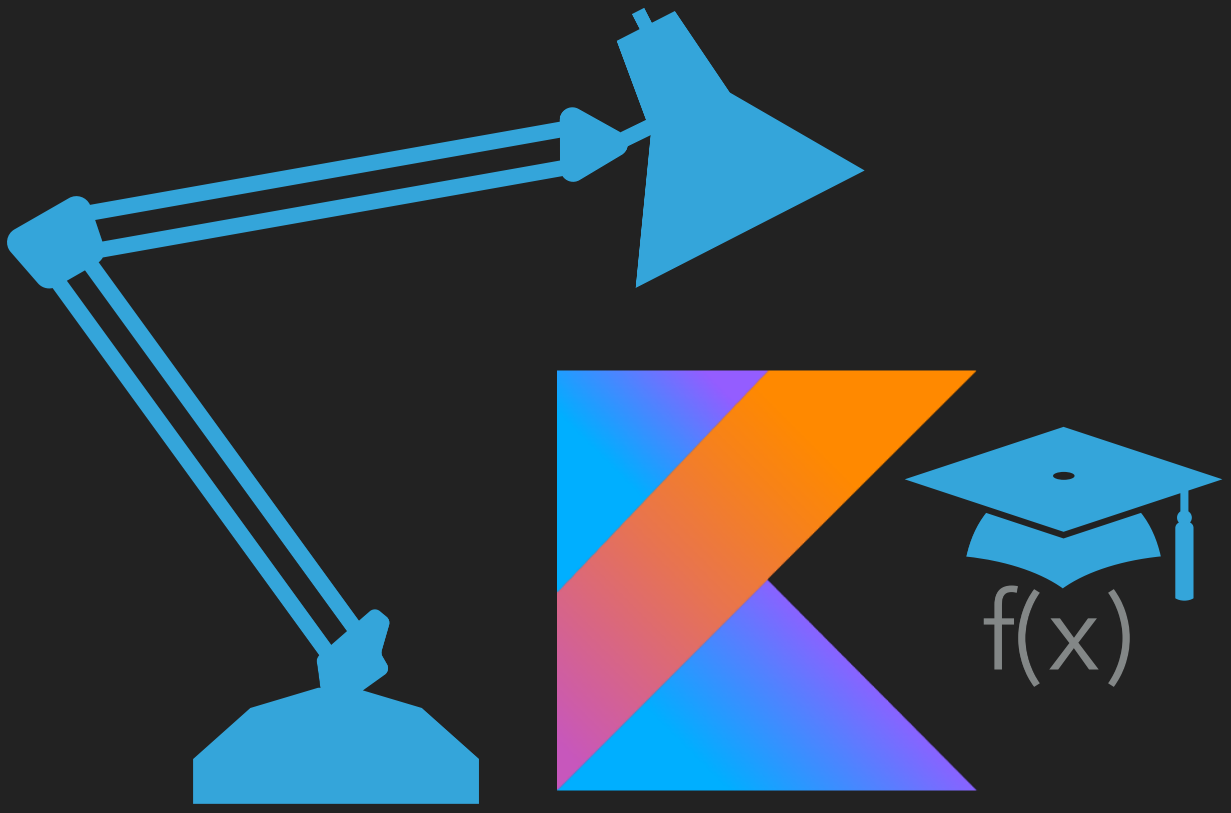
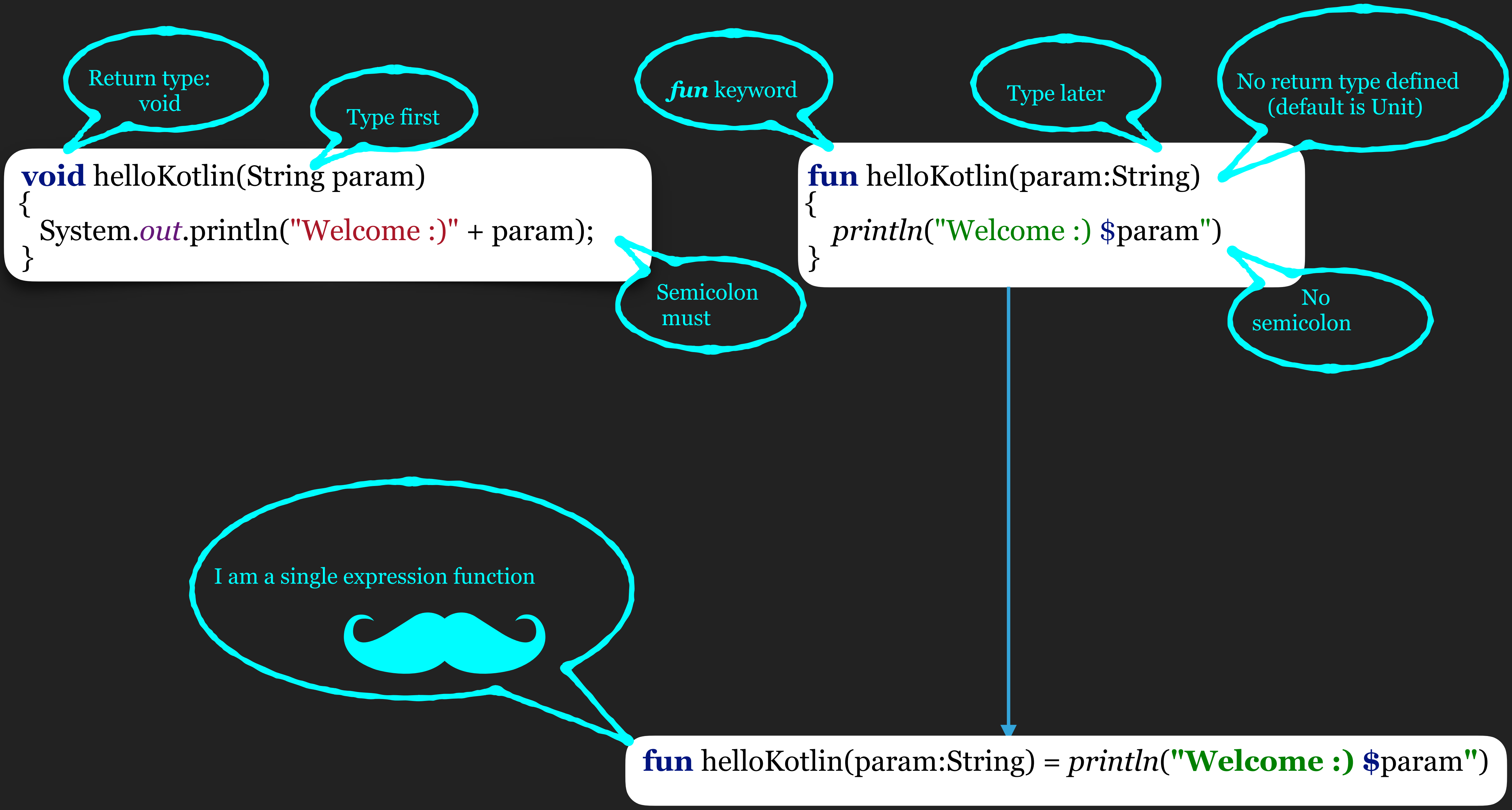


