BCS 371 Mobile Application Development I

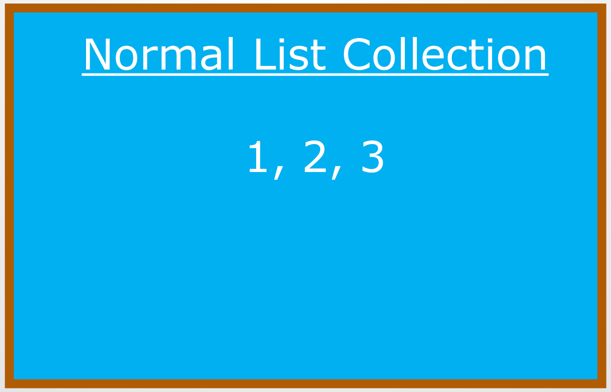
Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today's Lecture

- Kotlin Flow
- Using Kotlin Flow in Jetpack Compose

Normal List Collection

- Stores multiple pieces of data.
- Has functionality to add/remove data.
- If a client of a collection needs to know if new data has been added they need to iterate through the collection.



Normal List Collection

Generate and Use a Normal List

- The code below has a function to create a list and a function to use a list.

```
fun normalListGenerator(): List<Int> {
    val nums = mutableListOf<Int>()

    nums.add(1)
    nums.add(2)
    nums.add(3)

    return nums
}

fun testNormalList() {
    val normalList = normalListGenerator()
    for (i in normalList) {
        println(i)
    }
}
```

Return type is List<Int>

Add data to the list one item at a time

Returns a normal list (all values are returned at once)

Gets a normal list. All values are returned at one time.

Generate and Use a Normal List

Generate Normal List with a Delay

- A delay is added before adding each num. This code was run in a ViewModel.

```
suspend fun normalListDelayGenerator(): List<Int> {
    val nums = mutableListOf<Int>()
    delay(1000)
    nums.add(1)
    delay(1000)
    nums.add(2)
    delay(1000)
    nums.add(3)
    return nums
}
fun testNormalListDelay(){
    println("Getting list")
    viewModelScope.launch {
        val normalList = normalListDelayGenerator()
        for (i in normalList) {
            println(i)
        }
    }
}
```

There is a delay before adding each number

Try It Out

To try this out create a view model class and copy the following code in it. In the main screen composable, get a view model instance and call testNormalListDelay. Data will be displayed in the Logcat window.

IMPORTANT!!!

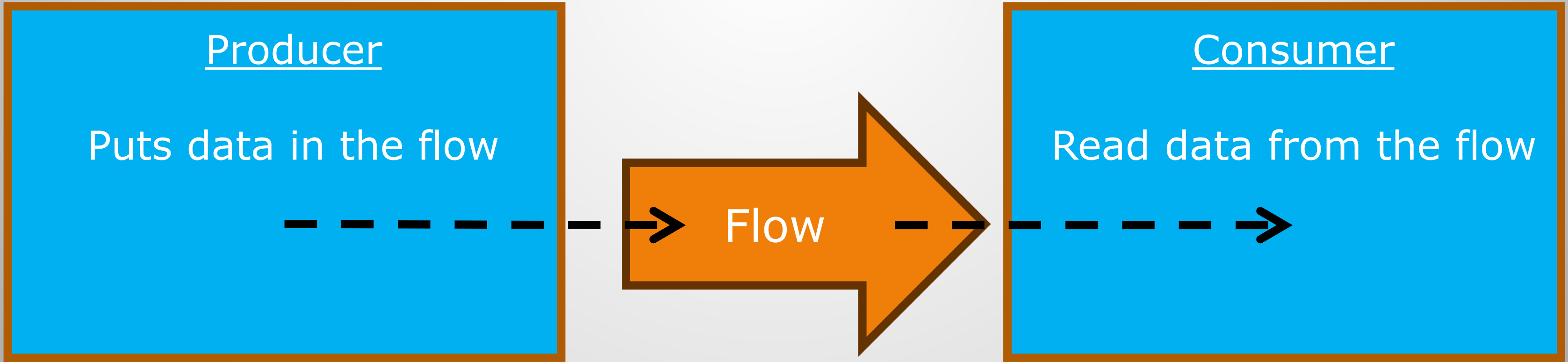
All values are returned at once. The function does not return the list until all the delays have occurred. There will be a 3 second delay then all number will appear at once.

