Chain operations

Synchronous or asynchronous

To chain operations synchronously, statements are usually written one after the other, using the result of the previous statement to execute the new statement:

```
// Build an animate action.
Animate animate = ...;
// Run the animate action synchronously.
animate.run();
// Build a listen action.
Listen listen = ...;
// Run the listen action synchronously.
listen.run();
```

Here, we run a Animate then run Listen, This is all done in a synchronous way

These operations can also be performed asynchronously: the

```
// Run the animate action asynchronously.
animate.async().run();
```

```
// Run the listen action asynchronously.
listen.async().run();
```

These snippets perform asynchronous operations, but they are not chained together: both actions will start at the same time.

What we want is to perform the second action (in this case listen when the animation is finished) when the first action provides a return result. We'll see below how to link these actions asynchronously.

Futures

What's a future

Future Wrapping objects for asynchronous operations.

It allows you to perform asynchronous operations while writing code in a sequential manner. </p

Future class is mainly used to perform the following operations asynchronously:

- creating objects such as actions and resources,
  Handling action execution (results, cancellations, and errors),Chain some actions to perform resource creation.

futurepresent state

一个 Future There are 3 states: it can provide results, it can be cancelled or it can fail. These states correspond to the different final states of the wrapped operation.

If a wrapped operation:

- returns a value, Future will provide this value when available,
- is cancelled, Future will be cancelled,
  Error encountered, Future would fail.

Future<T> is a generic class, where T is the type of value it can provide. For example Future<String> Being able to provide a String Value of type.

take note of

If the wrapped operation returns nothing, then the type of future is Future<Void>. In this case, if the operation succeeds Future<Void> it will end with the result state.

Future class provides get method to access the value of Future synchronously:

get method to synchronise access to the value of Future.

```
Future<String> stringFuture = ...;
String value = stringFuture.get(); // Synchronous call to get the value.
```

Calling the get method blocks the current thread: the call is synchronous.

Alert

Synchronous calls block, while asynchronous calls return the value immediately. </p

If an error occurs in a wrapped operation, the get call will throw an ExecutionException if the operation is cancelled. get The call will throw a CancellationException.

Use a try-catch to determine Future finished state:

```
Future<String> stringFuture = ...;
try {
    String value = stringFuture.get();
    // The future finished with value.
} catch (ExecutionException e) {
    // The future finished with error.
} catch (CancellationException e) {
    // The future was cancelled.
}
```

