ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΑΤΜΗΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΣΤΟΝ ΗΛΕΚΤΡΟΝΙΚΟ ΑΥΤΟΜΑΤΙΣΜΟ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παρουσίαση χαρακτηριστικών της γλώσσας προγραμματισμού Kotlin. Σύγκριση με υφιστάμενες δημοφιλείς γλώσσες προγραμματισμού και σχεδίαση και υλοποίηση εφαρμογής σε περιβάλλον Android.

Σωκράτης Σ. Αρβανίτης

Επιβλέπων: **Δημήτρης Κούτουλας,** Καθηγητής

ΑΘΗΝΑ

ΑΠΡΙΛΙΟΣ 2019

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παρουσίαση χαρακτηριστικών της γλώσσας προγραμματισμού Kotlin. Σύγκριση με υφιστάμενες δημοφιλείς γλώσσες προγραμματισμού και σχεδίαση και υλοποίηση εφαρμογής σε περιβάλλον Android.

Σωκράτης Σ. Αρβανίτης

Α.Μ.: 2015524

ΕΠΙΒΛΕΠΩΝ: **Δημήτρης Κούτουλας,** Καθηγητής

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Δημήτρης Κούτουλας,** Καθηγητής
Στάθης Χατζιευθυμιάδης, Καθηγητής
Δημήτρης Βαρουτάς, Αναπληρωτής Καθηγητής

Απρίλιος 2019

ΠΕΡΙΛΗΨΗ

Η Kotlin είναι μια στατική γλώσσα προγραμματισμού που μπορεί να εκτελεστεί στη Java Virtual Machine (JVM). Είναι μια γλώσσα προγραμματισμού ανοικτού κώδικα (OP), γενικής χρήσης που συνδυάζει τόσο αντικειμενοστραφή όσο και συναρτησιακά χαρακτηριστικά. Η Kotlin είναι σημαντική σήμερα για δύο λόγους. Έχει αναπτυχθεί ως λύση στα προβλήματα που αντιμετωπίζουν οι προγραμματιστές Android στην σημερινή χρονική περίοδο. Ως εκ τούτου, απαντά τα περισσότερα από τα βασικά ζητήματα που εμφανίστηκαν στη Java, παρέχοντας στους προγραμματιστές διαλειτουργικότητα, ασφάλεια, σαφήνεια και υποστήριξη πολλών εργαλείων. Ο ενθουσιασμός γύρω από την Kotlin δημιουργείται από το γεγονός ότι η Google την χαρακτήρισε επίσημα ως την γλώσσα για την ανάπτυξη εφαρμογών Android. Ο πρωταρχικός λόγος αυτής της υιοθέτησης ήταν η νεωτερικότητα - από την άποψη της ισχύος, της ευελιξίας και της ανοικτής προσέγγισης. Η Kotlin είναι επίσης συνεργάσιμη με τις υπάρχουσες γλώσσες Android.

Η διπλωματική εργασία παρουσιάζει την γλώσσα προγραμματισμού Kotlin, μία σύγκριση της Java και της Kotlin και μία κατασκευή ενδεικτικής εφαρμοφής στη γλώσσα προγραμματισμού Kotlin σε Android Studio.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Γλώσσες Προγραμματισμού στο JVM

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Βασικοί Τύποι, Συναρτήσεις, Αντικειμενοστραφής Προγραμματισμός, Συλλογές, Πίνακες, Λίστες, Σύνολα

ABSTRACT

Kotlin is a statically-typed programming language that can run on the Java Virtual Machine (JVM). It is an open source, general purpose and pragmatic computer programming language that combines both the object-oriented and functional programming features within it. Kotlin is relevant today because of two reasons. It has been developed as a solution to the problems that Android developers have faced over a period of time. Therefore, it answers most of the main issues that surfaced in Java, providing developers with inter-operability, safety, clarity, and tooling support. The excitement around Kotlin is generated by the fact that Google has lapped it up as the go-to language for Android app development. The prime reason for this adoption was its modernity – in terms of its power, flexibility and democratic approach. Kotlin is also interoperable with the existing Android languages.

The thesis presents the Kotlin programming language, a comparison of Java and Kotlin and an implementation of an indicative application in the Kotlin programming language on Android Studio.

SUBJECT AREA: Programming Languages in JVM

KEYWORDS: Basic Types, Functions, Object – Oriented Programming, Collections, Arrays, Lists, Sets

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να ευχαριστήσω θερμά τον καθηγητή μου κ. Δημήτρη Κούτουλα για την απεριόριστη υπομονή του, την αμέριστη υποστήριξη, τις ουσιώδεις συμβουλές, καθώς επίσης και την αδιάκοπη συμπαράσταση και ενθάρρυνση που μου παρείχε σε όλο αυτό το χρονικό διάστημα.

Επίσης θέλω να απευθύνω τις ευχαριστίες μου στην οικογένεια μου, για την στήριξη και την εμπιστοσύνη που μου έδειξε όλα αυτά τα χρόνια των σπουδών μου. Πέραν όμως από την πολύτιμη αυτή στήριξη, μου έδωσαν όλα τα εφόδια ώστε να γίνω ένας σωστός Άνθρωπος και αυτό είναι κάτι το οποίο δεν μαθαίνεται, αλλά μεταδίδεται.

ΠΕΡΙΕΧΟΜΕΝΑ

1. Βασικά Στοιχεία της Kotlin 9
1.1 Ένα πρόγραμμα “Hello World” 9
1.2 Βασικοί Τύποι Δεδομένων (Basic Types) 10
1.3 Μεταβλητές (Variables) 17
1.4 Null Safety 18
1.5 Σχόλια (Comments) 19
1.6 Έλεγχοι Ισότητας (Equality checks) 19
1.7 Ροές Ελέγχου (Control flows) 20
1.8 Διαχείρηση Εξαιρέσεων με τη Συνθήκη try…catch 23
1.9 Βρόχοι Επανάληψης (Loops) 24
1.10 Εύρη Τιμών (Ranges) 27

2. Συναρτήσεις 28
2.1 Όρισμός Συναρτήσεων (Define Functions) 28
2.2 Εν Σειρά Συναρτήσεις (Inline Functions) 33
2.3 Λάμδα Συναρτήσεις (Lamda Functions) 34
2.4 Συναρτήσεις Υψηλότερης Τάξης (High Order Functions) 36
2.5 Επεκτάσεις (Extensions) 38
2.6 Γενικές Συναρτήσεις (Generic Functions) 39
2.7 Τυπικές Συναρτήσεις (Standard Functions) 40

3. Αντικειμενοστραφής προγραμματισμός 43
3.1 Κλάσεις (Classes) και Κατασκευαστές (Constructors) 43
3.2 Κληρονομικότητα (Inheritance) 46
3.3 Αφηρημένες Κλάσεις (Abstract Classes) 47
3.4 Διεπαφές (Interfaces) 48
3.5.1 Ένθετες Κλάσεις (Nested Classes) 49
3.5.2 Εσωτερικές Κλάσεις (Inner Classes) 50

3.5.3 Σφραγισμένες Κλάσεις (Sealed Classes) 51
3.6.1 Κλάσεις Απαρίθμησης (Enum Classes) 52
3.6.2 Κλάσεις Δεδομένων (Data Classes) 53
3.7 Γενικές Κλάσεις (Generic Classes) 54

4. Συλλογές 55
4.1 Όρισμός Συλλογών (Define ollections) 55
4.2 Πίνακες (Arrays) 58
4.3 Λίστες (Lists) 65
4.4 Σύνολα (Sets) 81
4.5 Χάρτες (Maps) 87

5. Διαφορές Java και kotlin 90
5.1 Θεμελιώδης διαφορές Java και Kotlin 90
5.2 Διαφορές Java και Kotlin ως προς τα βασικά τους 92
5.3 Διαφορές Java και Kotlin ως προς τις συναρτήσεις 98
5.4 Διαφορές Java και Kotlin ως προς τις κλάσεις 102
5.5 Διαφορές Java και Kotlin ως προς τις συλλογές 110

ΑΚΡΩΝΥΜΙΑ 113

ΠΑΡΑΡΤΗΜΑ 114

ΑΝΑΦΟΡΕΣ 115

1. ΒΑΣΙΚΑ ΣΤΟΙΧΕΙΑ ΤΗΣ KOTLIN

1.1 Ένα πρόγραμμα “Hello World”

Ένα πρόγραμμα “Hello World” είναι ένα απλό πρόγραμμα, το οποίο τυπώνει στην οθόνη το μήνυμα “Hello World”. Δεδομένου ότι είναι πολύ απλό, χρησιμοποιείται συχνά για να εισάγει μια νέα γλώσσα προγραμματισμού.

Code Examples

```
/* Στη Kotlin ο κώδικας ορίζεται συνήθως μέσα σε πακέτα. Εάν
   δεν οριστεί κάποιο συγκεκριμένο πακέτο, θα χρησιμοποιηθεί
   το προεπιλεγμένο πακέτο (DP).*/
package org.kotlinlang.play

/* Το κύριο σημείο εισόδου σε μια Kotlin εφαρμογή, είναι μία
   συνάρτηση που ονομάζεται main. */
fun main(args: Array<String>) {

    print("Hello World!")

    /* Η συνάρτηση println εκτυπώνει στην οθόνη το μήνυμα:
       Hello World! και αλλάζει γραμμή στην οθόνη εμφάνισης.
       Σημειώνεται ότι το ερωτηματικό (;) είναι προαιρετικό. */
    println("Hello World!")
}
----------------------------------------------------------------


/* Από την έκδοση της Kotlin 1.3 και μετά η συνάρτηση main
   μπορεί να είναι μια συνάρτηση χωρίς παραμέτρους. */
fun main() {

    print("Hello World!")
    println("Hello World!")
}
```

1.2 Βασικοί Τύποι Δεδομένων (Basic Types)

Στη Kotlin όλα είναι ένα αντικείμενο. Οι βασικοί τύποι δεδομένων που χρησιμοποιούνται είναι: Αριθμοί (Numbers), Χαρακτήρες (Characters), Λογικές (Booleans), Πίνακες (Arrays) και Συμβολοσειρές (Strings).

Αριθμοί (Numbers):

Η Kotlin χειρίζεται τους αριθμούς με έναν τρόπο κοντά στην Java, αλλά δεν είναι ακριβώς το ίδιο. Παρακάτω δίδεται ένα πίνακας με τους ακόλουθους τύπους δεδομένων που αντιπροσωπεύουν τους αριθμούς.

Type	Double	Float	Long	Int	Short	Byte
Bit Width	64	32	64	32	16	8

Code Examples

```
/* Για την δημιουργία ενός αριθμού, μπορεί να
   χρησιμοποιηθεί μία από τις παρακάτω μορφές: */
val int = 123
val long = 123456L
val double = 12.34
val float = 12.34F
val hexadecimal = 0xAB
val binary = 0b01010101
-------------------------------------------------------------

/* Η κάτω παύλα ( _ ) χρησιμοποιείται, για να γίνουν
   οι σταθερές των αριθμών πιο ευανάγνωστες. */
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
-------------------------------------------------------------

val a: Int = 10000
println(a === a)                    // Εκτυπώνει: true
val boxedA: Int? = a
val anotherBoxedA: Int? = a
println(boxedA === anotherBoxedA)   // Εκτυπώνει: false
```

Ακολουθεί ο πλήρης κατάλογος των ενεργειών των δυαδικών ψηφίων (ο οποίος είναι διαθέσιμος μόνο για τύπους δεδομένων Int και Long):

- ✓ shl(bits) – signed shift left
- ✓ shr(bits) – signed shift right
- ✓ ushr(bits) – unsigned shift right
- ✓ and(bits) – bitwise and
- ✓ or(bits) – bitwise or
- ✓ xor(bits) – bitwise xor
- ✓ inv() – bitwise inversion

Code Example

```
// Παραδείγματα πράξεων με δυαδικούς αριθμούς.
val leftShift = 1 shl 2
val rightShift = 1 shr 2
val unsignedRightShift = 1 ushr 2
val and = 1 and 0x00001111
val or = 1 or 0x00001111
val xor = 1 xor 0x00001111
val inv = 1.inv()
```

Χαρακτήρες (Characters):

Οι χαρακτήρες αντιπροσωπεύονται από τον τύπο δεδομένων Char. Τα γράμματα των χαρακτήρων μπαίνουν σε μεμονωμένα εισαγωγικά (' '). Οι ειδικοί χαρακτήρες μπορούν να ξεφύγουν χρησιμοποιώντας μια πίσω κάθετο: \ t, \ b, \ n, \ r, \ ', \ ", \\ και \$. Για να κωδικοποιήσουμε οποιοδήποτε άλλο χαρακτήρα, χρησιμοποιούμε τη σύνταξη της ακολουθίας διαφυγής Unicode: ' \ uFF00 '.

Code Example

```
// Δήλωση ενός χαρακτήρα, ο οποίος είναι μη τροποποιήσιμος.
val value1 = 'A'
```

Code Examples

```
/* Εναλακτικός τρόπος δήλωσης ενός χαρακτήρα, ο οποίος είναι
   μη τροποποιήσιμος. */
val  value2: Char
value2= 'A'
-------------------------------------------------------------

fun main() {

    val letter: Char
    letter = 'k'

    // Εκτυπώνει: k
    println("$letter")
}
```

- *Οι χαρακτήρες δεν αντιμετωπίζονται σαν αριθμοί στη Kotlin.*

Λογικές (Booleans):

Ο τύπος δεδομένων Boolean αντιπροσωπεύει τις λογικές και έχει δύο τιμές: true και false.

Υποστηρίζουν τις συνήθεις λειτουργίες διάζευξης, σύζευξης και άρνησης:

- ✓ || - Διάζευξη
- ✓ && - Σύζευξη
- ✓ ! - Άρνηση

Code Example

```
val x = 1
val y = 2
val z = 2

/* Η σύζευξη και η διάζευξη αξιολογούνται χαλαρά. Εάν
   η αριστερή πλευρά ικανοποιεί τη ρήτρα, τότε η δεξιά
   πλευρά δεν θα αξιολογηθεί. */
val isTrue = x < y && x < z
val alsoTrue = x == y || y == z
```

Πίνακες (Arrays):

Οι Πίνακες αντιπροσωπεύονται από την κλάση Array, η οποία έχει τις συναρτήσεις get(), set() (που μετατρέπονται σε [ ] από τις συμβάσεις υπερφόρτωσης του χειριστή) και την ιδιότητα μεγέθους.

Για να δημιουργηθεί ένας πίνακας, χρησιμοποιείται μια συνάρτηση της βιβλιοθήκης: arrayOf(). Εναλλακτικά, η συνάρτηση της βιβλιοθήκης: arrayOfNulls() μπορεί να χρησιμοποιηθεί για να δημιουργήσει μια συστοιχία δεδομένου μεγέθους, η οποία είναι γεμάτη με μη μηδενικά στοιχεία.

Code Examples

```
/* Η συνάρτηση arrayOf δημιουργεί ένα πίνακα, ο οποίος δέχεται
   συμβολοσειρές (ονόματα). */
val friends = arrayOf("Sokrates", "Theodore", "Steve", "Helen")
----------------------------------------------------------------

/* Η συνάρτηση arrayOf δημιουργεί ένα πίνακα, ο οποίος δέχεται
   στοιχεία από διαφορετικούς τύπους δεδομένων. */
val info = arrayOf("Sokrates", 30, 7.11, "sokarvan@phys.uoa.gr")
----------------------------------------------------------------

/* Η συνάρτηση arrayOf δημιουργεί ένα πίνακα, ο οποίος δέχεται
   ακέραιους αριθμούς. */
val numbers = arrayOf(1, 3, 5, 7, 9)
----------------------------------------------------------------

/* Η συνάρτηση intArrayOf δημιουργεί ένα πίνακα, ο οποίος
   δέχεται ακέραιους αριθμούς. */
val oddNums = intArrayOf(1, 3, 5, 7, 9)
----------------------------------------------------------------

/* Η συνάρτηση arrayOf δημιουργεί ένα πίνακα, ο οποίος δέχεται
   στοιχεία από συμβολοσειρές για κάποιους χρήστες, όπως:
   ονοματεπώνυμο, τόπος διαμονής, αφμ και email. */
val users = arrayOf(
   User("Sok Arvanitis", "Athens", "027008090", "sa@phys.gr"),
   User("Mike Kainich", "Paris", "809006500", "mk@gmail.com"),
   User("John Kainich", "London", "809002700", "sk@gmail.com")
)
```

Code Examples

```
/* Η συνάρτηση arrayOf δημιουργεί ένα πίνακα, ο οποίος
   δέχεται στοιχεία από συμβολοσειρές για κάποιους
   χρήστες και αρχικοποιείται με μη μηδενικές τιμές. */
val serverUser: Array<User?> = arrayOf(null, null, null, null)
-----------------------------------------------------------------

/* Αρχικοποίηση ενός πίνακα με όνομα nullArray με μη μηδενικές
   τιμές, ο οποίος δέχεται στοιχεία συμβολοσειρών για εκατό
   χρήστες. */
val nullArray: Array<User?> = arrayOfNulls(100)
-----------------------------------------------------------------

/* Δημιουργία ενός πίνακα από συμβολοσειρές με τις τιμές:
   [0, 1, 4, 9, 16]. */
val asc = Array(5, { i -> (i * i).toString() })
asc.forEach { println(it) }
-----------------------------------------------------------------

/* Δημιουργία ένός πίνακα από συμβολοσειρές με τις τιμές:
   [0, 1, 4, 9, 16, 25, . . . 2500]. */
val squares = Array(51, {i -> i * i})
squares.forEach { println(it) }
```

- *Οι πίνακες είναι αμετάβλητοι στη Kotlin.*

Συμβολοσειρές (Strings):

Οι συμβολοσειρές αντιπροσωπεύονται από τον τύπο δεδομένων String και είναι αμετάβλητες. Αξιοσημείωτο είναι ότι μπορεί να προσθέθούν συμβολοσειρές χρησιμοποιώντας τον τελεστή της πρόσθεσης (+). Αυτό λειτουργεί για τη σύζευξη των συμβολοσειρών με τιμές άλλων τύπων δεδομένων, εφόσον το πρώτο στοιχείο της έκφρασης είναι μια συμβολοσειρά.

Code Example

```
val s = "abc" + 1
println(s + "def")
```

Αλφαριθμητικές Σταθερές (String Literals)

Οι αλφαριθμιτικές σταθερές μπορούν να δημιουργηθούν χρησιμοποιώντας διπλά (" ") ή τριπλά (" " ")εισαγωγικά. Τα διπλά εισαγωγικά μπορούν να περιέχουν σύμβολα διαφυγής. Ενώ τα τριπλά εισαγωγικά δημιουργούν μια ακατέργαστη συμβολοσειρά. Σε μια ακατέργαστη συμβολοσειρά, δεν υπάρχουν σύμβολα διαφυγής και όλοι οι χαρακτήρες μπορούν να συμπεριληφθούν.

Code Examples

```
// \n: Χρησιμοποιείται για αλλαγή γραμμής.
val s = "Hello World!\n"

val text = """
    for (c in "foo")
        print(c)
"""
-------------------------------------------------------------

fun main() {

    val myString = """
    for (character in "Hey!")
        println(character)
"""

    // Εκτυπώνει: for (character in "Hey!")
                     println(character)
    print(myString)
}
-------------------------------------------------------------

/* Η συνάρτηση trimMargin χρησιμοποιείται για την αφαίρεση
   των αρχικών κενών. */
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

Πρότυπα Συμβολοσειρών (String Templates):

Τα πρότυπα συμβολοσειρών μπορούν να περιέχουν εκφράσεις προτύπων, δηλαδή κομμάτια κώδικα που αξιολογούνται και τα αποτελέσματα αυτών συνδέονται στη συμβολοσειρά. Ένα πρότυπο συμβολοσειράς ξεκινά με ένα σύμβολο δολαρίου ($) και αποτελείται είτε από ένα απλό όνομα είτε με μια αυθαίρετη έκφραση σε άγκιστρα { }.

Code Examples

```
val i = 10
val s = "abc"

// Εκτυπώνει: i = 10
println("i = $i")

// Εκτυπώνει: abc.length is 3
println("$s.length is ${s.length}")
-----------------------------------------------------------------

/* Μια τιμή ή μια μεταβλητή μπορεί να είναι ενσωματωμένη απλά
   με το πρόθεμα του συμβόλου του δολαρίου $. */
val name = "Sok"
val str = "hello $name"

val name = " Sok"
val str = "hello $name. Your name has ${name.length}characters"
-----------------------------------------------------------------

fun main() {

    val myInt = 5;
    val myString = "myInt = $myInt"

    // Εκτυπώνει: myInt = 5
    println(myString)
}
```

- *Η χρήση προτύπων ή ακατέργαστων συμβολοσειρών είναι προτιμότερη από τη παράθεση συμβολοσειρών.*

1.3 Μεταβλητές (Variables)

Στη Kotlin υπάρχουν δύο λέξεις κλειδιά για την δήλωση μεταβλητών: var και val. Η διαφορά τους είναι ότι μία var μεταβλητή είναι τροποποιήσιμη οποιαδήποτε στιγμή κατά την εκτέλεση του προγράμματος, αφού μπορεί να ανατεθει . Ενώ μία val μεταβλητή δεν είναι τροποποιήσιμη και χρησιμοποιείται για να δηλώσει μια μεταβλητή μόνο για ανάγνωση. Αυτό ισοδυναμεί με τη δήλωση μιας τελικής μεταβλητής (final) στην Java.

Code Examples

```
// Άμεση ανάθεση της τιμής "initial" σε μία var μεταβλητή a.
var a: String = "initial"
println(a)
-----------------------------------------------------------------

// Άμεση ανάθεση της τιμής 1 σε μία val μεταβλητή b.
val b: Int = 1
-----------------------------------------------------------------

/* Δήλωση μιας val μεταβλητής c και αρχικοποίηση της. Ο
   μεταγλωττιστής συνάγει αυτόματα τον τύπο δεδομένων <Int>. */
val c = 3
-----------------------------------------------------------------

// Δήλωση μιας var μεταβλητής e, η οποία δεν αρχικοποιείται.
var e: Int
println(e)      // Σφάλμα, η μεταβλητή e δεν αρχικοποιείται.
-----------------------------------------------------------------

/* Η αρχικοποίηση των μεταβλητών μπορεί να αναβληθεί,
   αλλά πρέπει να αρχικοποιηθεί πριν από την πρώτη ανάγνωση. */

// Δήλωση μιας var μεταβλητής d, η οποία δεν αρχικοποιέιται.
val d: Int

if (someCondition()) {

    /* Αρχικοποίηση της val μεταβλητής d με διαφορετικές τιμές,
       ανάλογα με κάποια κατάσταση. */
    d = 1
} else {
    d = 2
}
println(d)                    // Η πρώτη χρήση της μεταβλητής d.
```

1.4 Null Safety

Σε μια προσπάθεια να απαλλαγούμε από τον κόσμο του Null Pointer Exception (NPE), οι μεταβλητοί τύποι δεδομένων στη Kotlin δεν επιτρέπουν τη null διαμόρφωση.

Code Examples