EXPLORING KOTLIN FUNCTIONS

Let’s have some **fun()**

ARQ
group

A QUICK RECAP



Return type: void

Type first

```
void helloKotlin(String param)
{
  System.out.println("Welcome :)" + param);
}
```

Semicolon must

fun keyword

Type later

No return type defined (default is Unit)

```
fun helloKotlin(param:String)
{
  println("Welcome :) $param")
}
```

No semicolon

I am a single expression function

```
fun helloKotlin(param:String) = println("Welcome :) $param")
```

WHY WE 💖 KOTLIN FUNCTIONS

FLEXIBILITY

Parameters

Return types

VARIATIONS

infix

Extension functions

Higher-order functions

inline

SCOPING OPTIONS

Top Level Functions

Member Functions

Local Functions

Companion Objects

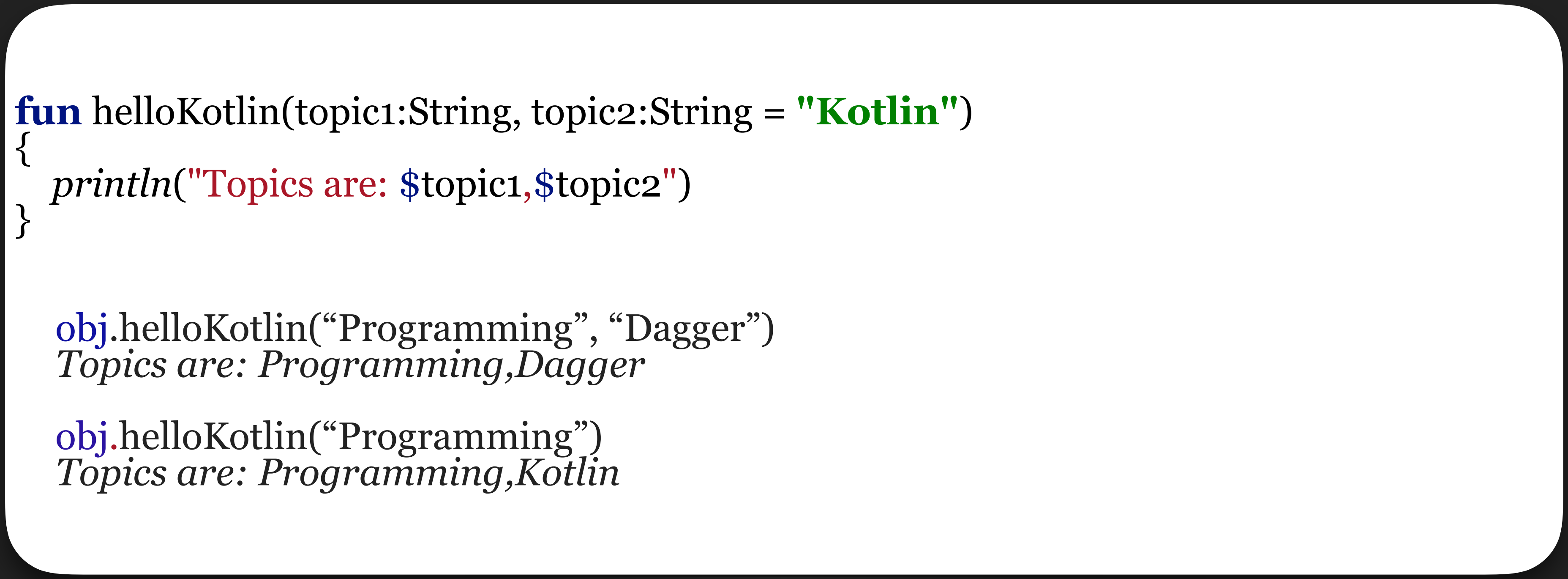
Anonymous functions

PARAMETERS

```
void helloKotlin(String topic1, String topic2)
{
   System.out.println(“Topics are: ” + topic1 + “,“ + topic );
}

   obj.helloKotlin(“Programming”, “Kotlin”);

   Topics are: Programming,Kotlin
```

- We do not have any functionality of providing default values.
- Overloading methods by repeating them again is only solution.
- Any new person working on code cannot figure out the best default value to start with.