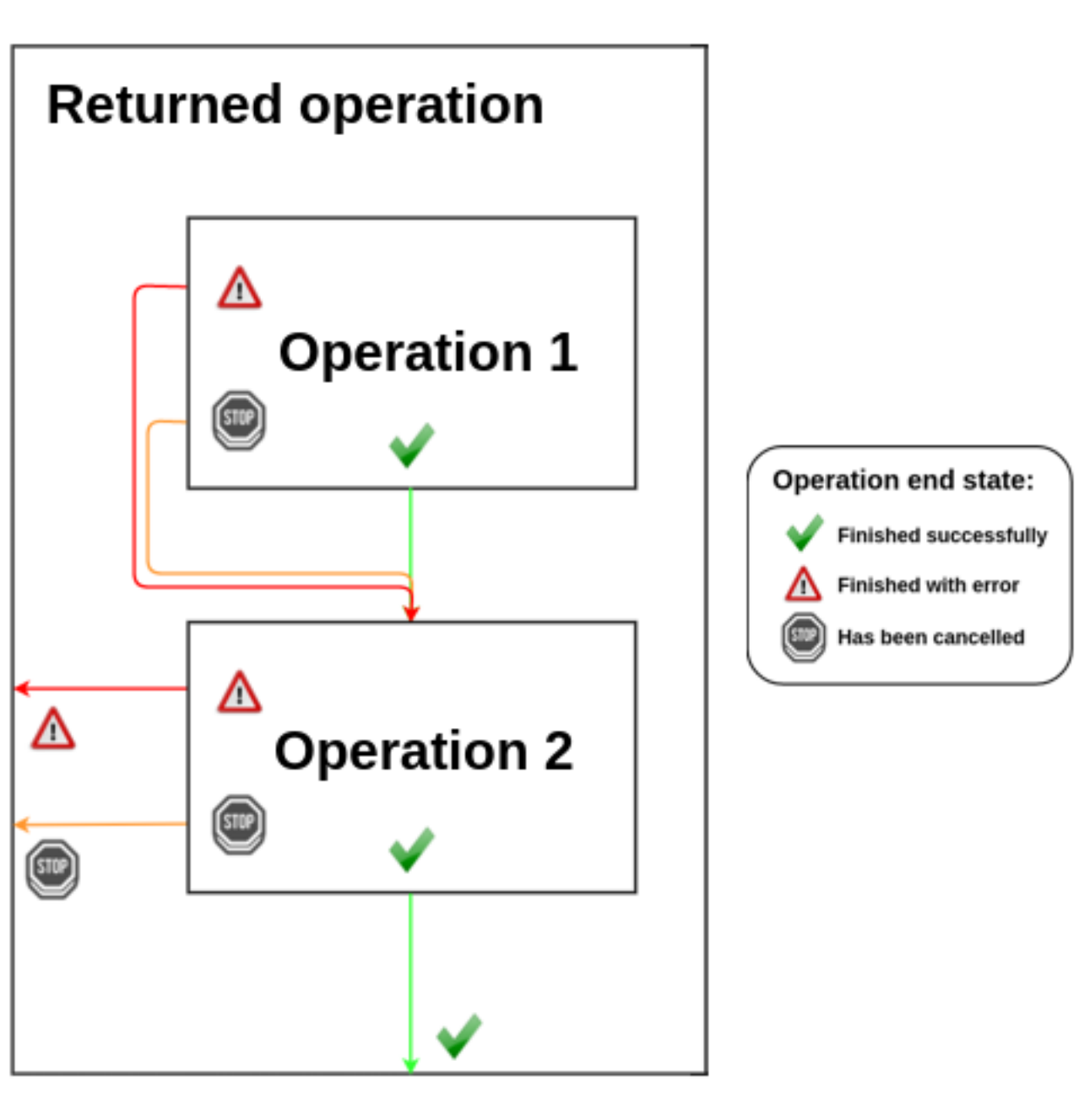
Operator

Future class contains several methods that let you chain several asynchronous operations:

- thenConsume ,
- thenApply ,
- thenCompose ,
- andThenConsume ,
- andThenApply ,
- andThenCompose .