```
// Δήλωση μιας μη μηδενικής String μεταβλητής (neverNull).
var neverNull: String = "This can't be null"
neverNull = null                                    // Σφάλμα
------------------------------------------------------------

// Δήλωση μιας μηδενικής String μεταβλητής με όνομα nullable.
var nullable: String? = "You can keep a null here"

// Βάζει στην μεταβλητή nullable την τιμή null.
nullable = null
```

Μερικές φορές τα προγράμματα στη Kotlin πρέπει να δουλεύουν με μηδενικές τιμές (null), όπως όταν αλληλεπιδρούν με εξωτερικούς κώδικες Java ή όταν αντιπροσωπεύουν μία απούσα κατάσταση. Για την αντιμετώπιση τέτοιων καταστάσεων, η Kotlin παρέχει null tracking.

Code Example

```
/* Δήλωση μιας συνάρτησης η οποία παίρνει σαν παράμετρο μια
   συμβολοσειρά, έχει αρχικοποιηθεί με null και επιστρέφει
   μια φράση, που την περιγράφει. */
fun describeString(maybeString: String?): String {

    /* Εάν το String δεν είναι null και δεν είναι και άδειο,
       δίνει το  μήκος του. */
    if (maybeString != null && maybeString.length > 0) {
        return "String of length ${maybeString.length}"
    } else {
        return "Empty or null string"
    }
}
```

1.5 Σχόλια (Comments)

Οι περισσότερες γλώσσες προγραμματισμού, όπως και η Kotlin υποστηρίζουν τα σχόλια. Είναι πολύ σημαντικό να χρησιμοποιούνται τα σχόλια για να βελτιώνεται η αναγνωσιμότητα του πηγαίου κώδικα. Τα σχόλια μπορεί να είναι μέρος του προγράμματος που προορίζεται για εμάς ή για τους συναδέλφους προγραμματιστές μας, για να κατανοήσουν τον κώδικα και αγνοούνται εντελώς απο τον μεταγλωττιστή. Υπάρχουν δύο τύποι σχολίων που υποστηρίζονται από τον Kotlin: Σχόλιο μιας γραμμής και Σχόλιο πολλών γραμμών.

Code Examples

```
// Hello Kotlin                    -> Σχόλιο μιας γραμμής
------------------------------------------------------------


/* Hello Kotlin.
   How are you today? */           -> Σχόλιο πολλών γραμμών
```

1.6 Έλεγχοι Ισότητας (Equality Checks)

Η Kotlin χρησιμοποιεί τα δύο ίσον (==) για δομική σύγκριση και τα τρία ίσον (===) για αναφορική σύγκριση.

Code Example

```
val authors = setOf("Shakespeare", "Hemingway", "Twain")
val writers = setOf("Twain", "Shakespeare", "Hemingway")

// Επιστρέφει: true
println(authors == writers)

/* Επιστρέφει: false, επειδή οι σταθερές: authors και writers
   είναι ξεχωριστές αναφορές. */
println(authors === writers)
```

1.7 Ροές Ελέγχου (Control Flows)

Οι συνθήκες των ροών ελέγχου: if...else και when είναι εκφράσεις. Αυτό σημαίνει ότι το αποτέλεσμα μπορεί να εκχωρηθεί απευθείας σε μια τιμή, η οποία είτε επιστρέφεται από μια συνάρτηση είτε μεταβιβάζεται ως ένα όρισμα σε μια άλλη συνάρτηση.

Η Συνθήκη if…else:

Η συνθήκη if…else χρησιμοποιείται για να ελέγξουμε εάν ικανοποιείται μία συνθήκη και επιστρέφει μια τιμή.

Code Examples

```
// Παραδοσιακή χρήση της συνθήκης if:
var max = a
if (a < b) max = b
-----------------------------------------------------------------

// Παραδοσιακή χρήση της συνθήκης if…else:
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}
-----------------------------------------------------------------

/* Η συνθήκη if…else σαν μία έκφραση, η οποία επιστρέφει \
   μία τιμή. */
fun max(a: Int, b: Int) = if (a > b) a else b

println(max(99, -42))
-----------------------------------------------------------------

val date = Date()
val today = if (date.year == 2019) true else false

fun isZero(x: Int): Boolean {
    return if (x == 0) true else false
}
```

Code Example

```
/* Οι εκφράσεις δεν χρειάζεται να είναι μεμονωμένες γραμμές.
   Μπορούν να είναι μέσα σε μπλοκ εντολών και η τελευταία
   γραμμή να είναι μια έκφραση. Αυτή η έκφραση είναι η
   τιμή που αξιολογεί το μπλοκ των εντολών. */
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

- *Όταν χρησιμοποιείτε η συνθήκη if...else, πρέπει να συμπεριλαμβάνεται και το else. Διαφορετικά ο μεταγλωττιστής θα εμφανίσει ένα σφάλμα, γιατί δεν θα ξέρει τι να κάνει εάν το if δεν αξιολογήθηκε ως αληθές (true).*

Η Συνθήκη when:

Η συνθήκη when πορεί να χρησιμοποιηθεί είτε ως έκφραση είτε ως δήλωση. Εάν χρησιμοποιείται ως έκφραση, η τιμή του ικανοποιημένου κλάδου γίνεται η τιμή της συνολικής έκφρασης. Ενώ εάν χρησιμοποιείται ως δήλωση, οι τιμές των μεμονωμένων κλάδων αγνοούνται.

Code Example

```
when (x) {

    // Ελέγχει εάν το x είναι ίσο με 1.
    1 -> print("x == 1")

    // Ελέγχει εάν το x είναι ίσο με 2.
    2 -> print("x == 2")

    // Ελέγχει εάν το x είναι ίσο με 3.
    3 -> print("x == 3")

    // Προκαθορισμένη δήλωση (DS)
    else -> print("x is neither 1 nor 2 nor 3")
}
```

Code Examples

```
/* Εάν πολλές περιπτώσεις πρέπει να αντιμετωπιστούν με τον
   ίδιο τρόπο, οι συνθήκες κλάδου μπορούν να συνδυαστούν με
   κόμμα(,) */
when (x) {
     0, 1 -> print("x == 0 or x == 1")
     else -> print("x is neither 1 nor 2")
}
-------------------------------------------------------------

when (x) {

    // Ελέγχει εάν η τιμή του x είναι στο διάστημα 1..10
    in 1..10 -> print("x is in the range")

    /* Ελέγχει εάν η τιμή του x είναι μέσα στη συλλογή
       validNumbers. */
    in validNumbers -> print("x is valid")

    // Ελέγχει εάν η τιμή του x δεν είναι στο διάστημα 10..20
    !in 10..20 -> print("x is outside of the range")
    else -> print("none of the above")
}
-------------------------------------------------------------

fun main() {

    val a = 12
    val b = 5

    // Εκτυπώνει: Enter operator either +, -, * or /
    println("Enter operator either +, -, * or /")
    val operator = readLine()

    when (operator) {
        "+" -> println("$a + $b = ${a + b}")
        "-" -> println("$a - $b = ${a - b}")
        "*" -> println("$a * $b = ${a * b}")
        "/" -> println("$a / $b = ${a / b}")
        else -> println("$operator is invalid")
    }
}
```

- *Εάν ο μεταγλωττιστής μπορεί να συμπεράνει ότι όλες οι πιθανές συνθήκες έχουν ικανοποιηθεί, τότε η προκαθορισμένη δήλωση (else ->) μπορεί να παραλειφθεί.*

1.8 Διαχείριση Εξαιρέσεων με τη Συνθήκη try…catch

Η συνθήκη try…catch χρησιμοποιείται για τον χειρισμό εξαιρέσεων στον κώδικα. Το μπλοκ των εντολών περικλείει τον κώδικα, ο οποίος μπορεί να κάνει μια εξαίρεση και η συνθήκη try…catch χρησιμοποιείται για την αντιμετώπιση αυτής της εξαίρεσης. Το μπλοκ των εντολών πρέπει να γράφεται μέσα στη μέθοδο.

Code Examples

```
fun main() {
val str = getNumber("10")
    println(str)                    // Εκτυπώνει: 10
}

fun getNumber(str: String): Int {

    /* Η συνθήκη try…catch είναι σαν μία έκφραση, η οποία
       επιστρέφει μία τιμή. */
    return try {
        Integer.parseInt(str)
    } catch (e: ArithmeticException) {
        0
    }
}
-----------------------------------------------------------------

fun main() {
val str = getNumber("10.5")
    println(str)                    // Εκτυπώνει: 0
}

fun getNumber(str: String): Int {

    /* Η συνθήκη try…catch επιστρέφει την τελευταία δήλωση
       από το μπλοκ εντολών */.
    return try {
        Integer.parseInt(str)
    } catch (e: NumberFormatException) {
        0
    }
}
```

1.9 Βρόχοι Επανάληψης (Loops)

Η Kotlin υποστηρίζει τρεις τρόπους για επαναληπτικούς βρόχους, οι οποίοι βρίσκονται στις περισσότερες γλώσσες προγραμματισμού: for, while και do…while.

Ο Βρόχος Επανάληψης for (for Loop):

Ο βρόχος επανάληψης for χρησιμοποιείται για την επανάληψη οποιουδήποτε αντικειμένου που καθορίζει μια συνάρτηση ή μια επέκταση συνάρτησης με το όνομα του στιγμιοτύπου.

Code Examples

```
/* Για να επαναλάβουμε έναν αριθμό, μπορούμε να χρησιμοποιούμε
   έναν πίνακα. */
for (i in array.indices) {
    println(array[i])
}
----------------------------------------------------------------

/* Για την επανάληψη ενός αριθμού, μπορεί να χρησιμοποιηθεί
   μία λίστα. */
val list = listOf(1, 2, 3, 4)
for (k in list) {
    println(k)
}
----------------------------------------------------------------

val cakes = listOf("carrot", "cheese", "chocolate")

// Επαναληπτικός βρόχος for για κάθε cake μέσα στη λίστα cakes.
for (cake in cakes) {
    println("Yummy, it's a $cake cake!")
}
----------------------------------------------------------------

// Επαναληπτικός βρόχος for δύο διαστάσεων: k, j.
val oneToTen = 1..10
for (k in oneToTen) {
    for (j in 1..5) {
        println(k * j)
    }
}
```

Ο Βρόχος Επανάληψης while (while Loop):

Ο βρόχος επανάληψης while χρησιμοποιείται για την επανάληψη ενός μέρους του προγράμματος αρκετές φορές. Έκτελεί το μπλοκ του κώδικα έως ότου η κατάσταση γίνει αληθής (true).

Code Examples

```
fun eatACake() = println("Eat a Cake!")

fun main() {
    var cakesEaten = 0

    /* Εκτελεί το βρόχο επανάληψης ενώ η συνθήκη while είναι
       λανθασμένη. */
    while (cakesEaten < 5) {
        eatACake()
        cakesEaten ++
    }
}
-----------------------------------------------------------------

fun main() {
    var i = 1
    while (i <= 5) {
        println(i)           /* Εκτυπώνει: 1
        i++                                 2
    }                                       3
}                                           4
                                            5 */
-----------------------------------------------------------------

/* Πρόγραμμα το οποίο υπολογίζει το σύνολο των φυσικών αριθμών
   από το 1 έως το 100. */
fun main() {
    var sum = 0
    var i = 100

    while (i != 0) {
        sum += i           // Ισοδύμανο με: sum = sum + i
        --i
    }
    println("Sum = $sum")
}
```

Ο Βρόχος Επανάληψης do…while (do…while Loop):

Ο βρόχος επανάληψης do...while είναι παρόμοιος με τον βρόχο επανάληψης while εκτός από μία βασική διαφορά. Ο βρόχος do...while εκτελεί πρώτα το μπλοκ των εντολών και μετά ελέγχει τη συνθήκη. Οπότε το μπλοκ των εντολών εκτελείται τουλάχιστον μία φορά ακόμα και εάν η συνθήκη είναι ψευδής.

Code Examples

```
fun bakeACake() = println("Bake a Cake!")

fun main() {
    var cakesBaked = 0

    /* Εκτελεί πρώτα το μπλοκ των εντολών και μετά αξιολογεί
       τη συνθήκη while. */
    do {
        bakeACake()
        cakesBaked++
    } while (cakesBaked < 5)

}
```

```
fun main() {
    var i = 1
    do {
        println(i)                          /* Εκτυπώνει: 1
        i++                                                2
    }                                                      3
    while (i <= 5);                                        4
}                                                          5 */
```

```
fun main() {
    var i = 1
    do {
        println("5 * $i = " + (5 * i))   /* Εκτυπώνει: 5*1=5
        i++                                            5*2=10
    } while (i <= 5)                                   5*3=15
}                                                      5*4=20
                                                       5*5=25 */
```

1.10 Εύρη Τιμών (Ranges)

Τα εύρη τιμών σχηματίζονται με την συνάρτηση RangeTo() και συμπληρώνονται από τα in και !in. Τα εύρη τιμών ορίζονται για κάθε συγκρίσιμο τύπο δεδομένων.

Code Examples

```
// Επαναληπτικός βρόχος για ένα εύρος τιμών από το 0 έως και το 3.
for (i in 0..3) {
    print(i)
}
----------------------------------------------------------------

/* Επαναληπτικός βρόχος για ένα εύρος τιμών από το 2 έως και το 8,
   που χρησιμοποιεί 2 βήματα αύξησης (step 2) για διαδοχικά
   στοιχεία. */
for(i in 2..8 step 2) {
    print(i)
}
----------------------------------------------------------------

// Επαναληπτικός βρόχος για ένα εύρος τιμών από το a έως και το d.
for (c in 'a'..'d') {
    print(c)
}
----------------------------------------------------------------

/* Επαναληπτικός βρόχος για ένα εύρος τιμών από το a έως και το g,
   που χρησιμοποιεί 2 βήματα αύξησης για διαδοχικά στοιχεία. */
for (c in 'a'..'g' step 2) {
    print(c)
}
----------------------------------------------------------------

/* Επαναληπτικός βρόχος για ένα εύρος τιμών από το 3 έως και το 0,
   χρησιμοποιώντας το downTo για αντίστροφη σειρά. */
for (i in 3 downTo 0) {
    print(i)
}
```

- *Τα εύρη τιμών (ranges) αντιμετωπίζονται με ειδικό τρόπο από τον μεταγλωττιστή και καταρτίζονται σε δείκτες για βρόχους επανάληψης που υποστηρίζονται απευθείας από το JVM (Java Virtual Machine).*

2. ΣΥΝΑΡΤΗΣΕΙΣ (FUNCTIONS)

2.1 Ορισμός Συναρτήσεων (Define Functions)

Καθώς μεγαλώνει ένα πρόγραμμα, η πολυπλοκότητα αυξάνεται. Εάν ο κώδικας δεν μπορεί να χειριστεί αυτήν την ανάπτυξη, είναι εύκολο να κατακλυστεί η πολυπλοκότητα της εφαρμογής. Ο καλύτερος τρόπος για να διαχειριστούμε τον κώδικα μας, είναι να το χωρίσουμε σε μικρά, αυτόνομα τμήματα και το περίπλοκο πρόβλημα μπορεί να λυθεί με την τοποθέτηση αυτών των τμημάτων. Η Kotlin μπορεί να μας βοηθήσει να διαιρέσουμε τον κώδικα μας σε μικρά κομμάτια. Εάν χρησιμοποιούνται σωστά, μας βοηθούν να διατηρούμε τον κώδικα καθαρό, οργανωμένο και εύκολο να τον κατανοήσουμε. Σε διαφορετικές γλώσσες προγραμματισμού, αυτή η τεχνική ονομάζεται μέθοδος, υπορουτίνα ή διαδικασία. Στη Kotlin, αυτή η τεχνική ονομάζεται συνάρτηση.

Για την δήλωση μιας συνάρτησης χρησιμοποιείται την λέξη fun, η οποία ακολουθείται από το όνομα της συνάρτησης με παρενθέσεις (). Το μπλοκ του κώδικα ορίζεται μέσα στις αγκύλες { }.

Code Example

```
/* Δήλωση μιας συνάρτησης με όνομα hello, η οποία δεν έχει
   παραμέτρους και εκτυπώνει το μύνημα: Hello from Kotlin! */
fun hello() {

    println("Hello from Kotlin!")
}

/* Δήλωση της συνάρτησης main, η οποία καλέι την συνάρτηση
   hello(). */
fun main() {

    hello()
}
```

Συναρτήσεις χωρίς Παραμέτρους και χωρίς Τύπο Επιστροφής:

Εάν μια συνάρτηση δεν έχει καμία τιμή επιστροφής, μπορεί να δηλωθεί ως Unit (τοποθετείται μετά το όνομα της συνάρτησης). Η λέξη Unit είναι προαιρετική. Εάν δεν αναφέρεται τίποτα, θα θεωρήθει η μονάδα ως προεπιλεγμένη τιμή.

Code Examples

```
/* Δήλωση μιας συνάρτησης που δεν παίρνει καμία παράμετρο
   και δεν έχει καμία τιμή επιστροφής. */
fun sayHello() {
    println("Hello from Kotlin!")
}
-------------------------------------------------------------

/* Δήλωση μιας συνάρτησης που δεν παίρνει καμία παράμετρο
   και επιστρέφει Unit (καμία τιμή επιστροφής). */
fun sayHello(): Unit { println("Hello from Kotlin!") }
```

Συναρτήσεις με Παραμέτρους και χωρίς Τύπο Επιστροφής:

Εάν η συνάρτηση δέχεται περισσότερες από μία παραμέτρους, τότε όλες οι παράμετροι χωρίζονται με κόμμα (,).

Code Examples

```
/* Δήλωση μιας συνάρτησης που παίρνει σαν παράμετρο μία
   μεταβλητή τύπου String και επιστρέφει Unit. */
fun hello(message: String) : Unit {
    println("Hello from $message")
}

fun main() { hello("Kotlin!") }
-------------------------------------------------------------

/* Δήλωση μιας συνάρτησης που παίρνει σαν παράμετρους δύο
   μεταβλητές τύπου Int και δεν έχει καμία τιμή επιστροφής. */
fun add(a: Int, b: Int) {
    println("Result of $a + $b is ${a+b}")
}

fun main(){ add(4,5) }
```

Συναρτήσεις με Παραμέτρους και Τύπο Επιστροφής:

Οι συναρτήσεις μπορούν να λαμβάνουν παραμέτρους και να επιστρέφουν κάποια τιμή ως αποτέλεσμα. Η Kotlin, όπως και άλλες γλώσσες προγραμματισμού χρησιμοποιούν τη λέξη return για να επιστρέψουν κάποια τιμή από μία συνάρτηση. Η τιμή που επιστρέφεται πρέπει να είναι ίδια με τον τύπο επιστροφής που ορίζεται στην υπογραφή της συνάρτησης.

Code Examples

```
/* Δήλωση μιας συνάρτησης που παίρνει σαν παράμετρο μία
   μεταβλητή τύπου String και επιστρέφει μία συμβολοσειρά. */
fun myFun(message: String): String {
    return "Hello from $message"
}

fun main() {
    val result = myFun("Author")
    println(result)
}
------------------------------------------------------------

/* Δήλωση μιας συνάρτησης που πέρνει σαν παράμετρους δύο
   ακέραιους αριθμούς και επιστρέφει έναν ακέραιο αριθμό. */
fun sum(x: Int, y: Int): Int {
    return x + y
}
```

Συναρτήσεις σαν μία Έκφραση:

Μια συνάρτηση μπορεί να συμπεριφέρεται ως μία έκφραση. Αξιοσημείωτο είναι ότι όταν μία συνάρτηση περιέχει μόνο μία γραμμή κώδικα, μπορεί να γραφεί ως μία έκφραση.

Code Examples

```
/* Δήλωση μιας συνάρτησης που συμπεριφέρεται σαν μία έκφραση
   και επιστρέφει έναν ακέραιο αριθμό. */
fun addValuesEx(a: Int, b: Int): Int = a + b
```

Συναρτήσεις με Προεπιλεγμένα Ορίσματα:

Εάν μία συνάρτηση ενεργοποιείται χωρίς να περάσει μια τιμή, τότε ο μεταγλωττιστής εκχωρεί αυτόματα μια προκαθορισμένη τιμή. Σε πολλές περιπτώσεις, το προεπιλεγμένο όρισμα (DA) είναι ένα πολύ χρήσιμο χαρακτηριστικό.

Code Example

```
/* Δήλωση μιας συνάρτησης, η οποία εκτυπώνει το μήνυμα:
   Hello World , εάν δεν περαστεί κάποια τιμή στη συνάρτηση. */
fun hello(message: String = "Kotlin"): Unit {
     println("Hello $message")
}

fun main() {
    hello()
    hello("World")
}
```

Συναρτήσεις με Ονομαστικές Παραμέτρους:

Η Kotlin καθιστά δυνατό τον προσδιορισμό του ονόματος του ορίσματος σε μια κλήση συνάρτησης. Αυτή η προσέγγιση κάνει την κλήση της συνάρτησης πιο ευανάγνωστη και μειώνει την πιθανότητα να περαστεί λάθος τιμή στη μεταβλητή, ειδικά όταν όλες οι μεταβλητές έχουν τον ίδιο τύπο δεδομένων.

Code Example

```
/* Δήλωση μιας συνάρτησης, η οποία παίρνει σαν παραμέτρους
   τρεις μεταβλητές τύπου Double και επιστρέφει ένα δεκαδικό
   αριθμό. */
fun ex (dollar: Double, currencyRate: Double,
   charges: Double = 5.0): Double {

     var total = dollar * currencyRate
     var fees = total * charges / 100
     total = total - fee
     return total
}
```

Συναρτήσεις και Μεταβλητά Ορίσματα (Varargs):

Όταν καλούμε μια συνάρτηση, μπορούμε να περάσουμε τα ορίσματα ένα προς ένα, τα οποία μετατρέπονται αυτόματα σε πίνακα. Αυτό ονομάζεται vararg (ένα μεταβλητό όρισμα).