DEFAULT PARAMETERS



```
fun helloKotlin(topic1:String, topic2:String = "Kotlin")
{
  println("Topics are: $topic1,$topic2")
}


   obj.helloKotlin("Programming", "Dagger")
   Topics are: Programming,Dagger

   obj.helloKotlin("Programming")
   Topics are: Programming,Kotlin
```

- We get to know default values at once.
- Overloading is also achieved without repeating similar code again.
- Less code, less bugs, better maintenance.

NAMED ARGUMENTS

```
fun helloKotlin(t1:String, t2:String="Kotlin", t1Subtitle:String = "", t2Subtitle:String="Functions")
{
  println("Topics are: $t1($t1Subtitle),$t2($t2Subtitle)")
}
```

```
obj.helloKotlin("Infix notation", "Dagger","", "DaggerSubtitle")
```

```
Topics are: Infix notation(),Dagger(DaggerSubtitle)
```

```
obj.helloKotlin(t1 = "Infix notation", t1Subtitle="Examples" )
```

```
Topics are: Infix notation(Examples),Kotlin(Functions)
```

- Improves code readability.
- Helps in functions with lot of parameters.
- NOTE: Once an argument name is specified, all subsequent parameters should be named too.

RETURN TYPES

- We can omit return type if we are not returning anything useful.
- Kotlin can infer return types e.g

Inferred return type is Int

```
fun sum(a: Int, b: Int) = a + b
```

- It is a very common use case that we want to return 2 values from a method, which can be same data type or different.
- Shall we create a new class every time for that?

We can use Pair or Triple to combine multiple values.

```
return Pair(“Kotlin”, “MeetUp”)
```

WHY WE 💖 KOTLIN FUNCTIONS

FLEXIBILITY

Parameters

Return types

VARIATIONS

infix

Extension functions

Higher-order functions

inline

SCOPING OPTIONS

Top Level Functions

Member Functions

Local Functions

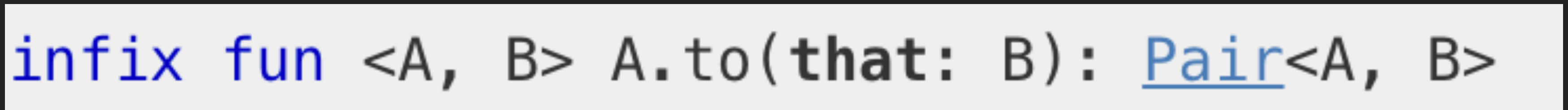
Companion Objects

Anonymous functions

ARQ group

INFIX

- infix keyword enables usage of infix notation.
- infix notation helps in using the functions without the use of dot and parenthesis.
- to() is a function that basically returns Pair<A,B> from two values.



```
infix fun <A, B> A.to(that: B): Pair<A, B>
```

```
"device" to "Android",
"language" to “Kotlin"
```

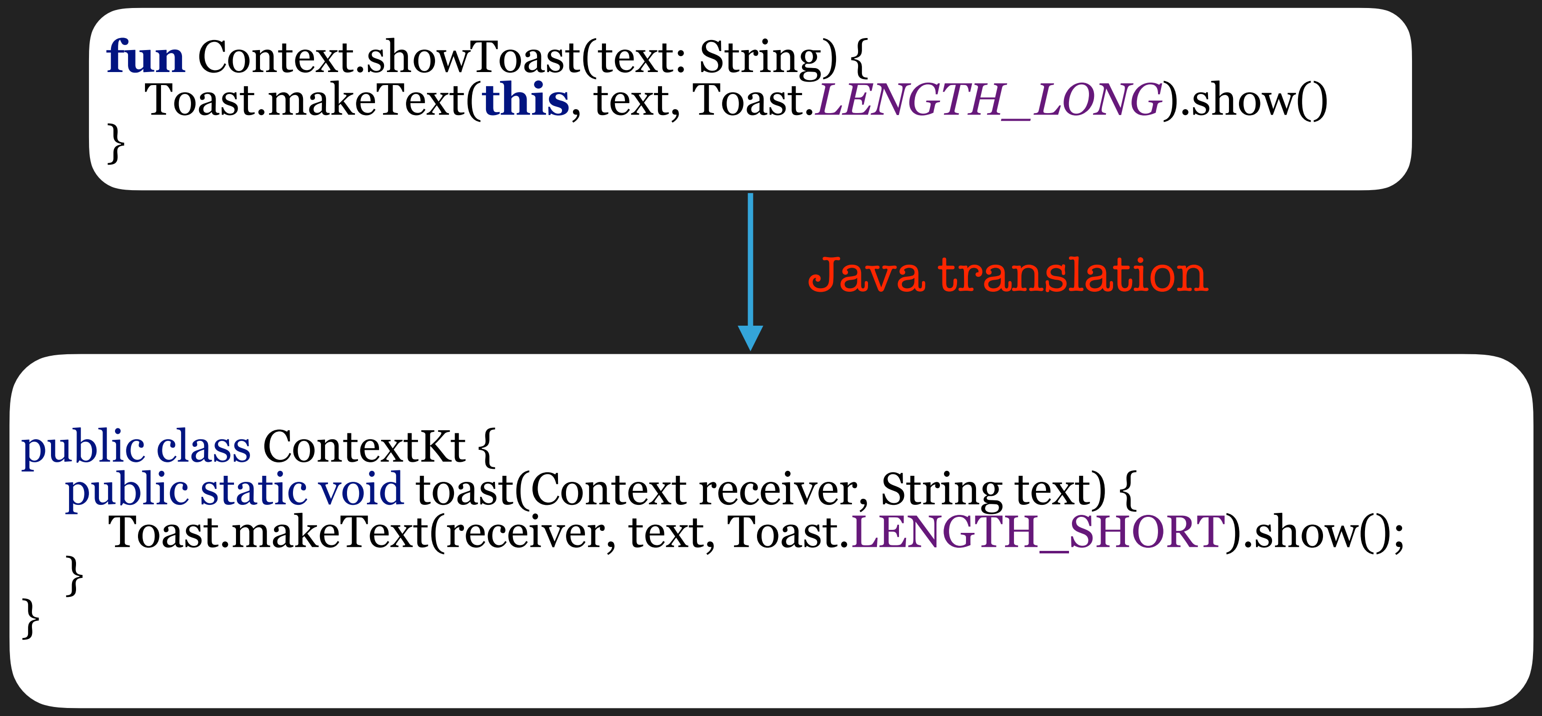
- Various Range operators like until, downTo, step are infix functions.
- e.g we can have “Hello” shouldBeEqualTo “Hello”

EXTENSION FUNCTIONS

- They provide an ability to extend the functionality of a class without actually inheriting it.

