Java Funkcionális interfész & Lambda

<< Java

Contents

Funkcionális Interfész alapok

Mire jó a funkcionális interfész

A funkcionális interfészek használatával megszabadulhatunk az úgynevezett boilerplate (közhelyes) code használatától, vagyis nagyon egyszer?en, osztály példányosítás nélkül tudunk interfész metódus implementációkat készíteni. Ennek az egyik kulcs eszköze a JAVA8-ban bejött LAMBDA és úgynevezett metódus referencia, amire majd látunk b?ven példát. Azt mondják hogy ez a java-ba valaha behozott legnagyobb újítás.

A fent leírtakat persze persze anonymous osztályokkal is el lehet érni, csak többet kell hozzá írni. Pl: A Theread osztályban szokásos anonymous osztályokat létrehozni a run() definiálására:

```
public class AnonymousClassExample {

    public static void main(String[] args) {

        Thread thread = new Thread(){
            public void run(){
                System.out.println("Understanding what is an anonymous class");
            }
        };
        thread.start();
    }
}
```

Lambda kifejezéssel ugyan ez így írható le:

```
public class AnonymousClassExample {

    public static void main(String[] args) {

        Runnable runnable = () -> {
            System.out.println("Understanding functional interfaces");
        };
        runnable.run();
    }
}
```

Hogyan m?ködik

A funkcionális interfész egy olyan interfész, amiben csak egy darab absztrakt metódus van. (vagyis ami implementálandó). Ezt kidomborítandó, a **@FunctionalInterface** annotációval is el lehet látni az interfészt.

Készítsünk egy egyedi String osztályt, amivel majd példálózunk (ez még nem kapcsolódik a funkcionális if-re)

```
public class MyString {

    private String input;

    public MyString(String input) {
        System.out.println("MyString consturctor is called with input:" + input);
        this.input = input;
    }

    public String getInput() {
```

```
        return input;
    }
}
```

Tekintsük az alábbi funkcionális if-et, aminek egy absztrakt metódusa van, ami egy String-et vár, és egy MyString-et ad vissza.

```
@FunctionalInterface
public interface MyFunction {

    MyString process(String input);
}
```

Egy funkcionális interfész implementációjának a megadására az alábbi 5 lehet?ségünk van:

- Hagyományos implementációs osztály példányosítása
- anonymous osztály használata
- Lambda kifejezés használata
- Metódus referencia használata: lásd #Java_Method_Reference_and_Constructor_Reference
- Konstruktor referencia használata lásd #Java_Method_Reference_and_Constructor_Reference

Implementációs osztály példányosítása

A klasszikus használata egy interfésznek, ha készítünk egy osztályt, ami implementálja az absztrakt metódusokat, majd példányosítjuk az osztályt, és klasszikus módon használjuk. Ehhez nincs szükség funkcionális IF-re, vagyis hogy csak egy absztrakt metódusa legyen az interfésznek, mivel az implementációban explicit megmondjuk, hogy melyiket implementáljuk:

```
public class MyFunctionImpl implements MyFunction {
    @Override
    public MyString process(String input) {
        return new MyString(input);
    }
}
```

Majd ezt így használhattuk:

```
MyFunctionImpl2 myFunctionImpl2 = new MyFunctionImpl2();
        MyString myString = myFunctionImpl2.process("MyInput");
        System.out.println(myString.getInput());
```

Anonymous osztály használata

Ahogy a bevezet?ben is láttuk, bármilyen interfésznek lehet anonymous osztállyal definiálni a még nem definiált interfészeit az alábbi szintaxissal:

```
new InterfaceName() {
     @Override
     public <implementálandó metódus definíció> {..}
};
```

Itt sincs szükség funkcionális interfészre, vagyis hogy csak egy absztrakt metódusa legyen az interfésznek, mert explicit megmondjuk hogy melyiket implementáljuk.
Gyakorlatban ez így néz ki az el?bbi példára:

```
MyFunction fm2 = new MyFunction() {
            @Override
            public MyString process(String input) {
                return new MyString(input);
            }
        };

        fm2.process("input");
```

Lambada kifejezés használata

A fenti anonymous osztály definíció egy sokkal rövidebb Lambda kifejezéssel kiváltható, de csak akkor ha a szóban forgó interfész egy **funkcionális interfész**, vagyis pontosan egy darab absztrakt metódusa van. A Lambda kifejezés szintaxisa az alábbi:

```
(arguments) -> {function body}
```

Ahol az argumentumok az egy szem absztrakt metódus input paraméterei, a function bdoy-ban pedig leírjuk, hogy mit tegyen ezekkel az input paraméterekkel az implementáció. Az argumentumok típus nélküliek, a típusuk implicit következik a felhasználási módból, lásd lentebb.

