Kotlin

* Can run anywhere where JVM can run
* Statically typed – the type of every variable is known by a compile time; it doesn’t figure out the types at runtime
* Object oriented
* However it can use Functional programming – you can use functions as values, store them in variables, return them from functions and pass them as arguments to functions, also you can declare immutable object; functional programming allows Kotlin to be concise(кратък, сбит, стегнат) which means you have to write less code to accomplish sth than in Java
* Guiding principles:
* Conciseness
* Safety – null pointer exceptions
* Pragmatism – it is not a research language, Kotlin is designed to solve real world problems; there is always more than one way to do sth in Kotlin; in Kotlin world no way is considered better than the other
* Interoperability – Kotlin was designed to be interoperable with Java

-No need to declare exceptions as in Java; ‘throws’ keyword doesn’t exist in Kotlin

- The Ternary operator is gone and replaced by the if expression

- The for loop doesn’t exist in Kotlin – e.g -> for(int i=0; i<20; i++)

- Kotlin doesn’t have a ‘static’ keyword – replaced with top-level functions, the concept is still there but not visible; the Kotlin code is still being compiled down to java bytecode when targeting the JVM

- No ‘new’ keyword when creating a new object of a class

- !!! Main difference here in Kotlin is the ‘==’ operator:

\* in Kotlin the ‘==’ operator checks for structural equality (while in Java it is only checking for referential equality and ‘.equals’ checks for structural equality) when it is comparing instances of classes; actually refers to the ‘.equals’ operator; it is also a safe operator and won’t give a null pointer expetion

\* in Kotlin the ‘===’ operator checks for a referential equality

-a function that returns Unit is like a void function in Java, however here a unit function actually returns a singleton unit instance

- ?: Elvis operator; lets you assign a default value when an expression evaluates to null

- safe call and safe cast operators used to ignore Null Pointer Exceptions; meaning that the app won’t crash

- with a safe call (?) we never get a Null Pointer Exception, however with (!!) operator we can get an exception and address the exception

-the default visibility is public; you can have as many public classes in one file as you want; here you can declare also a private class which means that everything in the same file will have an access to that class

- extension functions lets you extend any class you want; when we say extend this means adding functions to it; you are not extending it the case of sub-classing it; extension functions are a syntactical illusion; Receiver type(this is the type that is being extended) and Receiver object(this is the object that we use to call the extension function)

-extension are used a lot in Kotlin;

-when an inline function is being compiled it is not being compiled to a function but to its body; inline functions are usually used with lambda expressions because lambdas usually require the creation of a class and an object under the covers; so by inlining a function that takes a lambda this extra overhead isn’t required;

- the function body of an inline function will be directly generated into the bytecode wherever the function is called rather than having a function call;

- data classes are closed typed; they can’t be extended; they can’t be abstract; they can extend other classes but cannot be extended by other classes

- interfaces cannot be declared ‘open’ just because by default they are extendable; interfaces don’t have a backing field;

- when you want to use Singleton in Kotlin you can use the ‘object’ keyword because you can only have one instance of an Object Class; when you are declaring an Object Class you are saying that you want one and only one instance of a class

-companion objects – since we cannot use ‘static’ keyword in Kotlin, we can think of companion objects as if everything inside of them is being static and can be accessed without needing to create an instance of that class; we can also use them to call private constructors; in fact we can use them to implement the Factory pattern

- you can also use ‘object’ when you want to create anonymous instances???

- you use enum class to create an enum in Kotlin

- you can import top-level items like functions, extension functions;

-to import an extension function which are top-level functions, you can import by using the name

- ‘internal’ modifier can be used only in the same module but not be visible in another module

- to clean imports that are not used – Ctrl + Alt + O or Code -> Optimize imports

- if can be used as an expression; we can assign it to a variable, w can pass it to a function

- Lambdas can access local variables as long as they are declared before the lambda; when lambdas are used within a function, they can access the function parameters