then / andThen

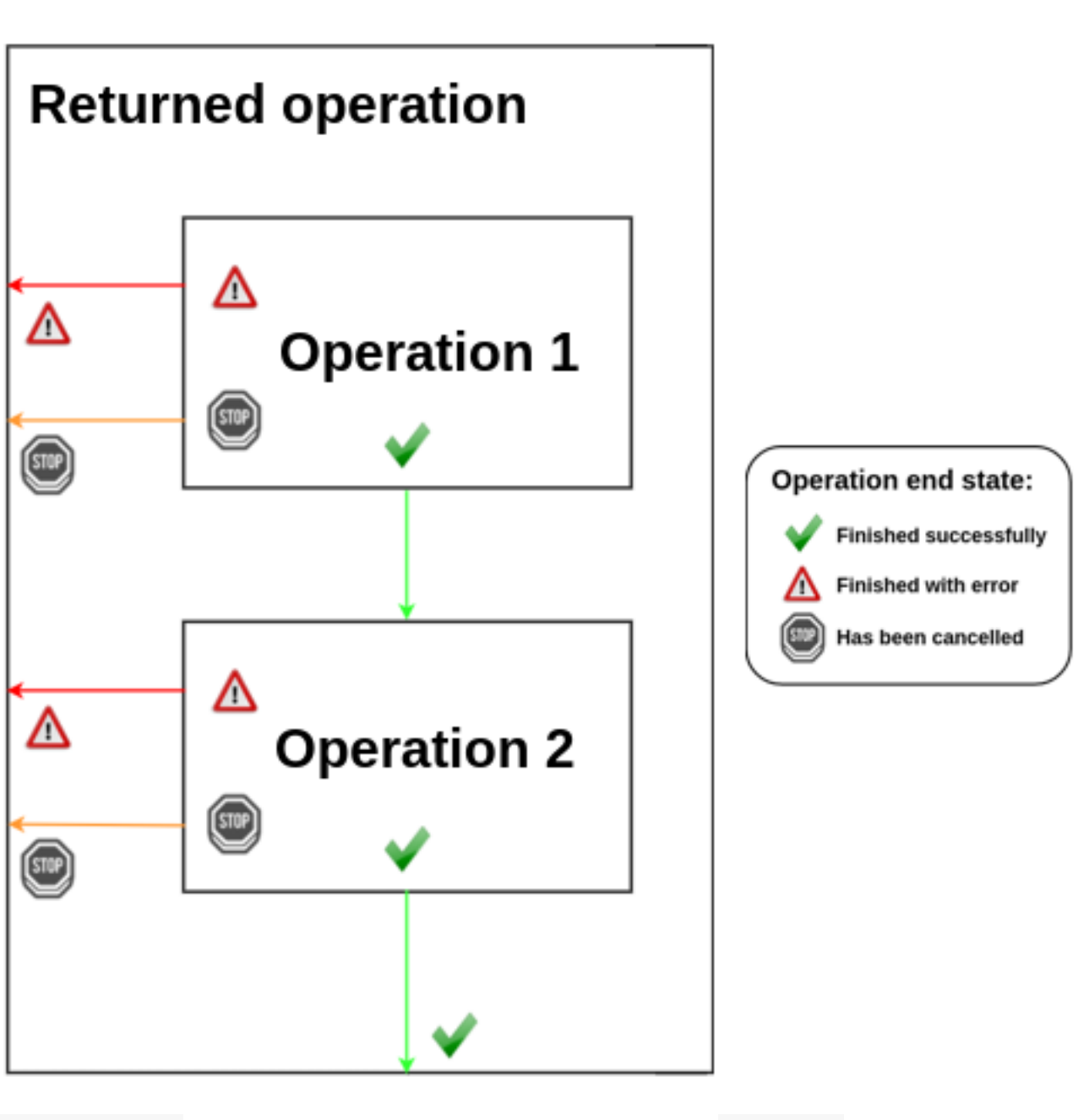
then... methods allow you to link 2 operations regardless of the end state of the first operation. They are useful when you want to chain operations that are independent of each other.



then... gives you access to the first operation of the Future wrapper. Use the isSuccess , hasError and isCancelled methods to determine the hasError and isCancelled methods to determine the status of the Future state.:

```
Future<String> future = ...;
future.thenConsume(stringFuture -> {
    if (stringFuture.isSuccess()) {
        // Handle success state.
        // Access the value with stringFuture.get().
    } else if (stringFuture.isCancelled()) {
        // Handle cancelled state.
    } else {
        // Handle error state.
        // Access the error with stringFuture.getError().
    }
});
```

andThen... This method allows you to chain 2 operations when and only when the first one ends in a result state. They can be useful when certain operations depend on the previous operation in the chain.



andThen... Directly gives you access to Future The value of the first operation of the package:

```
Future<String> future = ...;
future.andThenConsume(string -> {
    // Here you have access to the String.
});
```

