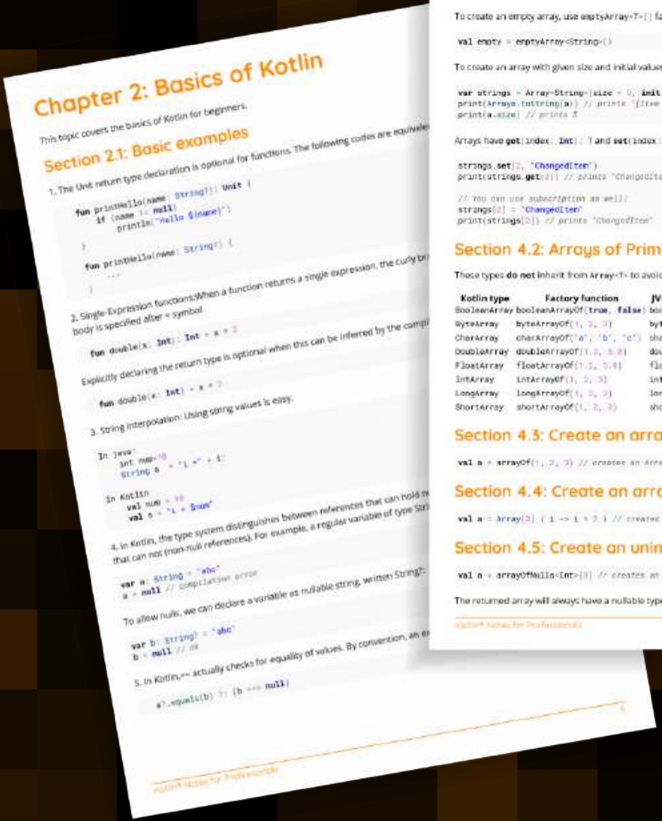
Kotlin®

Notes for Professionals



Chapter 2: Basics of Kotlin

This topic covers the basics of Kotlin for beginners.

Section 2.1: Basic examples

1. The Unit return type declaration is optional for functions. The following codes are equivale

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
}
fun printHello(name: String?) {
    ...
}
```

2. Single-Expression functions:When a function returns a single expression, the curly br
body is specified after = symbol

```
fun double(x: Int): Int = x * 2
```

Explicitly declaring the return type is optional when this can be inferred by the compi

```
fun double(x: Int) = x * 2
```

3. String interpolation: Using string values is easy.

```
In java:
    int num=10
    String s = "i =" + i;

In Kotlin
    val num = 10
    val s = "i = $num"
```

4. In Kotlin, the type system distinguishes between references that can hold n
that can not (non-null references). For example, a regular variable of type Stri

```
var a: String = "abc"
a = null // compilation error
```

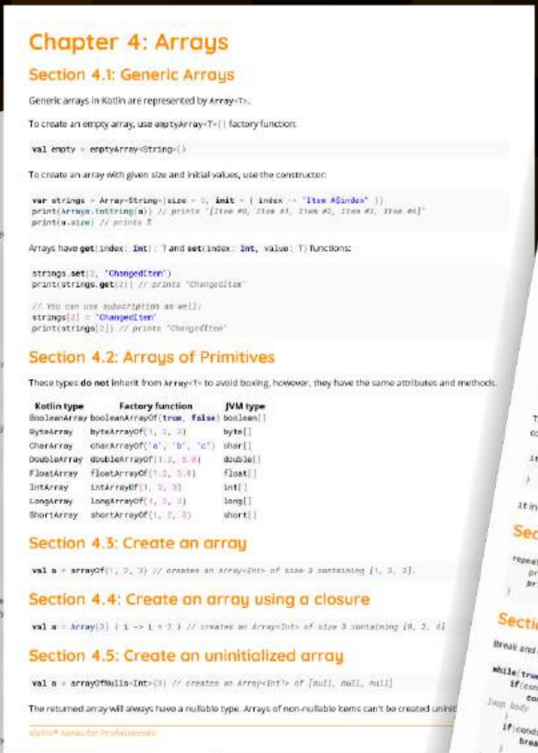
To allow nulls, we can declare a variable as nullable string, written String?:

```
var b: String? = "abc"
b = null // ok
```

5. In Kotlin,== actually checks for equality of values. By convention, an ex

```
a?.equals(b) ?: (b === null)
```

Kotlin® Notes for Professionals 6



Chapter 4: Arrays

Section 4.1: Generic Arrays

Generic arrays in Kotlin are represented by Array<T>.

To create an empty array, use emptyArray<T>() factory function:

```
val empty = emptyArray<String>()
```

To create an array with given size and initial values, use the constructor:

