Operator overloading and extension methods

```
data class Point(val first: Int, val second: Int)
```

Adding Points together

```
data class Point(val first: Int, val second: Int) {
    fun add(other: Point): Point = Point(
        this.first + other.first,
        this.second + other.second,
    )
}
```

`this` refers to the receiving object (the object on which a method is called).

```
val p1 = Point(1, 2)
val p2 = Point(2, 3)
println(p1.add(p2))
```

Looks cumbersome.

```


data class Point(val x: Int, val y: Int) {
    operator fun plus(other: Point): Point {
        val x1 = this.x + other.x
        val y1 = this.y + other.y
        return Point(x1, y1)
    }

    operator fun plus(scalar: Int): Point =
        Point(x + scalar, y + scalar)

    operator fun times(other: Point): Point {
        val x1 = this.x * other.x
        val y1 = this.y * other.y
        return Point(x1, y1)
    }
}

fun main() {
    println(Point(5, 6) + Point(7, 8)) // (12, 14)
    println(Point(5, 6) * Point(7, 8)) // (35, 48)
    println(Point(5, 6) + 3) // (8, 9)
}
```

Is `println(10 * Point(1, 2))` supported?

No: we do not have an overload of `*` that works on Int and Point.

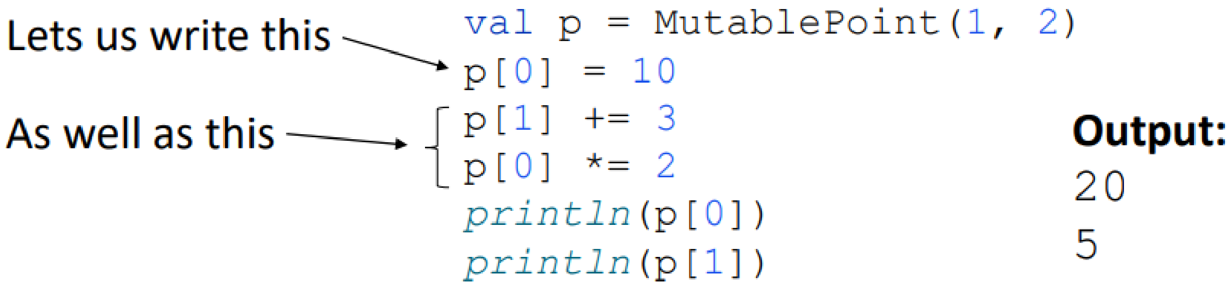
Order matters: this overload is in the Point class, so the first argument is the receiving object – a Point.

Overriding the “get” variant of `[]`

```
operator fun get(index: Int): Int =
    when (index) {
        0 -> x
        1 -> y
        else -> throw IndexOutOfBoundsException()
    }
```

Overriding the “set” variant of `[]`

```
class MutablePoint(private var first: Int, private var second: Int)
{
    operator fun set(index: Int, value: Int) {
        when (index) {
            0 -> first = value
            1 -> second = value
            else -> throw IndexOutOfBoundsException()
        }
    }
}
```



```
val p = MutablePoint(1, 2)
p[0] = 10
p[1] += 3
p[0] *= 2
println(p[0])
println(p[1])
```

Output:

```
20
5
```

The `[]` operator

The [] operators can take multiple indices, and indices need not be integers.

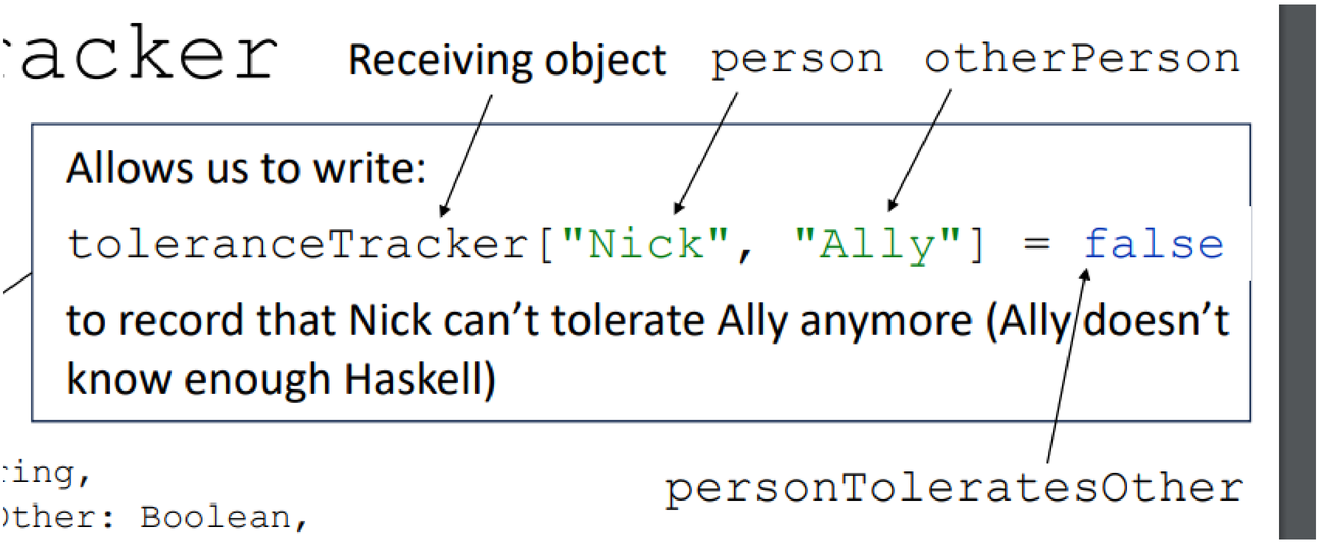
Let’s write a `ToleranceTracker` class:

- Tracks who can tolerate whom
- For people A and B (represented as strings), we can have:
  - A can tolerate B
  - A cannot tolerate B
  - Status is unknown because we lack tolerance information for A

```
enum class ToleranceStatus {
    CAN_TOLERATE,
    CANNOT_TOLERATE,
    UNKNOWN,
}
class ToleranceTracker {
    private val canTolerate: MutableMap<String, MutableSet<String>>
=
        mutableMapOf()

    operator fun get(person: String, otherPerson: String):
ToleranceStatus =
        canTolerate[person]?.let { tolerates ->
            if (tolerates.contains(otherPerson)) {
                ToleranceStatus.CAN_TOLERATE
            } else {
                ToleranceStatus.CANNOT_TOLERATE
            }
        } ?: ToleranceStatus.UNKNOWN

    operator fun set(
        person: String,
        otherPerson: String,
        personToleratesOther: Boolean,
    ) {
        val toleratedByPerson: MutableSet<String> =
            canTolerate.getOrPut(person) { mutableSetOf() }
        if (personToleratesOther) {
            toleratedByPerson.add(otherPerson)
        } else {
            toleratedByPerson.remove(otherPerson)
        }
    }
}
```

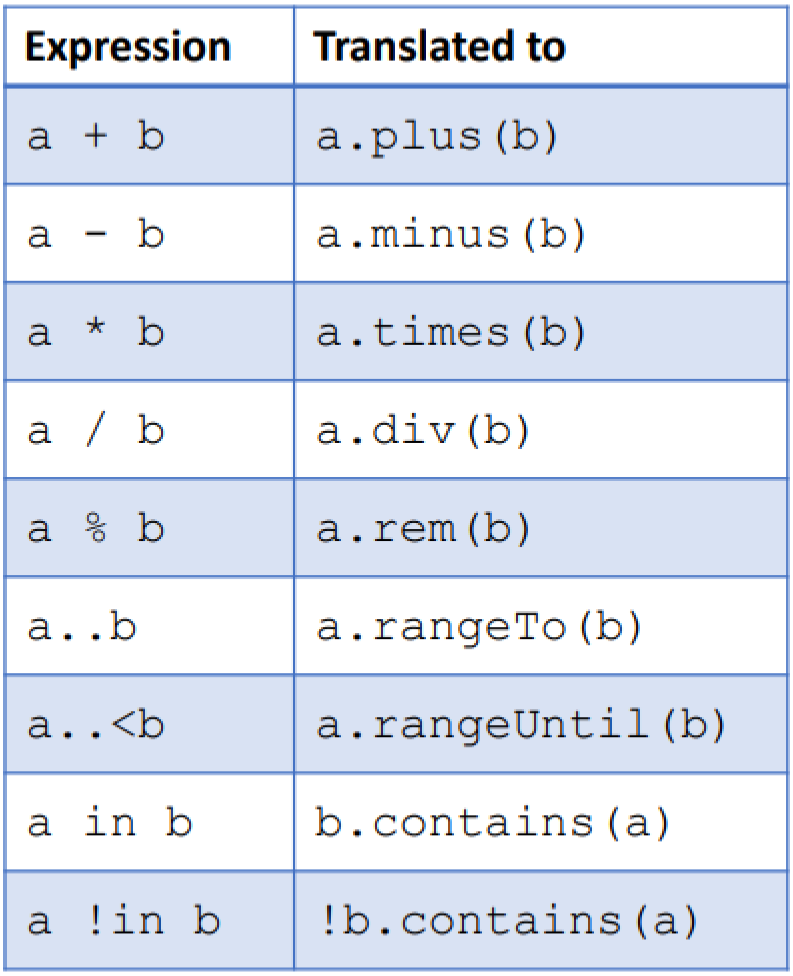


Allows us to write:

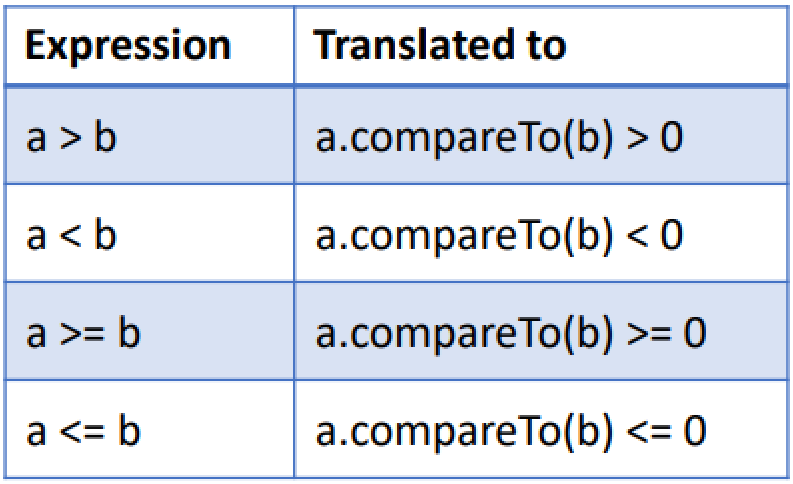
```
toleranceTracker["Nick", "Ally"] = false
```

to record that Nick can’t tolerate Ally anymore (Ally doesn’t know enough Haskell)

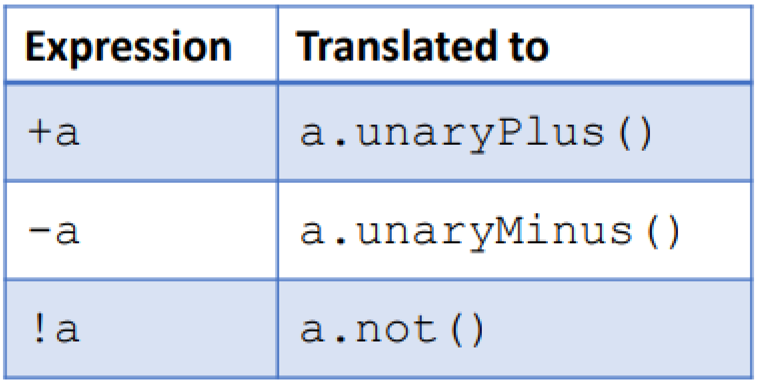
Operators that you can overload



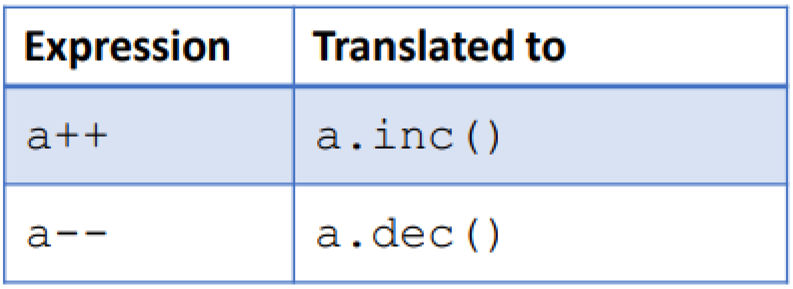
Expression	Translated to
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`
`a * b`	`a.times(b)`
`a / b`	`a.div(b)`
`a % b`	`a.rem(b)`
`a..b`	`a.rangeTo(b)`
`a..<b`	`a.rangeUntil(b)`
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`



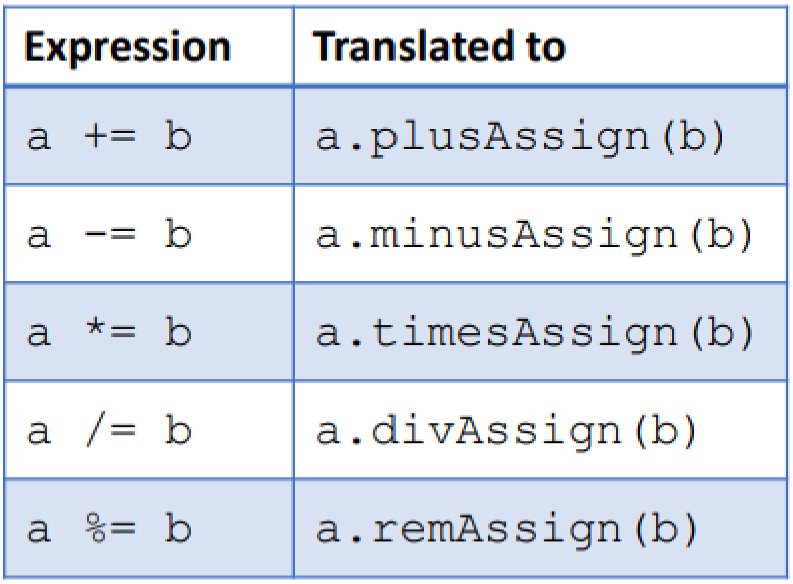
Expression	Translated to
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0



Expression	Translated to
`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`



Expression	Translated to
`a++`	`a.inc()`
`a--`	`a.dec()`



Expression	Translated to
`a += b`	`a.plusAssign(b)`
`a -= b`	`a.minusAssign(b)`
`a *= b`	`a.timesAssign(b)`
`a /= b`	`a.divAssign(b)`
`a %= b`	`a.remAssign(b)`