- A paraméterek neve tetsz?leges lehet, nem kell megegyezzen a funkcionális metódus paraméter neveivel. A sorrend viszont számít. A lényeg, hogy a body-ban az argumentumokban felsorolt nevekkel tudunk hivatkozni az input paraméterekre.
- Ha az interfész input paraméter generikus, csak akkor fognak típust kapni, amikor a generikus értékeket definiáljuk az interfész változó létrehozásánál.
- Ha a visszatérési érték is generikus az implementálandó metódusban, akkor annak a típusa is csak az adott felhasználásból fog következni.
- Ha a funkcionális interfész nem generikus, akkor az interfész input és return értékéb?l már explicit következik a lambdában használt paraméterek és visszatérési értékék típusa.

Nézzünk egy konkrét példát. A 'MyFunction' egy funkcionális interfész, mert pontosan egy absztrakt metódusa van: 'MyString process(String)'. Tehát a lambda kifejezéssel ennek a 'process(..)' absztrakt metódusnak az implementációját kell definiálni:

```
MyFunction fm = var1 -> {
                System.out.println("inside lambda body");
                return new MyString(var1);
        };
        System.out.println("Start");
        fm.process("input");
        System.out.println("End");
```

Láthatjuk, hogy a 'MyFunction' egy szem absztrakt metódusát egy lambda kifejezéssel definiáltuk, amivel egyben el is készítettük a MyFunction implementációjának példányosítását. A 'MyFunction'-enk egy darab absztrakt metódusa van, az alábbi szignatúrával: 'MyString process(String)', ebb?l következik, hogy a '->' elé pontosan egy változó nevet kell kitaláljunk, ami bármi lehet, lényeg, hogy a body-ban ezzel a névvel tudunk rá hivatkozni. Mi itt 'var1' nevet találtunk ki, aminek a típusa 'String', mivel a 'process(..) metódusban egy darab String input paraméter van. Ha a 'process(..)' generikus inputtal rendelkezne, akkor a lambada-ban a 'var1' nek a típusa csak használat közben derülne ki, majd kés?bb erre is látunk példát.
Fontos, hogy a 'var1'-nek a globálisan semmi jelentése nincs, nem létez? változó a fenti kódot futtató kontextusban. Ezt úgy kell elképzelni, mint egy metódus leírója, ami majd akkor kerül meghívásra csak, ha meghíjuk programozottan a 'process(..)' metódust, csak akkor fog feltölt?dni értékekkel.
Az stdout-ra a következ? kerül:

```
Start
inside lambda body
End
```

Tehát az alábbi két sor egyenl?:

```
MyFunction fm = var1 -> { return new MyString(var1); };
fm.process("input");
                  EGYENL?
MyFunction fm = new MyFunctionImpl();
fm.process("input");
```

!!!Mit fontos itt látni: A lambda definiálásának a sorába még nem fut le a lambda törzsében megadott kód, vagyis ez: return new MyString(var1); !!!

Lambda generikus interfésszel

Tegyük fel, hogy adott az alábbi generikus, funkcionális interfész. A funkcionális (absztrakt) metódusa egy T-t és H-t vár és R-t gyárt bel?le. Azt hogy hogyan, és hogy mi a T, H és mi az R, az implementációra van bízva.

```
@FunctionalInterface
public interface MyFunctionGeneric<R, T, H> {
    R doIt(T var, H var2);
}
```

Definiáljuk a 'MyFunctionGeneric' interfész absztrakt metódusát egy lambda kifejezéssel. A 'MyFunctionGeneric fm' változó létrehozásakor már definiálni kell az R és T és H típusokat, ebb?l fog következni, hogy a lambada-ban használt paraméter listának milyen értékei vannak, hogy mi kell legyen a visszatérési érték.