```
var strings = Array<String>(size = 5, init = { index -> "Item #$index" })
print(Arrays.toString(a)) // prints "[Item #0, Item #1, Item #2, Item #3, Item #4]"
print(a.size) // prints 5
```

Arrays have get(index: Int): T and set(index: Int, value: T) functions:

```
strings.set(2, "ChangedItem")
print(strings.get(2)) // prints "ChangedItem"

// You can use subscription as well:
strings[2] = "ChangedItem"
print(strings[2]) // prints "ChangedItem"
```

Section 4.2: Arrays of Primitives

These types **do not** inherit from Array<T> to avoid boxing, however, they have the same attributes and methods.

Kotlin type	Factory function	JVM type
BooleanArray	booleanArrayOf(true, false)	boolean[]
ByteArray	byteArrayOf(1, 2, 3)	byte[]
CharArray	charArrayOf('a', 'b', 'c')	char[]
DoubleArray	doubleArrayOf(1.2, 5.0)	double[]
FloatArray	floatArrayOf(1.2, 5.0)	float[]
IntArray	intArrayOf(1, 2, 3)	int[]
LongArray	longArrayOf(1, 2, 3)	long[]
ShortArray	shortArrayOf(1, 2, 3)	short[]

Section 4.3: Create an array

```
val a = arrayOf(1, 2, 3) // creates an Array<Int> of size 3 containing [1, 2, 3].
```

Section 4.4: Create an array using a closure

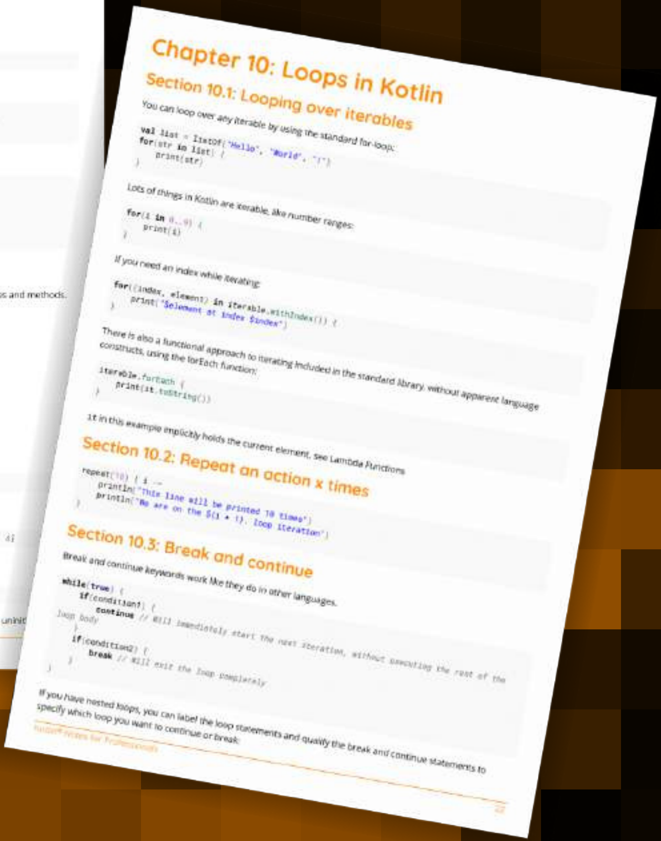
```
val a = Array(3) { i -> i * 2 } // creates an Array<Int> of size 3 containing [0, 2, 4]
```

Section 4.5: Create an uninitialized array

```
val a = arrayOfNulls<Int>(3) // creates an Array<Int?> of [null, null, null]
```

The returned array will always have a nullable type. Arrays of non-nullable items can't be created uninit

Kotlin® Notes for Professionals



Chapter 10: Loops in Kotlin

Section 10.1: Looping over iterables

You can loop over any iterable by using the standard for-loop:

```
val list = listOf("Hello", "World", "!")
for(str in list) {
    print(str)
}
```

Lots of things in Kotlin are iterable, like number ranges:

```
for(i in 0..9) {
    print(i)
}
```

If you need an index while iterating:

```
for((index, element) in iterable.withIndex()) {
    print("$element at index $index")
}
```

There is also a functional approach to iterating included in the standard library, without apparent language constructs, using the forEach function:

```
iterable.forEach {
    print(it.toString())
}
```

it in this example implicitly holds the current element, see Lambda Functions

Section 10.2: Repeat an action x times

```
repeat(10) { i ->
    println("This line will be printed 10 times")
    println("We are on the ${i + 1}. loop iteration")
}
```

Section 10.3: Break and continue

Break and continue keywords work like they do in other languages.

