ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΔΥΤΙΚΗΣ ΕΛΛΑΔΑΣ

ΣΧΟΛΗ ΔΙΟΙΚΗΣΗ ΚΑΙ ΟΙΚΟΝΟΜΙΑΣ

ΤΜΗΜΑ ΔΙΟΙΚΗΣΗΣ ΕΠΙΧΕΙΡΗΣΕΩΝ(ΠΑΤΡΑ)

Οι νέες δυνατότητες της Java 8

Πτυχιακή εργασία των

Στάικου Ταξιάρχη

Παλλαδινού Νικολάου

Επιβλέπων : Στάμος Κωνσταντίνος

ΠΑΤΡΑ 10/7/2015

ΕΥΧΑΡΙΣΤΙΕΣ

Ευχαριστούμε πολύ τον επιβλέποντα καθηγητή μας για την πολύτιμη βοήθεια του και την καθοδήγηση του κατά την διάρκεια εκπόνησης της πτυχιακής μας εργασίας

Επιπλέον θα θέλαμε να ευχαριστήσουμε τις οικογένειες μας και τους φίλους μας για την κατανόηση και την υποστήριξη τους όλο αυτό το χρονικό διάστημα.

ΠΕΡΙΛΗΨΗ

Σκοπός της παρούσας εργασίας είναι η παρουσίαση των νέων χαρακτηριστικών της Java 8 όπου κυρίως επικεντρωθήκαμε στις Λάμδα εκφράσεις.

Όπως αναφέραμε ήδη μία λάμδα έκφραση είναι ένα ανώνυμο μπλοκ κώδικα (ή αλλιώς μία ανώνυμη συνάρτηση) με μία λίστα από προτυποποιημένες παραμέτρους και ένα σώμα. Μία λάμδα έκφραση παρέχει ένα συνοπτικό τρόπο, συγκρινόμενο με τις ανώνυμες εσωτερικές κλάσεις, για τη δημιουργία ενός στιγμιοτύπου συναρτησιακών διεπαφών. Οι λάμδα εκφράσεις και οι προκαθορισμένες μέθοδοι (default methods) είναι οι διεπαφές που έχουν δώσει μία νέα πνοή στην Java.

Η βιβλιοθήκη συλλογής της Java έχει επωφεληθεί περισσότερο από τις λάμδα εκφράσεις. Το συντακτικό για τον ορισμό των λάμδα εκφράσεων είναι παρόμοιο με τον ορισμό μιας μεθόδου. Μία λάμδα έκφραση μπορεί να έχει μία λίστα από προτυποποιημένες παραμέτρους και ένα σώμα. Η λάμδα έκφραση υπολογίζεται σε ένα στιγμιότυπο μίας λειτουργικής διεπαφής. Το σώμα της λάμδα έκφρασης δεν εκτελείται όταν υπολογίζεται η έκφραση. Το σώμα της λάμδα έκφρασης εκτελείται όταν η μέθοδος της λειτουργικής διεπαφής καλείται.

Ένας από τους σχεδιαστικούς στόχους των λάμδα εκφράσεων είναι να κρατηθούν συνοπτικές και αναγνώσιμες (readable). Το συντακτικό της λάμδα έκφρασης υποστηρίζει στενογραφίες για κοινές περιπτώσεις χρήστη. Οι αναφορές μεθόδων είναι συντομογραφίες, ενώ για να καθορίσουν τις λάμδα εκφράσεις που χρησιμοποιούν υπάρχουσες μεθόδους.

Επίσης μία πολύ-έκφραση είναι μία έκφραση της οποίας ο τύπος εξαρτάται από το περιεχόμενο όπου χρησιμοποιείται. Μία λάμδα έκφραση είναι πάντα μία πολύ-έκφραση. Μία λάμδα έκφραση δε μπορεί να χρησιμοποιηθεί μόνη της. Ο τύπος της διερμηνεύεται από τον μεταφραστή με βάση το περιεχόμενο όπου βρίσκεται.

Μία έκφραση λάμδα μπορεί να χρησιμοποιηθεί σε αναθέσεις, κλήσεις μεθόδων, επιστροφές και casts. Όταν μία λάμδα έκφραση συμβαίνει μέσα σε μία μέθοδο, είναι λεξικολογικά ορισμένη. Αυτό σημαίνει ότι μια λάμδα έκφραση δεν ορίζει από μόνη της το «πεδίο δράσης της» αλλά συμβαίνει μέσα στο πεδίο δράσης μίας μεθόδου. Μία λάμδα έκφραση μπορεί να χρησιμοποιήσει αποτελεσματικά τις τοπικές μεταβλητές μίας μεθόδου καθώς και τις δηλώσεις:

- break,
- continue,
- return,
- και throw.

ΠΕΡΙΕΧΟΜΕΝΑ

ΕΙΣΑΓΩΓΗ

Η ανάπτυξη της Java ξεκίνησε το 1991, όταν η εταιρεία Sun Microsystems προσπαθούσε να βρει ένα εργαλείο που θα μπορούσε να χρησιμοποιηθεί ως πλατφόρμα ανάπτυξης λογισμικού για «έξυπνες» μικροσυσκευές. Οι αρχικοί πειραματισμοί έγιναν από τον James Gosling κάνοντας χρήση της C++ η οποία όμως δεν είχε τα επιθυμητά αποτελέσματα. Για το λόγο αυτό, ο Gosling αποφάσισε να πειραματιστεί με τη δημιουργία μιας νέας γλώσσας η οποία θα είχε ως βάση την C++ αλλά θα επεκτείνονταν και θα συμπληρώνονταν με τα απαραίτητα χαρακτηριστικά που απαιτούνταν για τον προγραμματισμό μικροσυσκευών.

Ύστερα από διάφορες προσπάθειες και μία σειρά ενδιάμεσων γλωσσών (C++ ++) κατέληξε στην Oak. Η συγκεκριμένη γλώσσα αν και διατηρούσε μεγάλη συγγένεια με τη C++ είχε πιο έντονο αντικειμενοστραφή χαρακτήρα και χαρακτηριζόταν από την απλότητά της. Το όνομα Oak όμως ήταν ήδη κατοχυρωμένο οπότε η καινούργια γλώσσα μετονομάστηκε σε Java.

Η επίσημη εμφάνιση της Java στη βιομηχανία της πληροφορικής έγινε το Μάρτιο του 1995 όταν η ***Sun*** την ανακοίνωσε στο συνέδριο ***Sun World*** 1995. Από εκείνη τη στιγμή και έως σήμερα, η Java παραμένει αυτό που στη βιομηχανία ονομάζεται de facto standard. Το 1998 παρουσιάζεται η έκδοση 2 της Java η οποία περιέχει αρκετές προσθήκες σε σχέση με την πρώτη έκδοση. Η επόμενη έκδοση της Java θα καθυστερήσει έξι χρόνια, αλλά το 2004 θα γίνει η επίσημη παρουσίαση της έκδοσης

ΚΕΦΑΛΑΙΟ 1

1.1 Η JAVA ΑΛΛΑΖΕΙ

Το 2006 λανσάρεται η Java 6 ενώ το ίδιο έτος (13/11/2006) γίνεται πλέον μια γλώσσα ανοιχτού κώδικα (GPL) όσον αφορά το μεταγλωττιστή (javac) και το πακέτο ανάπτυξης (JDK, Java Development Kit). Η Java 7 παρουσιάστηκε το 2011 η οποία ήταν και η τελευταία έκδοση μέχρι το 2014 όπου και έγινε διαθέσιμη στο κοινό η πιο πρόσφατη έκδοση της γλώσσας, - η οποία και αποτελεί το αντικείμενο της παρούσας εργασίας - Java 8.

Πίνακας 1: Ιστορικό εκδόσεων της Java

Έκδοση	Κωδική ονομασία	Ημερομηνία κυκλοφορίας
JDK 1.0	-	23-01-1996
JDK 1.1	-	19-02-1997
J_2SE 1.2	Playground	8-12-1998
J_2SE 1.3	Kestrel	8-05-2000
J_2SE 1.4	Merlin	6-02-2002
J_2SE 5.0	Tiger	30-09-2004
Java SE 6	Mustang	11-12-2006
Java SE 7	*Dolphin*	28-07-2011
Java SE 8	-	18-03-2014

Γιατί όμως η Java συνεχίζει όλα αυτά τα χρόνια να αλλάζει; Η απάντηση βρίσκεται στην άποψη που η επιστημονική κοινότητα έχει για τις γλώσσες προγραμματισμού. Οι επιστήμονες – ακαδημαϊκοί ύστερα από χρόνια ενασχόλησης με την επιστήμη της πληροφορικής κατέληξαν στο συμπέρασμα οι γλώσσες προγραμματισμού συμπεριφέρονται σαν ένα οικοσύστημα. Αυτό σημαίνει ότι συνεχώς νέες γλώσσες εμφανίζονται και οι παλαιότερες εκτοπίζονται και εξαφανίζονται εάν δεν προσαρμοστούν και δεν εξελιχθούν. Όλοι θα ήθελαν την δημιουργία μιας παγκόσμιας γλώσσας προγραμματισμού ικανής να αντιμετωπίσει οποιοδήποτε πρόβλημα σε οποιονδήποτε τομέα αλλά στην πραγματικότητα ορισμένες γλώσσες είναι καταλληλότερες από άλλες σε συγκεκριμένα ζητήματα. Αν και η μετάβαση σε μια νέα γλώσσα με καινούργια χαρακτηριστικά και προγραμματιστικές δυνατότητες δεν είναι εύκολη υπόθεση τελικά η νέα γλώσσα επικρατεί της παλαιότερης εκτός εάν η τελευταία καταφέρει να εξελιχτεί αρκετά γρήγορα και να συμβαδίσει με την γενικότερη εξέλιξη της τεχνολογίας. Η Java όμως, από την εμφάνιση της μέχρι σήμερα, έχει καταφέρει όχι απλά να επιβιώσει σε αυτό το

«οικοσύστημα», αλλά έχει επικρατήσει και έχει εκτοπίσει τους ανταγωνιστές της από ένα μεγάλο τμήμα της αγοράς που σχετίζεται με προγραμματιστικές εργασίες.

Κάποιοι από τους λόγους για τους οποίους η Java είναι τόσο επιτυχημένη και έχει μεγάλη απήχηση:

· Πρόκειται για μια καλά σχεδιασμένη αντικειμενοστραφή γλώσσα προγραμματισμού με πολλές χρήσιμες βιβλιοθήκες.

· Υποστηρίζει από την πρώτη κιόλας έκδοση της, μικρής κλίμακας συγχρονισμό με ολοκληρωμένη υποστήριξη νημάτων (threads) και κλειδαριών (locks).

· Η Java μεταγλωττίζεται σε έναν ενδιάμεσο κώδικα που ονομάζεται bytecode. Ο κώδικας αυτός εκτελείται σε μια εικονική μηχανή την JVM την οποία υποστηρίζουν όλοι οι περιηγητές (browsers). Αυτομάτως αυτό σημαίνει ότι αποτελεί την πρώτη επιλογή γλώσσας για τον προγραμματισμό διαδικτυακών μικροεφαρμογών (internet applets).

· Χρησιμοποιείται σε ποικίλα θέματα που σχετίζονται με ενσωματωμένα συστήματα πληροφορικής

Η τεχνολογία όμως αναπτύσσεται, προκύπτουν νέες ανάγκες και νέες δυνατότητες, οι προγραμματιστές αρχίζουν να ασχολούνται όλο και περισσότερο με σύνολα δεδομένων μεγέθους terabytes και πάνω, τα λεγόμενα «μεγάλα δεδομένα» (big data) και επιθυμούν να εκμεταλλευτούν την ύπαρξη υπολογιστών πολλών πυρήνων ή συμπλέγματα υπολογιστών (computing clusters) για την αποτελεσματική επεξεργασία τους. Αυτό απαιτεί τη χρήση παράλληλης επεξεργασίας, κάτι που παραδείγματος χάριν η Java δεν υποστήριζε παλαιότερε. Κοντολογίς οποία γλώσσα δεν θα καταφέρνει να προσαρμόζεται έγκαιρα στην εκάστοτε «αλλαγή κλίματος» θα εξαφανίζεται και από το «οικοσύστημα».

Το κύριο πλεονέκτημα της Java 8 σε έναν προγραμματιστή είναι ότι παρέχει περισσότερα εργαλεία προγραμματισμού και έννοιες (concepts) για να λύσει νέα ή ήδη υπάρχοντα προβλήματα προγραμματισμού πιο γρήγορα, με πιο περιεκτικό και πιο εύκολα διατηρήσιμο τρόπο.

Τρεις είναι οι βασικές έννοιες προγραμματισμού που οδήγησαν στην ανάπτυξη και την προσθήκη νέων χαρακτηριστικών στη Java 8 ώστε να εκμεταλλεύονται τον παραλληλισμό και να γράφουν πιο συνοπτικό κώδικα.

1. Stream επεξεργασία
2. Πέρασμα κώδικα σε μεθόδους με συμπεριφορά παραμετροποίησης (behavior parameterization)
3. Παραλληλισμός και κοινής χρήσης ευμετάβλητα δεδομένα

Οι κύριες αλλαγές στη Java 8 φανερώνουν μια προσπάθεια απομάκρυνσης από την κλασική αντικειμενοστραφή προσέγγιση, η οποία συχνά επικεντρώνεται απλά σε κάποιες αλλαγές τιμών, και προσανατολίζονται προς το φάσμα του προγραμματισμού

συναρτησιακού στυλ (functional-style programming) σύμφωνα με το οποίο αυτό που θέλει να επιτύχει ο προγραμματιστής (τι) θεωρείται πρωταρχικό και διαχωρίζεται από τον τρόπο (πως) με τον οποίο μπορεί να επιτευχτεί.

Στη συνέχεια παρουσιάζονται περιληπτικά, μια προς μια, οι νέες έννοιες της Java 8. Η αναλυτική παρουσίαση τους θα γίνει στα κεφάλαια που θα επακολουθήσουν.

1.2 ΣΥΝΑΡΤΗΣΕΙΣ ΣΤΗ JAVA

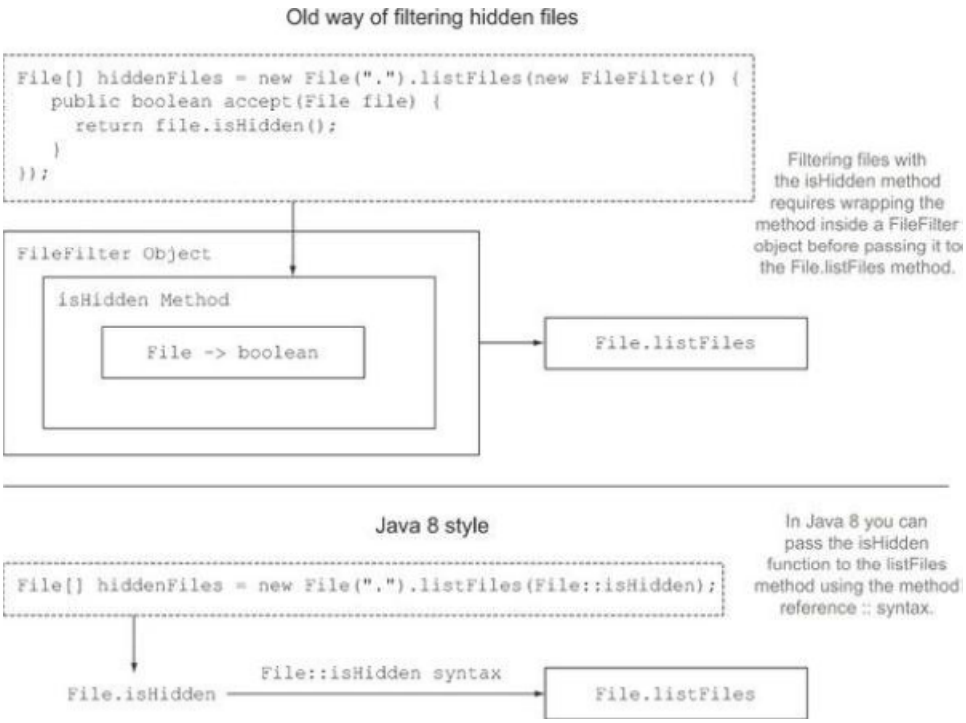
Η λέξη συνάρτηση (function) στις γλώσσες προγραμματισμού συχνά χρησιμοποιείται ως συνώνυμο για την μέθοδο (method) και συγκεκριμένα για την στατική μέθοδο (static method).

Η νέα έκδοση της Java, η Java 8 χρησιμοποιεί τις συναρτήσεις σαν να είναι κανονικές τιμές. Αυτές διευκολύνουν τη χρήση των steams (παρουσιάζονται παρακάτω), που η Java 8 παρέχει για τον παράλληλο προγραμματισμό για πολυπύρηνους επεξεργαστές.

Ο βασικός σκοπός μιας γλώσσας προγραμματισμού είναι η διαχείριση τιμών. Η κλασικές τιμές που αποτελούνται από αριθμούς (π.χ. 10 (int), 3.14 (double)) ονομάζονται τιμές πρώτης κατηγορίας (first-class values). Στις γλώσσες προγραμματισμού όμως μπορούν να υπάρξουν και κάποιες δομές (π.χ. μέθοδοι, κλάσεις) που ενώ βοηθούν στο να εκφραστούν οι δομές τιμών δεν μπορούν να περαστούν κατά την εκτέλεση του προγράμματος. Αυτές οι τιμές ονομάζονται τιμές δεύτερης κατηγορίας (second-class values). Αυτό που έκαναν οι σχεδιαστές της Java 8 ήταν να καταφέρουν να χρησιμοποιούν τις συναρτήσεις σαν να είναι τιμές πρώτης κατηγορίας. Αυτό έχει το πολύ μεγάλο κέρδος, ότι μπορούν οι συναρτήσεις να περνούν ως παράμετροι σε μεθόδους κατά την εκτέλεση του προγράμματος κάνοντας έτσι ευκολότερο το έργο του προγραμματιστή.

Το πρώτο καινούργιο χαρακτηριστικό της Java 8 είναι η χρήση των μεθόδων αναφοράς (method references).

Παράδειγμα:



Εικόνα 1: Περασμα της μεθοδου αναφορας File::isHidden στη μεθοδο listFiles[1]

Έχουμε ήδη την συνάρτηση isHidden διαθέσιμη και απλώς την περνάμε στην μέθοδο listFiles χρησιμοποιώντας την σύνταξη :: που σημαίνει «χρησιμοποίησε αυτή την μέθοδο σαν να είναι τιμή».

Με τον ίδιο τρόπο που οι μέθοδοι θεωρήθηκαν τιμές πρώτης κατηγορίες η Java 8 επιτρέπει με την ευρύτερη έννοια και οι συναρτήσεις να θεωρηθούν ως τιμές. Στις συναρτήσεις συμπεριλαμβάνονται και τα lambdas ή αλλιώς, ανώνυμες συναρτήσεις. Τα προγράμματα που χρησιμοποιούν αυτές τις έννοιες λέγεται ότι είναι γραμμένα με συναρτησιακού στυλ προγραμματισμό (functional-style programming) φράση που σημαίνει «συγγραφή προγραμμάτων που περνούν ως παραμέτρους συναρτήσεις ως τιμές πρώτης κατηγορίας.».

Το πέρασμα μεθόδων ως τιμές είναι σαφώς χρήσιμο, αλλά ίσως είναι χάσιμο χρόνου να πρέπει να γράφονται οι ορισμοί για σύντομες μεθόδους που θα χρησιμοποιηθούν μόνο μία ή δύο φορές σε όλο το πρόγραμμα. Προκειμένου να λύσει αυτό το πρόβλημα η Java 8 εισήγαγε μια νέα σημειογραφία τις ανώνυμες συναρτήσεις ή lambdas.

Η βασική σύνταξη ενός lambda είναι μια από τις παρακάτω:

· (*parameters*) ® *expression*
· (parameters) ® { statements; }

Έτσι δεν χρειάζεται κάποιος να γράψει έναν ορισμό μιας μεθόδου που χρησιμοποιείται μόνο μία φορά. Με αυτό τον τρόπο ο κώδικας είναι πιο ευκρινής και

με μεγαλύτερη σαφήνεια, επειδή δεν χρειάζεται να ψάξει να βρει ο προγραμματιστής μέσα στο πρόγραμμα το κομμάτι του κώδικα που περνά ως παράμετρο. Αλλά εάν μια τέτοια lambda έκφραση υπερβαίνει έναν αριθμό γραμμών και γίνεται περίπλοκο τότε είναι καλό να χρησιμοποιηθεί μια μέθοδος με συγκεκριμένο όνομα αντί για ένα ανώνυμο lambda.

Οι σχεδιαστές της Java 8 θα είχαν σταματήσει εδώ εάν δεν υπήρχαν οι πολυπύρηνοι επεξεργαστές. Προκειμένου να εκμεταλλευτεί τον παραλληλισμό, η Java 8 αντί να περιέχει μια ολόκληρη νέα Συλλογή, όπως την API που ονομάζεται Streams, περιέχει ένα ολοκληρωμένο σύνολο λειτουργιών που είναι παρόμοιο με το φιλτράρισμα με το οποίο η συναρτησιακού στυλ προγραμματιστές μπορεί να είναι εξοικειωμένοι, μαζί με τις μεθόδους για τη μετατροπή μεταξύ Συλλογών (Collections) και Streams.

1.3 ΕΙΣΑΓΩΓΗ ΣΤΑ STREAMS

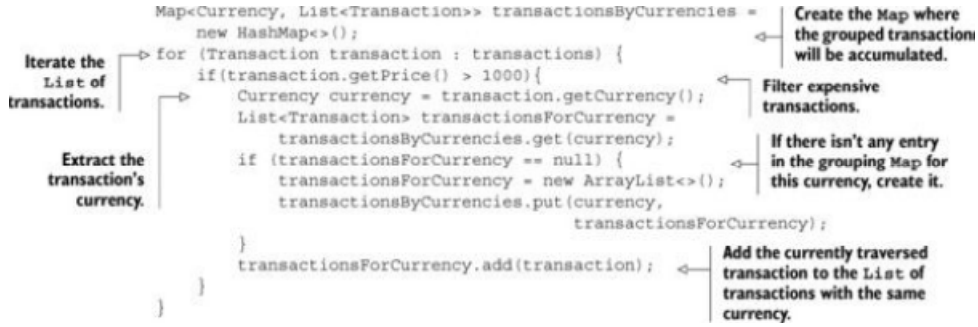
Ένα stream είναι μια ακολουθία αντικειμένων δεδομένων (data items) που παράγονται εννοιολογικά, ένα κάθε φορά. Ενα πρόγραμμα μπορεί να διαβάζει δεδομένα από ένα stream εισόδου, ένα προς ένα, όπως επίσης και να γράφει δεδομένα σε ένα stream εξόδου. Ένα stream εξόδου από ένα πρόγραμμα θα μπορούσε κάλλιστα να αποτελεί ένα stream εισόδου για κάποιο άλλο πρόγραμμα.

Η Java 8 πρόσθεσε τα Streams API στην βιβλιοθήκη java.util.stream. ένα Stream <T> είναι μια ακολουθία αντικειμένων τύπου T. Τα Streams API περιέχουν πολλές μεθόδους οι οποίες μπορούν να συνδεθούν διαδοχικά έτσι ώστε να σχηματίσουν μια σύνθετη σωλήνωση (complex pipeline). Με αυτό τον τρόπο δίνεται η δυνατότητα να προγραμματίσει κανείς στη Java 8 σε ένα υψηλότερο επίπεδο αφαίρεσης χρησιμοποιώντας και μετατρέποντας streams αντί για ένα στοιχείο την φορά.

Ένα άλλο πλεονέκτημα είναι ότι Java 8 μπορεί να τρέξει με διαφάνεια μια σωλήνωση λειτουργιών ενός Stream σε αρκετούς πυρήνες μιας CPU αρκει τα τμήματα της εισόδου να είναι ξένα μεταξύ τους. Επιτυγχάνεται έτσι γρήγορα και εύκολα ένας σχεδόν δωρεάν παραλληλισμός (parallelism almost for free), αντί της σκληρής δουλειάς που θα απαιτούσε η χρήση νημάτων (Threads).

Σχεδόν κάθε εφαρμογή Java κατασκευάζει και επεξεργάζεται συλλογές (collections). Όμως, η εργασία με συλλογές δεν είναι πάντα ιδανική, διότι παραδείγματος χάριν, για την υλοποίηση κάποιων ερωτημάτων επεξεργασίας δεδομένων μπορεί να χρειάζεται να γραφεί πολύς και πολύπλοκος κώδικας.

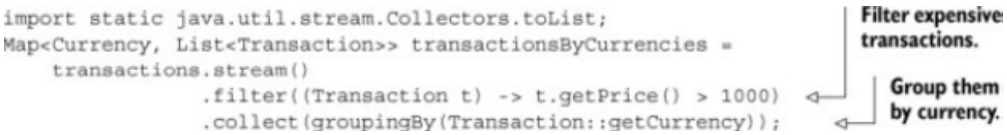
Παράδειγμα:



Εικόνα 2: Τμήμα κώδικα για φιλτράρισμα των ακριβών συναλλαγών από μια λίστα και ομαδοποίηση με βάση την αξία τους[1].

Αυτό αντιμετωπίζεται με την χρήση των Streams API. Ένα Streams API παρέχει έναν πολύ διαφορετικό τρόπο για την επεξεργασία δεδομένων σε σχέση με τις συλλογές API (Collections API).

Παράδειγμα:



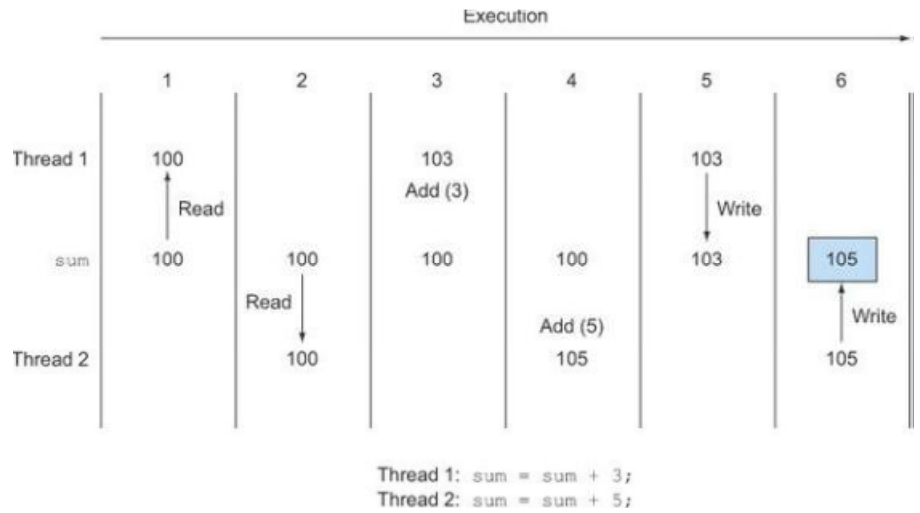
Εικόνα 3: Τμήμα κώδικα για φιλτράρισμα των ακριβών συναλλαγών από μια λίστα και ομαδοποίηση με βάση την αξία τους με χρήση του Streams API[1].

Στο συγκεκριμένο παράδειγμα, χρησιμοποιώντας μια συλλογή, ο προγραμματιστής διαχειρίζεται ο ίδιος την διαδικασία επανάληψης, Χρειάζεται να εκτελεστεί ένας for-each βρόχος για κάθε ένα στοιχείο ξεχωριστά και στη συνέχεια γίνεται η επεξεργασία των στοιχείων. Αυτός ο τρόπος επανάληψης πάνω στα δεδομένα ονομάζεται εξωτερική επανάληψη (external iteration). Αντίθετα, χρησιμοποιώντας τα Streams API ο προγραμματιστής απαλλάσσεται εντελώς από την χρήση επαναληπτικών βρόχων. Η επεξεργασία των δεδομένων γίνεται εσωτερικά εντός της βιβλιοθήκης. Αυτός ο τρόπος επανάληψης πάνω στα δεδομένα ονομάζεται εσωτερική επανάληψη (internal iteration).

Εάν υπήρχε ανάγκη για επεξεργασία μεγάλης ποσότητας δεδομένων θα ήταν καλό να χρησιμοποιηθεί ένας υπολογιστής με πολλούς επεξεργαστές. Το πρόβλημα είναι ότι ένα κλασικό πρόγραμμα γραμμένο σε Java θα χρησιμοποιούσε μόνο τον έναν επεξεργαστή και οι υπόλοιποι θα έμεναν ανεκμετάλλευτοι. Επίσης πολλές εταιρίες πλέον χρησιμοποιούν clusters (υπολογιστές που συνδέονται μεταξύ τους με γρήγορα δίκτυα) για να επεξεργάζονται μεγάλες ποσότητες δεδομένων. Η Java 8 παρέχει πολλές διευκολύνσεις με τα νέα στυλ προγραμματισμού με τα οποία εμπλουτίστηκε, προκειμένου να εκμεταλλευτεί στο έπακρο τις δυνατότητες των εν λόγω υπολογιστών.

Η προσπάθεια εκμετάλλευσης του παραλληλισμού με την χρήση πολυνηματικού κώδικα (multithreaded code) είναι δύσκολη. Ο τρόπος σκέψης είναι διαφορετικός διότι τα νήματα μπορούν να έχουν πρόσβαση και να ενημερώνουν διαμοιραζόμενες μεταβλητές ταυτόχρονα. Αυτό μπορεί να οδηγήσει σε αναπάντεχη αλλαγή κάποιον δεδομένων εάν τα νήματα δεν συνεργαστούν σωστά.

Παράδειγμα:



Εικόνα 4: Ένα πιθανό πρόβλημα με δύο νήματα χρησιμοποιούν μια κοινόχρηστη μεταβλητή για να κάνουν πρόσθεση. Το αποτέλεσμα είναι 105 αντί για το σωστό που είναι 108[1].

Και αυτό το πρόβλημα διαχείρισης πολλών επεξεργαστών, όπως και το προηγούμενο της επεξεργασίας συλλογών η Java 8 το αντιμετωπίζει με τα Streams API:

1. Υπάρχουν πολλές μορφές επεξεργασίας δεδομένων επαναλαμβάνονται μέσα σε ένα πρόγραμμα. Η διαμόρφωση και ο εμπλουτισμός της βιβλιοθήκης java.util.stream με λειτουργίες όπως: στοιχεία φιλτραρίσματος που βασίζονται σε κάποιο κριτήριο, εξαγωγή δεδομένων ή ομαδοποίηση δεδομένων κ.τ.λ. προσφέρει πολλά πλεονεκτήματα.
2. Οι εν λόγω λειτουργίες μπορούν συχνά να εκτελεστούν παράλληλα.

Τα νέα Streams API συμπεριφέρεται παρόμοια με της ήδη υπάρχουσες Συλλογές API της Java: και τα δύο παρέχουν πρόσβαση σε ακολουθίες αντικειμένων δεδομένων. Προς το παρόν αυτό που πρέπει να μείνει στον αναγνώστη, είναι ότι οι συλλογές χρησιμοποιούνται ως επί το πλείστον για την αποθήκευση και την πρόσβαση στα δεδομένα, ενώ τα streams είναι ως επί το πλείστον για την περιγραφή των υπολογισμών σε δεδομένα. Το βασικό σημείο εδώ είναι ότι τα streams επιτρέπουν και ενθαρρύνουν τα στοιχεία που βρίσκονται μέσα σε ένα stream να επεξεργάζονται παράλληλα.

Συνοπτικά ο παραλληλισμός στην Java 8 έγινε εύκολος για τον προγραμματιστή για δυο λόγους:

1. Η ίδια η βιβλιοθήκη χειρίζεται την τμηματοποίηση. Χωρίζει ένα μεγάλο stream σε πολλά μικρότερα για να επεξεργαστούν παράλληλα.
2. Αυτός ο σχεδόν δωρεάν παραλληλισμός χάρη στα streams, λειτουργεί μόνο αν οι μέθοδοι που πέρασαν στις μεθόδους της βιβλιοθήκης δεν αλληλεπιδρούν μεταξύ τους, για παράδειγμα με το να έχουν ευμετάβλητα κοινόχρηστα αντικείμενα.

Ένα από τα πρακτικά ζητήματα που αντιμετώπισαν οι σχεδιαστές της Java 8 κατά την επέκταση της προηγούμενης έκδοσης με όλα αυτά τα καινούργια χαρακτηριστικά ήταν η εξέλιξη των ήδη υπαρχόντων διεπαφών. Αυτό μπορεί να φαίνεται τετριμμένο, αλλά, πριν από την Java 8, η ενημέρωση μιας διεπαφής επιτυγχάνονταν μόνο εάν ενημερώνονταν όλες οι κλάσεις που την υλοποιούσαν, πράγμα χρονοβόρο και δύσκολο. Αυτό το θέμα επιλύθηκε στη Java 8 με την χρήση των προεπιλεγμένων μεθόδων (default methods).

1.4 DEFAULT METHODS

Οι προεπιλεγμένες μέθοδοι προστέθηκαν στην Java 8 κυρίως για να υποστηρίξουν την προσπάθεια των σχεδιαστών της βιβλιοθήκης να γράψουν περισσότερο εξελίξιμες (more evolvable) διεπαφές[2, 3]. Είναι αρκετά σημαντικές γιατί εμφανίζονται όλο και περισσότερο στις διεπαφές και διευκολύνουν την εξέλιξη του προγράμματος αντί να συμβάλουν στην συγγραφή κάποιου συγκεκριμένου προγράμματος.

Παράδειγμα:

· List<Apple> heavyApples1 =
inventory.***stream()***.filter((Apple a) -> a.getWeight() > 150)

.collect(toList());

· List<Apple> heavyApples2 =
inventory.***parallelStream()***.filter((Apple a) -> a.getWeight() > 150)

.collect(toList());

Μια List<T> πριν την Java 8 δεν διέθετε ούτε stream και parallelStream μεθόδους ούτε η συλλογή Collection <T> περιείχε κάποια διεπαφή που να τις υλοποιεί και αυτό γιατί αυτές οι μέθοδοι δεν είχαν ακόμη επινοηθεί. Χωρίς όμως αυτές τις μεθόδους το παραπάνω κομμάτι κώδικα δεν θα μπορούσε να μεταγλωττιστεί. Το πιο εύκολο για τους σχεδιαστές, θα ήταν να προστεθεί μια stream

μέθοδος στην Collection διεπαφή και να προστεθεί η υλοποίηση της στην κλάση ArrayList. Αυτό όμως θα ήταν πολύ δύσκολο για τους χρήστες γιατί υπάρχουν πολλά εναλλακτικά πλαίσια συλλογών που υλοποιούν διεπαφές από το Collections API.

Η προσθήκη μιας νέας μεθόδου σε μια διεπαφή θα σήμαινε ότι όλες οι συγκεκριμένες κλάσεις θα πρέπει να παρέχουν μια υλοποίηση για αυτή. Έπρεπε λοιπόν να βρεθεί ένας τρόπος να μπορούν να εξελιχτούν οι διεπαφές χωρίς να επηρεάζονται οι ήδη υπάρχουσες υλοποιήσεις. Η λύση στη Java 8 είναι να διαμορφώσουν έτσι την διεπαφή ώστε να περιέχει υπογραφές (signatures) της μεθόδου για τις οποίες η κλάση δεν παρέχει κάποια υλοποίηση. Τα τμήματα της μεθόδου που λείπουν δίνονται ως τμήμα της διεπαφής (εξ ου και προεπιλεγμένες υλοποιήσεις) και όχι ως τμήμα της υλοποιημένης κλασης. Αυτό παρέχει τη δυνατότητα σε ένα σχεδιαστή μιας διεπαφής να επεκτείνει την διεπαφή με την προσθήκη περισσότερων μεθόδων χωρίς να πρέπει να «σπάσει» ο κώδικας που ήδη υπάρχει. Η Java 8 χρησιμοποιεί την λέξη default ως την προεπιλεγμένη λέξη-κλειδί στην προδιαγραφή μιας διεπαφής για την επίτευξη αυτού του στόχου.

