**תכנות פונקציונלי בשפת ג'אווה**

החל מג'אווה 1.8.0 התווספו פיטצ'רים חדשים,

המאפשרים כתיבה בסגנון פונקציונאלי.

במסמך זה, אסביר מה זה תכנות פונקציונלי,

מה היתרונות של תכנות פונקציונלי

וכמובן כיצד כותבים בסגנון פונקציונאלי בשפת ג'אווה.

**הגדרה:**

תכנות פונקציונלי היא פרדיגמת תכנות אשר מתבססת על קומפוזיציית פונקציות ועל שימושן.

הסגנון מתאפיין בכתיבה דקלרטיבית, כאשר הפונקציות הן טהורות, נקיות מ side effects , מחזירות ערך ואומרות מה הפעולות שצריכות להתבצע.

בניגוד לכתיבה אימפרטיבית, המתאפיינת באוסף של הצהרות

האומרות איך לבצע כל פעולה.

**דקלרטיבי מול אימפרטיבי**

כתיבה אימפרטיבית, לרוב אינה מודולרית ומציינת בכל רגע נתון איך לבצע משימה.

סגנון זה מתאפיין בהתניות, לולאות ופעולות המבצעות מוטציה במשתנים קיימים.

סגנון זה מכריח את הקורא, לחשב מה תהיה התוצאה, אולי אפילו לדבאג רק על מנת להבין מה הקוד עושה.

ובנוסף, כאשר אנחנו משנים משתנים קיימים במקום לצור חדשים על בסיס הישנים, אנחנו חשופים ליותר שגיאות ככל שהתוכנית גדלה.

דוגמא:

List<String> friends = new ArrayList<>();  
friends.add("Efrat");  
friends.add("Yaron");  
friends.add("Itamar");  
  
  
for (int i = 0; i < friends.size(); i++) {  
 if (friends.get(i).charAt(0) == 'E' ||   
 friends.get(i).charAt(0) == 'e') {  
 friends.remove(friends.get(i));  
 }  
 friends.set(i, friends.get(i) + " Hello!");  
}

בדוגמא זאת, הקורא צריך לחשב מה קורה בכל איטרציה, ובנוסף גם מתבצעת מוטציה על הדאטא, מה שיכול לגרום לבאגים במידה הקוד גדל.

בניגוד לדוגמא הקודמת, אנחנו יכולים לכתוב באופן דקלרטיבי ולהגיד מה אנחנו רוצים שיקרה ללא שינוי של הדאטא.

דוגמא:

List<String> friends = Stream

.*of*("Efrat", "Yaron", "Itamar")  
 .filter(*not*(*isStartsWithE*)::apply)  
 .map(*add*(" Hello"))  
 .collect(Collectors.*toList*());

בדוגמא זאת, אנחנו כותבים מה אנחנו רוצים שיקרה,

ולא איך לבצע את המשימה.

אנחנו עושים את זה על ידי שימוש בפונקציות, והכי חשוב אנחנו לא מבצעים מוטציה על אוסף קיים.

**פונקציה טהורה**

פונקציה טהורה, היא נקייה מside effects ובכך אינה משפיעה על משתנים מחוץ ל scope שלה.

אחד הכלים למדוד פונקציה כטהורה, אם בהנתן אותו הinput נקבל את אותו ה output

דוגמא לפונקציה לא טהורה:

private static int *num* = 0;  
  
public static void main(String[] args) {  
 System.*out*.println(*addToNum*(2));  
 System.*out*.println(*addToNum*(2));  
 System.*out*.println(*addToNum*(2));  
}  
  
public static int addToNum(int num2){  
 return ++*num* + num2;  
}

ה input בדוגמא הזאת הוא 2 וה output שנקבל יהיה:

3 , 4 , 5

זאת אומרת שאנחנו קוראים לאותה הפונקציה כמה פעמים ומקבלים תוצאה שונה.

