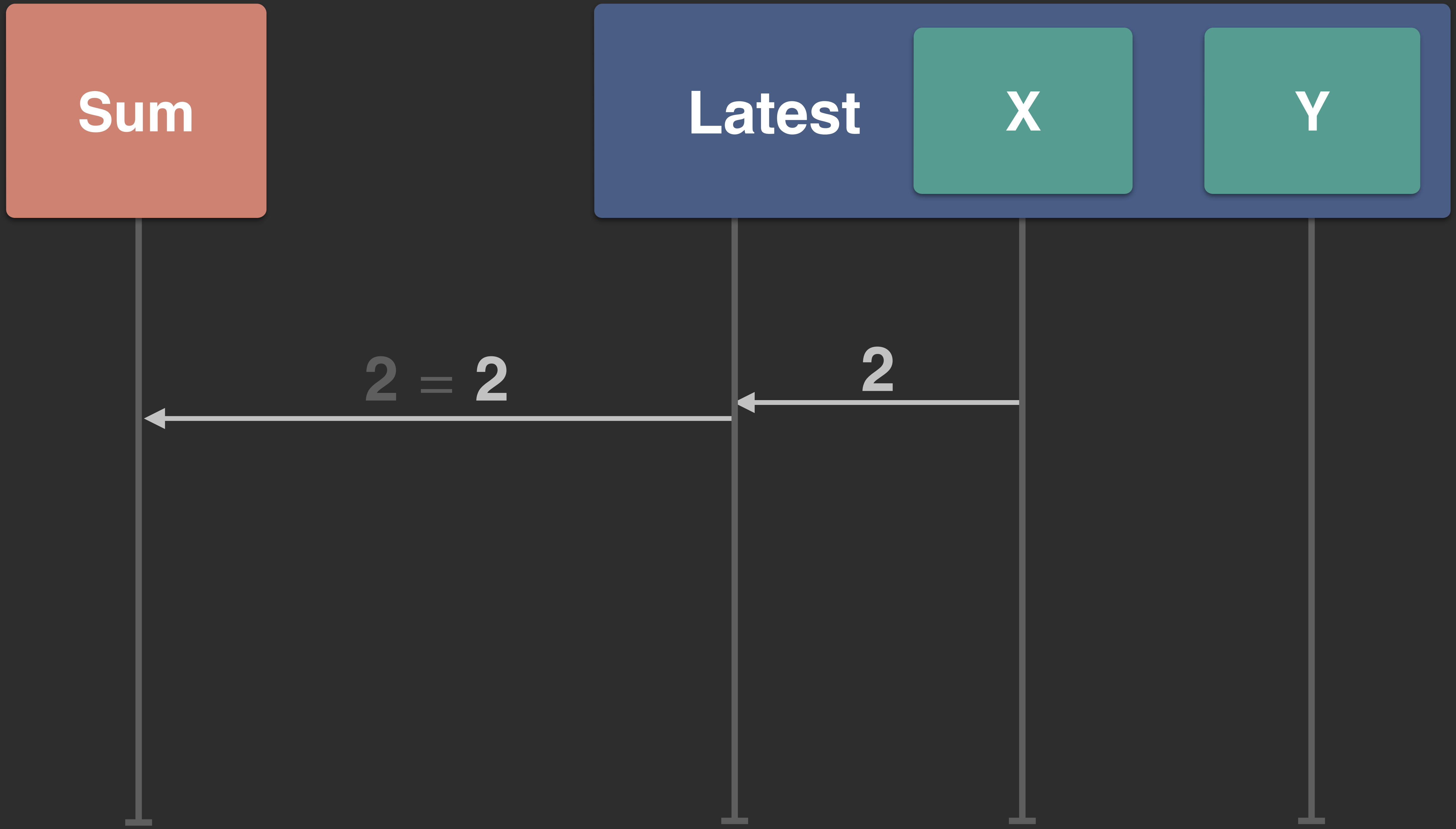
Not Reactive

Imperative

```
x = 2;
y = 3;
sum = x + y;
// sum == 5

x = 4;
// sum == still 5 of course
```