Παράδειγμα: Μια default μέθοδος στη Java 8 List διεπαφή, η οποία καλεί την στατική μέθοδο Collections.sort.

· ***default*** void sort(Comparator<? super E> c) {
Collections.sort(this, c);}

Αυτό σημαίνει ότι οποιεσδήποτε συγκεκριμένες κλάσεις (concrete classes) List δεν χρειάζεται να περιέχουν ρητά την υλοποίηση της sort προκειμένου να περάσουν επιτυχώς τη μεταγλώττιση, κάτι που σε προηγούμενες εκδόσεις της Java, δεν ήταν δυνατό.

Ανακεφαλαιώνοντας, σε αυτό το εισαγωγικό κεφάλαιο παρουσιάστηκαν δύο βασικές ιδέες από το συναρτησιακό προγραμματισμό που σήμερα αποτελούν μέρος της Java:

1. η χρήση μεθόδων και lambdas ως τιμές πρώτης κατηγορίας
2. οι κλήσεις συναρτήσεων και μεθόδων μπορούν να εκτελεστούν παράλληλα, αποτελεσματικά και με ασφάλεια εφόσον δεν χρησιμοποιούν ευμετάβλητα κοινόχρηστα δεδομένα

Και οι δύο αυτές ιδέες αξιοποιούνται από τα νέα Streams API που περιγράφηκαν νωρίτερα.

ΚΕΦΑΛΑΙΟ 2

Εκφράσεις Λάμδα

2.1 Εισαγωγή στις εκφράσεις λάμδα

Η γλώσσα Java, από το ξεκίνημα της, βασίζονταν στον αντικειμενοστρεφή προγραμματισμό. Η όλη λογική της, περιστρέφονταν γύρω από την ιδέα, ότι τα αντικείμενα μπορούν να μεταβάλλονται. Πιο συγκεκριμένα, μέθοδοι που περιέχονται μέσα σε κλάσεις, καλούνται πάνω σε αντικείμενα και συνήθως τροποποιούν τα στοιχεία τους. Στον αντικειμενοστρεφή προγραμματισμό, η σειρά επίκλησης των μεθόδων έχει σημασία διότι κάθε μέθοδος δυνητικά, μπορεί να τροποποιήσει την κατάσταση του αντικειμένου. Η ανάλυση του προγράμματος είναι δύσκολη, καθώς η κατάσταση του, εξαρτάται από την σειρά με την οποία θα εκτελεστεί ο κώδικας. Η διαχείριση των αντικειμένων, απαιτεί προγραμματιστική ικανότητα και εμπειρία, επειδή μπορεί να δημιουργηθεί μπέρδεμα, όταν διαφορετικά μέρη του προγράμματος προσπαθούν ταυτόχρονα να τροποποιήσουν τα στοιχεία του ίδιου αντικειμένου.

Επιπλέον σήμερα οι υπολογιστές διαθέτουν πολυ-πύρινους επεξεργαστές αυξάνοντας την ταχύτητα επεξεργασίας και ωθώντας τους προγραμματιστές στην συγγραφή παράλληλων προγραμμάτων που είναι σε θέση να αξιοποιήσουν όλους τους πυρήνες του επεξεργαστή. Και προς αυτή την κατεύθυνση η Java έχει μεριμνήσει, υποστηρίζοντας τον παράλληλο προγραμματισμό σχεδόν από την αρχή [4].

Ο συναρτησιακός προγραμματισμός, που βασίζεται στον λάμδα-λογισμό, υπήρχε πολύ πριν τον αντικειμενοστρεφή προγραμματισμό. Η κεντρική ιδέα στηρίζεται στην έννοια της συνάρτησης, ένα μπλοκ κώδικα που δέχεται τιμές, γνωστές ως παραμέτρους, το οποίο εκτελείται με σκοπό τον υπολογισμό κάποιου αποτελέσματος. Οι συναρτήσεις δεν τροποποιούν τα δεδομένα και για το λόγο αυτό, η σειρά εκτέλεσης τους δεν έχει σημασία στον συναρτησιακό προγραμματισμό. Μια ανώτερης τάξης (higher order) συνάρτηση είναι μια ανώνυμη συνάρτηση που μπορεί να χρησιμοποιηθεί ως αντικείμενο δεδομένων. Μπορεί να αποθηκευτεί σε μια μεταβλητή και να περαστεί ως παράμετρος από ένα σημείο του προγράμματος σε κάποιο άλλο. Ακόμη μπορεί να κληθεί και σε σημείο του προγράμματος στο οποίο δεν ορίζεται υποχρεωτικά

Τα τελευταία χρόνια, ο συναρτησιακός προγραμματισμός έχει γίνει δημοφιλής λόγω της καταλληλότητας του για ταυτόχρονο, παράλληλο και οδηγούμενο από τα γεγονότα (event-driven) προγραμματισμό. Σύγχρονες γλώσσες προγραμματισμού όπως η C#, Groovy, Python, Scala χρησιμοποιούν τον συναρτησιακό προγραμματισμό. Η Java προκειμένου να μην μείνει πίσω, εισήγαγε τις εκφράσεις λάμδα για να υποστηρίξει τον συναρτησιακό προγραμματισμό, ο οποίος μπορεί να συνδυαστεί με τα ήδη δημοφιλή αντικειμενοστρεφή χαρακτηριστικά της για την ανάπτυξη ισχυρών και παράλληλων προγραμμάτων

Στον αντικειμενοστρεφή προγραμματισμό, μια συνάρτηση ονομάζεται μέθοδος και αποτελεί πάντα μέρος μιας κλάσης. Στην Java για να αξιοποιηθεί μια λειτουργία, η εργασία δηλαδή που επιτελεί κάποια μέθοδος, πρέπει να δημιουργηθεί ένα αντικείμενο, να προστεθεί σε αυτό η μέθοδος και στη συνέχεια να περάσει το αντικείμενο ως παράμετρος όπου χρειάζεται. Μια έκφραση λάμδα στη Java, είναι σαν μια συνάρτηση ανώτερης τάξης του συναρτησιακού προγραμματισμού, η οποία είναι ένα ανώνυμο μπλοκ κώδικα που αντιπροσωπεύει μια λειτουργία που μπορεί να περαστεί όπως τα απλά δεδομένα. Μια έκφραση λάμδα μπορεί να «συλλάβει» (capture) τις μεταβλητές για τον προσδιορισμό του πεδίου εφαρμογής της και μπορεί να έχει πρόσβαση σε αυτές αργότερα σε ένα τμήμα κώδικα που δεν είναι απαραίτητο να ορίζονται. Αυτό το χαρακτηριστικό επιτρέπει στις εκφράσεις λάμδα να χρησιμοποιούνται για την υλοποίηση των closures στη Java.

Η Java 8 λοιπόν, εισήγαγε τις εκφράσεις λάμδα[1, 5, 6], που ουσιαστικά αντιπροσωπεύουν ένα στιγμιότυπο (instance) μιας λειτουργικής διεπαφής. Ο, τι γινόταν πριν από την Java 8 χρησιμοποιώντας τις ανώνυμες κλάσεις με την ογκώδη σύνταξη και τον περιττό κώδικα, επιτυγχάνεται τώρα με την βοήθεια των εκφράσεων λάμδα με μια πολύ πιο συνοπτική και κομψή σύνταξη. Να σημειωθεί ότι, οι λειτουργικές διεπαφές, δεν αποτελούν νέα προσθήκη στη Java 8. Υπήρχαν από την αρχή της δημιουργία της γλώσσας.

Παρακάτω ακολουθεί η αναλυτική περιγραφή των εκφράσεων λάμδα, ο τρόπος σύνταξης τους, το πού και το πώς χρησιμοποιούνται καθώς επίσης και τα οφέλη που προσφέρουν.

2.2 Η ΕΚΦΡΑΣΗ ΛΑΜΔΑ : ΟΡΙΣΜΟΣ ΚΑΙ ΤΡΟΠΟΣ ΣΥΝΤΑΞΗΣ

Μια έκφραση λάμδα (lambda expression) είναι ένα ανώνυμο μπλοκ κώδικα με μια λίστα τυπικών παραμέτρων και ένα σώμα. Μερικές φορές μια έκφραση λάμδα καλείται απλώς λάμδα. Ο όρος "λάμδα" έχει τις ρίζες της στον Λάμδα λογισμό που χρησιμοποιεί το ελληνικό γράμμα λάμδα (λ) για να υποδηλώσει μια λειτουργική αφαίρεση (function abstraction).

Ορισμός: Μια έκφραση λάμδα, αποτελείται από τρία μέρη:

1. Μια λίστα παραμέτρων
2. Ένα βέλος (®) που διαχωρίζει την λίστα παραμέτρων από το σώμα
3. Το σώμα του λάμδα.

Η λίστα παραμέτρων δηλώνεται με τον ίδιο τρόπο όπως και η λίστα των παραμέτρων για τις μεθόδους. Οι παράμετροι, περικλείονται σε παρενθέσεις, όπως και για τις μεθόδους. Το σώμα της έκφρασης λάμδα είναι ένα μπλοκ κώδικα μέσα σε αγκύλες. Όπως και στο σώμα μιας μεθόδου, στο σώμα μιας έκφρασης λάμδα μπορεί να δηλώνονται τοπικές μεταβλητές, να χρησιμοποιούνται εντολές συμπεριλαμβανομένων των break, continue, και return, να «πετάγονται» εξαιρέσεις, κ.λπ.

Σε αντίθεση με μια μέθοδο, μια έκφραση λάμδα:

1. Δεν έχει όνομα, είναι ανώνυμη.
2. Δεν έχει τύπο επιστροφής. Ο τύπος συνάγεται από το μεταγλωττιστή από το πλαίσιο χρήσης της έκφρασης και από το σώμα της.
3. Δεν έχει πρόταση throws: Συνάγεται από το περιεχόμενο της χρήσης της έκφρασης και το σώμα της.
4. Δεν ορίζει τον τύπο των παραμέτρων. Γι' αυτό μια έκφραση λάμδα δεν μπορεί να είναι γενικής φύσεως.

Μια έκφραση λάμδα περιγράφει μια ανώνυμη συνάρτηση. Η γενική σύνταξη για τη χρήση εκφράσεων λάμδα είναι :

· (<Lambda Parameters List>) ® { <Lambda Body> }

Να τονιστεί ότι μια έκφραση λάμδα δεν είναι μια μέθοδος, παρόλο που η δήλωση της μοιάζει με αυτή για μια μέθοδο. Όπως υποδηλώνει το όνομα, μια λάμδα έκφραση είναι μια έκφραση που αντιπροσωπεύει ένα στιγμιότυπο μιας λειτουργικής διεπαφής. Εκτενέστερα θα αναφερθούμε στις λειτουργικές διεπαφές σε επόμενη ενότητα.

Πίνακας 2: Παραδείγματα εκφράσεων Λάμδα

Παράδειγμα Έκφρασης Λάμδα	Ισοδύναμη Μέθοδος

(int x, int y) ® {
return x + y;
}
	int sum(int x, int y) {
	return x + y;
	}

(Object x) ® {
return x;
}
	Object identity(Object x) {
	return x;
	}

(int x, int y) ® {
if (x > y) {
return x;
}
else {
return y;
}
}
	int getMax(int x, int y) {
	if (x > y) {
	return x;
	}
	else {
	return y;
	}
	}

(String msg) ® {
System.out.println(msg);
}
	void print(String msg) {
	System.out.println(msg);
	}

( )® {
 	
System.out.println(LocalDate.now());
}
	void printCurrentDate( ) {
	
	System.out.println(LocalDate.now( ));
	}

( )® {	void doNothing( ) {

// No code goes here }	// No code goes here }

Ένας από τους στόχους των εκφράσεων λάμδα είναι να κρατήσουν τη σύνταξη τους συνοπτική και να αφήνουν τον μεταγλωττιστή να συναγάγει τα όποια στοιχεία απαιτούνται κάθε φορά. Μπορεί να παραλειφτεί ο δηλωμένος τύπος των παραμέτρων. Ο μεταγλωττιστής θα συναγάγει τα είδη των παραμέτρων από το πλαίσιο εντός του οποίου χρησιμοποιείται η έκφραση λάμδα.

// Οι τύποι των παραμέτρων δηλώνονται

· (int x, int y) ® { return x + y; }

// Οι τύποι των παραμέτρων παραλείπονται

· (x, y) ® { return x + y; }

Εάν παραλειφτούν οι τύποι των παραμέτρων, θα πρέπει να παραλειφτούν ή για όλες τις παραμέτρους ή για κανένα.. Η ακόλουθη έκφραση λάμδα δεν θα μεταγλωττιστεί, διότι δηλώνεται ο τύπος της μίας παραμέτρου και παραλείπεται για την άλλη.

// Σφάλμα μεταγλώττισης

· (int x, y) ® { return x + y; }

Μια λάμδα έκφραση που δεν δηλώνονται οι τύποι των παραμέτρων της είναι γνωστή ως έμμεση έκφραση λάμδα ή έμμεσου τύπου έκφραση λάμδα. Μια έκφραση λάμδα που δηλώνονται τύποι των παραμέτρων της, είναι γνωστή ως ρητή έκφραση λάμδα ή ρητού τύπου έκφραση λάμδα.

Μερικές φορές, μια έκφραση λάμδα έχει μόνο μία παράμετρο, της οποίας ο τύπος μπορεί να παραλειφτεί όπως και για μια λάμδα έκφραση με πολλές παραμέτρους. Επίσης άμα παραλειφτεί ο τύπος της παραμέτρου, τότε και μόνο τότε, μπορούν να πασαλειφθούν και οι παρενθέσεις που περιέχουν την παράμετρο.

// Δηλώνεται ο τύπος της παραμέτρου

· (String msg) ® { System.out.println(msg); }

// Παραλείπεται ο τύπος της παραμέτρου

· (msg) ® { System.out.println(msg); }

// Παραλείπεται ο τύπος της παραμέτρου και οι παρενθέσεις

· msg ® { System.out.println(msg); }

// Παραλείπονται οι παρενθέσεις, αλλά όχι ο τύπος της παραμέτρου γιατί δεν επιτρέπεται.

· String msg ® { System.out.println(msg); }

Αν μια έκφραση λάμδα δεν παίρνει καμία παράμετρο, τότε πρέπει να χρησιμοποιούνται υποχρεωτικά οι κενές παρενθέσεις.

// Δεν παίρνει καμία παράμετρο

· ( ) ® { System.out.println("Hello"); }

//Λάθος

· ® System.out.println("Hello"); }

Το σώμα μιας έκφρασης λάμδα μπορεί να αποτελείται από ένα μπλοκ εντολών ή μια μόνο έκφραση. Ένα μπλοκ εντολών περικλείεται μέσα σε αγκύλες ενώ μια μόνο έκφραση, δεν είναι απαραίτητο να βρίσκεται μέσα σε αγκύλες. Το σώμα εντολών μιας έκφρασης λάμδα, εκτελείται με τον ίδιο τρόπο όπως το σώμα μιας μεθόδου. Μια εντολή επιστροφής ή το τέλος του σώματος επιστρέφει τον έλεγχο στο σημείο του κώδικα μετά την κλήση της έκφρασης. Όταν μια έκφραση χρησιμοποιείται ως σώμα, αποτιμάται και επιστρέφεται στο σημείο από το οποίο κλήθηκε. Αν η επιστρεφόμενη τιμή είναι τύπου void, τότε δεν επιστρέφεται τίποτα.

// Χρησιμοποιεί μπλοκ εντολών. Παίρνει δύο παραμέτρους τύπου int και επιστρέφει το άθροισμά τους

· (int x, int y) ® { return x + y; }

// Χρησιμοποιεί μια έκφραση. Παίρνει δύο παραμέτρους τύπου int και επιστρέφει το άθροισμά τους

· (int x, int y) ® x + y

// Χρησιμοποιεί μπλοκ εντολών και δεν επιστέφει τίποτα (void)

· (String msg) ® { System.out.println(msg); }

// Χρησιμοποιεί μια έκφραση και δεν επιστέφει τίποτα (void)

· (String msg) ® System.out.println(msg)

2.3 ΠΡΟΣΔΙΟΡΙΣΜΟΣ ΤΥΠΟΥ ΕΚΦΡΑΣΕΩΝ ΛΑΜΔΑ

Κάθε έκφραση λάμδα έχει έναν τύπο, ο οποίος είναι κάποιος τύπος λειτουργικής διεπαφής. Πιο απλά μια έκφραση λάμδα, βοηθάει στην δημιουργία ενός στιγμιότυπου μιας λειτουργικής διεπαφής. Παρ 'όλα αυτά, η ίδια η έκφραση λάμδα δεν περιέχει πληροφορίες σχετικά με το ποια λειτουργική διεπαφή υλοποιείται. Ο τύπος της έκφρασης λάμδα εξαρτάται από το πλαίσιο στο οποίο χρησιμοποιείται η έκφραση.

Υπάρχουν δύο τύποι εκφράσεων στη Java:

1. **Αυτόνομες Εκφράσεις (Standalone Expressions):** Μια αυτόνομη έκφραση είναι μια έκφραση της οποίας ο τύπος μπορεί να

καθοριστεί από την έκφραση χωρίς να απαιτείται γνώση του πλαισίου χρήσης της

// Ο τύπος της έκφρασης είναι String

· new String("Hello")

// Ο τύπος της έκφρασης είναι String (ένα σκέτο String είναι επίσης έκφραση)

· "Hello"

// Ο τύπος της έκφρασης είναι ArrayList<String>

· new ArrayList<String>( )

2. **Πολυ - Εκφράσεις (Poly Expressions):** Μια πολυ-έκφραση είναι μια έκφραση που έχει διαφορετικούς τύπους σε διαφορετικά πλαίσια. Ο μεταγλωττιστής καθορίζει τον τύπο της έκφρασης. Τα πλαίσια που επιτρέπουν την χρήση των πολύ-εκφράσεων είναι γνωστά ως πολυ-πλαίσια (poly contexts). Όλες οι εκφράσεις λάμδα στη Java είναι πολύ-εκφράσεις. Παίρνουν τον τύπο τους από το πλαίσιο στο οποίο χρησιμοποιούνται.

// Ο τύπος του new ArrayList< >( ) είναι ArrayList<Long>

· ArrayList<Long> idList = new ArrayList< >( );
· ArrayList<String> nameList = new ArrayList< >( );

Ορισμός: Ο μεταγλωττιστής συνάγει τον τύπο της έκφρασης λάμδα. Το πλαίσιο στο οποίο χρησιμοποιείται μια έκφραση λάμδα αναμένει έναν τύπο, ο οποίος ονομάζεται **στόχος τύπος (target type)**. Η διαδικασία συναγωγής του τύπου της έκφρασης λάμδα από το πλαίσιο είναι γνωστή ως **προσδιορισμός τύπου στόχου (target typing).**

Παράδειγμα: Έστω ο παρακάτω ψευδοκώδικας για μια εντολή εκχώρησης όπου σε μια μεταβλητή τύπου T έχει ανατεθεί μια έκφραση λάμδα:

· T t = <LambdaExpression>;

Ο τύπος στόχος της έκφρασης στο πλαίσιο αυτό, είναι ο T. Ο μεταγλωττιστής χρησιμοποιεί τους ακόλουθους κανόνες για να ελέγξει αν η <LambdaExpression> έχει τύπο συμβατό με τον τύπο του στόχου της T:

1. Η T πρέπει να έχει ένα τύπο λειτουργικής διεπαφής.

2. Η έκφραση λάμδα πρέπει να έχει τον ίδιο αριθμό και τύπο παραμέτρων, όπως η αφηρημένη μέθοδο της T. Για μια έμμεση έκφραση λάμδα, ο μεταγλωττιστής θα συναγάγει τους τύπους των παραμέτρων από την αφηρημένη μέθοδο της T.

3. Ο τύπος της επιστρεφόμενης τιμής από το σώμα της έκφρασης λάμδα πρέπει να είναι συμβατός με τον τύπο επιστροφής της αφηρημένης μεθόδου της T.

4. Εάν το σώμα της έκφρασης λάμδα «πετάξει» εξαιρέσεις ελέγχου, οι εξαιρέσεις αυτές πρέπει να είναι συμβατές με τη δηλωθείσα πρόταση throws της αφηρημένης μεθόδου της T. Θεωρείται σφάλμα μεταγλώττισης μια εξαίρεση ελέγχου από το σώμα μιας έκφρασης λάμδα, εάν ο στόχος τύπος δεν περιέχει μια πρόταση throws.

Η ιδέα του στόχου τύπου επιτρέπει, η ίδια έκφραση λάμδα να μπορεί να συνδυαστεί με διαφορετικές λειτουργικές διεπαφές με την προϋπόθεση αυτές να έχουν συμβατές υπογραφές αφηρημένης μεθόδου.

Παράδειγμα: Χρήση των εκφράσεων λάμδα. Η ίδια έκφραση λάμδα μπορεί να χρησιμοποιηθεί με πολλές διαφορετικές λειτουργικές διεπαφές.

```
// Η λειτουργική διεπαφή Adder.java

    · package com.jdojo.lambda;
@FunctionalInterface

public interface Adder {

    double add(double n1, double n2);

}

// Η λειτουργική διεπαφή Joiner.java

    · package com.jdojo.lambda;
@FunctionalInterface

public interface Joiner {

    String join(String s1, String s2);

}

// TargetTypeTest.java

    · package com.jdojo.lambda;
public class TargetTypeTest {

    public static void main(String[ ] args) {

// Δημιουργία ενός Adder με την χρήση έκφρασης λάμδα

        Adder adder = (x, y) ®  x + y;
```

```
// Δημιουργία ενός Joiner με την χρήση έκφρασης λάμδα
                Joiner joiner = (x, y) ®  x + y;
// Πρόσθεση δυο doubles
                double sum1 = adder.add(10.34, 89.11);
// Πρόσθεση δυο ints
                double sum2 = adder.add(10, 89);
// Συνένωση δυο strings
                String str = joiner.join("Hello", " lambda");
                System.out.println("sum1 = " + sum1);
                System.out.println("sum2 = " + sum2);
                System.out.println("str = " + str);
        }
}
//Αποτελέσματα εκτέλεσης
        · sum1 = 99.45
sum2 = 99.0
str = Hello lambda
```

Ο μεταγλωττιστής της Java συμπεραίνει ποια λειτουργική διεπαφή να συνδυάσει με ποια έκφραση λάμδα από το πλαίσιο χρησιμοποίησης τους (τύπος στόχος). Αυτό σημαίνει επίσης ότι μπορεί να συναγάγει μια κατάλληλη υπογραφή για την έκφραση λάμδα επειδή ο λειτουργικός περιγραφέας είναι διαθέσιμος μέσω του τύπου στόχου. Το όφελος είναι ότι ο μεταγλωττιστής έχει πρόσβαση στους τύπους των παραμέτρων της έκφρασης λάμδα, οι οποίοι μπορεί να παραλείπονται κατά την σύνταξη της έκφρασης λάμδα. Με άλλα λόγια, ο μεταγλωττιστής Java συνάγει τους τύπους των παραμέτρων της έκφρασης λάμδα. Να σημειωθεί ότι μερικές φορές είναι προτιμότερο να περιλαμβάνονται οι τύποι των παραμέτρων ρητά (πιο ευανάγνωστος κώδικας) και μερικές φορές όχι.

Υπάρχουν όμως στιγμές, κατά τις οποίες ο μεταγλωττιστής δεν μπορεί να αποφασίσει από μόνος του, τους τύπους των παραμέτρων και το ποια λειτουργική διεπαφή να χρησιμοποιήσει. Σε τέτοιες περιπτώσεις πρέπει να δοθούν στον μεταγλωττιστή περισσότερες πληροφορίες.

Μερικοί τρόποι για να βοηθηθεί ο μεταγλωττιστής ώστε να επιλύσει την ασάφεια είναι:

1. Αν η έκφραση λάμδα είναι έμμεση, πρέπει να γίνει σαφής, διευκρινίζοντας τον τύπο των παραμέτρων.

2. Χρήση cast.

3. Η μη χρήση της έκφρασης λάμδα άμεσα ως παράμετρος της μεθόδου. Πρώτα πρέπει να εκχωρηθεί σε μια μεταβλητή του επιθυμητού τύπου, και στη συνέχεια, να περαστεί στη μέθοδο η μεταβλητή.

Οι εκφράσεις λάμδα εκτός των μεταβλητών που χρησιμοποιούν μέσα στο σώμα τους, έχουν επίσης τη δυνατότητα να χρησιμοποιούν ελεύθερες μεταβλητές (μεταβλητές που δεν είναι παράμετροι τους και ορίζονται σε ένα εξωτερικό πεδίο), ακριβώς όπως και οι ανώνυμες κλάσεις. Η διαδικασία αυτή ονομάζεται σύλληψη λάμδα (capturing lambdas).

Παράδειγμα: Σύλληψη της ελεύθερης μεταβλητής portNumber.

· int portNumber = 1337;
Runnable r = ( ) ® System.out.println(portNumber);

Οι λάμδα εκφράσεις μπορούν να συλλάβουν τις τοπικές μεταβλητές που τους έχουν ανατεθεί μόνο μία φορά.

Παράδειγμα: Ο κώδικας που ακολουθεί δεν μεταγλωττίζεται επειδή γίνετε ανάθεση τιμής στη μεταβλητή portNumber δύο φορές.

· int portNumber = 1337;
Runnable r = ( ) ® System.out.println(portNumber);

portNumber = 31337;

Στην Java 8 οι εκφράσεις λάμδα και ανώνυμες κλάσεις μπορούν να περαστούν ως ορίσματα σε μεθόδους και μπορούν να έχουν πρόσβαση σε μεταβλητές εκτός του πεδίου εφαρμογής τους. Αλλά έχουν ένα περιορισμό: δεν μπορούν να τροποποιήσουν το περιεχόμενο των τοπικών μεταβλητών μίας μεθόδου μέσα στην οποία ορίζεται η ίδια η έκφραση λάμδα. Οι μεταβλητές αυτές πρέπει να θεωρούνται τελικές (final).

Οι λάμδα εκφράσεις μπορούν να χρησιμοποιηθούν μόνο στα ακόλουθα πλαίσια:

· **Πλαίσιο ανάθεσης (assignment context):** Μια έκφραση λάμδα μπορεί να εμφανιστεί στην δεξιά πλευρά μιας εντολής εκχώρησης.

· **Πλαίσιο επίκλησης μεθόδου (method invocation context):** Μια έκφραση λάμδα μπορεί να εμφανιστεί ως παράμετρος σε μια μέθοδο ή κατά την κλήση ενός κατασκευαστή (constructor).

· **Πλαίσιο cast (cast context):** Μια έκφραση λάμδα μπορεί να χρησιμοποιηθεί εάν έχει προηγηθεί ένα cast. Ο τύπος που καθορίζεται στο cast είναι ο τύπος στόχος της.

· **Πλαίσιο επιστροφής (return context):** Μια έκφραση λάμδα μπορεί να εμφανιστεί σε μια εντολή return μέσα σε μια μέθοδο εφόσον ο τύπος στόχος του λάμδα, είναι ίδιος με τον δηλωμένο τύπο επιστροφής της μεθόδου.

Εν κατακλείδι, ο τύπος στόχος μπορεί να χρησιμοποιηθεί:

1. Για να ελεγχθεί εάν μια έκφραση λάμδα μπορεί να χρησιμοποιηθεί σε ένα συγκεκριμένο πλαίσιο.

Για να συναγάγει τους τύπους των παραμέτρων μιας έκφρασης λάμδα.

2.4 ΛΕΙΤΟΥΡΓΙΚΕΣ ΔΙΕΠΑΦΕΣ

Οι εκφράσεις λάμδα, χρησιμοποιούνται κατά κύριο λόγο στο πλαίσιο μιας λειτουργικής διεπαφής (functional interface).

Ορισμός: Μια λειτουργική διεπαφή είναι μια διεπαφή που καθορίζει ακριβώς μια αφηρημένη μέθοδο.

Ορισμός: Ο Λειτουργικός Περιγραφέας (Function descriptor) είναι μια αφηρημένη μέθοδος μιας λειτουργικής διεπαφής της οποίας η υπογραφή (signature) περιγράφει ουσιαστικά την υπογραφή της λάμδα έκφρασης.

Στην Java 8 μπορεί πλέον μια διεπαφή να περιέχει και προεπιλεγμένες μεθόδους (default methods) οι οποίες θα παρουσιαστούν αναλυτικά σε επόμενο κεφάλαιο. Μια διεπαφή μπορεί να είναι λειτουργική ακόμη και αν έχει πολλές προεπιλεγμένες μεθόδους αρκει να εξακολουθεί να καθορίζει ακριβώς μια αφηρημένη μέθοδο. Οι λάμδα εκφράσεις βοηθούν στην υλοποίηση μιας αφηρημένης μεθόδου μιας λειτουργικής διεπαφής directly inline και αντιμετωπίζουν ολόκληρη την έκφραση ως ένα στιγμιότυπο της λειτουργικής διεπαφής.

Η δήλωση μιας λειτουργικής διεπαφής μπορεί προαιρετικά να συμπληρώνεται με την ένδειξη @FunctionalInterface που βρίσκεται στο πακέτο java.lang. Αυτός ο συμβολισμός διαβεβαιώνει τον μεταγλωττιστή ότι ο δηλωμένος τύπος είναι μια λειτουργική διεπαφή. Ο μεταγλωττιστής θα επιστρέψει σφάλμα με σαφείς υποδείξεις, αν έχει οριστεί μια διεπαφή χρησιμοποιώντας τον συμβολισμό @FunctionalInterface και δεν είναι μια λειτουργική διεπαφή. Επιπλέον η χρήση του συγκεκριμένου συμβολισμού διασφαλίζει την ακούσια αλλαγή μιας λειτουργικής διεπαφής σε μη-λειτουργική, γιατί θα το «πιάσει» ο μεταγλωττιστής.

Παράδειγμα: Η παρακάτω δήλωση της διεπαφής Operations, δεν θα μεταγλωττιστεί παρόλο που χρησιμοποιείται ο συμβολισμός @FunctionalInterface γιατί η διεπαφή δεν είναι λειτουργική (ορίζει δυο αφηρημένες μεθόδους)

```
· @FunctionalInterface
public interface Operations {

    double add(double n1, double n2);

    double subtract(double n1, double n2);

}
```

Για να γίνει η μεταγλώττιση είτε πρέπει να αφαιρεθεί ο συμβολισμός @FunctionalInterface είτε να αφαιρεθεί μια από τις αφηρημένες μεθόδους.

Επιτρέπεται μια λειτουργική διεπαφή να έχει τύπους παραμέτρων. Γι' αυτό μια λειτουργική διεπαφή μπορεί να είναι γενική (generic).

```
· @FunctionalInterface
public interface Comparator<T> {

    int compare(T o1, T o2);

}
```

Μια λειτουργική διεπαφή μπορεί να έχει μια γενική αφηρημένη μέθοδο. Γι' αυτό η αφηρημένη μέθοδος μπορεί να δηλώσει τύπους παραμέτρων.

```
· @FunctionalInterface
public interface Processor {

    <T> void process(T[ ] list);

}
```

Μια έκφραση λάμδα δεν μπορεί να ορίσει τύπους παραμέτρων. Γι' αυτό δεν μπορεί να έχει έναν τύπο στόχο του οποίου η αφηρημένη μέθοδος είναι γενική. Σε τέτοιες περιπτώσεις χρησιμοποιούνται οι αναφορές μεθόδων που παρουσιάζοντα στην επόμενη ενότητα.

Προς το παρόν αυτό που πρέπει να μείνει στον αναγνώστη είναι ότι μια έκφραση λάμδα μπορεί να ανατεθεί σε μια μεταβλητή ή να περάσει σε μία μέθοδο αναμένοντας μια λειτουργική διεπαφή ως όρισμα, με την προϋπόθεση ότι η έκφραση λάμδα έχει την ίδια υπογραφή με την αφηρημένη μέθοδο της λειτουργικής διεπαφής. Το πέρασμα μιας λάμδα έκφρασης εκεί που αναμένεται μια λειτουργική διεπαφή προτιμήθηκε από τους σχεδιαστές της γλώσσας επειδή αυτός ο τρόπος είναι πιο φυσικός χωρίς να αυξάνεται η πολυπλοκότητα της γλώσσας.

Οι καινούργιες διεπαφές που προστέθηκαν στην Java 8 μπορούν να επαναχρησιμοποιηθούν για να περάσουν πολλές διαφορετικές εκφράσεις λάμδα.

Έτσι, προκειμένου να χρησιμοποιηθούν διαφορετικές εκφράσεις λάμδα, θα πρέπει να υπάρχει μια σειρά από λειτουργικές διεπαφές που να μπορούν να περιγράψουν κοινούς λειτουργικούς περιγραφείς. Εκτός λοιπόν από τις λειτουργικές διεπαφές που υπήρχαν ήδη – όπως η Comparable, Runnable, και Callable – οι σχεδιαστές της Java 8 πρόσθεσαν στην βιβλιοθήκη java.util.function package, αρκετές νέες λειτουργικές διεπαφές για την διευκόλυνση των προγραμματιστών.

Στον παρακάτω πίνακα παρουσιάζονται ονομαστικά οι πιο κοινές λειτουργικές διεπαφές της Java 8 που ορίζονται στο πακέτο java.util.function και στη συνέχεια ακολουθεί μια σύντομη περιγραφή για την κάθε μια από αυτές.

Πίνακας 3: Οι πιο συνηθισμένες λειτουργικές διεπαφές της Java 8

Λειτουργική Διεπαφή	**Λειτουργικός Περιγραφέας**	**Εξειδικεύσεις**
Predicate<T>	T ® boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	T ® void	IntConsumer, LongConsumer, DoubleConsumer
Function<T, R>	T ® R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>
Supplier<T>	()® T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T ® T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T, T) ® T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<L, R>	(L, R) ® boolean	
BiConsumer<T, U>	(T, U) ® void	ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>
BiFunction<T, U, R>	(T, U) ® R	ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>

· **Predicate:** Η διεπαφή java.util.function.Predicate<T> ορίζει μια αφηρημένη μέθοδο που ονομάζεται test που δέχεται ένα αντικείμενο γενικού τύπου T και επιστρέφει μια τιμή τύπου boolean. Μπορεί να χρησιμοποιηθεί,

όταν χρειάζεται να αναπαρασταθεί μια boolean έκφραση που χρησιμοποιεί ένα αντικείμενο τύπου T.

· **Consumer:** Η διεπαφή java.util.function.Consumer<T> ορίζει μια αφηρημένη μέθοδο που ονομάζεται accept που παίρνει ένα αντικείμενο γενικού τύπου T και επιστρέφει μια τιμή void (δηλαδή δεν επιστρέφει αποτέλεσμα). Μπορεί να χρησιμοποιηθεί, όταν χρειάζεται πρόσβαση σε ένα αντικείμενο τύπου T προκειμένου να εκτελεστούν κάποιες εργασίες σε αυτό.

· **Function:** Η διεπαφή java.util.function.Function<T, R> ορίζει μια αφηρημένη μέθοδο που ονομάζεται apply που παίρνει ένα αντικείμενο γενικού τύπου T ως είσοδο και επιστρέφει ένα αντικείμενο γενικού τύπου R. Μπορεί να χρησιμοποιηθεί, όταν χρειάζεται να καθοριστεί μια έκφραση λάμδα που αντιστοιχεί μια προς μια πληροφορίες από ένα αντικείμενο εισόδου σε μία έξοδο.