Code Example

```
/* Δήλωση μιας συνάρτησης, η οποία παίρνει σαν παράμετρο μία
   vararg συμβολοσειρά και εκτυπώνει τα αντικείμενα (item)
   μέσα στη λίστα (list). */
fun varargString(vararg list: String) {
    for (item in list) {
        println(item)
    }
}

fun main() {

    /* Εκτυπώνει: Monday
                  Tuesday
                  Wednesday */
    varargString("Monday", "Tuesday", "Wednesday")

    /* Εκτυπώνει: Thursday
                  Friday
                  Saturday
                  Sunday */
    varargString("Thursday", "Friday", "Saturday", "Sunday")
}
-------------------------------------------------------------

fun printNames(vararg names : String) {
    println("Name = ${names.get(0)}")
    println("Name = ${names.get(1)}")
    println("Name = ${names.get(2)}")
}

fun main() {

    /* Εκτυπώνει: Name = Kale
                  Name = Kate
                  Name = Ashley */
    printNames("Kale", "Kate", "Ashley")
}
```

2.2 Εν Σειρά Συναρτήσεις (Inline Functions)

Οι εν σειρά συνάρτησεις δηλώνονται με τη λέξη inline. Η χρήση εν σειράς συναρτήσεων ενισχύει την απόδοση των συναρτήσεων υψηλότερης τάξης. Η λέξη inline υποδεικνύει στον μεταγλωττιστή ότι η συνάρτηση που έχει επισημανθεί ως εν σειρά καθώς και οι παράμετροι της, θα πρέπει να διευρυνθούν και να δημιουργηθούν εν σειρά στην περιοχή κλήσεων, εξού και το όνομα της.

Code Example

```
inline fun inlineFunction (myFun:()-> Unit,nxtFun:()-> Unit) {
    myFun()
    nxtFun()
    print("Code inside inline function")
}

fun main() {

   // Εκτυπώνει: Calling inline functions
   inlineFunction({ println("Calling inline functions")
      return},{ println("Next parameter in inline functions")})
}
-----------------------------------------------------------------

fun sampleFunction(str: String, expression: (String) -> Unit) {
    println("This is Kotlin Inline Functions")
    expression(str)
}

fun main() {

    // Εκτυπώνει: Hey how are you doing
    println("Hey how are you doing")

    /* Εκτυπώνει: This is Kotlin Inline Functions
                  Functional Kotlin */
    sampleFunction("Functional Kotlin", ::println)
}
```

2.3 Λάμδα Συναρτήσεις (Lamda Functions)

Οι Λάμδα Συναρτήσεις είναι ένας απλός τρόπος για να δημιουργήσουμε Ad-Hoc συναρτήσεις. Είναι συναρτήσεις που δεν έχουν όνομα. Το Λάμδα ορίζεται μέσα σε αγγείλες { }, οι οποίες παίρνουν τη μεταβλητή ως μια παράμετρο (εάν υπάρχει) και το σώμα της συνάρτησης. Το σώμα της συνάρτησης γράφεται μετά από τη μεταβλητή (εάν υπάρχει) ακολουθούμενη από τον τελεστή **->**.

Code Examples

```
/* Το Λάμδα είναι το κομμάτι που βρίσκεται μέσα στις αγγείλες,
   το οποίο αποδίδεται σε μια μεταβλητή τύπου String. */
val upperCase1: (String)-> String = { str: String ->
                                          str.toUpperCase() }

// Εκτυπώνει: HELLO
println(upperCase1("hello"))
-----------------------------------------------------------------

/* Εισαγωγή τύπου στο εσωτερικό του λάμδα. Εισάγει τον τύπο
   της παραμέτρου λάμδα από τον τύπο της μεταβλητής που
   έχει αντιστοιχιστεί.*/
val upperCase2: (String) -> String = { str-> str.toUpperCase() }

// Εκτυπώνει: HELLO
println(upperCase2("hello"))
-----------------------------------------------------------------

/* Εισαγωγή τύπου στο εξωτερικό του λάμδα. Συνάγει τον τύπο
   της μεταβλητής βάσει του τύπο της παραμέτρου λάμδα και την
   τιμή επιστροφής. */
val upperCase3 = { str: String -> str.toUpperCase() }

// Εκτυπώνει: HELLO
println(upperCase3("hello"))
-----------------------------------------------------------------

/* Δεν μπορούν να γίνουν και τα δύο μαζί, γιατί ο μεταγλωττιστής
   δεν μπορεί να συμπεράνει τον τύπο με αυτόν τον τρόπο. */
val upperCase4 = { str -> str.toUpperCase() }
```

Επίσης μπορεί να υπάρχουν συναρτήσεις σαν εκφράσεις δημιουργώντας λάμδα. Οι λάμδα είναι συναρτήσεις δηλαδή, δεν δηλώνονται καθώς είναι εκφράσεις και μπορούν να περάστουν ως παράμετροι. Ωστόσο, δεν μπορούμε να δηλώσουμε τύπους επιστροφής σε λάμδα. Ο τύπος επιστροφής συνάγεται αυτόματα από τον μεταγλωττιστή στις περισσότερες περιπτώσεις. Ενώ για περιπτώσεις όπου δεν μπορεί να συναχθεί από μόνος του, χρησιμοποιούμε ανώνυμες λειτουργίες.

Code Examples

```
/* Όταν υπάρχει Λάμδα με μία μόνο παράμετρο, δεν χρειάζεται
   να ονομαστεί ρητά. Αντ 'αυτού, μπορεί να χρησιμοποιηθεί
   η σιωπηρή μεταβλητή it. */
val upperCase5: (String) -> String = { it.toUpperCase() }

// Εκτυπώνει: HELLO
println(upperCase5("hello"))
----------------------------------------------------------------

/* Όταν υπάρχει Λάμδα με μία κλήση συνάρτησης, μπορούν να
   χρησιμοποιηθούν οι δείκτες συναρτήσεων. * /
val upperCase6: (String) -> String = String::toUpperCase

// Εκτυπώνει: HELLO
println(upperCase6("hello"))
----------------------------------------------------------------

fun addNumber(a: Int, b: Int, mylambda: (Int) -> Unit ){
    val add = a + b
    mylambda(add)
}

fun main(args: Array<String>){

    val myLambda: (Int) -> Unit= {s: Int -> println(s)}

    // Εκτυπώνει: 15
    addNumber(5, 10, myLambda)
}
```

2.4 Συναρτήσεις υψηλότερης τάξης (High Order Functions)

Η πιο θεμελιώδης έννοια του συναρτησιακού προγραμματισμού είναι οι συναρτήσεις πρώτης τάξης. Μια γλώσσα προγραμματισμού με υποστήριξη συναρτήσεων πρώτης τάξης, θα επεξεργάζεται συναρτήσεις όπως οποιοσδήποτε άλλος τύπος. Αυτές οι γλώσσες θα μας επιτρέψουν να χρησιμοποιήσουμε συναρτήσεις ως μεταβλητές, παραμέτρους, τύπους επιστροφής, τύπους γενίκευσης κ.ο.κ. Μια συνάρτηση υψηλότερης τάξης είναι απλώς μια συνάρτηση που είτε δέχεται μία συνάρτηση ως παράμετρο είτε επιστρέφει μια συνάρτηση ή και τα δύο.

Code Example

```
/* Δήλωση μιας συνάρτηση υψηλότερης τάξης που παίρνει σαν
   παράμετρους: δύο ακέραιες μεταβλητές: x και y και μια
   άλλη συνάρτηση με όνομα operation (η οποία παίρνει σαν
   παραμέτρους δύο μεταβλητές τύπου Int και επιστρέφει
   έναν ακέραιο αριθμό. */
fun calculate(x: Int, y: Int, operation: (Int, Int) -> Int):
Int {

    /* Κλήση της συνάρτησης operation περνώντας τα παρεχόμενα
       ορίσματα. */
    return operation(x, y)
}

/* Δήλωση μιας συνάρτησης που συμπεριφέρεται σαν μία έκφραση
   και επιστρέφει έναν ακέραιο αριθμό. */
fun sum(x: Int, y: Int) = x + y

fun main() {

 /* Κλήση της συνάρτησης υψολότερης τάξης calculate, περνώντας
    ως ορίσματα συνάρτησης, το ::sum που υποδηλώνει τον τρόπο
    με τον οποίον καλούμε μια συνάρτηση με το όνομα της. */
    val sumResult = calculate(4, 5, ::sum)

    /* Κλήση της συνάρτησης υψολότερης τάξης calculate,
       περνώντας ως ορίσματα συνάρτησης μία λάμδα συνάρτηση. */
    val mulResult = calculate(4, 5) { a, b -> a * b }

    // Εκτυπώνει: sumResult 9, mulResult 20
    println("sumResult $sumResult, mulResult $mulResult")
}
```

Code Examples

```
/* Δήλωση μιας συνάρτησης υψηλότερης τάξης με όνομα operate,
   που δέχεται δύο ακέραιους παράμετρους τύπου και επιστρέφει
   μια συνάρτηση. */
fun operate (v1: Int, v2: Int, fn: (Int, Int) -> Int): Int {

    return fn (v1, v2)
}

fun sum (x1: Int, x2: Int) = x1 + x2

fun sub (x1: Int, x2: Int) = x1 - x2

fun multiply (x1: Int, x2: Int) = x1 * x2

fun divide (x1: Int, x2: Int) = x1 / x2

fun main () {

    val resu1t1 = operate (10, 5, ::sum)

    // Εκτυπώνει: Sum(10,5)= 15
    println("Sum(10,5)= $resu1t1")

    val result2 = operate (5, 2, ::sum)

    // Εκτυπώνει: Sum(5, 2)= 7
    println("Sum(5,2)= $result2")

    // Εκτυπώνει: Subtraction(100,40)= 60
    println("Subtraction(100,40)= ${operate(100, 40, :: sub)}")

    // Εκτυπώνει: Multiplexion(5,20)= 100
    println("Multiplexion(5,20)= ${operate(5, 20,:: multiply)}")

    // Εκτυπώνει: Division(10, 5)= 2
    println("Division(10, 5)= ${operate (10, 5, :: divide)}")
}
```

2.5 Επεκτάσεις (Extensions)

Η Kotlin παρέχει τη δυνατότητα να επεκτείνουμε μια κλάση με νέες συναρτήσεις χωρίς να χρειαστεί να κληρονομήσει από την κλάση ή να χρησιμοποιήσουμε οποιοδήποτε τύπο σχεδίου. Αυτό γίνεται μέσω ειδικών δηλώσεων που ονομάζονται επεκτάσεις. Η Kotlin υποστηρίζει συναρτήσεις και ιδιότητες επέκτασης.

Code Examples

```
/* Δήλωση μιας κλάσης δεδομένων με όνομα Item, η οποία παίρνει
   σαν παράμετρους δύο σταθερές: μία τύπου String και μία τύπου
   Float. */
data class Item(val name: String, val price: Float)

/* Δήλωση μιας κλάσης δεδομένων με όνομα Order που παίρνει σαν
   παράμετρο μία αυθαίρετη συλλογή των Item. */
data class Order(val items: Collection<Item>)

/* Δήλωση δύο συναρτήσεων που επεκτείνονται για κάθε τύπο
   παραγγελίας. */
fun Order.maxPricedItemValue(): Float =
    his.items.maxBy{it.price}?.price?:0F

fun Order.maxPricedItemName()=
    this.items.maxBy{it.price}?.name?:"NO_PRODUCTS"

// Δήλωση ιδιότητας επέκτασης.
val Order.commaDelimitedItemNames: String
    get() = items.map { it.name }.joinToString()


fun main() {

val order=Order(listOf(Item("Bread",25.0F),Item("Wine",29.0F),
                                          Item("Water",12.0F)))

    // Εκτυπώνει: Max priced name: Wine
    println("Max priced name: ${order.maxPricedItemName()}")

    // Εκτυπώνει: Max priced value: 29.0
    println("Max priced value: ${order.maxPricedItemValue()}")

    // Εκτυπώνει: Items: Bread, Wine, Water
    println("Items: ${order.commaDelimitedItemNames}") }
```

2.6 Γενικές Συναρτήσεις (Generic Functions)

Μία συνάρτηση μπορεί να λειτουργεί για έναν συγκεκριμένο τύπο δεδομένων και στη συνέχεια να χρησιμοποιηθεί για έναν άλλο τύπο δεδομένων. Αυτό επιτρέπει τη σύνταξη γενικών συναρτήσεων. Με άλλα λόγια οι γενικές συναρτήσεις μπορούν να λειτουργήσουν με οποιονδήποτε τύπο δεδομένων και όχι με κάποιο συγκεκριμένο. Οι γενικές συναρτήσεις δηλώνοται με το πρόθεμα <T>. Για να γίνει αυτό, καθορίζουμε τους τύπους των παραμέτρων στην υπογραφή λειτουργίας.

Code Example

```
fun main(){

    val stringList: ArrayList<String> = arrayListOf<String>
                                        ("Sokratis", "Theodore")
    val s: String = stringList[0]

    // Εκτυπώνει: Sokratis
    println("$s")

    /* Εκτυπώνει: Sokratis
                  Theodore */
    printValue(stringList)
    val floatList: ArrayList<Float> = arrayListOf<Float>
                                      (10.5f, 5.0f, 25.5f)

    /* Εκτυπώνει: 10.5
                  5.0
                  25.5 */
    printValue(floatList)
}

// Δήλωση μιας γενικής συνάρτησης με όνομα printValue.
fun <T> printValue(list: ArrayList<T>){

    for (element in list){
       println(element)
    }
}
```

- *Ο μεταγλωττιστής μπορεί να συμπεράνει τον γενικό τύπο από τις παραμέτρους της συνάρτησης printValue.*

2.7 Τυπικές συναρτήσεις (Standard functions)

Η Kotlin παρέχει μια τυπική βιβλιοθήκη που προορίζεται να αυξήσει και όχι να αντικαταστήσει την τυπική βιβλιοθήκη της Java. Υπάρχουν πολλές συναρτήσεις που προσαρμόζουν τους τύπους και τις μεθόδους της Java και μπορούν να χρησιμοποιηθούν ως ιδιωματικό της Kotlin. Οι τυπικές συναρτήσεις είναι γενικές συναρτήσεις χρησιμότητας στη τυπική βιβλιοθήκη της Kotlin που δέχονται λάμδα για να καθορίσουν την εργασία τους. Παρακάτω δίδονται παραδείγματα μερικών τυπικών συναρτήσεων.

Apply:

Η συνάρτηση apply μπορεί να θεωρηθεί ως μια συνάρτηση διαμόρφωσης, αφού μας επιτρέπει να καλέσουμε μια σειρά από συναρτήσεις σε ένα δέκτη, ο οποίος διαρμοφώνεται για χρήση. Μπορεί να χρησιμοποιηθεί για τη μείωση της ποσότητας επανάληψης κατά τη διαμόρφωση ενός αντικειμένου προς χρήση. Επίσης μας επιτρέπει να αφήσουμε το όνομα της μεταβλητής από κάθε κλήση συνάρτησης που εκτελείται για τη διαμόρφωση του δέκτη.

Code Examples

```
/* Διαμόρφωσης ενός στιγμιότυπου αρχείου χωρίς την συνάρτηση
   apply. */
val menuFile = File("menu-file.txt")
    menuFile.setReadable(true)
    menuFile.setWritable(true)
    menuFile.setExecutable(false)
----------------------------------------------------------------

/* Με τη συνάρτηση apply η ίδια διαμόρφωση μπορεί να επιτευχθεί
   με λιγότερη επανάληψη. */
val menuFile = File("menu-file.txt").apply {
    setReadable(true)
    setWritable(true)
    setExecutable(false)
}
```

Let:

Η εφαρμογή της συνάρτησης let είναι παρόμοια με την εφαρμογή της συνάρτησης apply, με τη βασική διαφορά ότι μπορεί να επιστρέψει την τιμή του ίδιου του κλεισίματος. Μπορεί να χρησιμοποιηθεί για σκοπούς οριοθέτησης και ελέγχους μηδενικών.

Code Examples

```
/* Η συνάρτηση let επιστρέφει τον πρώτο αριθμό της λίστας
   στο τετράγωνο. */
val firstItemSquared = listOf(1, 2, 3).first().let {
    it * it
}
----------------------------------------------------------------

/* Χωρίς την συνάρτηση let, θα πρέπει να εκχωρηθεί
   το πρώτο στοιχείο σε μια μεταβλητή για να γίνει
   ο πολλαπλασιασμός. */
val firstElement = listOf(1, 2, 3).first()
val firstItemSquared = firstElement * firstElement
```

With:

Η συνάρτηση with έχει σχεδιαστεί για περιπτώσεις που θέλουμε να καλέσουμε πολλαπλές συναρτήσεις σε ένα αντικείμενο και δεν θέλουμε να επαναλαμβάνουμε κάθε φορά τον δέκτη (το όνομά του).

Code Examples

```
// Η συνάρτηση with εκτυπώνει μία θύρα υποδοχής.
with(configuration) {
    println("$host:$port")
}
----------------------------------------------------------------

// Χωρίς την συνάρτηση with, θα πρέπει να γραφτεί:
println("${configuration.host}:${configuration.port}")
```

Also:

Η συνάτηση also περνά τον δείκτη που καλείται ως όρισμα σε ένα λάμδα που παρέχεται. Είναι ιδιαίτερα χρήσιμη για την προσθήκη πολλαπλών παρενεργειών από μια κοινή πηγή. Η λειτουγία της συνάρτησης είναι παρόμοια με τη λειτουργία της συνάτησης let, με τη βασική διαφορά ότι η συνάρτηση also επιστρέφει τον δέκτη και όχι το αποτέλεσμα του λάμδα.

Code Example

```
/* Η συνάρτηση also καλείται δύο φορές για οργανώσει
   δύο διαφορετικές λειτουργίες, όπου η μία εκτυπώνει
   το όνομα του αρχείου και η άλλη εκχωρεί σε μία
   μεταβλητή (fileContents) τα περιεχόμενα του αρχείου. */
var fileContents: List<String>
    File("file.txt")
            .also {
                print(it.name)
            }.also {
                fileContents = it.readLines()
            }
}
```

Takeif:

Η συνάρτηση takeif αξιολογεί μια συνθήκη που παρέχεται σε ένα λάμβδα, η οποία ονομάζεται κατηγορούμενο (predicate) και επιστρέφει είτε αληθές (true) είτε ψευδές (false) ανάλογα με τις καθορισμένες συνθήκες.

Code Examples

```
/* Η συνάρτηση takeif διαβάζει ένα αρχείο αν και μόνο αν είναι
   αναγνώσιμο και εγγράψιμο. */
val fileContents = File("myfile.txt")
    .takeIf { it.canRead() && it.canWrite() }
    ?.readText()
-----------------------------------------------------------------

// Χωρίς την συνάρτηση takeif, θα πρέπει να γραφτεί:
val file = File("myfile.txt")
val fileContents = if (file.canRead() && file.canWrite()) {
    file.readText() ...
```

3. ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΗΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

3.1 Κλάσεις (Classes) και Κατασκευαστές (Constructors)

Η Kotlin υποστηρίζει τόσο συναρτησιακό προγραμματισμό (FP) όσο και αντικειμενοστραφή προγραμματισμό (OOP). Ο προγραμματισμός αντικειμένων βασίζεται σε αντικείμενα και κατηγορίες σε πραγματικό χρόνο.

Οι κλάσεις αποτελούν τα κύρια δομικά στοιχεία οποιασδήποτε αντικειμενοστρεφούς γλώσσας προγραμματισμού. Όλα τα αντικείμενα, παρά το γεγονός ότι είναι μοναδικά, αποτελούν μέρος μιας τάξης και μοιράζονται μία κοινή συμπεριφορά. Η δήλωση μιας κλάση αποτελείται από το όνομα της κλάσης, την επικεφαλίδα της και το σώμα της. Τόσο η κεφαλίδα όσο και το σώμα είναι προαιρετικά. Εάν η κλάση δεν έχει σώμα, οι αγγείλες { } μπορούν να παραλειφθούν.

Code Example

```
/* Δήλωση μιας κλάσης με όνομα Lamp, η οποία δεν παίρνει καμία
   παραμέτρο κατασκευαστή. */
class Lamp {
    var isOn: Boolean = false

    fun turnOn() {
        isOn = true
    }

    fun turnOff() {
        isOn = false
    }
}
```

Ο κατασκευαστής είναι ένα μπλοκ κώδικα παρόμοιο με τη μέθοδο. Δηλώνεται με το ίδιο το όνομα της κλάσης, το οποίο ακολουθείται από τις παρενθέσεις ( ). Χρησιμοποιείται για την αρχικοποίηση των μεταβλητών τη στιγμή της δημιουργίας των αντικειμένων.

Υπάρχουν δύο ειδών κατασκευαστές: ο πρωτεύων και ο δευτερεύων κατασκευαστής. Ο πρωτεύων κατασκευαστής χρησιμοποιείται για την προετοιμασία μιας κλάσης με συνοπτικό τρόπο και ο δευτερεύων κατασκευαστής χρησιμοποιείται για την τοποθέτηση πρόσθετης λογικής αρχικοποίησης.

Μια κλάση στη Kotlin μπορεί να έχει έναν κύριο κατασκευαστή και έναν ή περισσότερους δευτερεύοντες κατασκευαστές. Ο κύριος κατασκευαστής είναι μέρος της κεφαλίδας της κλάσης και τοποθετείται μετά το όνομα της.

Code Examples

```
/* Δήλωση μιας κλάσης με όνομα Person, η οποία παίρνει σαν
   παραμέτρο κατασκευαστή τύπου String. */
class Person constructor(firstName: String) { ... }

/* Η λέξη constructor μπορεί να παραλειφθεί, εάν δεν υπάρχει
   κανένας τροποποιητής ορατότητας. */
class Person(firstName: String) { ... }
-------------------------------------------------------------

/* Κατά τη διάρκεια της αρχικοποίησης ενός στιγμιότυπου,
   τα μπλοκ αρχικοποίησης εκτελούνται με την ίδια σειρά,
   με την οποία εμφανίζονται στο σώμα κλάσης. */
class InitOrderDemo(name: String) {
   val firstProperty = "First property: $name".also(::println)

   init{
         println("First initializer block that prints ${name}")
   }

   val secondProperty = "Second property:
                         ${name.length}".also(::println)

   init{
        println("Second initializer block that prints
                                              ${name.length}")
   }
}
```

Code Examples

```
/* Δήλωση μιας κλάσης με όνομα Person, η οποία δέχεται τρεις
   σταθερές παραμέτρους κατασκευαστή. */
class Person(val firstName: String, val lastName: String,
                                              var age: Int) {...}
-----------------------------------------------------------------

/* Δήλωση μιας κλάσης με όνομα Contact, η οποία παίρνει δύο
   παραμέτρους κατασκευαστή (μία σταθερή παράμετρο τύπου Int
   και μία τροποποιήσιμη παράμετρο τύπου String). */
class Contact(val id: Int, var email: String)

fun main() {

    // Δημιουργία ενός στιγμιότυπου της κλάσης Customer.
    val customer = Customer()

    /* Δημιουργία ενός στιγμιότυπου της κλάσης Contact,
       χρησιμοποιώντας κατασκευαστή με δύο παραμέτρους. */
    val contact = Contact(1, "mary@gmail.com")

    // Αποκτά πρόσβαση στη σταθερά id της κλάσης Contact.
    println(contact.id)

    // Γράφει μία νέα τιμή στο email.
    contact.email = "jane@gmail.com"
}
-----------------------------------------------------------------

// Δήλωση μιας κλάσης με όνομα Person με πρωτεύων κατασκευαστή.
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
-----------------------------------------------------------------

/* Δήλωση μιας κλάσης με όνομα Person με δευτερεύων
   κατασκευαστή. */
class Person(val name: String) {
    constructor(name: String, parent: Person): this(name) {
        parent.children.add(this)
    }
}
```

3.2 Κληρονομικότητα (Inheritance)

Η κληρονομικότητα είναι θεμελιώδης χαρακτηριστικό του αντικειμενοστραφή προγραμματισμό. Επιτρέπει την δημιουργία νέων κλάσεων, οι οποίες επαναχρησιμοποιούνται, επεκτείνουν ή τροποποιούν τη συμπεριφορά των προϋπαρχόντων κλάσεων. Η προϋπάρχουσα κλάση ονομάζεται βασική κλάση (super class) και η καινούρια κλάση ονομάζεται παράγωγη κλάση. Μια παράγωγη κλάση θα αποκτήσει όλες τις γονικές κλάσεις, πεδία, ιδιότητες και μεθόδους. Ωστόσο μια παράγωγη κλάση μπορεί να προσθέσει πεδία, ιδιότητες ή νέες μεθόδους, επεκτείνοντας έτσι τη λειτουργικότητα που είναι διαθέσιμη μέσω του γονέα.

