



Lakehead
U N I V E R S I T Y

Teaching
Commons

Mobile Computing Technology

February 5, 2025

Conditional branching in **KOTLIN**



More of
KOTLIN

If-Statement

```
if ( condition ) {  
    body  
}
```

If-Statement

- Use the if keyword followed by the condition that you want to evaluate.
- You need to express the condition with a boolean expression.
- Expressions combine values, variables, and operators that return a value.
- Boolean expressions return a boolean value

```
if (  condition ) {  
     body  
}
```

If-Statement

```
1 == 1
```



The `==` comparison operator compares the values to each other. Which boolean value do you think this expression returns?

```
if ( condition ) {  
    body  
}
```

Comparison Operator

Find the boolean value of this expression:

1. Use [Kotlin Playground](#) to run your code.
2. In the function body, add a `println()` function and then pass it the `1 == 1` expression as an argument:

```
fun main() {  
    println(1 == 1)  
}
```



Boolean Operators

Besides the `==` comparison operator, there are additional *comparison operators* that you can use to create boolean expressions:

- Less than: `<`
- Greater than: `>`
- Less than or equal to: `<=`
- Greater than or equal to: `>=`
- Not equal to: `!=`

Practice the use of comparison operators with simple expressions:

1. In the argument, replace the `==` comparison operator with the `<` comparison operator:

```
fun main() {  
    println(1 < 1)  
}
```



Boolean Operators

Besides the `==` comparison operator, there are additional *comparison operators* that you can use to create boolean expressions:

- Less than: `<`
- Greater than: `>`
- Less than or equal to: `<=`
- Greater than or equal to: `>=`
- Not equal to: `!=`

Practice the use of comparison operators with simple expressions:

1. In the argument, replace the `==` comparison operator with the `<` comparison operator:

```
fun main() {  
    println(1 < 1)  
}
```



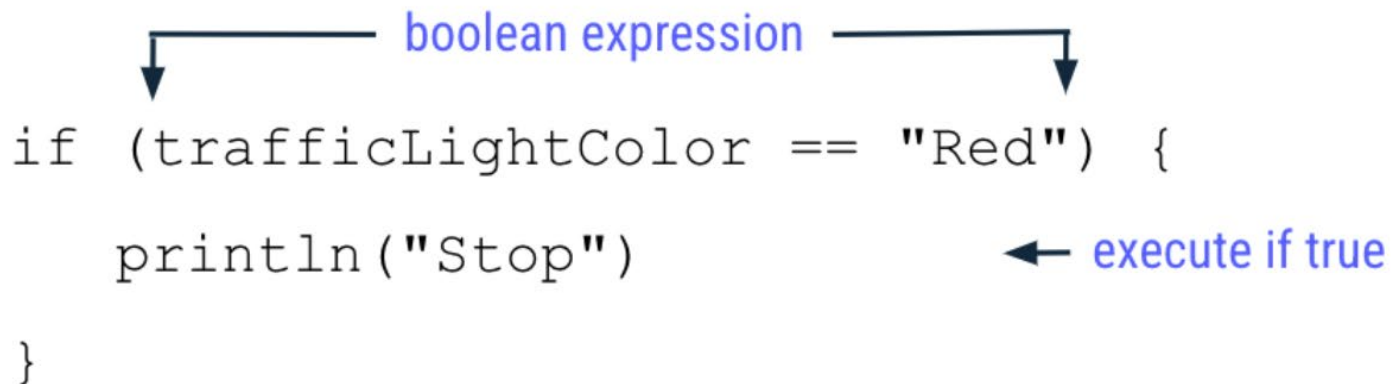
If-Statement: Try it

```
fun main() {  
    val trafficLightColor = "Red"  
  
    if (trafficLightColor == "Red") {  
        println("Stop")  
    }  
}
```



If-Statement

The `trafficLightColor == "Red"` expression returns a `true` value, so the `println("Stop")` statement is executed, which prints the `Stop` message.