- They are used as if they are members of the class but they do not actually modify classes they extend.

- Extension function is compiled to a static function with a receiver object as its first argument.



```
fun Context.showToast(text: String) {
   Toast.makeText(this, text, Toast.LENGTH_LONG).show()
}
```

Java translation

```
public class ContextKt {
   public static void toast(Context receiver, String text) {
      Toast.makeText(receiver, text, Toast.LENGTH_SHORT).show();
   }
}
```

EXTENSION FUNCTIONS

- Extension functions are resolved statically - their calls are resolved at compile time by the type on which function is invoked.

```
 open class MeetUp

fun MeetUp.prepare() = "MeetUp"

class KotlinMeetUp : MeetUp()

fun KotlinMeetUp.prepare() = "KotlinMeetUp"

fun printValue(meetUp: MeetUp) = meetUp.prepare()

    printValue(KotlinMeetUp())
```

→ Output is “MeetUp”

- If we have an extension function same as member function, then member function is given priority over it.

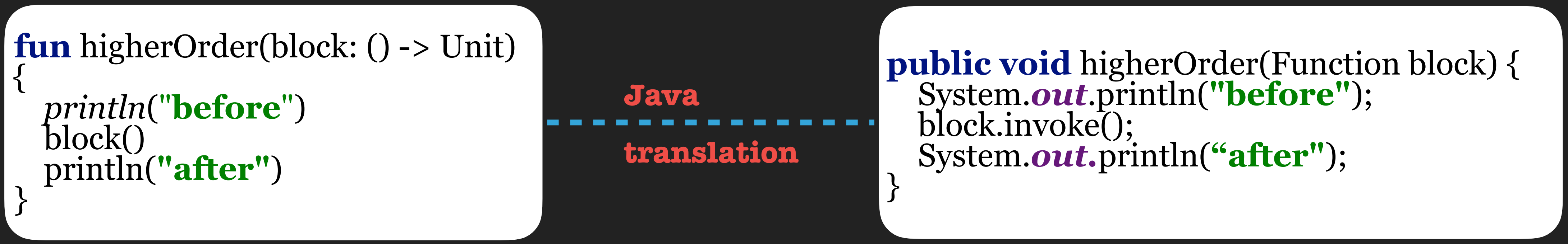
```
class TestExtension {
   fun test() {
       println("Member")
   }
}

fun TestExtension.test() {
   println("Extension")
}

    TestExtension().test()
```

→ Output is “Member”

HIGHER ORDER FUNCTIONS

- These are the functions that take functions as parameters or return functions as a result.

- Let's say we create a higher order function that takes a lambda of type ()->Unit (no parameters, no return value), and executes it like so:



```
fun higherOrder(block: () -> Unit)
{
   println("before")
   block()
   println("after")
}
```

Java translation

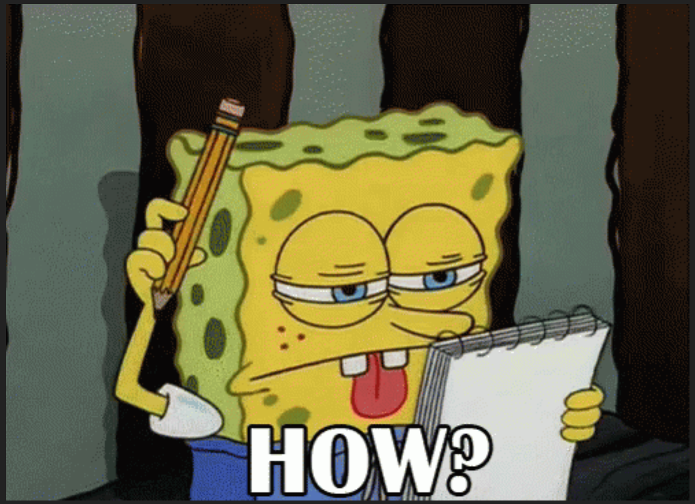
```
public void higherOrder(Function block) {
   System.out.println("before");
   block.invoke();
   System.out.println(“after");
}
```

INLINE

- Making a function inline is pretty simple, just add “inline” keyword.

```
inline fun someMethod():Int {
  return 2*a
}
```

- The code inside the inlined function will be copied to the call site.
- It helps in solving higher order functions performance hits.