מעבר לזה שקשה לנו לצפות מה הפונקציה תחזיר, אנחנו משפיעים על המשתנה num וכשהאפליקציה תגדל יהיה לנו קשה לדעת מה הערך ש num מחזיק ללא דיבאג.

דוגמא פשוטה לפונקציה טהורה:

public static void main(String[] args) {  
 System.*out*.println(*addToNum*(2, 3));  
 System.*out*.println(*addToNum*(2, 3));  
 System.*out*.println(*addToNum*(2, 3));  
}  
  
public static int addToNum(int num1, int num2){  
 return num1 + num2;  
}

אומנם פה אין שימוש בפיטצ'רים של ג"אווה 1.8.0 ומעלה,

אבל עדיין הפונקציה מקבלת ארגומנטים מבחוץ, אינה משפיעה על משתנים מחוץ ל scope שלה ובהינתן אותו ה input היא תחזיר את אותו ה output ובכך קל לנו לצפות מה הפונקציה תחזיר.

עכשיו כשאנחנו רואים את ההבדל בין כתיבה אימפרטיבית ודקלרטיבית ומהיא פונקציה טהורה בואו נלמד על ביטוי למדה, ואינטרפייס פונקציונלי.

**למדה ואינטרפייס פונקציונלי**

אינטרפייס פונקציונלי, הוא אינטרפייס המורכב ממתודה לא ממומשת אחת.

דוגמא:

@FunctionalInterface  
interface OneMethod {  
  
 void run();  
}

הסיבה שהוא נחשב פונקציונלי ושיש בו רק מתודה אחת,

היא שאנחנו יכולים לממש רק את אותה הפונקציה ללא יצירת קלאס המממש את אותו האינטרפייס.

ובעצם לממש רק את אותה הפונקציה על ידי שימוש בביטוי למדה.

דוגמא:

OneMethod oneMethod = () ->

System.*out*.println("Hello :)");

בסוגריים אנחנו מציינים אלו ארגומנטים אנחנו מקבלים,

ולאחר החץ אנחנו כותבים את המימוש שבסופו מחזיר את מה שהתחייבנו באינטריפייס.

לאחר מכן אנחנו יכולים לקרוא לאותה המתודה הממומשת.

oneMethod.run();

הנה כמה דוגמאות לאינטרפייסים פונקציונליים ולמימושם

מקבל ארגומנט ומחזיר ארגומנט:

@FunctionalInterface  
interface OneMethod<T, R>{  
  
 R apply(T t);  
}

OneMethod<String, Integer> oneMethod = str -> str.length();

מקבל שני ארגומנטים ומחשב תוצאה בתוך scope:

@FunctionalInterface  
interface OneMethod<T1, T2, R>{  
  
 R apply(T1 t1, T2 t2);  
}

OneMethod<Integer, Integer, Integer> add = (num1, num2) -> {  
 return num1 + num2;  
};

עכשיו שאנחנו יודעים כיצד לממש אינטרפייס פונקציונלי,

בואו נכיר את האינטרפייסים הקיימים כבר בג'אווה

**Predicate<T>**

Predicate<T> מקבל משתנה מכל סוג ומחזיר לנו בוליאן

דוגמא:

Predicate<String> isStartsWithE = str ->   
 str.trim().toLowerCase().startsWith("e");

המתודה שאנחנו מממשים נקראת test:

boolean isIt = isStartsWithE.test("Efraim");

ניתן להשתמש גם ב negate אשר יחזיר לנו את הבוליאן ההפוך מהתוצאה המקורית:

boolean isIt = isStartsWithE.negate().test("Effy");

וניתן גם לשרשר and ו or בשביל לבדוק יותר מתנאי אחד.

boolean isIt = isStartsWithE  
 .and(name -> name.endsWith("y"))  
 .test("efraim");

הסיבה שאנחנו משתמשים ב and ישר אחרי שם ה predicate

זה מכיוון ש and זאת מתודה ממומשת ב predicate ואנחנו פונים אליה.