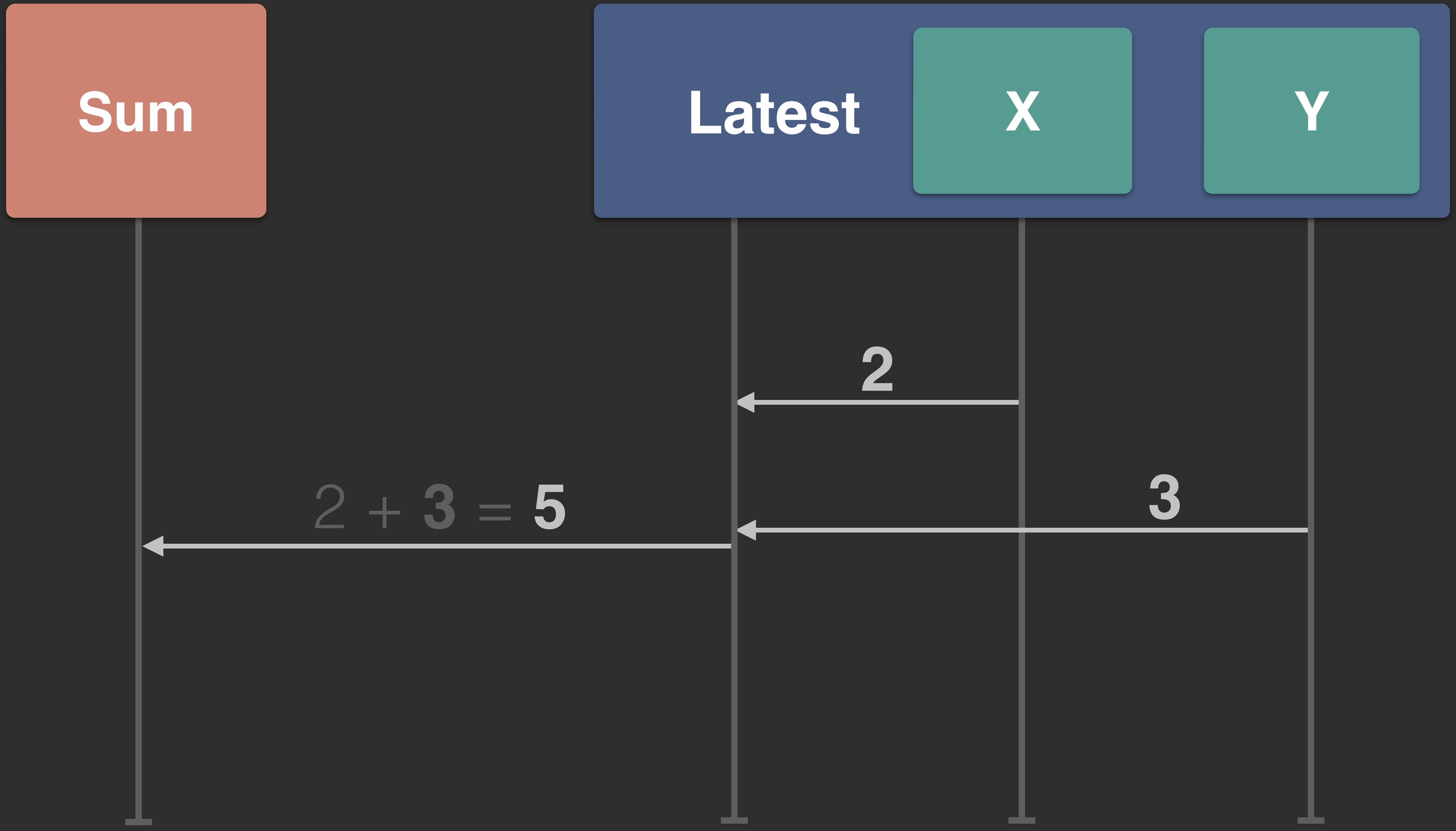
Reactive

Similar to a spreadsheet



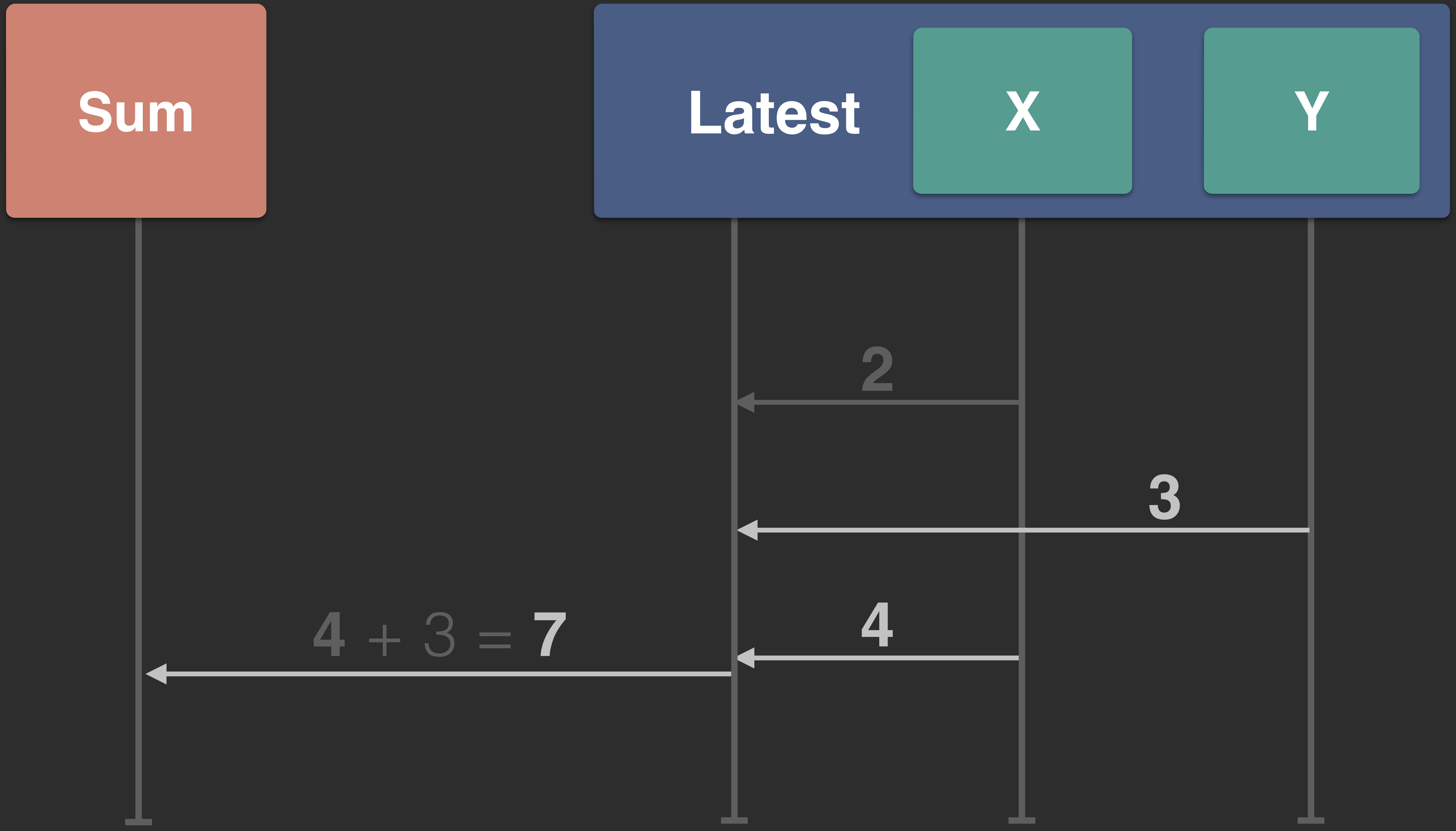
Reactive

Similar to a spreadsheet



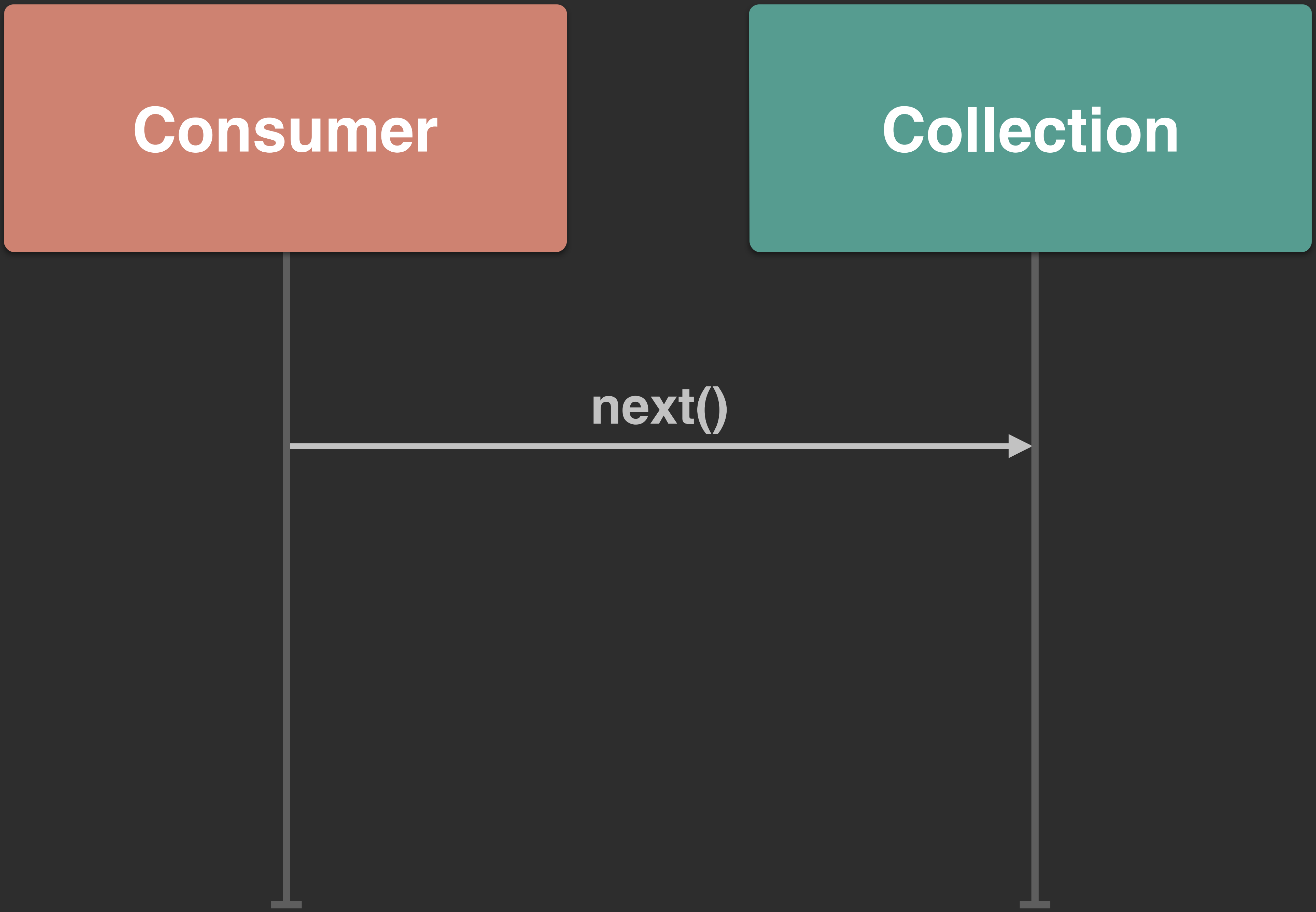
Reactive

Similar to a spreadsheet



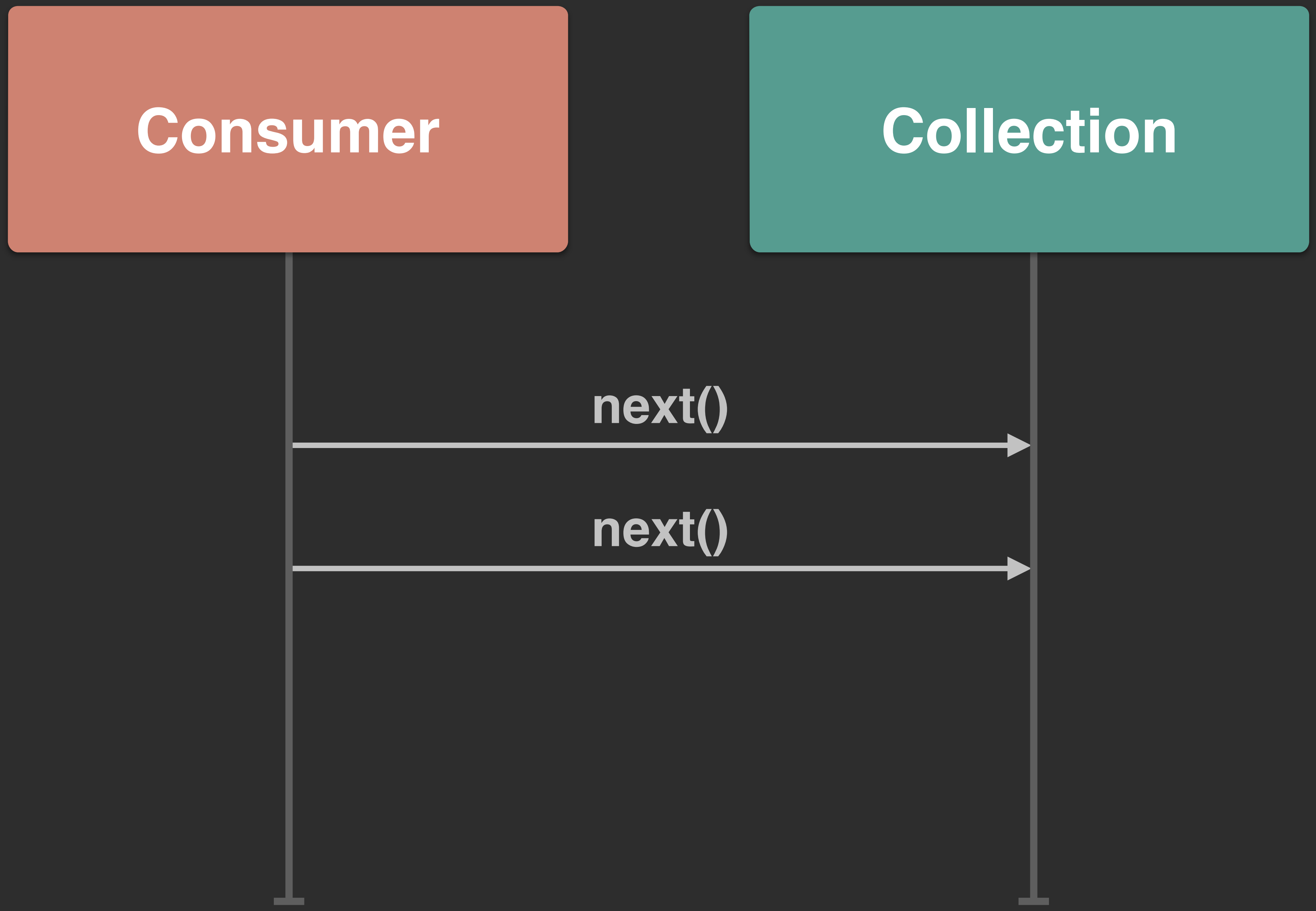
Iterable

Pulls from producer
Blocks consumer thread



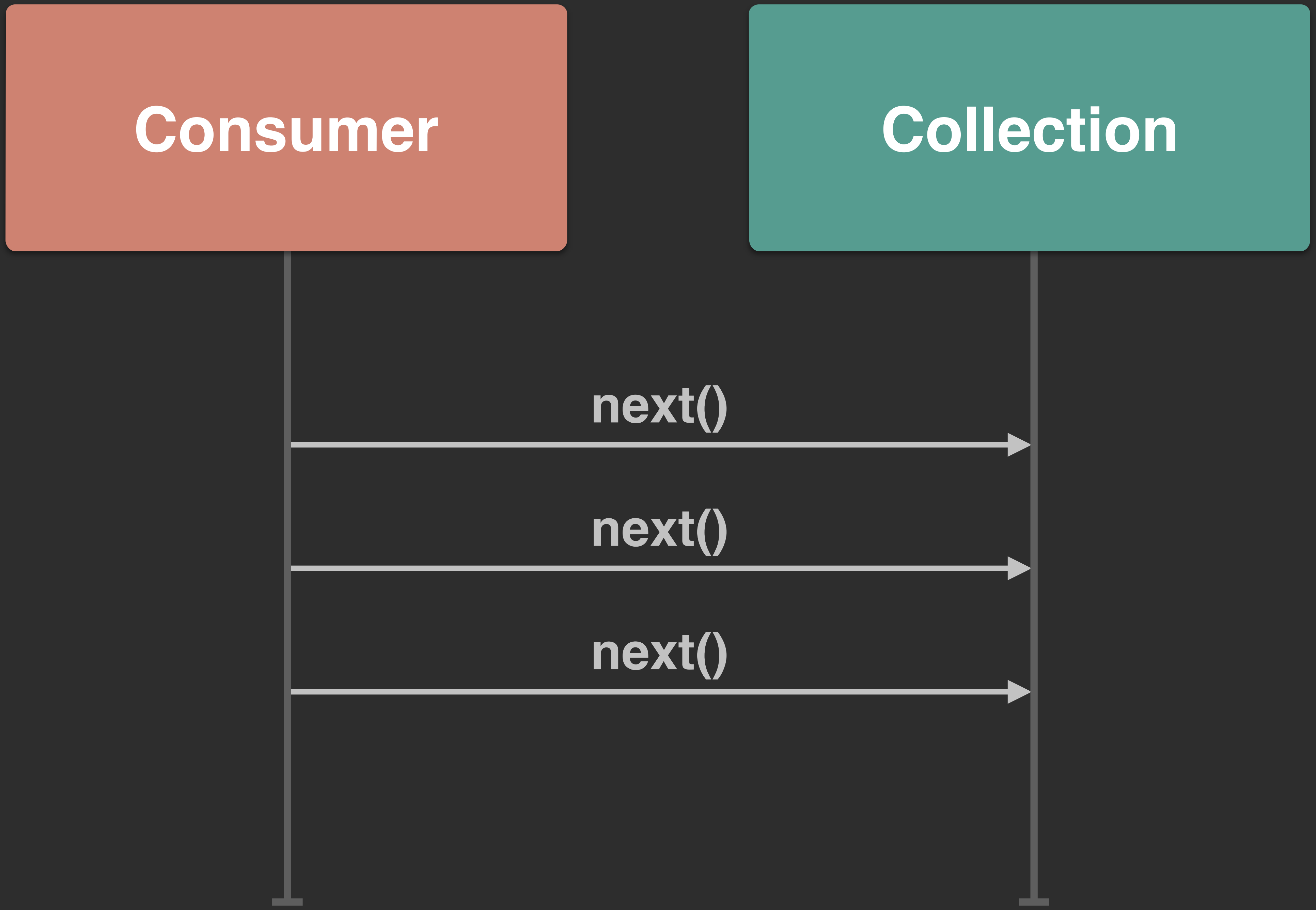
Iterable

Pulls from producer
Blocks consumer thread



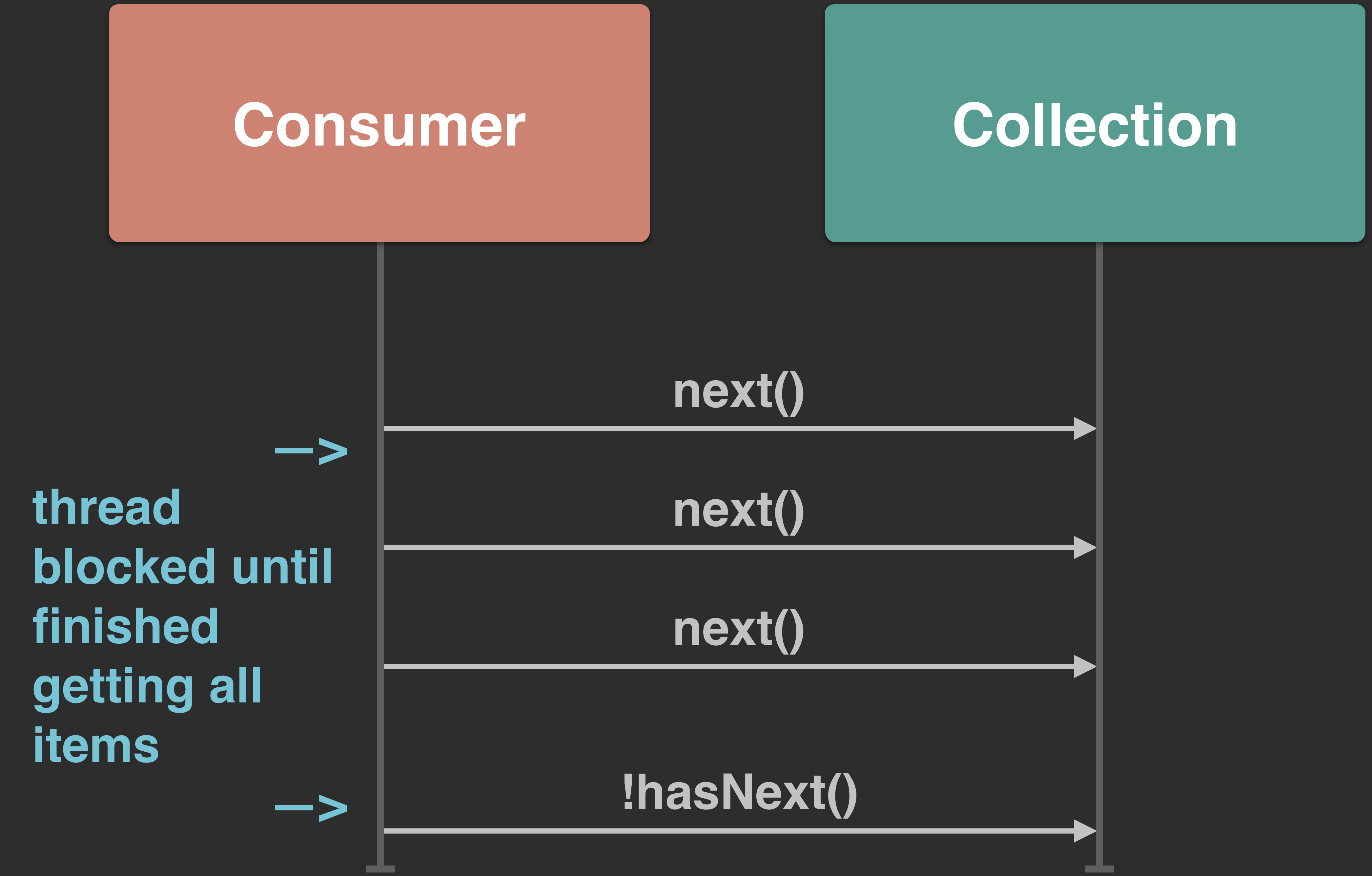
Iterable

Pulls from producer
Blocks consumer thread



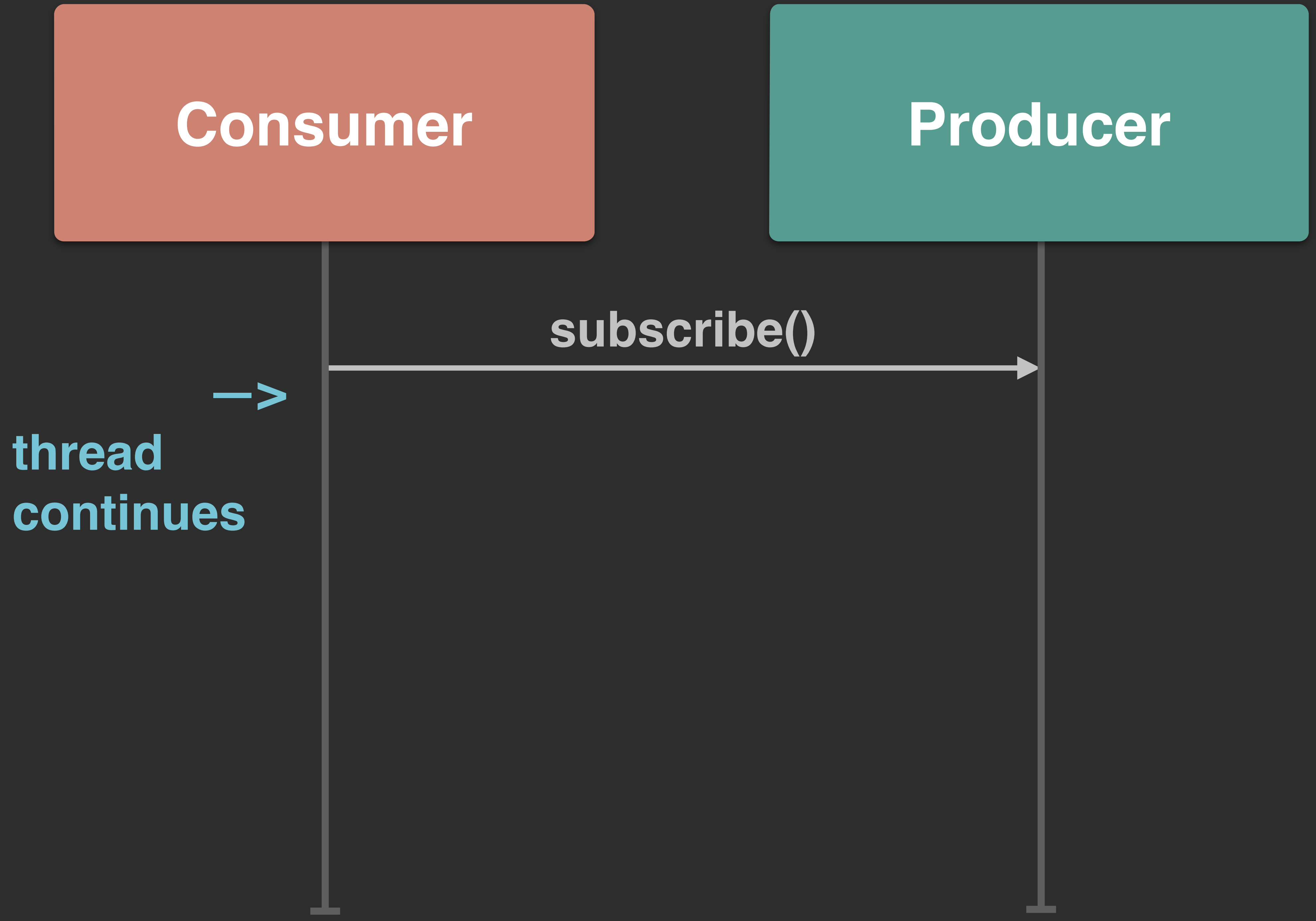
Iterable

Pulls from producer
Blocks consumer thread



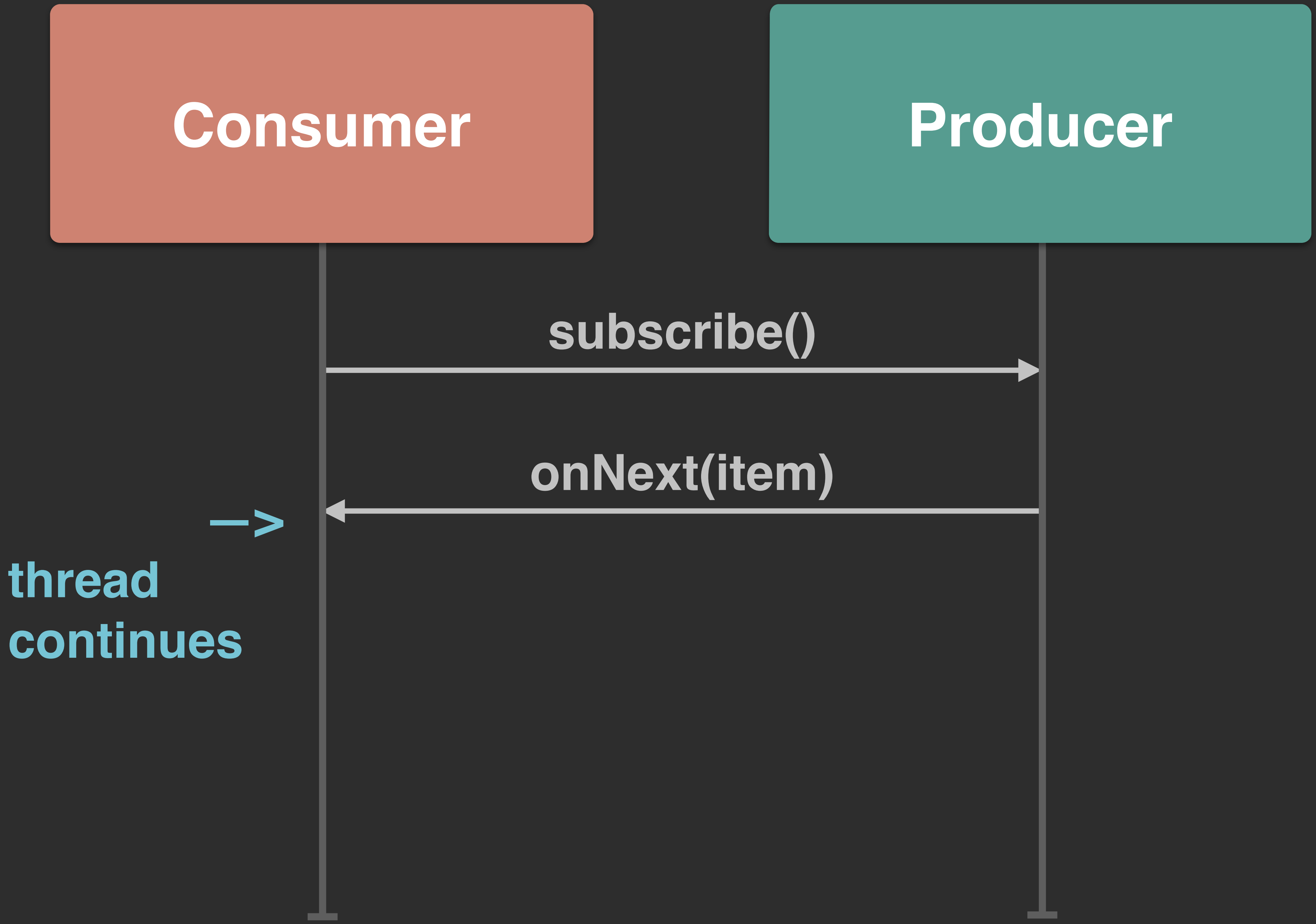
Observable

Reacts to producer
Does **not** block consumer



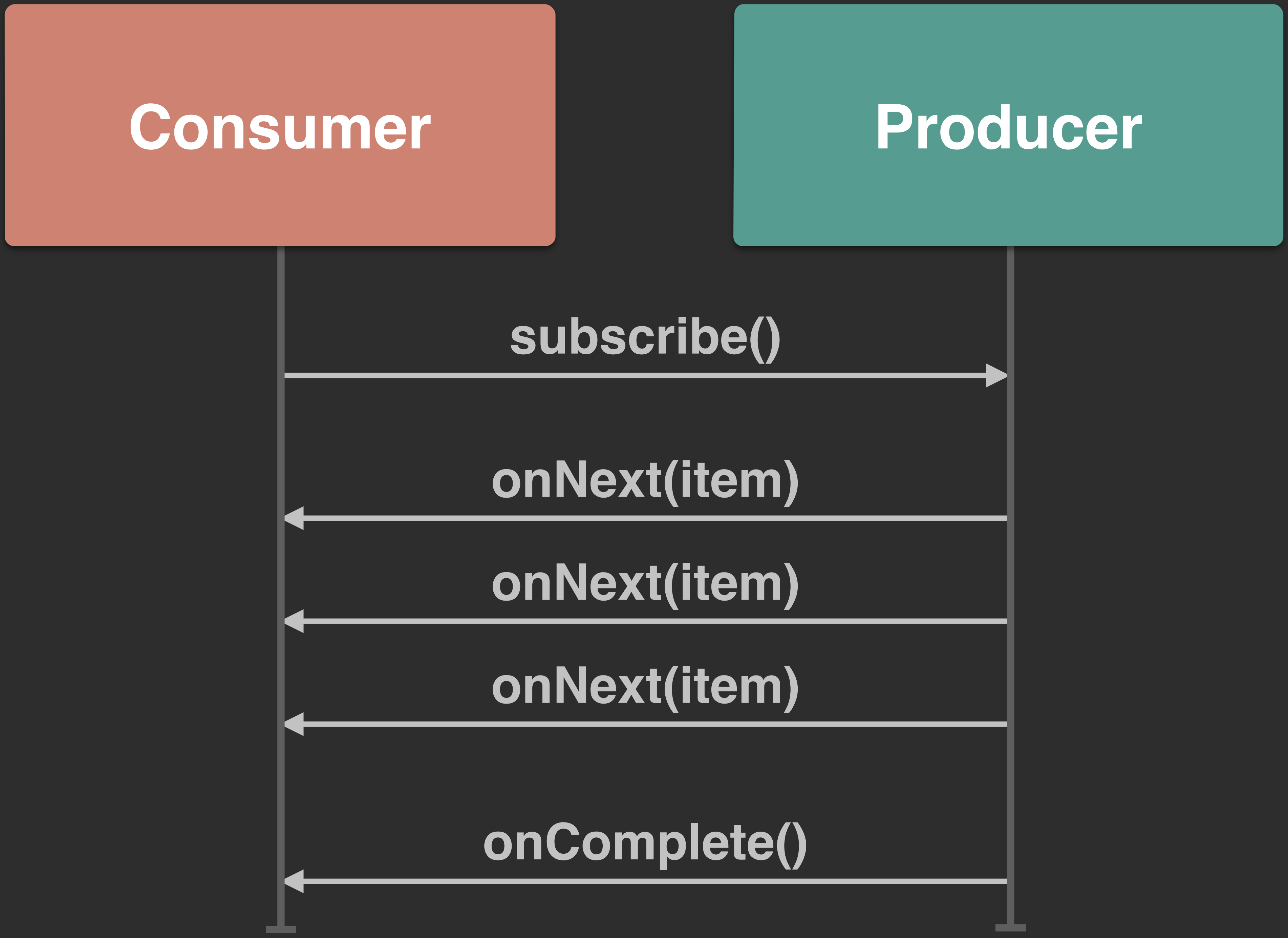
Observable

Reacts to producer
Does **not** block consumer



Observable

Reacts to producer
Does **not** block consumer