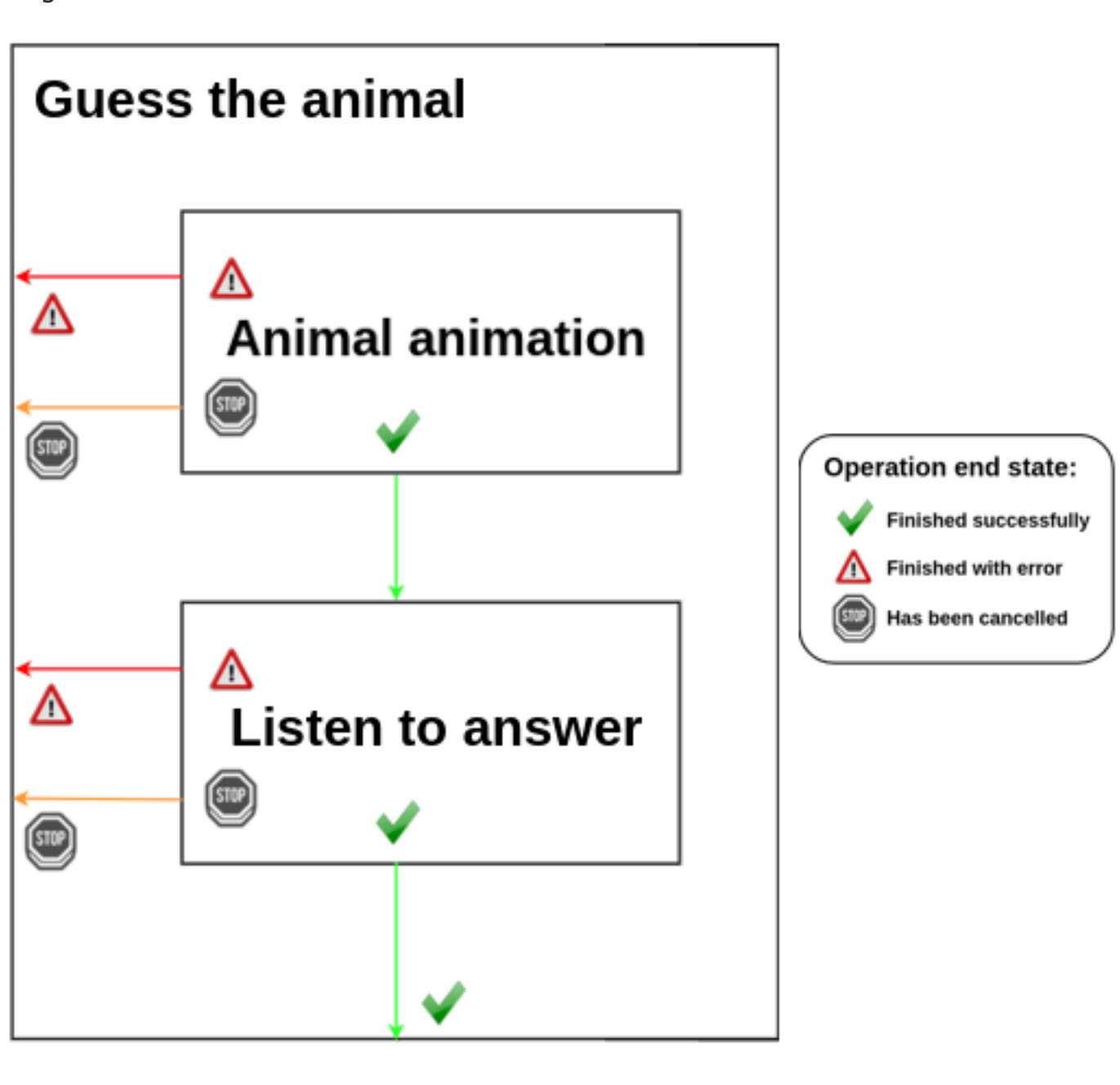
Suggestion</p

Could treat andThen... as boolean& operator: if left is false, don't look at right. as a Boolean & operator: if left is false, don't look at right.

Returned future:

The returned Future wrapped operation is equivalent to 2 consecutive operations.

