Function Types (Lambda storage in variable)

→ In Kotlin you can store lambda in a variable.

e.g:

```
val sum = { x:Int, y:Int → x+y }
```

→ But which type the variable have in the above case?

```
val sum: (Int, Int) → Int = { x, y → x+y }
```

→ So if we specify this type explicitly we see function type. In this case it takes two integer parameters and returns integer as a result.

```
val sum: (Int, Int) → Int = { x, y → x+y }
```

- (Int, Int) → Taking 2 integer parameters here
- Int = Resulting value.
- { x, y → x+y } : integer parameters; x+y : result type integer

Function Types (Storing Lambda in a Variable)

→ A lambda expression can be saved in a variable and can be reused by passing it to functions with collections in functional style like 'any()' or 'filter()'.

```
val isEven = { i = Int -> i % 2 == 0 }
val List = ListOf (1, 2, 3, 4)
List.any (isEven)    // true
```

↳ Above variable is plugged in.

```
List.filter (isEven) // [2,4].
```

→ Or you can call the stored variable.

e.g:

```
isEven(42)  // true.
```

→ isEven is being called as a regular function.
→ You call it just as a regular function.

Function Types (Calling Lambda).

→ You can call a lambda expression directly. e.g.:

```
{ println("hey!") }() // possible but
                      // looks strange
                      // Hard to read.
```

Instead you can use the keyword, "run"

```
run { println("hey!") } // using run instead.
                        // Readable code.
```

↳ using run instead.

Function Types : (Under the Hood).

concise syntactic form	
() → Boolean	Function0<Boolean>.
(Order) → Int	Function1<Order, Int>
(Int, Int) → Int	Function2<Int, Int, Int>.

→ Under the hood, these functions Types just correspond to regular interfaces, like 'Function0', 'Function1' and 'function2' with corresponding generic arguments.

→ You can find the declaration of these function interfaces in the library.

→ Whenever you use concise syntactic form, often under the hood it just uses the corresponding interfaces

SAM INTERFACES IN JAVA:

→ SAM → Single Abstract Method.

e.g: (Java)

```
public interface Runnable {
        public abstract void run();
}

void postponeComputation(int delay,
```

→ You can pass lambda as an argument to this Java method. (Runnable computation)

parameter interface defined above

Mixing Kotlin and Java:

e.g of SAM interface taken as a parameter

→ In Kotlin we have convinient function interfaces (covered in previous lesson) and nice syntax for function types.

→ However when you fix Java and Kotlin, In Java you can pass a lambda instead of a lot of interfaces.

↳ In Kotlin it works the same.

→ Whenever you call a Java Method which takes SAM interface as a parameter, you can pass lambda as an argument to this Java method.

e.g: (Kotlin)

```
postponeComputation(1000) { println(42) }
```

→ In Kotlin, creating an instance of the same of such interface, you may use the autogenerated SAM constructor, like in this 'Runnable' and pass lambda as its argument.

e.g: Kotlin.

```
val runnable = Runnable { println(42) }
```

→ You can store the lambda in a variable and pass to a method or use it for some other case.

e.g: (Kotlin).

```
val runnable = Runnable { println(42) }
postponeComputation(1000, runnable)
```

Function Types and nullability:

`() -> Int?` vs `(() -> Int)?`

`() -> Int?` → return type is nullable.

→ It means that you are making the return type of this function nullable. and not the whole type itself.

`(() -> Int)?` → The variable is nullable

→ If you use the parantheses like above, the whole type will become Nullable.

e.g:

#1 `val f1: () -> Int? = null`
- `() -> Int?` : function
- `Int?` → 1.1 Return type of this function is nullable.
- 1.2 However f1 is not null type.

#2 `val f2: () -> Int? = { null }`
- `() -> Int?` : function
- whole type becomes null.

#3 `val f3: (() -> Int)? = null`
- `(() -> Int)?` : function
- whole function type is null

#4 `val f4: (() -> Int)? = { null }`
- `(() -> Int)?` : function

#1, #4 will not compile.

Lambda expression expects an integer value or a null. We are passing only null.

Q. How to call a function when a function type is null?

```
val f:(() -> Int)? = null.

f()    // wont compile.
```

→ One way to call it is the following,

```
if (f != null) {
    f()    // Concise syntax.
}
```

→ Better way to call it is,

```
f?.invoke()
// calling
// safe success expression when
// you have a variable of nullable function type.
```

is to use a '?' question mark followed by a method called invoke.

→ All these interfaces have an internal function called invoke inside of them which can be called explicitly, which is actually called under the hood even if you use concise syntax.