The diagram illustrates the execution of an if-statement. A blue label "boolean expression" is positioned above the condition `trafficLightColor == "Red"`. Two arrows originate from this label: one points down to the opening curly brace of the if-statement, and the other points down to the closing curly brace. A second blue label, "execute if true", is positioned to the right of the `println("Stop")` statement, with an arrow pointing left towards it.

```
if (trafficLightColor == "Red") {  
    println("Stop")  
}
```

If-Else-Statement

You need to add an `else` branch to create an `if/else` statement. A *branch* is an incomplete part of code that you can join to form statements or expressions. An `else` branch needs to follow an `if` branch.

```
if ( condition ) {  
    body 1  
} else {  
    body 2  
}
```

After the closing curly brace of the `if` statement, you add the `else` keyword followed by a pair of curly braces. Inside the curly braces of the `else` statement, you can add a second body that only executes when the condition in the `if` branch is false.

If-Else-Statement: Try it

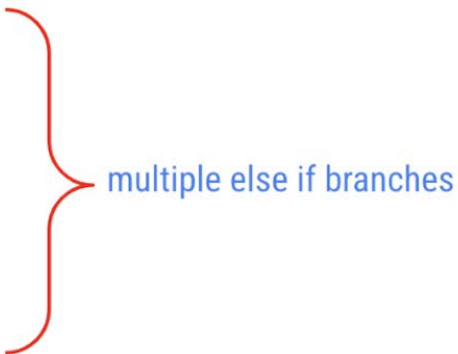
```
fun main() {  
    val trafficLightColor = "Red"  
  
    if (trafficLightColor == "Red") {  
        println("Stop")  
    } else {  
        println("Go")  
    }  
}
```



Else-If-Branch

The `else if` branch is always located after the `if` branch, but before the `else` branch. You can use multiple `else if` branches in a statement:

```
if ( condition 1 ) {  
    body 1  
} else if ( condition 2 ) {  
    body 2  
} else if ( condition 3 ) {  
    body 3  
} else {  
    body 4  
}
```



multiple else if branches

Else-If-Branch: Try it

```
fun main() {  
    val trafficLightColor = "Yellow"  
  
    if (trafficLightColor == "Red") {  
        println("Stop")  
    } else if (trafficLightColor == "Yellow") {  
        println("Slow")  
    } else {  
        println("Go")  
    }  
}
```