```
while(true) {
    if(condition1) {
        continue // Will immediately start the next iteration, without executing the rest of the
loop body
    }
    if(condition2) {
        break // Will exit the loop completely
    }
}
```

If you have nested loops, you can label the loop statements and qualify the break and continue statements to specify which loop you want to continue or break:

Kotlin® Notes for Professionals 22

80+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is not affiliated with official Kotlin® group(s) or company(s).
All trademarks and registered trademarks are the property of their respective owners

Contents

About 1

Chapter 1: Getting started with Kotlin 2

Section 1.1: Hello World 2

Section 1.2: Hello World using a Companion Object 2

Section 1.3: Hello World using an Object Declaration 3

Section 1.4: Main methods using varargs 4

Section 1.5: Compile and Run Kotlin Code in Command Line 4

Section 1.6: Reading input from Command Line 4

Chapter 2: Basics of Kotlin 6

Section 2.1: Basic examples 6

Chapter 3: Strings 7

Section 3.1: String Equality 7

Section 3.2: String Literals 7

Section 3.3: Elements of String 8

Section 3.4: String Templates 8

Chapter 4: Arrays 9

Section 4.1: Generic Arrays 9

Section 4.2: Arrays of Primitives 9

Section 4.3: Create an array 9

Section 4.4: Create an array using a closure 9

Section 4.5: Create an uninitialized array 9

Section 4.6: Extensions 10

Section 4.7: Iterate Array 10

Chapter 5: Collections 11

Section 5.1: Using list 11

Section 5.2: Using map 11

Section 5.3: Using set 11

Chapter 6: Enum 12

Section 6.1: Initialization 12

Section 6.2: Functions and Properties in enums 12

Section 6.3: Simple enum 12

Section 6.4: Mutability 12

Chapter 7: Functions 14

Section 7.1: Function References 14

Section 7.2: Basic Functions 15

Section 7.3: Inline Functions 16

Section 7.4: Lambda Functions 16

Section 7.5: Operator functions 16

Section 7.6: Functions Taking Other Functions 17

Section 7.7: Shorthand Functions 17

Chapter 8: Vararg Parameters in Functions 18

Section 8.1: Basics: Using the vararg keyword 18

Section 8.2: Spread Operator: Passing arrays into vararg functions 18

Chapter 9: Conditional Statements 19

Section 9.1: When-statement argument matching 19

Section 9.2: When-statement as expression 19

Section 9.3: Standard if-statement 19
Section 9.4: If-statement as an expression 19
Section 9.5: When-statement instead of if-else-if chains 20
Section 9.6: When-statement with enums 20

Chapter 10: Loops in Kotlin 22

Section 10.1: Looping over iterables 22
Section 10.2: Repeat an action x times 22
Section 10.3: Break and continue 22
Section 10.4: Iterating over a Map in kotlin 23
Section 10.5: Recursion 23
Section 10.6: While Loops 23
Section 10.7: Functional constructs for iteration 23

Chapter 11: Ranges 25

Section 11.1: Integral Type Ranges 25
Section 11.2: downTo() function 25
Section 11.3: step() function 25
Section 11.4: until function 25

Chapter 12: Regex 26

Section 12.1: Idioms for Regex Matching in When Expression 26
Section 12.2: Introduction to regular expressions in Kotlin 27

Chapter 13: Basic Lambdas 30

Section 13.1: Lambda as parameter to filter function 30
Section 13.2: Lambda for benchmarking a function call 30
Section 13.3: Lambda passed as a variable 30

Chapter 14: Null Safety 31

Section 14.1: Smart casts 31
Section 14.2: Assertion 31
Section 14.3: Eliminate nulls from an Iterable and array 31
Section 14.4: Null Coalescing / Elvis Operator 31
Section 14.5: Nullable and Non-Nullable types 32
Section 14.6: Elvis Operator (?:) 32
Section 14.7: Safe call operator 32

Chapter 15: Class Delegation 34

Section 15.1: Delegate a method to another class 34

Chapter 16: Class Inheritance 35

Section 16.1: Basics: the 'open' keyword 35
Section 16.2: Inheriting fields from a class 35
Section 16.3: Inheriting methods from a class 36
Section 16.4: Overriding properties and methods 36

Chapter 17: Visibility Modifiers 38

Section 17.1: Code Sample 38

Chapter 18: Generics 39

Section 18.1: Declaration-site variance 39
Section 18.2: Use-site variance 39

Chapter 19: Interfaces 41

Section 19.1: Interface with default implementations 41
Section 19.2: Properties in Interfaces 42
Section 19.3: super keyword 42
Section 19.4: Basic Interface 42

Section 19.5: Conflicts when Implementing Multiple Interfaces with Default Implementations 43
Chapter 20: Singleton objects 44
Section 20.1: Use as replacement of static methods/fields of java 44
Section 20.2: Use as a singleton 44
Chapter 21: coroutines 45
Section 21.1: Simple coroutine which delay's 1 second but not blocks 45
Chapter 22: Annotations 46
Section 22.1: Meta-annotations 46
