Operator overloading and extension methods

Alastair F. Donaldson

Aims of this lecture

- Introduce **operator overloading**
- Introduce **infix functions**
- Introduce **extension methods** as a way to add methods on existing classes and interfaces
- Study combinations of these – e.g. overloading an operator via an extension method
- Study extension methods on generic classes and interfaces

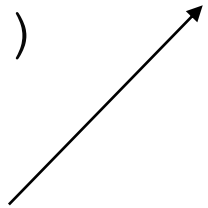
`Point` class

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

Aside: this is one of the first times a `Point` data type this has been used as an example in a university programming course

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

Inside the `Point` class, `first` and `this.first` mean the same thing, so `this` is not required above

Still, arguably clearer to write `this.first`, to explicitly distinguish from `other.first`

This looks a bit cumbersome

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

Aside: `p1` is (a reference to) the **receiving object**

Program output:

```
Point(first=3, second=5)
```

Nicer if we could write:

```
println(p1 + p2)
```

Overloading + for `Point`

The `operator` keyword indicates that we want to overload an operator

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

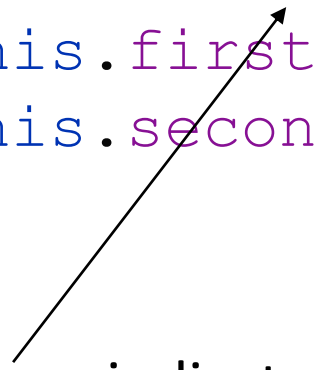
By naming the method `plus`, we indicate that we are overloading the + operator

Now we can apply + to `Points`: *println*`(p1 + p2)` works!

Overloading * for `Point`

If we want to allow component-wise multiplication of points, we can do:

```
data class Point(val first: Int, val second: Int) {
    ...
    operator fun times(other: Point): Point = Point(
        this.first * other.first,
        this.second * other.second,
    )
}
```



The name `times` indicates that we are overloading *

Now *println*`(Point(1, 2) * Point(2, 3))` prints:

```
Point(first=2, second=6)
```

Arguments to operator can differ

```
data class Point(val first: Int, val second: Int) {
    ...
    operator fun times(other: Point): Point = Point(
        this.first * other.first,
        this.second * other.second,
    )

    operator fun times(value: Int): Point = Point(
        first * value,
        second * value,
    )
}
```

Two overloads of * operator

Allows us to multiply a `Point` by an `Int`: `println(Point(1, 2) * 10)`

Prints: `Point(first=10, second=20)`

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

```
data class Point(val first: Int, val second: Int) {
    ...
    operator fun times(other: Point): Point = Point(
        this.first * other.first,
        this.second * other.second,
    )

    operator fun times(value: Int): Point = Point(
        first * value,
        second * value,
    )
}
```

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`

This could be useful syntax:

```
val p1 = Point(1, 2)
val p2 = Point(2, 3)
println(p1[0])
println(p2[1])
```

Desired output:

```
1
3
```

Let's allow it by overloading the `[]` operator

Overloading [ ]

The `operator` keyword indicates that we want to overload an operator

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

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

By naming the method `get`, we indicate that we are overloading the `[]` operator

Overriding the “set” variant of [ ]

```
class MutablePoint(private var first: Int, private var second: Int) {
    operator fun get(index: Int): Int = // Same as for Point

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

Lets us write this →

As well as this →

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

Output:

```
20
5
```

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**

ToleranceTracker

```
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
```

Receiving object `person` `otherPerson`

Allows us to write:

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

to find out whether Nick tolerates Ally

ToleranceTracker

Receiving object person otherPerson

Allows us to write:

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

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

personTolerates Other

```
    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)
        }
    }
}
```

ToleranceTracker

```
val toleranceTracker = ToleranceTracker()
toleranceTracker["Ally", "Nick"] = true
toleranceTracker["Ally", "Rishi"] = false
println(toleranceTracker["Ally", "Nick"])
println(toleranceTracker["Nick", "Ally"])
println(toleranceTracker["Ally", "Rishi"])
println(toleranceTracker["Rishi", "Ally"])
```

Output:

```
CAN_TOLERATE
UNKNOWN
CANNOT_TOLERATE
UNKNOWN
```

Various binary operators can be overloaded

See Kotlin documentation

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)`

Overloading comparison operators (which are also binary operators)

See Kotlin documentation

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

Various unary operators can be overloaded

See Kotlin documentation

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

Increments, decrements and augmented assignments

See Kotlin documentation

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

See Kotlin Documentation

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)`

Infix functions

`infix` specifies that this function's name can be placed between its arguments

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

Now these all mean the same thing (if `p1` and `p2` are `Point`s):