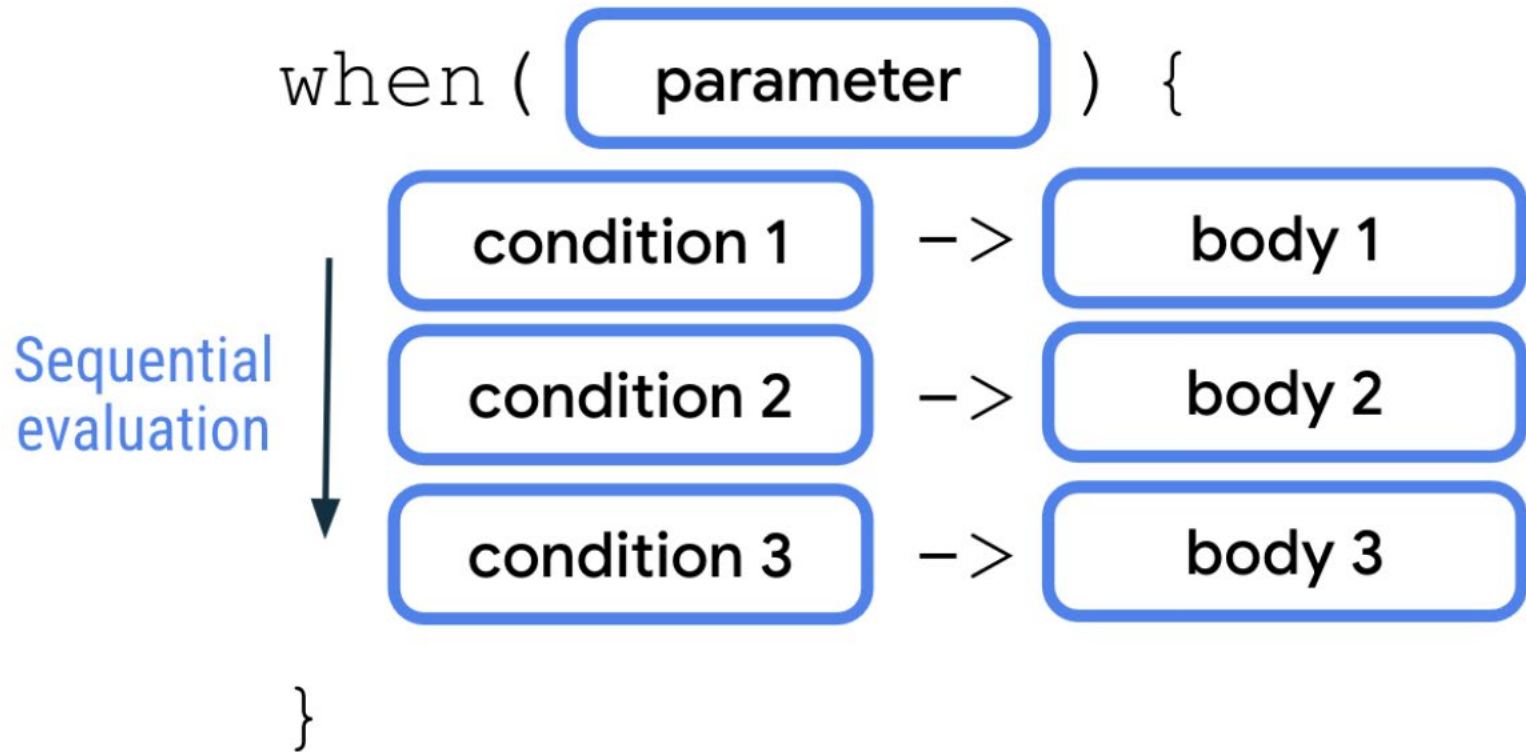


Else-If-Branch: Try it

```
fun main() {  
    val trafficLightColor = "Black"  
  
    if (trafficLightColor == "Red") {  
        println("Stop")  
    } else if (trafficLightColor == "Yellow") {  
        println("Slow")  
    } else if (trafficLightColor == "Green") {  
        println("Go")  
    } else {  
        println("Invalid traffic-light color")  
    }  
}
```



When-Statement



When-Statement

A **when** statement accepts a single value through the parameter. The value is then evaluated against each of the conditions sequentially. The corresponding body of the first condition that's met is then executed. Each condition and body are separated by an arrow (**->**). Similar to **if/else** statements, each pair of condition and body is called a branch in **when** statements. Also similar to the **if/else** statement, you can add an **else** branch as your final condition in a **when** statement that works as a catchall branch.

When-Statement: Try it

```
fun main() {  
    val x = 3  
  
    when (x) {  
        2 -> println("x is a prime number between 1 and 10.")  
        3 -> println("x is a prime number between 1 and 10.")  
        5 -> println("x is a prime number between 1 and 10.")  
        7 -> println("x is a prime number between 1 and 10.")  
        else -> println("x isn't a prime number between 1 and 10.")  
    }  
}
```



When-Statement: Try it

```
fun main() {  
    val x = 3  
  
    when (x) {  
        2, 3, 5, 7 -> println("x is a prime number between 1 and 10.")  
        3 -> println("x is a prime number between 1 and 10.")  
        5 -> println("x is a prime number between 1 and 10.")  
        7 -> println("x is a prime number between 1 and 10.")  
        else -> println("x isn't a prime number between 1 and 10.")  
    }  
}
```



When-Statement

Use the **in** keyword for a range of conditions

Besides the comma (,) symbol to denote multiple conditions, you can also use the **in** keyword and a range of values in **when** branches.

```
when ( parameter ) {  
  in range start . . range end -> body 1  
  condition 2 -> body 2  
}
```

When-Statement: Try it

```
fun main() {  
    val x = 4  
  
    when (x) {  
        2, 3, 5, 7 -> println("x is a prime number between 1 and 10.")  
        in 1..10 -> println("x is a number between 1 and 10, but not a prime number.")  
        else -> println("x isn't a prime number between 1 and 10.")  
    }  
}
```