Section 22.2: Declaring an annotation 46
Chapter 23: Type aliases 47
Section 23.1: Function type 47
Section 23.2: Generic type 47
Chapter 24: Type-Safe Builders 48
Section 24.1: Type-safe tree structure builder 48
Chapter 25: Delegated properties 49
Section 25.1: Observable properties 49
Section 25.2: Custom delegation 49
Section 25.3: Lazy initialization 49
Section 25.4: Map-backed properties 49
Section 25.5: Delegate Can be used as a layer to reduce boilerplate 49
Chapter 26: Reflection 51
Section 26.1: Referencing a class 51
Section 26.2: Inter-operating with Java reflection 51
Section 26.3: Referencing a function 51
Section 26.4: Getting values of all properties of a class 51
Section 26.5: Setting values of all properties of a class 52
Chapter 27: Extension Methods 54
Section 27.1: Potential Pitfall: Extensions are Resolved Statically 54
Section 27.2: Top-Level Extensions 54
Section 27.3: Lazy extension property workaround 54
Section 27.4: Sample extending Java 7+ Path class 55
Section 27.5: Sample extending long to render a human readable string 55
Section 27.6: Sample extending Java 8 Temporal classes to render an ISO formatted string 55
Section 27.7: Using extension functions to improve readability 55
Section 27.8: Extension functions to Companion Objects (appearance of Static functions) 56
Section 27.9: Extensions for easier reference View from code 57
Chapter 28: DSL Building 58
Section 28.1: Infix approach to build DSL 58
Section 28.2: Using operators with lambdas 58
Section 28.3: Overriding invoke method to build DSL 58
Section 28.4: Using extensions with lambdas 58
Chapter 29: Idioms 60
Section 29.1: Serializable and serialVersionUid in Kotlin 60
Section 29.2: Delegate to a class without providing it in the public constructor 60
Section 29.3: Use let or also to simplify working with nullable objects 61
Section 29.4: Use apply to initialize objects or to achieve method chaining 61
Section 29.5: Fluent methods in Kotlin 61
Section 29.6: Filtering a list 62
Section 29.7: Creating DTOs (POJOs/POCOs) 62

Chapter 30: RecyclerView in Kotlin 63
Section 30.1: Main class and Adapter 63
Chapter 31: logging in kotlin 65
Section 31.1: kotlin.logging 65
Chapter 32: Exceptions 66
Section 32.1: Catching exception with try-catch-finally 66
Chapter 33: JUnit 67
Section 33.1: Rules 67
Chapter 34: Kotlin Android Extensions 68
Section 34.1: Using Views 68
Section 34.2: Configuration 68
Section 34.3: Painful listener for getting notice, when the view is completely drawn now is so simple and awesome with Kotlin's extension 69
Section 34.4: Product flavors 69
Chapter 35: Kotlin for Java Developers 71
Section 35.1: Declaring Variables 71
Section 35.2: Quick Facts 71
Section 35.3: Equality & Identity 71
Section 35.4: IF, TRY and others are expressions, not statements 72
Chapter 36: Java 8 Stream Equivalents 73
Section 36.1: Accumulate names in a List 73
Section 36.2: Collect example #5 - find people of legal age, output formatted string 73
Section 36.3: Collect example #6 - group people by age, print age and names together 73
Section 36.4: Different Kinds of Streams #7 - lazily iterate Doubles, map to Int, map to String, print each 74
Section 36.5: Counting items in a list after filter is applied 75
Section 36.6: Convert elements to strings and concatenate them, separated by commas 75
Section 36.7: Compute sum of salaries of employee 75
Section 36.8: Group employees by department 75
Section 36.9: Compute sum of salaries by department 75
Section 36.10: Partition students into passing and failing 75
Section 36.11: Names of male members 76
Section 36.12: Group names of members in roster by gender 76
Section 36.13: Filter a list to another list 76
Section 36.14: Finding shortest string a list 76
Section 36.15: Different Kinds of Streams #2 - lazily using first item if exists 76
Section 36.16: Different Kinds of Streams #3 - iterate a range of Integers 77
Section 36.17: Different Kinds of Streams #4 - iterate an array, map the values, calculate the average 77
Section 36.18: Different Kinds of Streams #5 - lazily iterate a list of strings, map the values, convert to Int, find max 77
Section 36.19: Different Kinds of Streams #6 - lazily iterate a stream of Ints, map the values, print results 77
Section 36.20: How streams work - filter, upper case, then sort a list 78
Section 36.21: Different Kinds of Streams #1 - eager using first item if it exists 78
Section 36.22: Collect example #7a - Map names, join together with delimiter 78
Section 36.23: Collect example #7b - Collect with SummarizingInt 79
Chapter 37: Kotlin Caveats 81
Section 37.1: Calling a toString() on a nullable type 81

Click here to download full PDF material