Learn about operators and types

In this task, you learn about operators and types in the Kotlin programming language.

Step 1: Explore numeric operators

1. Open IntelliJ IDEA, if it's not already open.
2. To open the Kotlin REPL, select **Tools** | **Kotlin** | **Kotlin REPL**.

As with other languages, Kotlin uses `+`, `-`, `*` and `/` for plus, minus, times and division. Kotlin also supports different number types, such as `Int`, `Long`, `Double`, and `Float`.

1. Enter the following expressions in the REPL. To see the result, press `Control+Enter` (`Command+Enter` on a Mac) after each one.

Note: In this codelab, ⇒ indicates output from your code. In the latest version of the REPL, the output includes the result number and the type of the result.

```
1+1
⇒ res8: kotlin.Int = 2

53-3
⇒ res9: kotlin.Int = 50
```

```
50/10
⇒ res10: kotlin.Int = 5

1.0/2.0
⇒ res11: kotlin.Double = 0.5

2.0*3.5
⇒ res12: kotlin.Double = 7.0
```

Note that results of operations keep the types of the operands, so $1/2 = 0$, but $1.0/2.0 = 0.5$.

1. Try some expressions with different combinations of integer and decimal numbers.

```
6*50
⇒ res13: kotlin.Int = 300

6.0*50.0
⇒ res14: kotlin.Double = 300.0

6.0*50
⇒ res15: kotlin.Double = 300.0
```

1. Call some methods on numbers. Kotlin keeps numbers as primitives, but it lets you call methods on numbers as if they were objects.

```
2.times(3)
⇒ res5: kotlin.Int = 6

3.5.plus(4)
⇒ res8: kotlin.Double = 7.5

2.4.div(2)
⇒ res9: kotlin.Double = 1.2
```

Note: It is possible to create actual object wrappers around numbers, which is known as *boxing*. Boxing happens automatically, such as for collections, where numbers are boxed and unboxed as needed.

Warning: Using object wrappers takes more memory than storing just a number primitive. Do not use boxing unless it is needed, such as in a collection, which is covered later.

Step 2: Practice using types

Kotlin does not implicitly convert between number types, so you can't assign a short value directly to a long variable, or a `Byte` to an `Int`. This is because implicit number conversion is a common source of errors in programs. You can always assign values of different types by casting.

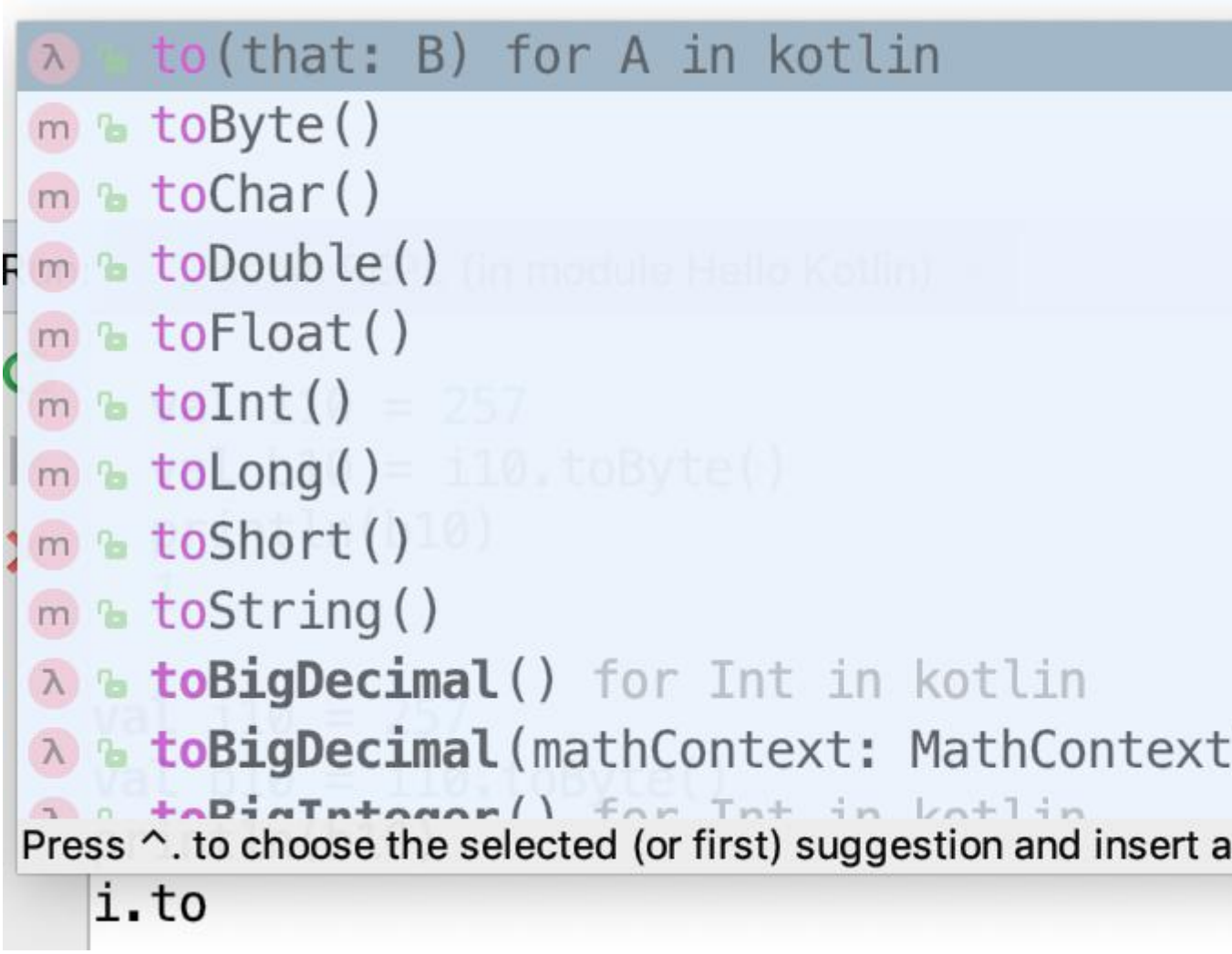
1. To see some of the casts that are possible, define a variable of type `Int` in the REPL.

```
val i: Int = 6
```