Code Example

```
/* Οι κλάσεις είναι final από προεπιλογή. Εάν θέλουμε να
   επιτρέψουμε την υπέρβαση μιας κλάσης (overridind),
   πρέπει να προσημειωθεί με την λέξη open. */
open class Dog {

   /* Οι μέθοδοι είναι και αυτοί final από προεπιλογή. Εάν
      θέλουμε να επιτρέψουμε την υπέρβαση μιας μεθόδου
      (overriding), πρέπει να προσημειωθεί με την λέξη open. */
    open fun sayHello() {
        println("Wow Wow!")
    }
}

/* Μία υποκλάση επεκτείνει μία αρχική κλάση τοποθετώντας το
  ονομα της αρχικης κλασης μετα την άνω και κάτω τελεία (:). */
class Yorkshire: Dog() {

 /* Εάν θέλουμε να πραγματοποιήσουμε την υπέρβαση μιας
    μεθόδου, θα πρέπει να προσημειωθεί με την λέξη override. */
    override fun sayHello() {
        println("Wif Wif!")
    }
}

fun main() {

// Δημιουργία ενός στιγμιότυπου με όνομα dog της κλάσης Dog.
    val dog: Dog = Yorkshire()
    dog.sayHello()
}
```

3.3 Αφηρημένες Κλάσεις (Abstract Classes)

Μια αφηρημένη κλάση είναι μία μερικώς καθορισμένη κλάση, όπου οι ιδιότητες και οι μεθόδοι της που δεν εφαρμόζονται μέσα στην κλάση, αλλά υλοποιούνται σε μια παράγωγη κλάση. Σκοπός των αφηρημένων κλάσεων είναι να παρέχουν λειτουργίες υλοποίησης μέσω της κληρονομικότητας σε υποκλάσεις που έχουν δημιουργηθεί.

Code Example

```
// Δήλωση μίας αφηρημένης κλάσης με όνομα Person.
abstract class Person(name: String) {
    init {
        println("My name is $name.")
    }

    fun displaySSN(ssn: Int) {
        println("My SSN is $ssn.")
    }

    // Δήλωση μίας αφηρημένης συνάρτησης με όνομα displayJob.
    abstract fun displayJob(description: String)
}

/* Δήλωση μιας κλάσης με όνομα Job, η οποία κληρονομεί
   όλα τα στοιχεία από την κλάση Person. */
class Job(name: String): Person(name) {

    // Πραγματοποιεί υπέρβαση της συνάρτησης displayJob.
    override fun displayJob(description: String) {
        println(description)
    }
}

fun main() {

    // Εκτυπώνει: My name is Sokratis Arvanitis
    val sok = Job("Sokratis Arvanitis")

    // Εκτυπώνει: I'm a Kotlin Developer!
    sok.displayJob("I'm a Kotlin Developer!")

    // Εκτυπώνει: My SSN is 2015524
    sok.displaySSN(2015524)
}
```

3.4 Διεπαφές (Interfaces)

Μια διεπαφή επιτρέπει τον καθορισμό κοινών ιδιοτήτων και συμπεριφορών που υποστηρίζονται από ένα υποσύνολο κλάσεων μέσα σε ένα πρόγραμμα, χωρίς να απαιτείται να οριστεί ο τρόπος με τον οποίο θα εφαρμοστούν. Χρησιμοποιώντας μια διεπαφή, μια ομάδα κλάσεων μπορεί να έχει κοινές ιδιότητες ή λειτουργίες, χωρίς να μοιράζονται μια υπερκλάση ή μια υποκατηγορία μεταξύ τους.

Code Example

```
/* Δήλωση μιας διεπαφής με όνομα Compute, η οποία περιέχει
   τρεις συναρτήσεις. */
interface Compute {

    fun computePerimeter(): Double
    fun computeArea(): Double
    fun print()
}

/* Δήλωση μιας κλάσης με παράμετρο κατασκευαστή τύπου Double,
   η οποία κληρονομεί όλα τα στοιχεία της κλάσης Compute. */
class Square(var squareLength: Double): Compute {

    // Πραγματοποιεί υπέρβαση της συνάρτησης computePerimeter.
    override fun computePerimeter()= squareLength*4

    // Πραγματοποιεί υπέρβαση της συνάρτησης computeArea.
    override fun computeArea()= squareLength*squareLength

    // Πραγματοποιεί υπέρβαση της συνάρτησης print.
    override fun print()
    {
        println("Square's Perimeter: ${computePerimeter()}")
        println("Square's Area: ${computeArea()}")
    }
}

fun main() {

    var squareLength = 5.0

  // Δημιουργία ενός στιγμιότυπου της κλάσης Square με όνομα s.
    var s = Square(squareLength)
    s.print()
}
```

3.5.1 Ένθετες Κλάσεις (Nested Classes)

Η ένθετη κλάση είναι μία κλάση η οποία ορίζεται μέσα σε μία άλλη κλάση. Στη Kotlin, η ένθετη κλάση είναι στατική από προεπιλογή. Επομένως τα δεδομένα και οι ισιότητες των μελών μπορούν να προσπελαθούν χωρίς να δημιουργηθεί ένα αντικείμενο κλάσης. Η ένθετη κλάση δεν μπορεί να έχει πρόσβαση στο δεδομένα των μελών της εξωτερικής κλάσης.

Code Example

```
class outCl {

    var a = 6
    fun printAB () {
        var b_ = inCl().b
        println("a = $a and b = $b_ from inside outCl")
    }

    // Δήλωση μιας ένθετης κλάσης με όνομα inCl.
    class inCl {

        var b = "9"
        fun printB() {
            println("b = $b from inside inCl")
        }
    }
}


fun main() {

    var a1 = outCl()

    // Εκτυπώνει: a = 6 and b = 9 from inside outCl
    a1.printAB()

    // Εκτυπώνει: b = 9 from inside inCl
    outCl.inCl().printB()
}
```

3.5.2 Εσωτερικές Κλάσεις (Inner Classes)

Η εσωτερική κλάση είναι μια κλάση η οποία ορίζεται μέσα σε μια άλλη κλάση με τη λέξη κλειδί inner. Δηλαδή μια ένθετη κλάση που προσημειώνεται με τη λέξη κλειδί inner ονομάζεται εσωτερική τάξη. Η εσωτερική κλάση δεν μπορεί να οριστεί μέσα σε διεπαφές ή σε μη εσωτερικές ένθετες κλάσεις. Το πλεονέκτημα της εσωτερικής κλάσης από την ένθετη κλάση, είναι ότι είναι σε θέση να έχει πρόσβαση σε μέλη της εξωτερικής κλάσης, ακόμη και εάν είναι ιδιωτικά. Η εσωτερική κλάση κρατά μια αναφορά σε ένα αντικείμενο της εξωτερικής κλάσης.

Code Example

```
class outCl {

    var a = 6
    fun printAB () {
        var b_ = inCl().b
        println("a = $a and b = $b_ from inside outCl")
    }

    // Δήλωση μιας εσωτερικής κλάσης με όνομα inCl.
    inner class inCl {

        var b = "9"
        fun printAB() {
            println("a = $a and b = $b from inside inCl")
        }
    }
}


fun main() {

    var a = outCl()

    // Εκτυπώνει: a = 6 and b = 9 from inside outCl
    a.printAB()

    // Εκτυπώνει: a = 6 and b = 9 from inside inCl
    a.inCl().printAB()
}
```

3.5.3 Σφραγισμένες Κλάσεις (Sealed Classes)

Μια σφραγισμένη κλάση είναι μια αφηρημένη κλάση, η οποία μπορεί να επεκταθεί με υποκλάσεις που ορίζονται ως ένθετες κλάσεις μέσα στην ίδια τη σφραγισμένη κλάση. Αποτελεί μία από τις πιο ισχυρές επιλογές απαρίθμησης. Η ιεραρχία των σφραγισμένων κλάσεων περιέχει ένα σταθερό σύνολο πιθανών επιλογών. Είναι ιδανικές για τον ορισμό αλγεβρικών τύπων δεδομένων. Ωστόσο, σε αντίθεση με τις κλάσεις απαρίθμησης, όπου κάθε επιλογή αντιπροσωπεύεται από ένα στιγμιότυπο, οι παράγωγες κλάσεις μιας σφραγισμένης κλάσης μπορούν να έχουν πολλά στιγμιότυπα.

Code Example

```
/* Δήλωση μιας σφραγισμένης κλάσης με όνομα Mammal, η οποία
   παίρνει μία σταθερή παράμετρο τύπου String και περιέχει
   δύο υποκλάσεις που πρέπει να είναι στο ίδιο αρχείο. */
sealed class Mammal(val name: String)

class Cat(val catName: String) : Mammal(catName)

class Human(val humanName: String, val job: String):
                                                Mammal(humanName)

fun greetMammal(mammal: Mammal): String {

    /* Η σφραγισμένη κλάση χρησιμοποιείται ως όρισμα για την
       συνθήκη when του ελέγχου ροής. */
    when (mammal) {

        // Πραγματοποιεί αυτόματο cast από Mammal σε Human.
        is Human -> return "Hello ${mammal.name}; You're
                                working as a ${mammal.job}"

        // Πραγματοποιεί αυτόματο cast από Mammal σε Cat.
        is Cat -> return "Hello ${mammal.name}"

    /* Η περίπτωση "else" δεν είναι απαραίτητη, επειδή όλες
       οι υποκλάσεις της σφραγισμένης κλάσεις καλύπτονται. */
    }
}

fun main() {

    println(greetMammal(Cat("Snowy")))
}
```

3.6.1 Κλάσεις Απαρίθμησης (Enum Classes)

Οι κλάσεις απαρίθμησης είναι παρόμοιες με τις σφραγισμένες κατηγορίες, με τη μόνη διαφορά ότι στις κλάσεις απαρίθμησης όλες οι τιμές των enum περιπτώσεων είναι του ίδιου τύπου. Οι κλάσεις απαρίθμησης χρησιμοποιούνται όταν το αναμενόμενο αποτέλεσμα είναι μέσα σε ένα μικρό σετ, όπως μια μικρή ποικιλία χρωμάτων ή οι ημέρες της εβδομάδας.

Code Examples

```
/* Δήλωση μιας κλάσης απαρίθμησης με όνομα State, η οποία
   περιέχει τρεις περιπτώσεις απαρίθμησης. */
enum class State {
    IDLE, RUNNING, FINISHED
}
/* Δήλωση μιας κλάσης απαρίθμησης με όνομα State, η οποία
   περιέχει μία ιδιότητα και μία μέθοδο. */
enum class Color (val rgb: Int) {
    /* Κάθε στιγμιότυπο πρέπει να περάσει ένα όρισμα για την
       παράμετρο κατασκευαστή. */
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF),
    YELLOW(0xFFFF00);
 /* Τα μέλη διαχωρίζονται από τα στιγμιότυπα,
    με ένα ερωτηματικό (;). */
    fun containsRed() = (this.rgb and 0xFF0000 != 0)
}

fun main() {
    val red = Color.RED
    /* Εκτυπώνει: RED, επειδή η συνάρτηση toString από
       προεπιλογή, επιστρέφει το όνομα του στιγμιότυπου. */
    println(red)
    println(red.containsRed())
    println(Color.BLUE.containsRed())
    val state = State.RUNNING
    val message = when (state) {
        State.IDLE -> "It's idle"
        State.RUNNING -> "It's running"
        State.FINISHED -> "It's finished"
    }
    println(message)
}
```

3.6.2 Κλάσεις Δεδομένων (Data Classes)

Οι κλάσεις δεδομένων αναδεικνύουν εύκολη τη δήλωση κατηγοριών που χρησιμοποιούνται για την αποθήκευση ορισμένων τιμών. Αυτό συμβαίνει συχνά όταν πρέπει να ορίσουμε κλάσεις με μοναδικό σκοπό τη διατήρηση δεδομένων. Όλα όσα χρειάζονται για τη χρήση τους σε συλλογές, έχουν μια χρήσιμη αναπαράσταση συμβολοσειρών και δημιουργούν αυτόματα αντίγραφα.

Code Example

```
// Δήλωση μιας κλάσης δεδομένων με όνομα User.
data class User(val name: String, val id: Int)

fun main() {

    // Η συνάρτηση toString δημιουργείται αυτόματα.
    val user = User("Alex", 1)
    println(user)
    val secondUser = User("Alex", 1)
    val thirdUser = User("Max", 2)
    println("user == secondUser: ${user == secondUser}")
    println("user == thirdUser: ${user == thirdUser}")

    /* Οι κλάσεις δεδομένων με τα ίδια περιεχόμενα εχουν το
       ίδιο hashCode. */
    println(user.hashCode())
    println(thirdUser.hashCode())

    /* Η προκαθορισμένη συνάρτηση αντιγραφής διευκολύνει
       την δημιουργία  ενός νέου στιγμιότυπου. */
    println(user.copy())

  // Οι τιμές των πεδίων μπορούν να αλλαξουν κατά την αντγραφή.
    println(user.copy("Max", 2))

    // Είναι δυνατή η αλλαγή μόνο ορισμένων τιμών.
    println(user.copy("Max"))

    /* Χρησιμοποιεί τα ορίσματα για να αλλάξει την δεύτερη
       τιμή, χωρίς να αλλάξει την πρώτη τιμή. */
    println(user.copy(id = 2))
    println("name = ${user.component1()}")
    println("id = ${user.component2()}")
}
```

3.7 Γενικές Κλάσεις (Generic Classes)

Τα Generics είναι ένας μηχανισμός γενικότητας που γίνεται πρότυπο στις σύγχρονες γλώσσες. Αυτά είναι τα ισχυρά χαρακτηριστικά που επιτρέπουν τον ορισμό κλάσεων και συναρτήσεων, τα οποία μπορούν να προσεγγιστούν χρησιμοποιώντας διαφορετικούς τύπους δεδομένων. Οι διαφορετικοί τύπου δεδομένων των κλάσεων και των συναρτήσεων ελέγχονται κατά τον χρόνο της σύνταξης, έτσι ώστε να μπορεί να αποφύγει τυχόν προβλήματα κατά το χρόνο εκτέλεσης.

Ο γενικός τύπος δεδομένων των κλάσεων ή των συνάρτησεων δηλώνεται ως παραμετροποιημένος τύπος. Ένας παραμετροποιημένος τύπος δεδομένων είναι ένα στιγμιότυπο γενικού τύπου με επιχειρήματα πραγματικού τύπου. Οι παραμετροποιημένοι τύποι δηλώνονται χρησιμοποιώντας γωνιακούς βραχίονες (< >).

Code Example

```
/* Δήλωση μιας γενικής κλάσης με όνομα Person, όπου T
   ονομάζεται η παράμετρος γενικού τύπου. */
class Person <T> (age: T){
    var age: T = age
    init{
         this.age= age

    /* Εκτυπώνει: 30
                  40 */
         println(age)
        }
}

fun main(){

    /* Δημιουργία αντικειμένου της κλάσης Person τύπου Int
       (Person<Int>(30)), το οποίο αντικαθιστά την κλάση
       Person τύπου T. */
    var ageInt: Person<Int> = Person<Int>(30)

    /* Δημιουργία αντικειμένου της κλάσης Person τύπου String
       (Person<Int>(40)), το οποίο αντικαθιστά την κλάση
       Person τύπου T. */
    var ageString: Person<String> = Person<String>("40")
}
```

4. ΣΥΛΛΟΓΕΣ

4.1 Ορισμός Συλλογών (Define Collections)

Η Kotlin διαθέτει εκατοντάδες προκατασκευασμένες κλάσεις και συναρτήσεις που μπορούν να χρησιμοποιηθούν στον κώδικα. Η πρότυπη βιβλιοθήκη (standard library) της Kotlin περιλαμβάνει κλάσεις που προσφέρουν εξαιρετικές εναλλακτικές λύσεις σε πίνακες. Οι κλάσεις και οι συναρτήσεις ομαδοποιούνται σε πακέτα. Κάθε κλάση ανήκει σε ένα πακέτο και κάθε πακέτο έχει ένα όνομα. Για παράδειγμα, το πακέτο kotlin περιλαμβάνει βασικές συναρτήσεις και τύπους και το πακέτο kotlin.collections περιλαμβάνει μια σειρά από κλάσεις που επιτρέπουν την μαζική ομαδοποίηση των αντικειμένων σε μια συλλογή.

Η δομή των συλλογών είναι ένα σύνολο από κλάσεις και διεπαφές που παρέχει μια ενιαία αρχιτεκτονική για την εκτέλεση κοινών ομάδων δεδομένων που σχετίζονται με κάποιες συναρτήσεις, όπως είναι η εύρεση (searching), η ταξινόμηση (sorting), η εισαγωγή (insertion), η διαγραφή (deletion) και ο χειρισμός (manipulation). Οι δομές των συλλογών περιέχουν διεπαφές, εφαρμογές και αλγορίθμους.

Οι συλλογές στη Kotlin υποστηρίζονται από τις συλλογές της Java, αν και η Kotlin προσθέτει περισσότερες σε αυτές. Η πιο σημαντική πτυχή είναι ότι η Kotlin διαθέτει δύο τύπους συλλογών: τροποποιήσιμες (mutable) και μη τροποποιήσιμες (immutable). Ο μη τροποποιήσιμος τύπος συλλογών δημιουργεί συλλογές χρησιμοποιώντας συγκεκριμένες λειτουργίες. Από προεπιλογή, οι συλλογές δεν είναι τροποποιήσιμες. Εάν θέλουμε να χρησιμοποιήσουμε τροποποιήσιμες συλλογές, θα πρέπει να προσημειωθούν με το πρόθεμα mutable.

Η Kotlin διαθέτει τρεις κύριους τύπους συλλογών: Λίστες (Lists), Σύνολα (Sets) και Χάρτες (Maps), όπου ο καθένας έχει το δικό του ξεχωριστό σκοπό.

Code Example

```
// Εισαγωγή της βιβλιοθήκης kotlin.collections.listOf
import kotlin.collections.listOf

// Εισαγωγή της βιβλιοθήκης kotlin.collections.*
import kotlin.collections.*

fun main() {

/* Η συνάρτηση listOf δημιουργεί μία νέα λίστα από ακέραιους
   αριθμούς, η οποία είναι μη τροποποιήσιμη.  */
    val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

    /* Η συνάρτηση first παίρνει το πρώτο στοιχείο από
       την λίστα. */
    val first = list.first()

    /* Η συνάρτηση last παίρνει το τελευταίο στοιχείο από
       την λίστα. */
    val last = list.last()

    /* Η συνάρτηση filter επιλέγει από την λίστα τους αριθμούς,
       οι οποίοι διαιρούνται ακριβώς με το 2. */
    val filter2 = list.filter { it % 2 == 0 }

    /* Η συνάρτηση filter επιλέγει από την λίστα τους αριθμούς,
       οι οποίοι διαιρούνται ακριβώς με το 4. */
    val filter4 = list.filter { it % 4 == 0 }

    // Εκτυπώνει: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    println("$list")

    // Εκτυπώνει: 1
    println("$first")

    // Εκτυπώνει: 10
    println("$last")

    // Εκτυπώνει: [2, 4, 6, 8, 10]
    println("$filter2")

    // Εκτυπώνει: [4, 8]
    println("$filter4")
}
```

Code Example

```
import kotlin.collections.listOf
import kotlin.collections.*

fun main() {

    /* Η συνάρτηση mutableListOf δημιουργεί μία νέα λίστα από
       συμβολοσειρές(ονόματα), η οποία είναι τροποποιήσιμη. */
    var mutableList1 = mutableListOf ("Sokratis", "Theodore")

    // Η συνάρτηση add προσθέτει στην λίστα δύο νέα ονόματα.
    mutableList1.add("Vivian")
    mutableList1.add("Theodore")

    var mutableList2 = mutableListOf<String>()
    mutableList2.add("Sokratis")
    mutableList2.add("Theodore")
    mutableList2.add("Vivian")

    /* Ο επανάληπτικός βρόχος for εκτελεί επανάληψη σε κάθε
       στοιχείο της λίστας mutableList1 και το εκτυπώνει. */
    for (element in mutableList1){

 /* Εκτυπώνει:  Sokratis
                Theodore
                Vivian
                Theodore */
        println(element)
    }
    println()

    /* Ο επανάληπτικός βρόχος for εκτελεί επανάληψη σε κάθε
       στοιχείο της λίστας mutableList2 και το εκτυπώνει. */
    for (element in mutableList2){

 /* Εκτυπώνει:  Sokratis
                Theodore
                Vivian */
        println(element)
    }
}
```

4.2 Πίνακες (Arrays)

Ένας πίνακας είναι μια συλλογή σταθερού αριθμού τιμών. Τα αντικείμενα του πίνακα ονομάζονται στοιχεία του πίνακα. Κάθε στοιχείο μπορεί να αναφέρεται από ένα δείκτη. Οι πίνακες στη Kotlin είναι μηδενικοί.

Αρχικοποίηση Πινάκων (Arrays Initialization):

Οι πίνακες δημιουργούνται με τις συναρτήσεις arrayOf και intArrayOf ή με τις κλάσεις intArray και floatArray. Το παράδειγμα που δίνεται παρακάτω, δείχνει τρόπους αρχικοποίησης πινάκων και δημιουργεί τρεις πίνακες.