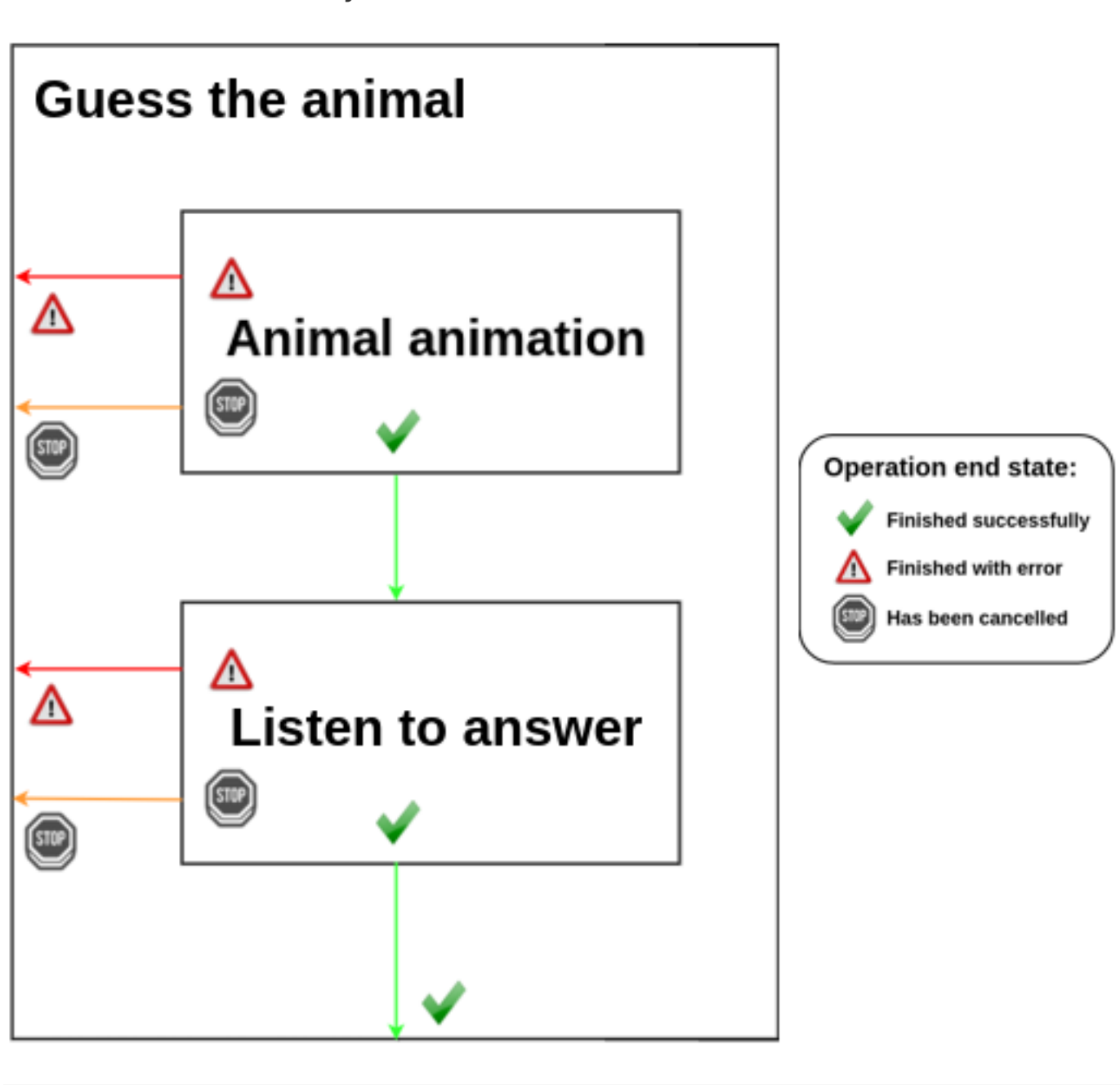
We want Pepper to perform a dance consisting of several animations. the animations must be chained together so that the dance continues even if one animation fails: we use the then... .




```
// Build an animate action.
Animate firstAnimate = ...;
// Run the animate action asynchronously.
Future<Void> firstDance = firstAnimate.async().run();
// Chain several animations.
Future<Void> ballet = firstDance.thenCompose(dance -> {
    // Build an animate action.
    Animate secondAnimate = ...;
    // Run the animate action asynchronously.
    Future<Void> secondDance = secondAnimate.async().run();
    return secondDance;
});
```