1. Create a new variable, then enter the variable name shown above, followed by `.to`.

```
val b1 = i.to
```

IntelliJ IDEA displays a list of possible completions. This auto-completion works for variables and objects of any type.



1. Select `toByte()` from the list, then print the variable.

```
val b1 = i.toByte()
println(b1)
⇒ 6
```

1. Assign a `Byte` value to variables of different types.

```
val b2: Byte = 1 // OK, literals are checked statically
println(b2)⇒ 1

val i1: Int = b2⇒ error: type mismatch: inferred type is Byte
but Int was expected

val i2: String = b2⇒ error: type mismatch: inferred type is
Byte but String was expected
```

```
val i3: Double = b2⇒ error: type mismatch: inferred type is
Byte but Double was expected
```

1. For the assignments that returned errors, try casting them instead.

```
val i4: Int = b2.toInt() // OK!
println(i4)⇒ 1

val i5: String = b2.toString()
println(i5)⇒ 1

val i6: Double = b2.toDouble()
println(i6)⇒ 1.0
```

1. To make long numeric constants more readable, Kotlin allows you to place underscores in the numbers, where it makes sense to you. Try entering different numeric constants.

```
val oneMillion = 1_000_000
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

Note: Because Kotlin is strongly typed, the compiler can usually infer the type for variables, so you don't need to explicitly declare it.

Step 3: Learn the value of variable types

Kotlin supports two types of variables: changeable and unchangeable. With `val`, you can assign a value once. If you try to assign something again, you get an error. With `var`, you can assign a value, then change the value later in the program.

1. Define variables using `val` and `var` and then assign new values to them.

```
var fish = 1
fish = 2
val aquarium = 1
aquarium = 2
⇒ error: val cannot be reassigned
```

You can assign `fish` a value, then assign it a new value, because it is defined with `var`. Trying to assign a new value to `aquarium` gives an error because it is defined with `val`.

The type you store in a variable is inferred when the compiler can figure it out from context. If you want, you can always specify the type of a variable explicitly, using the colon notation.

1. Define some variables and specify the type explicitly.

```
var fish: Int = 12var lakes: Double = 2.5
```

Once a type has been assigned by you or the compiler, you can't change the type, or you get an error.

Step 4: Learn about strings

Strings in Kotlin work pretty much like strings in any other programming language using `"` for strings and `'` for single characters, and you can concatenate strings with the `+` operator. You can create string templates by combining them with values; the `$variable` name is replaced with the text representing the value. This is called variable interpolation.

1. Create a string template.

```
val numberOfFish = 5
val numberOfPlants = 12"I have $numberOfFish fish" + "
and $numberOfPlants plants"
⇒ res20: kotlin.String = I have 5 fish and 12 plants
```

1. Create a string template with an expression in it. As in other languages, the value can be the result of an expression. Use curly braces `{}` to define the expression.

```
"I have ${numberOfFish + numberOfPlants} fish and
plants"
⇒ res21: kotlin.String = I have 17 fish and plants
```