INLINE BENEFITS

- Let’s take a look again at Higher order function example:

```
fun higherOrder(block: () -> Unit)
{
   println("before")
   block()
   println("after")
}
```

Java

translation

```
public void higherOrder(Function block) {
   System.out.println("before");
   block.invoke();
   System.out.println(“after");
}
```

Call in Kotlin

```
higherOrder {
   println("This is great!!")
}
```

```
Function lambda = new Function() {
  public Object invoke() {
    println("This is great!!")
  }

 higherOrder(lambda)
```

A new Function object is getting created

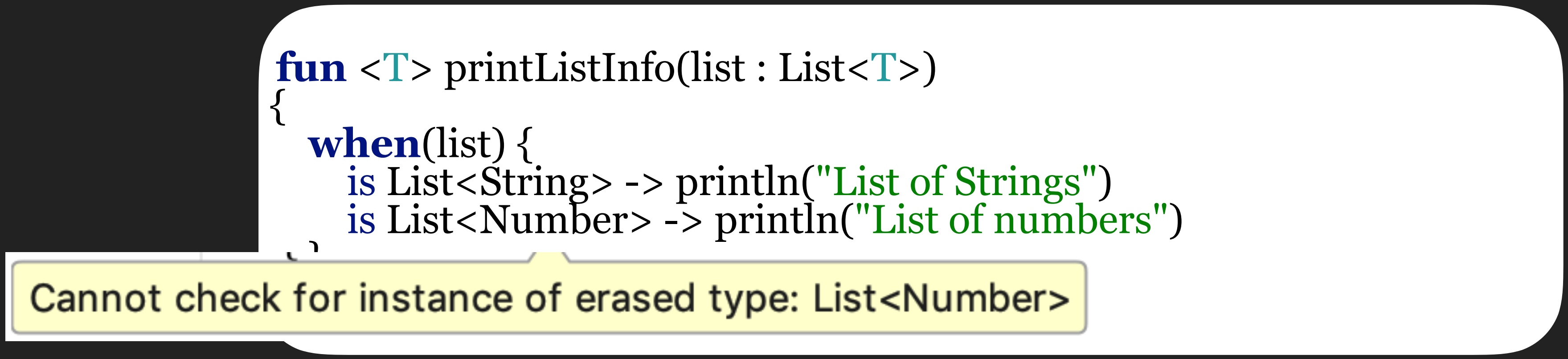
Add inline keyword

```
inline fun higherOrder(block: () -> Unit) {
   println("before")
   block()
   println("after")
}
```

```
public void higherOrder() {
   System.out.println("before");
   System.out.println(“This is great!!”);
   System.out.println("after");
}
```

TYPE ERASURE

- List<T> is a generic interface and we can pass on ArrayList<String> and hence T= String.



```
fun <T> printListInfo(list : List<T>)
{
   when(list) {
      is List<String> -> println("List of Strings")
      is List<Number> -> println("List of numbers")
   }
```

Cannot check for instance of erased type: List<Number>

- This information is not available at runtime, though at compile time all the type checking is done to make sure we do not mess up.

- The compiler throws away this understanding of T as String and this is known as “Type Erasure”.

- What if we want to keep this information at runtime?

REIFIED TYPE PARAMETER

- We can use the concept of **reify-ing**(To keep existence of type information and hence no type erasure happens).

- We qualified the type parameter with the reified modifier and make the function inline.

```
inline fun <reified T:Any> printListInfo(list : List<T>)
{
  if(T::class == String::class)
  {
    println(" This is a list of Strings")
  }
}
```

- Type reasure is not done and compiler modifies the generated bytecode to use the corresponding class directly.

- Therefore calls like myVar is T becomes myVar is String.

WHY WE 💖 KOTLIN FUNCTIONS

FLEXIBILITY

Parameters
Return types

VARIATIONS

infix
Extension functions
Higher-order functions
inline

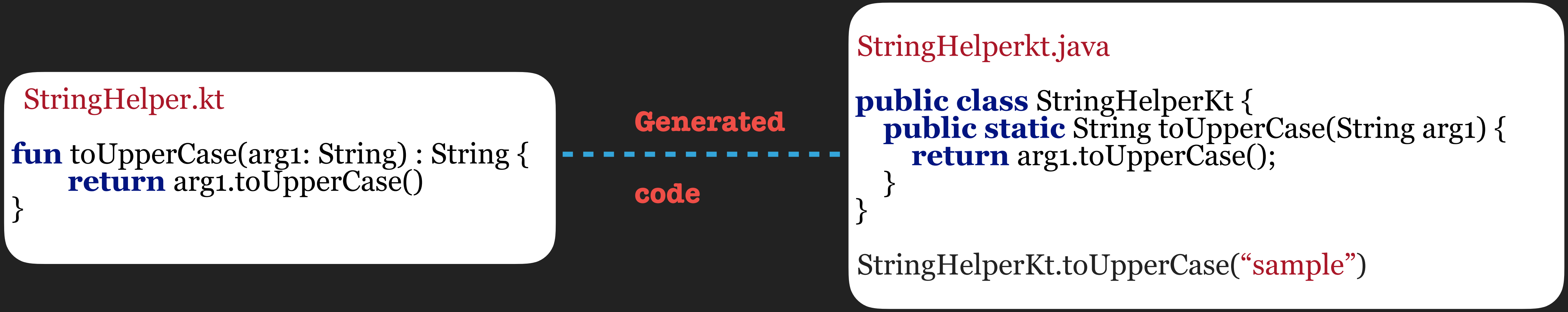
SCOPING OPTIONS

Top Level Functions
Member Functions
Local Functions
Companion Objects
Anonymous functions

BENEFITS OF SCOPING OPTIONS

TOP LEVEL FUNCTIONS

- Just write function inside a Kotlin file.
- Belong to their declared file’s package and have a global scope.
- Can be imported to use within other packages



StringHelper.kt

```
fun toUpperCase(arg1: String) : String {
      return arg1.toUpperCase()
}
```

Generated

code

StringHelperkt.java

```
public class StringHelperKt {
   public static String toUpperCase(String arg1) {
      return arg1.toUpperCase();
   }
}

StringHelperKt.toUpperCase(“sample”)
```

MEMBER FUNCTIONS

- Just like Java, Member functions are functions inside class.
- They have access to private members and methods inside class

LOCAL FUNCTIONS

- Local function is a function that is scoped inside another function
- It has access to all the parameters and variables of parent function.
- They are not accessible form outside the function they are declared in and they must be declared before they are referenced.

