Overloading Operators

We'll cover the following

Need for operator overloading

Traditionally we use operators on numeric types to create expressions, for example, $2 + 3$ or $4.2 * 7.1$. Operator overloading is a feature where the language extends the capability to use operators on user-defined data types.

Compare the two lines in the following code snippet:

```
bigInteger1.multiply(bigInteger2)

bigInteger1 * bigInteger2
```

The first line uses the `multiply` method from the JDK. The second line uses the `*` operator on the instances of `BigInteger`; that operator comes from the Kotlin standard library. The second line takes less effort—both to write and to read—than the first line, even though they both accomplish the same thing. The operator `*` makes the code look more natural, fluent, less code-like than the use of the `multiply()` method. Java doesn't permit us to overload operators on user-defined datatypes, but Kotlin does. That leads to greater fluency, less clutter, and makes it a joy to program in Kotlin.

Benefits of operator overloading

In addition to using operators like `+`, `-`, and `*` on numeric types, languages that support operator overloading also permit using such operators on objects. For support operator overloading also permit using such operators on objects. For

instance, given a reference `today` of a `Date class`, `today + 2` may mean that's two days from now.

> **Don't Overuse Operator Overloading**
>
> Programmers have a love-hate relationship with operator overloading because it's a feature that can be abused easily. For example, does today + 2 mean two days or two months from now? It's not clear, and that's frustrating to the person reading the code. Don't use operator overloading unless it'll be obvious to the reader.

When used correctly, the benefits of operator overloading are amazing. For example, `wishList + appleWatch` is fairly intuitive; we're appending an item to a `List`. When overloading an operator on a user-defined class, we have to be careful to honor the intended behavior of the operator. For example, we know that `+` is a pure function; it doesn't modify either of the operands. The writer of `+` on the class `List` shouldn't modify either of the operands provided to the `+` operator. That's part of the "used correctly" effort. The names used for the variables also contribute to the readability of code with operators, so pick names judiciously. For example, `wishList + appleWatch`, rather than `w + x`, is better to convey that we're using overloaded operator `+` on an instance of `List`.

Since the JVM doesn't support operator overloading (and Kotlin compiles down to Java bytecode), Kotlin makes operator overloading possible by mapping operators to specially named methods. For example, `+` will result in a call to a `plus()` method. By writing these specialized methods, you may overload operators on your own classes, although the precedence of operators is fixed and you can't change the precedence. Also, using the extension functions that we'll see soon, you may overload operators on third-party classes.

How is it done?

To overload an operator, define a function and mark it with the operator keyword. Here's an example of overloading `+` on a `Pair<Int, Int>` to add a pair of numbers:

```
// pairplus.kts
```

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

The function is named `plus()` which is the specialized method name for `+`. It operates on an implicit object, the left operand, referenced using `this` and on the right operand `other`. Instead of writing `this.first + other.first`, we used a short form `first + other.first` to compute the sum of the first values in the two pairs. The `plus()` extension function returns a new `Pair<Int, Int>`, where the first value is sum of the first values in each of the given pairs and second is the sum of the second values in each of the given pairs.

To overload operators for your own classes, write the appropriate specialized methods as member functions within your classes. For example, here's how you'd overload `*`, on a class that represent a complex number, to multiply two complex numbers:

```
import kotlin.math.abs

data class Complex(val real: Int, val imaginary: Int) {
  operator fun times(other: Complex) =
    Complex(real * other.real - imaginary * other.imaginary,
        real * other.imaginary + imaginary * other.real)

  private fun sign() = if (imaginary < 0) "-" else "+"

  override fun toString() = "$real ${sign()} ${abs(imaginary)}i"
}

println(Complex(4, 2) * Complex(-3, 4)) //-20 + 10i
println(Complex(1, 2) * Complex(-3, 4)) //-11 - 2i
```

complex.kts

How could one not love a language where `fun times` is a valid syntax—the `times()` method marked as operator stands in for the `*` operator. If you don't mark it as operator, but try to use `*`, then you'll get an error. Also, if you mark a method with a non-specialized name with operator, you'll get an error.

Mapping operators

The mapping from operator to the corresponding special name for methods, shown in the following table, is fairly easy to remember and it's often intuitive. All operations are pure—they don't mutate or cause side effects, unless otherwise noted.

Operator	Corresponds To	Observations
+x	x.unaryPlus()	
-x	x.unaryMinus()	
!x	x.not()	
x + y	x.plus(y)	
x - y	x.minus(y)	
x * y	x.times(y)	
x / y	x.div(y)	
x % y	x.rem(y)	
++x	x.inc()	x must be assignable
x++	x.inc()	x must be assignable
–x	x.dec()	x must be assignable
x–	x.dec()	x must be assignable
x == y	x.equals(y)	
x != y	!(x.equals(y))	
x < y	x.compareTo(y)	Also used for <=, >, >=
x[i]	x.get(i)	
x[i] = y	x.set(i, y)	

y in x	x.contains(y)	Also used for !in
x..y	x.rangeTo(y)	
x()	x.invoke()	
x(y)	x.invoke(y)	

The functions for the composite operators `+=`, `-=`, `*=`, `/=`, and `%=` take on the word *Assign* after the special name for the first operator, for example, `plusAssign()` for `+=`. You shouldn't implement both `plus()` and `plusAssign()` for the same class. Likewise, for other composite operators. If you implement `plus()`, for example, then `+=` will use that method appropriately. If not, to resolve `+=`, the compiler will look for `plusAssign()`. If neither `plus()` nor `plusAssign()` are found, then the compilation of `+=` for instances of your class will fail. While `plus()` is a pure function and returns a new instance, `plusAssign()` will change the state of the instance on which it operates and thus expects the object to be mutable.

Rules to follow!

You have to abide by some rules when overloading operators. Honor the conventional wisdom of the behavior associated with an operator. For instance, don't mutate an object in the `+` or `-` operator-overloaded functions. This rule extends even to operators that are normally perceived as mutating. For example, let's overload the increment and decrement operators. The function `inc()` is used both for pre-increment `++x` and post-increment `x++`. Likewise, `dec()` is used for both pre-decrement and post-decrement.

```
class Counter(val value: Int) {
  operator fun inc() = Counter(value + 1)

  operator fun dec() = Counter(value - 1)

  override fun toString() = "$value"
}

var counter = Counter(2)
println(counter)    //2
println(++counter)  //3
println(counter)    //3
println(counter++)  //3
println(counter)    //4
```

counter.kts

Within the `inc()` method we're not mutating anything. Instead, we return a new object with new state. When used as pre-increment, Kotlin will save the returned value into the variable on which the operator is applied. That's the reason why we see `3` as a result of `++counter`. On the other hand, when used as a post-increment operator, Kotlin will save the result into the said variable but return the previous value as a result of the expression. That explains why the result of `counter++` is `3` instead of the more recent value `4`, which is the value held in the object that `counter` now refers to.

Operator overloading is a powerful feature, but follow a few recommendations when using it:

- Use sparingly.
- Overload only when the use will be obvious to the readers.
- Abide by the commonly understood behavior of the operators.
- Use meaningful names for variables so it's easier to follow the context of overloading.

Operators reduce noise and, when used correctly, can make the code intuitive and readable.

In the next lesson, we'll see how we can take readability further, beyond operators.

QUIZ

1 What is the correct syntax to overload an operator with two operands?

2

The operator `[]` corresponds to which specialized method name?

Retake Quiz

In the next lesson, we'll see how we can take readability further, beyond operators.