```
p1 + p2      ←——— Operator form (enabled by operator)
p1 plus p2   ←——— Infix form (enabled by infix)
p1.plus(p2)  ←——— Regular form
```

In all cases, `p1` is the **receiver** of the method call

Extension methods: motivation

Suppose we often need to count occurrences of a character in a string:

```
if (someString.count { it == 'a'} >
    someOtherString.count { it == 'a' }) {
    ...
}
```

This is more readable:

```
if (someString.count('a') >
    someOtherString.count ('a')) {
    ...
}
```

Problem: `String` does not provide `count` method that takes a `Char`

We can provide this overload of `count` as an **extension method**

Introduce this declaration at file-level scope (not in any class)

```
fun String.count(c: Char): Int = this.count { it == c }
```

Indicates that the method is being added to `String` – the **receiving object** of a call to `count` will be a `String`

`this` refers to the string that receives the call

When there is no ambiguity, `this` can be omitted :

```
fun String.count(c: Char): Int = count { it == c }
```

Exercise: write some extension methods

- Equip `Int` with an `isPowerOfTwo()` method – returns true if and only if the receiving integer is a power of two
- Equip `String` with an `isPalindrome()` method – returns true if and only if the receiving string is a palindrome
- `isPalindrome()` should be case-sensitive by default, but it should also be possible to provide a boolean argument controlling case sensitivity
- Equip `Double` (64-bit floating point) with a `sameAsFloat()` method – returns true if and only if the Double's value can be represented as a `Float` (32-bit floating point) with no change in value due to rounding

In Python you can do...

```
print("Hello" * 3)
HelloHelloHello
```

Does not work in Kotlin:

```
println("Hello" * 3)
```

Compile error: Unresolved reference.

Let's make this possible!

```
operator fun String.times(count: Int): String =
    repeat(count)
```

Indicates that we are **overloading** an **operator**

Indicates that the operator is `*`

This operator is being overloaded for `String` and `Int`

If we wish, we can use `this` to refer to the `String` receiver

```
operator fun String.times(count: Int): String =
    this.repeat(count)
```

In Python you can also do…

```
print(3 * "Hello")
HelloHelloHello
```

Does not work in Kotlin, even with our extension method:

```
println(3 * "Hello")
```

Compile error: Unresolved reference.

Our overload of `*` has a `String` receiver – here we have an `Int` receiver

Solution: also extend `Int` with overloaded +

Operator is being overloaded for `Int` and `String`

```
operator fun Int.times(toBeRepeated: String): String
        = toBeRepeated.repeat(this)
```

Do not be afraid of descriptive names!
Make life easy for your readers

We **must** use `this` to refer to the receiving `Int` – we don't have any other name for it!

Another formulation – what are the pros and cons?

```
operator fun Int.times(toBeRepeated: String): String
        = toBeRepeated * this
```

Exercise: adding `Int` and `Point`

Recall from earlier that `println(10 * Point(1, 2))` was not supported

Write a suitable extension method to allow this syntax

Extension methods on generic classes

Kotlin provides a generic `Pair<A, B>` class – represents pairs of objects of any two given types

How do we write an extension method to check whether the components of a pair are equal?

```
fun Pair.equalComponents() = first == second
```

Wrong: compiler says “2 type arguments expected for class Pair<A, B>”

Extension methods on generic classes

Second attempt:

```
fun Pair<A, B>.equalComponents() = first == second
```

Wrong: compiler says “Unresolved reference: A” and “Unresolved reference: B”

Type parameters `A` and `B` must be introduced:

```
fun <A, B> Pair<A, B>.equalComponents() =
    first == second
```

Means: “The function will be defined in terms of two arbitrary types A and B”

A and B now refer to these types

Exercise: why does `equalComponents()` work?

- It could make sense to ask whether the components of a `Pair<Int, Int>` are equal, or those of a `Pair<String, String>`, etc.
- It probably does not make sense to ask whether the components of e.g. a `Pair<Int, String>` are equal – they won't be!
- However, we **can** ask whether the components of any pair are equal – why is this possible?

Exercise: extend `Pair` with a `swap` method

- Write an extension method `swap()`. When invoked on a pair whose components have the same type, it should return a new pair with these components but in reverse order.

Extending a generic class for specific types

Does not make sense to overload `+` on `Pair<A, B>`

In general: no way to add components

But we **can** provide this extension for pairs of `Doubles`