אם היינו מנסים להשתמש ב and בסוף, לא היינו יכולים כי מתודת ה test מחזירה בוליאן – תוצאה , וכידוע לבלויאן אין מתודות להציע לנו.

איך ששמתם לב, isStartsWithE נורא ספציפית ומתאימה רק לאות E

פה אנחנו יכולים להשתמש ב קומפוזיציה של פונקציות על מנת לקבל פונקציה יותר גנרית.

מכיוון שאנחנו יכולים להשתמש באינטרפייס כפונקציה, על ידי זה שנממש אותו באמצעות ביטוי למדה, אנחנו יכולים להכריז שמתודה מחזירה פונקציה, מקבלת פונקציה ואפילו מגדירה פונקציה בתוכה.

זה מאפשר לנו בעצם ליצור סוג של קונסטרקטור לפונקציות, ובכך לפרק את הפונקציה שלנו לכמה שלבים.

קומפזיצה של פונקציה נעשת על ידי פונקציות שמחזירות פונקציות

לדוגמא:

private static Predicate<String> isStartsWith(String letter) {  
 return str -> str.startsWith(letter);  
}

boolean isIt = *isStartsWith*("e").test("efraim");

אני יכול כמו בדוגמא לפרק את הלוגיקה שלי לשני שלבים.

הראשון הוא לקבוע איזה Predicate יחזור אליי על ידי מתן האות “e” והשלב השני, לקרוא לפונקציה עם הערך “efraim”

כמובן, אני יכול גם לשמור את הPredicate שחזר לתוך משתנה חדש.

דוגמא:

Predicate<String> isStartsWithE = *isStartsWith*("e");

**BiPredicate<T, U>**

BiPredicate למעשה הוא כמו Predicate רק שהוא מקבל שני ארגומנטיים אשר יהיו זמינים לנו בעת המימוש

דוגמא:

BiPredicate<String, Integer> isLengthGreaterThan = (str, max) -> str.length() > max;

**Consumer<T>**

Consumer<T> לא נחשב לפונקציונלי, מכיוון שאינו מחזיר ערך,

אלא רק צורך משתנה ומבצע side effect על אותו המשתנה.

אנחנו כן יכולים לממש אותו על ידי ביטוי למדה, אך הוא אינו פונקציונלי עבורינו מכיוון שאינו מחזיר ערך.

Consumer<String> printWithHello = str -> System.*out*.println(str + " Hello");

למתודה הממומשת קוראים accept

printWithHello.accept("Avner");

לאותו המשתנה ‘Avner’ אנחנו יכולים לשרשר כמה Consumers

על ידי שימוש במתודה הממומשת andThen

המתודה מקבלת Consumer נוסף שאליו יכנס המשתנה לאחר הקריאה ל Consumer הראשון

דוגמא:

printWithHello

.andThen(name -> System.*out*.println(name + " Bye"))  
.accept("Avner");

בדוגמא זאת, אנחנו עושים שימוש ביטוי למדה.

andThen מקבל Consumer ולכן אנחנו לא חייבים לממש אותו במקום, אנחנו יכולים ליצור Consumer נוסף על ידי שימוש בקומפוזיציה ולהשתמש בו במקום.

דוגמא:

private static Consumer<String> printWith(String toAdd) {  
 return str -> System.*out*.println(str + " " + toAdd);  
}

*printWith*("Hello")  
 .andThen(*printWith*("How are you?"))  
 .andThen(*printWith*("Bye..."))  
 .accept("Avner");

הסטייל הזה נקרא Point Free ומשפר את הקריאות של הקוד.

**BiConsumer<T, U>**

BiConsumer הוא אותו דבר כמו ה Consumer רק שהוא מקבל שני ארגומנטים.