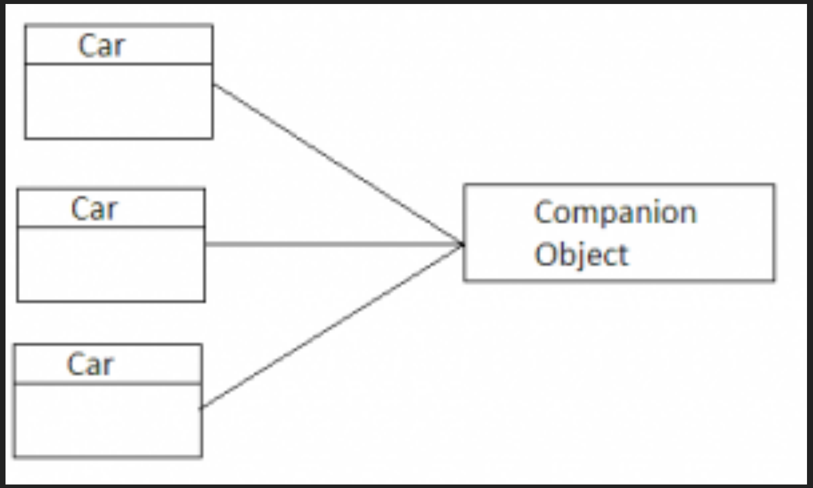
```
fun kotlinExample ()
{
    fun validateValue(str: String) {}

    validateValue("Kotlin")
    validateValue(“Functions")
}
```

- Grouping code into a smaller function will make it more clearer, we do not have access to local function which is inside.

COMPANION OBJECTS

- What should we do in case we need a factory method to get a new instance of class ?
  - ~~Create a static function and return instance of the class.~~ X

    There are no static methods in Kotlin.
  - Companion objects come to the rescue.
- A companion object is basically a Singleton object associated with class name.There can be only one companion object in a class.



- Members, such as methods and properties, defined inside a companion object may be accessed similarly to the way we access static fields and methods in Java.

COMPANION OBJECTS

- A companion object is initialised when the corresponding class is loaded (resolved), matching the semantics of a Java static initialiser.

- Companion objects can be named as well (the default name used is Companion).

- Examples:

```
class KotlinMeetUp {
   companion object {
      fun create(): KotlinMeetUp = KotlinMeetUp()
   }
}

  KotlinMeetUp.create()
  KotlinMeetUp.Companion.create()
```

```
class KotlinMeetUp {
   companion object Factory{
      fun create(): KotlinMeetUp = KotlinMeetUp()
   }
}

   KotlinMeetUp.create()
  KotlinMeetUp.Factory.create()
```

ANONYMOUS FUNCTIONS

- Anonymous functions are same as normal functions except that they do not have a name between fun keyword and parameters passed.

- Return types are inferred like normal functions, the parameter types can also be omitted if they can be inferred from context.e.g

```
var list = listOf(1,2,3)
list.filter(fun(item) = item > 0 )
```

- They are by default treated as objects which can be used and passed around in functions.

```
var a = fun(i: Int) = i * 2
 println(a(10))
```

There is an easy way to achieve similar behaviour and it is called Lambda expressions.

LAMBDAS

- The simplest way to define anonymous functions is Lambda expressions(we do not need even fun keyword).
- Notation : {arguments -> function body}
- By default, the value of last expression in a function body is returned from lambdas.

```
{ 1 }
```

A lambda expression that takes no arguments and returns 1. Its type is ()->Int.

```
{ s: String -> println(s) }
```

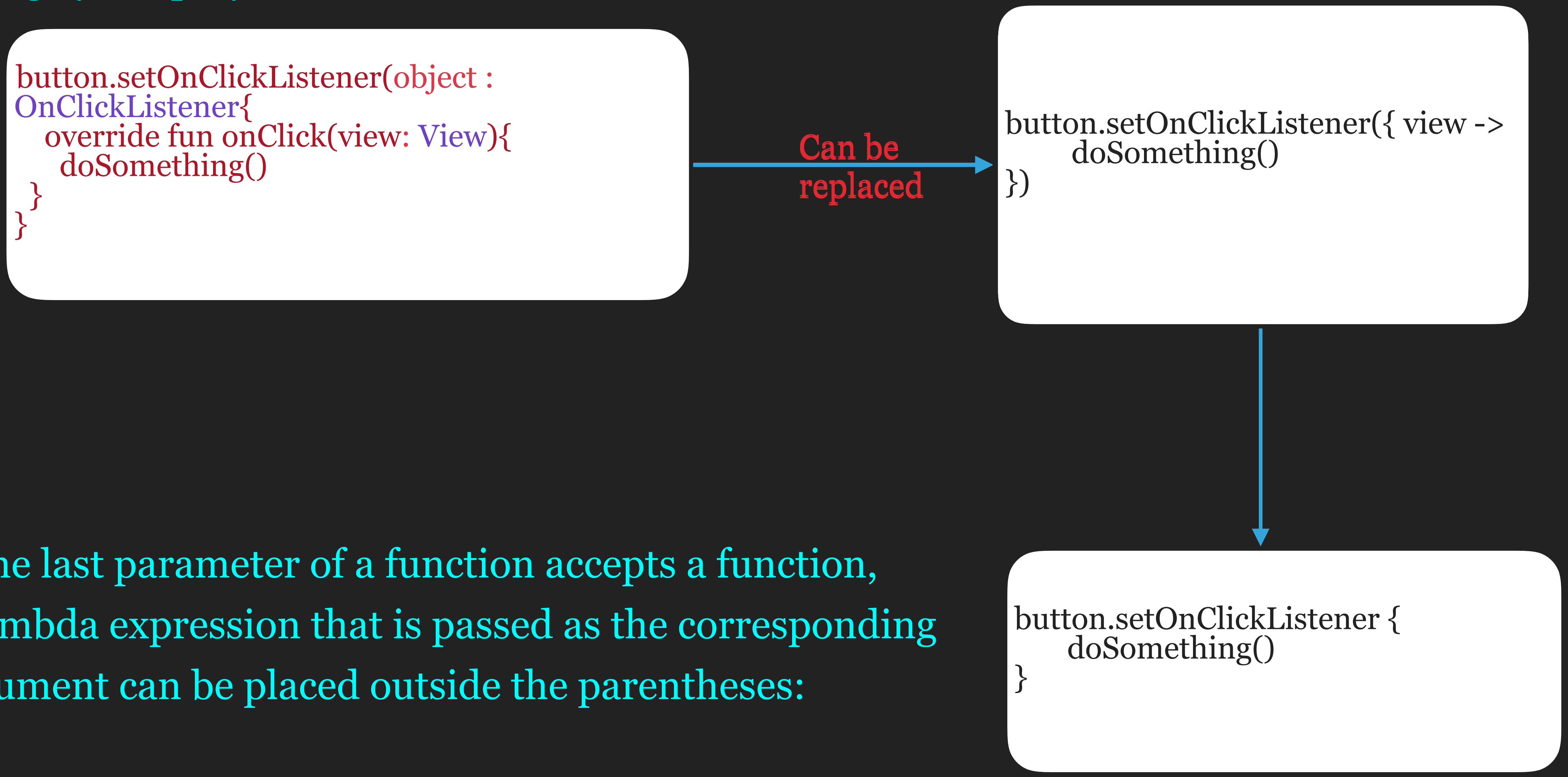
A lambda expression that takes one argument of type String, and prints it. It returns Unit. Its type is (String)->Unit.

```
{ a: Int, b: Int -> a + b }
```

