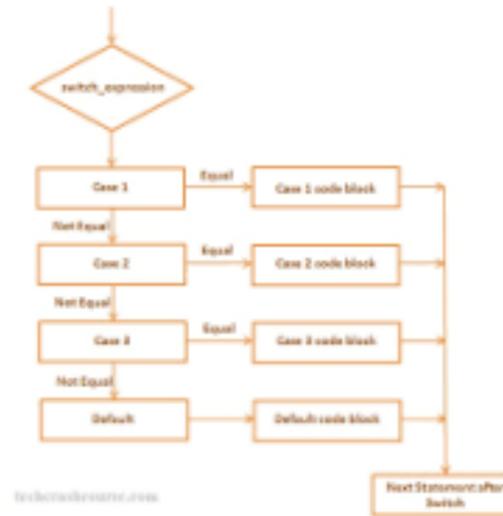# Control Flow

Kotlin Control Flow

Switch Statement Flow Diagram

if, when, for and while statements

If Expression

In Kotlin, `if` is an expression, i.e. it returns a value. Therefore there is no ternary operator (condition ? then : else), because ordinary `if` works fine in this role.

```kotlin
// Traditional usage
var max = a
if (a < b) max = b

// With else
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}

// As expression
val max = if (a > b) a else b
```

`if` branches can be blocks, and the last expression is the value of a block:

```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

If you're using `if` as an expression rather than a statement (for example, returning its value or assigning it to a variable), the expression is required to have an `else` branch.

When Expression

when replaces the switch operator of C-like languages. In the simplest form it looks like this

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // Note the block
        print("x is neither 1 nor 2")
    }
}
```

when matches its argument against all branches sequentially until some branch condition is satisfied. when can be used either as an expression or as a statement. If it is used as an expression, the value of the satisfied branch becomes the value of the overall expression. If it is used as a statement, the values of individual branches are ignored. (Just like with `if`, each branch can be a block, and its value is the value of the last expression in the block.)

The `else` branch is evaluated if none of the other branch conditions are satisfied. If `when` is used as an expression, the `else` branch is mandatory, unless the compiler can prove that all possible cases are covered with branch conditions (as, for example, with enum class entries and sealed class subtypes).

If many cases should be handled in the same way, the branch conditions may be combined with a comma:

```kotlin
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

We can use arbitrary expressions (not only constants) as branch conditions

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}
```

We can also check a value for being **in** or **!in** a range or a collection:

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

Another possibility is to check that a value `is` or `!is` of a particular type. Note that, due to smart casts, you can access the methods and properties of the type without any extra checks.

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

`when` can also be used as a replacement for an `if-else if` chain. If no argument is supplied, the branch conditions are simply boolean expressions, and a branch is executed when its condition is true:

```
when {
    x.isOdd() -> print("x is odd")
    x.isEven() -> print("x is even")
    else -> print("x is funny")
}
```

For Loops

`for` loop iterates through anything that provides an iterator. This is equivalent to the `foreach` loop in languages like C#. The syntax is as follows:

```
for (item in collection) print(item)
```

The body can be a block.

```
for (item: Int in ints) {
    // ...
}
```

As mentioned before, `for` iterates through anything that provides an iterator, i.e.

— has a member- or extension-function `iterator()`, whose return type

   — has a member- or extension-function `next()`, and

   — has a member- or extension-function `hasNext()` that returns `Boolean`.

All of these three functions need to be marked as `operator`.

To iterate over a range of numbers, use a [range expression](range%20expression):
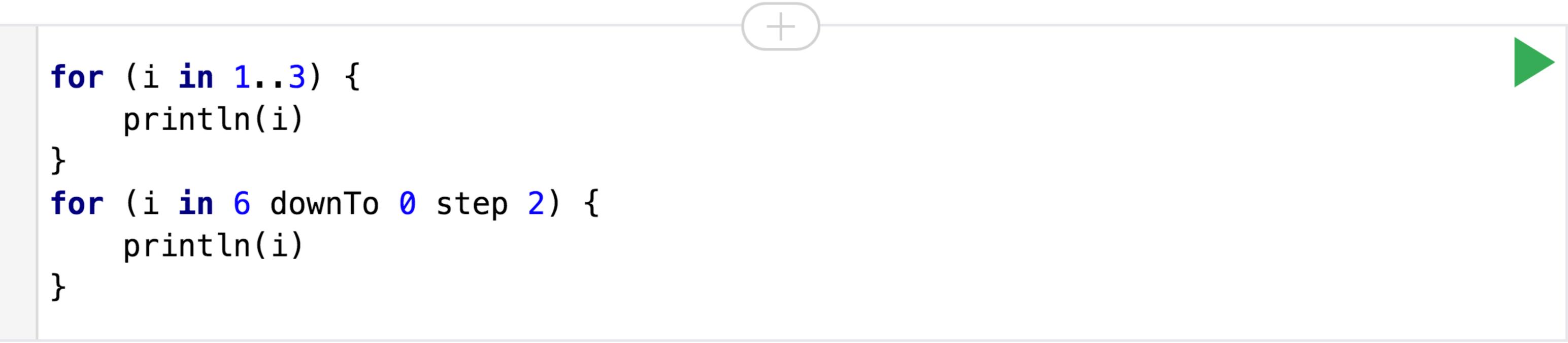
```kotlin
for (i in 1..3) {
    println(i)
}
for (i in 6 downTo 0 step 2) {
    println(i)
}
```

A `for` loop over a range or an array is compiled to an index-based loop that does not create an iterator object.

If you want to iterate through an array or a list with an index, you can do it this way:

```kotlin
for (i in array.indices) {
    println(array[i])
}
```

Alternatively, you can use the `withIndex` library function:

```kotlin
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

While Loops

while and do..while work as usual

```
while (x > 0) {
    x--
}


do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

Break and continue in loops

Kotlin supports traditional `break` and `continue` operators in loops.