· **Supplier:** Αντιπροσωπεύει έναν supplier που επιστρέφει μια τιμή

· **UnaryOperator:** Κληρονομεί από την Function <T, T>. Αντιπροσωπεύει μια συνάρτηση που παίρνει ένα όρισμα και επιστρέφει ένα αποτέλεσμα του ίδιου τύπου.

· **BinaryOperator:** Κληρονομεί από την BiFunction <T, T, T>. Αντιπροσωπεύει μια συνάρτηση που παίρνει δύο ορίσματα του ίδιου τύπου και επιστρέφει ένα αποτέλεσμα που είναι επίσης του ίδιου τύπου.

· **BiPredicate:** Αντιπροσωπεύει ένα Predicate με δύο ορίσματα

· **BiConsumer:** Αντιπροσωπεύει μια πράξη που παίρνει δυο ορίσματα, εκτελεί πάνω τους εργασίες που προκαλούν αλλαγές και δεν επιστρέφει κανένα αποτέλεσμα.

· **BiFunction :** Αντιπροσωπεύει μια Function που παίρνει δύο ορίσματα τύπων T και U, και επιστρέφει ένα αποτέλεσμα τύπου R.

Ο παρακάτω πίνακας περιέχει μερικά ενδεικτικά παραδείγματα εκφράσεων λάμδα και λειτουργικών διεπαφών στις οποίες μπορούν να χρησιμοποιηθούν. Αντίστοιχες

Πίνακας 4: Κάποιες εκφράσεις λάμδα και οι αντίστοιχες λειτουργικές διεπαφές τους

Περίπτωση Χρήσης	Παράδειγμα λάμδα	Αντιστοίχιση με λειτουργική διεπαφή

Μια Boolean έκφραση	(List<String> list) ® list.isEmpty()	Predicate<List<String >>
Δημιουργία αντικειμένων	() ® new Apple(10)	Supplier<Apple>
Κατανάλωση από ένα αντικείμενο	(Apple a) ® System.out.println(a.getWeight())	Consumer<Apple>
Επιλογή / απόσπασμα από ένα αντικείμενο	(String s) ® s.length()	Function<String, Integer> or ToIntFunction<String >
Συνδυασμός δύο τιμών	(int a, int b) ® a * b	IntBinaryOperator
Σύγκριση δυο αντικειμένων	(Apple a1, Apple a2) ® a1.getWeight().compareTo (a2.getWeight())	Comparator<Apple> or BiFunction<Apple, Apple, Integer> or ToIntBiFunction<Ap ple, Apple>

Εκτός από τις γενικού τύπου διεπαφές, υπάρχουν λειτουργικές διεπαφές που εξειδικεύονται σε συγκεκριμένους τύπους. Να θυμίσουμε ότι στην Java κάθε τύπος μπορεί να είναι είτε τύπος αναφοράς (reference type) όπως για παράδειγμα Byte, Integer, Object, List είτε αρχικός τύπος (primitive type) όπως για παράδειγμα int, double, byte, char. Οι γενικού τύπου παράμετροι μπορούν να δεσμεύονται μόνο σε τύπους αναφοράς. Αυτό έχει ως αποτέλεσμα, στη Java να υπάρχει ένας μηχανισμός για τη μετατροπή των αρχικών τύπων στους αντίστοιχους τύπους αναφοράς. Αυτός ο μηχανισμός ονομάζεται boxing. Η αντίθετη προσέγγιση (δηλαδή, η μετατροπή ενός τύπου αναφοράς στον αντίστοιχο αρχικό τύπο) ονομάζεται unboxing. Η Java έχει επίσης ένα μηχανισμό που ονομάζεται autoboxing με τον οποίο οι λειτουργιές boxing και unboxing γίνονται αυτόματα.

Η boxing λειτουργία όμως απαιτεί και αρκετή μνήμη και αρκετές προσπελάσεις μνήμης για τις αντίστοιχες μετατροπές τύπων. Η Java 8, προκειμένου να αποφευχθούν οι autoboxing λειτουργιές όταν η είσοδος και η έξοδος είναι αρχικοί τύποι διαθέτει εξειδικευμένες εκδόσεις (βλπ. Πίνακα 2) των λειτουργικών διεπαφών.

Να σημειωθεί ότι οι λειτουργικές διεπαφές δεν επιτρέπουν σε μία εξαίρεση ελέγχου να «πεταχτεί». Εάν χρειάζεται μια έκφραση λάμδα να πετάξει μια εξαίρεση υπάρχουν δυο επιλογές:

1. Ορισμός νέας λειτουργικής διεπαφής από τον προγραμματιστή που θα δηλώνει την εξαίρεση ελέγχου.

2. Προσθήκη στην έκφραση λάμδα ενός μπλοκ try/catch.

Γενικά, οι λειτουργικές διεπαφές χρησιμοποιούνται σε δύο περιπτώσεις από δύο διαφορετικούς τύπους χρηστών:

1. Από τους σχεδιαστές βιβλιοθηκών για το σχεδιασμό APIs, προκειμένου να καθορίσουν τον τύπο των παραμέτρων και τον τύπο επιστροφής στις δηλώσεις της μεθόδου. Γενικά χρησιμοποιούνται με τον ίδιο τρόπο που χρησιμοποιούνται και οι μη λειτουργικές διεπαφές Οι λειτουργικές διεπαφές όπως ειπώθηκε και παραπάνω, υπήρχαν στην Java από την αρχή και η Java 8 δεν έχει αλλάξει τον τρόπο με τον οποίο χρησιμοποιούνται στο σχεδιασμό των APIs.

Από τους χρήστες βιβλιοθηκών για τη χρήση των APIs. Στην Java 8, οι χρήστες των βιβλιοθηκών χρησιμοποιούν τις λειτουργικές διεπαφές ως τύπους στόχους για τις εκφράσεις λάμδα. Δηλαδή, όταν μία μέθοδος στο API παίρνει μια λειτουργική διεπαφή ως παράμετρο, ο χρήστης του API θα πρέπει να χρησιμοποιήσει μια έκφραση λάμδα για να περάσει την παράμετρο. Η χρήση των εκφράσεων λάμδα έχει το πλεονέκτημα ότι κάνει τον κώδικα συνοπτικό και πιο ευανάγνωστο.

2.5 ΑΝΑΦΟΡΕΣ ΜΕΘΟΔΩΝ

Ορισμός: Μια μέθοδος αναφοράς (method reference) είναι μια συντομογραφία για την δημιουργία μιας έκφρασης λάμδα χρησιμοποιώντας μια υπάρχουσα μέθοδο.

Η βασική ιδέα είναι, ότι εάν μια έκφραση λάμδα μέσα στο σώμα της, ζητάει να κληθεί άμεσα κάποια μέθοδος, είναι καλύτερο να αναφέρεται στη μέθοδο με βάση το όνομα της και όχι μια περιγραφή του πώς αυτή αποκαλείται. Μια αναφορά μεθόδου επιτρέπει την δημιουργία μιας έκφρασης λάμδα από μια ήδη υπάρχουσα υλοποιημένη μέθοδο. Με την δυνατότητα ρητής αναφοράς στο όνομα της μεθόδου, ο κώδικας γίνεται πιο ευανάγνωστος και σαφείς.

Η μέθοδος αναφοράς δεν είναι ένας νέος τύπος της Java. Δεν χρησιμοποιείται σαν δείκτης σε συνάρτηση όπως χρησιμοποιείται σε άλλες γλώσσες προγραμματισμού. Είναι απλώς μια συντομογραφία για τη σύνταξη μιας έκφρασης λάμδα χρησιμοποιώντας μια υπάρχουσα μέθοδο. Μπορεί να χρησιμοποιηθεί μόνο στην περίπτωση που μπορεί να χρησιμοποιηθεί η λάμδα έκφραση.

Η γενική σύνταξη για μια αναφορά μεθόδου είναι:

· <Qualifier> :: <MethodName>

Ο <Qualifier> εξαρτάται από τον τύπο της μεθόδου αναφοράς. Δύο διαδοχικές διπλές τελείες (::) λειτουργούν ως διαχωριστής. Το <MethodName> είναι το όνομα της μεθόδου. Η αναφορά μεθόδου δεν καλεί τη μέθοδο όταν αυτή δηλώνεται. Η μέθοδος καλείται αργότερα, όταν καλείται η μέθοδος του τύπου στόχου. Η σύνταξη για τις αναφορές μεθόδων καθορίζει μόνο το όνομα της μεθόδου. Δεν καθορίζει τους

τύπους των παραμέτρων και το τύπο επιστροφής της μεθόδου. Υπενθυμίζεται ότι η μέθοδος αναφοράς είναι συντομογραφία για μια λάμδα έκφραση. Ο τύπος στόχος, ο οποίος είναι πάντα μια λειτουργική διεπαφή, καθορίζει τις λεπτομέρειες της μεθόδου. Αν η μέθοδος είναι μια υπερφορτωμένη μέθοδος, ο μεταγλωττιστής θα επιλέξει την καταλληλότερη μέθοδο βασιζόμενος στο ευρύτερο πλαίσιο.

Παράδειγμα: Έστω το παρακάτω τμήμα κώδικα:

· import java.util.function.ToIntFunction;
...

ToIntFunction<String> lengthFunction = str ® str.length();

String name = "Ellen";

int len = lengthFunction.applyAsInt(name);

System.out.println("Name = " + name + ", length = " + len);

· Name = Ellen, length = 5

Ο κώδικας χρησιμοποιεί μια έκφραση λάμδα για να ορίσει μια ανώνυμη συνάρτηση που παίρνει ένα String ως όρισμα και επιστρέφει το μήκος του. Το σώμα της έκφρασης λάμδα αποτελείται από μία μόνο κλήση της μεθόδου length( ) της κλάσης String. Η παραπάνω έκφραση λάμδα χρησιμοποιώντας μια μέθοδο αναφοράς για τη μέθοδο length( ) της κλάσης String μπορεί να ξαναγραφεί ως εξής:

· import java.util.function.ToIntFunction;
...

ToIntFunction<String> lengthFunction = String::length;

String name = "Ellen";

int len = lengthFunction.applyAsInt(name);

System.out.println("Name = " + name + ", length = " + len);

· Name = Ellen, length = 5

Οι βασικοί τύποι αναφορών μεθόδων είναι:

1. **Στατική Μέθοδος Αναφοράς (Static Method Reference):** Μια στατική μέθοδος αναφοράς χρησιμοποιείται για να χρησιμοποιήσει μια στατική μέθοδο ενός τύπου σαν λάμδα έκφραση. Ο τύπος μπορεί να είναι μια κλάση, μια διεπαφή, ή ένα enum.

Παράδειγμα:

· static String toBinaryString(int i)

```
// Χρήση έκφρασης λάμδα
Function<Integer, String> func1 = x ® Integer.toBinaryString(x);
System.out.println(func1.apply(17));
// Χρήση αναφοράς μεθόδου
Function<Integer, String> func2 = Integer::toBinaryString;
System.out.println(func2.apply(17));
```

2. **Αναφορά μεθόδου στιγμιότυπου (Instance Method Reference):** Μια μέθοδος στιγμιότυπου καλείται «πάνω» στην αναφορά ενός αντικειμένου. Η αναφορά στο αντικείμενο επί του οποίου γίνεται η επίκληση της μεθόδου στιγμιότυπου είναι γνωστή ως δέκτης (receiver) της μεθόδου επίκλησης. Ο δέκτης μιας μεθόδου επίκλησης μπορεί να είναι μια αναφορά στο αντικείμενο ή μια έκφραση που αποτιμάται σε αναφορά ενός αντικειμένου

Παράδειγμα:

```
· String name = "Kannan";
// Το name είναι ο δέκτης της μεθόδου length( )
· int len1 = name.length( );
// Το "Hello" είναι ο δέκτης της μεθόδου length( )
· int len2 = "Hello".length( );
// Το (new String("Kannan")) είναι ο δέκτης της μεθόδου length( )
· int len3 = (new String("Kannan")).length( );
```

Σε μια μέθοδο αναφοράς για μια μέθοδο στιγμιότυπου, μπορεί να καθοριστεί ο δέκτης της μεθόδου επίκλησης ρητά ή έμμεσα, όταν γίνεται επίκληση της μεθόδου. Ο πρώτος ονομάζεται οριοθετημένος δέκτης (bound receiver) και ο δεύτερος μη οριοθετημένος δέκτης (unbound receiver). Η σύνταξη για την αναφορά μεθόδου στιγμιότυπου υποστηρίζει δύο παραλλαγές:

· objectRef :: instanceMethod
· ClassName :: instanceMethod

3. **Αναφορά μεθόδου στιγμιότυπου υπερ-τύπου (Supertype Instance Method Reference):** Η λέξη-κλειδί «υπέρ» χρησιμοποιείται ως προσδιοριστικό για να κληθεί η 12 σε μια κλάση ή μια διεπαφή. Η λέξη-κλειδί είναι διαθέσιμη μόνο σε ένα πλαίσιο στιγμιότυπου. Χρησιμοποιείται η ακόλουθη σύνταξη για την κατασκευή μιας αναφοράς μεθόδου στιγμιότυπου υπέρ-τύπου:

· TypeName.super::instanceMethod

4. **Αναφορά κατασκευαστή (Constructor Reference):** Μερικές φορές το σώμα μιας λάμδα έκφρασης μπορεί να είναι απλώς μια έκφραση για την δημιουργία αντικειμένων. Η σύνταξη για τη χρήση ενός κατασκευαστή είναι:

· ClassName :: new
· ArrayTypeName :: new

Το ClassName στην έκφραση ClassName :: new είναι το όνομα της κλασης που μπορεί να αρχικοποιηθεί και δεν μπορεί να είναι το όνομα μιας αφηρημένης κλασης. Η λέξη-κλειδί new αναφέρεται στον κατασκευαστή της κλάσης. Μια κλάση μπορεί να έχει πολλούς κατασκευαστές. Η σύνταξη δεν παρέχει κάποιον ξεκάθαρο τρόπο για να αναφερθεί σε έναν συγκεκριμένο κατασκευαστή. Ο μεταγλωττιστής επιλέγει ένα συγκεκριμένο κατασκευαστή με βάση το πλαίσιο συμφραζομένων. Εξετάζει τον τύπο στόχο και τον αριθμό των παραμέτρων στην αφηρημένη μέθοδο του τύπου στόχου. Ο κατασκευαστής του οποίου ο αριθμός των παραμέτρων ταιριάζει με τον αριθμό των παραμέτρων στην αφηρημένη μέθοδο του τύπου στόχου είναι αυτός που επιλέγεται.

Παράδειγμα:

· Supplier<Item> func1 = ( )® new Item( );
Function<String,Item> func2 = name ® new Item(name);

BiFunction<String,Double, Item> func3 = (name, price) ® new Item(name, price);

· System.out.println(func1.get());
System.out.println(func2.apply("Apple"));

System.out.println(func3.apply("Apple", 0.75));

//Αποτελέσματα εκτέλεσης

· Constructor Item() called.
name = Unknown, price = 0.0

Constructor Item(String) called.

name = Apple, price = 0.0

Constructor Item(String, double) called.

name = Apple, price = 0.75

Αντικαθιστώντας τις εκφράσεις λάμδα με τον κατασκευαστή αναφοράς Item :: new:

· Supplier<Item> func1 = Item::new;
Function<String,Item> func2 = Item::new;

BiFunction<String,Double, Item> func3 = Item::new;

· System.out.println(func1.get());
System.out.println(func2.apply("Apple"));

System.out.println(func3.apply("Apple", 0.75));

//Αποτελέσματα εκτέλεσης

· Constructor Item() called.
name = Unknown, price = 0.0

Constructor Item(String) called.

name = Apple, price = 0.0

Constructor Item(String, double) called.

name = Apple, price = 0.75

Οι πίνακες στην Java δεν διαθέτουν κατασκευαστές. Υπάρχει μια ειδική σύνταξη για την χρήση αναφορών κατασκευαστή για πίνακες. Οι κατασκευαστές για πίνακες, παίρνουν ένα όρισμα τύπου int που αντιστοιχεί στο μέγεθος του πίνακα.

Παράδειγμα:

// Χρήση έκφρασης λάμδα

· IntFunction<int[ ]> arrayCreator1 = size ® new int[size];
int[ ] empIds1 = arrayCreator1.apply(5); // Δημιουργία ενός πίνακα τύπου int με πέντε στοιχεία

// Χρήση αναφοράς κατασκευαστή για πίνακα

· IntFunction<int[ ]> arrayCreator2 = int[ ]::new;
int[ ] empIds2 = arrayCreator2.apply(5); // Δημιουργία ενός πίνακα τύπου int με πέντε στοιχεία

5. Γενική Μέθοδος Αναφοράς (Generic Method Reference): Ο μεταγλωττιστής υπολογίζει τον πραγματικό τύπο για τις γενικού τύπου

παραμέτρους όταν μια αναφορά μεθόδου αναφέρεται σε μια γενική μέθοδο. Η σύνταξη για μια αναφορά μεθόδου υποστηρίζει επίσης τον καθορισμό των πραγματικών τύπων παραμέτρων για γενικούς τύπους. Οι πραγματικοί τύποι παραμέτρων προσδιορίζονται πριν τις δυο διπλές τελείες. Για παράδειγμα, η αναφορά κατασκευαστή ArrayList <Long> :: new καθορίζει τον τύπο Long ως τον πραγματικό τύπο παραμέτρου για την γενική κλάση ArrayList <T> .

Παράδειγμα: Προσδιορισμός του πραγματικού τύπου παράμετρο για τη γενική
μέθοδος Arrays.asList ().

· import java.util.Arrays;
import java.util.List;

import java.util.function.Function;

...

Function<String[ ],List<String>>asList = Arrays::<String>asList;

String[ ] namesArray = {"Jim", "Ken", "Li"};

List<String> namesList = asList.apply(namesArray);

for(String name : namesList) {

System.out.println(name);}

Γενικά, ο μεταγλωττιστής ακολουθεί παρόμοια διαδικασία για τον έλεγχο του τύπου της αναφοράς μεθόδου, όπως για τις εκφράσεις λάμδα. Για να καταλάβει αν μια μέθοδος αναφοράς είναι έγκυρη με μια συγκεκριμένη λειτουργική διεπαφή ελέγχει αν η υπογραφή της μεθόδου αναφοράς ταιριάζει με τον τύπο του πλαισίου (context).

Στον παρακάτω πίνακα συγκεντρώνονται περιληπτικά, οι τύποι των μεθόδων αναφοράς που παρουσιάστηκαν παραπάνω.

Πίνακας 5: Συνοπτικά οι τύποι των μεθόδων αναφοράς

Σύνταξη	Σύντομη Περιγραφή
TypeName::staticMethod	Μια μέθοδος αναφοράς σε μια στατική μέθοδο μιας κλάσης, μιας διεπαφής, ή ενός enum
objectRef::instanceMethod	Μια μέθοδος αναφοράς σε μια μέθοδο στιγμιότυπου ενός συγκεκριμένου αντικειμένου
ClassName::instanceMethod	Μία μέθοδος αναφοράς σε μια μέθοδο στιγμιότυπου ενός αυθαίρετου αντικείμενου μιας συγκεκριμένης κλάσης
TypeName.super::instanceMethod	Μια μέθοδος αναφοράς σε μια μέθοδο στιγμιότυπου ενός υπερτύπου ενός συγκεκριμένου αντικειμένου

ClassName::new	Μια αναφορά κατασκευαστή στον κατασκευαστή μιας συγκεκριμένης κλάσης
ArrayTypeName::new	Μια αναφορά κατασκευαστή πίνακα στον κατασκευαστή ενός συγκεκριμένου τύπου πίνακα

ΚΕΦΑΛΑΙΟ 3

STREAMS

3.1 Εισαγωγή στα Streams

Οι σχεδιαστές της γλώσσας Java 8 προκειμένου να εξοικονομήσουν πολύτιμο χρόνο και να κάνουν τη ζωή των προγραμματιστών ευκολότερη εμπλούτισαν το Java API με τα streams[2, 3, 5]. Γενικά, τα steams είναι μια ενημέρωση το Java API που επιτρέπουν τον χειρισμό των συλλογών των δεδομένων με δηλωτικό τρόπο. Το νέο Streams API είναι πολύ εκφραστικό.

Ορισμός: Μια συλλογή (collection) είναι ένα API της Java για την ομαδοποίηση και την επεξεργασία δεδομένων. Είναι το πιο πολυχρησιμοποιημένο API και σχεδόν κάθε Java εφαρμογή κατασκευάζει και επεξεργάζεται συλλογές.

Έχουν γίνει πολλές προσπάθειες για την δημιουργία και παροχή καλύτερων βιβλιοθηκών Java - Guava, Apache και lambdaj - για την διαχείριση των συλλογών. Τώρα η Java 8 έρχεται με τη δική της επίσημη βιβλιοθήκη για την επεξεργασία συλλογών με ένα αρκετά δηλωτικό στυλ.

Συνοψίζοντας, το Streams API της Java 8 επιτρέπει την συγγραφή κώδικα που είναι:

- Δηλωτικός (Declarative), είναι πιο συνοπτικός και ευανάγνωστος
- Συνθέσιμος (Composable), έχει μεγαλύτερη ευελιξία
- Παραλληλοποιήσιμος (Parallelizable), έχει καλύτερη απόδοση

Πιο συγκεκριμένα:

Ορισμός: Μια συγκεντρωτική πράξη (aggregate operation) υπολογίζει μία μοναδική τιμή μέσα από ένα σύνολο τιμών. Το αποτέλεσμα της πράξης μπορεί να είναι μια αρχική (primitive) τιμή, ένα αντικείμενο, ή τίποτα (void). Ένα stream είναι μια ακολουθία από στοιχεία δεδομένων που υποστηρίζουν σειριακές και παράλληλες συγκεντρωτικές πράξεις.

Από τον ορισμό των streams, φαίνεται ότι είναι παρόμοιο με τις συλλογές. Και τα δυο είναι αφηρημένα σχέδια για τη συλλογή στοιχείων δεδομένων. Η διαφορά εντοπίζεται στο εξής σημείο: Οι συλλογές επικεντρώνονται στην αποθήκευση στοιχείων δεδομένων για αποτελεσματική πρόσβαση ενώ τα streams επικεντρώνονται στην εκτέλεση συγκεντρωτικών πράξεων πάνω στα στοιχεία δεδομένων από μια πηγή δεδομένων που είναι συνήθως, αλλά όχι απαραίτητα, συλλογές.

Παρακάτω παρουσιάζονται τα βασικά χαρακτηριστικά των streams σε σύγκριση με αυτά των συλλογών:

1. **Δεν διαθέτουν αποθηκευτικό χώρο:** Μια συλλογή είναι μία στη μνήμη δομή δεδομένων που αποθηκεύει όλα τα στοιχεία της. Όλα τα στοιχεία θα πρέπει να υπάρχουν στη μνήμη πριν προστεθούν στη συλλογή. Ένα stream δεν αποθηκεύει τα στοιχεία. Αντλεί τα στοιχεία από μια πηγή δεδομένων αν και όταν ζητηθεί (on-demand) και τα περνάει σε μια σωλήνωση εργασιών για επεξεργασία.
2. **Μπορούν να αναπαραστήσουν μια ακολουθία από άπειρα στοιχεία:** Μια συλλογή, σε αντίθεση με ένα stream, δεν μπορεί να αναπαραστήσει μια ομάδα άπειρων στοιχείων. Μια συλλογή αποθηκεύει όλα τα στοιχεία της στη μνήμη και ως εκ τούτου, δεν είναι δυνατόν να έχουμε έναν άπειρο αριθμό στοιχείων σε μια συλλογή, διότι θα απαιτούνταν άπειρη ποσότητα μνήμης. Ένα stream τραβάει στοιχεία από μια πηγή δεδομένων που μπορεί να είναι μια συλλογή, μια συνάρτηση που παράγει δεδομένα, ένα κανάλι I / O, κ.λπ. Επειδή μια συνάρτηση μπορεί να δημιουργήσει έναν άπειρο αριθμό στοιχείων και ένα stream μπορεί να τραβήξει δεδομένα από αυτήν, αν και όταν ζητηθεί, μπορεί να αναπαραστήσει μια άπειρη ακολουθία στοιχείων δεδομένων.
3. **Ο σχεδιασμός τους βασίζεται στην ιδέα της εσωτερικής επανάληψης:** Οι συλλογές βασίζονται στην εξωτερική επανάληψη. Τα streams είναι σχεδιασμένα να χρησιμοποιούν εσωτερική επανάληψη. Με το πέρασμα ενός αλγορίθμου, χρησιμοποιώντας εκφράσεις λάμδα, δίνονται οδηγίες στο stream για το τι πρέπει να κάνει. Το stream εφαρμόζει τον αλγόριθμο σε κάθε στοιχείο δεδομένων με την επανάληψη πάνω στα στοιχεία να γίνεται εσωτερικά και στο τέλος δίνει το αποτέλεσμα.
4. **Έχουν σχεδιαστεί για να επεξεργάζονται παράλληλα χωρίς πρόσθετη εργασία από τους προγραμματιστές:** Χρησιμοποιώντας εξωτερική επανάληψη, συνήθως, παράγεται σειριακός κώδικας. Δηλαδή, κώδικας που μπορεί να εκτελεστεί μόνο από ένα νήμα. Τα streams επεξεργάζονται τα στοιχεία τους, παράλληλα, χωρίς αυτό να γίνεται αντιληπτό από τον χρήστη. Αυτό δεν σημαίνει ότι το stream αποφασίζει αυτόματα για το αν η επεξεργασία των στοιχείων θα γίνει σε σειρά ή παράλληλα. Απλά αρκει να ζητηθεί από το stream να χρησιμοποιήσει παράλληλη επεξεργασία και αυτό θα φροντίσει για τα υπόλοιπα εσωτερικά.
5. **Υποστηρίζουν τον συναρτησιακό προγραμματισμό:** Οι συλλογές υποστηρίζουν τον επιτακτικό προγραμματισμό (imperative programming) ενώ τα streams υποστηρίζουν τον συναρτησιακό προγραμματισμό. Αυτό έχει να κάνει και με το ότι, οι συλλογές υποστηρίζουν την εξωτερική επανάληψη, ενώ τα streams την εσωτερική. Όταν χρησιμοποιούνται συλλογές, πρέπει ο προγραμματιστής να γνωρίζει το «τι» θέλει και το "πώς" να το πραγματοποιήσει. Όταν χρησιμοποιούνται streams,

πρέπει να καθοριστεί μόνο «τι» χρειάζεται να γίνει. Για το «πώς», φροντίζει το Streams API.

6. **Υποστηρίζουν «τεμπέλικες» λειτουργίες:** Οι σχεδιαστές της γλώσσας Java εμπλούτισαν το Streams API με μια εκτενή λίστα λειτουργιών που μπορούν να χρησιμοποιηθούν για να εκφράσουν πολύπλοκα ερωτήματα επεξεργασίας δεδομένων. Οι λειτουργίες αυτές ορίζονται στην βιβλιοθήκη java.util.stream.Stream της διεπαφής Stream και μπορούν να ταξινομηθούν σε δυο μεγάλες κατηγορίες:

· Ενδιάμεσες λειτουργίες, γνωστές και ως τεμπέλικες (*lazy*) λειτουργίες, που μπορούν να συνδεθούν μεταξύ τους

· Τερματικές λειτουργίες, γνωστές και ως πρόθυμες (*eager*) λειτουργίες που κλείνουν ένα stream

Σε ποιον τύπο ανήκει η κάθε λειτουργία, εξαρτάται από τον τρόπο που αντλούν τα στοιχεία δεδομένων από την πηγή δεδομένων. Οι ενδιάμεσες λειτουργίες επιστρέφουν ένα άλλο stream, ως τύπο επιστροφής. Αυτό επιτρέπει στις λειτουργίες να συνδέονται προκειμένου να σχηματίσουν ένα ερώτημα. Αυτό που είναι σημαντικό είναι ότι οι ενδιάμεσες λειτουργίες δεν επιτελούν καμία επεξεργασία στα δεδομένα, γι' αυτό και η ονομασία τεμπέλικες, μέχρι να κληθεί κάποια τερματική λειτουργία στη σωλήνωση των streams Αυτό οφείλεται στο γεγονός ότι οι ενδιάμεσες λειτουργίες μπορούν συνήθως να συγχωνευθούν και να υποβληθούν σε επεξεργασία σε ένα μόνο πέρασμα από μια τερματική λειτουργία. Οι τερματικές λειτουργίες παράγουν ένα αποτέλεσμα από μια σωλήνωση streams. Το αποτέλεσμα μπορεί να είναι οποιαδήποτε όχι-stream (nonstream) τιμή, όπως μια λίστα, ένας ακέραιος αριθμός, ή ακόμα και τίποτα (void). Οι λειτουργίες των streams έχουν δύο σημαντικά χαρακτηριστικά:

· Σωλήνωση (Pipelining): Πολλές stream λειτουργίες επιστρέφουν οι ίδιες ένα stream, επιτρέποντας στις λειτουργίες να αλληλοσυνδέονται και να σχηματίζουν μια μεγαλύτερη σωλήνωση. Αυτό επιτρέπει ορισμένες βελτιστοποιήσεις, όπως την τεμπελιά (*laziness*) και το βραχυκύκλωμα (short-circuiting).

· Εσωτερική επανάληψη (Internal iteration): Σε αντίθεση με τις συλλογές, οι οποίες επαναλαμβάνονται ρητά κάνοντας χρήση ενός επαναλήπτη (iterator), οι stream λειτουργίες εκτελούν την επανάληψη στο παρασκήνιο για χάρη του προγραμματιστή.

7. **Μπορούν να είναι διατεταγμένα ή όχι:** Ένα διατεταγμένο stream διατηρεί τη σειρά των στοιχείων του. Το Streams API επιτρέπει την μετατροπή ενός διατεταγμένου stream σε ένα μη διατεταγμένο. Ένα stream μπορεί να είναι διατεταγμένο επειδή αναπαριστά μια διατεταγμένη πηγή δεδομένων, όπως μια λίστα ή ένα ταξινομημένο σύνολο. Επίσης η μετατροπή ενός μη διατεταγμένου stream σε διατεταγμένο μπορεί να επιτευχτεί με την εφαρμογή μιας ενδιάμεσης λειτουργίας, όπως π.χ. είναι η sorting.

8. **Δεν μπορούν να επαναχρησιμοποιηθούν:** Σε αντίθεση με τις συλλογές, τα streams δεν είναι επαναχρησιμοποιήσιμα. Πρόκειται για μιας χρήσης (one-shot) αντικείμενα. Ένα stream δεν μπορεί να επαναχρησιμοποιηθεί μετά την κλήση μιας τερματικής λειτουργίας σε αυτό. Αν χρειαστεί να εκτελεστεί εκ νέου ένας υπολογισμός πάνω στα ίδια στοιχεία από την ίδια πηγή δεδομένων, θα πρέπει η σωλήνωση των stream να ξανά δημιουργηθεί.

Συνοψίζοντας, η χρήση streams περιλαμβάνει τρία στοιχεία:

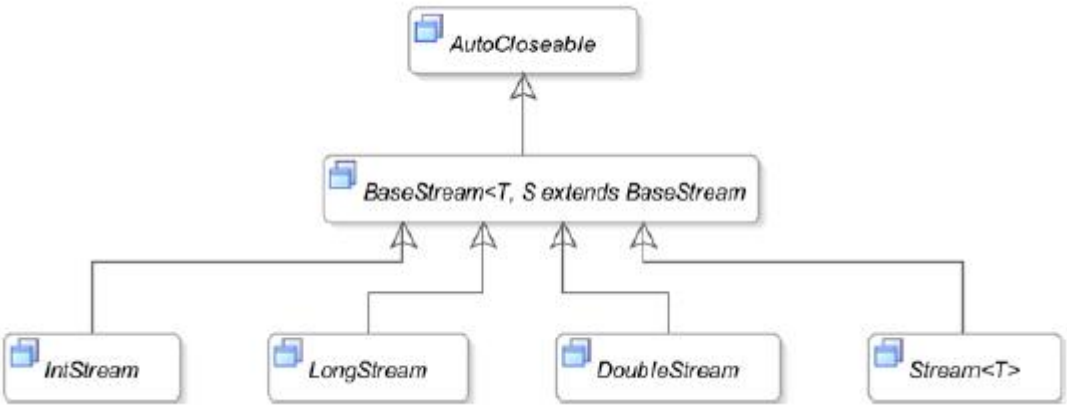
· Μία πηγή δεδομένων (όπως μια συλλογή) για να εκτελεστεί ένα ερώτημα σε αυτή.

· Μια αλυσίδα ενδιάμεσων λειτουργιών που σχηματίζουν μια σωλήνωση από streams.

· Μια τερματική λειτουργία που εκτελεί σωλήνωση από streams και παράγει ένα αποτέλεσμα.

3.2 Η ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΤΟΥ STREAMS API

Στην παρακάτω εικόνα φαίνεται το διάγραμμα κλάσεων για τις διεπαφές που σχετίζονται με τα streams. Οι διεπαφές και οι κλάσεις που σχετίζονται με streams βρίσκονται στο πακέτο java.util.stream.



Εικόνα 5: Διάγραμμα κλάσεων διεπαφών streams[1].

Όλες οι stream διεπαφές κληρονομούν από τη διεπαφή BaseStream, η οποία κληρονομεί από τη διεπαφή AutoCloseable από το πακέτο java.lang. Στην πράξη, τα περισσότερα streams χρησιμοποιούν συλλογές ως πηγή δεδομένων τους και οι συλλογές δεν είναι απαραίτητο να είναι κλειστές. Όταν ένα stream βασίζεται σε

κάποια πηγή δεδομένων που μπορεί να κλείσει (closeable), όπως ένα αρχείο I / O καναλιού, μπορεί να δημιουργηθεί ένα στιγμιότυπο stream, χρησιμοποιώντας μια δοκιμή-με-πόρους (try-with-resources) εντολή, για να κλείσει αυτόματα.

Οι μέθοδοι που είναι κοινές για όλους τους τύπους streams δηλώνονται στη διεπαφή BaseStream.

· Iterator <T> iterator ( ): Επιστρέφει έναν iterator για το stream. Πολύ σπάνια χρησιμοποιείται αυτή η μέθοδος στον κώδικά. Πρόκειται για τερματική λειτουργία. Μετά την κλήση αυτής της μεθόδου, δεν μπορεί να κληθεί κάποια άλλη μέθοδος πάνω στο stream.

· S sequential( ): Επιστρέφει ένα διαδοχικό stream. Εάν το stream είναι ήδη διαδοχικό, επιστρέφει το ίδιο. Η μέθοδος αυτή χρησιμοποιείται για να μετατρέψει ένα παράλληλο stream σε διαδοχικό. Πρόκειται για ενδιάμεση λειτουργία.

· S parallel( ): Επιστρέφει ένα παράλληλο stream. Εάν το stream είναι ήδη παράλληλο, επιστρέφει το ίδιο. Είναι μια ενδιάμεση λειτουργία που χρησιμοποιείται για να μετατρέψει ένα παράλληλο stream σε διαδοχικό.

· boolean isParallel( ): Επιστρέφει true αν το stream είναι παράλληλο, αλλιώς false. Το αποτέλεσμα είναι απρόβλεπτο, όταν η μέθοδος αυτή κληθεί μετά από την επίκληση μιας τερματικής stream λειτουργίας.

· S unordered( ): Επιστρέφει μια μη διατεταγμένη έκδοση του stream. Εάν το stream είναι ήδη μη διατεταγμένο, επιστρέφει το ίδιο. Είναι και αυτή μια ενδιάμεση λειτουργία.