דוגמא:

BiConsumer<String, String> printTwoFriends =

(friend1, friend2) ->

System.*out*.println(friend1 + ", " + friend2);

**Function<T, R>**

Function זאת פונקציה שמקבלת ארוגמנט אחד ומחזירה תוצאה

דוגמא:

Function<String, String> greeter = name ->

"Hello " + name;

למתודה הממומשת קוראים apply

דוגמא:

String greetAvi = greeter.apply("Avi");

הדוגמא הזאת אומנם גנרית, אבל הברכה אינה גנרית.

אנחנו יכולים ,’לזכור’ את אחד המשתנים ( הברכה או השם )

על ידי שימוש ב partial application

ולעשות closure על אחד מהמשתנים

דוגמא:

private static Function<String, String> greeter(String name) {  
 return text -> text + " " + name;  
}

Function<String, String> greetAvi = *greeter*("Avi");  
greetAvi.apply("Hello!");

או על ידי שימוש בcurrying

דוגמא:

System.*out*.println(  
 *greeter*("Avi").apply("Hello")  
);

ל Function גם יש את הפונקציה andThen שמאפשרת לי לשרשר פונקציות.

הפונקציה מקבלת את מה שחזר מתוך קריאה ל apply

ומעבירה אות התוצאה לפונקציה חדשה

דוגמא:

System.*out*.println(  
 *greeter*("Avi")  
 .andThen(greeting -> greeting + ", How are you?")  
 .andThen(greeting -> greeting + " Hope its all good :)")  
 .apply("Hello")  
);

או בסגנון ‘point free’ :

private static <T> Function<T, String> add(String text) {  
 return t -> t + text;  
}

System.*out*.println(  
 *greeter*("Avi")  
 .andThen(*add*(" How are you?"))  
 .andThen(*add*(" Hope its all good :)"))  
 .apply("Hello")  
);

תוצאה:

Hello Avi How are you? Hope its all good :)

או על ידי שימוש ב compose

System.*out*.println(  
 *greeter*("Avi")  
 .compose(*add*(" Hope its all good :)"))  
 .compose(*add*(" How are you?"))  
 .apply("Hello")  
);

שפה הסדר הפוך, אבל התוצאה תהיה זהה.

**BiFunction<T, U, R>**

BiFunction בדיוק כמו Function רק שמקבל שני ארגומנטים.

דוגמא:

BiFunction<Integer, Integer, Integer> add = (num1, num2) ->   
 num1 + num2;

גם פה אנחנו יכולים להשתמש ב partial application או ב currying

דוגמא:

private static Function<Integer, Integer> add(int num1) {  
 return num2 -> num1 + num2;  
}

Partital application:

Function<Integer, Integer> add9 = *add*(9);  
int ten = add9.apply(1);

Currying:

int ten = *add*(9).apply(1);

Supplier<T>

Supplier כמו שהוא נשמע, תפקידו לספק לנו תוצאה של חישוב מסוים.

הוא אינו מקבל שום ארגומנט, וקל לחשוב בגלל זה שהוא אינו פונקציונלי.

אבל הוא אכן כן. מכיוון שהוא עושה lazy evaluation

דוגמא:

Supplier<Integer> result = () -> 10 \* 20 \* 30 \* 100;  
int evaluatingNow = result.get();

דוגמא מקוד אמיתי:

@Override  
public Category getOrCreate(CategoryType type) {  
 return findByType(type)

.orElseGet(() -> repository.save(new Category(type)));  
}

בקוד למעלה, מכיוון ש orElseGet מקבל Supplier אז הוא לא יקרא לפונקציית ה get אם יהיה קיים שם ערך, מה שאומר ש repository save לא יחושב, רק במידה ובאמת לא נמצא את ה category לפי ה ID

**Runnable**

Runnable הוא האחרון והכי פחות פונקציונלי.

מכיוון שאינו מקבל שום דבר, ואינו מחזיר שום דבר.

ולמעשה, עוסק רק ב side effect.