```
operator fun Pair<Double, Double>.plus(
    other: Pair<Double, Double>,
): Pair<Double, Double> =
    Pair(first + other.first, second + other.second)
```

Provides extension specifically for `Pair<Double, Double>`

Now we can write: `println(Pair(1.2, 3.4) + Pair(5.6, 7.8))`

Output: `(6.8, 11.2)`

Exercise: extension to `List<Boolean>`

Extend `List<Boolean>` with the following methods

- `allTrue()`: returns true if and only if every list element is **true**
- `allFalse()`: returns true if and only if every list element is **false**
- `someTrue()`: returns true if and only if some list element is **true**
- `someFalse()`: returns true if and only if some list element is **false**

Avoid duplicate code as much as you can when implementing these methods

Exercise: Providing `and` and `or` methods on `Boolean`

- In Kotlin you can write $\mathbf{e_1}$ `and` $\mathbf{e_2}$ to compute the logical and of two expressions
- Similarly, you can write $\mathbf{e_1}$ `or` $\mathbf{e_2}$ for logical or
- If these were not already available, how would you provide them as extension methods?
- Do your proposed `and` and `or` methods behave identically to `&&` and `||`?
- What about the `and` and `or` methods provided by Kotlin – are they functionally the same as `&&` and `||`?

An extension method inside a class

Let’s make a redacting string builder – checks each string that is passed to append against a list of bad words

Example usage:

```
val builder = RedactingStringBuilder(setOf("Haskell", "monad", "category", "functor"))
builder.append("My")
builder.append(" ")
builder.append("favourite")
builder.append(" ")
builder.append("programming")
builder.append(" ")
builder.append("language")
builder.append(" ")
builder.append("is")
builder.append(" ")
builder.append("Haskell")
builder.append(" ")
println(builder.toString())
```

Output:

```
My favourite programming language is _______
```

An extension method inside a class

```
class RedactingStringBuilder(private val badWords: Set<String>) {
    private val stringBuilder = StringBuilder()

    val length: Int = stringBuilder.length

    fun append(text: String) = stringBuilder.append(text.redact())

    override fun toString(): String {
        return stringBuilder.toString()
    }

    fun String.redact(): String =
        if (this in badWords) {
            "_".repeat(length)
        } else {
            this
        }
}
```

`text` has type `String`, and `String` does not have a `redact` method

We provide `redact` as an extension

This extension can only be called by a function that has `RedactingStringBuilder` as its receiver

An extension method inside a class

```
class RedactingStringBuilder(private val badWords: Set<String>) {
    private val stringBuilder = StringBuilder()

    val length: Int = stringBuilder.length

    ...

    fun String.redact(): String =
        if (this in badWords) {
            "_".repeat(length)
        } else {
            this
        }
}
```

Which `length` does this refer to?

- `length` property of `RedactingStringBuilder`?
- `length` property `String`?

Answer: `length` of `String`, because the receiver of this method is a `String`

The `length` property of `RedactingStringBuilder` is **shadowed**

An extension method inside a class

To refer to enclosing class's version of a shadowed property, use `this@EnclosingClassName`

```
class RedactingStringBuilder(private val badWords: Set<String>) {
    private val stringBuilder = StringBuilder()

    val length: Int = stringBuilder.length

    ...

    fun String.redact(): String {
        println(this@RedactingStringBuilder.length)
        if (this in badWords) {
            return "_".repeat(length)
        }
        return this
    }
}
```

Provides access to `length` of the enclosing `RedactingStringBuilder`

Provides access to `length` of the receiving `String` – equivalent to `this.length`

Can we invoke such an extension method from outside the class?

```
fun main() {
    val builder = RedactingStringBuilder(setOf(
        "Haskell",
        "monad",
        "category",
        "functor"),
    )
    How can we ask for a redacted version of a string right here?
}
```

Can we invoke such an extension method from outside the class?

```
fun main() {
    val builder = RedactingStringBuilder(setOf(
        "Haskell",
        "monad",
        "category",
        "functor"),
    )
    println("Haskell".redact())
}
```

Compile error: Unresolved reference: redact

Makes sense: redact is not declared in the current scope

Can we invoke such an extension method from outside the class?

```
fun main() {
    val builder = RedactingStringBuilder(setOf(
        "Haskell",
        "monad",
        "category",
        "functor"),
    )
    with (builder) {
        println("Haskell".redact())
    }
}
```

The *with* statement makes the target object `this` for the scope that follows

All methods of `RedactingStringBuilder` are in scope here, including its *redact* extension to `String`

Extension methods: there's no magic!

Writing this:

```
fun String.count(c: Char): Int = this.count { it == c }
 ...
println("Hello".count('l'))
```

Is equivalent to writing this:

```
fun count(target: String, c: Char): Int = target.count { it == c }
 ...
println(count("Hello", 'l'))
```

Extension methods can help with look and feel, but are really just **syntactic sugar**

Extension method do not get access to private properties and methods

```
class Noodle(private val oodle: Int)

fun Noodle.showOodle() = println(oodle)
```

Compile error: Cannot access 'oodle': it is private in 'Noodle'

An extension method only has access to the **service** provided by the class it extends – the public properties and methods

An extension method is a **client** of the class it extends

An extension method adds **convenience services** to the class for **other clients** to use