Az alábbi példában létrehoztuk az fm3 'MyFunctionGeneric' változót, és itt meghatároztuk, hogy a visszatérési érték Integer, és a bement két String. A lambda kifejezés input paramétereiben már két értéket kell megadni, ezért zárójelet kell alkalmazni.

```
MyFunctionGeneric<Integer, String, String> fm3 = (myInput1, myInput2) -> myInput1.length();
        fm3.doIt("my input", "my input2");
```

Az input argumentumoknak megint csak bármilyen nevet megadhatunk, a fenti kódot futtató kontextusban nem léteznek, ez csak egy metódus tervrajz, nem metódus futtatás. A 'myInput1' és 2-nek a típusa következik az fm3-ban megadott generikus típusokból, vagyis mind a kett? String. A visszatérési érték pedig Integer. Ha a lambda body-ban nem akarunk több soros m?velet végrehajtani, csak egy 'return (one line logic)' sort tartalmazna, akkor a 'return' kulcsszó elhagyható. Fontos megint csak látni, hogy a doIt(..) metódus hívás hatására fog csak lefutni a lambda-ban definiált kód.

Lambda használati példák

Tipikus használati minták

..TODO: mindig egy metódus beljesében van, és a metódus lambda-t vár input paraméterben, és az osztály belsejében lév? változókkal szokott operálni. ...

List-forEach()

```
List<String> items = new ArrayList<>();
items.add("A");
items.add("B");
items.add("C");
items.add("D");
items.add("E");

//lambda
//Output : A,B,C,D,E
items.forEach(item->System.out.println(item));
```

Ez miért m?ködik?

A List a java.lang.Iterale osztály leszármazottja. Ebben van egy forEach(..) nev? metódus ami egy úgynevezett funkcionális interfész implementációt vár input paraméterben, vagyis egy **Consumer** implementációt.

```
package java.lang;

public interface Iterable<T> {
...
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
...
```

Ez a **Consumer<? super T>** generikus interfész implementációt várja paraméterként, ahol a T a List-ben lév? lista elemek típusa. A T listaelemek bármilyen leszármazottját elfogadja. Láthatjuk az implementációból hogy a forEach metódus az összes listaelemen végigmegy, és minden egyes listaelemre meghívja a Consumer<T> implementáció **accept()** metódusát. A lista elemeket a saját osztály példányának a példány változójából szedi (vagyis ha van egy ArrayList típusú objektumunk, akkor az maga tárolja a lista elemeket egy bels? változójában). A Consumer interfész implementáción múlik, hogy mit fog csinálni az adott listaelemmel.

A Consumer implementáció (Consumer<? super T> action) csak egy cs?, amibe felülr?l be tudunk dobni egy T elemet és kidob alul egy R elemet (jelen esetben nincs R elem, mert void típusú a Funkcionális függvény). Tehát az implementációnak nincs állapota. Kap egy inputot és generál bel?le egy outputot.

A Consumer interfészben az accept az egyetlen absztrakt metódus, tehát a Consumer egy funkcionális interfész, ahogy ez az annotációból is látszik:

```
package java.util.function;
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

Note
A **@FunctionalInterface** annotáció hatására a fordító ellen?rzi, hogy az adott interfész valóban megfelel e a funkcionális interfészekkel támasztott követelményeknek.

- Csak interfészre kerülhet ilyen annotáció
- Az interfésznek pontosan 1 darab absztrakt metódusa van (vagyis implementálandó)

Nézzünk egy lehetséges implementációt:

```
public class MyConstumer implements Consumer<String>{
@Override
public void accept(String t) {
System.out.println(t);
}
}
```

A MyConsumer osztály implementálja a Consumer interfész egyetlen metódusát, az accept-et, egy String-et vár inputba és nem ad vissza semmit.

Ennek az implementációs osztálynak ez lenne egy lehetséges felhasználása:

```
public static void main(String[] args) {
List<String> items = new ArrayList<>();
items.add("A");
items.add("B");
items.add("C");
items.add("D");
items.add("E");

Consumer<String> myConsumer = new MyConstumer();
items.forEach(myConsumer);
}
```

Azonban Java8-tól kezdve bevezették a ? kifejezéseket, mivel név nélküli inline függvény implementációkat készíthetünk, ezek tulajdonképpen input és output paraméterrel rendelkez? kód blokkok az alábbi szintaxissal (alább három kül. példa)

```
parameter -> expression
(parameter1, parameter2) -> expression
(parameter1, parameter2) -> { code block }
```

Warning
Ezek a paraméterek nem a "küls?", funkcionális interfészt implementáló java osztályt meghívó metódus bemeneti paraméterei, hanem bels? változok. A **forEach** esetén pl ezek a lista elemek

Az el?z? példa Lambda alakja a következ? lenne:

```
Consumer<String> myConsumer = item -> System.out.println(item);
items.forEach(myConsumer);
```

És a lambda kifejezést írhatjuk kapásból a forEach argumentumába hogy rövidítsünk:

```
items.forEach(item->System.out.println(item));
```

Note
Az item kívülr?l nézve nem értelmezhet?, szemben egy "normál" függvény hívással, ahol kintr?l adunk át neki paramétereket. Tehát ez NEM egy küls? paraméter. Értelmet csak a forEach belsejében nyer. A lambad kifejezés csak egy állapot nélküli transzformációs függvény, ami inputnak megkapja az '*item* nev? változót és csinál vele valamit:

A Lambda kifejezést kicserélhetjük method reference-re is:

```
items.forEach(System.out::print);
```

Ez egy statikus metódus referencia, amivel azt mondjuk meg, hogy a Consumer implementációja a System.out.print(T) metódust fogja meghívni. A lényeg, hogy egy argumentumos legyen a megadott metódus és fel tudjon dolgozni T lista típusú elemet.

Példa 2: saját implementáció

Az alábbi osztálynak van egy darab osztály változója (variable1) és van egy darab metódusa: processVariable. Ez a metódus egy funkcionális interfészt vár input paraméterként.

```
package hu.otp.auth.loginHelper.util;
import java.util.function.Function;

public class MyClassUsingFunctionalif {

private String variable1;

public MyClassUsingFunctionalif(String a) {
this.variable1 = a;
}

public void processVariable(Function<String, Integer> fn) {

Integer length = fn.apply(variable1);
System.out.println(length.toString());
}
}
```

A processVariable belsejében meghívja a funkcionális interfész **apply()** metódusát. (Definiálhattunk volna magunknak saját funkcionális interfész típust, mi most itt egy gyári típust használunk).

```
package java.util.function;

import java.util.Objects;

@FunctionalInterface
public interface Function<T, R> {
    R apply(T var1);   <<<<<<<<<<<----------------------------------------------------ez itt a lényeg!!!

    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
        Objects.requireNonNull(before);
        return (v) -> {
            return this.apply(before.apply(v));
        };
    }

    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (t) -> {
            return after.apply(this.apply(t));
        };
    }

    static <T> Function<T, T> identity() {
        return (t) -> {
            return t;
        };
    }
}
```

Láthatjuk, hogy egy implementálandó metódusa van a "Function" interfésznek, ami T típust vár és R típus ad vissza. Tehát ez egy generikus interfész, és bármi is legyen az implementáció, ott majd meg kell mondani, hogy a pl T=String és R=Intger. De az hogy a Function az egy generikus interfész (generic) annak semmi köze ahhoz hogy ? egy Funkcionális interfész.

Hagyományos implementáció

A Function interfésznek elkészíthetjük a saját implementációnkat is:

```
class MyFunctionImpl implements Function<String, Integer> {
        @Override
        public Integer apply(String s) {
            return Integer.valueOf(s.length());
        }
    }
```

Az el?bb láthattunk, hogy egy metódust kell kötelez?en definiálni a Function interfészen, ez az apply. Az implementáció els? sorában kikötöttük, hogy T és R String és Integer lesz. Tehát az apply() implementációnknak S-t kell kapnia paraméterként és Integer-t kell visszaadnia. Ez itt teljesül. Nem csinál mást mint a kapott String hosszát visszaadja.

Ahhoz hogy ezt használni tudjuk, példányosítani kell, és át kell adni processVariable() metódusnak, ami egy példányosított funkcionális interfészt

implementáló osztályt vár. Vagyis egy olyan objektumot, ami egy olyan osztályból készült, ami implementálja a funkcionális interfész metódusát.

```
MyClassUsingFunctionalif MyClassUsingFunctionalif = new MyClassUsingFunctionalif("adam");

        MyFunctionImpl myFunction = new MyFunctionImpl();

        MyClassUsingFunctionalif.processVariable(myFunction);
```

A processVariable() implementációját mi készítettük, tudjuk hogy nem csinál mást, mint meghívja a myFunction.apply() metódust.

? implementáció

Nézzünk az alábbi lehetséges ? implementációt (inline). Az apply-nak átadja a saját osztály változóját, majd az apply által visszaadott Integer-t kiírja a sysout-ra. Az hogy az apply belsejében hogy áll el? az Ingerer és hogy mit csinál a bemen? String paraméterrel teljes egészében az apply implementációra van bízva.
Nézzünk meg egy lehetséges implementációt:

```
public static void main(String[] args) {

MyClassUsingFunctionalif MyClassUsingFunctionalif = new MyClassUsingFunctionalif("adam");
MyClassUsingFunctionalif.processVariable(var1 -> { return Integer.valueOf(var1.length()); });

}
```

Ebb?l az apply lambad kifejezéssel történ? inline implementációja az alábbi:

```
var1 -> { return Integer.valueOf(var1.length()); }
```

Ami nem csinál mást, mint visszaadja a kapott string hosszát. Kívülr?l nézve a var1 nem értelmezhet?, az mindig a lamdát futtató osztály egy osztály változója, kívülr?l nem megadható.

Warning
Itt a 'var1' nem kívülr?l jöv?, a metódus meghívásakor el?állt paraméter! Ez a 'MyClassUsingFunctionalif' bels? változója, vagyis ide kívülr?l nem tudunk változót betolni a metódus meghívásakor

Tehát, ahogy ezt majd látni fogjuk a CompletionSage-nél, a 'FunctionalInterface'-t futtató osztályt kell el?re feltölteni minden olyan változóval, amire szükség van a lamda kifejezés futtatására. A fenti példákban ez egyrészt a 'MyClassUsingFunctionalif', vagy az els? példában a ArrayList osztályok.

- A 'MyClassUsingFunctionalif' konstruktorában adtuk át azt a string-et, amit a 'processVariable' feldolgoz, attól függetlenül, hogy milyen lamda kifejezéssel implementáljuk a funkcionális interfészét.
- Az ArrayList pedig bels? változóiban tárolja

Példa 3: CompletionStage.thenCompose()

Mi is ez?

- **CompletionStage Interface**:
  - ezek egymás után futtatott stage-ek, ezért hívják "Comletion"-nek. Mikor az egyik stage véget ér, potenciálisan elindít egy másik "CompletionStage"-et és így tovább, egymásba láncolva. Minden metódusa egy másik CompletionStage-et ad vissza. De ez csak egy interfész, vmilyen implementációját kell használni.
  - java.util.concurrent.CompletionStage<T> interface represents a commutation task (either synchronous or asynchronous). As all methods declared in this interface return an instance of CompletionStage itself, multiple CompletionStages can be chained together in different ways to complete a group of tasks.
- **CompletableFuture<T> implements CompletionStage<T> and Future<T>**: The static factory methods in this class are the starting points for executing tasks.

Nézzünk egy példát a CompletionStage lánchívására. A CompletionStage minden metódusa CompletionStage-el tér vissza, köztük a thenComponse() is, ezért hívható láncban.

```
method1().thenCompose(var1 -> method2(var1))
         .thenCompose(this::method3))
         .thenCompose(var2 -> { System.out.println(var2.printMe());
                                System.out.println("That was it"); return new CompletionStage(..) ;
                              } )
         .whenComplete((result, error) -> { ...}
```

A láncot indító metódusnak értelem szer?en egy **CompletionStage** példánnyal kell visszatérnie.

A **thenCompose** szignatúrája az alábbi:

```
<U> LogCompletionStage<U> thenCompose(Function<? super T, ? extends CompletionStage<U>> var1);
```

Láthatjuk, hogy egy **Function** interfészt vár paraméterül. A 'Function' interfésze nem más mint egy "gyári" funkcionális interfész típus (olyan mint az el?z? példába a 'Consumer' interfész volt). A funkcionális interfészekkel szemben támasztott követelmény, hogy egy nem implementált metódusuk legyen, ami ebben az esetben az **apply()**. A **T** az input típusa, míg **R** a visszatérési típus.

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T var1);
    ...
}
```

Tehát a **thenCompose(..)** vár egy **Function** interfész implementációt és belül meg fogja hívni az **apply(..)** metódusát, aminek át fog adni egy T típusú változót, és R típusú változót fog visszavárni visszatérési értékként. Nézzük meg a **thenCompose** implementációját:

```
public <U> CompletableFuture<U> thenCompose(
        Function<? super T, ? extends CompletionStage<U>> fn) {
        return uniComposeStage(null, fn);
    }

    private <V> CompletableFuture<V> uniComposeStage(
        Executor e, Function<? super T, ? extends CompletionStage<V>> f) {
        if (f == null) throw new NullPointerException();
        CompletableFuture<V> d = newIncompleteFuture();
        Object r, s; Throwable x;
        if ((r = result) == null)
            unipush(new UniCompose<T,V>(e, d, this, f));
        else if (e == null) {
            if (r instanceof AltResult) {
                if ((x = ((AltResult)r).ex) != null) {
                    d.result = encodeThrowable(x, r);
                    return d;
                }
                r = null;
            }
            try {
                @SuppressWarnings("unchecked") T t = (T) r;
                CompletableFuture<V> g = f.apply(t).toCompletableFuture();
        ....
    }
```