Generate Normal List with a Delay

Kotlin Flow

- A Kotlin Flow is like a collection, but the values are returned as they appear in the flow (not all at once).
- When using flows, there is a "producer" for the flow and a "consumer" for the flow.
- The producer puts data into the flow.
- The consumer reads data from the flow.
- Here is a link describing flows:

https://developer.android.com/kotlin/flow



Kotlin Flow

Hot vs Cold Flow

- Cold Flow
  - A cold flow must have a terminal operation be called on it before values can be taken from it (collect is a terminal operation). This means it needs a "consumer" to start getting values from it.
  - A basic Kotlin Flow is cold.
  - For example: Flow<Int>

- Hot Flow
  - A hot flow is usable immediately. It has data there and ready to be used regardless of whether or not a consumer is there.
  - A StateFlow is hot.
  - For example: StateFlow<Int>

Hot vs Cold Flow

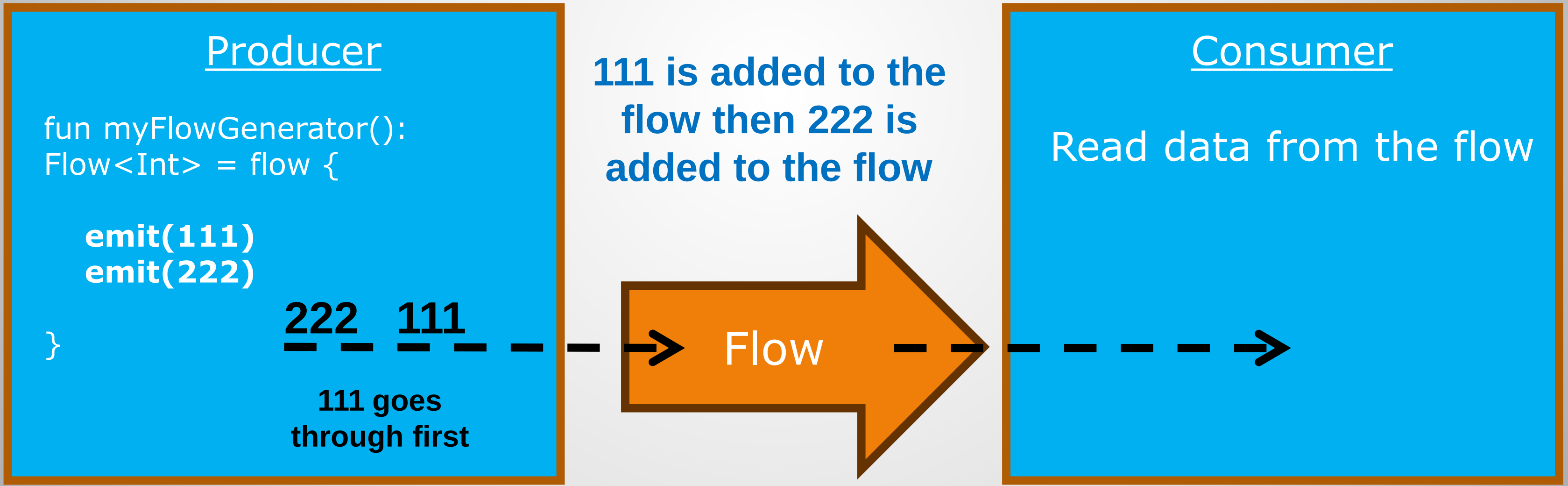
- Now on to basic flows (cold)…

Flow

Adding Data to a Flow

Adding Data to a Flow (Cold Flow)

- Call **emit** from within a flow builder to add data to the flow (this is similar to add in a normal list).
- flow { } is a flow builder.
- A flow builder is executed in a coroutine (the coroutine functionality is built into the flow builder).



```
fun myFlowGenerator(): Flow<Int> = flow {
    emit(111)
    emit(222)
}
```

Flow Builder - Function to Generate a Flow (Producer)

- A flow builder is used to put values into a flow.
- Values are added to the flow using emit.
- As soon as a value is added to the flow it is usable.
- A flow builder (flow { } block) is used to create the flow. Code inside the flow builder can be suspended.
- The code inside the flow builder only runs when the flow is collected. The code inside the flow builder runs every time the flow is collected.