Η Stream<T> διεπαφή αντιπροσωπεύει ένα stream ενός στοιχείου τύπου T. Να σημειωθεί ότι η Stream<T> διεπαφή δέχεται μια παράμετρο τύπου T, πράγμα που σημαίνει ότι μπορεί να χρησιμοποιηθεί μόνο για να επεξεργαστεί στοιχεία του τύπου αναφοράς. Εάν πρέπει να γίνει χρήση ενός stream με στοιχεία που ο τύπος τους είναι αρχικός (primitive type), όπως int, long και double, το Stream<T> θα συνεπάγεται επιπλέον κόστος για το boxing και το unboxing των στοιχείων όταν απαιτούνται αρχικές τιμές (primitive values). Η Java 8 εισάγει στο Streams API τρεις αρχικές εξειδικευμένες stream (primitive stream specializations) διεπαφές για την επεξεργασία streams που τα στοιχεία τους είναι αριθμοί, τις IntStream, DoubleStream, και LongStream, οι οποίες αντίστοιχα εξειδικεύουν τα στοιχεία του stream να είναι τύπου int, long και double και γλιτώνουν το κρυφό κόστος για το boxing. Κάθε μια από αυτές τις διεπαφές φέρνει νέες μεθόδους για την εκτέλεση κοινών αριθμητικών υπολογισμών. Επιπλέον, έχουν μεθόδους για τη μετατροπή ενός αριθμητικού stream πίσω στο αντίστοιχο stream αντικειμένων όταν είναι απαραίτητο. Αυτό που πρέπει να μείνει είναι ότι αυτές οι εξειδικεύσεις δεν έχουν μεγαλύτερη πολυπλοκότητα εξαιτίας των streams, αλλά λόγω του boxing.

3.3 ΔΗΜΙΟΥΡΓΩΝΤΑΣ STREAMS

Υπάρχουν πολλοί τρόποι για να δημιουργηθεί ένα stream[6]. Πολλές από τις υπάρχουσες κλάσεις στις βιβλιοθήκες της Java έχουν ανανεωθεί και έχουν προσθέσει

νέες μεθόδους που επιστρέφουν ένα stream. Με βάση την πηγή δεδομένων, η δημιουργία των streams μπορεί να κατηγοριοποιηθεί ως εξής:

- Streams από τιμές
- Κενά streams
- Streams από συναρτήσεις
- Streams από πίνακες
- Streams από συλλογές
- Streams από αρχεία
- Streams από άλλες πηγές

Streams από τιμές: Η Stream διεπαφή περιέχει τις παρακάτω δύο static of( ) μεθόδους για την δημιουργία ενός stream από μια μόνο τιμή και από πολλαπλές τιμές:

- <T> Stream<T> of(T t)
- <T> Stream<T> of(T...values)

Παράδειγμα:

// Δημιουργία stream με ένα στοιχείο τύπου string

- Stream<String> stream = Stream.of("Hello");

// Δημιουργία stream με τέσσερα στοιχεία τύπου string

- Stream<String> stream = Stream.of("Ken", "Jeff", "Chris", "Ellen");

Κενά streams: Ένα κενό stream είναι ένα stream χωρίς στοιχεία. Η Stream διεπαφή περιέχει την στατική μέθοδο empty( ) για την δημιουργία ενός κενού stream. Οι διεπαφές IntStream, LongStream, και DoubleStream περιέχουν επίσης μια στατική μέθοδο empty( ) για την δημιουργία ενός κενού stream αρχικών τύπων.

Παράδειγμα:

// Δημιουργία ενός κενού steam από συμβολοσειρές

- Stream<String> stream = Stream.empty();

// Δημιουργία ενός κενού steam από ακεραίους

- IntStream numbers = IntStream.empty();

Streams από συναρτήσεις: Ένα άπειρο stream είναι ένα stream με μια πηγή δεδομένων ικανή να παράγει άπειρο αριθμό στοιχείων. Προσοχή: η πηγή δεδομένων να είναι «σε θέση να παράγει" άπειρο αριθμό στοιχείων, όχι να περιέχει έναν άπειρο αριθμό στοιχείων. Είναι αδύνατη η δημιουργία και η αποθήκευση άπειρου αριθμού στοιχείων κάθε είδους, λόγω των περιορισμών σε μνήμη και χρόνο. Ωστόσο, είναι δυνατόν μια συνάρτηση να μπορεί να δημιουργήσει άπειρο αριθμό τιμών αν και όταν

ζητηθεί. Η Stream διεπαφή περιέχει δύο στατικές μεθόδους για τη δημιουργία ενός άπειρου stream:

· <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
· <T> Stream<T> generate(Supplier<T> s)

Η μέθοδος **iterator ( )** δημιουργεί ένα διατεταγμένο stream. Παίρνει δύο ορίσματα: ένα seed και μια συνάρτηση. Το πρώτο όρισμα είναι ένα seed που αντιστοιχεί στο πρώτο στοιχείο του stream. Το δεύτερο στοιχείο δημιουργείται από την εφαρμογή της συνάρτησης στο πρώτο στοιχείο. Το τρίτο στοιχείο δημιουργείται από την εφαρμογή της συνάρτησης στο δεύτερο και ούτω καθεξής.

Παράδειγμα:

// Δημιουργία ενός stream φυσικών αριθμών

· Stream<Long> naturalNumbers = Stream.iterate(1L, n ® n + 1);

// Δημιουργία ενός stream περιττών φυσικών αριθμών

· Stream<Long> oddNaturalNumbers = Stream.iterate(1L, n ® n + 2);

Η μέθοδος **generate(Supplier<T> s)** χρησιμοποιεί τον συγκεκριμένο Supplier για να δημιουργήσει ένα άπειρο μη διατεταγμένο stream.

Παράδειγμα:

//Εκτύπωση τυχαίων αριθμών ανάμεσα στο 0.0 και στο 1.0 με την βοήθεια της στατικής μεθόδου random( ) της κλασης Math

· Stream.generate(Math::random)
.forEach(System.out::println);

Οι διεπαφές IntStream, LongStream, και DoubleStream περιέχουν επίσης τις στατικές μεθόδους iterator( ) και generate( ) που παίρνουν συγκεκριμένες παραμέτρους ανάλογα με τον αρχικό τους τύπο.

Παράδειγμα:

· IntStream iterate(int seed, IntUnaryOperator f)
· IntStream generate(IntSupplier s)

Streams από πίνακες: Η κλάση Arrays στο πακέτο java.util περιλαμβάνει την **stream( )**, μια υπερφορτωμένη στατική μέθοδο για τη δημιουργία streams από πίνακες. Χρησιμοποιείται για την δημιουργία ενός IntStream από έναν int πίνακα, ενός LongStream από έναν long πίνακα, ενός DoubleStream από έναν double πίνακα και ενός Stream<T> από έναν πίνακα με τύπο αναφοράς T.

// Δημιουργία ενός stream από έναν int πίνακα με στοιχεία 1, 2, 3

· IntStream numbers = Arrays.stream(new int[ ]{1, 2, 3});
// Δημιουργία ενός stream από έναν String πίνακα με στοιχεία "Ken", "Jeff"

· Stream<String> names = Arrays.stream(new String[ ] {"Ken", "Jeff"});

Streams από συλλογές: Η διεπαφή Collection περιέχει τις μεθόδους **stream( )** και **parallelStream( )** που δημιουργούν διαδοχικά και παράλληλα streams από μια συλλογή, αντίστοιχα.

Παράδειγμα:

· import java.util.HashSet;
import java.util.Set;

import java.util.stream.Stream;

...

// Δημιουργία και να συμπλήρωση ενός συνόλου από strings

Set<String> names = new HashSet<>();

names.add("Ken");

names.add("jeff");

// Δημιουργία ενός διαδοχικού stream από το σύνολο

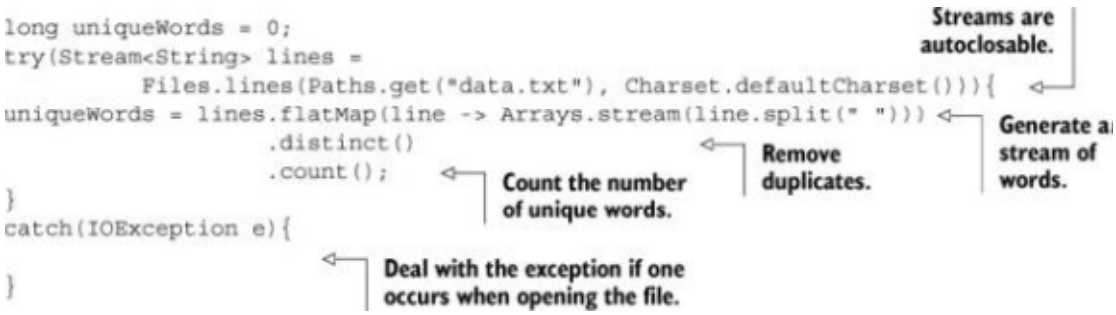
Stream<String> sequentialStream = names.stream();

// Δημιουργία ενός παράλληλου stream από το σύνολο

Stream<String> parallelStream = names.parallelStream();

Streams από αρχεία: Η Java 8 έχει προσθέσει πολλές μεθόδους στις κλάσεις στα πακέτα java.io και java.nio.file για να υποστηρίξει τις λειτουργίες I / O που χρησιμοποιούν streams.

Παράδειγμα:



Εικόνα 6: Χρήση της μεθόδου Files.lines. η οποία επιστρέφει ένα stream γραμμών ως συμβολοσειρές από ένα δεδομένο αρχείο. Η Files.lines μπορεί να

χρησιμοποιηθεί για την εύρεση του αριθμού των μοναδικών λέξεων σε ένα αρχείο[1].

Streams από άλλες πηγές: Η Java 8 έχει προσθέσει μεθόδους σε πολλές άλλες κλάσεις που επιστρέφουν το περιεχόμενο που αναπαριστούν σε ένα stream. Δύο από αυτές τις μεθόδους που χρησιμοποιούνται πιο συχνά είναι.

· Η μέθοδος chars( ) της διεπαφής CharSequence που επιστρέφει ένα IntStream του οποίου τα στοιχεία είναι int τιμές που αναπαριστούν τους χαρακτήρες της CharSequence.

Παράδειγμα:

- String str = "5 apples and 25 oranges";
str.chars()

.filter(n ® !Character.isDigit((char)n) &&
!Character.isWhitespace((char)n))

.forEach(n ® System.out.print((char)n));

//Αποτέλεσμα

applesandoranges

· Η μέθοδος splitAsStream(CharSequence input) της κλασης java.util.regex.Pattern που επιστρέφει ένα stream τύπου String, του οποίου τα στοιχεία ταιριάζουν με το συγκεκριμένο μοτίβο.

Παράδειγμα:

- String str = "Ken,Jeff,Lee";
Pattern.compile(",")

.splitAsStream(str)

.forEach(System.out::println);

//Αποτέλεσμα

Ken

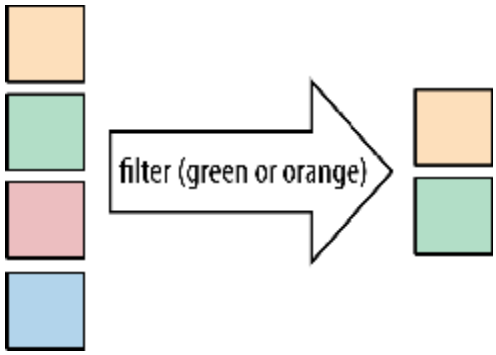
Jeff

Lee

3.4 ΛΕΙΤΟΥΡΓΙΕΣ - ΜΕΘΟΔΟΙ ΤΩΝ STREAMS

Σε αυτή την ενότητα θα παρουσιαστούν οι πιο συχνά χρησιμοποιούμενες μέθοδοι του Streams API, ο τύπος τους και μια σύντομη περιγραφή τους με κάποια αντιπροσωπευτικά παραδείγματα – όπου χρειάζεται – για καλύτερη κατανόηση.

Η διεπαφή Streams υποστηρίζει τη μέθοδο **filter** για την επιλογή στοιχείων από ένα stream. Η μέθοδος αυτή παίρνει ως όρισμα ένα κατηγόρημα (predicate), δηλαδή μια συνάρτηση που επιστρέφει μια τιμή τύπου boolean και επιστρέφει ένα stream, που περιέχει όλα τα στοιχεία που ταιριάζουν με το κατηγόρημα.



Εικόνα 7: Τρόπος λειτουργίας της μεθόδου filter[1].

Τα streams υποστηρίζουν επίσης μια μέθοδο που ονομάζεται **distinct** η οποία επιστρέφει ένα stream με μοναδικά στοιχεία. Άλλη μια μέθοδος είναι η **limit(n)**, η οποία επιστρέφει ένα άλλο stream μεγέθους το πολύ n στοιχείων. Το απαιτούμενο μέγεθος n, περνάει ως όρισμα για τον περιορισμό. Εάν το αρχικό stream έχει τα στοιχεία του διατεταγμένα, τα πρώτα στοιχεία επιστρέφονται μέχρι το μέγιστο του n. Η limit(n) λειτουργεί και για steams που τα στοιχεία τους δεν είναι διατεταγμένα. Η μέθοδος **skip(n)** επιστρέφει ένα stream που απορρίπτει τα πρώτα n στοιχεία. Εάν το αρχικό stream περιέχει λιγότερα στοιχεία από n, τότε επιστρέφεται ένα άδειο stream. Να σημειωθεί ότι οι μέθοδοι limit(n) και skip(n) είναι συμπληρωματικές.

Παράδειγμα:

```
· Stream.iterate(2L, PrimeUtil::next)
.skip(100)

.limit(5)

.forEach(System.out::println);
```

//Αποτελέσματα

547

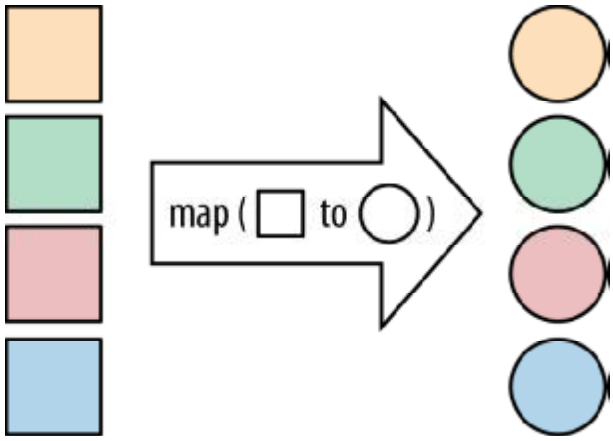
557

563

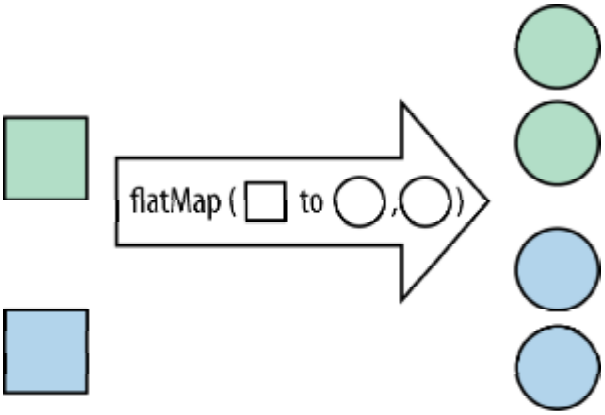
569

571

Μια πολύ κοινή διαδικασία επεξεργασίας δεδομένων είναι η συλλογή πληροφοριών από συγκεκριμένα αντικείμενα. Το streams API παρέχει αυτή τη δυνατότητα μέσα από τις μεθόδους map και flatMap. Η μέθοδος **map**, παίρνει μια συνάρτηση ως όρισμα. Η συνάρτηση εφαρμόζεται σε κάθε στοιχείο, το οποίο το αντιστοιχίζει (mapping) σε ένα νέο στοιχείο. Η μέθοδος **flatMap** αντιστοιχίζει κάθε πίνακα όχι με ένα stream, αλλά με τα περιεχόμενα του εν λόγω stream. Με λίγα λόγια, η μέθοδος flatMap επιτρέπει την αντικατάσταση κάθε τιμής του stream με ένα άλλο stream και στη συνέχεια συνενώνει όλα τα stream που δημιούργησε σε ένα ενιαίο μοναδικό stream.

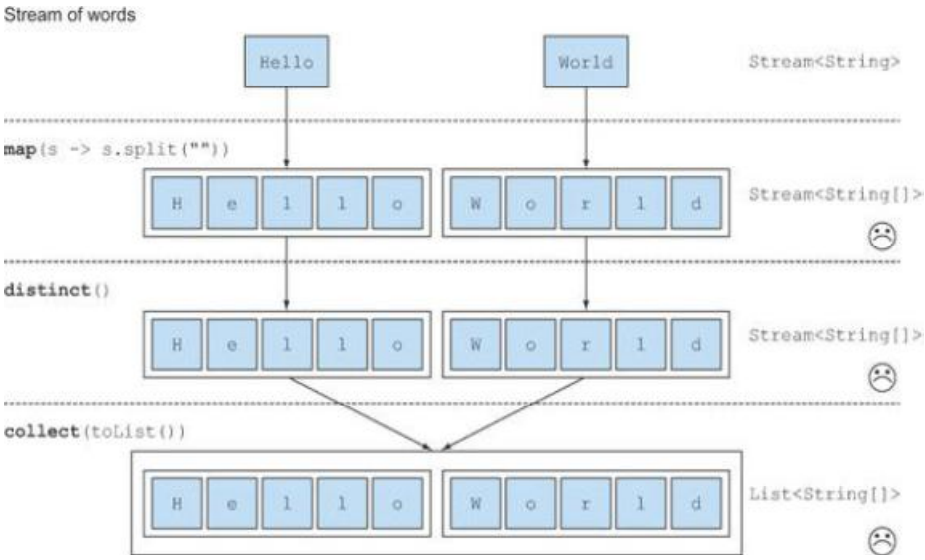


Εικόνα 8: Τρόπος λειτουργίας της μεθόδου map[1].

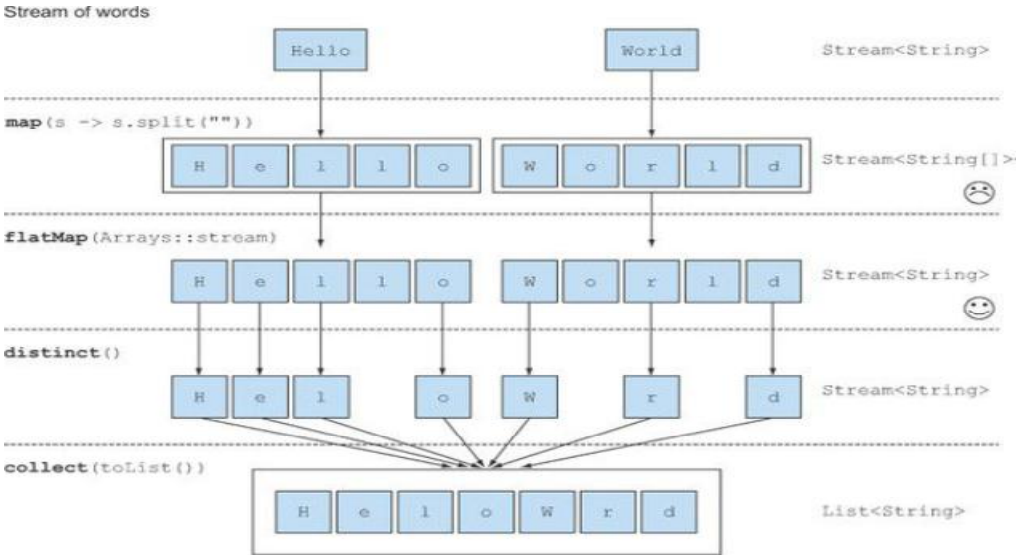


Εικόνα 9: Τρόπος λειτουργίας της μεθόδου flatMap[1]

Παράδειγμα:



Εικόνα 10: Λανθασμένη χρήση του map για την εύρεση των μοναδικών χαρακτήρων από μια λίστα με λέξεις[1].



Εικόνα 11: Χρήση του flatMap για την εύρεση των μοναδικών χαρακτήρων από μια λίστα με λέξεις[1].

Σε ένα stream μπορεί να εφαρμοστεί μια σειρά από μεθόδους. Κάθε μέθοδος μετατρέπει τα στοιχεία του stream εισόδου, παράγοντας ένα άλλο stream ή ένα αποτέλεσμα. Μερικές φορές χρειάζεται να εξεταστούν τα στοιχεία των streams καθώς αυτά περνούν μέσω μιας σωλήνωσης. Αυτό επιτυγχάνεται με την μέθοδο **peek**(Consumer<? super T> action) της διεπαφής Stream<T> που προορίζεται μόνο για σκοπούς εντοπισμού σφαλμάτων. Παράγει ένα stream μετά την εφαρμογή μιας ενέργειας σε κάθε στοιχείο εισόδου.

Παράδειγμα:

· int sum = Stream.of(1, 2, 3, 4, 5)
.peek(e ® System.out.println("Taking integer: " + e))
.filter(n ® n % 2 = = 1)
.peek(e ® System.out.println("Filtered integer: " + e))
.map(n ® n * n)
.peek(e ® System.out.println("Mapped integer: " + e))
.reduce(0, Integer::sum);

System.out.println("Sum = " + sum);

//Αποτελέσματα: οι ζυγοί αριθμοί που λαμβάνονται από την πηγή δεδομένων, αλλά δεν περνούν από την filter.

Taking integer: 1

Filtered integer: 1

Mapped integer: 1

Εικόνα 12: Οι λειτουργίες των μεθόδων filter, map και reduce σε σωλήνωση.

Taking integer: 2

Taking integer: 3

Filtered integer: 3

Mapped integer: 9

Taking integer: 4

Taking integer: 5

Filtered integer: 5

Mapped integer: 25

Sum = 35

Μια ακόμη κοινή διαδικασία επεξεργασίας δεδομένων είναι η εξέταση για το αν ορισμένα στοιχεία ενός συνόλου δεδομένων έχουν μια συγκεκριμένη ιδιότητα. Το streams API παρέχει τέτοιου είδους διευκολύνσεις μέσω των μεθόδων allMatch, anyMatch, noneMatch, findFirst, και findAny οι οποίες παρουσιάζονται στη συνέχεια αναλυτικά.

Η μέθοδος **anyMatch** μπορεί να χρησιμοποιηθεί για να απαντήσει στο αν υπάρχει ένα στοιχείο μέσα στο stream που να ταιριάζει (matching) με κάποιο συγκεκριμένο κατηγόρημα. Η μέθοδος anyMatch επιστρέφει μια τιμή τύπου boolean επομένως είναι μια τερματική λειτουργία. Η μέθοδος **allMatch** λειτουργεί παρόμοια με την anyMatch αλλά ελέγχει να δει αν όλα τα στοιχεία του stream ταιριάζουν με το δεδομένο κατηγόρημα. Η μέθοδος **noneMatch** κάνει ακριβώς το αντίθετο από την allMatch. Εξασφαλίζει ότι δεν υπάρχει κανένα στοιχείο του stream που να ταιριάζει με το δεδομένο κατηγόρημα. Οι τρεις αυτές μέθοδοι anyMatch, allMatch, και noneMatch κάνουν χρήση αυτού που αποκαλείται βραχυκύκλωμα (short-circuiting). Δεν χρειάζεται να επεξεργαστούν όλα τα δεδομένα του stream προκειμένου να καταλήξουν σε κάποιο αποτέλεσμα. Μόλις βρεθεί ένα στοιχείο που δεν ταιριάζει με το κατηγόρημα, μπορεί να δοθεί αποτέλεσμα αγνοώντας τα υπόλοιπα στοιχεία.

- boolean allMatch(Predicate<? super T> predicate)
- boolean anyMatch(Predicate<? super T> predicate)
- boolean noneMatch(Predicate<? super T> predicate)

Η μέθοδος **findAny** επιστρέφει ένα αυθαίρετο στοιχείο του τρέχοντος stream. Μπορεί να χρησιμοποιηθεί σε συνδυασμό με άλλες λειτουργίες των stream ανάλογα με το εκάστοτε πρόβλημα. Μερικά streams έχουν μια προκαθορισμένη εσωτερική διάταξη (encounter order) που καθορίζει τη σειρά με την οποία λογικά εμφανίζονται τα στοιχεία μέσα στο stream. Σε τέτοια streams μπορεί να χρειάζεται να βρεθεί το πρώτο στοιχείο. Γι' αυτό υπάρχει η μέθοδος **findFirst**, η οποία λειτουργεί με παρόμοιο τρόπο με την findAny. Για το ποια από τις findFirst και findAny θα χρησιμοποιείται κάθε φορά εξαρτάται από το εάν χρειάζεται παραλληλισμός. Η εύρεση του πρώτου στοιχείου είναι πιο περιοριστική κατά την παράλληλη εκτέλεση. Αν δεν υπάρχει θέμα για το ποιο στοιχείο θα επιστρέφεται, χρησιμοποιείται η findAny επειδή είναι λιγότερο περιοριστική όταν χρησιμοποιούνται παράλληλα streams.

- Optional<T> findAny()
- Optional<T> findFirst()

Παράδειγμα:

// FindAndMatch.java

```
· package com.jdojo.streams;
import java.util.List;

    import java.util.Optional;

public class FindAndMatch {

    public static void main(String[ ] args) {

        // Η λίστα των προσώπων
```

```
List<Person> persons = Person.persons();
// Έλεγχος αν όλα τα πρόσωπα είναι άνδρες
boolean allMales = persons.stream()
        .allMatch(Person::isMale);
System.out.println("All males: " + allMales);
// Έλεγχος εάν κάποιο πρόσωπο γεννήθηκε το 1970
boolean anyoneBornIn1970 =
        persons.stream()
                .anyMatch(p -> p.getDob().getYear() == 1970);
                        System.out.println("Anyone born in
                    1970: " + anyoneBornIn1970);
// Έλεγχος εάν κάποιο πρόσωπο γεννήθηκε το 1955
boolean anyoneBornIn1955 =
        persons.stream()
                .anyMatch(p -> p.getDob().getYear() == 1955);
                        System.out.println("Anyone born in
                    1955: " + anyoneBornIn1955);
// Εύρεση κάθε άντρα
Optional<Person> anyMale = persons.stream()
                                .filter(Person::isMale)
                                .findAny();
if (anyMale.isPresent()) {
        System.out.println("Any male: " + anyMale.get());
}
else {
        System.out.println("No male found.");
}
// Εύρεση του πρώτου άντρα στη λίστα
Optional<Person> firstMale = persons.stream()
                        .filter(Person::isMale)
```

```
                .findFirst();
        if (firstMale.isPresent()) {
            System.out.println("First male: " + anyMale.get());
        }
        else {
            System.out.println("No male found.");
        }
    }
}
//Αποτελέσματα
    All males: false
    Anyone born in 1970: true
    Anyone born in 1955: false
    Any male: (1, Ken, MALE, 1970-05-04, 6000.00)
    First male: (1, Ken, MALE, 1970-05-04, 6000.00)
```

Η λειτουργία **foreach** εκτελεί μια ενέργεια για κάθε στοιχείο του stream. Η Stream<T> διεπαφή περιέχει δύο μεθόδους για την εκτέλεση της foreach:

- void forEach(Consumer<? super T> action)
- void forEachOrdered(Consumer<? super T> action)

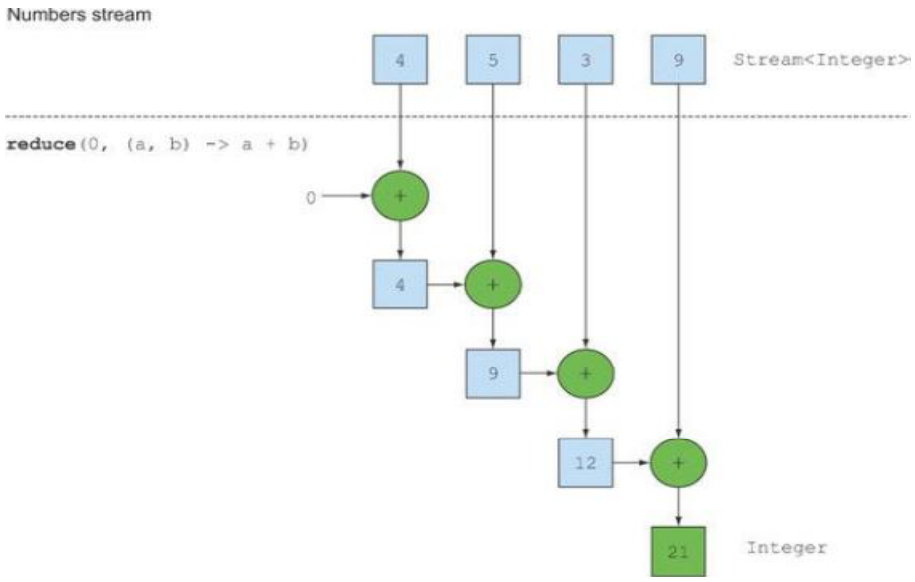
Μια άλλη δυνατότητα που υπάρχει είναι να συνδυαστούν τα στοιχεία του stream για να εκφράσουν σύνθετα ερωτήματα. Τέτοια ερωτήματα συνδυάζουν όλα τα στοιχεία στο stream επανειλημμένα για να παράγουν μια ενιαία τιμή όπως έναν ακέραιο. Αυτά τα ερωτήματα μπορούν να χαρακτηριστούν ως πράξεις-λειτουργίες μείωσης (reduction operations).

Η μέθοδος **reduce** παίρνει δυο ορίσματα:

- Μια αρχική τιμή
- Ένα BinaryOperator<T> για να συνδυαστούν δυο τιμές και να παραχθεί μια νέα

Παράδειγμα:

- int sum = numbers.stream().reduce(0, (a, b) ® a + b);



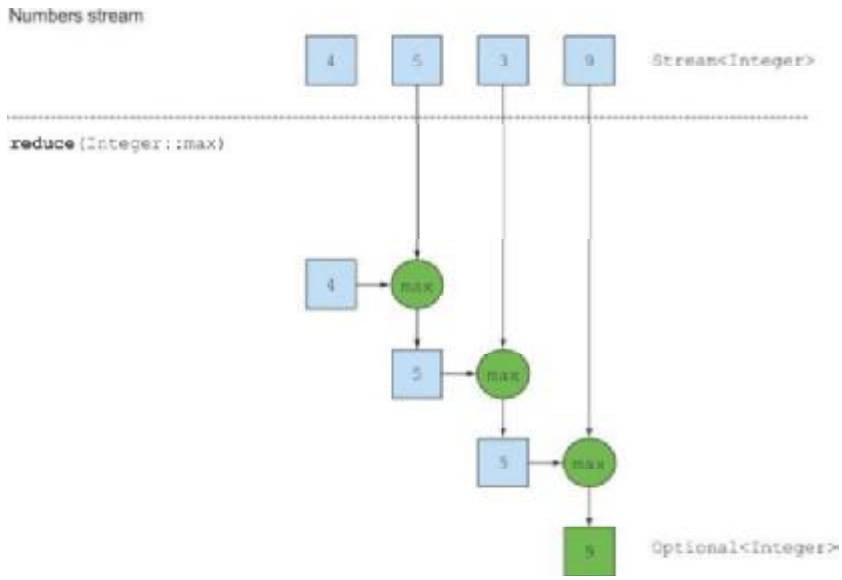
Εικόνα 13: Το άθροισμα όλων των στοιχείων ενός stream μπορεί να υπολογιστεί με την μέθοδο reduce[1].

Ο παραπάνω κώδικας μπορεί να γράφει πιο συνοπτικά, χρησιμοποιώντας μια μέθοδο αναφοράς αντί για την έκφραση λάμδα. Στην Java 8, η κλάση Integer προσφέρει μια στατική μέθοδο άθροισης για να προσθέτει δύο αριθμούς.

· int sum = numbers.stream( ).reduce(0, Integer::sum);

Η μέθοδος reduce μπορεί επίσης να χρησιμοποιηθεί για τον υπολογισμό του μέγιστου και του ελάχιστου. Αρκει η έκφραση λάμδα που θα περάσει ως παράμετρος να παίρνει δύο στοιχεία και να επιστρέφει το μέγιστο (ή αντίστοιχα το ελάχιστο) από αυτά. Η μέθοδος reduce θα χρησιμοποιήσει τη νέα τιμή με το επόμενο στοιχείο του stream ώστε να παραχθεί ένα νέο μέγιστο/ελάχιστο όριο μέχρι ότου τελειώσουν όλα τα στοιχεία του stream.

Παράδειγμα:



Εικόνα 14: Χρήση της μεθόδου reduce για την εύρεση του μεγίστου[1]

Το όφελος από τη χρήση της reduce σε σύγκριση με την επαναληπτική βήμα προς βήμα διαδικασία υπολογισμού π.χ. για το άθροισμα, είναι ότι η επανάληψη γίνεται με εσωτερική επανάληψη και είναι αφηρημένη, πράγμα που επιτρέπει στην εσωτερική υλοποίηση να επιλέξει να εκτελέσει τη λειτουργία της reduce παράλληλα.

Οι μέθοδοι map και filter παίρνουν κάθε στοιχείο από το stream εισόδου και παράγουν μηδέν ή κάποιο αποτέλεσμα στο stream εξόδου. Οι μέθοδοι αυτές είναι έτσι σε γενικές γραμμές stateless: δεν έχουν εσωτερική κατάσταση. Αντιθέτως ορισμένες μέθοδοι, όπως η sorted ή η distinct ενώ φαίνονται εκ πρώτης να συμπεριφέρονται όπως η map και η filter, όλες παίρνουν ένα stream και παραγάγουν ένα άλλο stream, εντούτοις υπάρχει μια σημαντική διαφορά. Η sorted και η distinct χρειάζεται να γνωρίζουν την «προηγούμενη ιστορία» του stream για να κάνουν τη δουλειά τους. παραδείγματος χάριν για να δουλέψει η sorted πρέπει πρώτα όλα τα στοιχεία του stream να έχουν ρυθμιστεί (all the elements to be buffered). Αυτό μπορεί να προκαλέσει προβλήματα εάν τα δεδομένα του stream είναι μεγάλα ή ακόμη και άπειρα. Οι μέθοδοι αυτές ονομάζονται stateful. Τέλος αν η εσωτερική κατάσταση είναι μικρή οι μέθοδοι είναι φραγμένου μεγέθους (bounded size) αλλιώς εάν υπάρχει απαίτηση για μεγάλο αποθηκευτικό χώρο, τότε οι μέθοδοι είναι μη-φραγμένου μεγέθους (unbounded size).

Πίνακας 6: Οι ενδιάμεσες και τερματικές stream μέθοδοι.

Μέθοδος	**Τύπος**	**Επιστρεφόμενος τύπος**	**Λειτουργική διεπαφή**	**Λειτουργικός περιγραφέας**
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
distinct	Intermediate (statef	Stream<T>		

	ul-unbounded)			
skip	Intermediate (stateful-bounded)	Stream<T>	long	
limit	Intermediate (stateful-bounded)	Stream<T>	long	
map	Intermediate	Stream<R>	Function<T, R>	T -> R
flatMap	Intermediate	Stream<R>	Function<T, Stream<R>>	T -> Stream<R>
sorted	Intermediate (stateful-unbounded)	Stream<T>	Comparator<T>	(T, T) -> int
anyMatch	Terminal	boolean	Predicate<T>	T -> boolean
noneMatch	Terminal	boolean	Predicate<T>	T -> boolean
allMatch	Terminal	boolean	Predicate<T>	T -> boolean
findAny	Terminal	Optional<T>		
findFirst	Terminal	Optional<T>		
forEach	Terminal	void	Consumer<T>	T -> void
collect	Terminal	R	Collector<T, A, R>	
reduce	Terminal (stateful-bounded)	Optional<T>	BinaryOperator<T>	(T, T) -> T
count	Terminal	long		

Μια συνηθισμένη ανάγκη όταν υπάρχουν αριθμοί, είναι η ανάγκη για ένα εύρος από αριθμητικές τιμές. Η Java 8 εισάγει δύο στατικές μεθόδους που διατίθενται στην IntStream και LongStream και βοηθούν στη δημιουργία τέτοιων διαστημάτων αριθμών: την **range** και την **rangeClosed**. Και οι δύο μέθοδοι παίρνουν την αρχική τιμή του εύρους ως πρώτη παράμετρο και την τελική τιμή του ως δεύτερη παράμετρο. Η range παίρνει το ανοικτό διάστημα των αριθμών ενώ rangeClosed το κλειστό.