A lambda expression that takes two Int arguments and returns the sum of them. Its type is (Int, Int)->Int.

LAMBDAS

- Kotlin lets us write interfaces with a single function as if they were a lambda, so we can hugely simplify our code.



```
button.setOnClickListener(object :
OnClickListener{
   override fun onClick(view: View){
     doSomething()
  }
}
```

Can be replaced

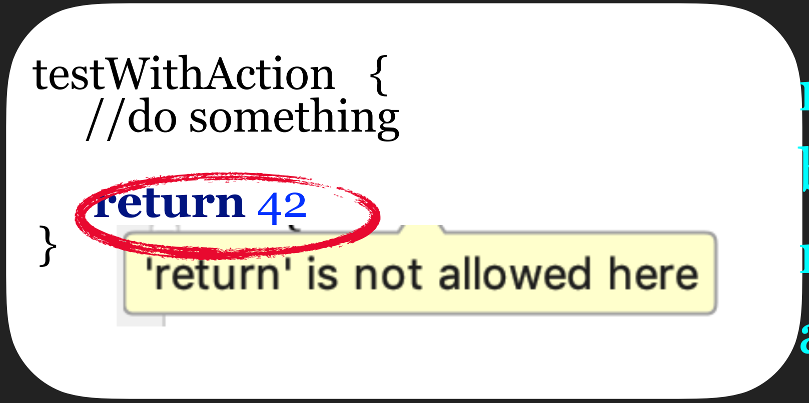
```
button.setOnClickListener({ view ->
        doSomething()
})
```

If the last parameter of a function accepts a function, a lambda expression that is passed as the corresponding argument can be placed outside the parentheses:

```
button.setOnClickListener {
       doSomething()
}
```

RETURNING FROM LAMBDAS

```
fun testWithAction(action:() -> Int) {
    println("start test")
    action()
    println("end test”)
}

testWithAction {
   //do Something
   42
 }
```



```
testWithAction  {
    //do something

   return 42
}
```

'return' is not allowed here

return can only be used to exit a named function or an anonymous function.

- If you want to make an explicit *return* statement in a lambda, use the **return@label** syntax.This is also known as labelled return.

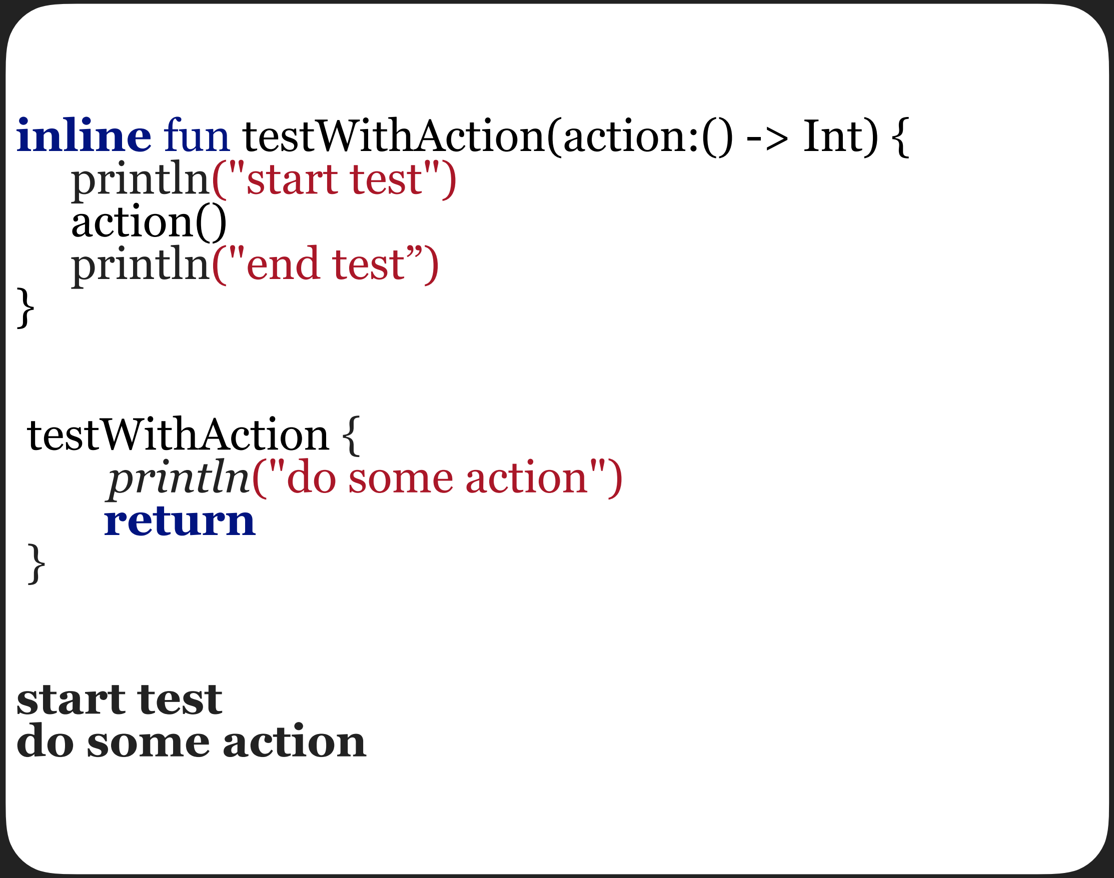
```
testWithAction {
   //do something
   return@testWithAction 42
}
```

```
testWithAction marker@ {
   //do something
   return@marker 42
}
```