This Flow contains Int values

The flow { } block is the flow builder. Code in a flow builder can be suspended (code in the flow builder is automatically run in a coroutine)

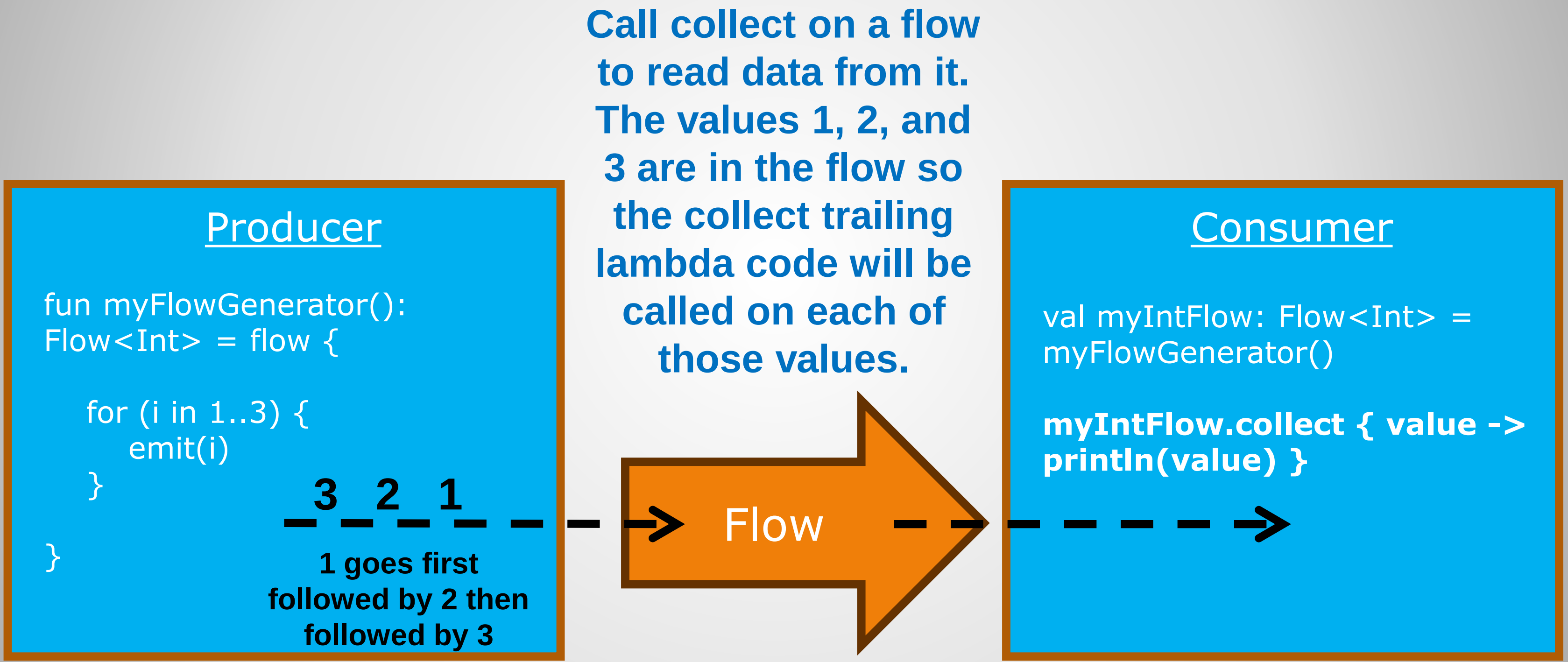
```
fun myFlowGenerator(): Flow<Int> = flow {

    for (i in 1..3) {
        emit(i)
    }
}
```

Put values in the flow using emit. Calling emit once puts one value in the flow. This loop calls emit three times putting the values 1, 2, and 3 in the flow.

Function to Generate a Flow

Collecting Data from a Flow

- Call **collect** on the flow to read data from it.


Call collect on a flow to read data from it. The values 1, 2, and 3 are in the flow so the collect trailing lambda code will be called on each of those values.
Producer
3 2 1
1 goes first followed by 2 then followed by 3
Flow
Consumer

```
fun myFlowGenerator():
Flow<Int> = flow {

    for (i in 1..3) {
        emit(i)
    }

}
```

```
val myIntFlow: Flow<Int> =
myFlowGenerator()

myIntFlow.collect { value ->
println(value) }
```

Collecting Data from a Flow

Reading from a Flow (Consumer)

- The collect function must be called on the Flow to read from it.
- A trailing lambda is passed to collect. This function is called each time a value appears in the Flow.