Παράδειγμα:

Εικόνα 15: Χρήση της rangeClosed για την παραγωγή όλων των αριθμών από το 1 έως και το 100[1].

Λίγα λόγια για την κλάση Optional<T> για μόνο για λόγους πληρότητας.

Η κλάση Optional<T> (java.util.Optional) είναι μια κλάση που αναπαριστά την ύπαρξη ή την απουσία μιας τιμής. Υπάρχουν διαθέσιμες μέθοδοι στην κλάση αυτή, που κάνουν αυστηρούς ελέγχους για την παρουσία ή την απουσία κάποιας τιμής και αναλόγως εκτελούν τις προβλεπόμενες ενέργειες:

· Η isPresent ( ) επιστρέφει true αν η Optional περιέχει μια τιμή, αλλιώς false.

· Η ifPresent(Consumer<T> block) εκτελεί το block, αν μια τιμή υπάρχει.

· Η T get( ) επιστρέφει την τιμή εάν αυτή υπάρχει, αλλιώς «πετάει» την αντίστοιχη εξαίρεση NoSuchElement-Exception.

· Η T orElse(T other) επιστρέφει την τιμή εάν υπάρχει, αλλιώς επιστρέφει μια προεπιλεγμένη τιμή.

3.5 ΣΥΛΛΕΓΟΝΤΑΣ ΔΕΔΟΜΕΝΑ ΜΕ STREAMS

Η μέθοδος collect είναι μια λειτουργία μείωσης, όπως η reduce, η οποία παίρνει ως παραμέτρους διάφορες «συνταγές» (recipes) για τη συγκέντρωση των στοιχείων ενός stream σε ένα συνοπτικό αποτέλεσμα. Αυτές οι «συνταγές» ορίζονται από μια νέα διεπαφή που ονομάζεται Collector και ονομάζονται collectors.

Αυτή η παρατήρηση (ότι δηλαδή οι collectors μπορούν να θεωρηθούν ως προχωρημένες λειτουργίες μείωσης) αναδεικνύει ακόμη ένα όφελος του καλά σχεδιασμένου συναρτησιακού API, τον υψηλότερο βαθμό συνθετικότητας και επαναχρησιμοποίησης. Οι collectors είναι εξαιρετικά χρήσιμοι επειδή παρέχουν έναν συνοπτικό αλλά ταυτόχρονα ευέλικτο τρόπο καθορισμού των κριτηρίων που η μέθοδος collect χρησιμοποιεί για την παραγωγή του αποτελέσματος της συλλογής. Πιο συγκεκριμένα, η κλήση της μεθόδου collect πάνω σε ένα stream ενεργοποιεί μια λειτουργία μείωσης (με βάση τις παραμέτρους, τους collectors) πάνω στα στοιχεία του ίδιου του stream.

Τυπικά, ο Collector εφαρμόζει μια λειτουργία μετασχηματισμού σε ένα και συγκεντρώνει χωρίς τάξη το αποτέλεσμα σε μια δομή δεδομένων που σχηματίζει το τελικό αποτέλεσμα εξόδου αυτής της διαδικασίας.

Η υλοποίηση των μεθόδων της διεπαφής Collector καθορίζει τον τρόπο με τον οποίο θα εκτελεστεί μια λειτουργία μείωσης στο stream. Η κλάση Collector παρέχει

πολλές στατικές «εργοστασιακές» μεθόδους για την εύκολη δημιουργία στιγμιότυπων για τους πιο κοινούς collectors που είναι διαθέσιμοι. Ο πιο απλός και συχνά χρησιμοποιούμενος collector είναι η στατική μέθοδος **toList**, η οποία συγκεντρώνει όλα τα στοιχεία ενός stream σε μια λίστα:

· List<Transaction> transactions =
transactionStream.collect(Collectors.toList());

Παράδειγμα:

```
· List<String> names = Person.persons()
                        .stream()
                        .map(Person::getName)
                        .collect(Collectors.toList());
System.out.println(names);
//Αποτέλεσμα
[Ken, Jeff, Donna, Chris, Laynie, Li]
```

Οι προκαθορισμένοι collectors (Predefined collectors), μπορούν να δημιουργηθούν από τις μεθόδους που παρέχονται από την κλάση Collectors.

Οι collectors αυτοί προσφέρουν τρεις βασικές λειτουργίες οι οποίες θα περιγραφούν αναλυτικά στη συνέχεια:

· Μείωση και σύνοψη των στοιχείων ενός stream σε μια μοναδική τιμή
· Ομαδοποίηση στοιχείων ενός stream
· Διαμέριση στοιχείων ενός stream

Οι collectors, οι παράμετροι δηλαδή της stream μεθόδου collect, χρησιμοποιούνται συνήθως σε περιπτώσεις που είναι απαραίτητο να αναδιοργανωθούν τα στοιχεία του stream σε μια συλλογή. Αλλά γενικότερα, μπορούν να χρησιμοποιούνται κάθε φορά που χρειάζεται να συνδυαστούν όλα τα στοιχεία του stream σε ένα μοναδικό αποτέλεσμα. Το αποτέλεσμα μπορεί να είναι οποιουδήποτε τύπου.

Όλες οι στατικές μέθοδοι της κλασης Collectors είναι διαθέσιμες με την εισαγωγή της βιβλιοθήκης java.util.stream.Collectors. Παραδείγματος χάριν, εφόσον έχουμε εισάγει τη βιβλιοθήκη java.util.stream.Collectors στην αρχή ενός προγράμματος και θέλουμε π.χ. να καλέσουμε την counting μέθοδο, γράφουμε counting( ) αντί για Collectors.counting( ).

Δύο collectors, οι **Collectors.maxBy** και **Collectors.minBy**, χρησιμοποιούνται για να υπολογιστεί η μέγιστη ή η ελάχιστη τιμή σε ένα stream. Αυτοί οι δύο collectors παίρνουν ως όρισμα έναν Comparator για να συγκρίνουν τα στοιχεία που βρίσκονται

μέσα στο stream. Μια άλλη κοινή λειτουργία μείωσης που επιστρέφει μόνο μία τιμή είναι το άθροισμα των τιμών ενός αριθμητικού πεδίου αντικειμένων σε ένα stream, ή η εύρεση του μέσου όρου των τιμών κτλ. Οι λειτουργίες αυτές ονομάζονται λειτουργίες σύνοψης (summarization operations).

Η κλάση Collectors παρέχει μια συγκεκριμένη μέθοδο για την άθροιση, την **Collectors.summingInt**. Η μέθοδος αυτή, δέχεται ως όρισμα μια συνάρτηση που αντιστοιχεί ένα αντικείμενο σε έναν ακέραιο (int), ο οποίος, αθροίζεται και να επιστρέφει έναν collector, ο οποίος, όταν περάσει ως παράμετρος στην συνηθισμένη μέθοδο collect, εκτελεί την ζητούμενη σύνοψη (summarization). Οι μέθοδοι **Collectors.summingLong** και **Collectors.summingDouble** συμπεριφέρονται ακριβώς με τον ίδιο τρόπο και μπορούν να χρησιμοποιηθούν αντίστοιχα εκεί όπου το πεδίο που προορίζεται για άθροιση είναι τύπου long ή double. Για τον υπολογισμό του μέσου όρου ενός συνόλου τιμών υπάρχουν οι μέθοδοι, **Collectors.averagingInt**, **Collectors.averagingLong** και **Collectors.averagingDouble**.

Συχνά, μπορεί να χρειάζεται να ανακτηθούν δύο ή περισσότερα αποτελέσματα που προκύπτουν από την χρήση των collectors. Αυτό μπορεί να επιτευχθεί κάνοντας χρήση μόνο μιας μεθόδου. Σε αυτήν την περίπτωση, μπορεί να χρησιμοποιηθεί ο collector που επιστρέφεται από τη μέθοδο **summarizingInt**. Αυτός ο collector συγκεντρώνει όλες τις πληροφορίες σε μια κλάση που ονομάζεται **IntSummaryStatistics** που παρέχει βολικές μεθόδους ανάκτησης (getter methods) για την πρόσβαση στα αποτελέσματα. Αντίστοιχα υπάρχουν και οι μέθοδοι **summarizingLong** και **summarizingDouble** με τους συναφείς τύπους **LongSummaryStatistics** και **DoubleSummaryStatistics** που χρησιμοποιούνται όταν τα στοιχεία για συλλογή είναι τύπου long και double.

Παράδειγμα:

```
// SummaryStats.java

        · package com.jdojo.streams;
import java.util.DoubleSummaryStatistics;

public class SummaryStats {

public static void main(String[] args) {

        DoubleSummaryStatistics stats = new DoubleSummaryStatistics();

                stats.accept(100.0);

        stats.accept(500.0);

        stats.accept(400.0);

        // Λήψη στατιστικών

        long count = stats.getCount();
```

```
        double sum = stats.getSum();
        double min = stats.getMin();
        double avg = stats.getAverage();
        double max = stats.getMax();
                System.out.printf("count=%d, sum=%.2f, min=%.2f,
            average=%.2f, max=%.2f%n", count, sum, min, max, avg);
        }
      }
```

//Αποτελέσματα

count=3, sum=1000.00, min=100.00, average=500.00, max=333.33

Η μέθοδος **joining** συνενώνει σε μια ενιαία συμβολοσειρά όλες τις συμβολοσειρές που προκύπτουν από την κλήση της μεθόδου toString σε κάθε αντικείμενο μέσα στο stream. Να σημειωθεί ότι η joining στο εσωτερικό της κάνει χρήση ενός StringBuilder για να προσθέσει τις παραγόμενες συμβολοσειρές σε μία. Επίσης η μέθοδος joining έχει μια υπερφορτωμένη έκδοση που δέχεται μια συμβολοσειρά-οριοθέτη μεταξύ δύο διαδοχικών στοιχείων, ώστε η λίστα που προκύπτει να είναι διαχωρισμένη.

- joining( )
- joining(CharSequence delimiter)
- joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)

Παράδειγμα:

// CollectJoiningTest.java

```
· package com.jdojo.streams;
import java.util.List;
        import java.util.stream.Collectors;
public class CollectJoiningTest {
        public static void main(String[ ] args) {
                List<Person> persons = Person.persons();
                        String names = persons.stream()
                        .map(Person::getName)
                        .collect(Collectors.joining());
                String delimitedNames = persons.stream()
```

```
                .map(Person::getName)
                .collect(Collectors.joining(", "));
        String prefixedNames = persons.stream()
            .map(Person::getName)
            .collect(Collectors.joining(", ", "Hello ", ". Goodbye."));
        System.out.println("Joined names: " + names);
                System.out.println("Joined,
            delimited names: " + delimitedNames);
        System.out.println(prefixedNames);
    }
}
//Αποτελέσματα
Joined names: KenJeffDonnaChrisLaynieLi
Joined, delimited names: Ken, Jeff, Donna, Chris, Laynie, Li
Hello Ken, Jeff, Donna, Chris, Laynie, Li. Goodbye.
```

Όλοι οι collectors που συζητήθηκαν μέχρι στιγμής είναι, στην πραγματικότητα, είναι βολικές εξειδικεύσεις της διαδικασίας μείωσης που ορίζονται με τη χρήση της μεθόδου reduce. Η μέθοδος Collectors.reducing αποτελεί μια γενίκευση όλων αυτών.

Η μέθοδος Collectors.reducing παίρνει ως ορίσματα τρεις παραμέτρους:

· Το πρώτο όρισμα είναι η τιμή εκκίνησης της λειτουργίας μείωσης και θα είναι, επίσης, η τιμή που επιστρέφεται στην περίπτωση που το stream δεν έχει στοιχεία

· Το δεύτερο όρισμα είναι η ίδια λειτουργία

· Το τρίτο όρισμα είναι ένας δυαδικός τελεστής BinaryOperator που συγκεντρώνει δύο στοιχεία σε μια μοναδική τιμή του ίδιου τύπου.

Υπάρχουν κάποιες διάφορες μεταξύ των μεθόδων collect και reduce της Stream διεπαφής.

· Σημασιολογική διαφορά: η μέθοδος reduce συνδυάζει δύο τιμές και παράγει μια νέα. Αυτή είναι μια αμετάβλητη μείωση. Σε αντίθεση, η μέθοδος collect μεταλλάσει ένα δοχείο (container) ώστε να συγκεντρώσει το αποτέλεσμα που παράγει.

· Πρακτική διαφορά: η μέθοδος reduce δεν είναι κατάλληλη για παραλληλισμό ενώ η μέθοδος collect είναι χρήσιμη για να εκφράσει εργασίες μείωσης πάνω σε ένα ευμετάβλητο δοχείο (mutable container) με τρόπο φιλικό προς τον παραλληλισμό.

Μια κοινή λειτουργία των βάσεων δεδομένων είναι η ομαδοποίηση (grouping) των στοιχείων σε ένα σύνολο, με βάση μια ή περισσότερες ιδιότητες. Αυτή η εργασία μπορεί εύκολα να εκτελεστεί με την μέθοδο **Collectors.groupingBy**. Η μέθοδος παίρνει ως παράμετρο μια συνάρτηση με τη μορφή αναφοράς μεθόδου. Όταν η συνάρτηση χρησιμοποιείται για να κατηγοριοποιήσει τα στοιχεία του stream σε διαφορετικές ομάδες ονομάζεται συνάρτηση ομαδοποίησης (classification function). Το αποτέλεσμα της λειτουργίας ομαδοποίησης είναι τύπου Map που έχει ως κλειδί την τιμή που επιστρέφεται από τη συνάρτηση ομαδοποίησης και ως αντίστοιχη τιμή, την λίστα όλων των στοιχείων του stream που έχει αυτή την διαβαθμισμένη τιμή. Δεν αρκει πάντα μια συνάρτηση ομαδοποίησης ως αναφορά μεθόδου, γιατί μπορεί να χρειάζεται τα στοιχεία να ομαδοποιηθούν με βάση μια πιο σύνθετη ιδιότητα.

· groupingBy(Function<? super T,? extends K> classifier)
· groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)
· groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)

Παράδειγμα:

· Map<Person.Gender, List<Person>> personsByGender =
Person.persons()

.stream()

.collect(Collectors.groupingBy(Person::getGender));

System.out.println(personsByGender);

//Αποτελέσματα

{FEMALE=[(3, Donna, FEMALE, 1962-07-29, 8700.00), (5, Laynie, FEMALE, 2012-12-13, 0.00)],

MALE=[(1, Ken, MALE, 1970-05-04, 6000.00), (2, Jeff, MALE, 1970-07-15, 7100.00), (4, Chris, MALE,

1993-12-16, 1800.00), (6, Li, MALE, 2001-05-09, 2400.00)]}

Η λειτουργία της ομαδοποίησης είναι ισχυρή, διότι μπορεί να συνδυαστεί αποτελεσματικά. Είναι δυνατή μια πολυεπίπεδη ομαδοποίηση με χρήση ενός

collector που δημιουργήθηκε με μια έκδοση της μεθόδου Collectors.groupingBy που δέχεται δυο ορίσματα. Η μέθοδος, εκτός από τη συνάρτηση ομαδοποίησης παίρνει και ένα δεύτερο όρισμα τύπου collector. Έτσι για να εκτελεστεί μια ομαδοποίηση δύο επιπέδων, μια εσωτερική groupingBy μπορεί να περάσει ως παράμετρος σε μια εξωτερική groupingBy, ορίζοντας ένα δεύτερο κριτήριο για την ομαδοποίηση των στοιχείων του stream. Η ομαδοποίηση πολλών επιπέδων, μπορεί να επεκταθεί σε οποιοδήποτε αριθμό επιπέδων. Μια Ν-επίπεδη ομαδοποίηση έχει ως αποτέλεσμα έναν Ν-επίπεδο αντικείμενο τύπου Map που μοντελοποιεί μια Ν-επίπεδη δομή δέντρου. Η groupingBy μπορεί να θεωρηθεί ότι δουλεύει με «κάδους» (“buckets”). Η πρώτη groupingBy δημιουργεί έναν κάδο για κάθε κλειδί. Στη συνέχεια συλλέγονται τα στοιχεία σε κάθε κάδο σύμφωνα με τον κάθε collector και ούτω κάθε εξής. Έτσι επιτυγχάνεται μια Ν-επίπεδη ομαδοποίηση. Γενικότερα, ο δεύτερος collector που περνάει ως όρισμα στην πρώτη groupingBy δεν χρειάζεται να είναι κατ’ ανάγκη και αυτός groupingBy. Μπορεί να είναι οποιοσδήποτε τύπος collector. Επίσης η κλασσική με ένα όρισμα groupingBy (f), όπου f η συνάρτηση ομαδοποίησης, είναι στην πραγματικότητα απλώς μια συντομογραφία της groupingBy (f, toList ( )). Να σημειωθεί ότι ο collector groupingBy προσθέτει ένα νέο κλειδί στην ομαδοποίηση Map μόνο την πρώτη φορά που βρίσκει ένα στοιχείο στο stream. Το κλειδί παράγεται κατά την εφαρμογή σε αυτό των κριτηρίων ομαδοποίησης που χρησιμοποιούνται.

Παράδειγμα:

· Map<Person.Gender, Long> countByGender =
Person.persons()

.stream()

.collect(Collectors.groupingBy(Person::get
Gender, Collectors.counting()));

System.out.println(countByGender);

//Αποτέλεσμα

{MALE=4, FEMALE=2}

Παράδειγμα:

// NestedGroupings.java

· package com.jdojo.streams;
import java.time.Month;

import java.util.Map;

import java.util.stream.Collectors;

public class NestedGroupings {

public static void main(String[] args) {

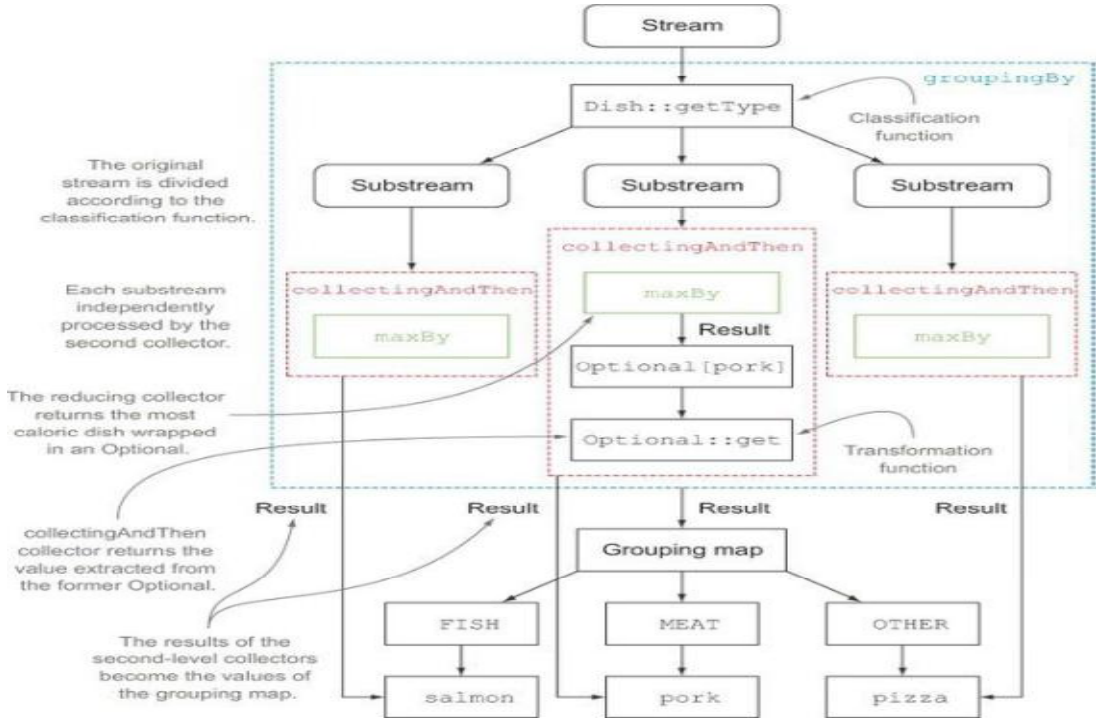
Map<Person.Gender, Map<Month, String>>
personsByGenderAndDobMonth = Person.persons()

```
                .stream()
                .collect(Collectors.groupingBy(Person::getGender,
                Collectors.groupingBy(p -> p.getDob().getMonth(),
                        Collectors.mapping(Person::getName,
                    Collectors.joining(", "))));
        System.out.println(personsByGenderAndDobMonth);
        }
}
//Αποτελέσματα
        {FEMALE={DECEMBER=Laynie, JULY=Donna},
    MALE={DECEMBER=Chris, JULY=Jeff, MAY=Ken, Li}}
```

Για την προσαρμογή του αποτελέσματος που επιστρέφεται από έναν collector σε διαφορετικό τύπο, μπορεί να χρησιμοποιηθεί η μέθοδος **Collectors.collectingAndThen**. Η μέθοδος παίρνει δύο ορίσματα, τον collector που πρέπει να προσαρμοστεί και μια συνάρτηση μετατροπής, και επιστρέφει ένα άλλο collector. Αυτός ο επιπλέον collector λειτουργεί ως περιτύλιγμα για τον παλιό και αντιστοιχίζει την τιμή που επιστρέφει χρησιμοποιώντας τη συνάρτηση μετατροπής ως τελευταίο βήμα της λειτουργίας collect. Γενικότερα, ο collector που περνάει ως δεύτερο όρισμα στη μέθοδο groupingBy χρησιμοποιείται για να εκτελέσει μια περαιτέρω λειτουργία μείωσης σε όλα τα στοιχεία του stream που έχουν ομαδοποιηθεί στην ίδια ομάδα.

· collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)

Είναι συνηθισμένο να χρησιμοποιούνται πολλαπλοί εμφωλευμένοι collectors.



Εικόνα 16: Συνδυασμός πολλαπλών εμφωλευμένων collectors[1].

Ένας άλλος collector, που συχνά συνδυάζεται με την groupingBy, παράγεται από την μέθοδο **mapping**. Αυτή η μέθοδος παίρνει δύο ορίσματα: μια συνάρτηση μετασχηματισμού των στοιχείων του stream και έναν περαιτέρω collector που συγκεντρώνει τα αντικείμενα που προκύπτουν από το μετασχηματισμό αυτό. Σκοπός της μεθόδου είναι, να προσαρμόσει ένα collector που αποδέχεται τα στοιχεία ενός συγκεκριμένου τύπου σε έναν άλλο collector που «εργάζεται» σε αντικείμενα διαφορετικού τύπου, εφαρμόζοντας μια συνάρτηση αντιστοίχησης σε κάθε στοιχείο εισόδου πριν τη συγκέντρωση τους,

· mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)

Παράδειγμα:

```
· Map<Person.Gender, String> namesByGender =
Person.persons()
        .stream()
        .collect(Collectors.groupingBy(Person::getGender,
                Collectors.mapping(Person::getName,
                Collectors.joining(", "))));
System.out.println(namesByGender);
//Αποτελέσματα
{MALE=Ken, Jeff, Chris, Li, FEMALE=Donna, Laynie}
```

Η διαμέριση (Partitioning) αποτελεί μια ειδική περίπτωση ομαδοποίησης. Η συνάρτηση ομαδοποίησης που δέχεται ως όρισμα η μέθοδος **partitioningBy**, είναι μια συνάρτηση που επιστρέφει μια τιμή τύπου boolean (predicate), που ονομάζεται συνάρτηση διαμέρισης (partitioning function). Το γεγονός ότι η συνάρτηση διαμέρισης επιστρέφει μια Boolean τιμή σημαίνει ότι προκύπτουσα ομαδοποίηση Map θα έχει μια Boolean τιμή ως βασικό κλειδί και ως εκ τούτου μπορεί να υπάρχουν το πολύ δύο διαφορετικές ομάδες. Μια για την τιμή true και μία για την τιμή false. Η διαμέριση έχει το πλεονέκτημα της διατήρησης των δύο λιστών των στοιχείων του stream, για τα οποία η εφαρμογή της συνάρτησης διαμέρισης επιστρέφει true ή false. Η μέθοδος partitioningBy έχει μια υπερφορτωμένη έκδοση η οποία παίρνει και δεύτερο collector ως όρισμα.

· partitioningBy(Predicate<? super T> predicate)
· partitioningBy(Predicate<? super T> predicate, Collector<? super T,A,D> downstream)

Παραδείγματα:

· Map<Boolean, List<Person>> partionedByMaleGender = Person.persons()

.stream()

.collect(Collectors.partitioningBy(Person::isMale));

System.out.println(partionedByMaleGender);

//Αποτέλεσμα

{false=[(3, Donna, FEMALE, 1962-07-29, 8700.00), (5, Laynie, FEMALE, 2012-12-13, 0.00)], true=[(1, Ken, MALE, 1970-05-04, 6000.00), (2, Jeff, MALE, 1970-07-15, 7100.00), (4, Chris, MALE, 1993-12-16, 1800.00), (6, Li, MALE, 2001-05-09, 2400.00)]}

· Map<Boolean,String> partionedByMaleGender = Person.persons()

.stream()

.collect(Collectors.partitioningBy(Person::isMale,

Collectors.mapping(Person::getName, Collectors.joining(", "))));

System.out.println(partionedByMaleGender);

//Αποτελέσματα

{false=Donna, Laynie, true=Ken, Jeff, Chris, Li}

Στον παρακάτω πίνακα συγκεντρώνονται όλες οι στατικές μέθοδοι που είναι διαθέσιμες από την κλάση Collectors μαζί με τον τύπο επιστροφής τους όταν εφαρμόζονται σε ένα stream<T> καθώς και ο λόγος που χρησιμοποιούνται.

Πίνακας 7: Οι στατικές μέθοδοι της κλάσης Collectors.

Μέθοδος	Επιστρεφόμενος τύπος	Χρήση
toList	List<T>	Συγκέντρωση όλων των στοιχείων του stream σε μια λίστα List.
toSet	Set<T>	Συγκέντρωση όλων των στοιχείων του stream σε ένα σύνολο Set, εξαλείφοντας τα διπλότυπα.
toCollection	Collection<T>	Συγκέντρωση όλων των στοιχείων του stream σε μια συλλογή collection που δημιουργείται από τον παρεχόμενο supplier.
counting	Long	Μέτρηση του αριθμού των στοιχείων του stream.
summingInt	Integer	Άθροιση των τιμών με την ιδιότητα Integer των στοιχείων του stream.
averagingInt	Double	Υπολογισμός μέσου όρου των τιμών με την ιδιότητα Integer των στοιχείων του stream.
summarizing Int	IntSummary-Statistics	Συλλογή στατιστικών στοιχείων των στοιχείων με την ιδιότητα Integer του stream, όπως μέγιστο, ελάχιστο, ολικό άθροισμα, μέσος όρος.
joining	String	Συνένωση των συμβολοσειρών που προκύπτουν από την κλήση της μεθόδου toString πάνω σε κάθε στοιχείο του stream.
maxBy	Optional<T>	Ένα Optional περιτύλιγμα του μέγιστου στοιχείου του stream, σύμφωνα με τη δεδομένη σύγκριση ή Optional.empty () αν το stream είναι άδειο.
minBy	Optional<T>	Ένα Optional περιτύλιγμα του ελάχιστου στοιχείου του stream, σύμφωνα με τη δεδομένη σύγκριση ή Optional.empty () αν το stream είναι άδειο.
reducing	Ο τύπος που παράγεται από τη λειτουργία μείωσης	Μείωση του stream σε μία μοναδική τιμή, αρχίζοντας από μια αρχική τιμή που χρησιμοποιείται ως accumulator και επαναληπτικά συνδυάζεται με κάθε στοιχείο του stream, χρησιμοποιώντας έναν δυαδικό τελεστή BinaryOperator.

collectingAndThen	Ο τύπος που επιστρέφεται από τη συνάρτηση μετασχηματισμού	Τύλιγμα (Wrap) ενός άλλου collector και εφαρμογή μιας συνάρτησης μετασχηματισμού πάνω στο αποτέλεσμα του λειτουργία το αποτέλεσμά της.
groupingBy	Map<K, List<T>>	Ομαδοποίηση των στοιχείων του stream με βάση την τιμή κάποιας από τις ιδιότητές τους και χρήση αυτών των τιμών ως κλειδιά στον χάρτη Map που προκύπτει.
partitioningBy	Map<Boolean, List<T>>	Διαμέριση των στοιχείων του stream με βάση το αποτέλεσμα της εφαρμογής ενός κατηγορήματος (predicate) σε καθένα από αυτά.

Η διεπαφή Collector αποτελείται από ένα σύνολο μεθόδων που παρέχουν οδηγίες για το πώς θα εφαρμοστούν οι ειδικές λειτουργίες μείωσης, δηλαδή, οι collectors. Έκτος από τους «έτοιμους» collectors που είναι ήδη διαθέσιμοι από την διεπαφή, δίνεται η δυνατότητα στον προγραμματιστή να δημιουργήσει ο ίδιος, τις δικές του προσαρμοσμένες μεθόδους μείωσης ανάλογα με τις ανάγκες του και να τις συμπεριλάβει στην διεπαφή Collector. Με λίγα λόγια μπορεί να φτιάξει την δίκη του προσαρμοσμένη διεπαφή Collector που θα ορίζει τους δικούς του προσαρμοσμένους collectors.

Ανακεφαλαιώνοντας, ένα stream είναι μια ακολουθία από στοιχεία δεδομένων που υποστηρίζουν διαδοχικές και παράλληλες συγκεντρωτικές πράξεις. Οι Συλλογές στην Java επικεντρώνονται στην αποθήκευση δεδομένων και την πρόσβαση στα δεδομένα, ενώ τα streams στους υπολογισμούς πάνω στα δεδομένα. Τα streams δεν έχουν μνήμη. Παίρνουν τα δεδομένα από μια πηγή δεδομένων, η οποία είναι συνήθως μια συλλογή. Ωστόσο, ένα stream μπορεί να πάρει τα δεδομένα του και από άλλες πηγές, όπως ένα αρχείο I / O, μια συνάρτηση, κλπ. Ένα stream μπορεί επίσης να βασίζεται σε μια πηγή δεδομένων που είναι ικανή να παράγει άπειρα στοιχεία δεδομένων.

Τα streams συνδέονται μέσω μεθόδων οι οποίες σχηματίζουν μια σωλήνωση. Τα streams υποστηρίζουν δύο τύπους μεθόδων: τις ενδιάμεσες και τις τερματικές. Μια ενδιάμεση μέθοδος πάνω σε ένα stream παράγει ένα άλλο stream που μπορεί με την σειρά του να χρησιμεύσει ως stream εισόδου για μια άλλη ενδιάμεση μέθοδο. Μια τερματική μέθοδος παράγει ένα αποτέλεσμα υπό τη μορφή μιας μοναδικής τιμής. Τα streams δεν μπορούν να επαναχρησιμοποιηθούν μετά την κλήση μιας τερματικής μεθόδου πάνω τους.

Ορισμένες λειτουργίες στα streams ονομάζονται λειτουργίες βραχυκυκλώματος. Μια λειτουργία βραχυκύκλωμα δεν χρειάζεται αναγκαστικά να επεξεργαστεί όλα τα δεδομένα του stream. Τα streams είναι εγγενώς «τεμπέλικα». Επεξεργάζονται δεδομένα ανάλογα με τη ζήτηση. Τα στοιχεία δεν αλλάζουν, όταν

καλούνται ενδιάμεσες λειτουργίες πάνω τους. Μόνο η κλήση μιας τερματικής λειτουργίας αλλάζει τα δεδομένα του stream. Μια σωλήνωση από streams μπορούν να εκτελεστούν σε σειρά ή παράλληλα. Από προεπιλογή, τα streams είναι σειριακά. Η μετατροπή ενός σειριακού stream σε παράλληλο επιτυγχάνεται με την κλήση της μεθόδου parallel( ). Αντίστροφα ένα παράλληλο stream μετατρέπεται σε σειριακό με την μέθοδο sequential( ).

Το Streams API υποστηρίζει τις περισσότερες από τις μεθόδους που υποστηρίζονται στον συναρτησιακό προγραμματισμό, όπως τις filter, map, forEach, reduce, allMatch, anyMatch, findAny, findFirst, κτλ. Το Streams API παρέχει επίσης συλλέκτες που χρησιμοποιούνται για τη συλλογή δεδομένων σε συλλογές, όπως ένα χάρτη, μια λίστα, ένα σύνολο, κ.λπ. Η κλάση Collectors είναι μια βοηθητική κλάση που παρέχει πολλές υλοποιήσεις συλλεκτών. Η αντιστοίχιση, η ομαδοποίηση και η διαμέριση των δεδομένων ενός stream πραγματοποιούνται εύκολα με χρήση της μεθόδου collect( ) χρησιμοποιώντας τον εκάστοτε συλλέκτη που παρέχεται.

Τα παράλληλα stream εκμεταλλεύονται τους πολυπύρηνους επεξεργαστές. Χρησιμοποιούν το Fork/Join πλαίσιο για την παράλληλη επεξεργασία των στοιχείων του stream.

