


KOTLIN CONCEPTS



TODAY'S AGENDA



- Kotlin Concepts
 - ▣ Functions
 - ▣ Exceptions
 - ▣ Java Interops

- 
- Functions are at the heart of Kotlin, the following kinds of functions in Kotlin:
 - ▣ top-level functions
 - ▣ Member functions
 - ▣ **Local/Nested Functions**
 - ▣ lambda expressions or function literals
 - ▣ Higher order functions
 - ▣ anonymous functions
 - ▣ infix functions

TOP LEVEL FUNCTIONS

- Top-level functions are functions inside a Kotlin package that are defined outside of any class, object, or interface. This means that they are functions you call directly, without the need to create any object or call any class.
- A typical example is the Collections class in the `java.util` package and its static methods.
- Top-level functions in Kotlin can be used as a replacement for the static utility methods inside helper classes we code in Java. Let's look at how to define a top-level function in Kotlin.

Member Functions

- Member functions are functions which are defined inside a class or an object.
- Member functions are called on the objects of the class using the dot(.) notation

Local/Nested Functions

- Kotlin allows you to nest function definitions. These nested functions are called Local functions. Local functions bring more encapsulation and readability to your program -
-

Lambda Expressions

- Lambda expression or simply lambda is an anonymous function; a function without name. These functions are passed immediately as an expression without declaration.
- **Lambda With Parameters**
- A lambda expression can have one or more parameters. You specify the parameters before the \rightarrow symbol followed by codeBody. The compiler interprets the return type of lambda by the last executable statement in its body:

A diagram illustrating the components of a lambda expression. The code is `val lambda: (Int, Int) → Unit = { param1: Int, param2: Int → param1 + param2 }`. Red arcs and labels identify the parts: 'variable' points to `val lambda`, 'parameters' points to `(Int, Int)`, 'type' points to `→ Unit`, and 'body' points to `{ param1: Int, param2: Int → param1 + param2 }`.

```
val lambda: (Int, Int) → Unit = { param1: Int, param2: Int → param1 + param2 }
```


Higher-Order Function

- ❑ Kotlin has a great support for functional programming. You can pass functions as arguments to other functions. Also, you can return a function from other functions. These functions are called higher-order functions.
- ❑ Often, lambda expressions are passed to higher-order function (rather than a typical function) for convenience.

Anonymous Function

- An anonymous function is very similar to regular function except for the name of the function which is omitted from the declaration. The body of the anonymous function can be either an expression or block.
- **Example 1:** Function body as an **expression**
 - `fun(a: Int, b: Int) : Int = a * b`
- **Example 2:** Function body as a **block**
 - `fun(a: Int, b: Int): Int { val mul = a * b return mul }`

Infix functions

- Ever imagined calling a public function of a class without dot and parentheses of the parameter in Kotlin. Kotlin provides infix notation with which we can call a function with the class object without using a dot and parentheses across the parameter. Using infix function provides more readability to a function similar to other operators like in, is, as in Kotlin.

- To make a function infix notation enabled, add infix keyword before the function.



```
infix fun Int.add(b : Int) : Int = this + b  
val x = 10.add(20)  
val y = 10 add 20 // infix call
```

- But an Infix function must satisfy the following requirements
 - They must be member functions or extension functions.
 - They must have a single parameter.
 - The parameter must not accept a variable number of arguments and must have no default value.

Exceptions

- ❑ Exceptions are unwanted issues that can occur at runtime of the program and terminate your program abruptly. Exception handling is a process, using which we can prevent the program from such exceptions that can break our code.
- ❑ There are two types of exceptions:
 1. Checked exceptions that are declared as part of method signature and are checked at the compile time, for example `IOException`
 2. Unchecked exceptions do not need to be added as part of method signature and they are checked at the runtime, for example `NullPointerException`.
- ❑ **Note: In Kotlin all exceptions are unchecked.**
- ❑ Handling of exception in Kotlin is same as Java. We use try, catch and finally block to handle the exceptions in the kotlin code.