Flow variable

Call myFlowGenerator to get flow instance

```
val myIntFlow: Flow<Int> = myFlowGenerator()
```

IMPORTANT!!! collect is a suspend function and must be run in a coroutine (not shown here)

```
myIntFlow.collect { value -> println(value) }
```

Call collect on the flow to read data (cold flows need to have collect called on them before they can be used)

The lambda body is executed for EVERY value in the Flow (its applied to all values coming through the flow)

```
fun myFlowGenerator(): Flow<Int> =
flow {
    for (i in 1..3) {
        emit(i)
    }
}
```

The myFlowGenerator function returns a Flow<Int> (returns a flow instance)

Reading from a Flow

Try It Out – Reading from a Flow

<u>Try It Out - Reading from a Flow</u>

- Put the following code in a view model (need to create the view model class):

```
val myIntFlow: Flow<Int> = myFlowGenerator()
fun runCollectOnFlow() {
    viewModelScope.launch {
        println("Call flow collect")
        myIntFlow.collect { value -> println(value) }
    }
}
fun myFlowGenerator(): Flow<Int> =
    flow {
        for (i in 1..3) {
            emit(i)
        }
    }
```

- Call runCollectOnFlow from a composable function (assumes viewModel has been set):

```
viewModel.runCollectOnFlow()
```

Generate Flow with a Delay

- A delay is added before adding each num. This code was run in a ViewModel.

```
fun myFlowGenerator(): Flow<Int> = flow { // flow builder
    delay(1000)
    emit(1)
    delay(1000)
    emit(2)
    delay(1000)
    emit(3)
}

fun testFlow() {
    viewModelScope.launch {
        val myIntFlow: Flow<Int> = myFlowGenerator()
        myIntFlow.collect { value -> println(value) }
    }
}
```

There is a delay before emitting each number to simulate a long running operation

After calling collect, values are read when they appear in the Flow (not all at once). There will be a one second delay between printing each number.

Generate Flow with a Delay

Collecting Flow Multiple Times

```
val myIntFlow: Flow<Int> = myFlowGenerator()
fun myFlowGenerator(): Flow<Int> = flow { // flow builder
    delay(1000)
    emit(1)
    delay(1000)
    emit(2)
    delay(1000)
    emit(3)
}

fun testFlow() {
    viewModelScope.launch {
        myIntFlow.collect { value -> println(value) }
    }
}

testFlow()
testFlow()
```

Could declare the flow variable as a class member variable (like in a ViewModel) and initialize it once

Call collect on the myIntFlow member variable in this function

Each call to testFlow will cause a separate call to collect.

Important! Each call to collect will cause the flow to emit its whole sequence of values from beginning to end.

Colllecting Flow Multiple Times

Flow Item Types

- A flow can have different types of items.
- The data type inside < > determines the item type.

$$\textbf{Flow<}\underbrace{\textbf{Int}}_{\text{Takes Int values}}\textbf{>} \qquad \textbf{Flow<}\underbrace{\textbf{String}}_{\text{Takes String values}}\textbf{>}$$

- Flow can have whole collections as values.
- This means each value that comes through the flow is a whole collection.

$$\textbf{Flow<}\underbrace{\textbf{List<Int>}}_{\substack{\text{Takes List<Int> values} \\ \text{(each item is a whole list)}}}\textbf{>}$$

Flow Item Types

Flow of List

- Items in a flow can be whole collections.
- The code below shows a flow that contains lists of Int.
- Each item in this flow is a whole list of Int.