Code Example

```
/* Εισαγωγή της βιβλιοθήκης java.util.Arrays, για να
   καταλάβει ο μεταγλωτιστής ότι η μεταβλητή Arrays
   δεν είναι άκυρη αναφορά. */
import java.util.Arrays

fun main() {

    /* Η συνάρτηση arrayOf δημιουργεί ένα πίνακα από
       ακέραιους αριθμούς. */
    val nums = arrayOf(1, 2, 3, 4, 5)

   /* Η συνάρτηση intArray δημιουργεί ένα πίνακα από
      ακέραιους αριθμούς, λαμβάνοντας ως παραμέτρους:
      τον αριθμό τον στοιχείων του πίνακα και μία συνάρτηση. */
    val nums2 = IntArray(5, { i -> i * 2 + 3})

    /* Η συνάρτηση toStirng χρησιμοποιείται για να
       εκτυπώσει τα περιεχόμενα των πινάκων.
       Εκτυπωνει: [1, 2, 3, 4, 5]  */
    println(Arrays.toString(nums))

    // Εκτυπωνει: [3, 5, 7, 9, 11]
    println(Arrays.toString(nums2))
}
```

Βασικά Στοιχεία Πινάκων (Arrays Basics):

Οι πίνακες υποστηρίζουν κάποιες ενσωματωμένες συναρτήσεις: count, max, min, sum, average. Πιο συγκεκριμένα οι συναρτήσεις αυτές υπολογίζουν τον αριθμό των στοιχείων, το μέγιστο και το ελάχιστο στοιχείο, το άθροισμα και τον μέσο όρο των στοιχείων ενός πίνακα.

Code Example

```
fun main() {

    /* Η συνάρτηση intArrayOf δημιουργεί ένα πίνακα από
       ακέραιους αριθμούς. */
    val nums = intArrayOf(1, 2, 3, 4, 5)

    /* Η συνάρτηση count υπολογίζει τον αριθμό των στοιχείων
       του πίνακα. */
    val nOfValues = nums.count()

   // Η συνάτηση max υπολογίζει το μέγιστο στοιχείο του πίνακα.
    val maxValue = nums.max()

  // Η συνάτηση min υπολογίζει το ελάχιστο στοιχείο του πίνακα.
    val minValue = nums.min()

    /* Η συνάρτηση sum υπολογίζει το άθροισμα των στοιχείων
       του πίνακα. */
    val sumOfValues = nums.sum()

    /* Η συνάρτηση sum υπολογίζει τον μέσο όρο των στοιχείων
       του πίνακα. */
    val avg = nums.average()

    // Εκτυπώνει: There are 5 elements
    println("There are $nOfValues elements")

    // Εκτυπώνει: The maximum is 5
    println("The maximum is $maxValue")

    // Εκτυπώνει: The minimum is 1
    println("The minimum is $minValue")

    // Εκτυπώνει: The sum of values is 15
    println("The sum of values is $sumOfValues")
    println("The average is $avg") }
```

Ευρετήριο Πίνακα (Array Indexing):

Κάθε στοιχείο ενός πίνακα έχει ένα δείκτη. Οι δείκτες στους πίνακες ξεκινούν από το μηδέν και το τελευταίο στοιχείο τους έχει δείκτη -1.

Code Example

```
fun main (){

    val nums = intArrayOf(1, 2, 3, 4, 5)

    /* Εκτυπώνει την τιμή με τον δείκτη 2, δηλαδή το τρίτο
       στοιχείο του πίνακα. Ο δείκτης ενός στοιχείο του
       πίνακα προστίθεται μεταξύ ενός ζεύγους αγκύλων.

       Εκτυπώνει: 1  */
    println(nums[0])

    // Εκτυπώνει: 2
    println(nums[1])

    // Εκτυπώνει: 3
    println(nums[2])

    nums[0] = 11
    nums[1] = 12
    nums[2] = 15

    // Εκτυπώνει: 11
    println(nums[0])

    // Εκτυπώνει: 12
    println(nums[1])

    // Εκτυπώνει: 15
    println(nums[2])
}
```

Βασικές Συναρτήσεις Πινάκων (Arrays Basic Operations):

Οι πίνακες διαθέτουν ορισμένες βασικές συναρτήσεις: get, set, plus, sliceArray, first, last και indexOf. Πιο συγκεκριμένα οι συναρτήσεις αυτές χρησιμοποιούνται για ανάκτηση και τροποποιήση κάποιων στοιχείων, πρόσθεση κάποιου στοιχείου, δημιουργία ένας υποπίνακα, ανάκτηση του πρώτου και του τελευταίου στοιχείου και λήψη ενός δείκτη από κάποιο στοιχείο του πίνακα.

Code Example

```
import java.util.Arrays

fun main() {
    val nums = arrayOf(1, 2, 3, 4, 5)

    /* Η συνάρτηση get παίρνει ένα στοιχείο από τον πίνακα με
       δείκτη 0. Εκτυπώνει: 1 */
    println(nums.get(0))

    /* Η συνάρτηση set θέτει τα στοιχεία του πίνακα σε
       καθορισμένη τιμή. */
    nums.set(0, 0)

    // Εκτυπώνει: [0, 2, 3, 4, 5]
    println(Arrays.toString(nums))

   /* Η συνάρτηση plus προσθέτει ένα νέο στοιχείο στον πίνακα,
      δημιουργώντας ένα νέο πίνακα. Δημιουργήθηκε νέος πίνακας,
      γιατί οι πίνακες στη Kotlin έχουν σταθερό μέγεθος. */
    val nums2 = nums.plus(1)

    // Εκτυπώνει: [0, 2, 3, 4, 5, 1]
    println(Arrays.toString(nums2))

    /* Η συνάρτηση sliceArray δημιουργεί έναν υποπίνακα
       από τον αρχικό πίνακα. */
    val slice = nums.sliceArray(1..3)

    println(Arrays.toString(slice))    // Εκτυπώνει: [2, 3, 4]

    println(nums.first())              // Εκτυπώνει: 0

    println(nums.last())               // Εκτυπώνει: 5
    println(nums.indexOf(5))}          // Εκτυπώνει: 4
```

Επανάληψη σε Πίνακα (Array Iterate):

Το παρακάτω παράδειγμα παρουσιάζει βρόχους επανάληψης σε πίνακες, χρησιμοποιώντας τέσσερις διαφορετικούς τρόπους μετάβασης σε αυτούς.

Code Example

```
fun main() {
    val nums = arrayOf(1, 2, 3, 4, 5, 6, 7)

 /* Η συνάρτηση forEach διασχίζει όλο το πίνακα και εφαρμόζει
    μία ενέργεια σε κάθε στοιχείο του πίνακα, στην περίπτωσή
    μας το εκτυπώνει.  Εκτυπώνει: 1 2 3 4 5 6 7 */
    nums.forEach({ e -> print("$e ") })
    println()

   /* Η συνάρτηση forEachIndexed εκτελεί τη δεδομένη
      ενέργεια σε κάθε στοιχείο, παρέχοντας διαδοχικό
      δείκτη με το στοιχείο.
                                Εκτυπώνει: nums[0] = 1
                                           nums[1] = 2
                                           nums[2] = 3
                                           nums[3] = 4
                                           nums[4] = 5
                                           nums[5] = 6
                                           nums[6] = 7 */
    nums.forEachIndexed({i, e -> println("nums[$i] = $e")})

    /* Ο βρόχος επανάληψης for διασχίζει όλο τον πίνακα
       στοιχείο προς στοιχείο. */
    for (e in nums) {
        print("$e ")          // Εκτυπώνει: 1 2 3 4 5 6 7
    }
    println()

    /* Εκτελεί επανάληψη στη λίστα χρησιμοποιώντας το
       ListIterator και το βρόχο επανάληψης while. */
    val it: Iterator<Int> = nums.iterator()

    while (it.hasNext()) {
        val e = it.next()

        // Εκτυπώνει: 1 2 3 4 5 6 7
        print("$e ")
    }
}
```

Ταξινόμηση Πινάκων (Sorting Arrays):

Τα στοιχεία ενός πίνακα μπορούν να ταξινομηθούν είτε σε αύξουσα είτε σε φθίνουσα σειρά. Η συνάρτηση sortedArray ταξινομεί τα στοιχεία του πίνακα σε αύξουσα σειρά και η συνάρτηση sortedArrayDescending σε φθίνουσα σειρά.

Code Example

```
import java.util.Arrays

fun main() {
    val nums = arrayOf(7, 3, 3, 4, 5, 9, 1)

    /* Η συνάρτηση sortedArray ταξινομεί τον πίνακα
       σε αύξουσα σειρά. */
    val sortedNums = nums.sortedArray()

    /* Η συνάρτηση sortedArrayDescending ταξινομεί τον πίνακα
       σε φθίνουσα σειρά. */
    val sortedNumsDesc = nums.sortedArrayDescending()

    // Εκτυπώνει: [1, 3, 3, 4, 5, 7, 9]
    println(Arrays.toString(sortedNums))

    // Εκτυπώνει: [9, 7, 5, 4, 3, 3, 1]
    println(Arrays.toString(sortedNumsDesc))
}
```

Πίνακες Δύο Διαστάσεων (Two-Dimensional Arrays):

Στο παράδειγμα που δίδεται παρακάτω, δημιουργεί ένα πίνακα δύο διαστάσεων με την εμφωλευμένη συνάρτηση intArrayOf μέσα στην συνάρτηση arrayOf.

Code Example

```
import java.util.Arrays

fun main() {
    val array = arrayOf(intArrayOf(1, 2),
                        intArrayOf(3, 4),
                        intArrayOf(5, 6, 7))

    // Εκτυπώνει: [[1, 2], [3, 4], [5, 6, 7]]
    println(Arrays.deepToString(array)) }
```

Φιλτράρισμα Πινάκων (Filtering Arrays):

Το φιλτράρισμα είναι μια λειτουργία στην οποία περνούν μόνο τα στοιχεία που πληρούν ορισμένα κριτήρια. Το φιλτράρισμα ενός πίνακα πραγματοποιείται με την συνάρτηση filter.

Code Example

```
fun main(args: Array<String>) {

    val nums = arrayOf(1, -2, 3, 4, -5, 7)

    /* Η συνάρτηση filter χρησιμοποιείται για να πάρει
       μόνο τις θετικές τιμές και η συνάρτηση forEach
       για να τις εκτυπώσει.
       Εκτυπώνει: 1 3 4 7 */
    nums.filter { e -> e > 0 }.forEach { e -> print("$e ") }
}
```

Αναγωγή Πινάκων (Array Reduction):

Η αναγωγή είναι μια τερματική λειτουργία που συγκεντρώνει τις τιμές του πίνακα σε μία μόνο τιμή. Η συνάρτηση reduce εφαρμόζει μια λειτουργία ενάντια σε έναν συσσωρευτή και σε κάθε στοιχείο του πίνακα (από αριστερά προς τα δεξιά) για να το μειώσει σε μία μόνο τιμή.

Code Example

```
fun main() {

    val nums = intArrayOf(2, 3, 4, 5, 6, 7)

    /* Η συνάρτηση reduce συσσωρεύει τα στοιχεία του πίνακα.
       Η λέξη product είναι ο συσσωρευτής και η λέξη next
       είναι η επόμενη τιμή στον πίνακα. */
    val total = nums.reduce { product, next -> product * next }

    // Εκτυπώνει: 5040
    println(total)
}
```

4.3 Λίστες (Lists)

Οι λίστες είναι γενικές ταξινομημένες συλλογές στοιχείων, οι οποίες επιτρέπουν τα αντίγραφα. Η Kotlin διαθέτει λίστες μόνο για ανάγνωση (read-only lists) ή αλλιώς μη τροποποιήσιμες και μεταβλητές λίστες (mutable lists) ή αλλιώς τροποποιήσιμες. Πιο συγκεκριμένα η συνάρτηση listOf δημιουργεί λίστες μόνο για ανάγνωση και η συνάρτηση mutableListOf δημιουργεί μεταβλητές λίστες.

Code Example

```
fun main() {

 /* Η συνάρτηση listOf δημιουργεί ένα νέα λίστα από λέξεις,
    η οποίο διατίθεται μόνο για ανάγνωση (μη τροποποιήσιμη). */
    val words = listOf("pen", "cup", "dog", "spectacles")

    // Εκτυπώνει: The list contains 4 elements.
    println("The list contains ${words.size} elements.")
}
-----------------------------------------------------------------

fun main() {

    /* Η συνάρτηση mutableSetOf δημιουργεί ένα νέα λίστα από
       ακέραιους αριθμούς, η οποία είναι τροποποιήσιμη. */
    val nums = mutableListOf(3, 4, 5)

    /* Η συνάρτηση add προσθέτει ένα νέο στοιχείο στο τέλος
       της λίστας. */
    nums.add(6)

    /* Η συνάρτηση addAll προσθέτει πολλά στοιχεία στο τέλος
       της λίστας. */
    nums.addAll(listOf(7, 8, 9))

    // Εκτυπώνει: [3, 4, 5, 6, 7, 8, 9]
    println(nums)

    /* Η συνάρτηση shuffle τοποθετεί τα στοιχεία της λίστας
       σε τυχαία σειρά. */
    nums.shuffle()

    // Εκτυπώνει: [5, 3, 8, 7, 9, 6, 4]
    println(nums)
}
```

Βασικά Στοιχεία Λιστών (Lists Basics):

Οι λίστες, υποστηρίζουν κάποιες ενσωματωμένες συναρτήσεις: count, max, min, sum, average. Πιο συγκεκριμένα οι συναρτήσεις αυτές υπολογίζουν τον αριθμό των στοιχείων, το μέγιστο και το ελάχιστο στοιχείο, το άθροισμα και τον μέσο όρο των στοιχείων μιας λίστας.

Code Example

```
fun main() {

    /* Η συνάρτηση listOf δημιουργεί μία λίστα από ακέραιους
       αριθμούς, η οποία είναι μη τροποποιήσιμη. */
    val nums = listOf(11, 5, 3, 8, 1, 9, 6, 2)

    /* Η συνάρτηση count υπολογίζει τον αριθμό των στοιχείων
       της λίστας. */
    val len = nums.count()

    /* Η συνάτηση max υπολογίζει το μέγιστο στοιχείο
       της λίστας. */
    val max = nums.max()

    /* Η συνάτηση min υπολογίζει το ελάχιστο στοιχείο
       της λίστας. */
    val min = nums.min()

    /* Η συνάρτηση sum υπολογίζει το άθροισμα των στοιχείων
       της λίστας. */
    val sum = nums.sum()

    /* Η συνάρτηση average υπολογίζει τον μέσο όρο των
       στοιχείων της λίστας. */
    val avg = nums.average()

    val msg = """
                max: $max, min: $min,
                count: $len, sum: $sum,
                average: $avg
               """

 /* Εκτυπώνει: max: 11, min: 1,
               count: 8, sum: 45,
               average: 5.625 */
    println(msg.trimIndent()) }
```

Ευρετήριο Λίστας (List Indexing):

Κάθε στοιχείο μιας λίστας έχει ένα δείκτη. Οι δείκτες στις λίστες ξεκινούν από το μηδέν και το τελευταίο στοιχείο τους έχει δείκτη -1.

Code Example

```
fun main() {

    val words = listOf("pen", "cup", "dog", "person", "cement",
                       "coal", "spectacles", "cup", "bread")

    /* Η συνάρτηση get ανακτά το στοιχείο της λίστας
       με δείκτη 0 (pen). */
    val w1 = words.get(0)

    // Εναλλακτικός τρόπος ανάκτησης στοιχείου από λίστα.
    val w2 = words[0]

    /* Η συνάρτηση indexOf επιστρέφει τον δείκτη,
       στον οποίο εμφανίστηκε για πρώτη φορά η λέξη cup. */
    val i1 = words.indexOf("cup")

    /* Η συνάρτηση lastIndexOf επιστρέφει τον δείκτη,
       στον οποίο εμφανίστηκε για τελευταία φορά η λέξη cup. */
    val i2 = words.lastIndexOf("cup")

    /* Η συνάρτηση lastIndex επιστρέφει τον δείκτη του
       τελευταίο στοιχείου της λίστας ή -1 εάν είναι κενή. */
    val i3 = words.lastIndex

    // Εκτυπώνει: pen
    println(w1)

    // Εκτυπώνει: pen
    println(w2)

    // Εκτυπώνει: The first index of cup is 1
    println("The first index of cup is $i1")

    // Εκτυπώνει: The last index of cup is 7
    println("The last index of cup is $i2")

    // Εκτυπώνει: The last index of the list is 8
    println("The last index of the list is $i3")
}
```

Μετρητής Λίστας (List Count):

Η συνάρτηση count επιστρέφει τον αριθμό των στοιχείων της λίστας. Στο παρακάτω παράδειγμα, υπολογίζει τον αριθμό τον στοιχείων της λίστας, τον αριθμό των στοιχείων με αρνητικές τιμές και τον αριθμό των στοιχείων με άρτιο περιεχόμενο.

Code Example

```
fun main() {

    /* Η συνάρτηση listOf δημιουργεί μία λίστα από αριθμούς,
       η οποία είναι μη τροποποιήσιμη. */
    val nums = listOf(4, 5, 3, 2, 1, -1, 7, 6, -8, 9, -12)

    /* Η συνάρτηση count υπολογίζει τον αριθμό των στοιχείων
       της λίστας. */
    val len = nums.count()

    /* Η ιδιότητα size καθορίζει τον αριθμό των στοιχείων
       της λίστας. */
    val size = nums.size

    /* Η συνάρτηση count μπορεί να πάρει μία προκαθορισμένη
       συνάρτηση ως παράμετρο. Επιστρέφει true για τιμές οι
       οποίες είναι χαμηλότερες από μηδέν. */
    val n1 = nums.count { e -> e < 0 }

    /* Η συνάρτηση count υπολογίζει τον αριθμό των στοιχείων
       της λίστας, που έχουν άρτιο περιεχόμενο. */
    val n2 = nums.count { e -> e % 2 == 0 }

    // Εκτυπώνει: There are 11 elements
    println("There are $len elements")

    // Εκτυπώνει: The size of the list is 11
    println("The size of the list is $size")

    // Εκτυπώνει: There are 3 negative values
    println("There are $n1 negative values")

    // Εκτυπώνει: There are 5 even values
    println("There are $n2 even values")
}
```

Πρώτο και Τελευταίο Στοιχείο μίας Λίστας (First and Last):

Η συναρτήση first χρησιμοποιείται για την ανάκτηση του πρώτου στοιχείου μίας λίστας και η συνάρτηση last για το τελευταίο στοιχείο της. Στο παράδειγμα που δίδεται παρακάτω, δημιουργεί μία λίστα από συμβολοσειρές και υπολογίζει το πρώτο και το τελευταίο στοιχείο.

Code Example

```
fun main() {

    /* Η συνάρτηση listOf δημιουργεί μία λίστα από
       συμβολοσειρές, η οποία είναι μη τροποποιήσιμη. */
    val words = listOf("pen", "cup", "dog", "person",
                       "cement", "coal", "spectacles")

    // Η συνάρτηση first παίρνει το πρώτο στοιχείο της λίστας.
    val w1 = words.first()

    /* Η συνάρτηση last παίρνει το τελευταίο στοιχείο της
       λίστας. */
    val w2 = words.last()

    /* Η συνάρτηση findlast ανακτά το τελευταίο στοιχείο της
       λίστας, που ξεκινά με το γράμμα 'c' (coal). */
    val w3 = words.findLast { w -> w.startsWith('c') }

    /* Η συνάρτηση first ανακτά το πρώτο στοιχείο της
       λίστας, που ξεκινά με το γράμμα 'c' (cup). */
    val w4 = words.first { w -> w.startsWith('c') }

    // Εκτυπώνει: pen
    println(w1)

    // Εκτυπώνει: spectacles
    println(w2)

    // Εκτυπώνει: coal
    println(w3)

    // Εκτυπώνει: cup
    println(w4)
}
```

Επανάληψη σε Λίστα (List Iterate):

Η λίστα επαναλήψεων είναι η διαδικασία μετάβασης των στοιχείων μιας λίστας σε μία άλλη λίστα, ένα προς ένα. Το παράδειγμα που δίδεται παρακάτω, παρουσιάζει τρεις τρόπους επανάληψης σε μία λίστα.

Code Example

```
fun main() {

    /* Η συνάρτηση listOf δημιουργεί μία λίστα από
       συμβολοσειρές, η οποία είναι μη τροποποιήσιμη. */
    val words = listOf("pen", "cup", "dog", "person",
                                    "cement", "coal")

    /* Η συνάρτηση forEach εκτελεί την επανάληψη σε κάθε
       στοιχείο της λίστας.
       Εκτυπώνει: pen cup dog person cement coal */
    words.forEach { e -> print("$e ") }
    println()

    /* Ο βρόχος επανάληψης for διασχίζει όλη τη λίστα
       στοιχείο προς στοιχείο. */
    for (word in words) {

        // Εκτυπώνει: pen cup dog person cement coal
        print("$word ")
    }
    println()

    /* Εκτελεί επανάληψη στη λίστα χρησιμοποιώντας το
       ListIterator και το βρόχο επανάληψης while. */
    val it: ListIterator<String> = words.listIterator()

    while (it.hasNext()) {
        val e = it.next()

        // Εκτυπώνει: pen cup dog person cement coal
        print("$e ")
    }
    println()
}
```

Ταξινόμηση Λίστας (List Sorting):

Τα στοιχεία μιας λίστας μπορούν να ταξινομηθούν είτε σε αύξουσα είτε σε φθίνουσα σειρά. Δεδομένου ότι οι λίστες που χρησιμοποιούνται είναι μόνο για ανάγνωση, οι συναρτήσεις δεν αλλάζουν τη λίστα αλλά επιστρέφουν μία νέα τροποποιημένη λίστα.

Code Example

```
data class Car(var name: String, var price: Int)

fun main() {
    val nums = listOf(11, 5, 3, 8, 1, 9, 6, 2)

    /* Η συνάρτηση sorted ταξινομεί τα στοιχεία της λίστας
       σε αύξουσα σειρά. */
    val sortAsc = nums.sorted()

    /* Η συνάρτηση sortedDescending ταξινομεί τα στοιχεία της
       λίστας σε φθίνουσα σειρά. */
    val sortDesc = nums.sortedDescending()

    // Η συνάρτηση reversed αντιστρέφει τα στοιχεία της λίστας.
    val revNums = nums.reversed()

    // Εκτυπώνει: [1, 2, 3, 5, 6, 8, 9, 11]
    println(sortAsc)

    // Εκτυπώνει: [11, 9, 8, 6, 5, 3, 2, 1]
    println(sortDesc)

    // Εκτυπώνει: [2, 6, 9, 1, 8, 3, 5, 11]
    println(revNums)

    val cars = listOf(Car("Mazda", 6300), Car("Toyota", 12400),
                      Car("Skoda", 5670), Car("Mercedes", 18600))

    /* Η συνάρτηση sortedBy ταξινομεί τα αυτοκίνητα
       με τα ονόματά τους σε αύξουσα σειρά. */
    val res = cars.sortedBy { car -> car.name }

    /* Εκτυπώνει: Car(name=Mazda, price=6300)
                  Car(name=Mercedes, price=18600)
                  Car(name=Skoda, price=5670)
                  Car(name=Toyota, price=12400) */
    res.forEach { e -> println(e) } }
```

List Contain:

Η συνάρτηση contains ελέγχει εάν μία λίστα περιέχει κάποια καθορισμένα στοιχεία. Στο παρακάτω παράδειγμα, ελέγχει εάν μία λίστα περιέχει ένα ή περισσότερα καθορισμένα στοιχεία.