"andThen…" Example:

We want Pepper to imitate an animal and then ask someone to guess the animal. Pepper listens to the answer when and only when the animation is successful: this is when we use the andThen...




```
// Build an animate action.
Animate animate = ...;
// Run the animate action asynchronously.
Future<Void> imitation = animate.async().run();
// Chain the animation with a listen action.
Future<ListenResult> guess = imitation.andThenCompose(animateFuture -> {
    // Build a listen action.
    Listen listen = ...;
    // Run the listen action asynchronously.
    Future<ListenResult> answerListening = listen.async().run();
    return answerListening;
});
```

consume / apply / compose

Depending on the result returned by the second action, you will use one of the following methods:

- ...Consume ,
- ...Apply ,
- ...Compose .

consume:

Using ... .Consume consumes the Future and does not return any value:

```
Future<String> future = ...;
Future<Void> returnedOperation = future.andThenConsume(string -> Log.i(TAG, "Success: " + string));
```

apply:

Using ... .Apply transforms the value of Future and returns the special type:

```
Future<String> future = ...;
Future<Integer> returnedOperation = future.andThenApply(string -> string.length());
```

compose:

Using ... .Compose to return a Future . This is mainly used to chain actions:

```
Say say = ...;
Future<Void> sayFuture = say.async().run();
Future<Void> returnedOperation = sayFuture.andThenCompose(ignore -> {
    Animate animate = ...;
    Future<Void> animateFuture = animate.async().run();
    return animateFuture;
});
```

The state of the operation:

Operations represented by lambda have different states, depending on the operations completed inside the lambda:

Operation	State	
	If lambda throws	**Else**
Consume	Error	Success
Apply	Error	Success
Compose	Error	Inner Future 's state*

The internal Future is the first step in the Using the ... .Compose lambda returned

.

Callback Callbacks

Callbacks Callbacks

Each chain operation takes a different callback as a parameter:

Operator	andThen…	then…
…Consume	Consumer<T>	Consumer<Future<T>>
…Apply	Function<T, R>	Function<Future<T>, R>
…Compose	Function<T, Future<R>>	Function<Future<T>, Future<R>>

Consumer

Consumer The interface contains the consume method. It is used to consume Future .

attention

consume A method for consumer whose interface executes on a **background thread.**

Example:

Record Future results:

```
// Java 8:
Future<String> future = ...;
future.andThenConsume(string -> Log.i(TAG, "Success: " + string));
```

```
// Java 7:
Future<String> future = ...;
future.andThenConsume(new Consumer<String>() {
    @Override
    public void consume(String string) throws Throwable {
        Log.i(TAG, "Success: " + string);
    }
});
```

Function

Function The interface contains the execute method. It is used to return a new value / Future .

attention

Function The execute method of the execute method, whose interface is run in a background thread. Its interface is run in a **background thread.**

Example.

Example:

Transforming Future results:

```
// Java 8:
Future<String> future = ...;
future.andThenApply(string -> string.length());
```

```
// Java 7:
Future<String> future = ...;
future.andThenApply(new Function<String, Integer>() {
    @Override
    public Integer execute(String string) throws Throwable {
        return string.length();
    }
});
```

actions:

```
// Java 8:
Animate animate = ...;
animate.async().run().andThenCompose(ignore -> {
    Say say = ...;
    return say.async().run();
});
```

```
// Java 7:
Animate animate = ...;
animate.async().run().andThenCompose(new Function<Void, Future<Void>>() {
    @Override
    public Future<Void> execute(Void ignore) throws Throwable {
        Say say = ...;
        return say.async().run();
    }
});
```

See also

javadoc