Each item in the flow is a whole list

```
fun myFlowGeneratorList() : Flow<List<Int>> = flow {

    emit(listOf(4, 7, 5, 9))
    emit(listOf(8, 1, 3, 2, 4))
    emit(listOf(5, 3, 7))

}
```

Each call to emit is passed a whole list of numbers

Flow of List

Try It Out - Generate Flow of List

- Put the following code in a view model (need to create the view model class):

```
val myListIntFlow: Flow<List<Int>> = myFlowGeneratorList()

fun myFlowGeneratorList() : Flow<List<Int>> = flow {
    emit(listOf(4, 7, 5, 9))
    emit(listOf(8, 1, 3, 2, 4))
    emit(listOf(5, 3, 7))
}

fun testFlowList() {
    viewModelScope.launch {
        myListIntFlow.collect { value -> println(value) }
    }
}
```

Send three whole lists of numbers to the flow

Each value in the flow is a whole list (not one number)

Output

```
[4, 7, 5, 9]
[8, 1, 3, 2, 4]
[5, 3, 7]
```

- Call testFlowList from a composable function (assumes viewModel has been set):

```
viewModel.testFlowList()
```

Each call to testFlowList will cause the flow of List<int> to be emitted from beginning to end.

Try It Out - Generate Flow of List

StateFlow

- Now on to StateFlow (hot)…

StateFlow (Hot Flow)

- A StateFlow represents one value (not a sequence of values).
- The one value that is stored can be updated over time.
- The one value can be a whole list of something. The list is treated as one value though.
- When the value changes any consumers of this flow are notified.
- The value in the StateFlow must be initialized when this flow is created.
- A StateFlow is hot (it is usable immediately). The value in the StateFlow is there and ready to use regardless of whether or not there are any consumers.
- Good for representing UI state. The UI can observe it and update itself when the value in the StateFlow changes.

StateFlow<Int>
Stores one int. For example:
555

StateFlow<Employee>
Stores one employee. For example:
```
{ name="Rose",
  dept="IT",
  salary=100000
}
```

StateFlow<List<Int>>
Stores one list of int. For example:
(10,20,30)

This is one instance of a List<Int> (not three Ints).

StateFlow

Example 1 - StateFlow - Set Value (MutableStateFlow<Int>)

- StateFlow and MutableStateFlow are defined in Kotlin libraries (not Jetpack Compose).
- Use the value member variable to set the value in the state flow.
- Here are member variable declarations for a state flow that stores an Int:

```
var testStateFlow = MutableStateFlow(0)
    private set
```

Declare a MutableStateFlow member variable and make the set private.

```
// Set value in flow
testStateFlow.value = 555
```

Put the number 555 in the state flow. The 0 value is overwritten.

StateFlow<Int>
Stores one int (value changes from 0 to 555) :
~~0~~
555

A notification is sent out when the value is changed (for example, code in the UI is notified)

Check next slide…

Example 1 - StateFlow - Set Value (MutableStateFlow<Int>)

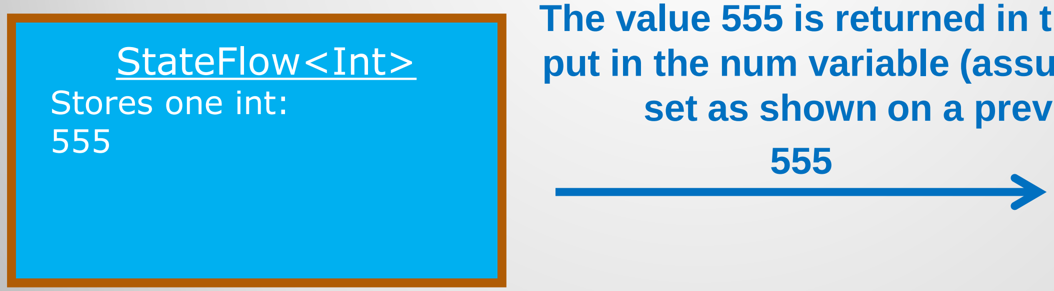
Example 1 - StateFlow - Collect Value (MutableStateFlow<Int>)

- Use the collectAsState function to get the current state from the state flow.
- collectAsState converts Kotlin state flow to Jetpack Compose state.
- collectAsState is a composable function so it must be called from inside another composable function (cannot be called from the view model).
- Here is collection code for a composable function:

num has type Int

Call collectAsState on the flow. It returns a State<Int> type in this case. Use the value member of that state object to get the num.

```
val num = viewModel.testStateFlow.collectAsState().value
```



The value 555 is returned in this example and put in the num variable (assuming that it was set as shown on a previous slide)

Example 1 - StateFlow - Collect Value (MutableStateFlow<Int>)

Try It Out – Example 1 StateFlow

- Put the following code in a view model (need to create the view model class):

```
var testStateFlow = MutableStateFlow(0)
    private set

fun setStateFlowValue(num:Int) {
    // Set value in flow
    testStateFlow.value = num
}
```

Output

It should initially display 0 (default value in StateFlow). After pressing the button, it should automatically display 555.