Code Example

```
fun main() {

    val nums = listOf(4, 5, 3, 2, 1, -1, 7, 6, -8, 9, -12)

    /* Η συνάρτηση contains ελέγχει έαν η λίστα περιέχει
       την τιμή 4 και επιστρέφει μία λογική τιμή (boolean). */
    val r = nums.contains(4)

    if (r) println("The list contains 4")
    else println("The list does not contain 4")

    /* Η συνάρτηση containsAll ελέγχει έαν η λίστα περιέχει
       τις τιμές 1 και -1. */
    val r2 = nums.containsAll(listOf(1, -1))

    if (r2) println("The list contains -1 and 1")
    else println("The list does not contain -1 and 1")
}
```

Map Λίστας (List Map):

Η συνάρτηση map επιστρέφει μια τροποποιημένη λίστα, εφαρμόζοντας μια λειτουργία μετασχηματισμού σε κάθε ένα από τα στοιχεία της λίστας.

Code Example

```
fun main() {
    val nums = listOf(1, 2, 3, 4, 5, 6)

    /* Η συνάρτηση map πολλαπλασιάζει κάθε στοιχείο της
       λίστας με τον αριθμό 2. */
    val nums2 = nums.map { e -> e * 2 }

    // Εκτυπώνει: [2, 4, 6, 8, 10, 12]
    println(nums2) }
```

Φιλτράρισμα Λίστας (List Filter):

Το φιλτράρισμα είναι μια λειτουργία στην οποία περνούν μόνο τα στοιχεία που πληρούν ορισμένα κριτήρια. Το φιλτράρισμα μιας λίστας πραγματοποιείται με την συνάρτηση filter.

Code Example

```
data class Car(var name: String, var price: Int)

fun main() {

    /* Η συνάρτηση listOf δημιουργεί μία λίστα από
       συμβολοσειρές, η οποία είναι μη τροποποιήσιμη. */
    val words = listOf("pen", "cup", "dog", "person",
                       "cement", "coal", "spectacles")

    /* Η συνάρτηση filter παίρνει μια προκαθορισμένη συνάρτηση
       σαν παράμετρο. Φιλτράρει λέξεις των οποίων το μήκος
       ισούται με 3. */
    val words2 = words.filter { e -> e.length == 3 }

    /* Η συνάρτηση filterNot φιλτράρει λέξεις των οποίων
       το μήκος δεν ισούται με 3. */
    val words3 = words.filterNot { e -> e.length == 3 }

    // Εκτυπώνει: pen cup dog
    words2.forEach { e -> print("$e ") }
    println()

    // Εκτυπώνει: person cement coal spectacles
    words3.forEach { e -> print("$e ") }
    println()

    val cars = listOf(Car("Mazda", 6300), Car("Toyota", 12400),
                      Car("Skoda", 5670), Car("Mercedes", 18600))

    /* Η συνάρτηση filter φιλτράρει τα αυτοκινήτα, των οποίων
       η τιμή είναι μεγαλύτερη από 10000. */
    val res = cars.filter { car -> car.price > 10000 }

    /* Εκτυπώνει: Car(name=Toyota, price=12400)
                  Car(name=Mercedes, price=18600) */
    res.forEach { e -> println(e) }
}
```

Μέγιστο Στοιχείο Λίστας (List Maximum):

Η συνάρτηση max χρησιμοποιείται για την εύρεση της μέγιστης τιμής σε μία λίστα. Το παρακάτω παράδειγμα, χρησιμοποιεί την συνάρτηση max για την εύρεση του μέγιστου ακέραιου αριθμού της λίστας.

Code Example

```
data class Car(var name: String, var price: Int)

fun main() {

    val nums = listOf(11, 5, 23, 8, 1, 9, 6, 2)

    /* Η συνάρτηση max υπολογίζει το μέγιστο στοιχείο
       της λίστας.  Εκτυπώνει: 23 */
    println(nums.max())
}
```

List Slices:

Οι υπολίστες αποτελούν τμήματα μιας λίστας και δημιουργούνται με τη συνάρτηση slice. Η συνάρτηση slice παίρνει τους δείκτες των στοιχείων που πρέπει να ληφθούν.

Code Example

```
fun main() {

    val nums = listOf(1, 2, 3, 4, 5, 6)

    /* Η συνάρτηση slice δημιουργεί μία υπολίστα με τα στοιχεία
       που έχουν δείκτες 1, 2 και 3. */
    val nums2 = nums.slice(1..3)

    // Παρέχει ρητά μία λίστα με δείκτες 3, 4 και 5.
    val nums3 = nums.slice(listOf(3, 4, 5))

    // Εκτυπώνει: [2, 3, 4]
    println(nums2)

    // Εκτυπώνει: [4, 5, 6]
    println(nums3)
}
```

Αναγωγή σε Λίστα (List Reduction):

Η αναγωγή είναι μια τερματική λειτουργία που συγκεντρώνει τις τιμές της λίστας σε μία μόνο τιμή. Η συνάρτηση reduce εφαρμόζει μια λειτουργία ενάντια σε έναν συσσωρευτή και σε κάθε στοιχείο της λίστας (από αριστερά προς τα δεξιά) για να το μειώσει σε μία μόνο τιμή.

Code Example

```
fun main() {

    val nums = listOf(4, 5, 3, 2, 1, 7, 6, 8, 9)

    /* Η συνάρτηση reduce υπολογίζει το άθροισμα των
       στοιχείων της λίστας. Η λέξη total αναφέρεται στον
       συσσωρευτή και η λέξη next στο επόμενο στοιχείο της
       λίστας. */
    val sum = nums.reduce { total, next -> total + next }

    // Εκτυπώνει: 45
    println(sum)
}
```

Αναδίπλωση Λίστας (List Fold):

Η συνάρτηση αναδίπλωσης είναι παρόμοια με τη συνάρτηση μείωσης. Η αναδίπλωση είναι μια τερματική λειτουργία που συγκεντρώνει τις τιμές της λίστας σε μία μόνο τιμή. Η διαφορά είναι ότι η αναδίπλωση αρχίζει με μια αρχική τιμή.

Code Example

```
fun main() {

    val expenses = listOf(20, 40, 80, 15, 25)
    val cash = 550

    /* Η συνάρτηση fold συνάγει όλα τα έξοδα από τα μετρητά
       και επιστρέφει την υπόλοιπη αξία. */
    val res = expenses.fold(cash) {total, next -> total - next}

    // Εκτυπώνει: 370
    println(res)
}
```

List Chunks:

Η συνάρτηση chunked χρησιμοποιείται για να χωρίσει τη λίστα σε μία λίστα από υπολίστες.

Code Example

```
fun main() {

    val nums = listOf(1, 2, 3, 4, 5, 6)

    /* Διαχωρίζει τη λίστα σε μια λίστα δύο στοιχείων και
       εφαρμόζει αναδίπλωση σε αυτήν. Η επόμενη είναι μια
       λίστα, στην οποία μπορούμε να χρησιμοποιήσουμε τις
       λειτουργίες των δεικτών. */
    val res = nums.chunked(2).fold(0) {total, next ->
                                        total + next[0]*next[1]}

    // Εκτυπώνει: 44
    println(res)
}
```

Διαχωρισμός Λίστας (List Partition):

Η λειτουργία διαμερισμού χωρίζει την αρχική συλλογή σε ζεύγος λιστών. Η πρώτη λίστα περιέχει στοιχεία για τα οποία το συγκεκριμένο κατηγόρημα αποδίδει αληθές, ενώ η δεύτερη λίστα περιέχει στοιχεία για τα οποία το κατηγόρημα αποδίδει ψευδές.

Code Example

```
fun main() {

    val nums = listOf(4, -5, 3, 2, -1, 7, -6, 8, 9)

    /* Η συνάρτηση partition διαιρεί τη λίστα σε δύο υπολίστες
       με μία κίνηση. */
    val (nums2, nums3) = nums.partition { e -> e < 0 }

    // Εκτυπώνει: [-5, -1, -6]
    println(nums2)

    // Εκτυπώνει: [4, 3, 2, 7, 8, 9]
    println(nums3)
}
```

List Group By:

Η συνάρτηση groupBy ομαδοποιεί τα στοιχεία της αρχικής λίστας με το κλειδί που επιστρέφεται από τη δοσμένη λειτουργία, που εφαρμόζεται σε κάθε στοιχείο της λίστας. Επιστρέφει έναν χάρτη όπου κάθε κλειδί ομάδας συσχετίζεται με μια λίστα αντίστοιχων στοιχείων. Το παρακάτω παράδειγμα δείχνει τη χρήση της συνάρτησης groupBy().

Code Example

```
fun main() {

    /* Η συνάρτηση listOf δημιουργεί μία λίστα από ακέραιους
       αριθμούς, η οποία είναι μη τροποποιήσιμη. */
    val nums = listOf(1, 2, 3, 4, 5, 6, 7, 8)

    /* Η συνάρτηση listOf δημιουργεί μία λίστα από
       συμβολοσειρές, η οποία είναι μη τροποποιήσιμη. */
    val words = listOf("as", "pen", "cup", "doll", "my",
                                        "dog", "spectacles")

    /* Η συνάρτηση groupBy δημιουργεί ένα χάρτη, ο οποίος
       έχει δύο κλειδιά: "even" και "odd". Το "even" δείχνει
       σε μια λίστα με άρτιες τιμές και το "odd" σε μια λίστα
       με περιττές τιμές. */
    val res = nums.groupBy {if (it % 2 == 0) "even" else "odd"}

    /* Δημιουργεί έναν χάρτη με ακέραια κλειδιά. Κάθε λέξη
       ομαδοποιεί λέξεις που έχουν ένα ορισμένο μήκος. */
    val res2 = words.groupBy { it.length }

    // Εκτυπώνει: {odd=[1, 3, 5, 7], even=[2, 4, 6, 8]}
    println(res)

 /* Εκτυπώνει:
    {2=[as, my], 3=[pen, cup, dog], 4=[doll], 10=[spectacles]}
                                                          */
    println(res2)
}
```

List Any:

Η συνάρτηση any επιστρέφει αληθές (true) εάν τουλάχιστον ένα στοιχείο ταιριάζει με τη δεδομένη συνθήκη.

Code Example

```
fun main() {
    val nums = listOf(4, 5, 3, 2, -1, 7, 6, 8, 9)
    val r = nums.any { e -> e > 10 }
    if (r) println("There is a value greater than ten")
    else println("There is no value greater than ten")

    /* Η συνάρτηση any ελέγχει εάν η λίστα περιέχει τουλάχιστον
       μία αρνητική τιμή. Επιστρέφει μια λογική τιμή (true). */
    val r2 = nums.any { e -> e < 0 }
    if (r2) println("There is a negative value")
    else println("There is no negative value")
}
```

List All:

Η συνάρτηση all επιστρέφει true εάν όλα τα στοιχεία ικανοποιούν τη δεδομένη συνθήκη.

Code Example

```
fun main() {
   val nums = listOf(4, 5, 3, 2, -1, 7, 6, 8, 9)
   val nums2 = listOf(-3, -4, -2, -5, -7, -8)

   /* Η συνάρτηση all ελέγχει εάν η λίστα περιέχει μόνο
      θετικές τιμές. */
   val r = nums.all { e -> e > 0 }
   if (r) println("nums list contains only positive values")
   else println("nums list not contain only positive values")

    // Ελέγχει εάν η λίστα περιέχει μόνο αρνητικές τιμές.
    val r2 = nums2.all { e -> e < 0 }
    if (r2) println("nums2 list contains only negative values")
    else println("nums2 list not contain only negative values")
}
```

List Drop:

Η συνάρτηση drop χρησιμοποιείται για να αποκλείσει ορισμένα στοιχεία από τη λίστα.

Code Example

```
fun main() {

    /* Η συνάρτηση listOf δημιουργεί μία λίστα από αριθμούς,
       η οποία είναι μη τροποποιήσιμη. */
    val nums = listOf(4, 5, 3, 2, 1, -1, 7, 6, -8, 9, -12)

    /* Η συνάρτηση drop αποκλείει τα τρία πρώτα στοιχεία
       της λίστας. */
    val nums2 = nums.drop(3)

    /* Η συνάρτηση dropLast αποκλείει τα τρία τελευταία
       στοιχεία της λίστας. */
    val nums3 = nums.dropLast(3)

    /* Η συνάρτηση dropWhile αποκλείει τα πρώτα n στοιχεία
       της λίστας, που ικανοποιούν τη δεδομένη συνθήκη. */
    val nums4 = nums.sorted().dropWhile { e -> e < 0 }

    /* Η συνάρτηση dropLastWhile αποκλείει τα τελευταία
       n στοιχεία της λίστας, που ικανοποιούν τη δεδομένη
       συνθήκη. */
    val nums5 = nums.sorted().dropLastWhile { e -> e > 0 }

    // Εκτυπώνει: [2, 1, -1, 7, 6, -8, 9, -12]
    println(nums2)

    // Εκτυπώνει: [4, 5, 3, 2, 1, -1, 7, 6]
    println(nums3)

    // Εκτυπώνει: [1, 2, 3, 4, 5, 6, 7, 9]
    println(nums4)

    // Εκτυπώνει: [-12, -8, -1]
    println(nums5)
}
```

List Take:

Η συνάρτηση take χρησιμοποιείται για να δημιουργήσει μία νέα λίστα, επλέγοντας μερικά από τα στοιχεία της αρχικής λίστας.

Code Example

```
fun main() {
    val nums = listOf(4, 5, 3, 2, 1, -1, 7, 6, -8, 9, -12)

    /* Η συνάρτηση take δημιουργεί μια νέα λίστα με τα τρία
       πρώτα στοιχεία της αρχικής λίστας. */
    val nums2 = nums.take(3)

    /* Η συνάρτηση takeLast δημιουργεί μια νέα λίστα με τα
       τρία τελευταία στοιχεία της αρχικής λίστας. */
    val nums3 = nums.takeLast(3)
    val nums4 = nums.sorted().take(3)

    /* Η συνάρτηση takeWhile παίρνει τα πρώτα n στοιχεία
       που ικανοποιούν τη δεδομένη συνθήκη. */
    val nums5 = nums.takeWhile { e -> e > 0 }
    val nums6 = nums.sortedDescending().takeWhile {e -> e > 0 }

    /* Η συνάρτηση takeIf παίρνει όλα τα στοιχεία της λίστας
       εάν πληρείται η δεδομένη συνθήκη. */
    val nums7 = nums.takeIf { e -> e.contains(6) }

    // Εκτυπώνει: [4, 5, 3]
    println(nums2)

    // Εκτυπώνει: [-8, 9, -12]
    println(nums3)

    // Εκτυπώνει: [-12, -8, -1]
    println(nums4)

    // Εκτυπώνει: [4, 5, 3, 2, 1]
    println(nums5)

    // Εκτυπώνει: [9, 7, 6, 5, 4, 3, 2, 1]
    println(nums6)

    // Εκτυπώνει: [4, 5, 3, 2, 1, -1, 7, 6, -8, 9, -12]
    println(nums7)
}
```

4.4 Σύνολα (Sets)

Τα σύνολα (sets) είναι μη ταξινομημένες συλλογές στοιχείων, τα οποία δεν επιτρέπουν τα αντίγραφα. Η Kotlin διαθέτει σύνολα μόνο για ανάγνωση (read-only sets) ή αλλιώς αμετάβλητα σύνολα και μεταβλητά σύνολα (mutable sets). Η συνάρτηση setOf δημιουργεί σύνολα μόνο για ανάγνωση που δεν περιέχουν διπλά αντικείμενα και μπορούν να μεταβληθούν με τη συνάρτηση mutableSetOf.

Code Example

```
 fun main() {

    /* Η συνάρτηση setOf δημιουργεί ένα νέο σύνολο από
       συμβολοσειρές (λέξεις), το οποίο διατίθεται μόνο
       για ανάγνωση (αμετάβλητο). */
    val words = setOf("pen", "cup", "dog", "spectacles")

     // Εκτυπώνει: The set contains 4 elements.
    println("The set contains ${words.size} elements.")
}
-----------------------------------------------------------------

/* Παρόλου που έχουν προσθεθεί δύο λέξεις με το ίδιο περιεχόμενο
   (pen), θα υπάρχει μόνο μία στην εκτύπωση. Γιατί η συνάρτηση
   setOf δεν μπορεί να περιέχει διπλότυπα αντικείμενα. */
fun main() {
    val words2 = setOf("pen", "cup", "dog", "pen", "spectacles")

            /* Εκτυπώνει: pen
                          cup
                          dog
                          spectacles */
    words2.forEach { e -> println(e)}
}
-----------------------------------------------------------------

fun main() {

    /* Η συνάρτηση mutableSetOf δημιουργεί ένα νέο σύνολο από
       συμβολοσειρές (λέξεις), το οποίο είναι μεταβλητό. */
    val words3 = mutableSetOf("pen", "cup", "dog", "cat", "ball")

    // Εκτυπώνει: The set contains 5 elements.
    println("The set contains ${words3.size} elements.") }
```

Βασικά Στοιχεία Συνόλων (Sets Basics):

Τα σύνολα, όπως οι πίνακες και οι λίστες, υποστηρίζουν κάποιες ενσωματωμένες συναρτήσεις: count, max, min, sum, average. Πιο συγκεκριμένα οι συναρτήσεις αυτές υπολογίζουν τον αριθμό των στοιχείων, το μέγιστο και το ελάχιστο στοιχείο, το άθροισμα και τον μέσο όρο των στοιχείων ενός συνόλου.

Code Example

```
fun main() {

    /* Η συνάρτηση setOf δημιουργεί ένα σύνολο από ακέραιους
       αριθμούς, το οποίο διατίθεται μόνο για ανάγνωση. */
    val nums = setOf(11, 5, 3, 8, 1, 9, 6, 2)

    /* Η συνάρτηση count υπολογίζει τον αριθμό των στοιχείων
       του συνόλου. */
    val len = nums.count()

    /* Η συνάτηση max υπολογίζει το μέγιστο στοιχείο του
       συνόλου. */
    val max = nums.max()

    /* Η συνάτηση min υπολογίζει το ελάχιστο στοιχείο του
       συνόλου. */
    val min = nums.min()

    /* Η συνάρτηση sum υπολογίζει το άθροισμα των στοιχείων
       του συνόλου. */
    val sum = nums.sum()

    /* Η συνάρτηση sum υπολογίζει τον μέσο όρο των στοιχείων
       του συνόλου. */
    val avg = nums.average()

    val msg = """
                max: $max, min: $min,
                count: $len, sum: $sum,
                average: $avg
               """

 /* Εκτυπώνει: max: 11, min: 1,
               count: 8, sum: 45,
               average: 5.625 */
    println(msg.trimIndent()) }
```

Ευρετήριο Συνόλου (Set Indexing):

Κάθε στοιχείο ενός συνόλου έχει ένα δείκτη. Οι δείκτες στα σύνολα ξεκινούν από το μηδέν και το τελευταίο στοιχείο τους έχει δείκτη -1.

Code Example

```
fun main() {

    /* Η συνάρτηση setOf δημιουργεί ένα νέο σύνολο από
       συμβολοσειρές (λέξεις), το οποίο διατίθεται μόνο
       για ανάγνωση (αμετάβλητο). */
    val words = setOf("pen", "cup", "dog", "person", "cement",
                      "coal", "spectacles", "cup", "bread")

    /* Η συνάρτηση elementAt ανακτά το στοιχείο του συνόλου με
       δείκτη 0 (pen). Πιο συγκεκριμένα παίρνει το στοιχείο του
       συνόλου με δείκτη 0 για να το ανακτήσει ως παράμετρο. */
    val w1 = words.elementAt(0)

    // Εκτυπώνει: pen
    println(w1)

    /* Η συνάρτηση indexOf επιστρέφει τον δείκτη,
       στον οποίο εμφανίστηκε για πρώτη φορά η λέξη cup. */
    val i1 = words.indexOf("cup")

    /* Η συνάρτηση lastIndexOf επιστρέφει τον δείκτη,
       στον οποίο εμφανίστηκε για τελευταία φορά η λέξη cup. */
    val i2 = words.lastIndexOf("cup")

    // Εκτυπώνει: The first index of cup is 1
    println("The first index of cup is $i1")

    // Εκτυπώνει: The last index of cup is
    println("The last index of cup is $i2")
}
```

Μετρητής Συνόλου (Set Count):

Η συνάρτηση count επιστρέφει τον αριθμό των στοιχείων του συνόλου. Στο παρακάτω παράδειγμα, υπολογίζει τον αριθμό τον στοιχείων του συνόλου, τον αριθμό των στοιχείων με αρνητικές τιμές και τον αριθμό των στοιχείων με άρτιο περιεχόμενο.

Code Example

```
fun main() {

    /* Η συνάρτηση setOf δημιουργεί ένα νέο σύνολο από
       αριθμούς, το οποίο διατίθεται μόνο για ανάγνωση. */
    val nums = setOf(4, 5, 3, 2, 1, -1, 7, 6, -8, 9, -12)

    /* Η συνάρτηση count υπολογίζει τον αριθμό των στοιχείων
       του συνόλου. */
    val len = nums.count()

    /* Η ιδιότητα size καθορίζει τον αριθμό των στοιχείων του
       συνόλου. */
    val size = nums.size

    /* Η συνάρτηση count μπορεί να πάρει μία προκαθορισμένη
       συνάρτηση ως παράμετρο. Επιστρέφει true για τιμές οι
       οποίες είναι χαμηλότερες από μηδέν. */
    val n1 = nums.count { e -> e < 0 }

    /* Η συνάρτηση count υπολογίζει τον αριθμό των στοιχείων
       του συνόλου, που έχουν άρτιο περιεχόμενο. */
    val n2 = nums.count { e -> e % 2 == 0 }

    // Εκτυπώνει: There are 11 elements
    println("There are $len elements")

    // Εκτυπώνει: The size of the set is 11
    println("The size of the set is $size")

    // Εκτυπώνει: There are 3 negative values
    println("There are $n1 negative values")

    // Εκτυπώνει: There are 5 even values
    println("There are $n2 even values")
}
```

Πρώτο και Τελευταίο Στοιχείο ενός Συνόλου (First and Last):

Η συναρτήση first χρησιμοποιείται για την ανάκτηση του πρώτου στοιχείου ενός συνόλου και η συνάρτηση last για το τελευταίο στοιχείο του. Στο παράδειγμα που δίδεται παρακάτω, δημιουργεί ένα σύνολο από συμβολοσειρές και υπολογίζει το πρώτο και το τελευταίο στοιχείο.