דוגמא:

Runnable printHi = () -> System.*out*.println("Hi!");  
printHi.run();

עכשיו כשהכרנו את האינטרפייסים הפונקציונלים אפשר לעבור לנושא הבא, מונדות.

**מונדות**

מונדה היא אובייקט העוטף אובייקט\ים אחרים,

והיא מתפקדת כפונקצייה טהורה הנקייה מ side effect

והיא מממשת map ו flatMap ככלי לטרנספורמציה של דאטא.

היא קושרת פונקציות אשר מחזירות מופע של אותה המונדה רק עם ערך חדש.

דוגמא:

class Monad<T> {  
   
 private final T t;  
   
 private Monad(T t) {  
 this.t = t;  
 }  
   
 public static <T> Monad<T> of(T t) {  
 return new Monad<>(Objects.*requireNonNull*(t));  
 }  
}

מימוש לmap :

public <R> Monad<R> map(Function<T, R> mapper) {  
 return Monad.*of*(mapper.apply(t));

ומימוש ל get :

public T get() throws NoSuchMethodException {  
 if (t == null) {  
 throw new NoSuchMethodException(

"Element is equals to null"

);  
 }  
 return t;  
}

מימשו ל orElse :

public T orElse(T other) {  
 return t == null ? other : t;  
}

מימוש ל orElseGet

public T orElseGet(Supplier<T> supplier) {  
 return t == null ? supplier.get() : t;  
}

מימוש ל orElseThrow

public <X extends Throwable> T orElseThrow(Supplier<? extends X> exception) throws X {  
 if (t != null) {  
 return t;  
 } else {  
 throw exception.get();  
 }  
}

עכשיו אני יכול להשתמש במונדה על מנת לעשות טרנספורמציה של דאטא ללא side effects

int myLength = Monad.*of*("Enosh")  
 .map(String::length)  
 .orElse(0);

או על ידי שימוש באחד מהבאים:

int myLength = Monad.*of*("Enosh")  
 .map(String::length)  
 .orElseGet(() -> 0);

try {  
 int myLength = Monad.*of*("Enosh")  
 .map(String::length)  
 .orElseThrow(() -> new Exception("Exception message"));  
   
} catch (Exception e) {  
 e.printStackTrace();  
}

**Optional<T>**

Optional הוא מונדה, והוא נועד בעיקרו להגן עלינו מפני null poiner exception

אנחנו בדרך כלל נשתמש בו כאשר אנחנו מצפים לקבל משתנה היכול להיות null

יש לו מתודת map המאפשרת לנו לעשות טרנספורמציה על המשתנה שהוא nullable , כמובן ללא שינוי המשתנה עצמו, אלא על ידי קבלת משתנה חדש.

יש לו מתודה flatMap אשר מחזירה לנו Optional חדש עם ערך חדש עטוף בOptional למקרה, שמשתנה בתוך האובייקט גם הוא יכול להיות null

יש לו עוד כל מיני מתודות כמו ifPresent שמקבלת Consumer ומאפשרת לנו לבצע side effect על אותו המשתנה שיכול להיות null

וisPresent המחזירה לנו בוליאן האם המשתנה null או לא.

בואו נראה את השימוש האימפרטיבי ב Optional ולאחר מכן, את השימוש הדקלרטיבי והיותר פונקציונלי.

דוגמא:

Optional<String> maybeThere = Optional.*ofNullable*(stringFromDataBase);  
if (maybeThere.isPresent()) {  
 String withHello = maybeThere.get() + " Hello";  
 System.*out*.println(withHello);  
}

בדוגמא זאת, אנחנו עושים שימוש אימפרטיבי בOptional

מתודת הget תזרוק לנו NoSuchElementException במידה והערך בתוך ה Optional יהיה null

אחד היתרונות בOptional שהקורא יודע שהמשתנה העטוף יכול להיות null ובכך מכריח אותו להתמודד עם nullability

דוגמא נוספת, פחות אימפרטיבית אבל עדיין לא פונקציונלית:

Optional<String> maybeThere = Optional.*ofNullable*(stringFromDataBase);  
maybeThere.ifPresent(maybe -> {  
 String withHello = maybe + " Hello";  
 System.*out*.println(withHello);  
});

בדוגמא זאת, ifPresent מקבל Consumer של String ( האובייקט העטוף ) ומאפשר לנו לבצע side effects על אותו המשתנה.