- Set and display the value of the StateFlow in a composable:

```
val viewModel = viewModel { MainScreenViewModel() }
val num = viewModel.testStateFlow.collectAsState().value

Column(modifier) {
    Text(num.toString())
    Button(onClick = { viewModel.setStateFlowValue(555) })
    { Text("Set Value to 555") }
}
```

Try It Out – Example 1 StateFlow

Example 2 - StateFlow - Set Value (MutableStateFlow<List<Int>>)

- This example uses a whole list as the one value being stored.
- Preferences DataStore, Room, and Firestore all use StateFlows where the one value being stored is a list.

The type of items is List<Int>. Pass in an empty list as the initial value in the flow.

```
var testListStateFlow = MutableStateFlow(listOf(10, 20, 30))
    private set
```

```
// Set value in flow
testListStateFlow.value = listOf(40,50,60)
```

Put the list (40,50,60) in the state flow variable. The (10,20,30) list value is overwritten.

StateFlow<List<Int>>

Stores one List<Int> :

~~(10,20,30)~~

(40,50,60)

A notification is sent out when the value is changed (for example, code in the UI is notified)

Check next slide…

Example 2 - StateFlow - Set Value (MutableStateFlow<List<Int>>)

Example 2 - StateFlow - Collect Value (StateFlow<List<Int>>)

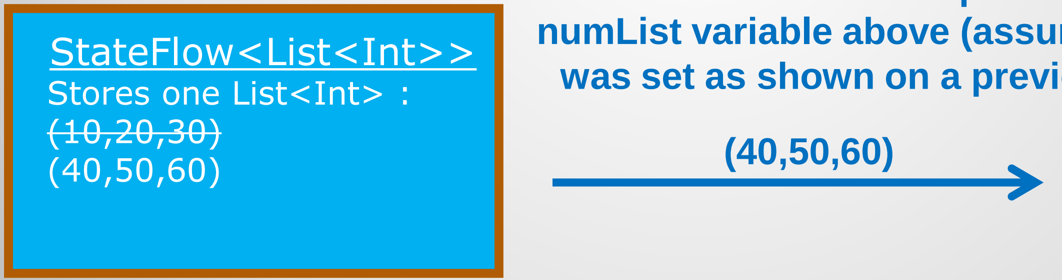
- Use the collectAsState function to get the current state from the state flow.
- Here is collection code for a composable function:

list has type List<Int>

Call collectAsState on the flow. It returns a State<List<Int>> type. Use the value member of that state object to get the list .

```
val list = viewModel.testListStateFlow.collectAsState().value
```

Collecting causes the value (40,50,60) to be returned in this example and put in the numList variable above (assuming that it was set as shown on a previous slide)



Example 2 - StateFlow - Collect Value (StateFlow<List<Int>>)

<u>Try It Out – Example 2 StateFlow</u>

- Put the following code in a view model (need to create the view model class):

```
var testListStateFlow = MutableStateFlow(listOf(10, 20, 30))
    private set

fun setStateFlowValue(list:List<Int>) {
    // Set value in flow
    testListStateFlow.value = list
}
```

<u>Output</u>

It should initially display [10,20,30] (default value in StateFlow). After pressing the button, it should automatically display [40,50,60].