Code Example

```
fun main() {

    /* Η συνάρτηση setOf δημιουργεί ένα νέο σύνολο από
       συμβολοσειρές (λέξεις), το οποίο διατίθεται μόνο
       για ανάγνωση (αμετάβλητο). */
    val words = setOf("pen", "cup", "dog", "person",
                      "cement", "coal", "donkey", "spectacles")

    // Η συνάρτηση first παίρνει το πρώτο στοιχείο του συνόλου.
    val w1 = words.first()

    /* Η συνάρτηση last παίρνει το τελευταίο στοιχείο του
       συνόλου. */
    val w2 = words.last()

    /* Η συνάρτηση findlast ανακτά το τελευταίο στοιχείο του
       συνόλου που ξεκινά με το γράμμα 'd' (donkey). */
    val w3 = words.findLast { w -> w.startsWith('d') }

    /* Η συνάρτηση first ανακτά το πρώτο στοιχείο του συνόλου
       που ξεκινά με το γράμμα 'd' (dog). */
    val w4 = words.first { w -> w.startsWith('d') }

    // Εκτυπώνει: pen
    println(w1)

    // Εκτυπώνει: spectacles
    println(w2)

    // Εκτυπώνει: donkey
    println(w3)

    // Εκτυπώνει: dog
    println(w4)
}
```

Ένωση Συνόλων (Set Union):

Η συναρτήση union επιστρέφει ένα σύνολο που περιέχει όλα τα διακριτά στοιχεία και από τις δύο συλλογές συνόλων. Στο παρακάτω παράδειγμα, δημιουργεί δύο σύνολα από ακέραιους αριθμούς και ενώνει τα σύνολα με τη συνάρτηση union.

Code Example

```
fun main() {

    val nums = setOf(1, 2, 3)
    val nums2 = setOf(3, 4, 5)

    val nums3 = nums.union(nums2)

    // Εκτυπώνει: [1, 2, 3, 4, 5]
    println(nums3)
}
```

Διάφορες Συναρτήσεις Συνόλων:

Τα σύνολα, όπως και οι λίστες, διαθέτουν και κάποιες άλλες ενσωματωμένες συναρτήσεις, που είναι οι ακόλουθες: iterator (συνάρτηση μετάβασης των στοιχείων ενός συνόλου σε ένα άλλο, ένα προς ένα), sorted και sortedDescending (συνάρτησεις ταξινόμησης των στοιχείων ενός συνόλου σε αύξουσα και φθίνουσα σειρά), reversed (συνάρτηση αντιστροφής των στοιχείων ενός συνόλου), contains (συνάρτηση ελέγχου κάποιας προκαθορισμένης τιμής), max (συνάρτηση εύρεσης του μέγιστου στοιχείου ενός συνόλου), filter και filterNot (συνάρτησεις στις οποίες περνούν μόνο ορισμένα στοιχεία ενός συνόλου που πληρούν ή όχι ορισμένα κριτήρια), map (διαδικασία μετασχηματισμού), reduce (διαδικασία αναγωγής), fold (διαδικασία αναδίπλωσης), chunked, partition (συνάρτηση διαμερισμού), groupBy (συνάρτηση ομαδοποιήσης), any, all, drop (συνάρτηση αποκλεισμού), take.

4.5 Maps

Τα maps είναι συλλογές που χρησιμοποιούν ζεύγη κλειδιών – τιμών. Μπορεί να υπάρχουν και δύο κλειδιά που αναφέρονται στο ίδιο αντικείμενο, αλλά δεν επιτρέπονται τα αντίγραφα των κλειδιών. Γιατί σε ένα map τα κλειδιά είναι μοναδικά και δεν μπορούν να αντιγραφούν. Η Kotlin διαθέτει maps μόνο για ανάγνωση (read-only maps) και μεταβλητά maps (mutable maps). Η συνάρτηση mapOf δημιουργεί maps μόνο για ανάγνωση και μπορούν να μεταβληθούν με τη συνάρτηση mutableMapOf.

Code Examples

```
fun main() {

    /* Η συνάρτηση mapOf δημιουργεί ένα αμετάβλητο map
       των ζευγαριών κλειδιών – τιμών, που παρέχονται ως
       παράμετροι. */
    val person = mapOf("name" to "Sokratis", "age" to "29")

    // Εκτυπώνει: {name=Sokratis, age=29}
    println("$person")
}
----------------------------------------------------------------

fun main() {

    /* Η συνάρτηση mutableMapOf δημιουργεί ένα μεταβλητό map,
       ο οποίος μπορεί να τροποποιηθεί. */
    val person2 = mutableMapOf("name" to "Anna", "age" to "26")

    // Εκτυπώνει: {name=Anna, age=26}
    println("$person2")

    // Η συνάρτηση put τοποθετεί μία νέα τιμή στο map.
    person2.put("location", "Greece")

    // Εκτυπώνει: {name=Anna, age=26, location=Greece}
    println("$person2")

    // Η συνάρτηση remove διαγράφει μία τιμή από το map.
    person2.remove("age")

    // Εκτυπώνει: {name= Anna, location=Greece}
    println("$person2")
}
```

Code Example

```
fun main() {

    val person3 = mutableMapOf("name" to "Kate", "age" to "30")

    // Εκτυπώνει: {name=Kate, age=30}
    println("$person3")

    // Η συνάρτηση filter φιλτράρει το map μόνο με το όνομα.
    val onlyName = person3.filter { it.key == "name" }

    // Εκτυπώνει: {name=Kate}
    println("$onlyName")

    // Η συνάρτηση clear διαγράφει όλα τις τιμές από το map.
    person3.clear()

    // Εκτυπώνει: {}
    println("$person3")
}
```

Βασικές συναρτήσεις χαρτών (Maps basic operations):

Τα maps διαθέτουν ορισμένες βασικές συναρτήσεις: sortedMapOf, hashMapOf και likedMapOf. Πιο συγκεκριμένα οι συναρτήσεις αυτές χρησιμοποιούνται για ταξινόμηση των ζευγαριών τιμών – κλειδιών των maps και για κάποιες εφαρμογές κατακερματισμού.

Code Example

```
fun main() {
    val person4 = mapOf("name" to "Theodore", "age" to "25",
                                    "location" to "Greece")

    // Εκτυπώνει: {name=Theodore, age=25, location=Greece}
    println("$person4")

    // Η συνάρτηση sortedMapOf ταξινομεί τις τιμές του map.
    val person5 = sortedMapOf("name" to "Theodore", "age" to
                                  "25", "location" to "Greece")

    // Εκτυπώνει: {age=25, location=Greece, name=Theodore}
    println("$person5")
}
```

Code Examples

```
fun main() {

    /* Η συνάρτηση hashMapOf χρησιμοποιείται για την δημιουργία
       ενός χάρτη. */
    val person6 = hashMapOf("name" to "Peter", "age" to "31",
                                        "location" to "Greece")

    // Εκτυπώνει: {name=Peter, age=31, location=Greece}
    println("$person6")

    // Εναλλακτικός τρόπος δημιουργίας ενός χάρτη.
    val person7 = linkedMapOf("name" to "Dimitris", "age" to
                                    "33", "location" to "Greece")

    // Εκτυπώνει: {name=Dimitris, age=33, location=Greece}
    println("$person7")
}
```

5. ΔΙΑΦΟΡΕΣ JAVA ΚΑΙ KOTLIN

5.1 Θεμελιώδης διαφορές Java και Kotlin

Η Java είναι μια γλώσσα αντικειμενοστραφούς προγραμματισμού (OOP) που τέθηκε σε χρήση το 1995. Αναπτύχθηκε από την εταιρία SUN Microsystems, την οποία αργότερα απέκτησε η εταιρία Oracle. Τα προγράμματα ή οι εφαρμογές που αναπτύσσονται στην Java, εκτελούνται σε μία εικονική της μηχανή (JVM), με την οποία μπορούν να εκτελεστούν τα ίδια πρόγραμμα σε πολλαπλές πλατφόρμες και συστήματα. Η Java χρησιμοποιείται ως επί το πλείστον για αυτόνομες εφαρμογές ή για back-end ανάπτυξη. Σχεδιάστηκε από τον James Gosling και η κύρια εφαρμογή της ήταν το OpenJDK. Είναι η κύρια επιλογή για τους περισσότερους προγραμματιστές όταν πρόκειται για την ανάπτυξη εφαρμογών Android, καθώς το ίδιο το Android είναι γραμμένο σε Java.

Η Kotlin είναι μια νέα γλώσσα προγραμματισμού που αναπτύχθηκε από προγραμματιστές από το IDE της εταιρίας Jet Brains. Εμφανιστήστηκε για πρώτη φορά το 2011 και η επίσημη κυκλοφορία της ήταν το 2016. Είναι μια γλώσσα ανοιχτού κώδικα. Επίσης είναι μια στατική γλώσσα προγραμματισμού, όπως η Java, C και C ++. Οι στατικώς πληκτρολογημένες γλώσσες προγραμματισμού είναι εκείνες στις οποίες οι μεταβλητές δεν χρειάζεται να οριστούν πριν χρησιμοποιηθούν. Η Kotlin βασίζεται σε JVM (Java Virtual Machine) αλλά μπορεί να μεταγλωττιστεί σε JavaScript, Android, Native και σε iOS. Είναι μια καλή επιλογή για την ανάπτυξη εφαρμογών διακομιστή, γιατί επιτρέπει στους χρήστες να γράφουν συνοπτικό και εκφραστικό κώδικα. Είναι πλήρως συμβατή με τις υπάρχουσες στοίβες της Java με μια ομαλή καμπύλη μάθησης. Η μετάβαση από την Java στη Kotlin είναι πολύ εύκολη καθώς χρειάζεται να εγκαταστήθεί μόνο ένα Plugin. Η εταιρεία Google χαρακτήρισε την Kotlin, ως την επίσημη γλώσσα προγραμματισμού για την ανάπτυξη εφαρμογών Android.

Η Kotlin επιδιορθώνει μια σειρά ζητημάτων από τα οποία υποφέρει η Java, όπως:

- ✓ Οι null αναφορές ελέγχονται από τον τύπο του συστήματος.
- ✓ Δεν υπάρχουν ακατέργαστοι τύποι δεδομένων.
- ✓ Παρέχει κατάλληλους τύπους συναρτήσεων.
- ✓ Οι πίνακες είναι αμετάβλητοι.

Παρουσίαση χαρακτηριστικών της Γλώσσας Προγραμματισμού Kotlin.

Ο πίνακας που δίνεται παρακάτω παρουσιάζει τις θεμελιώδεις διαφορές μεταξύ της γλώσσας Java και της Kotlin.

	Java	**Kotlin**
Checked Exceptions	✓	
Companion Objects		✓
Coroutines		✓
Data Classes		✓
Declaration-Site Variance		✓
Extension Functions		✓
First-Class Delegation		✓
Inline Functions		✓
Lambda Expressions		✓
Mutable Collections		✓
Non-Private Fields	✓	
Null-Safety		✓
Operator Overloading		✓
Primitive Types (that are not classes)	✓	
Primary Constructors		✓
Properties		✓
Range Expressions		✓
Separate Interfaces (only for read)		✓
Static Members	✓	
Singletons		✓
Smart Casts		✓
String Templates		✓
Ternary-Operator	✓	
Type Inference (for variables)		✓
Wildcard-Types	✓	

5.2 Διαφορές Java και Kotlin ως προς τα Βασικά τους

print and println:

Το παράδειγμα που δίνεται παρακάτω εκτυπώνει στην κονσόλα του υπολογιστή το μήνυμα: "Hello World!", το οποίο είναι γραμμένο σε Java και στη συνέχεια σε Kotlin.

Code Example

```java
// Java version:
public class hello {
     public static void main(String[] args) {
          System.out.print("Hello World!");
          System.out.println("Hello World!");
     }
 }
```

```kotlin
// Kotlin version:
fun main(){
     print("Hello World!")
     println("Hello World!")
}
```

Το ίδιο πρόγραμμα απαιτεί δύο λιγότερες γραμμές κώδικα στην παραλλαγή Kotlin και δεν απαιτεί να ολοκληρωθεί μέσα σε μία κλάση. Η εκτύπωση στην κονσόλα δεν χρειάζεται πλέον το System.out.print ή System.out.println και στη συνέχεια ένα ερωτηματικό (;) στο τέλος της γραμμής από την Java, αλλά μπορεί να πραγματοποιηθεί με ένα πιο απλό print ή println στη Kotlin.

int, final int, var and val:

Οι τιμές των μεταβλητών μπορεί να είναι είτε μεταβλητές είτε αμετάβλητες, ανάλογα με το εάν η τιμή τους μπορεί να μεταβάλλεται ή όχι. Εάν η τιμή μιας μεταβλητής μεταβάλλεται: χρησιμοποιείται η λέξη int, εάν πρόκειται για ακέραιο τύπο δεδομένων στη Java, ενώ στη Kotlin η λέξη var. Εάν η τιμή μιας μεταβλητής δεν μεταβάλλεται: χρησιμοποιείται η λέξη final στη Java, ενώ στη Kotlin η λέξη val.

Code Example

```
// Java version:
int w;
int z = 2;
z = 3;
w = 1;

final int x;
final int y = 1;
```

```
// Kotlin version:
var w: Int
var z = 2
z = 3
w = 1

val x: Int
val y = 1
```

Παρατηρείται ότι η Kotlin συνάγει τύπους δεδομένων με μεταβλητές. Αυτό σημαίνει ότι ο μεταγλωττιστής (compiler) θα αποφασίσει ποιος θα είναι ο τύπος δεδομένων της μεταβλητής κατά την εκτέλεση του προγράμματος, με βάση τα δεδομένα που έχουν εκχωρηθεί στη μεταβλητή.

Strings:

Στο επόμενο παράδειγμα δημιουργούνται δύο συμβολοσειρές με το όνομα και το επίθετο του χρήστη, στις γλώσσες προγραμματισμού Java και Kotlin.

Code Example

```
// Java version:
String name = "Sokratis";
String lastName = "Arvanitis";
String text = "My name is:" + name + " " + lastName;
String otherText = "My name is:" + name.substring(2);
```

```
// Kotlin version:
val name = "Sokratis"
val lastName = "Arvanitis"
val text = "My name is: $name $lastName"
val otherText = "My name is: ${name.substring(2)}"
```

Παρατηρείται ότι η εκτύπωση στη Java ακολουθείται από το πρόθεμα της πρόσθεσης (+) και στη συνέχεια το όνομα της μεταβλητής, ενώ στη Kotlin ακολουθείται από το πρόθεμα του δολαρίου ($) και στη συνέχεια το όνομα της μεταβλητής.

Bits Operations:

Το παρακάτω παράδειγμα εκτελεί κάποιες δυαδικές πράξεις, πρώτα στη Java και έπειτα στη Kotlin.

Code Example

```java
// Java version:
final int andResult  = a & b;
final int orResult   = a | b;
final int xorResult  = a ^ b;
final int rightShift = a >> 2;
final int leftShift  = a << 2;
```

```kotlin
// Kotlin version:
val andResult  = a and b
val orResult   = a or b
val xorResult  = a xor b
val rightShift = a shr 2
val leftShift  = a shl 2
```

Ternary Operator:

Η Java διαθέτει ternary operator, αντί αυτού η Kotlin χρησιμοποιεί την παραδοσική χρήση του if … else.

Code Example

```java
// Java 8 version:
String text = x > 5 ? "x > 5" : "x <= 5";
```

```kotlin
// Kotlin version:
val text = if    (x > 5)
               "x > 5"
           else "x <= 5"
```

Smart Cast:

Η Kotlin έχει την υποστήριξη του έξυπνου cast, το οποίο προσδιορίζει αμετάβλητους τύπους δεδομένων και εκτελεί σιωπηρά το cast από τον μεταγλωττιστή, ενώ στην Java πρέπει να προσδιορίστει και να εκτελέστει το cast.

Code Example

```
// Java 8 version:
if (a instanceof String) {
     final String result = ((String) a).substring(1);
}
```

```
// Kotlin version:
if (a is String) {
  val result = a.substring(1)
}
```

Null Safety:

Στη Java μπορούν να ορίστουν μεταβλητές με τον τύπο δεδομένων τους, ακολουθούμενες από το όνομα τους και είτε παίρνουν μια τιμή είτε είναι μηδενικές. Το ερωτηματικό χρησιμοποιείται για να τερματίσει τη γραμμή κώδικα. Στη Kotlin δηλώνεται ότι δημιουργείται μια μεταβλητή και στη συνέχεια το όνομα της μεταβλητής, ακολουθούμενη από μια τιμή ή τον τύπο δεδομένων και το ερωτηματικό χρησιμοποιείται για null.

Code Example

```
// Java version:
int number = 0;
-----------------------------

final String name = null;

String lastName;
lastName = null;
-----------------------------

if (text != null) {
  int length = text.length();}
```

```
// Kotlin version:
var number = 0
var nullNumber: Int?
-----------------------------

val name: String? = null

var lastName: String?
lastName = null
-----------------------------

val length = text?.length
val length = text!!.length
```

Παρατηρείται ότι στην Java μπορούν να ορίστουν μηδενικές τιμές, αλλά όταν προσπαθεί να έχει πρόσβαση σε αντικείμενα που δείχνουν σε τιμές null δημιουργεί μια εξαίρεση. Ενώ στη Kotlin, δεν εκχωρούνται μηδενικές τιμές σε μεταβλητές ή τιμές επιστροφής.

switch και when:

Η Java διαθέτει τη συνθήκη switch για τον έλεγχο κάποιων περιπτώσεων, ενώ η Kotlin διαθέτει τη συνθήκη when.

Code Example

```java
// Java version:
final int x =      // value;
final String xResult;

switch (x) {
  case 0:
  case 11:
    xResult = "0 or 11";
    break;
  case 1:
  case 2:
    //...
  case 10:
    xResult = "from 1...10";
    break;
  default:
    if (x < 12 && x > 14) {
    xResult = "not 12...14";
      break;
    }
    if (isOdd(x)) {
      xResult = "is odd";
      break;
    }
    xResult = "otherwise";
}
```

```kotlin
// Kotlin version:
val x =      // value


val xResult = when (x) {
  0, 11 -> "0 or 11"
  in 1..10 -> "from 1...10"
  !in 12..14 ->"not 12...14"
  else->if (isOdd(x)){"isodd"}
  else {"otherwise"}
}
```

Παρατηρείται ότι ο ίδιος κώδικας γραμμένος στην γλώσσα προγραμματισμού Kotlin απαιτεί πολύ λιγότερες γραμμές κώδικα.

Επαναληπτικός Βρόχος for:

Στο παράδειγμα που δίνεται εν συνεχεία, παρουσιάζονται έξι βρόχοι επανάληψης for, στη Java και κατόπιν στη Kotlin.

Code Example

```
// Java version:
for (int i=1; i<11; i++) { }

for (int i=11; i<21; i++) { }

for (int i=1; i<11; i+=2) { }

for (int i=11; i<21; i+=4) { }

for (String item: collection) { }

for (Map.Entry<String, String>entry: map.entrySet()) { }
```

```
// Kotlin version:
for (i in 1 until 11) { }

for (i in 11 until 21) { }

for (i in 1..10 step 2) { }

for (i in 11..21 step 4) { }

for (item in collection) { }
for ((index, item) in collection.withIndex()) { }

for ((key, value) in map) { }
```

5.3 Διαφορές Java και Kotlin ως προς τις Συναρτήσεις

Η Java δεν διαθέτει υποστήριξη συναρτησιακού προγραμματισμού έως την έκδοση Java 8. Η Kotlin είναι ένα μείγμα διαδικαστικής και συναρτησιακής γλώσσας προγραμματισμού που αποτελείται από πολλές χρήσιμες μεθόδους, όπως είναι οι λάμδα, οι συναρτήσεις υψηλότερης τάξης.

Συναρτήσεις:

Το παράδειγμα που δίνεται παρακάτω παρουσιάζει συναρτήσεις οι οποίες δεν δέχονται καμία παράμετρο και δεν έχουν τύπο επιστροφής.

Code Example

```
// Java version:
public void hello() {
    System.out.print("Hello World!");
}
```

```
// Kotlin version:
fun hello() {
    println("Hello World!")
}
```

Συνάρτηση Main:

Στο επόμενο παράδειγμα εμφανίζεται ο τρόπος με τον οποίο δηλώνεται η συνάρτηση main στις γλώσσες προγραμματισμού Java και Kotlin.

Code Example

```
// Java version:
public class MyClass {
    public static void main (String[] args){
    }
}
```

```
// Kotlin version:
fun main() { }
```

Επιστροφή Συνάρτησης:

Το παρακάτω παράδειγμα προβάλλει την διαδικασία με την οποία επιστρέφεται μία τιμή από μία συνάρτηση.

Code Example

```java
// Java version:
public boolean hasItems() {
    return true;
}
```

```kotlin
// Kotlin version:
fun hasItems(): Boolean {
    return true
}
```

Συναρτήσεις με Παραμέτρους και χωρίς Τύπο Επιστροφής:

Στο επόμενο παράδειγμα υποδηλώνονται συναρτήσεις οι οποίες δέχονται μία συμβολοσειρά (String) ως παράμετρο και δεν έχουν καμία τιμή επιστροφής, σε Java και κατόπιν σε Kotlin.

Code Example

```java
// Java version:
public void hello (String name) {

  System.out.print("Hello " + name + "!");
}
```

```kotlin
// Kotlin version:
fun hello(name: String) {

    println("Hello $name!")
}
```

Συναρτήσεις με Προεπιλεγμένα Ορίσματα:

Στο παράδειγμα παρακάτω, παρουσιάζονται συναρτήσεις οι οποίες έχουν προεπιλεγμένες τιμές στα ορίσματά τους.

Code Example

```
// Java version:
public void hello (String name) {
    if (name == null) {
        name = "World";
    }
    System.out.print("Hello " + name + "!");
}
```

```
// Kotlin version:
fun hello(name: String = "World") {
    println("Hello $name!")
}
```

- *Η έκδοση της Java 8 δεν υποστηρίζει συναρτήσεις με προεπιλεγμένες τιμές στα ορίσματα, ενώ η έκδοση της Java 10 υποστηρίζει.*

Συναρτήσεις σαν Μία Έκφραση:

Στο παράδειγμα που προβάλλεται παρακάτω, εμφανίζονται συναρτήσεις που συμπεριφέρονται σαν μία έκφραση.