The is-keyword

Use the `is` keyword to check data type

You can use the `is` keyword as a condition to check the data type of an evaluated value.

```
when ( parameter ) {  
  is type -> body 1  
  condition 2 -> body 2  
}
```

is-keyword: Try it

```
fun main() {  
    val x: Any = 20  
  
    when (x) {  
        2, 3, 5, 7 -> println("x is a prime number between 1 and 10.")  
        in 1..10 -> println("x is a number between 1 and 10, but not a prime number.")  
        is Int -> println("x is an integer number, but not between 1 and 10.")  
        else -> println("x isn't an integer number.")  
    }  
}
```



If-Else-Expression: Conditionals as expressions

```
val name = if ( condition ) expression 1 else expression 2
```

Conditionals as expressions: Try it

```
fun main() {  
    val trafficLightColor = "Black"  
  
    val message = if (trafficLightColor == "Red") {  
        println("Stop")  
    } else if (trafficLightColor == "Yellow") {  
        println("Slow")  
    } else if (trafficLightColor == "Green") {  
        println("Go")  
    } else {  
        println("Invalid traffic-light color")  
    }  
}
```



Conditionals as expressions: Try it

```
fun main() {  
    val trafficLightColor = "Black"  
  
    val message =  
        if (trafficLightColor == "Red") "Stop"  
        else if (trafficLightColor == "Yellow") "Slow"  
        else if (trafficLightColor == "Green") "Go"  
        else "Invalid traffic-light color"  
  
    println(message)  
}
```



Conditionals as expressions: Try it

```
fun main() {  
    val trafficLightColor = "Amber"  
  
    val message = when(trafficLightColor) {  
        "Red" -> "Stop"  
        "Yellow", "Amber" -> "Slow"  
        "Green" -> "Go"  
        else -> "Invalid traffic-light color"  
    }  
    println(message)  
}
```



Summary

- In Kotlin, branching can be achieved with `if/else` or `when` conditionals.
- The body of an `if` branch in an `if/else` conditional is only executed when the boolean expression inside the `if` branch condition returns a `true` value.
- Subsequent `else if` branches in an `if/else` conditional get executed only when previous `if` or `else if` branches return `false` values.
- The final `else` branch in an `if/else` conditional only gets executed when all previous `if` or `else if` branches return `false` values.
- It's recommended to use the `when` conditional to replace an `if/else` conditional when there are more than two branches.
- You can write more complex conditions in `when` conditionals with the comma (,), `in` ranges, and the `is` keyword.
- `if/else` and `when` conditionals can work as either statements or expressions.



Lakehead
U N I V E R S I T Y

Teaching
Commons

Mobile Computing Technology

March 10, 2025

How to use
SENSORS

Android Sensor Framework

The Android platform supports three broad categories of hardware-based as well as some virtual sensors:

- **Motion sensors**
 - Measure acceleration forces and rotational forces along three axes.
 - accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.
- **Environmental sensors**
 - Measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity.
 - barometers, photometers, and thermometers.
- **Position sensors**
 - Measure the physical position of a device.
 - orientation sensors and magnetometers.

Android Sensor Framework

The Android platform provides

- **Access to sensors available on the device to acquire raw sensor data**
- **Several classes and interfaces to do the following:**
 - Determine which sensors are available on a device.
 - Determine an individual sensor's capabilities, such as its maximum range, manufacturer, power requirements, and resolution.
 - Acquire raw sensor data and define the minimum rate at which you acquire sensor data.
 - **Register and unregister sensor event listeners that monitor sensor changes.**

Android Sensor Framework

Table 1. Sensor types supported by the Android platform.