Functional Programming

Not Functional

Imperative

```
integers = [0, 1, 2, 3, 4, 5, 6, 7, 9];

doubled = new int[integers.length];
for(int i = 0; i < integers.length; i++) {
    doubled[i] = integers[i] * 2;
}
```

Not Functional

Imperative

```
integers = [0, 1, 2, 3, 4, 5, 6, 7, 9];

doubled = new int[integers.length];
for(int i = 0; i < integers.length; i++) {
    doubled[i] = integers[i] * 2;
}

filtered = new ArrayList<>();
for(int d : doubled) {
    if (d < 10) {
        filtered.add(d);
    }
}
```

Not Functional

Imperative

```
integers = [0, 1, 2, 3, 4, 5, 6, 7, 9];

doubled = new int[integers.length];
for(int i = 0; i < integers.length; i++) {
    doubled[i] = integers[i] * 2;
}

filtered = new ArrayList<>();
for(int d : doubled) {
    if (d < 10) {
        filtered.add(d);
    }
}
```

Functional

Apply to collections

```
integers = [0, 1, 2, 3, 4, 5, 6, 7, 9];

doubled = integers.map(i -> i * 2);

filtered = doubled.filter(i -> i < 10);
```

Functional

Apply to collections

```
integers = [0, 1, 2, 3, 4, 5, 6, 7, 9];

doubled = integers.map(i -> i * 2);

filtered = doubled.filter(i -> i < 10);
```

Functional

Apply to collections

```
integers = [0, 1, 2, 3, 4, 5, 6, 7, 9];

doubled = integers.map(i -> i * 2);

filtered = doubled.filter(i -> i < 10);

sum = filtered.sum();

sorted = filtered.sort(Integer::compare);
```

Functional

Composing

```
integers = [0, 1, 2, 3, 4, 5, 6, 7, 9];

doubledAndFilteredSum = integers.map(i -> i * 2)
                                .filter(i -> i < 10)
                                .sum();
```

Functional

Composing

```
integers = [0, 1, 2, 3, 4, 5, 6, 7, 9];

doubledAndFilteredSum = integers.map(i -> i * 2)
                                .filter(i -> i < 10)
                                .sum();
```