Anélkül hogy nagyon belemennénk a részletekbe, a lényeg itt is az mint a 2. példában. A CompletableFuture példánynak vannak példány változói. Fogja a T típusú változóját és meghívja vele az **apply(..)** metódust, és visszavár egy CompletionStage leszármazottat. Mivel a CompletionStage hívások láncba f?zhet?ek, a hívások egymás után hajtódnak végre. A fenti kódrészlet egy kis varázslással végrehajtja és el?veszi a láncban az el?z? CompletionStage<R> értékét és ebb?l az R-t adja oda az apply(...) metódusnak, hogy végezze el rajta a szükséges m?veleteket.

Az apply(..) metódus inline implementációja tehát mindig egy CompletionStage<R> példánnyal kell hogy visszatérjen, ahonnan az R-t fogja megkapni a következ? thenCompose hívás inline implementációja T-ként. Pszeudokóddal leírva ez így néz ki:

```
...
   .thenCompose(var1 -> {  ...
                        //Create a future with string inside
                        new CompletableFuture<String> future = new CompletableFuture<String>();
                        future.complete("adam");
                        return future;
                    })
   .thenCompose(var1 -> {
                         System.out.println(var1); //will print "adam"
                        ...
                        })
   ...
```

Az els? thenCompose(..) vissza fog térni egy CompletableFuture-el, amiben egy String-et tárol el. A második thenCompose(..) hívás végre fogja hajtani az els?t, majd annak az eredményéb?l ki fogja nyerni az "adam" Stringet és azzal fogja meghívni a Funkcionális interfész inline implementációt vagyis olyan mint ha ezt csinálná:

```
fn.apply("adam");
```

Note
Tehát az apply(..) inline implementációja csak egy cs?, amibe a thenCompose() bedob egy változót és visszavár egy végeredményt. A változó itt is a saját példány változója (történetesen az el?z? futás eredménye)

Java Method Reference and Constructor Reference

https://www.amitph.com/java-method-and-constructor-reference/

Method Reference

A metódus referencia (akár statikus akár nem) arra jó, hogy egy meglév? osztály egy metódusát át tudjuk adni a funkcionális IF definíciót váró metódusnak ahelyett hogy definiálnánk helyben lambda belsejét. Azonban ez nem egyezik meg a Funkcionális interfész implementációját tartalmazó osztály használatával. Ha metódus referenciát használunk, akkor a funkcionális IF implementáció helyére megadott osztály és metódus nem kell hogy implementálja a funkcionális interfész funkcionális metódusát, elég ha megfelel a karakterisztikája az éppen használt funkcionális if kifejezésnek, vagyis az elvárt paraméterekkel rendelkezik, és a visszatérési érték típusa is megfelel a várt funkcionális if-nek kifejezésnek.

Nézzük ismét az alábbi példa class-t, aminek van egy metódusa, ami egy funkcionális interfészt vár (processVariable). Itt egy beépített funkcionális IF-et használunk megint

```
package hu.otp.auth.loginHelper.util;
import java.util.function.Function;

public class MyClassUsingFunctionalif {

private String variable1;
```

```
public MyClassUsingFunctionalif(String a) {
this.variable1 = a;
}

public void processVariable(Function<String, Integer> fn) {

Integer length = fn.apply(variable1);
System.out.println("Value given by function IF impl: "+ length.toString());
}
}
```

Emlékeztet?ül, a java gyári Function nev? funkcionális IF implementációja így néz ki:

```
package java.util.function;

@FunctionalInterface
public interface Function<T, R> {
    R apply(T var1);
...
```

Láthatjuk, hogy egy implementálandó metódusa van, ezt lehet egy lambda in line IF-el megadni, vagy akár egy implementációs osztállyal definiálni.