NON LOCAL RETURNS

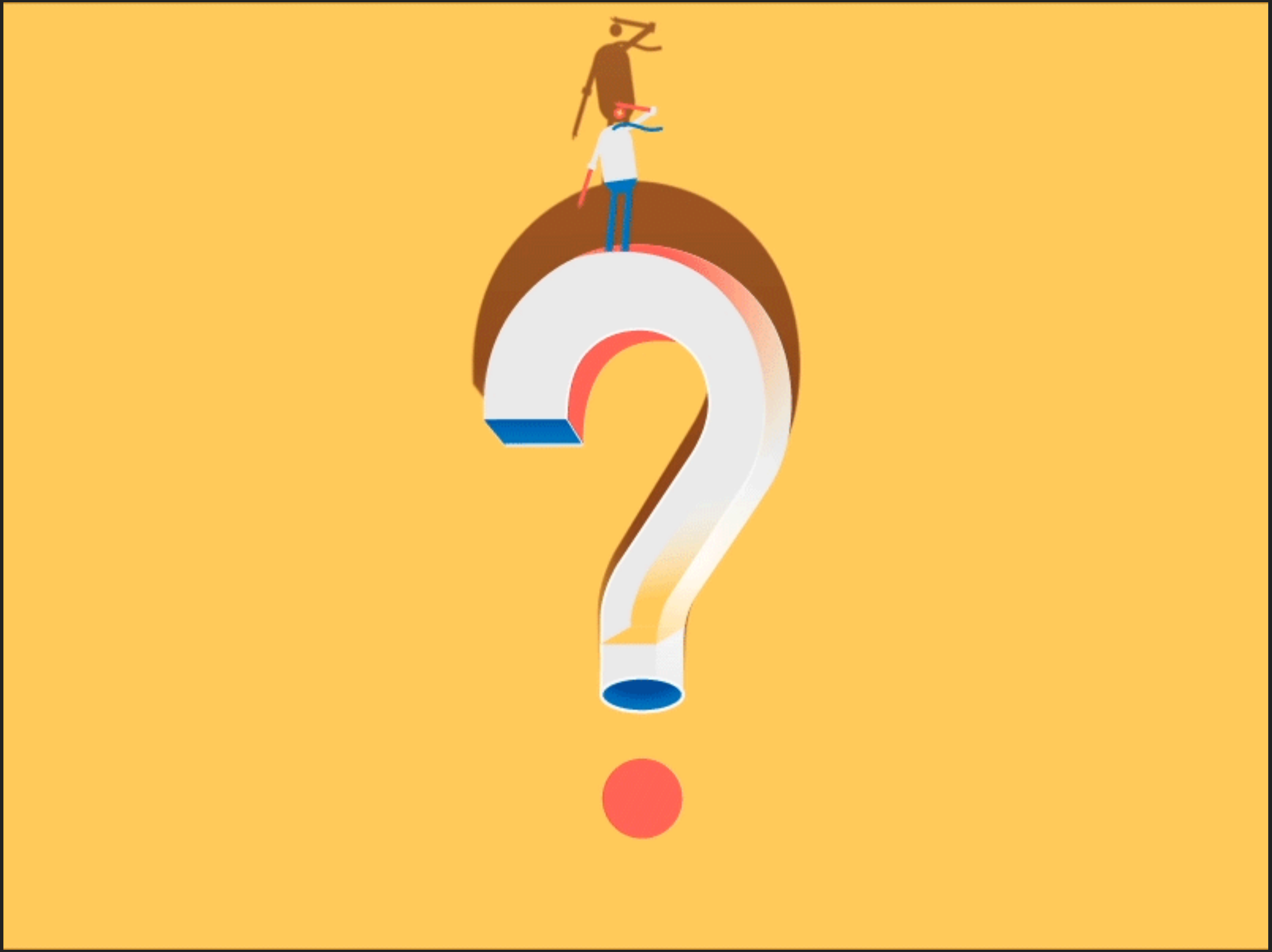
- What if I want to exit the function in which lambda is being invoked?

- If the function lambda is passed to is inlined, the return is inlined too and we can exit the enclosing function(also called non-local returns).



```
inline fun testWithAction(action:() -> Int) {
    println("start test")
    action()
    println("end test”)
}


 testWithAction {
       println("do some action")
       return
 }
```

```
start test
do some action
```



Thanks!