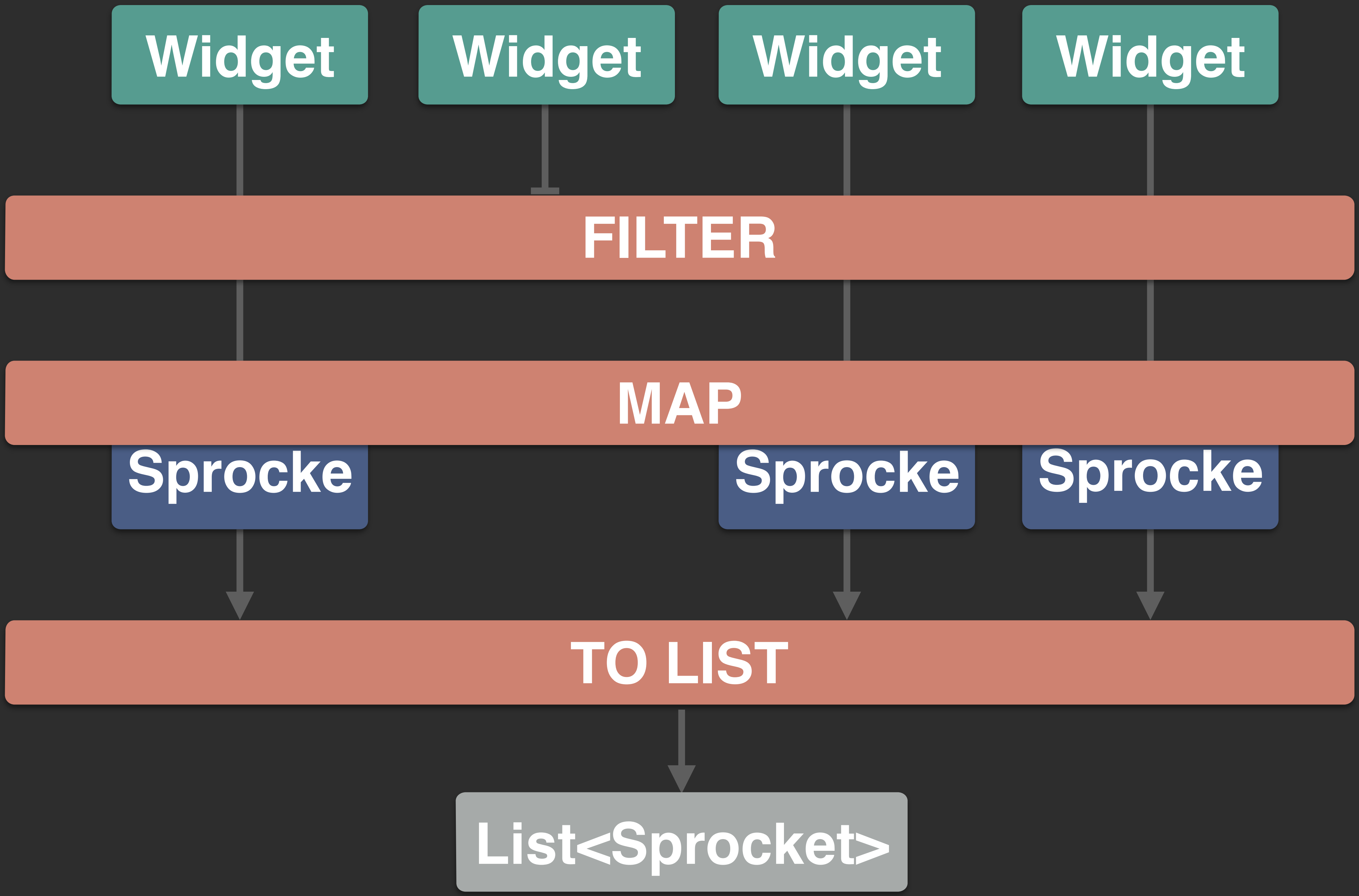
More expressive. Less error prone.

Functions Inputs, output, no side-effects

```
public static float fahrenheitFromKelvin(float kelvin) {
    return (kelvin - 273.15) * 1.80 + 32.00;
}
```

```
public static Customer customerFromJson(JSONObject customerJson) {
    Customer customer = new Customer();
    customer.setFirstName(customerJson.getString("firstName");
    customer.setFirstName(customerJson.getString("lastName");
    return customer;
}
```

Parallelism



Lambdas

Basically a one method class

```
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View button) {
        doSomething();
    }
});
```

Lambdas

Basically a one method class

```
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View button) {
        doSomething();
    }
});
```

Lambdas

Basically a one method class

```
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View button) {
        doSomething();
    }
});
```

```
button.setOnClickListener(button -> doSomething());
```

Method References

```
.map(json -> new Customer(json))
```

```
.map(Customer::new)
```

Method References

```
.map(json -> new Customer(json))
```

```
.map(Customer::new)
```

Retrolambda

- Lambda syntax in Android projects
- Gradle plugin
- Converts Java 8 bytecode to Java 7
- Risk: Jack and Jill compiler

Functional + Reactive Programming

The Code

Declaration

```
Observable<List<Customer>> localCustomerList =
      webService.getCustomers()
                .flatMap(r -> Observable.from(r.customerList)
                .map(json -> new Customer(json))
                .filter(c -> c.isLocal())
                .toList();
```

The Code

Declaration

```
Observable<List<Customer>> localCustomerList =
      webService.getCustomers()
                .flatMap(r -> Observable.from(r.customerList)
                .map(json -> new Customer(json))
                .filter(c -> c.isLocal())
                .toList();
```

Web Service Example

```
{
    customers: [
        {
            firstName: "Paul",
            lastName: "Cicero",
            isLocal: true
        },
        {
            firstName: "Tommy",
            lastName: "DeVito",
            isLocal: false
        },
        {
            firstName: "Billy",
            lastName: "Batts",
            isLocal: true
        }
    ]
}
```

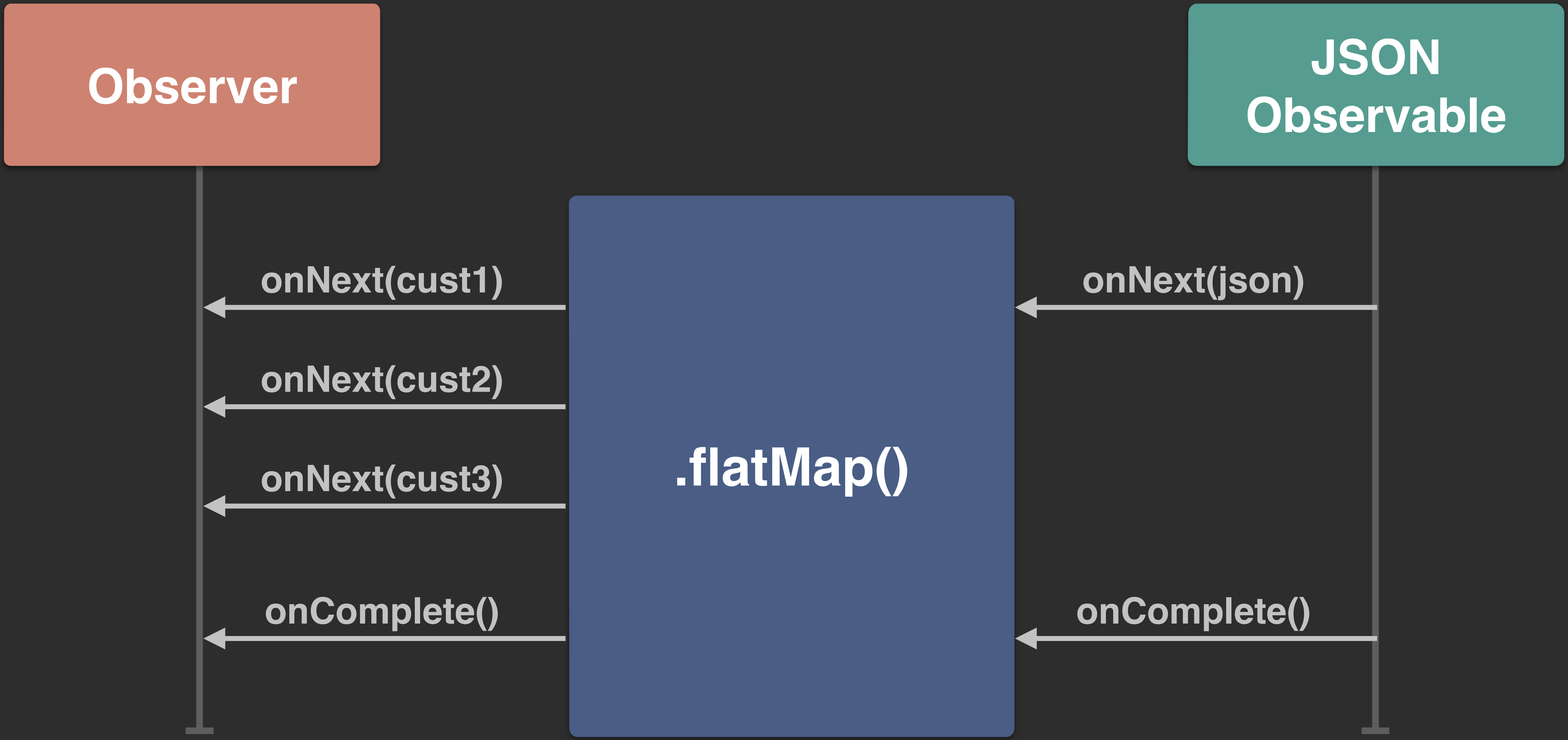
The Code

Declaration

```
Observable<List<Customer>> localCustomerList =
      webService.getCustomers()
                .flatMap(r -> Observable.from(r.customerList))
                .map(json -> new Customer(json))
                .filter(c -> c.isLocal())
                .toList();
```