ΚΕΦΑΛΑΙΟ 4

DEFAULT METHODS

4.1 Η ΕΞΕΛΙΞΗ ΤΩΝ API

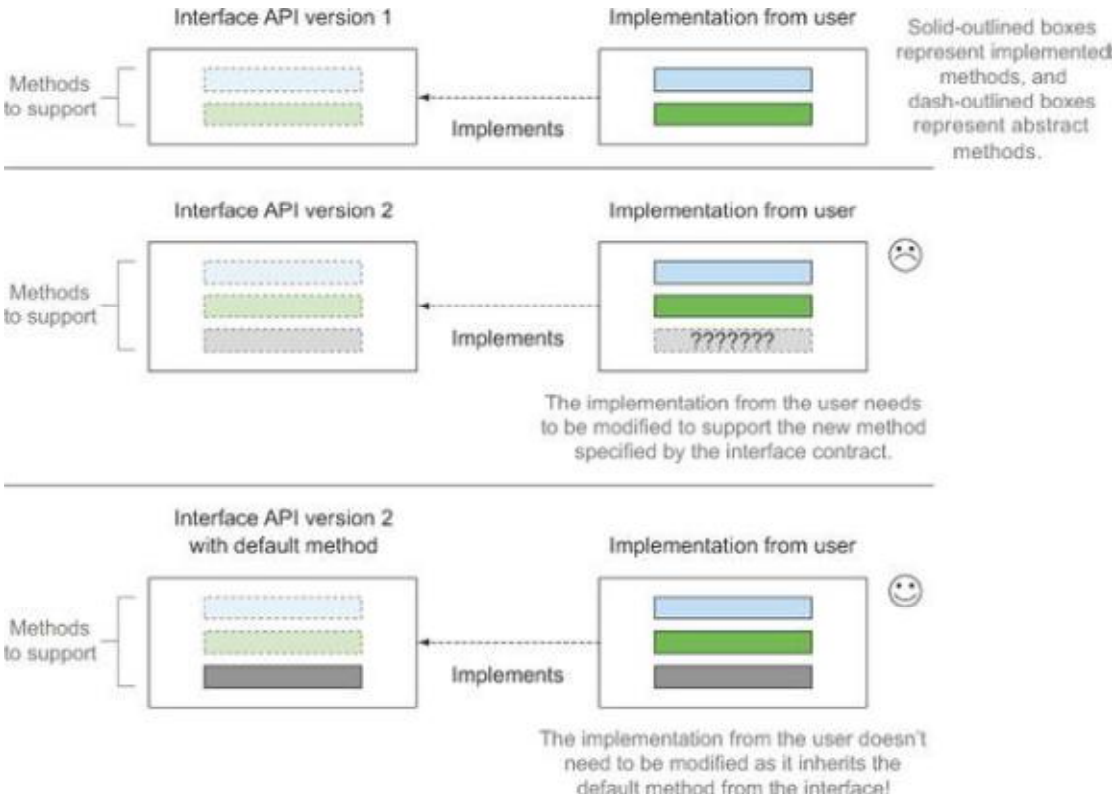
Παραδοσιακά, μια διεπαφή Java συγκεντρώνει τις συναφείς μεθόδους μαζί, σε μια σύμβαση (contract). Κάθε κλάση που υλοποιεί μια διεπαφή πρέπει να παρέχεται η ίδια, μία υλοποίηση για κάθε μέθοδο που ορίζεται από την διεπαφή ή να κληρονομεί την υλοποίηση από κάποια υπερκλάση. Η ανάγκη αυτή, προκαλεί πρόβλημα όταν οι σχεδιαστές μιας βιβλιοθήκης πρέπει να ενημερώσουν μια διεπαφή προσθέτοντας μια νέα μέθοδο γιατί, οι ήδη υπάρχουσες κλάσεις θα πρέπει να τροποποιηθούν ώστε να λάβουν υπόψη τους την νέα σύμβαση της διεπαφής. Το πρόβλημα γίνεται ακόμη μεγαλύτερο για το νέο API της Java 8 που εισάγει πολλές νέες μεθόδους στις υπάρχουσες διεπαφές.

Για να αντιμετωπίσει αυτό το ζήτημα η Java 8 εισάγει ένα νέο μηχανισμό. Οι διεπαφές στη Java 8 μπορούν πλέον να δηλώνουν μεθόδους μαζί με τον κώδικα υλοποίησης τους[2, 3]. Αυτό γίνεται με δύο τρόπους:

1. Με την ύπαρξη στατικών μεθόδων (static methods) μέσα στις διεπαφές
2. Με την εισαγωγή ενός νέου χαρακτηριστικού, των προεπιλεγμένων μεθόδων (default methods) που επιτρέπουν την παροχή προεπιλεγμένων υλοποιήσεων για τις μεθόδους μιας διεπαφής.

Οι διεπαφές δηλαδή, μπορούν να παρέχουν συγκεκριμένη υλοποίηση για τις μεθόδους, με αποτέλεσμα, οι υπάρχουσες κλάσεις που υλοποιούν μια διεπαφή να κληρονομούν αυτόματα τις προεπιλεγμένες υλοποίησης σε περίπτωση που δεν διαθέτουν οι ίδιες κάποια συγκεκριμένη υλοποίηση . Με αυτό τον τρόπο οι διεπαφές μπορούν να εξελίσσονται «ανώδυνα» (nonintrusively).

Οι βασικοί χρήστες των προεπιλεγμένων μεθόδων είναι οι σχεδιαστές των βιβλιοθηκών και αυτό γιατί οι προεπιλεγμένες μέθοδοι εισήχθησαν προκειμένου να εξελιχθούν οι βιβλιοθήκες, όπως η Java API με συμβατό τρόπο όπως φαίνεται στην παρακάτω εικόνα.



Εικόνα 17: Πρόσθεση νέας μεθόδου σε μια διεπαφή κάνοντας χρήση των προεπιλεγμένων μεθόδων[1].

Αυτό είναι το κίνητρο για τις προεπιλεγμένες μεθόδους. Επιτρέπουν στις κλάσεις να κληρονομούν αυτόματα μια προεπιλεγμένη υλοποίηση από μια διεπαφή. Έτσι εξελίσσονται οι διεπαφές χωρίς να χρειάζονται τροποποιήσεις στις υπάρχουσες υλοποιήσεις. Επίσης οι προεπιλεγμένες μέθοδοι βοηθούν στη διάρθρωση των προγραμμάτων, παρέχοντας έναν ευέλικτο μηχανισμό για πολλαπλή κληρονομικότητα συμπεριφοράς: μια κλάση μπορεί να κληρονομήσει προεπιλεγμένες μεθόδους από πολλές διεπαφές.

Προκειμένου να γίνει κατανοητή η δυσκολία της εξέλιξης ενός API από τι στιγμή που θα δημοσιευτεί θα δοθεί ένα παράδειγμα.

Παράδειγμα: Έστω μια βιβλιοθήκη της Java για σχεδίαση σχημάτων που περιέχει την διεπαφή Resizable που ορίζει διάφορες μεθόδους αλλαγής μεγέθους για σχήματα.

```
· public interface Resizable extends Drawable{
int getWidth();

int getHeight();

void setWidth(int width);

void setHeight(int height);

void setAbsoluteSize(int width, int height);
```

}

Έστω ότι ένας χρήστης δημιουργεί μια δική του υλοποίηση της Resizable που την ονομάζει Ellipse:

· public class Ellipse implements Resizable {
...

}

Και στη συνέχεια δημιουργεί ένα παιχνίδι (κλάση Game) που επεξεργάζεται διάφορα είδη σχημάτων Resizable συμπεριλαμβανομένης και της Ellipse.

```
public class Game{
    public static void main(String…args){
        List<Resizable> resizableShapes =
            Arrays.asList(new Square(), new Rectangle(), new Ellipse());
        Utils.paint(resizableShapes);
    }
}
public class Utils{
    public static void paint(List<Resizable> l){
        l.forEach(r -> {
                        r.setAbsoluteSize(42, 42);
                        r.draw();
                      });
    }
}
```

A list of shapes that are resizable

Calling the setAbsoluteSize method on each shape

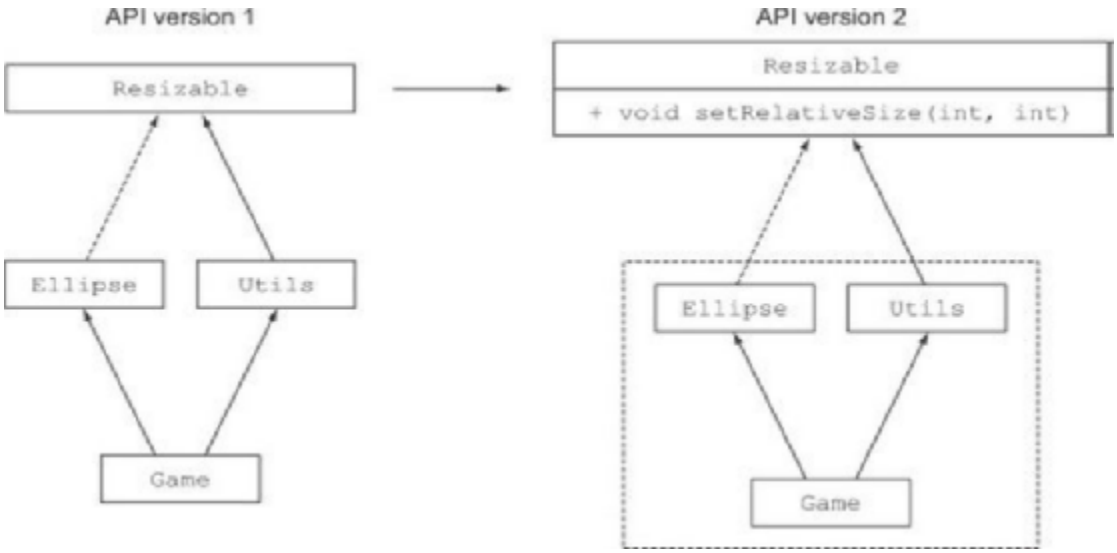
Εικόνα 18: Παράδειγμα υλοποίησης χρήστη[1].

Για κάποιο λόγο η υλοποίηση της Resizable ενημερώνεται και προστίθεται σε αυτή η μέθοδος setRelativeSize:

```
public interface Resizable {
    int getWidth();
    int getHeight();
    void setWidth(int width);
    void setHeight(int height);
    void setAbsoluteSize(int width, int height);
    void setRelativeSize(int wFactor, int hFactor);
}
```

Adding a new method for API version 2

Εικόνα 19: Εξέλιξη του API με την πρόσθεση μιας νέας μεθόδου, της setRelativeSize[1].



Εικόνα 20: Η Μεταγλώττιση της εφαρμογής παράγει λάθη, διότι εξαρτάται από την διεπαφή Resizable[1].

Τότε όμως η υλοποίηση της Ellipse του χρήστη δεν υλοποιεί την μέθοδο setRelativeSize. Αυτό σημαίνει ότι ήδη υπάρχουσες υλοποιήσεις της κλασης θα συνεχίσουν να λειτουργούν χωρίς την υλοποίηση της νέας μεθόδου, αν δεν γίνει προσπάθεια να μεταγλωττιστούν όλες οι μέθοδοι ξανά. Έτσι το παιχνίδι του χρήστη θα συνεχίσει να λειτουργεί μέχρι ότου περαστεί στην λίστα παραμέτρων της Resizable ένα αντικείμενο τύπου Ellipse. Τότε θα προκύψει λάθος κατά το χρόνο εκτέλεσης (run-time error) επειδή η μέθοδος setRelativeSize δεν έχει υλοποιηθεί:

· Exception in thread "main" java.lang.AbstractMethodError: lambdasinaction.chap9.Ellipse.setRelativeSize(II)V

Εάν ο χρήστης επιχειρήσει να ξαναφτιάξει ολόκληρη την εφαρμογή (συμπεριλαμβανομένης και της Ellipse), θα προκύψει λάθος κατά την μεταγλώττιση (compile error):

· lambdasinaction/chap9/Ellipse.java:6: error: Ellipse is not abstract and does not override abstract method setRelativeSize(int,int) in Resizable

Κατά συνέπεια, η ενημέρωση ενός δημοσιευμένου API δημιουργεί ασυμβατότητες προς τα πίσω (backward incompatibilities). Αυτός είναι ο λόγος που η εξέλιξη των API, προκαλεί προβλήματα για τους χρήστες των API. Υπάρχουν κάποιες εναλλακτικές λύσεις για την εξέλιξη ενός API, αλλά αποτελούν κακές επιλογές. Η καλύτερη λύση είναι οι προεπιλεγμένες μέθοδοι οι οποίες όπως ειπώθηκε και παραπάνω, δίνουν τη δυνατότητα στους σχεδιαστές των βιβλιοθηκών να εξελίσσουν τα API χωρίς το σπάσιμο του υπάρχοντα κώδικα, εφόσον οι κλάσεις υλοποιούν μια ενημερωμένη διεπαφή αυτόματα κληρονομώντας μια προεπιλεγμένη υλοποίηση.

Κατά την πρόσθεση μια μεθόδου σε μια διεπαφή ή πιο γενικά κατά την εισαγωγή μιας αλλαγής σε ένα πρόγραμμα Java, υπάρχουν τρία κύρια είδη της συμβατότητας:

1. Δυαδική συμβατότητα (Binary compatibility): σημαίνει ότι τα εκτελέσιμα (binaries) που ήδη υπάρχουν και που «έτρεχαν» χωρίς σφάλματα συνεχίζουν να είναι συνδεδεμένα (continue to link) χωρίς σφάλμα και μετά την εισαγωγή μιας αλλαγής.
2. Πηγαία συμβατότητα (source compatibility): σημαίνει ότι ένα υπάρχον πρόγραμμα θα εξακολουθήσει να μεταγλωττίζεται και μετά την εισαγωγή μιας αλλαγής.

Συμπεριφορική συμβατότητα (behavioral compatibility): σημαίνει ότι αν «τρέξει» ένα πρόγραμμα μετά την εισαγωγή μιας αλλαγής με τις ίδιες τιμές εισόδου, θα έχει ως αποτέλεσμα την ίδια συμπεριφορά.

4.2 ΕΙΣΑΓΩΓΗ ΣΤΙΣ DEFAULT METHODS

Ορισμός: Οι προεπιλεγμένες μέθοδοι είναι ένα νέο χαρακτηριστικό που προστέθηκε στη Java 8 για να βοηθήσει στην εξέλιξη των API με συμβατό τρόπο. Μια διεπαφή μπορεί πλέον να περιέχει υπογραφές μεθόδων για τις οποίες μια υλοποιημένη κλάση δεν παρέχει κάποια υλοποίηση. Τα σώματα των μεθόδων που λείπουν παρέχονται ως τμήματα της διεπαφής και όχι της κλάσης[2, 3].

Μια μέθοδος αναγνωρίζεται ως προεπιλεγμένη όταν ξεκινάει με τον τροποποιητή default (default modifier) και περιέχει ένα σώμα, ακριβώς όπως μια μέθοδος που δηλώνονται μέσα στην κλάση.

Παράδειγμα: Μια διεπαφή με το όνομα Sized με την αφηρημένη μέθοδο size και την προεπιλεγμένη μέθοδο isEmpty.

```
public interface Sized {
    int size();
    default boolean isEmpty() {
        return size() == 0;
    }
}
```

A default method

Εικόνα 21: Η μέθοδος isEmpty είναι προεπιλεγμένη επειδή πριν από τον τύπο επιστροφής της υπάρχει η λέξη default[1].

Η προσθήκη μιας μεθόδου σε μια διεπαφή με μία προεπιλεγμένη υλοποίηση δεν έχει πηγαία ασυμβατότητα. Όποια κλάση στο παραπάνω παράδειγμα υλοποιήσει την διεπαφή Sized αυτόματα θα κληρονομήσει και την υλοποίηση της isEmpty.

Δυο είναι οι βασικές διαφορές ανάμεσα σε μια αφηρημένη κλάση και σε μια διεπαφή παρόλο που και οι δυο μπορούν να περιέχουν αφηρημένες μεθόδους και μεθόδους με σώμα.

1. Μια κλάση μπορεί να επεκταθεί μόνο από μία 0αφηρημένη κλάση, αλλά μπορεί να υλοποιήσει πολλαπλές διεπαφές.

Μια αφηρημένη κλάση μπορεί να επιβάλει μια κοινή κατάσταση μέσω στιγμιότυπων μεταβλητών (instance variables) (πεδία). Μια διεπαφή δεν μπορεί να έχει στιγμιότυπα μεταβλητών.

4.3 ΧΡΗΣΙΜΟΠΟΙΩΝΤΑΣ ΠΡΟΤΥΠΑ ΓΙΑ ΤΙΣ DEFAULT METHODS

Εκτός από την εξέλιξη των βιβλιοθηκών με συμβατό τρόπο, οι προεπιλεγμένες μέθοδοι μπορούν να αξιοποιηθούν από τον ίδιο το χρήστη, προκειμένου να δημιουργήσει τις δικές του διεπαφές που θα περιέχουν προεπιλεγμένες μεθόδους. Αυτό μπορεί να χρειαστεί σε δυο περιπτώσεις:

1. Προαιρετικές μέθοδοι (optional methods)
2. Πολλαπλή κληρονομικότητα συμπεριφοράς (multiple inheritance of behavior)

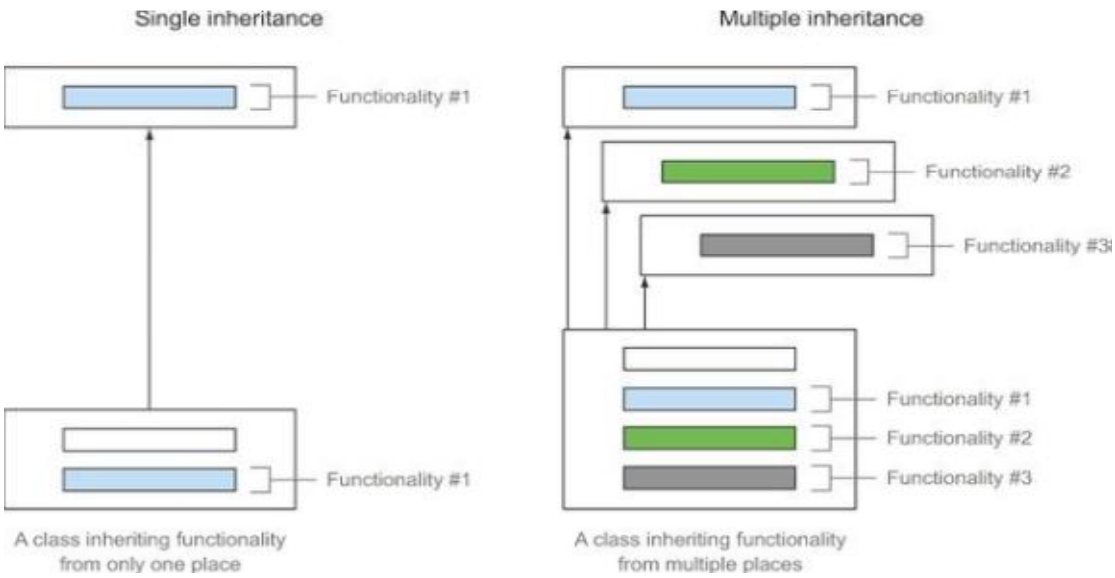
Μπορεί να υπάρχουν κλάσεις που να υλοποιούν μια διεπαφή αλλά αφήνουν κενές κάποιες υλοποιήσεις μεθόδων. Πριν την Java 8 αυτές οι μέθοδοι μπορεί να αγνοούνταν επειδή ίσως ο χρήστης αποφάσιζε να μην τις χρησιμοποιήσει όπως παραδείγματος χάριν για την μέθοδο remove της διεπαφής Iterator. Οι κλάσεις όμως που υλοποιούνταν, υλοποιούσαν ταυτόχρονα και τις κενές μεθόδους με αποτέλεσμα να υπάρχει περιττός στερεότυπος (boilerplate) κώδικας. Τη λύση δίνουν οι προεπιλεγμένες μέθοδοι οι οποίες παρέχουν μια προεπιλεγμένη υλοποίηση για τις κενές μεθόδους έτσι ώστε οι συγκεκριμένες κλάσεις (concrete classes), να μην χρειάζεται να προβλέπουν ρητά την υλοποίηση των κενών μεθόδων.

Παράδειγμα: Η διεπαφή Iterator στην Java 8 παρέχει μια προεπιλεγμένη υλοποίηση για την μέθοδο remove:

```
· interface Iterator<T> {
boolean hasNext();

T next();

default void remove() {

        throw new UnsupportedOperationException();

}

}
```

Είναι φανερό ότι έτσι μειώνεται ο στερεότυπος (boilerplate) κώδικα. Κάθε κλάση που υλοποιεί την διεπαφή Iterator δεν χρειάζεται να δηλώσει μια κενή remove μέθοδο και να την αγνοεί, εφόσον τώρα έχει μια προεπιλεγμένη υλοποίηση της.

Οι προεπιλεγμένες μέθοδοι επίσης, επιτρέπουν και κάτι που δεν γινόταν και με πολύ κομψό τρόπο πριν: την πολλαπλή κληρονομικότητα συμπεριφοράς. Την ικανότητα δηλαδή μιας κλασης, να επαναχρησιμοποιεί κώδικα από πολλαπλά μέρη.



Εικόνα 22: Διαφορά απλής και πολλαπλής κληρονομικότητας[1].

Προσοχή. Μια κλάση στη Java μπορεί να κληρονομήσει από μια και μόνο μια άλλη κλάση. Επειδή όμως οι μέθοδοι μιας διεπαφής μπορούν να έχουν υλοποιήσεις στη Java 8, οι κλάσεις μπορούν να κληρονομήσουν τη συμπεριφορά (κώδικα υλοποίησης) από πολλαπλές διεπαφές.

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable,
               Serializable, Iterable<E>, Collection<E> {
}
```

Inherits from one class

Implements six interfaces

Εικόνα 23: Δυνατότητα υλοποίησης (implements) πολλαπλών διεπαφών από μια κλάση[1]

Η κληρονομικότητα δεν θα πρέπει να είναι η μόνη επιλογή σε ότι έχει να κάνει με επαναχρησιμοποίηση κώδικα. Για παράδειγμα, το να κληρονομεί κάποια κλάση από μια κλάση που έχει 100 μεθόδους και πεδία μόνο για να επαναχρησιμοποιήσει μία μέθοδο είναι κακή ιδέα, διότι προσθέτει περιττή πολυπλοκότητα. Θα ήταν προτιμότερο να χρησιμοποιηθεί μια αντιπροσωπία (delegation): να δημιουργηθεί δηλαδή μια μέθοδος που θα καλεί άμεσα τη μέθοδο της κλασης που χρειάζεστε μέσω μιας μεταβλητής μέλους. Η ίδια ιδέα ισχύει και για τις διεπαφές με τις προεπιλεγμένες μεθόδους. Διατηρώντας την διεπαφή ελάχιστη, επιτυγχάνεται μεγαλύτερη σύνθεση, με την επιλογή εκείνων μόνο των υλοποιήσεων που είναι απαραίτητες.

4.4 ΚΑΝΟΝΕΣ ΕΠΙΛΥΣΗΣ ΣΥΓΚΡΟΥΣΕΩΝ

Στη Java μια κλάση μπορεί να επεκταθεί μόνο από μια γονική κλάση αλλά μπορεί να υλοποιήσει πολλές διεπαφές. Χάρη στις προεπιλεγμένες μεθόδους, στη Java 8[5], υπάρχει η δυνατότητα κλάση να κληρονομήσει περισσότερες από μία μεθόδους με την ίδια υπογραφή. Το ποια έκδοση της μεθόδου θα χρησιμοποιηθεί όταν εμφανιστούν συγκρούσεις – εμφανίζονται αρκετά σπάνια στην πράξη- το ρυθμίζουν οι κανόνες επίλυσης που καθορίζουν τον τρόπο με τον οποίο θα επιλυθεί η σύγκρουση.

Παράδειγμα: Έστω τρεις διεπαφές: η A που εκτυπώνει ένα μήνυμα, η B που επεκτείνει την A και τυπώνει ένα άλλο μήνυμα και η C που τις υλοποιεί (implements) και τις δυο.

```
public interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}
public interface B extends A {
    default void hello() {
        System.out.println("Hello from B");
    }
}

public class C implements B, A {
    public static void main(String... args) {
        new C().hello();                          <- What gets printed?
    }
}
```

Εικόνα 24: Παραδείγματα διεπαφών που επεκτείνουν (extends) και υλοποιούν (implements) άλλες διεπαφές[1].

Για το ποια μέθοδος θα κληθεί στην διεπαφή C η Java 8 παρέχει κάποιους κανόνες για να επιλύσει αυτό το ζήτημα.

Υπάρχουν τρεις κανόνες επίλυσης συγκρούσεων που ακολουθούνται όταν μια κλάση κληρονομεί μια μέθοδο με την ίδια υπογραφή από πολλαπλές θέσεις (όπως μια άλλη κλάση ή διεπαφή):

1. Οι κλάσεις πάντα κερδίζουν. Η μέθοδος που είναι δηλωμένη στην κλάση ή την υπερκλάση έχει προτεραιότητα έναντι κάθε άλλης δηλωμένης προεπιλεγμένης μεθόδου.

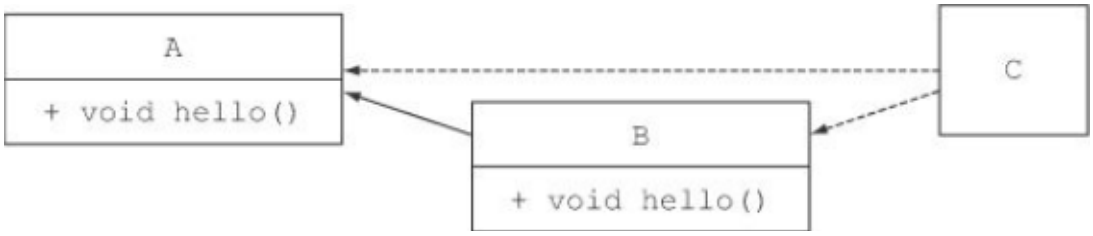
2. Σε αντίθετη περίπτωση, οι υπο-διεπαφές κερδίζουν. Από τις μεθόδους που έχουν την ίδια υπογραφή επιλέγεται εκείνη που παρέχεται από την πιο συγκεκριμένη διεπαφή. (Αν η διεπαφή B επεκτείνει την A, τότε η B θεωρείται πιο συγκεκριμένη από A).

3. Εάν υπάρχει ακόμα ασάφεια σχετικά με το ποια μέθοδος πρέπει να κληθεί, η κλάση που κληρονομεί από πολλαπλές διεπαφές πρέπει

ρητά να επιλεγεί ποια προεπιλεγμένη μέθοδος υλοποίησης θα χρησιμοποιηθεί, παρακάμπτοντας την (overriding) και καλώντας την επιθυμητή μέθοδο ρητά.

Παρακάτω παρουσιάζονται κάποιες πιθανές περιπτώσεις συγκρούσεων και ο τρόπος επίλυσης τους, για να γίνει πλήρως κατανοητός ο τρόπος με τον οποίο λειτουργούν οι κανόνες επίλυσης.

Παράδειγμα: Σύμφωνα με τον κανόνα 2 (η πιο συγκεκριμένη διεπαφή κερδίζει) θα τυπωθεί το μήνυμα της διεπαφής B, “Hello from B”.



Εικόνα 25: Η πιο συγκεκριμένη διεπαφή κερδίζει.

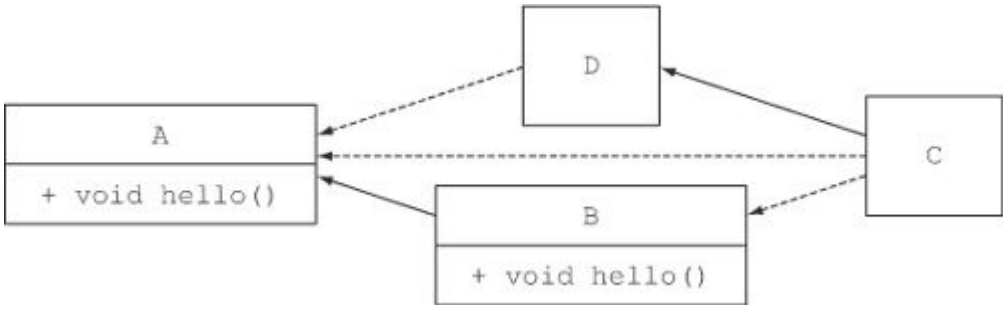
Παράδειγμα: Έστω ότι υπάρχει και μια κλάση D που εφαρμόζει την διεπαφή A και η κλάση C επεκτείνει την D.

```
public class D implements A{ }

public class C extends D implements B, A {
    public static void main(String... args) {
        new C().hello();
    }
}
```

What gets printed?

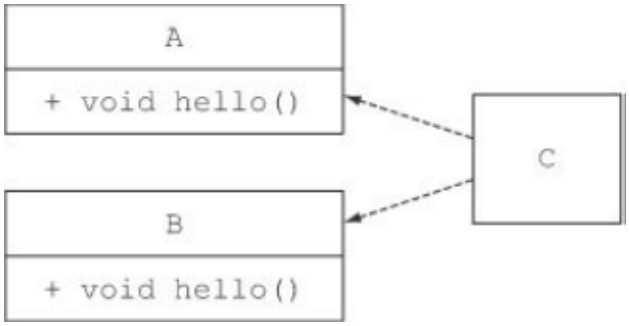
Εικόνα 26: Η κλάση D κληρονομεί από μια κλάση (C) και υλοποιεί δύο διεπαφές (B και A).

Ο κανόνας 1 λέει ότι μια δηλωμένη στην κλάση μέθοδο έχει πάντα προτεραιότητα. Αλλά η κλάση D δεν παρακάμπτει (override) την μέθοδο hello, αλλά εφαρμόζει (implements) την διεπαφή A. Κατά συνέπεια, έχει μια προεπιλεγμένη μέθοδο από τη διεπαφή A. Ο κανόνας 2 λέει ότι αν δεν υπάρχουν μέθοδοι στην κλάση ή την υπερκλάση, τότε επιλέγεται η μέθοδος με την πιο ειδική διεπαφή. Άρα, ο μεταγλωττιστής έχει να επιλέξει μεταξύ της μεθόδου Hello από τη διεπαφή A και της μεθόδου Hello από τη διεπαφή B. Επειδή η B είναι πιο συγκεκριμένη, το πρόγραμμα τελικά θα εκτυπώσει και πάλι “Hello from B”.



Εικόνα 27: Επίλυση σύγκρουσης. Η διεπαφή B είναι πιο συγκεκριμένη και κερδίζει.

Παράδειγμα: Έστω τώρα ότι η διεπαφή B δεν επεκτείνει πλέον την A.



Εικόνα 28: Υλοποίηση δυο ξεχωριστών μεθόδων A και B.

Η διεπαφή A και η κλάση C μένουν ως ήταν, αλλά ο κώδικας για την B αλλάζει και γίνεται:

· public interface A {
void hello() {

System.out.println("Hello from A");

}

}

· public interface B {
void hello() {

System.out.println("Hello from B");

}

}

· public class C implements B, A { }

Ο κανόνας 2 τώρα, δεν θα βοηθήσει, γιατί δεν υπάρχει πλέον πιο συγκεκριμένη διεπαφή για να επιλεγεί. Και οι δύο μέθοδοι Hello από τις διεπαφές A και B μπορούν να είναι έγκυρες επιλογές. Αυτό έχει ως αποτέλεσμα, ο μεταγλωττιστής της Java να παράγει ένα σφάλμα μεταγλώττισης, επειδή δεν γνωρίζει ποια μέθοδος είναι η πιο κατάλληλη: “Error: class C inherits unrelated defaults for hello() from types B and A.”

Δεν υπάρχουν πολλές λύσεις για την επίλυση της σύγκρουσης μεταξύ δύο έγκυρων επιλογών. θα πρέπει να αποφασιστεί ρητά ποια μέθοδος θα χρησιμοποιηθεί στην κλάση C. Για να επιτευχθεί αυτό, παρακάμπτεται (override) η μέθοδος Hello στην κλάση C και στη συνέχεια μέσα στο σώμα της, καλείται ρητά η μέθοδος που θα χρησιμοποιηθεί. Η Java 8 εισάγει τη νέα σύνταξη X.super.m (...) όπου το X είναι μια υπερδιεπαφή (superinterface) της οποίας η μέθοδος m καλείται.

Παράδειγμα: Επιλογή της μεθόδου Hello από την διεπαφή B, για να κληθεί στην κλάση C, με την βοήθεια της X.super.m (...).