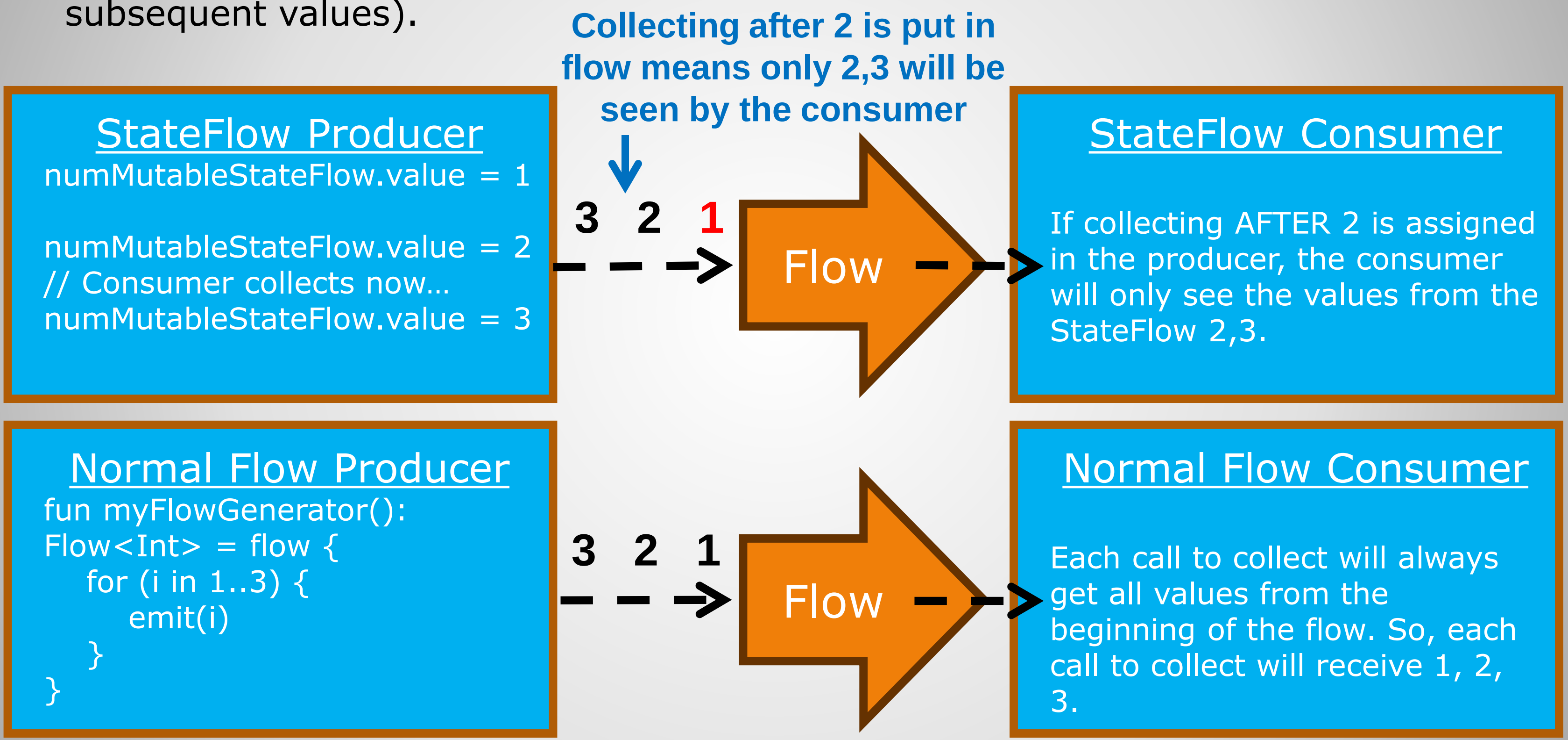
- Set and display the value of the StateFlow in a composable:

```
val viewModel = viewModel { MainScreenViewModel() }
val list = viewModel.testListStateFlow.collectAsState().value

Column(modifier) {
    Text(list.toString())
    Button(onClick = { viewModel.setStateFlowValue(listOf(40,50,60)) })
    { Text("Set Value to list 40,50,60") }
}
```

Try It Out – Example 2 StateFlow

StateFlow vs Normal Flow

- When collect is called on a normal flow, the consumer causes the producer to generate ALL values from the beginning of the flow.
- When collecting a StateFlow, the consumer only gets the current value (and subsequent values).


Collecting after 2 is put in flow means only 2,3 will be seen by the consumer
3 2 1
Flow
3 2 1
Flow

StateFlow Producer

```
numMutableStateFlow.value = 1

numMutableStateFlow.value = 2
// Consumer collects now…
numMutableStateFlow.value = 3
```

StateFlow Consumer

If collecting AFTER 2 is assigned in the producer, the consumer will only see the values from the StateFlow 2,3.

Normal Flow Producer

```
fun myFlowGenerator():
Flow<Int> = flow {
    for (i in 1..3) {
        emit(i)
    }
}
```

Normal Flow Consumer

Each call to collect will always get all values from the beginning of the flow. So, each call to collect will receive 1, 2, 3.

StateFlow vs Normal Flow

- End of Slides

End of Slides