FlatMap

Convert each item to Observable



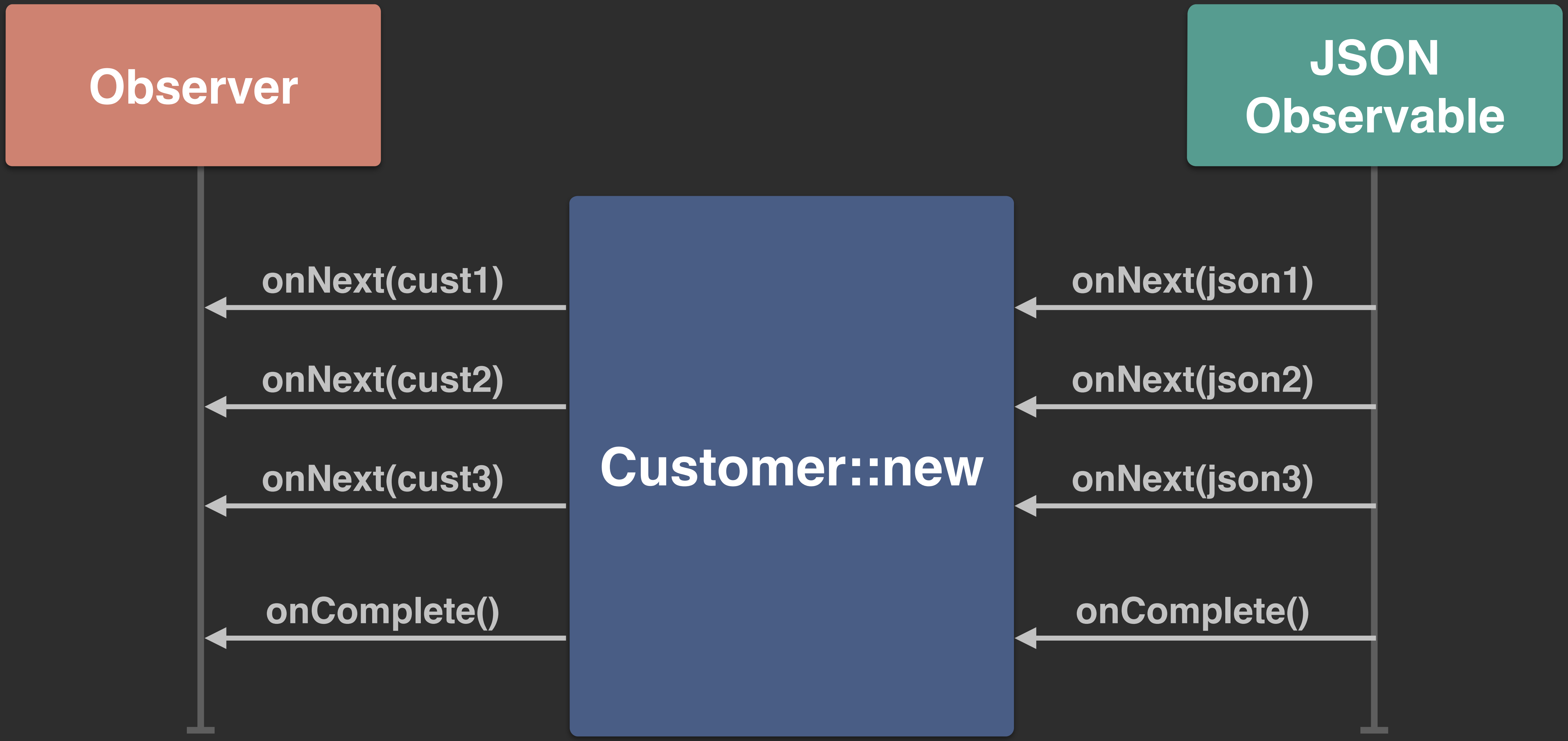
The Code

Declaration

```
Observable<List<Customer>> localCustomerList =
      webService.getCustomers()
                .flatMap(r -> Observable.from(r.customerList)
                .map(json -> new Customer(json))
                .filter(c -> c.isLocal())
                .toList();
```

Map

Transform each item to another item



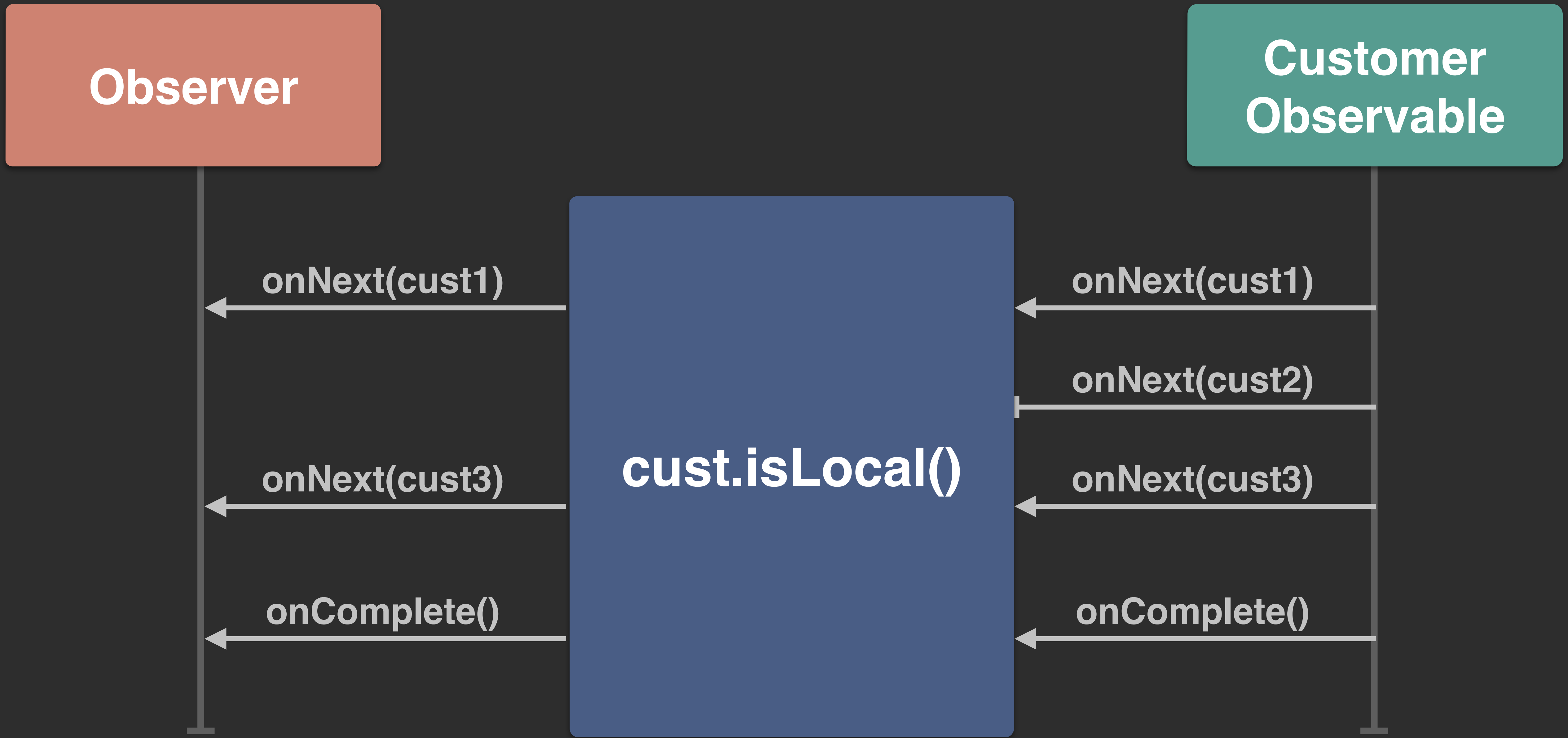
The Code

Declaration

```
Observable<List<Customer>> localCustomerList =
      webService.getCustomers()
                .flatMap(r -> Observable.from(r.customerList)
                .map(json -> new Customer(json))
                .filter(c -> c.isLocal())
                .toList();
```

Filter

Select items that match criteria

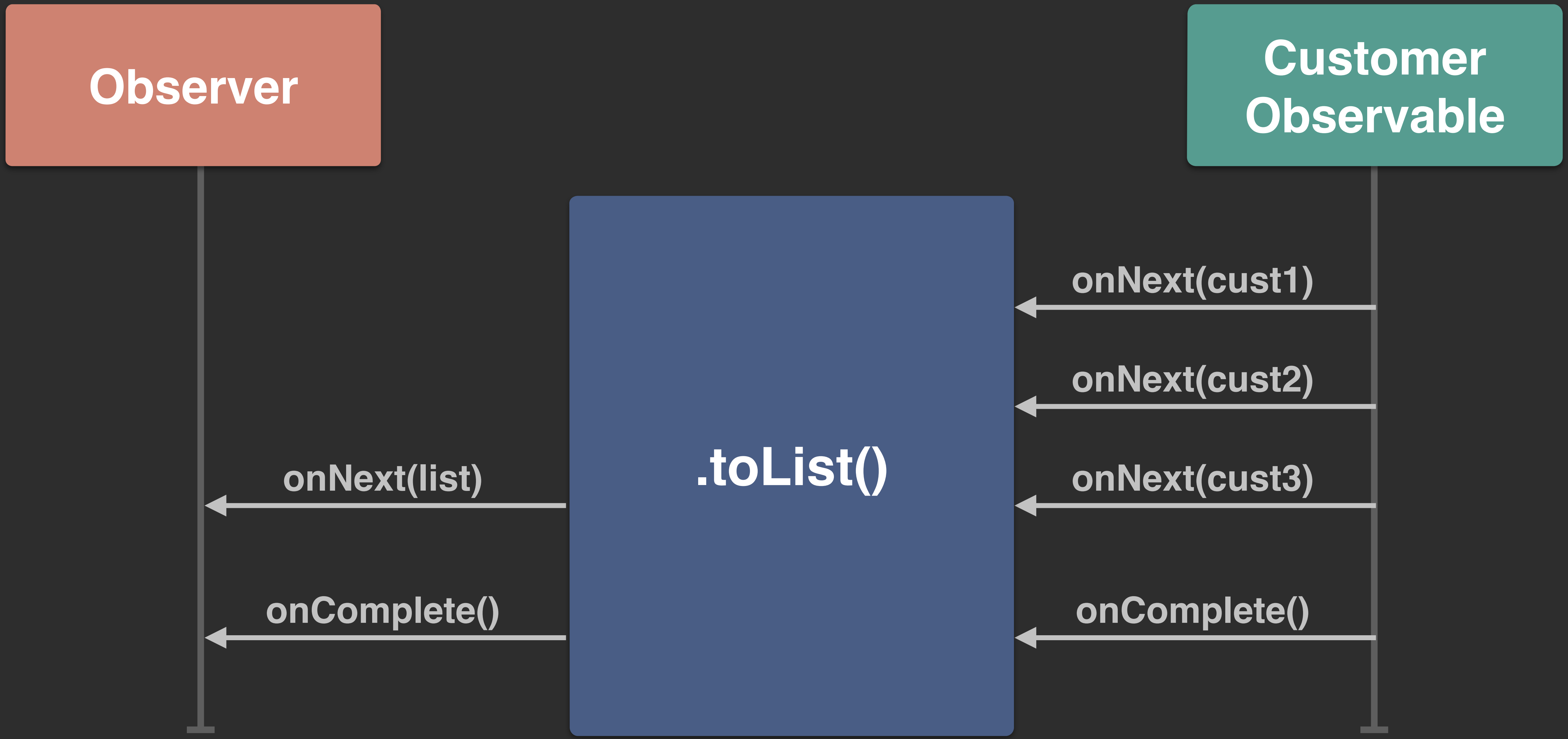


The Code

Declaration

```
Observable<List<Customer>> localCustomerList =
      webService.getCustomers()
                .flatMap(r -> Observable.from(r.customerList)
                .map(json -> new Customer(json))
                .filter(c -> c.isLocal())
                .toList();
```