Sensor	Type	Description	Common Uses
<code>TYPE_ACCELEROMETER</code>	Hardware	Measures the acceleration force in m/s^2 that is applied to a device on all three physical axes (x, y, and z), including the force of gravity.	Motion detection (shake, tilt, etc.).
<code>TYPE_AMBIENT_TEMPERATURE</code>	Hardware	Measures the ambient room temperature in degrees Celsius ($^{\circ}\text{C}$). See note below.	Monitoring air temperatures.
<code>TYPE_GRAVITY</code>	Software or Hardware	Measures the force of gravity in m/s^2 that is applied to a device on all three physical axes (x, y, and z).	Motion detection (shake, tilt, etc.).
<code>TYPE_GYROSCOPE</code>	Hardware	Measures a device's rate of rotation in rad/s around each of the three physical axes (x, y, and z).	Rotation detection (spin, turn, etc.).
<code>TYPE_LIGHT</code>	Hardware	Measures the ambient light level (illumination) in lx.	Controlling screen brightness.
<code>TYPE_LINEAR_ACCELERATION</code>	Software or Hardware	Measures the acceleration force in m/s^2 that is applied to a device on all three physical axes (x, y, and z), excluding the force of gravity.	Monitoring acceleration along a single axis.
<code>TYPE_MAGNETIC_FIELD</code>	Hardware	Measures the ambient geomagnetic field for all three physical axes (x, y, z) in μT .	Creating a compass.
<code>TYPE_ORIENTATION</code>	Software	Measures degrees of rotation that a device makes around all three physical axes (x, y, z). As of API level 3 you can obtain the inclination matrix and rotation matrix for a device by using the gravity sensor and the geomagnetic field sensor in conjunction with the <code>getRotationMatrix()</code> method.	Determining device position.

Android Sensor Framework

TYPE_PRESSURE	Hardware	Measures the ambient air pressure in hPa or mbar.	Monitoring air pressure changes.
TYPE_PROXIMITY	Hardware	Measures the proximity of an object in cm relative to the view screen of a device. This sensor is typically used to determine whether a handset is being held up to a person's ear.	Phone position during a call.
TYPE_RELATIVE_HUMIDITY	Hardware	Measures the relative ambient humidity in percent (%).	Monitoring dewpoint, absolute, and relative humidity.
TYPE_ROTATION_VECTOR	Software or Hardware	Measures the orientation of a device by providing the three elements of the device's rotation vector.	Motion detection and rotation detection.
TYPE_TEMPERATURE	Hardware	Measures the temperature of the device in degrees Celsius (°C). This sensor implementation varies across devices and this sensor was replaced with the TYPE_AMBIENT_TEMPERATURE sensor in API Level 14	Monitoring temperatures.

Android Sensor Framework

Sensor Framework

You can access these sensors and acquire raw sensor data by using the Android sensor framework. The sensor framework is part of the `android.hardware` package and includes the following classes and interfaces:

SensorManager

You can use this class to create an instance of the sensor service. This class provides various methods for accessing and listing sensors, registering and unregistering sensor event listeners, and acquiring orientation information. This class also provides several sensor constants that are used to report sensor accuracy, set data acquisition rates, and calibrate sensors.

Sensor

You can use this class to create an instance of a specific sensor. This class provides various methods that let you determine a sensor's capabilities.

SensorEvent

The system uses this class to create a sensor event object, which provides information about a sensor event. A sensor event object includes the following information: the raw sensor data, the type of sensor that generated the event, the accuracy of the data, and the timestamp for the event.

SensorEventListener

You can use this interface to create two callback methods that receive notifications (sensor events) when sensor values change or when sensor accuracy changes.

Android Sensor Framework

If you want to list all of the sensors of a given type, you could use another constant instead of `TYPE_ALL` such as `TYPE_GYROSCOPE`, `TYPE_LINEAR_ACCELERATION`, or `TYPE_GRAVITY`.

You can also determine whether a specific type of sensor exists on a device by using the `getDefaultSensor()` method and passing in the type constant for a specific sensor. If a device has more than one sensor of a given type, one of the sensors must be designated as the default sensor. If a default sensor does not exist for a given type of sensor, the method call returns null, which means the device does not have that type of sensor. For example, the following code checks whether there's a magnetometer on a device:

Kotlin

Java

```
private lateinit var sensorManager: SensorManager
...
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
if (sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD) != null) {
    // Success! There's a magnetometer.
} else {
    // Failure! No magnetometer.
}
```

Android Sensor Framework

ToDo

- Create a project in Android Studio using the attached code
- Build and run the app
- Determine:
 - What type of data does your proximity sensor provide? Binary ("near" or "far"), numerical, ordinal?
 - Compare results with other students
 - Change the output text
 - Replace proximity sensor with temperature sensor