```
public class C implements B, A {
    void hello(){
        B.super.hello();          <- Explicitly choosing to call the method from interface B
    }
}
```

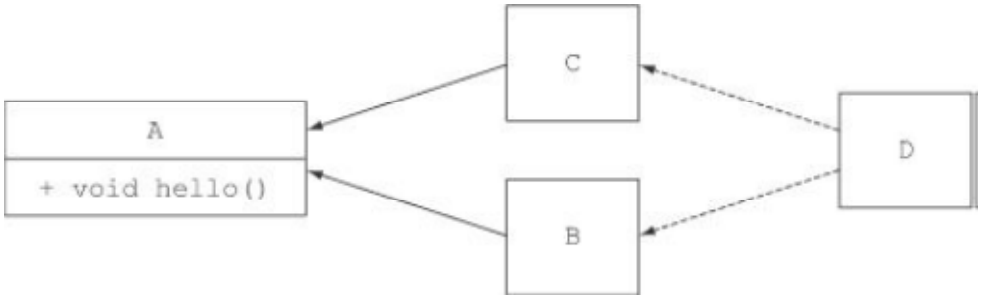
Εικόνα 29: Χρήση της μεθόδου X.super.m (...).

Παράδειγμα: Μια τελευταία περίπτωση είναι το λεγόμενο πρόβλημα διαμάντι (diamond problem)

```
public interface A{
    default void hello(){
        System.out.println("Hello from A");
    }
}

public interface B extends A { }

public interface C extends A { }

public class D implements B, C {
    public static void main(String... args) {     <- What gets printed?
        new D().hello();
    }
}
```

Εικόνα 30: Περίπτωση σύγκρουσης. Η κλάση D υλοποιεί τις διεπαφές B και C, που και οι δυο επεκτείνουν την διεπαφή A[1].

Ονομάζεται πρόβλημα διαμάντι επειδή το σχήμα του διαγράμματος μοιάζει με διαμάντι.



Εικόνα 31: Το πρόβλημα διαμάντι σχηματικά.

Η κλάση D κληρονομεί τους ορισμούς των προεπιλεγμένων μεθόδων των διεπαφών B και C. Στην πραγματικότητα όμως υπάρχει μόνο ένας ορισμός μεθόδου για να επιλεγεί . Μόνο η διεπαφή A ορίζει μια προεπιλεγμένη μέθοδο. Επειδή η διεπαφή A είναι μια υπερδιεπαφή της κλασης D, ο κώδικας τελικά θα εκτυπώσει “Hello from A”.

Ο μηχανισμός επίλυσης συγκρούσεων προεπιλεγμένων μεθόδων λοιπόν, είναι αρκετά απλός όταν μια κλάση κληρονομεί από διάφορες μεθόδους με την ίδια υπογραφή. Αρκει να ακολουθηθούν συστηματικά οι τρεις κανόνες που παρουσιάστηκαν και περιγράφηκαν παραπάνω και όλες οι πιθανές συγκρούσεις θα επιλυθούν.

ΚΕΦΑΛΑΙΟ 5

NEW DATE AND TIME API

5.1 ΕΙΣΑΓΩΓΗ

Γενικά το API της Java[2] περιλαμβάνει πολλά χρήσιμα στοιχεία που μπορούν να βοηθήσουν στην κατασκευή σύνθετων εφαρμογών. Αυτό όμως δεν ίσχυε πάντα. Πιο συγκεκριμένα, η υποστήριξη με μεθόδους που σχετίζονταν με την ημερομηνία και την ώρα ήταν αρκετά προβληματική πριν την Java 8.

Στην Java 1.0 η μόνη υποστήριξη για την ημερομηνία και την ώρα ήταν η κλάση java.util.Date. Παρά το όνομά της, η κλάση δεν αντιπροσώπευε μια ημερομηνία, αλλά ένα σημείο στο χρόνο με ακρίβεια χιλιοστών του δευτερολέπτου. Ακόμη χειρότερα, η χρηστικότητα της κλασης υπονομεύονταν από κάποιες αόριστες επιλογές σχεδίασης.

Παράδειγμα: Στην Java 1.0, τα χρόνια ξεκινούν από το 1900, ενώ οι μήνες ξεκινούν με δείκτη το 0. Για την αναπαράσταση μιας ημερομηνίας –π.χ. March 18, 2014- έπρεπε να δημιουργηθεί ένα στιγμιότυπο της κλασης Date της μορφής :

· Date date = new Date(114, 2, 18);

Η εκτύπωση της συγκεκριμένης ημερομηνίας έχει την όχι και τόσο εύχρηστη μορφή:

· Tue Mar 18 00:00:00 CET 2014

Τα προβλήματα και οι περιορισμοί της κλάσης Date έγιναν αμέσως εμφανή όταν βγήκε η Java 1.0, αλλά ήταν επίσης σαφές ότι το πρόβλημα δεν μπορούσε να επιδιορθωθεί χωρίς να σπάσει την προς τα πίσω συμβατότητα. Ως αποτέλεσμα, στην Java 1.1 πολλές από τις μεθόδους της κλάσης Date αποσύρθηκαν, και η κλάση αντικαταστάθηκε με την εναλλακτική κλάση, java.util.Calendar. Δυστυχώς όμως, και η κλάση Calendar είχε παρόμοια προβλήματα και ελαττώματα σχεδιασμού που οδηγούσαν στη δημιουργία κώδικα επιρρεπή σε λάθη. Ακόμα χειρότερα, η παρουσία και των δύο κλάσεων (Date και Calendar) προκαλούσε σύγχυση ανάμεσα στους προγραμματιστές για το ποια έπρεπε να χρησιμοποιηθεί. Τέλος και οι δυο κλάσεις είναι ευμετάβλητες, πράγμα που σημαίνει ότι μικρές τροποποιήσεις, οδηγούν σε μεγάλα προβλήματα συντήρησης. Όλες αυτές οι ατέλειες και οι αντιφάσεις ενθάρρυναν τη χρήση βιβλιοθηκών ημερομηνίας και ώρας που έχουν σχεδιαστεί από τρίτους, όπως η βιβλιοθήκη Joda-Time.

Προκειμένου λοιπόν να δοθεί λύση σε αυτό το ζήτημα, η Oracle αποφάσισε να προσφέρει υψηλής ποιότητας υποστήριξη για την ημερομηνία και την ώρα μέσω του

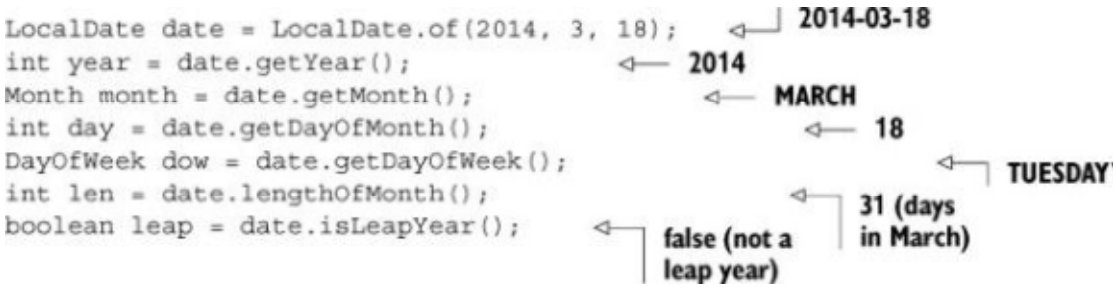
Java API. Ως αποτέλεσμα, Java 8 εισάγει ένα εντελώς νέο API για την ημερομηνία και την ώρα που ενσωματώνει πολλά από τα χαρακτηριστικά της Joda-Time στο πακέτο java.time.

Το πακέτο java.time περιλαμβάνει πολλές καινούργιες κλάσεις για την δημιουργία απλών ημερομηνιών, ωρών και χρονικών διαστημάτων. Από τις σημαντικότερες είναι οι: LocalDate, LocalTime, LocalDateTime, Instant, Duration και Period.

Η πιο συχνά χρησιμοποιούμενη κλάση του νέου Date και Time API της Java είναι η LocalDate. Ένα στιγμιότυπο αυτής της κλασης είναι ένα αμετάβλητο αντικείμενο που αναπαριστά μια απλή ημερομηνία χωρίς την ώρα της ημέρας. Συγκεκριμένα, δεν περιέχει καμία πληροφορία σχετικά με τη ζώνη ώρας.

Η δημιουργία ενός στιγμιότυπου LocalDate επιτυγχάνεται με την χρήση της στατικής μεθόδου of. Ένα στιγμιότυπο LocalDate παρέχει πολλές μεθόδους που διαβάζουν τις πιο συχνά χρησιμοποιούμενες τιμές της κλασης, όπως το έτος, ο μήνας, η ημέρα της εβδομάδας κ.τ.λ.

Παράδειγμα: Δημιουργία ενός στιγμιότυπου LocalDate και κάποιες μέθοδοι για το διάβασμα των τιμών της.



Εικόνα 32: Κάποιες από τις μεθόδους που διαθέτει η κλάση LocalDate για το διάβασμα των τιμών της[1].

Η τρέχουσα ημερομηνία μπορεί επίσης να ληφθεί από το ρολόι του συστήματος με τη χρήση της μεθόδου now:

· LocalDate today = LocalDate.now();

Η πρόσβαση στις παραπάνω πληροφορίες, μπορεί να επιτευχθεί και με το πέρασμα της διεπαφής TemporalField στη μέθοδο get. Η TemporalField είναι μια διεπαφή που καθορίζει τον τρόπο πρόσβασης στην τιμή ενός συγκεκριμένου πεδίου ενός χρονικού αντικειμένου (temporal object). Η απαρίθμηση ChronoField υλοποιεί αυτή την διεπαφή, έτσι ώστε κάθε στοιχείο της απαρίθμησης να μπορεί να χρησιμοποιηθεί εύκολα με την μέθοδο get.

Παράδειγμα: Χρήση της διεπαφής TemporalField για την ανάγνωση κάποιων τιμών της κλασης LocalDate:

· int year = date.get(ChronoField.YEAR);
· int month = date.get(ChronoField.MONTH_OF_YEAR);
· int day = date.get(ChronoField.DAY_OF_MONTH);

Η κλάση LocalTime αναπαριστά την ώρα της ημέρας.

Ένα στιγμιότυπο της LocalTime μπορεί να δημιουργηθεί χρησιμοποιώντας δύο υπερφορτωμένες στατικές μεθόδους που ονομάζονται of. Η πρώτη δέχεται ως ορίσματα ώρες και λεπτά ενώ η δεύτερη δέχεται επιπλέον και τα δευτερόλεπτα. Όπως και η κλάση LocalDate, έτσι και η LocalTime παρέχει αντίστοιχες μεθόδους get που έχουν πρόσβαση στις τιμές της.

Παράδειγμα: Δημιουργία ενός στιγμιότυπου LocalTime και κάποιες μέθοδοι για το διάβασμα των τιμών της.

```
LocalTime time = LocalTime.of(13, 45, 20);   <-- 13:45:20
int hour = time.getHour();                   <-- 13
int minute = time.getMinute();               <-- 45
int second = time.getSecond();               <-- 20
```

Εικόνα 33: Κάποιες από τις μεθόδους που διαθέτει η κλάση LocalTime για το διάβασμα των τιμών της.

Τόσο η LocalDate όσο και η LocalTime μπορούν να δημιουργηθούν από την ανάλυση ενός String που τις αναπαριστά. Οι στατικές μέθοδοι parse το καταφέρνουν αυτό:

· LocalDate date = LocalDate.parse("2014-03-18");
· LocalTime time = LocalTime.parse("13:45:20");

Σε μια μέθοδο parse μπορεί να περάσει ως παράμετρος ένας DateTimeFormatter, ο οποίος διευκρινίζει το πώς θα διαμορφωθεί ένα αντικείμενο τύπου ημερομηνία (date) ή / και τύπου ώρα (time).

Η σύνθετη κλάση LocalDateTime συνδυάζει σε ζεύγη μια LocalDate και μια LocalTime. Αναπαριστά ταυτόχρονα μια ημερομηνία και μια ώρα, χωρίς μια ζώνη ώρας, και ένα στιγμιότυπο της, μπορεί να δημιουργηθεί είτε απευθείας είτε συνδυάζοντας μια ημερομηνία και μια ώρα.

Παράδειγμα: Δημιουργία στιγμιότυπου LocalDateTime απευθείας και με συνδυασμό ημερομηνίας και ώρας.

· // 2014-03-18T13:45:20
· LocalDateTime dt1 = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45, 20);
· LocalDateTime dt2 = LocalDateTime.of(date, time);
· LocalDateTime dt3 = date.atTime(13, 45, 20);
· LocalDateTime dt4 = date.atTime(time);
· LocalDateTime dt5 = time.atDate(date);

Να σημειωθεί ότι μια LocalDateTime μπορεί να δημιουργηθεί περνώντας μια ώρα (time) σε μια LocalDate, ή αντιστρόφως, μια ημερομηνία (date) σε μια

LocalTime, χρησιμοποιώντας αντίστοιχα τις μεθόδους atTime ή atDate. Η LocalDate ή η LocalTime μπορούν να εξαχθούν από την LocalDateTime χρησιμοποιώντας τις μεθόδους toLocalDate και toLocalTime.

Παράδειγμα: Εξαγωγή LocalDate και LocalTime με τις μεθόδους toLocalDate και toLocalTime.

```
LocalDate date1 = dt1.toLocalDate();
LocalTime time1 = dt1.toLocalTime();
```

2014-03-18

13:45:20

Εικόνα 34: Χρήση των μεθόδων toLocalDate και toLocalTime.

Η αναπαράσταση της ημερομηνίας και της ώρας σε μορφή κατανοητή από τον άνθρωπο δεν είναι εύχρηστη για έναν υπολογιστή.

Από την οπτική γωνία της μηχανής, η πιο φυσική μορφή για τη μοντελοποίηση του χρόνου, είναι με ένα μεγάλο αριθμό που αντιπροσωπεύει ένα σημείο σε μια συνεχή λωρίδα χρόνου (timeline). Αυτή η προσέγγιση χρησιμοποιείται από την νέα κλάση java.time.Instant, η οποία ουσιαστικά αναπαριστά τον αριθμό των δευτερολέπτων που έχουν περάσει από τα μεσάνυχτα της πρώτης Ιανουαρίου του 1970 UTC, την κατά σύμβαση ημερομηνία εμφάνισης των Unix.

Ένα στιγμιότυπο αυτής της κλασης μπορεί να δημιουργηθεί με το πέρασμα του αριθμού των δευτερολέπτων στην στατική μέθοδο που διαθέτει, την ofEpochSecond. Επιπλέον, η κλάση Instant υποστηρίζει ακρίβεια της τάξης των νανοδευτερολέπτων. Υπάρχει μια συμπληρωματική υπερφορτωμένη έκδοση της στατικής μεθόδου ofEpochSecond που δέχεται μια δεύτερη παράμετρο, που είναι μια προσαρμογή σε νανοδευτερόλεπτα του αριθμού των δευτερολέπτων που πέρασαν. Αυτή η υπερφορτωμένη έκδοση ρυθμίζει το την παράμετρο των νανοδευτερολέπτων, διασφαλίζοντας ότι το αποθηκευμένο κλάσμα νανοδευτερολέπτων είναι μεταξύ 0 και 999.999.999.

Παράδειγμα: Όλες οι κλήσεις της ofEpochSecond θα επιστρέψουν την ίδια τιμή.

```
Instant.ofEpochSecond(3);
Instant.ofEpochSecond(3, 0);
Instant.ofEpochSecond(2, 1_000_000_000);
Instant.ofEpochSecond(4, -1_000_000_000);
```

One billion nanoseconds (1 second) after 2 seconds

One billion nanoseconds (1 second) before 4 seconds

Εικόνα 35: Κλήσεις της μεθόδου ofEpochSecond με διαφορετικές παραμέτρους που δίνουν το ίδιο αποτέλεσμα.

Η κλάση Instant υποστηρίζει επίσης μια άλλη στατική μέθοδο που ονομάζεται now. Η μέθοδος now επιτρέπει την σύλληψη μιας χρονικής σήμανσης (timestamp) της τρέχουσας στιγμής. Να επισημανθεί ότι η κλάση Instant προορίζεται μόνο για χρήση από μια μηχανή. Αποτελείται από έναν αριθμό δευτερολέπτων και

νανοδευτερολέπτων και κατά συνέπεια, δεν παρέχει καμία δυνατότητα χειρισμού μονάδων χρόνου που έχουν νόημα για τον άνθρωπο.

Παράδειγμα: Η δήλωση

· int day = Instant.now().get(ChronoField.DAY_OF_MONTH);

θα πετάξει την εξαίρεση

· java.time.temporal.UnsupportedTemporalTypeException: Unsupported field: DayOfMonth

Όλες οι παραπάνω κλάσεις υλοποιούν την διεπαφή Temporal, η οποία καθορίζει τον τρόπο διαβάσματος και διαχείρισης των τιμών ενός αντικειμένου που μοντελοποιεί ένα γενικό σημείο στο χρόνο (generic point in time).

Υπάρχει όμως και η ανάγκη για την δημιουργία μιας χρονικής διάρκειας (duration) μεταξύ δύο χρονικών αντικειμένων. Η στατική μέθοδος between της κλασης Duration εξυπηρετεί ακριβώς αυτόν τον σκοπό. Μπορούν να δημιουργηθούν χρονικές διάρκειες μεταξύ δύο LocalTimes, δύο LocalDateTimes, ή δύο Instants:

· Duration d1 = Duration.between(time1, time2);
· Duration d1 = Duration.between(dateTime1, dateTime2);
· Duration d2 = Duration.between(instant1, instant2);

Η LocalDateTime και η Instant, χρησιμοποιούνται αντίστοιχα από τον άνθρωπο και τον υπολογιστή. Για το λόγο αυτό δεν επιτρέπεται να αναμειχτούν. Η προσπάθεια δημιουργίας μιας χρονικής διάρκειας μεταξύ τους θα οδηγούσε σε μια εξαίρεση της μορφής: DateTimeException

Στην περίπτωση που το χρονικό διάστημα χρειάζεται να είναι μεταξύ ετών, μηνών και ημερών υπάρχει η κλάση Period. Η διαφορά μεταξύ δύο LocalDates μπορεί να βρεθεί κάνοντας χρήση της μεθόδου between της κλασης Period.

· Period tenDays = Period.between(LocalDate.of(2014, 3, 8),
LocalDate.of(2014, 3, 18));

Η κλάση Duration και η κλάση Period έχουν και οι δυο παρόμοιες βολικές μεθόδους για την δημιουργία στιγμιότυπων άμεσα. Δηλαδή, χωρίς να χρειάζεται ο προσδιορισμός της διαφοράς μεταξύ δύο χρονικών αντικειμένων.

· Duration threeMinutes = Duration.ofMinutes(3);
· Duration threeMinutes = Duration.of(3, ChronoUnit.MINUTES);
· Period tenDays = Period.ofDays(10);
· Period threeWeeks = Period.ofWeeks(3);
· Period twoYearsSixMonthsOneDay = Period.of(2, 6, 1);

Οι πιο συνηθισμένες μέθοδοι των κλάσεων ημερομηνίας και ώρας που αναπαριστούν ένα χρονικό διάστημα (interval) συγκεντρώνονται στον παρακάτω πίνακα.

Πίνακας 8: Μέθοδοι για την αναπαράσταση χρονικών διαστημάτων

Μέθοδος	Στατική	Περιγραφή
between	Ναι	Δημιουργεί ένα χρονικό διάστημα ανάμεσα σε δύο χρονικά σημεία
from	Ναι	Δημιουργεί ένα χρονικό διάστημα από μία χρονική μονάδα
of	Ναι	Δημιουργεί ένα στιγμιότυπο του χρονικού διαστήματος από τα συστατικά του μέρη
parse	Ναι	Δημιουργεί ένα στιγμιότυπο του χρονικού διαστήματος από ένα String
addTo	Όχι	Δημιουργεί ένα αντίγραφο του χρονικού διαστήματος προσθέτοντας σε αυτό, το καθορισμένο χρονικό αντικείμενο
get	Όχι	Διαβάζει μέρος της δήλωσης (part of the state) του χρονικού διαστήματος
isNegative	Όχι	Ελέγχει αν το χρονικό διάστημα είναι αρνητικό, εκτός από μηδέν
isZero	Όχι	Ελέγχει αν το χρονικό διάστημα έχει μηδενικό μήκος
minus	Όχι	Δημιουργεί ένα αντίγραφο του χρονικού διαστήματος αφαιρώντας ένα χρονικό διάστημα
multipliedBy	Όχι	Δημιουργεί ένα αντίγραφο του χρονικού διαστήματος πολλαπλασιαζόμενο με το δεδομένο βαθμό (scalar)
negated	Όχι	Δημιουργεί ένα αντίγραφο του χρονικού διαστήματος αναιρώντας (negated) το μήκος
plus	Όχι	Δημιουργεί ένα αντίγραφο του χρονικού διαστήματος προσθέτοντας ένα χρονικό διάστημα
subtractFrom	Όχι	Αφαιρεί το χρονικό διάστημα από ένα καθορισμένο χρονικό αντικείμενο

Όλες οι κλάσεις που περιγράφηκαν μέχρι στιγμής είναι αμετάβλητες (immutable). Η σχεδιαστική αυτή επιλογή επιτρέπει ένα πιο συναρτησιακό στυλ προγραμματισμού (functional programming style), εξασφαλίζει νηματική-ασφάλεια (thread-safety), και διατηρεί τη συνοχή του μοντέλου. Παρ 'όλα αυτά, το νέο Date και Time API προσφέρει μερικές εύχρηστες μεθόδους για την δημιουργία τροποποιημένων εκδόσεων των εν λόγω αντικειμένων.

5.2 ΔΙΑΧΕΙΡΙΣΗ ΑΝΑΛΥΣΗ ΚΑΙ ΜΟΡΦΟΠΟΙΗΣΗ ΗΜΕΡΟΜΗΝΙΩΝ

Η πιο άμεσος και εύκολος τρόπος για να δημιουργηθεί μια τροποποιημένη έκδοση μιας ήδη υπάρχουσας LocalDate είναι να αλλάξει κάποιο από τα χαρακτηριστικά της, χρησιμοποιώντας μία από τις μεθόδους withAttribute. Να σημειωθεί ότι όλες αυτές οι μέθοδοι επιστρέφουν ένα νέο αντικείμενο με την τροποποιημένη ιδιότητα, χωρίς να μεταλλάσσουν το υπάρχον αντικείμενο. Το ίδιο μπορεί να γίνει και με την πιο γενική μέθοδο with που παίρνει ως πρώτο όρισμα ένα TemporalField.

```
LocalDate date1 = LocalDate.of(2014, 3, 18);     <— 2014-03-18
LocalDate date2 = date1.withYear(2011);                <— 2011-03-18
LocalDate date3 = date2.withDayOfMonth(25);                  <— 2011-03-25
LocalDate date4 = date3.with(ChronoField.MONTH_OF_YEAR, 9);  <— 2011-09-25
```

Εικόνα 36: Διαχείριση ιδιοτήτων της LocalDate με απόλυτο τρόπο

Η μέθοδος with όπως και η μέθοδος get, δηλώνονται στη διεπαφή Temporal που υλοποιείται από όλες τις κλάσεις του Date και Time API, οι οποίες ορίζουν ένα μοναδικό σημείο στο χρόνο. Πιο συγκεκριμένα, οι δυο μέθοδοι επιτρέπουν αντίστοιχα το διάβασμα και την τροποποίηση τις τιμής ενός πεδίου ενός αντικειμένου τύπου Temporal. Εάν το ζητούμενο πεδίο δεν υποστηρίζεται από τη συγκεκριμένη διεπαφή Temporal, οι μέθοδοι θα πετάξουν μια εξαίρεση Unsupported-TemporalTypeException

Οι γενικές μέθοδοι plus και minus, που δηλώνονται στη διεπαφή Temporal, επιτρέπουν την μετακίνηση προς τα πίσω ή προς τα εμπρός ενός δεδομένου χρονικού διαστήματος, κατά έναν αριθμό συν μια Temporal-Unit, όπου η απαρίθμηση ChronoUnit προσφέρει μια βολική υλοποίηση της διεπαφής TemporalUnit.

```
LocalDate date1 = LocalDate.of(2014, 3, 18);     <— 2014-03-18
LocalDate date2 = date1.plusWeeks(1);                  <— 2014-03-18
LocalDate date3 = date2.minusYears(3);                       <— 2011-03-25
LocalDate date4 = date3.plus(6, ChronoUnit.MONTHS);          <— 2011-09-25
```

Εικόνα 37: Διαχείριση ιδιοτήτων της LocalDate με σε σχετικό

Οι πιο συνηθισμένες μέθοδοι των κλάσεων ημερομηνίας και ώρας που αναπαριστούν ένα σημείο στο χρόνο (point in time) συγκεντρώνονται στον παρακάτω πίνακα.

Πίνακας 9: Μέθοδοι για την αναπαράσταση σημείων στο χρόνο (point in time).

Μέθοδος	Στατική	Περιγραφή
from	Ναι	Δημιουργεί ένα στιγμιότυπο της κλασης από το χρονικό αντικείμενο που δέχεται ως παράμετρο
no	Ν	Δημιουργεί ένα χρονικό αντικείμενο από το ρολόι του

w	αι	συστήματος
of	Ναι	Δημιουργεί ένα στιγμιότυπο του χρονικού αντικειμένου από τα συστατικά του μέρη
parse	Ναι	Δημιουργεί ένα στιγμιότυπο του χρονικού αντικειμένου από ένα String
atOffset	Όχι	Συνδυάζει το χρονικό αντικείμενο με το offset της ζώνης
atZone	Όχι	Συνδυάζει το χρονικό αντικείμενο με μια ζώνη ώρας
format	Όχι	Μετατρέπει το χρονικό αντικείμενο σε String χρησιμοποιώντας το συγκεκριμένο διαμορφωτή (δεν διατίθεται για την Instant)
get	Όχι	Διαβάζει μέρος της δήλωσης (part of the state)του χρονικού αντικειμένου
minus	Όχι	Δημιουργεί ένα αντίγραφο του χρονικού αντικειμένου αφαιρώντας ένα χρονικό διάστημα
plus	Όχι	Δημιουργεί ένα αντίγραφο του χρονικού αντικειμένου προσθέτοντας ένα χρονικό διάστημα
with	Όχι	Δημιουργεί ένα αντίγραφο του χρονικού αντικειμένου αλλάζοντας ένα μέρος της δήλωσης του (part of the state)

Μερικές φορές, μπορεί να χρειαστεί να εκτελεστούν πιο προηγμένες λειτουργίες, όπως η ρύθμιση ημερομηνίας για την επόμενη Κυριακή, την επόμενη εργάσιμη ημέρα, ή την τελευταία ημέρα του μήνα. Σε τέτοιες περιπτώσεις, χρησιμοποιείται η υπερφορτωμένη έκδοση της μεθόδου with, στην οποία περνάει ως παράμετρος ένας TemporalAdjuster, που παρέχει έναν πιο προσαρμόσιμο τρόπο για τον καθορισμό της διαχείρισης που απαιτείται για να λειτουργήσει σε μια συγκεκριμένη ημερομηνία. Το Date και Time API παρέχει ήδη πολλούς προκαθορισμένους ρυθμιστές χρόνου (Temporal-Adjusters) για τις πιο συνηθισμένες περιπτώσεις χρήσης. Η πρόσβαση στους χρονικούς ρυθμιστές γίνεται χρησιμοποιώντας τις στατικές μεθόδους που περιέχονται στην κλάση TemporalAdjusters.

```
import static java.time.temporal.TemporalAdjusters.*;

LocalDate date1 = LocalDate.of(2014, 3, 18);                    2014-03-18
LocalDate date2 = date1.with(nextOrSame(DayOfWeek.SUNDAY)); <-- 2014-03-23
LocalDate date3 = date2.with(lastDayOfMonth());                 2014-03-31
```

Εικόνα 38: Χρήση των προκαθορισμένων TemporalAdjusters.

Στον παρακάτω πίνακα παρουσιάζονται οι μέθοδοι που είναι διαθέσιμες από την κλάση TemporalAdjusters:

Πίνακας 10: Οι μέθοδοι της κλάσης TemporalAdjusters:

Μέθοδος	Περιγραφή
dayOfWeekInMonth	Δημιουργεί μια νέα ημερομηνία με τον ίδιο μήνα με τη τακτική (ordinal) ημέρα της εβδομάδας

firstDayOfMonth	Δημιουργεί μια νέα ημερομηνία που ορίζεται από την πρώτη ημέρα του τρέχοντος μήνα
firstDayOfNextMonth	Δημιουργεί μια νέα ημερομηνία που ορίζεται από την πρώτη ημέρα του επόμενου μήνα
firstDayOfNextYear	Δημιουργεί μια νέα ημερομηνία που ορίζεται από την πρώτη ημέρα του επόμενου έτους
firstDayOfYear	Δημιουργεί μια νέα ημερομηνία που ορίζεται από την πρώτη ημέρα του τρέχοντος έτους
firstInMonth	Δημιουργεί μια νέα ημερομηνία με τον ίδιο μήνα με την πρώτη αντίστοιχη ημέρα της εβδομάδας
lastDayOfMonth	Δημιουργεί μια νέα ημερομηνία που ορίζεται από την τελευταία ημέρα του τρέχοντος μήνα
lastDayOfNextMonth	Δημιουργεί μια νέα ημερομηνία που ορίζεται από την τελευταία ημέρα του επόμενου μήνα
lastDayOfNextYear	Δημιουργεί μια νέα ημερομηνία που ορίζεται από την τελευταία ημέρα του επόμενου έτους
lastDayOfYear	Δημιουργεί μια νέα ημερομηνία που ορίζεται από την τελευταία ημέρα του τρέχοντος έτους
lastInMonth	Δημιουργεί μια νέα ημερομηνία με τον ίδιο μήνα με την τελευταία αντίστοιχη ημέρα της εβδομάδας
next previous	Δημιουργεί μια νέα ημερομηνία που ορίζεται από την πρώτη εμφάνιση της συγκεκριμένης ημέρας της εβδομάδας, μετά / πριν από την ημερομηνία που προσαρμόζεται (adjusted)
nextOrSame	Δημιουργεί μια νέα ημερομηνία που ορίζεται από την πρώτη εμφάνιση της συγκεκριμένης ημέρας της εβδομάδας μετά / πριν από την ημερομηνία που ρυθμίζεται εκτός εάν είναι ήδη εκείνη η ημέρα, οπότε επιστρέφεται το ίδιο αντικείμενο
previousOrSame	

Οι μέθοδοι της κλασης TemporalAdjusters επιτρέπουν την εκτέλεση πιο πολύπλοκων χειρισμών ημερομηνίας που μπορούν να διαβαστούν σαν την εκφώνηση του προβλήματος. Επιπλέον, είναι σχετικά απλό για το χρήστη να δημιουργήσει τη δική του προσαρμοσμένη TemporalAdjuster υλοποίηση εάν δεν βρει κάποιον προκαθορισμένο TemporalAdjuster που να ταιριάζει στις ανάγκες του.

Στην πραγματικότητα, η διεπαφή TemporalAdjuster ορίζει μόνο μία μέθοδο:

· @FunctionalInterface
public interface TemporalAdjuster {

Temporal adjustInto(Temporal temporal);

}

Αυτό σημαίνει ότι μια υλοποίηση της διεπαφής TemporalAdjuster ορίζει πώς γίνεται η μετατροπή από ένα αντικείμενο τύπου Temporal σε ένα άλλο αντικείμενο τύπου Temporal.

Μια ακόμη συνηθισμένη λειτουργία που μπορεί να εκτελεστεί στα αντικείμενα ημερομηνίας και ώρας είναι η εκτύπωση τους σε διάφορες μορφές ανάλογα με τις ανάγκες του χρήστη. Επίσης, μπορεί να χρειαστεί και η μετατροπή κάποιων Strings που αναπαριστούν ημερομηνίες σε αυτές τις μορφές, σε πραγματικά αντικείμενα ημερομηνίας (actual date objects).

Η μορφοποίηση (formatting) και η ανάλυση (parsing) είναι βασικές λειτουργίες, όταν ασχολείται κανείς με ημερομηνίες και ώρες. Το νέο πακέτο java.time.format είναι εξ ολοκλήρου αφιερωμένο σε αυτό το σκοπό. Η πιο σημαντική κλάση του πακέτου είναι η DateTimeFormatter. Ο ευκολότερος τρόπος να δημιουργηθεί ένας διαμορφωτής (formatter) είναι μέσω των στατικών μεθόδων και των σταθερών που διαθέτει η συγκεκριμένη κλάση. Οι σταθερές όπως η BASIC_ISO_DATE και η ISO_LOCAL_DATE είναι απλώς προκαθορισμένα στιγμιότυπα (instances) της κλασης DateTimeFormatter. Όλοι οι DateTimeFormatters μπορούν να χρησιμοποιηθούν για την δημιουργία ενός String που αναπαριστά μια συγκεκριμένη ημερομηνία ή ώρα σε μια συγκεκριμένη μορφή.

Παράδειγμα: Παραγωγή Strings με χρήση δυο διαφορετικών διαμορφωτών.