Aggregate Operators

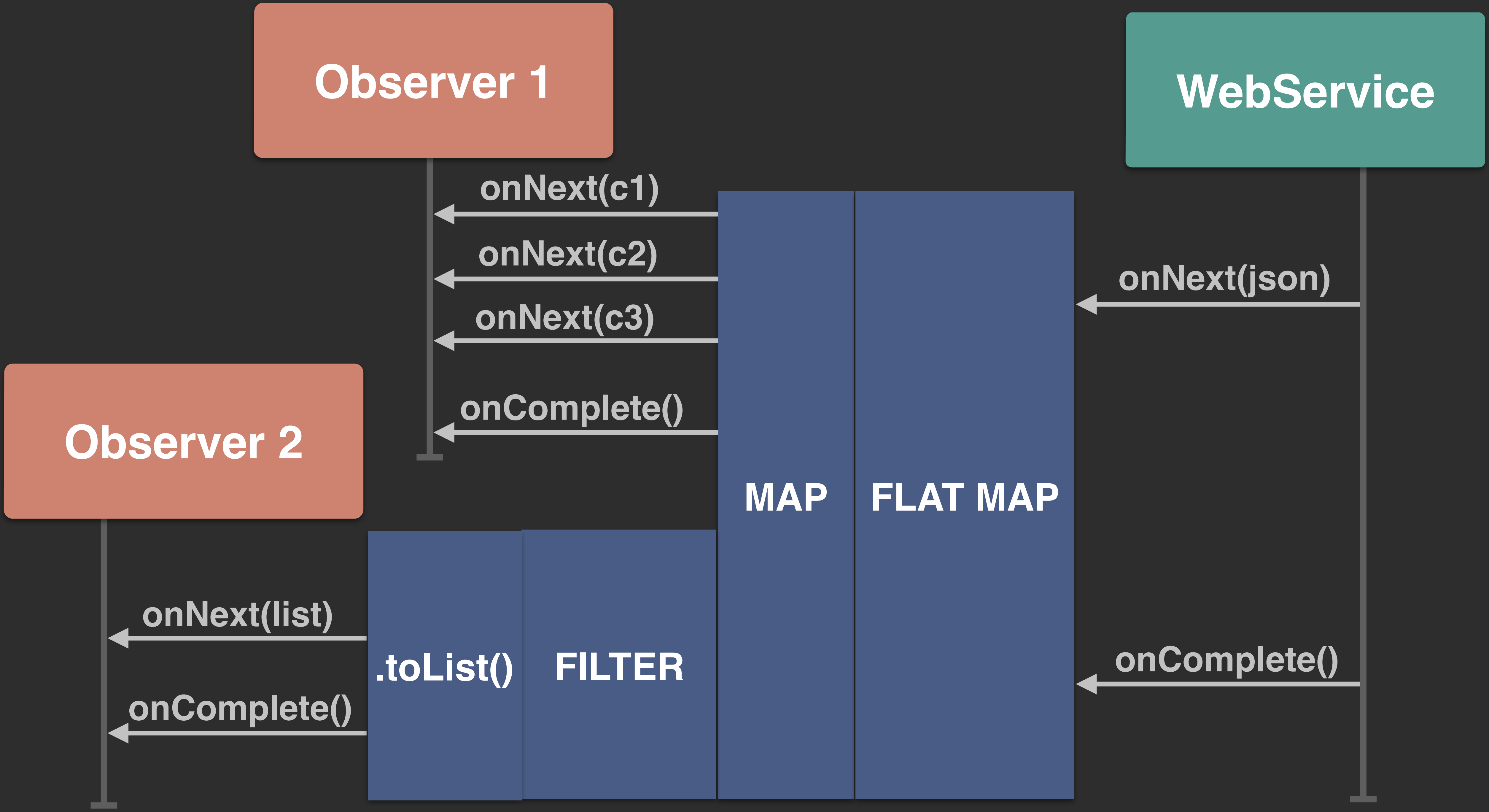


The Code

Declaration

```
Observable<Customer> allCustomers =
      webService.getCustomers()
                .flatMap(r -> Observable.from(r.customerList)
                .map(json -> new Customer(json));



Observable<List<Customer>> localCustomerList =
      allCustomers.filter(c -> c.isLocal())
                  .toList();
```

Functional Pipelines



Subscribing

Declaration

```
Observable<List<Customer>> localCustomerList =
      webService.getCustomers()
                .flatMap(r -> Observable.from(r.customerList)
                .map(json -> new Customer(json))
                .filter(c -> c.isLocal())
                .toList();
```

Consume

```
localCustomerList.subscribe(
    list -> display(list),
    e -> handleError(e) <— error handling!
);
```

Subscribing

Declaration

```
Observable<List<Customer>> localCustomerList =
      webService.getCustomers()
                .flatMap(r -> Observable.from(r.customerList)
                .map(json -> new Customer(json))
                .filter(c -> c.isLocal())
                .toList();
```

Consume

```
localCustomerList.subscribe(
    list -> display(list),
    e -> handleError(e)
);
```

or

```
localCustomerList.subscribe(
    this::display,
    this::handleError
);
```

RxAndroid

RxAndroid

- **AndroidSchedulers**
  - Observe on UI thread
- **AppObservable**
  - Protects against destroyed Fragment / Activity
- **LifecycleObservable**
  - Prevents leaking Fragment / Activity

Threading

```
localCustomerList.subscribe(
    this::display,
    this::handleError
);
```

Threading

Work in background thread pool

```
localCustomerList
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(
        this::display,
        this::handleError
    );
```

Threading

Handle on UI thread

```
localCustomerList
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(
        this::display,
        this::handleError
    );
```

Threads

```
localCustomerList
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(
        this::display,
        this::handleError
    );
```

AppObservable

```
AppObservable.bindFragment(this, localCustomerList)
    .subscribeOn(Schedulers.io())
~~    .observeOn(AndroidSchedulers.mainThread())~~
    .subscribe(
        this::display,
        this::handleError
    );
```

AppObservable

```
AppObservable.bindFragment(this, localCustomerList)
    .subscribeOn(Schedulers.io())
    .subscribe(
        this::display,
        this::handleError
    );
```

AppObservable

```
Observable<List> localCustomerList =
      webService.getCustomers() <— move inside here
                .flatMap(r -> Observable.from(r.customerList)
                .map(json -> new Customer(json))
                .filter(c -> c.isLocal())
                .toList();


AppObservable.bindFragment(this, localCustomerList)
~~    .subscribeOn(Schedulers.io())~~
    .subscribe(
        this::display,
        this::handleError
    );
```

AppObservable

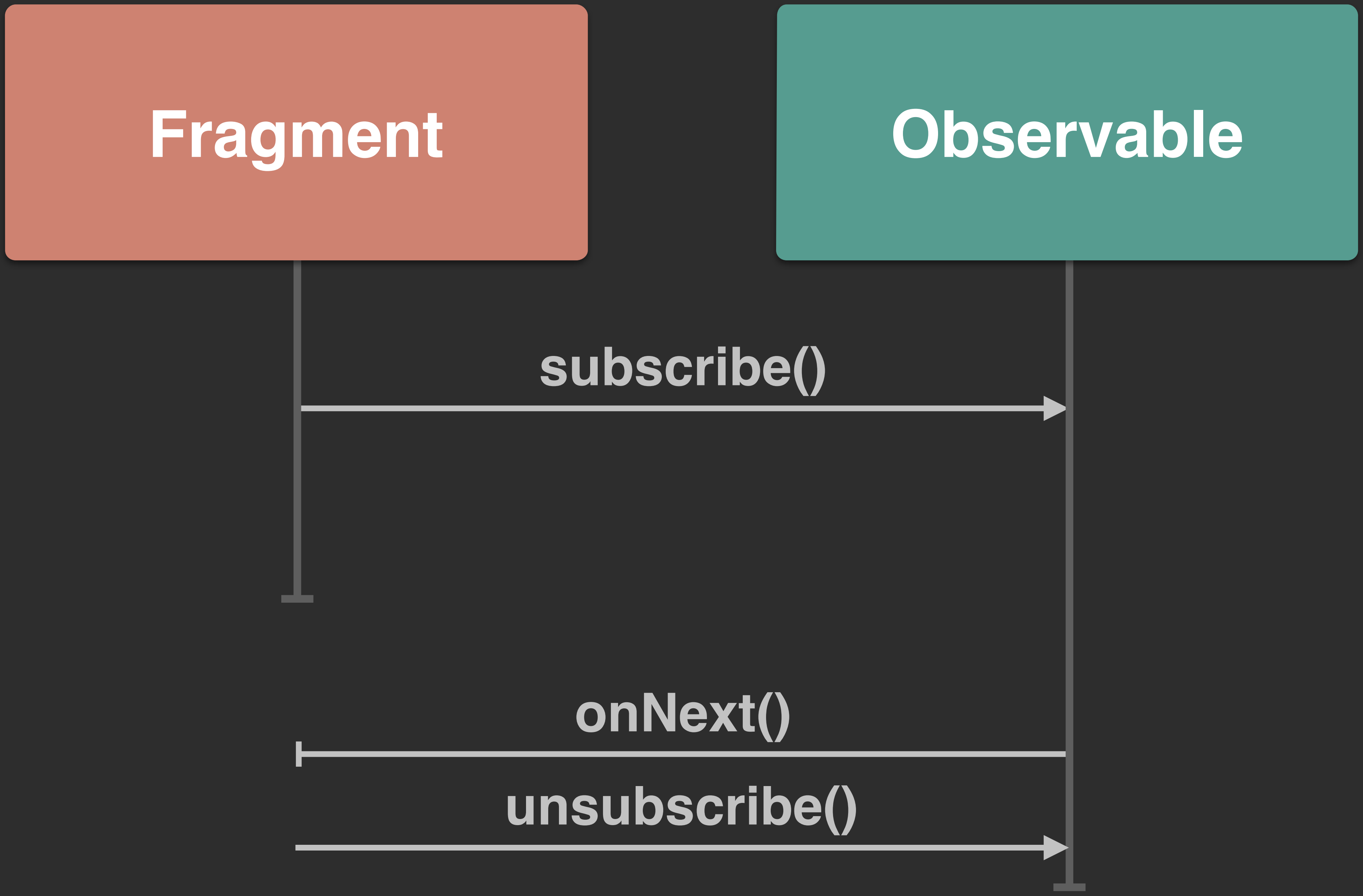
```
Observable<List> localCustomerList =
      webService.getCustomers()
                .flatMap(r -> Observable.from(r.customerList)
                .map(json -> new Customer(json))
                .filter(c -> c.isLocal())
                .toList();


~~AppObservable.~~bindFragment(this, localCustomerList)
    .subscribe(
        this::display,
        this::handleError
    );
```