Code Example

```
// Java version:
public double cube (double x) {
    return x * x * x;
}
```

```
// Kotlin version:
fun cube(x: Double) : Double = x * x * x
```

- *Οι συναρτήσεις της Java 8 δεν μπορούν να συμπεριφέρονται σαν εκφράσεις, ενώ οι συναρτήσεις της Java 10 μπορούν.*

Συναρτήσεις με Ονοματικά Ορίσματα:

Code Example

```
// Java version::
public static void main (String[]args){
    openFile("file.txt", true);
}
public static File openFile(String filename,boolean readOnly){}
```

```
// Kotlin version:
fun main() {
    openFile("file.txt", readOnly = true)
}
fun openFile(filename: String, readOnly: Boolean) : File { }
```

Γενικές Συναρτήσεις:

Στο παράδειγμα που δίνεται εν συνεχεία, παρουσιάζεται ο τρόπος με τον οποίο δημιουργούνται οι γενικές συναρτήσεις, στη Java και έπειτα στη Kotlin.

Code Example

```
// Java version:
public void init() {
    List<String> moduleInferred = createList("net");
}

public <T> List<T> createList(T item) { }
```

```
// Kotlin version:
fun init() {
    val module = createList<String>("net")
    val moduleInferred = createList("net")
}

fun <T> createList(item: T): List<T> { }
```

5.4 Διαφορές Java και Kotlin ως προς τις Κλάσεις

Κλάσεις:

Το παράδειγμα που δίνεται παρακάτω παρουσιάζει τον τρόπο με τον οποίο δηλώνονται οι κλάσεις στη Java και έπειτα στη Kotlin.

Code Examples

```java
// Java version:
public final class User { }
------------------------------

public class User { }
```

```kotlin
// Kotlin version:
class User
------------------------------

open class User
```

Κατασκευαστές (Constructors):

Στο επόμενο παράδειγμα εμφανίζεται η διαδικασία με την οποία δημιουργούνται οι κατασκευαστές.

Code Example

```java
// Java version:
final class User {
    private String name;

    public User(String name) {
        this.name = name;
    }
    public String getName() {
         return name;
    }
    public void setName(String name) {
         this.name = name;
    }
 }
```

```kotlin
// Kotlin version:
class User(var name: String)
```

Κατασκευαστές με Προεπιλεγμένα Ορίσματα:

Το παρακάτω παράδειγμα προβάλλει κατασκευαστές οι οποίοι δέχονται προεπιλεγμένα ορίσματα, σε Java και κατόπιν σε Kotlin.

Code Example

```
// Java version:
final class User {
     private String name;
     private String lastName;
     public User (String name) {
         this(name, "");
     }
     public User(String name, String lastName) {
         this.name = name;
         this.lastName = lastName;
     }
     // And Getters & Setters
 }
```

```
// Kotlin version:
class User(var name: String, var lastName: String = "")
```

Αμετάλλακτες Ιδιότητες (Final Attributes):

Code Example

```
// Java version:
final class User {
    private final String name;
    public User(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
 }
```

```
// Kotlin version:
class User (val name: String)
```

Ιδιότητες (Properties):

Στο παράδειγμα παρακάτω, παρουσιάζονται οι εμβέλειες των πεδίων - πρόσβαση, στη Java και έπειτα στη Kotlin.

Code Example

```java
// Java version:
public class Document {

    private String id = "00x";

     public String getId() {
         return id;
     }

     public void setId(String id) {

         if(id != null && !id.isEmpty()) {
             this.id = id;
         }
     }
 }
```

```kotlin
// Kotlin version:
class Document{

    var id : String = "00x"
        set(value) {

            if (value.isNotEmpty()) field = value
        }
}
```

Αφηρημένες Κλάσεις (Abstract Class):

Το επόμενο παράδειγμα προβάλλει τον τρόπο με τον οποίο δημιουργούνται οι αφηρημένες κλάσεις.

Code Example

```
// Java version:
public abstract class Document{

    public abstract int calculateSize();
}

public class Photo extends Document {

    @Override
    public int calculateSize() {
    }
}
```

```
// Kotlin version:
abstract class Document {

    abstract fun calculateSize(): Int
}

class Photo: Document() {

    override fun calculateSize(): Int {
    }
}
```

Διεπαφές (Interfaces):

Στο επόμενο παράδειγμα εμφανίζεται ο τρόπος με τον οποίο δημιουργούνται οι διεπαφές, στη Java και εν συνεχεία στη Kotlin.

Code Example

```java
// Java version:
public interface Printable {
    void print();
}

public class Document implements Printable {
    @Override
    public void print() { }
}
```

```kotlin
// Kotlin version:
interface Printable{
    fun print()
}

class Document: Printable{
    override fun print() { }
}
```

Ένθετες Κλάσεις (Nested Classes):

Το παρακάτω παράδειγμα δημιουργεί ένθετες κλάσεις σε Java και Kotlin.

Code Example

```java
// Java version:
public class Documment {

    public static class InnerClass{ }
}
```

```kotlin
// Kotlin version:
class Document {

    class InnerClass
}
```

Κλάσεις Δεδομένων (Data Classes):

Στο παράδειγμα που προβάλλεται παρακάτω, παρουσιάζεται ο τρόπος με τον οποίο δημιουργούνται οι κλάσεις δεδομένων στη Java και έπειτα στη Kotlin.

Code Example

```
// Java version:
public static void main (String[]args) {
    Book book = createBook();
    System.out.println(book);
    System.out.println("Title: " + book.title);
}
public static Book createBook(){
    return new Book("title_01", "author_01");
}
public class Book {
    final private String title;
    final private String author;
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
    public String getTitle() {
        return title;
    }
    public String getAuthor() {
        return author;
    }
    @Override
    public String toString() {
        return "Title: " + title + " Author: " + author; }
}
```

```
// Kotlin version:
fun main() {
    val book = createBook();
    println(book)
    println("Title: $title")
}
fun createBook( : Book{
    return Book("title_01", "author_01")
}
data class Book(val title: String, val author: String)
```

Επεκτάσεις Κλάσεων (Extension Classes):

Στη Java για να επεκταθεί η λειτουργικότητα της υπάρχουσας κλάσης, πρέπει να δημιουργήθεί μια νέα τάξη και να κληρονομήσει από την τάξη των γονέων. Επομένως, οι συναρτήσεις επέκτασης δεν είναι διαθέσιμες στην Java (από την έκδοση της Java 10 και μετά είναι διαθέσιμες οι επεκτάσεις). Ενώ η Kotlin παρέχει στους προγραμματιστές τη δυνατότητα να επεκτείνουν μια υπάρχουσα κλάση με νέες συναρτήσεις.

Code Example

```
// Java version:
public class ByteArrayUtils {
    public static String toHexString(byte[] data) {
    }
}

    final byte[] dummyData = new byte[10];
    final String hexValue =
                          ByteArrayUtils.toHexString(dummyData);
```

```
// Kotlin version:
fun ByteArray.toHex() : String {
}

val dummyData = byteArrayOf()
val hexValue = dummyData.toHex()
```

Εσωτερικές Κλάσεις (Inner Classes):

Στο παράδειγμα που εμφανίζεται παρακάτω, αναδεικύεται ο τρόπος με τον οποίο δημιουργούνται οι εσωτερικές κλάσεις.

Code Example

```
// Java version:
public class Documment {
    class InnerClass { }
}
```

```
// Kotlin version:
class Document {
    inner class InnerClass
}
```

Singleton:

Singleton ονομάζεται η κλάση η οποία έχει ένα μόνο στιγμιότυπο. Παρακάτω δίδεται ένα παράδειγμα Singleton στη Java και εν συνεχεία στη Kotlin,

Code Example

```
// Java version:
public class Document {
    private static final Document INSTANCE = new Document();

    public static Document getInstance() {
        return INSTANCE;
    }
}
```

```
// Kotlin version:
object Document { }
```

5.5 Διαφορές Java και Kotlin ως προς τις Συλλογές

Συλλογές (Collections):

Το παράδειγμα που δίνεται παρακάτω παρουσιάζει τον τρόπο με τον οποίο δημιουργούνται οι συλλογές στη Java και έπειτα στη Kotlin.

Code Example

```
// Java version:
for (int number: numbers) {
    System.out.println(number);
}

for (int number: numbers) {
    if (number>5) {
        System.out.println(number);
    }
}
```

```
// Kotlin version:
numbers.forEach {
    println(it)
}

numbers.filter  {it>5}
       .forEach {println(it)}
```

Λίστες (Lists):

Στα επόμενα παράδειγματα δημιουργούνται λίστες στις γλώσσες προγραμματισμού Java και Kotlin.

Code Example

```
// Java version:
final List<Integer> evens = new ArrayList<>();
final List<Integer> odds = new ArrayList<>();

for (int number: numbers) {

    if ((number & 1) == 0) {
        evens.add(number);
    }else {
        odds.add(number);
    }
    groups.get("odd").add(number);
}
```

```
// Kotlin version:
val (evens, odds) = numbers.partition {it and 1 == 0}
```

Code Example

```
// Java version:
final List<User> users = getUsers();

Collections.sort(users, new Comparator<User>() {

    public int compare(User user, User otherUser) {
        return user.lastname.compareTo(otherUser.lastname);
    }
});
```

```
// Kotlin version:
val users = getUsers()
users.sortedBy {it.lastname}
```

Maps:

Στα παράδειγματα που εμφανίζονται παρακάτω, παρουσιάζεται ο τρόπος με τον οποίο δημιουργούνται τα maps στη Java και κατόπιν στη Kotlin.

Code Example

```
// Java version:
final List<Integer> numbers = Arrays.asList(1, 2, 3);

final Map<Integer, String> map = new HashMap<Integer,String>();
map.put(1, "One");
map.put(2, "Two");
map.put(3, "Three");
```

```
// Kotlin version:
val numbers = listOf(1, 2, 3)

val map = mapOf(1 to "One",
                2 to "Two",
                3 to "Three")
```

Code Example

```
// Java version:
final Map<String, List<Integer>> groups = new HashMap<>();
for (int number : numbers) {
    if((number & 1) == 0){
      if(!groups.containsKey("even")) {
        groups.put("even", new ArrayList<>());
      }
      groups.get("even").add(number);
      continue;
    }
    if(!groups.containsKey("odd")) {
      groups.put("odd", new ArrayList<>());
    }
    groups.get("odd").add(number);
}
```

```
// Kotlin version:
val groups = numbers.groupBy {
                if (it and 1 == 0) "even" else "odd" }
```

ΑΚΡΩΝΥΜΙΑ

JVM	Java Virtual Machine
OP	Open Source
DP	Default Package
NPE	Null Pointer Exception
DS	Default Statement
DA	Default Argument
FP	Functional Programming
OOP	Object Oriented Programmimg
OpenJDK	Open Java Development Kit
IDE	Integrated Development Enviroment
iOS	iPhone Operating System

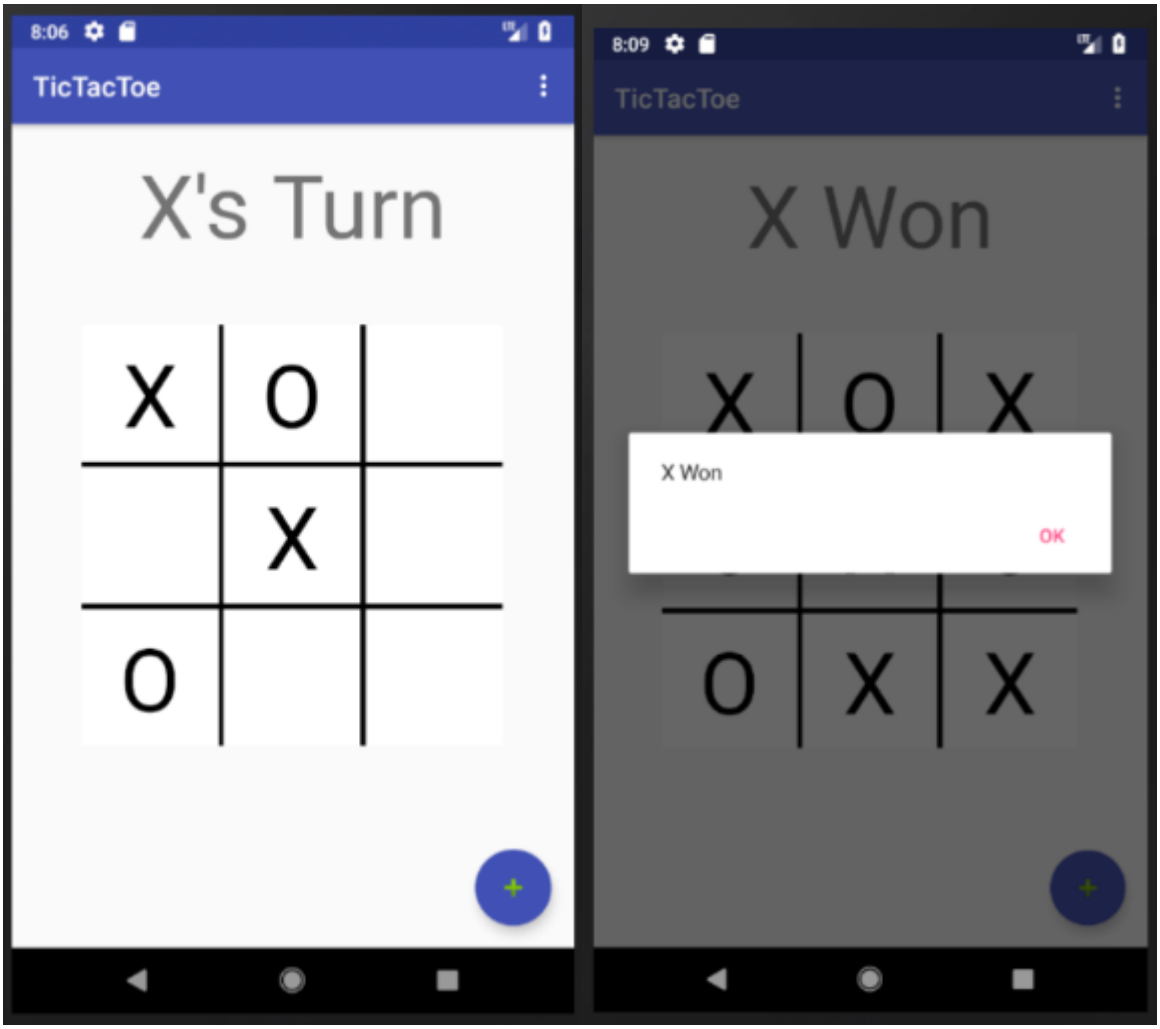
ΠΑΡΑΡΤΗΜΑ

Κώδικας:

Ο κώδικας της εφαρμογής Tic Tac Toe (ο οποίος είναι γραμμένος στη γλώσσα προγραμματισμού Kotlin σε Android Studio) βρίσκεται στην παρακάτω διεύθυνση: https://github.com/Swkraths/TicTacToe

Κονσόλα:

Παρακάτω προβάλλονται μερικές φωτογραφίες από την εφαρμογή Tic Tac Toe:



ΑΝΑΦΟΡΕΣ

1) Dmitry Jemerov and Svetlana Isakova, “Kotlin in Action”, December 2016

2) Stephen Samuel and Stefan Bocutiu, “Programming Kotlin”, *Familiarize yourself with all of Kotlin’s features with this in-depth guide*, January 2017

3) Antonio Leiva, “Kotlin for Android Developers”, *Learn Kotlin the easy way while developing an Android App*, June 2017

4) Marcin Moskala and Igor Wojda, “Android Development with Kotlin”, *Enhance your skills for Android development using Kotlin*, August 2017

5) Mike James, “Programmer's Guide to Kotlin”, September 2017

6) Miloš Vasić, “Mastering Android Development with Kotlin”, *Master the powerful Kotlin standard library through practical code examples*, November 2017

7) Ashish Belagali, Hardik Trivedi and Akshay Chordiya, “Kotlin Blueprints”, A practical guide to building industry-grade web, mobile, and desktop applications in Kotlin using frameworks such as Spring Boot and Node.js, December 2017

8) Rivu Chakraborty, “Reactive Programming in Kotlin”, *Design and build non-blocking, asynchronous Kotlin applications with RXKotlin, Reactor-Kotlin, Android, and Spring*, December 2017

9) Juan Antonio Medina Iglesias, “Hands-On Microservices With Kotlin”, *Build reactive and cloud-native microservices with Kotlin using Spring 5 and Spring Boot 2.0*, January 2018

10) Aanand Shekhar Roy and Rashi Karanpuria, “Kotlin Programming Cookbook”, *Explore more than 100 recipes that show how to build robust mobile and web applications with Kotlin, Spring Boot and Android*, January 2018

11) Pierre-Yves Saumont, “The Joy of Kotlin Version 8”, January 2018

12) Mario Arias and Rivu Chakraborty, “Functional Kotlin”, *Extend your OOP skills and implement functional techniques in Kotlin and Arrow*, February 2018

13) Iyanu Adelekan, “Kotlin Programming by Example”, *Build real-world Android and web applications the Kotlin way*, March 2018

14) Miloš Vasić, “Building Applications with Spring 5 and Kotlin”, *Build scalable and reactive applications with Spring, combined with the productivity of Kotlin*, May 2018

15) Alexey Soshin, “Hands-On Design Patterns with Kotlin”, *Build scalable applications using traditional, reactive, and concurrent design patterns in Kotlin*, June 2018

16) Eunice Adutwumwaa Obugyei and Natarajan Raman, “Learning Kotlin by building Android Applications”, *Explore the fundamentals of Kotlin by building real-world Android applications*, June 2018

17) Samuel Urbanowicz, “Kotlin Standard Library Cookbook”, *Master the powerful Kotlin standard library through practical code examples*, July 2018

18) Miguel Angel Castiblanco Torres, “Learning Concurrency in Kotlin”, *Build highly efficient and robust applications*, July 2018

19) David Greenhalgh and Josh Skeen, “Kotlin Programming First Edition”, *The Big Nerd Ranch Guide*, July 2018

20) Irina Galata, Joe Howard, Dick Lucas and Ellen Shapiro, “Kotlin Apprentice First Edition”, *Beginning Programming with Kotlin*, July 2018

21) Tom Blankenship and Darryl Bayliss, “Android Apprentice First Edition”, *Beginning Android Development with Kotlin 1.2*, July 2018

22) Miloš Vasić, “Fundamental Kotlin Second Edition”, Everything you need to know about Kotlin, August 2018

23) Peter Späth, “Pro Android with Kotlin”, *Developing Modern Mobile Apps*, September 2018

24) Abid Khan and Igor Kucherenko, “Hands-On Object-Oriented Programming with Kotlin”, *Build robust software with reusable code using OOP principles and design patterns in Kotlin*, October 2018

25) Dawn Griffiths and David Griffiths, “Head First Kotlin”, *A Brain-Friendly Guide, A learner’s guide to Kotlin programming*, February 2019

26) Rivu Chakraborty and Chandra Sekhar Nayak, “Hands-On Data Structures and Algorithms with Kotlin”, *Master Data Structure and Algorithms in Kotlin to develop high performance durable apps*, February 2019

27) Kotlin Tutorial with Example for Beginner, https://tutorialwing.com/kotlin-tutorial-example-beginner/

28) Kotlin Basics, https://www.tutorialsandyou.com/kotlin/kotlin-hello-world-9.html

29) Learn Kotlin, https://kotlinlang.org/docs/reference/

30) Kotlin Tutorial, https://www.javatpoint.com/kotlin-tutorial

31) Kotlin Hello World, Your First Kotlin Program, https://www.programiz.com/kotlin-programming/hello-world

32): Learn Kotlin by Example, https://play.kotlinlang.org/byExample/overview

33) The Programmers Guide to Kotlin - Arrays & Strings, https://www.i-programmer.info/programming/other-languages/10896-the-programmers-guide-to-kotlin-arrays-a-strings.html

34) Kotlin Functions to Create Lists, Maps and Sets, https://alvinalexander.com/kotlin/kotlin-functions-to-create-lists-maps-sets

35) Filtering Kotlin Collections, https://www.baeldung.com/kotlin-filter-collection

36) How to create Maps in Kotlin Using 5 Different Factory Functions, https://www.deadcoderising.com/how-to-create-maps-in-kotlin-using-5-different-factory-functions-2/

37) Kotlin – How to Loop a Map, https://www.mkyong.com/kotlin/kotlin-how-to-loop-a-map/

38) Algorithms in Kotlin and Binary Trees, https://developerlife.com/2018/08/16/algorithms-in-kotlin-6/#binary-trees

39) Some Major Differences Between Java and Kotlin, https://www.quora.com/What-are-some-major-differences-between-Java-and-Kotlin

40) Comparison to Java Programming Language, https://kotlinlang.org/docs/reference/comparison-to-java.html

41) Differences Between Java vs Kotlin, https://www.educba.com/java-vs-kotlin/

42) Kotlin vs Java, What is the Difference, http://androiddeveloper.galileo.edu/2017/10/16/kotlin-vs-java-what-is-the-difference/

43) Java 8 Stream API Analogies in Kotlin, https://www.baeldung.com/java-8-stream-vs-kotlin

44) Java vs Kotlin: It's Time to Expand Android Development, https://hackernoon.com/java-vs-kotlin-its-time-to-expand-android-development-f08e3d6a72b6

45) Kotlin vs Java: Which One You Should Choose for Your Next Android App, https://www.netguru.com/blog/kotlin-java-which-one-you-should-choose-for-your-next-android-app

46) Java vs Kotlin: Which is the Better Option for Android App Development, https://clearbridgemobile.com/java-vs-kotlin-which-is-the-better-option-for-android-app-development/

47) Kotlin vs Java: Key Differences Between Android's Officially-Supported Languages, https://www.androidauthority.com/kotlin-vs-java-783187/

48) Why I Refuse to code Android apps in Kotlin - https://hackernoon.com/why-i-refuse-to-code-android-apps-in-kotlin-1046edc455f2

49) Overview of Kotlin and Comparison Between Kotlin and Java, https://www.xenonstack.com/blog/overview-kotlin-comparison-kotlin-java/

50) Android Developer Portal with Tools, Libraries, and Apps, https://android-arsenal.com/tag/205?category=3

51) Get Started with Kotlin on Android, https://developer.android.com/kotlin/get-started?gclid=CjwKCAjw1dzkBRBWEiwAROVDLFSUpluup9HBikzEEE9eRH7BEHAdCM15DHR_GC52y3UTzX0R9-hpgxoCwicQAvD_BwE

52) Create Your First Kotlin Project, https://www.linkedin.com/learning/kotlin-essential-training/create-your-first-kotlin-project

53) Build Your First Android App in Kotlin, https://codelabs.developers.google.com/codelabs/build-your-first-android-app-kotlin/index.html#0

54) 10 Android Apps Written in Kotlin - Examples of Successful Applications, https://www.netguru.com/blog/10-android-apps-written-in-kotlin