החסרון, שלא חוזר לנו שום ערך מתוך הפונקציה.

בדוגמא הבאה נראה שימוש פונקציונלי ב Optional

String withHello = Optional.*ofNullable*(stringFromDataBase)  
 .map(*add*("Hello"))  
 .orElse("No value inside");

בדוגמא זאת הערך הנמצא בorElse מחושב.

אם נרצה לבצע lazy evaluation, נשתמש בorElseGet

String withHello = Optional.*ofNullable*(stringFromDataBase)  
 .map(*add*("Hello"))  
 .orElseGet(() -> "No value inside");

ובמידה ונרצה לזרוק exception נוכל להשתמש בorElseThrow

try {  
   
 String withHello = Optional.*ofNullable*(stringFromDataBase)  
 .map(*add*("Hello"))  
 .orElseThrow(() -> new Exception("No value inside"));  
   
} catch (Exception e) {  
 e.printStackTrace();  
}

שימוש ב map מאפשר לנו להחזיר ערך חדש שיכול להיות גם מסוג אחר, דוגמא:

int length = Optional.*ofNullable*(stringFromDataBase)  
 .map(String::length)  
 .orElse(0);

לפעמים נראה לבצע פעולות על אובייקט שיכול להיות null שנמצא בתוך אובייקט אחר שגם הוא יכול להיות null

במצב הזה, נרצה להשתמש ב flatMap

דוגמא:

User maybeUser = new User(new Address(null));  
  
int cityLength = Optional.*ofNullable*(maybeUser)  
 .flatMap(user -> Optional.*ofNullable*(user.getAddress()))  
 .flatMap(address -> Optional.*ofNullable*(address.getCity()))  
 .map(String::length)  
 .orElse(0);

גם את הדוגמא הזאת אנחנו יכולים להחליף בסגנון ‘point free’

Function<User, Optional<Address>> getAddressAsOptional = user ->  
 Optional.*ofNullable*(user.getAddress());  
  
Function<Address, Optional<String>> getCityAsOptional = address ->  
 Optional.*ofNullable*(address.getCity());

int cityLength = Optional.*ofNullable*(maybeUser)  
 .flatMap(getAddressAsOptional)  
 .flatMap(getCityAsOptional)  
 .map(String::length)

לפעמים לא נרצה לבדוק nullability אלא פשוט לפלטר משתנה מסוים.

זאת אומרת, אם משתנה לא עונה לקריטריונים שלנו נרצה לתת לו חלופה אחרת.

במקרה הזה אנחנו גם יכולים להשתמש ב Optional אבל במקום ofNullable אנחנו יכולים להשתמש ב of .

במקרה הזה, אסור שהמשתנה יהיה null כי אז ישר יזרק לנו NullPointerException

דוגמא:

String maybeEmpty = "";  
String value = Optional.*of*(maybeEmpty)  
 .filter(*not*(String::isEmpty)::apply)  
 .map(String::toUpperCase)  
 .orElse("I should not be empty");

**Stream<T>**

Stream הוא גם מונדה היכול להכיל אוסף של משתנים.

אנחנו משתמשים בו כאשר אנחנו רוצים ליצור דאטא חדש מתוך אוסף של משתנים.

הוא גם מממש map flatMap ו filter ויש לו עוד אוסף רחב של פונקציות המאפשרות לנו ליצור דאטא חדש מתוך אוסף קיים.

