**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 שימוש בהן.

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.

עכשיו כשאנחנו רואים את ההבדל בין כתיבה אימפרטיבית ודקלרטיבית, 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 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 לאינטרפייס ולמימוש

מקבל ארגומנט ומחזיר ארגומנט:

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

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

מקבל כמה 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.

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");

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 אחר\ים,

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("Why is it null?"));  
   
} catch (Exception e) {  
 e.printStackTrace();  
}