```
LocalDate date = LocalDate.of(2014, 3, 18);
String s1 = date.format(DateTimeFormatter.BASIC_ISO_DATE);   <- 20140318
String s2 = date.format(DateTimeFormatter.ISO_LOCAL_DATE);   <- 2014-03-18
```

Εικόνα 39: Δυο διαφορετικοί διαμορφωτές για την δημιουργία του ίδιου String.

Υπάρχει επιπλέον η δυνατότητα ανάλυσης ενός String που αναπαριστά μια ημερομηνία ή μια ώρα σε αυτή τη μορφή, για να δημιουργηθεί εκ νέου το ίδιο το αντικείμενο ημερομηνίας. Η μέθοδος parse, η οποία παρέχεται από όλες τις κλάσεις του Date και Time API και αναπαριστά ένα σημείο στο χρόνο ή ένα χρονικό διάστημα, εκτελεί αυτή την λειτουργία:

- LocalDate date1 = LocalDate.parse("20140318",
  DateTimeFormatter.BASIC_ISO_DATE);
- LocalDate date2 = LocalDate.parse("2014-03-18",
  DateTimeFormatter.ISO_LOCAL_DATE);

Σε σύγκριση με την παλιά τάξη java.util.DateFormat, όλα τα DateTimeFormatter στιγμιότυπα είναι νηματικά-ασφαλή (thread-safe). Για το λόγο αυτό, είναι δυνατή η δημιουργία μεμονωμένων διαμορφωτών (singleton formatters), όπως αυτών που ορίζονται από τις σταθερές της DateTimeFormatter, οι οποίοι μπορούν να διαμοιράζονται μεταξύ πολλών νημάτων. Η κλάση DateTimeFormatter υποστηρίζει επίσης την στατική μέθοδο ofPattern που επιτρέπει την δημιουργία ενός διαμορφωτή σύμφωνα με κάποιο συγκεκριμένο μοτίβο.

Παράδειγμα: Δημιουργία μιας DateTimeFormatter με βάση ένα συγκεκριμένο πρότυπο

· DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
· LocalDate date1 = LocalDate.of(2014, 3, 18);
· String formattedDate = date1.format(formatter);
· LocalDate date2 = LocalDate.parse(formattedDate, formatter);

Η ofPattern μέθοδος έχει μια υπερφορτωμένη έκδοση που επιτρέπει την δημιουργία ενός διαμορφωτή για ένα δεδομένο τόπο (Locale).

Παράδειγμα: Δημιουργία μιας τοπικής DateTimeFormatter

· DateTimeFormatter italianFormatter = DateTimeFormatter.ofPattern("d. MMMM yyyy", Locale.ITALIAN);
· LocalDate date1 = LocalDate.of(2014, 3, 18);
· String formattedDate = date.format(italianFormatter); // 18. marzo 2014
· LocalDate date2 = LocalDate.parse(formattedDate, italianFormatter);

Τέλος, σε περίπτωση που χρειάζεται ακόμη περισσότερος έλεγχος, η κλάση DateTimeFormatterBuilder επιτρέπει τον ορισμό βήμα προς βήμα σύνθετων διαμορφωτών χρησιμοποιώντας μεθόδους με νόημα (meaningful methods). Επιπλέον, παρέχει τη δυνατότητα case-insensitive ανάλυσης, lenient ανάλυσης, padding, και optional sections του διαμορφωτή.

Παράδειγμα: Δημιουργία μιας DateTimeFormatter με χρήση της κλασης DateTimeFormatterBuilder

· DateTimeFormatter italianFormatter = new DateTimeFormatterBuilder()
.appendText(ChronoField.DAY_OF_MONTH)

.appendLiteral(". ")

.appendText(ChronoField.MONTH_OF_YEAR)

.appendLiteral(" ")

.appendText(ChronoField.YEAR)

.parseCaseInsensitive()

.toFormatter(Locale.ITALIAN);

Μέχρι τώρα περιγράφηκε ο τρόπος για το πώς δημιουργούνται, διαχειρίζονται, μορφοποιούνται και αναλύονται είτε οι χρονικές στιγμές είτε τα χρονικά διαστήματα. Το μόνο που έμεινε είναι η αντιμετώπιση κάποιων λεπτών ζητημάτων που αφορούν την ημερομηνία και την ώρα.

5.3 ΔΟΥΛΕΥΟΝΤΑΣ ΜΕ ΔΙΑΦΟΡΕΤΙΚΕΣ ΧΡΟΝΙΚΕΣ ΖΩΝΕΣ ΚΑΙ ΗΜΕΡΟΛΟΓΙΑ

Η αντιμετώπιση ζητημάτων που έχουν να κάνουν με τις ζώνες ώρας είναι ένα σημαντικό θέμα που ήδη έχει απλοποιηθεί από το νέο Date και Time API[5]. Η νέα κλάση java.time.ZoneId που αντικατέστησε την παλαιότερη κλάση java.util.TimeZone, έχει ως στόχο την παροχή «προστασίας» από τις περιπλοκές που σχετίζονται με τις ζώνες ώρας, όπως την αντιμετώπιση της Θερινής ώρας (Daylight Saving Time - DST). Όπως και οι άλλες κλάσεις του Date και Time API, είναι και αυτή αμετάβλητη.

Η ζώνη ώρας είναι ένα σύνολο κανόνων που αντιστοιχεί σε μια περιοχή, στην οποία η χειμερινή ώρα (standard time) είναι η ίδια. Υπάρχουν περίπου 40 κανόνες που περιέχονται στα στιγμιότυπα της κλασης ZoneRules. Με την κλήση της μεθόδου getRules() σε μια κλάση ZoneId μπορούν να αποκτηθούν οι βασικοί κανόνες για μια συγκεκριμένη ζώνη ώρας. Η ακριβής ZoneId καθορίζεται από το χαρακτηριστικό αναγνωριστικό (ID) της κάθε περιοχής. Τα αναγνωριστικά των περιοχών έχουν όλα την μορφή “{περιοχή} / {πόλη}” και το σύνολο των διαθέσιμων τοποθεσιών παρέχεται από τη βάση δεδομένων IANA Ζώνη ώρας.

· ZoneId romeZone = ZoneId.of("Europe/Rome");

Η μετατροπή από ένα παλιό αντικείμενο τύπου TimeZone σε ένα αντικείμενο τύπου ZoneId επιτυγχάνεται με τη χρήση της νέας μεθόδου toZoneId:

· ZoneId zoneId = TimeZone.getDefault().toZoneId();

Ένα αντικείμενο ZoneId, μπορεί να συνδυαστεί με μια LocalDate, μια LocalDateTime, ή μια Instant, για να μετατραπεί σε ZonedDateTime στιγμιότυπα, τα οποία αναπαριστούν χρονικά σημεία που σχετίζονται με κάποια συγκεκριμένη ζώνη ώρας.

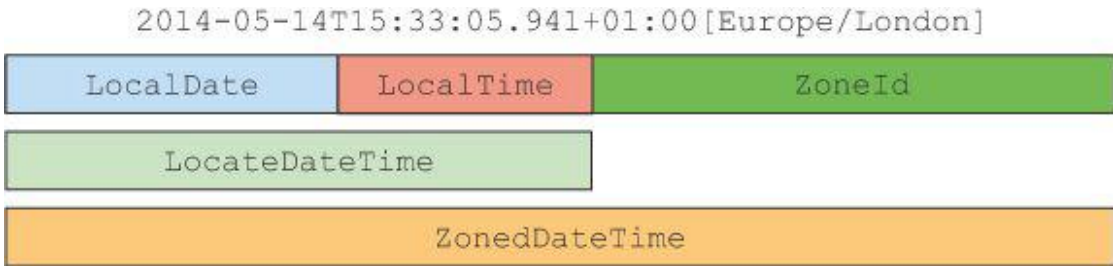
Παράδειγμα: Εφαρμογή μιας ζώνης ώρας σε ένα χρονικό σημείο (a point in time)

· LocalDate date = LocalDate.of(2014, Month.MARCH, 18);
· ZonedDateTime zdt1 = date.atStartOfDay(romeZone);

· LocalDateTime dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45);
· ZonedDateTime zdt2 = dateTime.atZone(romeZone);

· Instant instant = Instant.now();
· ZonedDateTime zdt3 = instant.atZone(romeZone);

Τα συστατικά μιας ZonedDateTime προκειμένου να γίνουν κατανοητές οι διαφορές μεταξύ των LocaleDate, LocalTime, LocalDateTime και ZoneId.



Εικόνα 40: ZonedDateTime και η σχεση της με τις LocaleDate, LocalTime, ZoneId και LocalDateTime.

Η μετατροπή ενός αντικειμένου LocalDateTime σε ένα αντικείμενο Instant μπορεί να γίνει και με την χρήση της ZoneId

· LocalDateTime dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45);
· Instant instantFromDateTime = dateTime.toInstant(romeZone);

ή

· Instant instant = Instant.now();
· LocalDateTime timeFromInstant = LocalDateTime.ofInstant(instant, romeZone);

Ένας άλλος συνηθισμένος τρόπος για να εκφραστεί μια ζώνη ώρας είναι με τη σταθερή αντιστάθμιση (offset) από την ζώνη UTC / Greenwich. Σε τέτοιες περιπτώσεις, χρησιμοποιείται η κλάση ZoneOffset, μια υποκλάση της ZoneId που αναπαριστά τη διαφορά μεταξύ μιας ώρας και της ώρας Greenwich:

Παράδειγμα: Η ώρα της Νέας Υόρκης σε σχέση με την ώρα Greenwich

· ZoneOffset newYorkOffset = ZoneOffset.of("-05:00");

Η ZoneOffset ορίζεται με τρόπο που δεν προβλέπει την διαχείρισης της Θερινής ώρας και για το λόγο αυτό δεν συνιστάται στην πλειονότητα των περιπτώσεων.

Ένα OffsetDateTime, αναπαριστά μια ημερομηνία-ώρα με μια μετατόπιση (offset) από το UTC / Greenwich στο ημερολόγιο του συστήματος ISO-8601 και μπορεί να δημιουργηθεί ως εξής:

· LocalDateTime dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45);
· OffsetDateTime dateTimeInNewYork = OffsetDateTime.of(date, newYorkOffset);

Ένα ακόμη προηγμένο χαρακτηριστικό του νέου Date και Time API είναι η υποστήριξη που παρέχεται για τα συστήματα ημερολογίου χωρίς ISO (non-ISO calendaring systems).

Το ISO-8601 ημερολόγιο συστήματος είναι το de facto παγκόσμιο πολιτικό σύστημα ημερολογίου. Όμως, η Java 8 παρέχει τέσσερα πρόσθετα ημερολογιακά συστήματα με την αντίστοιχη ειδική κλάση ημερομηνίας για το καθένα από αυτά:

- ThaiBuddhistDate
- MinguoDate
- JapaneseDate
- HijrahDate

Όλα αυτές οι κλάσεις, μαζί με LocalDate υλοποιούν την διεπαφή ChronoLocalDate που προορίζεται για την διαμόρφωση μιας ημερομηνίας σε μια αυθαίρετη χρονολογία.

Οποιοδήποτε στιγμιότυπο τύπου Temporal μπορεί να δημιουργηθεί με την βοήθεια των στατικών μεθόδων των παραπάνω κλάσεων:

- LocalDate date = LocalDate.of(2014, Month.MARCH, 18);
- JapaneseDate japaneseDate = JapaneseDate.from(date);

Εναλλακτικά, μπορεί να δημιουργηθεί ρητά ένα σύστημα ημερολογίου για μια συγκεκριμένη τοποθεσία Locale και να δημιουργηθεί ένα στιγμιότυπο μιας ημερομηνίας για την εν λόγω τοποθεσία. Στο νέο Date και Time API, η διεπαφή Chronology είναι αυτή που μοντελοποιεί ένα ημερολογιακό σύστημα και με την στατική μέθοδο ofLocale που διαθέτει, μπορεί να αποκτηθεί ένα στιγμιότυπο της.

- Chronology japaneseChronology = Chronology.ofLocale(Locale.JAPAN);
- ChronoLocalDate now = japaneseChronology.dateNow();

Οι σχεδιαστές του Date και Time API στις περισσότερες περιπτώσεις χρησιμοποιούν την LocalDate αντί της Chrono-LocalDate για να κάνουν ενημερώσεις και αυτό επειδή κάποιος προγραμματιστής θα μπορούσε να κάνει υποθέσεις (π.χ. ο χρόνος αποτελείται από 12 μήνες ή κάθε χρόνος έχει σταθερό αριθμό μηνών κ.τ.λ.) για τον κώδικά τους, που δεν θα ισχύουν σε ένα πολύ-ημερολογιακό (multicalendar) σύστημα.. Για τους λόγους αυτούς, συνιστάται η χρησιμοποίηση της LocalDate από τον προγραμματιστή κατά την διάρκεια της εφαρμογής, συμπεριλαμβανομένων όλων των αποθηκευτικών χώρων, το χειρισμό και την ερμηνεία των κανόνων, ενώ η Chrono-LocalDate καλό είναι να χρησιμοποιείται μόνο όταν πρέπει να εντοπιστεί η είσοδος ή η έξοδος του προγράμματος.

ΚΕΦΑΛΑΙΟ 6

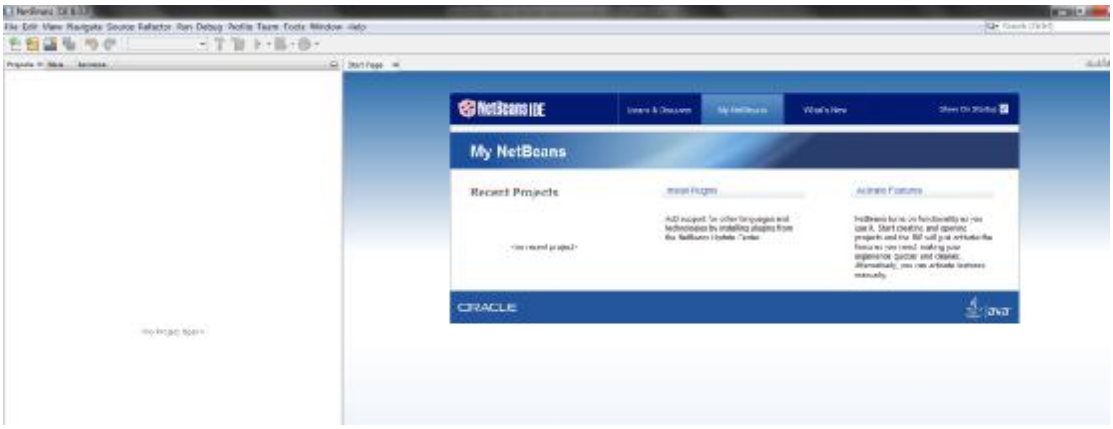
ΥΛΟΠΟΙΗΣΕΙΣ ΚΑΙ LAMDAS EXPRESSIONS

6.1 ΕΙΣΑΓΩΓΗ

Σκοπός του παρόντος κεφαλαίου είναι να δούμε σε παραδείγματα θέματα σχετικά με τις λάμδα εκφράσεις που είδαμε στα προηγούμενα κεφάλαια[1-6]. Βασικό χαρακτηριστικό των λάμδα εκφράσεων είναι η ικανότητα του να περάσουμε συμπεριφορές σε μεθόδους. Πριν τη Java 8 , εάν θέλαμε να περάσουμε συμπεριφορά σε μία μέθοδο θα έπρεπε να χρησιμοποιήσουμε μία ανώνυμη κλάση και έξι γραμμές κώδικα. Η λάμδα έκφραση αντικαθιστά τις ανώνυμες κλάσεις δίνοντάς μας την ικανότητα να γράψουμε κώδικα πιο αναγνώσιμο και εκφραστικό. Στη συνέχεια θα παρουσιάσουμε έναν αριθμό από απλά και σύντομα παραδείγματα.

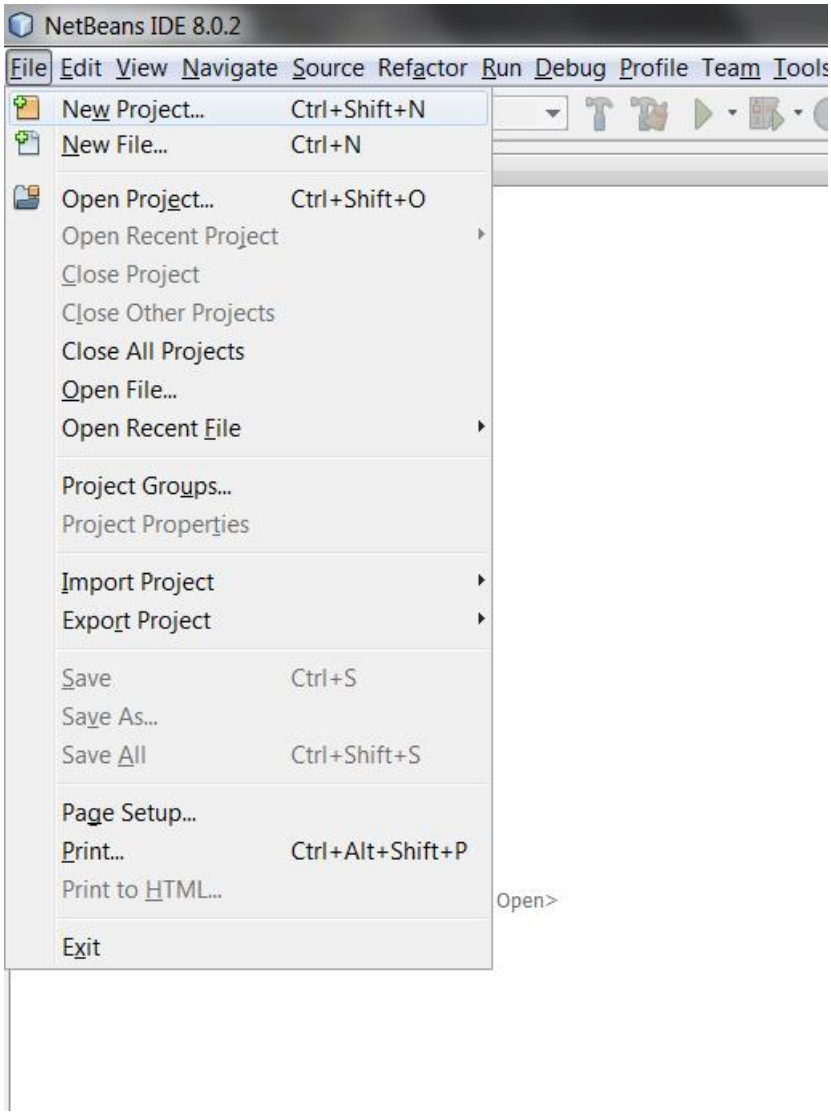
6.2 ΥΛΟΠΟΙΗΣΗ ΛΑΜΔΑ ΕΚΦΡΑΣΕΩΝ

Για την υλοποίηση του κώδικα χρησιμοποιήσαμε το πρόγραμμα Netbeans IDE 8.* και φυσικά την έκδοση Java 8. Ανοίγοντας το πρόγραμμα έχουμε το περιβάλλον ανάπτυξης που φαίνεται στην παρακάτω εικόνα.



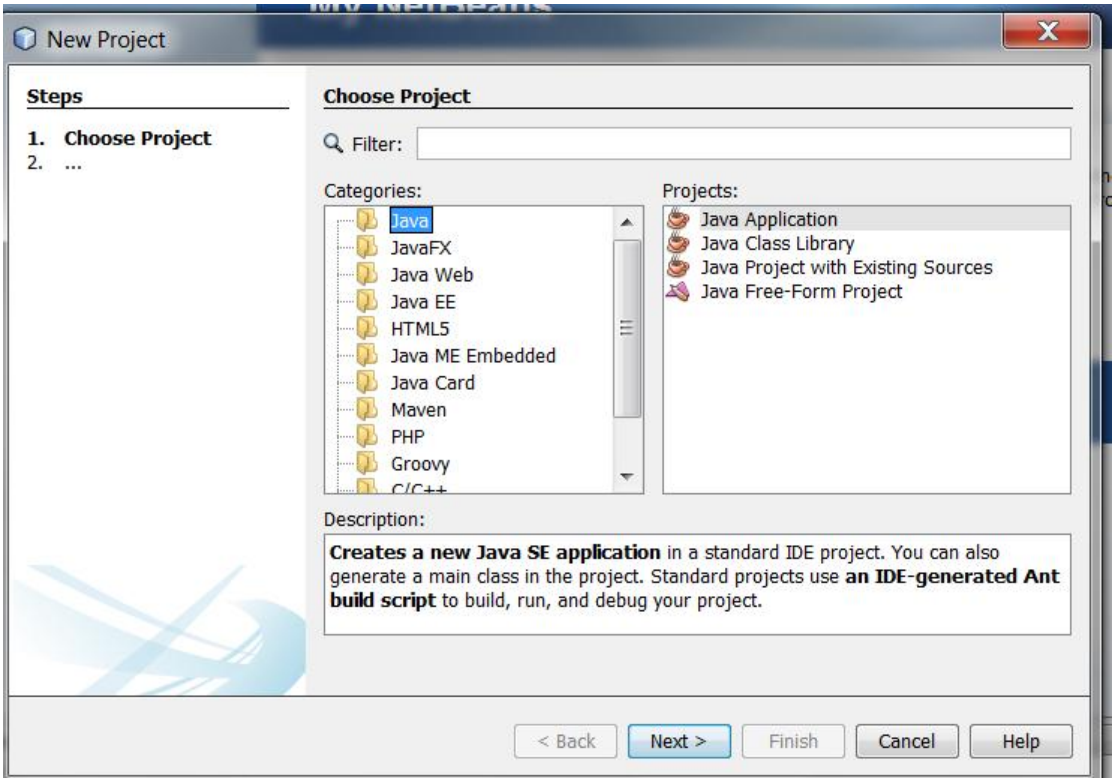
Εικόνα 41: Περιβάλλον ανάπτυξης Netbeans IDE

Για να τρέξουμε το κώδικα που θα γράψουμε, θα πρέπει πρώτα να φτιάξουμε ένα έργο(project). Επιλέγουμε από το μενού File -> New Project.



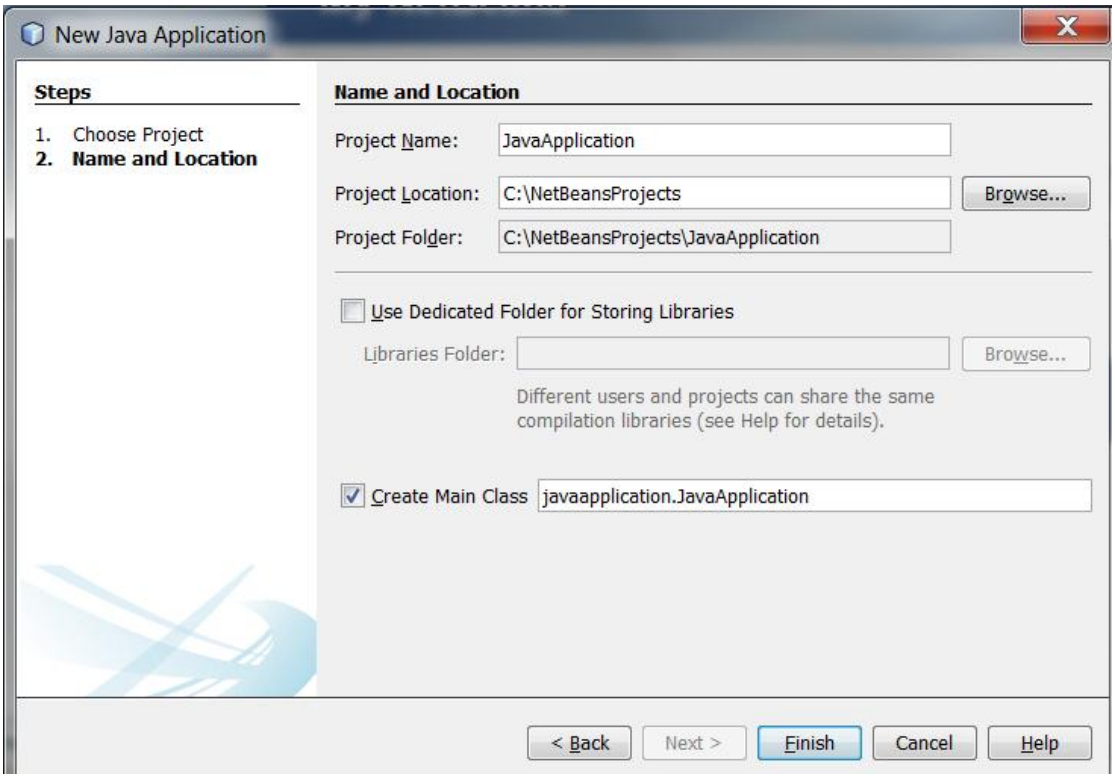
Εικόνα 42: Δημιουργία ενός νέου έργου στο Netbeans περιβάλλον

Στην επόμενη εικόνα επιλέγουμε το έργο που θέλουμε να δημιουργήσουμε.



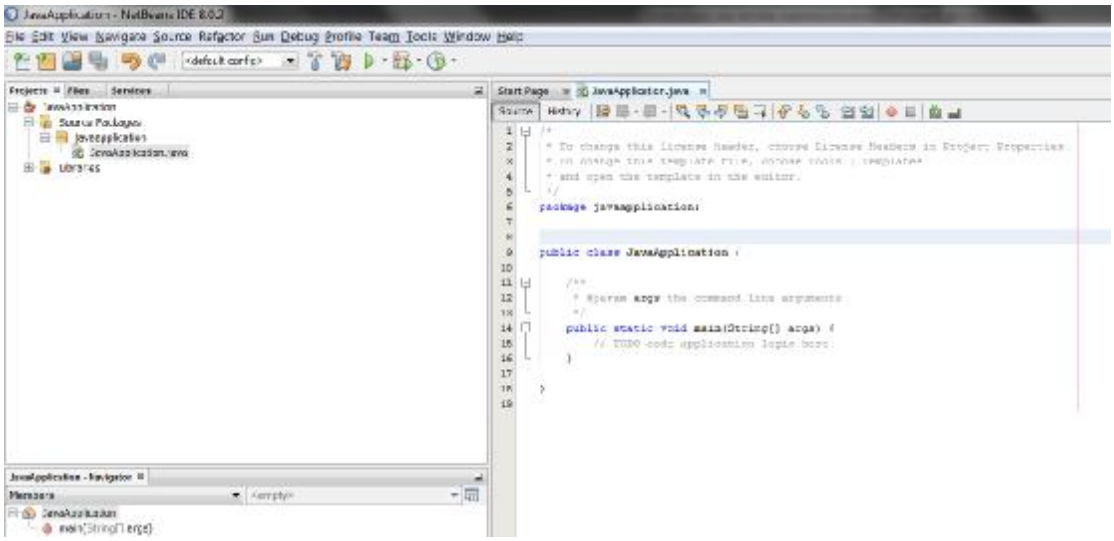
Εικόνα 43: Δημιουργία ενός νέου έργου στο Netbeans περιβάλλον- βήμα 2

Επιλέγουμε Java Application και πατάμε Next. Στην επόμενη εικόνα επιλέγουμε το όνομα του έργου μας και το που θα δημιουργηθεί και πατάμε Finish.



Εικόνα 44 Δημιουργία ενός νέου έργου στο Netbeans περιβάλλον- βήμα 3

Το αποτέλεσμα φαίνεται στην επόμενη εικόνα.



Εικόνα 45: Δημιουργία του πρώτου μας έργου

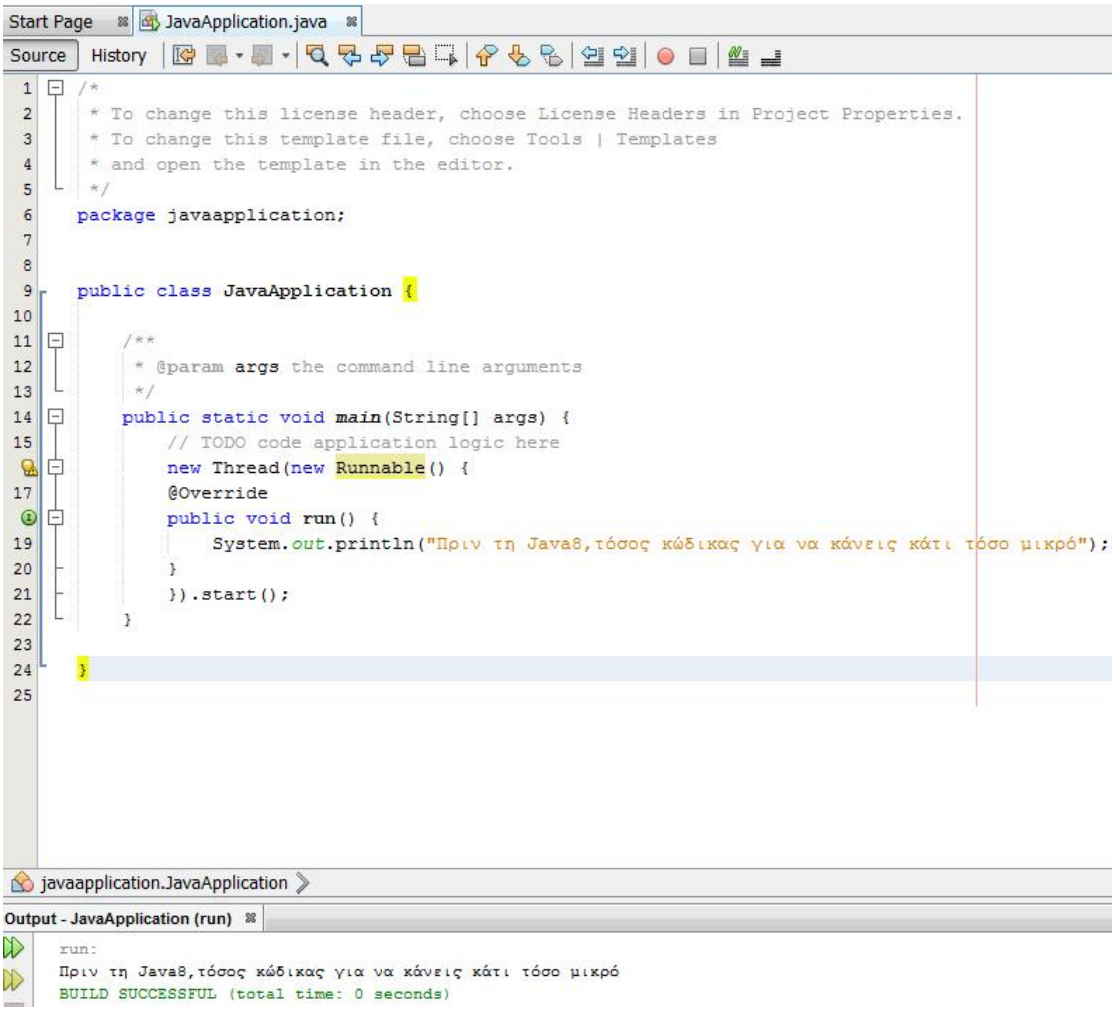
6.2.1 Παράδειγμα 1 – Υλοποιώντας την Runnable χρησιμοποιώντας τη Λάμδα έκφραση

Το πρώτο πράγμα που θα κάνουμε είναι να αντικαταστήσουμε μία ανώνυμη κλάση με μια λάμδα έκφραση. Θα προχωρήσουμε στην υλοποίηση της διεπαφής Runnable. Ο κώδικας υλοποίησης πριν τη Java 8 θα μας πάρει 4 γραμμές, ενώ με τη λάμδα έκφραση μία γραμμή.

Πριν τη Java 8 ο κώδικας είναι

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Πριν τη Java8,τόσος κώδικας για να κάνεις κάτι τόσο μικρό");
    }
}).start();
```

Εκτελώντας το στον πρόγραμμα Netbeans έχουμε το εξής αποτέλεσμα.

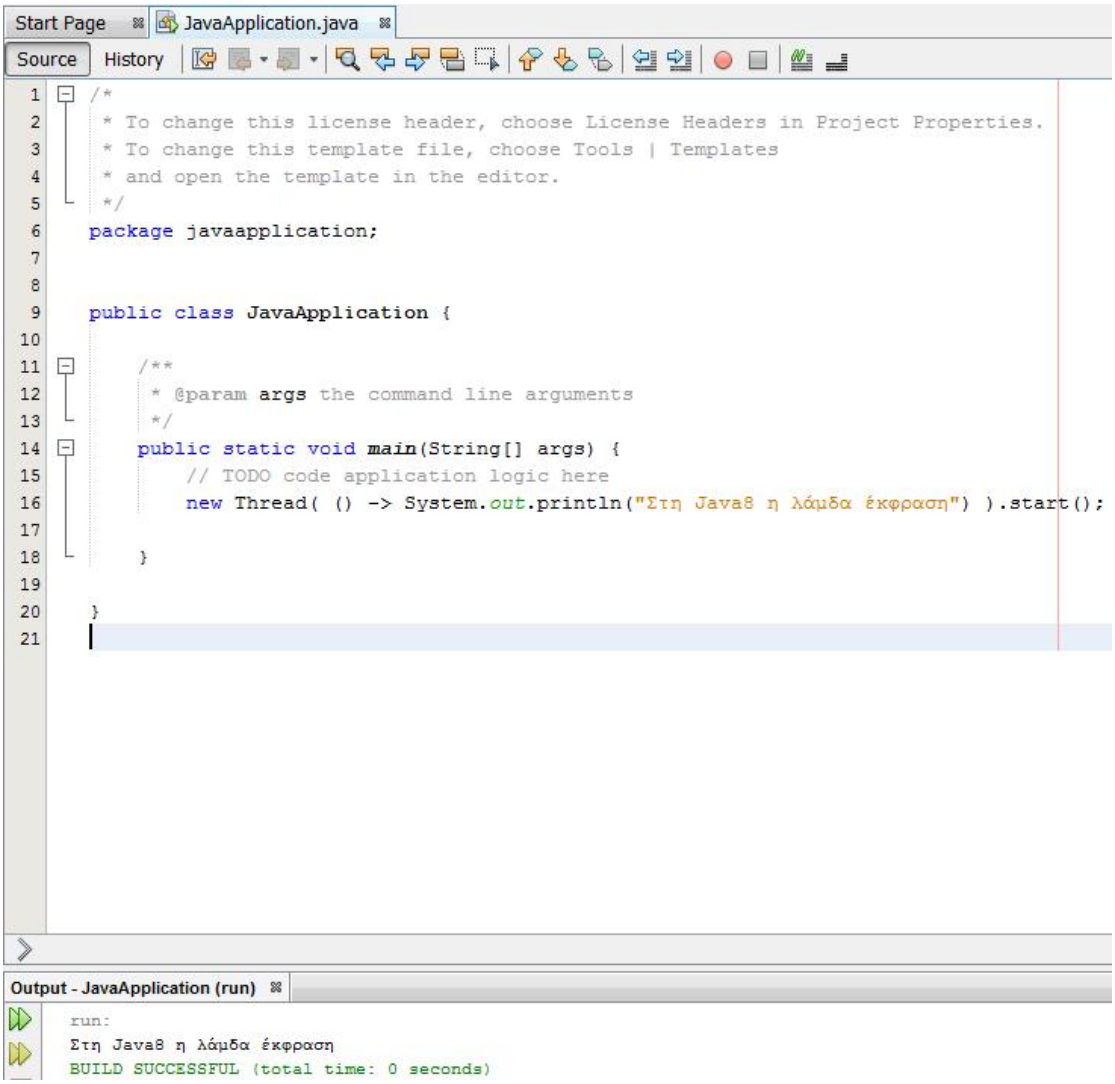


Εικόνα 46: Διεπαφή Runnable

Στην Java 8 το παράδειγμά μας μπορεί να γραφεί ως εξής:

```
new Thread( () -> System.out.println("Στη Java8 η λάμδα έκφραση") ).start();
```

Η εικόνα στο πρόγραμμα Netbeans.



Εικόνα 47: Λάμδα έκφραση

6.2.2 Παράδειγμα 2 – Διαχείριση συμβάντος χρησιμοποιώντας λάμδα εκφράσεις

Ο even Listener είναι μία ακόμη ανώνυμη έκφραση που θα αντικαταστήσουμε με λάμδα έκφραση.

Πριν τη Java 8 ο κώδικας είναι

```
JButton show =  new JButton("Δείξε");
show.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(java.awt.event.ActionEvent e) {
        System.out.println("Διαχείριση συμβάντος χωρίς λάμδα έκφραση");
        throw new UnsupportedOperationException("");
      }
    });
```

Με τη χρήση της λάμδα έκφρασης της Java 8

```
JButton show =  new JButton("Δείξε");
show.addActionListener((java.awt.event.ActionEvent e) -> {
        System.out.println("Διαχείριση συμβάντος με λάμδα έκφραση");
        throw new UnsupportedOperationException(""); //To change body of
generated methods, choose Tools | Templates.
    });
```

6.2.3 Παράδειγμα επανάληψης σε μία λίστα χρησιμοποιώντας τις λάμδα εκφράσεις

Η Java από τη στιγμή που είναι γλώσσα επιτακτική, ο κώδικας επανάληψης πριν τη Java 8 ήταν ακολουθιακός και υπάρχει μόνο ένας τρόπος για παράλληλη επεξεργασία των αντικειμένων της λίστας. Εάν θέλουμε να κάνουμε παράλληλο φιλτράρισμα πρέπει να γράψουμε τον δικό μας κώδικα.

Η εισαγωγή των λάμδα εκφράσεων και των προκαθορισμένων μεθόδων, ξεχώρισε το τι θέλουμε να κάνουμε με το πώς θέλουμε να το κάνουμε, πράγμα που σημαίνει ότι η Java Collection γνωρίζει πώς να προχωρήσει σε επανάληψη, και μας παρέχει παράλληλη επεξεργασία σε στοιχεία μιας συλλογής σε API επίπεδο.

Στο επόμενο παράδειγμα θα δούμε πως προχωράμε σε μία επανάληψη σε μία λίστα χωρίς και με λάμδα εκφράσεις.

Πριν τη Java 8

```
List features = Arrays.asList("Λάμδας", "Προκαθορισμένες μέθοδοι", "Stream
API", "Date and Time API");
for (Object feature : features) {
  System.out.println(feature);
}
```

Κώδικας στη Java 8

```
List features = Arrays.asList("Λάμδας", "Προκαθορισμένες μέθοδοι", "Stream
API", "Date and Time API");
features.forEach(n -> System.out.println(n));
```

6.2.4 Παράδειγμα 4 – Εφαρμόζοντας μια συνάρτηση σε κάθε αντικείμενο της λίστας

Συχνά χρειάζεται να εφαρμόσουμε συγκεκριμένη συνάρτηση σε κάθε στοιχείο μίας λίστας, για παράδειγμα ο πολλαπλασιασμός του με ένα συγκεκριμένο αριθμό ή η διαίρεσή του. Αυτές οι λειτουργίες ταιριάζουν στη συνάρτηση map() της Java όπου μπορούμε να παρέχουμε το μετασχηματισμό λογικής στην συνάρτηση map() ως λάμδα έκφραση και να μετασχηματίσει κάθε στοιχείο της συλλογής αυτής όπως φαίνεται στο παρακάτω παράδειγμα.

```
List<String> countries = Arrays.asList ("Ιαπωνία", "Αμερική", "Γερμανία", "Γαλλία", "Ιταλία", "Αγγλία","Καναδάς");
String Countries = countries.stream().map(x -> x.toUpperCase()).collect(Collectors.joining(", "));
System.out.println(Countries);
```

Η έξοδος του προγράμματος είναι:

```
ΙΑΠΩΝΙΑ, ΑΜΕΡΙΚΉ, ΓΕΡΜΑΝΙΑ, ΓΑΛΛΊΑ, ΙΤΑΛΊΑ, ΑΓΓΛΊΑ, ΚΑΝΑΔΆΣ
```

6.2.5 Παράδειγμα 5 – Δημιουργία μίας υπολίστας

Στο παρακάτω παράδειγμα θα χρησιμοποιήσουμε την distinct μέθοδο της κλάσης Stream για να φιλτράρει τις διπλές εγγραφές σε μία συλλογή.

```
List<Integer> numbers = Arrays.asList(8, 1, 4, 3, 7, 3, 4);
List<Integer> distinct = numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
System.out.printf("Αρχική λίστα : %s,  Τετράγωνα χωρίς διπλοεγγραφές : %s %n", numbers, distinct);
```

Η έξοδος του προγράμματος είναι:

```
Αρχική λίστα : [8, 1, 4, 3, 7, 3, 4],  Τετράγωνα χωρίς διπλοεγγραφές : [64, 1, 16, 9, 49]
```

6.2.6 Παράδειγμα 6 – Υπολογίζοντας την μέγιστη τιμή, την ελάχιστη τιμή, το άθροισμα και το μέσο όρο των στοιχείων μίας λίστας

Υπάρχει μία χρήσιμη μέθοδο με το όνομα summaryStattics()στις κλάσεις Stream η οποία επιστρέφει:

- IntSummaryStatistics,
- LongSummaryStatistics
- DoubleSummaryStatistics

περιγράφοντας διάφορα αθροιστικά δεδομένα σχετικά με τα στοιχεία του συγκεκριμένου Stream. Στο επόμενο παράδειγμά μας χρησιμοποιούμε την μέθοδο αυτή για να υπολογίσουμε τον μέγιστο και τον ελάχιστο αριθμό μίας λίστας. Επίσης χρησιμοποιούμε δύο ακόμη συναρτήσεις για να πάρουμε το άθροισμα και τη μέση τιμή όλων των αριθμών της λίστας.

Ο κώδικάς μας

```
List<Integer> primes = Arrays.asList(4, 6, 7, 9, 13, 15, 19, 20, 24, 28);
IntSummaryStatistics stats = primes.stream().mapToInt((x) ->
x).summaryStatistics();
System.out.println("Ο μεγαλύτερος αριθμός στη λίστα : " + stats.getMax());
System.out.println("Ο μικρότερος αριθμός στη λίστα : " + stats.getMin());
System.out.println("Άθροισμα όλων των αριθμών : " + stats.getSum());
System.out.println("Μέση τιμή των αριθμών : " + stats.getAverage());
```

Η έξοδος του προγράμματος

```
Ο μεγαλύτερος αριθμός στη λίστα : 28
Ο μικρότερος αριθμός στη λίστα : 4
Άθροισμα όλων των αριθμών : 145
Μέση τιμή των αριθμών : 14.5
```

Βιβλιογραφία

[1] M. F. Raoul-Gabriel Urma, and Alan Mycroft, Java 8 in Action: Manning publications, 2014.

[2] K. Sharan, Beginning Java 8 Fundamentals: Apress, 2014.

[3] K. Sharan, Beginning Java 8 Language Features

Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams: Apress, 2014.

[4] C. S. Horstmann, Java SE 8 for the Really Impatient: Pearson Education, 2014.

[5] R. Warburton, Java 8 Lambdas: O'Reilly, 2014.

[6] R. Fischer, Java Closures and Lambda: Apress, 2014.