Tehát, normál esetben ez m?ködne:

```
public class MyFunctionImpl implements Function<String, Integer> {
    @Override
    public Integer apply(String s) {
        return s.length();
    }
}
```

</br> Majd ha meghajtjuk a metódust ezzel az implementációval:

```
MyClassUsingFunctionalif MyClassUsingFunctionalif = new MyClassUsingFunctionalif("process_this_during_functional_if_call");
        MyClassUsingFunctionalif.processVariable(new MyFunctionImpl());
```

Akkor a konzolon meg fog jelenni hogy: "38"

Mert hogy:
A **processVariable()** metódus belsejében azt mondtuk meg, hogy fogja meg a konstruktorban kapott változót, adja azt át a **Function** funkcionális if-et implementáló osztály **apply** metódusának és amit az visszaad, azt írja ki a képerny?re. A **MyFunctionImpl** -ban rögzítettük, hogy a generikus IF milyen típusokkal fog rendelkezni (String megy be, és Integer jön ki). Azt hogy ezt hogy állítja el? az implementáció, err?l a **processVariable()** metódusban semmit sem tudunk, teljesen az implementációra van bízva.

Persze, ezt lambdával is megadhattuk volna, akkor így nézne ki:

```
MyClassUsingFunctionalif MyClassUsingFunctionalif = new MyClassUsingFunctionalif("process_this_during_functional_if_call");
        MyClassUsingFunctionalif.processVariable(input -> {return input.length();});  ---> 38
```

Itt inline, lambda definíciót használtunk. A generikus változói a **Function** funkcionális interfésznek (T és R) explicit lett definiálva azáltal, hogy a **processVariable(..)** belsejében a funkcionális IF (apply) String paraméterrel van meghívva és Integer-t vár vissza.

Na már most, ezt lehet nagyban egyszer?síteni bármilyen olyan Osztály statikus vagy nem statikus metódusával, ami String-et vár és Integer-t ad vissza. Erre szolgála **::** operátor. Statikus metódus használata esetén nem kell példányosítani az osztályt, ennyi a különbség. Ami a lényeg: **ezen osztálynak nem kell implementálnia a várt funkcionális interfész itt használt metódusát!!**

```
MyClassUsingFunctionalif MyClassUsingFunctionalif = new MyClassUsingFunctionalif("process_this_during_functional_if_call");
        MyClassUsingFunctionalif.processVariable(String::length);  ----> 38
```

Ezzel azt mondjuk meg, hogy az **R pply(T)** helyett inkább hívd meg az **Integer Sring.length(String input)** metódust.

VAGY ha nem statikus:

Tegyük fel hogy van egy ilyen osztályom, ami nem implementál semmilyen IF-et:

```
public class MyString {

    public Integer getLengthOfImput(String input) {
        return  input.length();
    }
}
```

Akkor a **getLengthOfImput(..)'** ***is átadható példányosítás után a*** *::* operátorral, mint behelyettesítés:

```
MyClassUsingFunctionalif MyClassUsingFunctionalif = new MyClassUsingFunctionalif("process_this_during_functional_if_call");

        MyString myString = new MyString();
        MyClassUsingFunctionalif.processVariable(myString::getLengthOfImput); ---> 38
```

Vagyis ahelyett hogy meghívná a funkcionális IF **R apply(T)** metódusát, inkább meghívja majd a **getLengthOfImput** metódust.

Constructor Reference

A konstruktor referencia egy speciális Metódus referencia, ami csavar még egyet a fenti metódus referencia koncepción, itt már nagyon kell figyelni. Ugyebár minden funkcionális IF visszaad egy objektumot, és vagy kap vagy nem input paramétereket. A lényeg, hogy egy el?re nem definiált módon gyártson le egy olyan objektumot, amit visszaad akkor mikor meghívjuk a funkcionális metódusát, az el?bbi példában az **R apply(T)** metódust. Ahol kap egy T-t és tök mindegy hogy hogy, csak gyártson le egy **R** osztály példányt. (ez tökre hajaz egy konstruktor definíciójára, na nézzük csak meg:))

Tekintsük az alábbi funkcionális IF-et:

```
@FunctionalInterface
public interface MyFunction {

    MyString process(String input);
}
```

Bármi is legyen az implementáció (impl osztállyal vagy inline lambda kifejezéssel) a lényeg, hogy el?állítson a MyString objektumot.
A konstruktor referencia lehet?vé teszi, hogy a funkcionális IF implementációt egy osztály konstruktorával helyettesítsük be. Vagyis a funkcionális metódus visszatérési objektumát úgy állítja el?, hogy szimplán példányosítja a megadott osztályt, jelen esetben a MyString-et. Az osztálynak rendelkeznie kell egy olyan konstruktorral, ami a funkcionális metódus input paramétereivel megegyezik. A fenti esetben tehát akkor fogjuk tudni a MyString -et konstruktor referenciában használni, ha rendelkezik egy olyan konstruktorral, ami String-et vár.