דוגמא:

String greetFriends = Stream.*of*("Elly", "Omer", "Lior")  
 .map(*add*("Hello"))  
 .collect(Collectors.*joining*(", "));

הנה דוגמא לfilter ול findAny שמחזיר Optional

String greetFriend = Stream.*of*("Elly", "Omer", "Lior")  
 .filter(*isStartsWith*("o"))  
 .map(*add*("Hello"))  
 .findAny()  
 .orElse("Hello nobody :(");

אנחנו יכולים גם ליצור Stream של פונקציות

List<Integer> results = Stream.*of*(*add*(3), *add*(5), *add*(10))  
 .map(fn -> fn.apply(5))  
 .collect(Collectors.*toList*());

או להשתמש בreduce בשביל לאחד את התוצאות

int results = Stream.*of*(*add*(3), *add*(5), *add*(10))  
 .map(fn -> fn.apply(5))  
 .reduce(Integer::*sum*)  
 .orElse(0);

או למשל דוגמא עם greeter

String greeting = Stream.*of*(  
 *greeter*("Avi"),   
 *greeter*("Liat"),   
 *greeter*("Efrat"))  
 .map(greet -> greet.apply("Hello!"))  
 .collect(Collectors.*joining*(", "));

אנחנו יכולים גם להשתמש בflatMap בשביל לשטח אוסף של אוספים, למשל כאשר יש לנו מבנה נתונים כזה:

[[“Avi”, “Efrat”], [“Elly”, “Yaron”]]

ואנחנו רוצים לקבל מבנה נתונים כזה:

[“Avi”, “Efrat”, “Elly”, “Yaron”]

ולבצע עליו map

דוגמא:

String[] friends1 = { "Avi", "Ofer", "Amos"};  
String[] friends2 = { "Guy", "Chen", "Sapir", "Victor"};  
  
String greeting = Stream.*of*(friends1, friends2)  
 .flatMap(friend -> Stream.*of*(friend).map(y -> y + " Hello"))  
 .collect(Collectors.*joining*(", "));

או על ידי שימוש בסגנון ‘point free’

דוגמא:

Function<String[], Stream<String>> addHello = strArray ->  
 Stream.*of*(strArray).map(*add*(" Hello"));  
  
String[] friends1 = { "Avi", "Ofer", "Amos"};  
String[] friends2 = { "Guy", "Chen", "Sapir", "Victor"};  
  
String greeting = Stream.*of*(friends1, friends2)  
 .flatMap(addHello)

.sorted()  
 .collect(Collectors.*joining*(", "));

בדוגמא זאת גם השתמשתי בsorted בשביל למיין את הסטרינגים.

**Method Reference**

לפעמים נרצה להעביר משתנה ( למדה ) לתוך מתודה המצפה לקבל אותו, ומכיוון שזה המשתנה היחידי שאמור להכנס לתוך המתודה אנחנו יכול לספק רפרנס לאותה המתודה.

המתודה:

private String sayBye(String name) {  
 return name + " bye!";  
}

רפרנס דרך המופע:

Main main = new Main();  
  
String byeTo = Stream.*of*("Yuval", "Ella")  
 .map(main::sayBye)  
 .collect(Collectors.*joining*(", "));

רפרנס סטטי:

private static String sayBye(String name) {  
 return name + " bye!";  
}

String byeTo = Stream.*of*("Yuval", "Ella")  
 .map(Main::*sayBye*)  
 .collect(Collectors.*joining*(", "));

**Vavr – External Library**

Vavr היא ספרייה המיועדת לתכנות פונקציונלי, מתבססת על גרסת ג'אווה 1.8.0 ומעלה.

היא מכילה בתוכה immutable collections, מונדות ומגוון רחב של פונקציות.

השם במקור היה ‘java slang’ והתפתח להיות vavr כשנקרע חצי מהלוגו – java slang ונשאר רק הJAVA מכיוון שהלוג הוצג הפוך, משם בא השם vavr - JAVA הפוך.