AppObservable

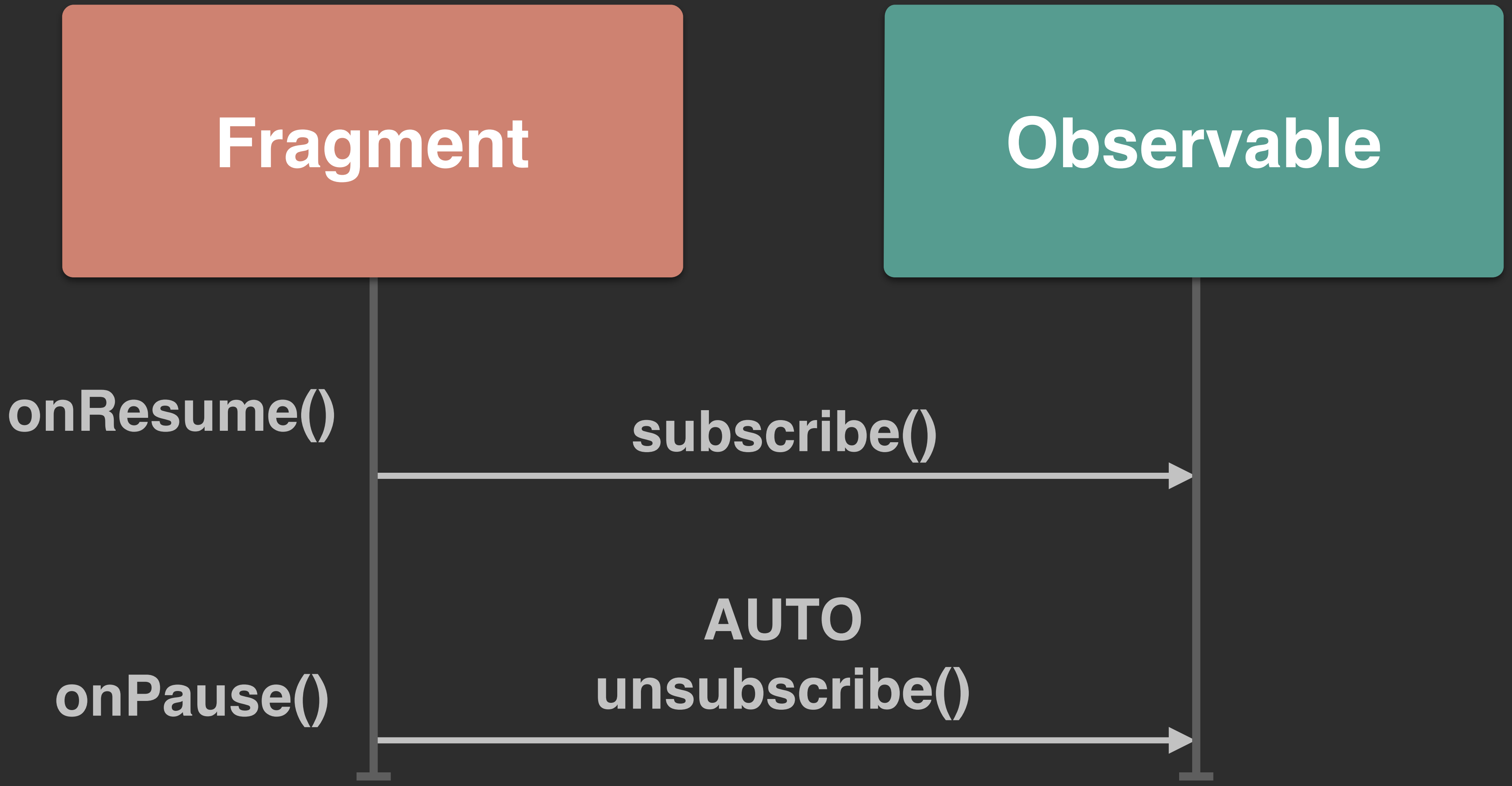
```
Observable<List> localCustomerList =
      webService.getCustomers()
                .flatMap(r -> Observable.from(r.customerList)
                .map(json -> new Customer(json))
                .filter(c -> c.isLocal())
                .toList();


bindFragment(this, localCustomerList)
   .subscribe(
        this::display,
        this::handleError
    );
```

AppObservable

Safeguards



LifecycleObservable



Fragment

Observable

onResume()

subscribe()

onPause()

AUTO
unsubscribe()

More Power

Complex threading

```
public Observable<List<CityWeather>> getWeatherForLargeUsCapitals() {
    return cityDirectory.getUsCapitals()
        .flatMap(cityList -> Observable.from(cityList))
        .filter(city -> city.getPopulation() > 500,000)
        .flatMap(city -> weatherService.getCurrentWeather(city))
        .toSortedList((cw1,cw2) -> cw1.getName().compare(cw2.getName()));
}
```

Complex threading

```
public Observable<List<CityWeather>> getWeatherForLargeUsCapitals() {
    return cityDirectory.getUsCapitals()
        .flatMap(cityList -> Observable.from(cityList))
        .filter(city -> city.getPopulation() > 500,000)
        .flatMap(city -> weatherService.getCurrentWeather(city))
        .toSortedList((cw1,cw2) -> cw1.getName().compare(cw2.getName()));
}
```

Complex threading

```
public Observable<List<CityWeather>> getWeatherForLargeUsCapitals() {
    return cityDirectory.getUsCapitals()
        .flatMap(cityList -> Observable.from(cityList))
        .filter(city -> city.getPopulation() > 500,000)
        .flatMap(city -> weatherService.getCurrentWeather(city))
        .toSortedList((cw1,cw2) -> cw1.getName().compare(cw2.getName()));
}
```

Complex threading

```
public Observable<List<CityWeather>> getWeatherForLargeUsCapitals() {
    return cityDirectory.getUsCapitals()
        .flatMap(cityList -> Observable.from(cityList))
        .filter(city -> city.getPopulation() > 500,000)
        .flatMap(city -> weatherService.getCurrentWeather(city))
        .toSortedList((cw1,cw2) -> cw1.getName().compare(cw2.getName()));
}
```

Complex threading

```
public Observable<List<CityWeather>> getWeatherForLargeUsCapitals() {
    return cityDirectory.getUsCapitals()
        .flatMap(cityList -> Observable.from(cityList))
        .filter(city -> city.getPopulation() > 500,000)
        .flatMap(city -> weatherService.getCurrentWeather(city))
        .toSortedList((cw1,cw2) -> cw1.getName().compare(cw2.getName()));
}
```

Complex threading

```
public Observable<List<CityWeather>> getWeatherForLargeUsCapitals() {
    return cityDirectory.getUsCapitals()
        .flatMap(cityList -> Observable.from(cityList))
        .filter(city -> city.getPopulation() > 500,000)
        .flatMap(city -> weatherService.getCurrentWeather(city))
        .toSortedList((cw1,cw2) -> cw1.getName().compare(cw2.getName()));
}
```

Complex threading

```
public Observable<List<CityWeather>> getWeatherForLargeUsCapitals() {
    return cityDirectory.getUsCapitals()
        .flatMap(cityList -> Observable.from(cityList))
        .filter(city -> city.getPopulation() > 500,000)
        .flatMap(city -> weatherService.getCurrentWeather(city))
        .toSortedList((cw1,cw2) -> cw1.getName().compare(cw2.getName()));
}
```

Complex threading

```
public Observable<List<CityWeather>> getWeatherForLargeUsCapitals() {
    return cityDirectory.getUsCapitals()
        .flatMap(cityList -> Observable.from(cityList))
        .filter(city -> city.getPopulation() > 500,000)
        .flatMap(city -> weatherService.getCurrentWeather(city))
        .toSortedList((cw1,cw2) -> cw1.getName().compare(cw2.getName()));
}
```

Debouncing

```
temperatureChanges
    .debounce(2, SECONDS)
    .subscribe(WebService::save);
```

Debouncing

```
temperatureChanges
    .debounce(2, SECONDS)
    .subscribe(WebService::save);
```

```
WidgetObservable.text(searchBox)
    .debounce(500, MILLISECONDS)
    .flatMap(webService::search)
    .subscribe(this::displayResults);
```

Cache on rotation

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setRetainInstance(true);
    weatherObservable = weatherManager.getWeather().cache();
}

public void onViewCreated(...) {
    super.onViewCreated(...)
    weatherObservable.subscribe(this);
}
```

Why RxJava / RxAndroid?

- More robust interface than AsyncTask
- Easy to do complex threading
- Functional nature is more expressive

Why Retrolambda?

- Lambdas for Android
- Cleaner, simpler, more expressive
- Actually feels more functional