**Pattern Matching ( Match )**

Match מאפשר לנו לבצע השמה למשתנה על ידי בחינת משתנה אחר.

במידה ואחד הדפוסים נמצא מתאים, הערך שמקושר אליו יחזור.

למעשה, Match מזכיר switch רק שהוא expression במקום statement.

בנוסף, יש לו כל מיני פונקציות פנימיות שמקלות עלינו לבצע pattern matching, וגם הוא מאפשר לנו להכניס פונקציות כקריטריון ל Match

דוגמא:

String result = *Match*(friend).of(  
 *Case*(*$*("Yoni"), "Very surprising..."),  
 *Case*(*$*("Efrat"), "I dont think so..."),  
 *Case*(*$*("Avi"), "Now we are talking!"),  
 *Case*(*$*(), "I think it was Avi...")  
);

כמו שניתן לראות, כל אחד מהקייסים מחזיר ערך אחר, והקייס האחרון הוא בעצם ה default

נוכל להשתמש גם בפונקציות built in שמאפשרות לנו לבצע כל מיני Match יותר ספציפיים

דוגמא:

String result = *Match*(friend).of(  
 *Case*(*$*(*isIn*("Effy", "Efraim")), "Very surprising..."),  
 *Case*(*$*(*isNull*()), "Why null?"),  
 *Case*(*$*(*isStartsWith*("a")), "Now we are talking!"),  
 *Case*(*$*(String::isEmpty), "Empty"),  
 *Case*(*$*(fr -> fr.length() > 7), "Greater than 7"),  
 *Case*(*$*(), "I think it was Avi...")  
);

**Try**

Try היא למעשה מונדה העושה encapsulation לפעולת ה try וה catch

היא מממשת map ועוד מגוון רחב של פונקציות.

דוגמא:

Try<Integer> devideByZero = Try.*of*(() -> 4 / 0);  
int result = devideByZero.getOrElse(0);

בנוסף, קורה פה lazy evaluation זאת אומרת, עד שאני לא מבקש את הresult גוף ה Try לא מחושב.

לTry גם יש פונקציות המאפשרות לבצע side effects הנותן מענה לכשלון והצלחה, למשל כמו כתיבה ללוג.

Try<Integer> devideByZero = Try.*of*(() -> 4 / 0);

devideByZero.onFailure(throwable ->

*log*.error("Error: {}", throwable.getMessage()));

devideByZero.onSuccess(result ->

*log*.info("Success: {}", result));

אפשר גם לתאם exception עם Pattern Matching

באמצעות המתודה recover

דוגמא:

Try<String> makeFullName = Try.*of*(() -> fname + lname);

String result = makeFullName.recover(throwable ->

*Match*(throwable).of(  
 *Case*(  
 *$*(*instanceOf*(NullPointerException.class)),  
 "Null Pointer"  
 ),  
 *Case*(  
 *$*(*instanceOf*(IllegalStateException.class)),  
 "Illegal State"  
 )  
 )).getOrElse("John Doe");

אפשר גם לשרשר Try במידה והנסיון הראשון לא עבד.

דוגמא:

String fullName = Try.*of*(() -> fname + lname)  
 .orElse(Try.*of*(() -> fname + " Doe"))  
 .getOrElse("John Doe");

אפשר גם להשתמש בmap על מנת ליצור ערך חדש מתוך ההצלחה של Try

דוגמא:

String greeting = Try.*of*(() -> fname + " " + lname)  
 .map(*add*(" Hello"))  
 .getOrElse(() -> "Hello nobody :(");

יש עוד מגוון פונקציות עבור side effects כמו andThen

Try.*of*(() -> fname + " " + lname)  
 .map(*add*(" Hello"))  
 .andThen(() ->

System.*out*.println("Im a side effect!"));

ו finally

Try.*of*(() -> fname + " " + lname)  
 .map(*add*(" Hello"))  
 .andFinally(() ->   
 System.*out*.println("Side effect executes as finally")  
 );