pl:

```
public class MyString {

    private String input;

    public MyString(String input) {
        System.out.println("MyString consturctor is called with input:" + input);
        this.input = input;
    }

    public String getInput() {
        return input;
    }
}
```

A konstruktor referenciát ::new val kel megadni, vagy a **:: *operátor után a** new'* kulcsszót kell írni. Egy lehetséges példa:

Tegyük fel, hogy van egy java osztályunk, amiben van egy metódus, ami felhasználja a **MyFunction** funkcionális if-et:

```
public class MyClassUsingFunctionalif {

    private String variable1;

    public MyClassUsingFunctionalif(String a) {
        System.out.println("MyClassUsingFunctionalif consturctor is called with input:" + a);
        this.variable1 = a;
    }

    public void processVariable2(MyFunction mf) {

        System.out.println("MyClassUsingFunctionalif.processVariable2 START");
        MyString myString = mf.process(variable1);

        System.out.println("MyString instance created by function if call: "+ myString.getInput());

        System.out.println("MyClassUsingFunctionalif.processVariable2 END");
    }
}
```

Vagyis vagy a MyFunction implementálásával vagy egy inline lambda kifejezéssel gyártsunk olyan kódot, ami a saját osztály String változóját (varibale1) átalakítja egy MyString osztállyá, amit ? aztán fel tud használni a metódusban (jelen esetben kíiratjuk).

És ahelyett hogy:

- Implementációt készítünk a MyFunction-hez
- lambda kifejezésben adjuk meg inline
- Metódus referenciát használunk egy olyan osztály metódusával, ami String-et vár és gyárt bel?le egy MyString-et

Simán megfogjuk a MyString osztályt és a konstruktorára hivatkozunk a **MyClassUsingFunctionalif.processVariable2(..)** metódus használatakor.

```
MyClassUsingFunctionalif myClassUsingFunctionalif = new MyClassUsingFunctionalif("process_this_during_functional_if_call");
        myClassUsingFunctionalif.processVariable2(MyString::new);
```

És ez kerül az output-ra:

```
MyClassUsingFunctionalif consturctor is called with input:process_this_during_functional_if_call
MyClassUsingFunctionalif.processVariable2 START
MyString consturctor is called with input:process_this_during_functional_if_call
MyString instance created by function if call: process_this_during_functional_if_call
MyClassUsingFunctionalif.processVariable2 END
```

Tehát az egész **MyString myString = mf.process(variable1);** hívást helyettesítette a MyString konstruktor meghívásával. Láthatjuk a log sorrendb?l is, hogy ebben a sorban:

```
myClassUsingFunctionalif.processVariable2(MyString::new);
```

Még nem példányosodik a MyString osztály. Csak akkor mikor a funkcionális IF meghívásra kerül, de azt simán behelyettesíti egy olyan konstruktor hívással, ami ugyan azokat a paramétereket várja, és a **::new**-val megadott osztálynak semmi köze a funkcionális IF-hez (MyFunction), nem implementálja azt.

Functional interface in ENUM constructors

- Ehhez els?ként ajánlott elolvasni az ENUM alapokat: Java Enum
- Második lépésnek pedig ezt: Java_Lambda#Constructor_Reference

Alapok áttekintése

A Java Enum cikkben láthattuk, hogy az enumoknak is lehet konstruktort, metódusokat és osztály változókat készíteni. Ismétlés képen itt egy enum, aminek van egy int változója, és ezt a konstruktor inicializálja, vagyis minden ENUM érték esetében meg kell adni egy 'int' számot is, amit el fog rakni az enum példányba:

```
public enum AdamTest {

    LALI(2),
    ADAM(3)
    ;

    private int enumInt;

    private AdamTest(int a) {
       this.enumInt=a;
    }
}
```

Ha olyan osztály változót készítünk az ENUM-nak, ami egy funkcionális interfész, és ezt szintén a konstruktorral inicializáljuk, akkor minden egyes ENUM érték esetében a konstruktorban meg kell adni ennek a funkcionális interfésznek a definícióját.

Tegyük fel, hogy adott az alábbi osztály, ami egy saját String változat:

```
public class MyString {

    private String input;

    public MyString(String input) {
        System.out.println("MyString consturctor is called with input:" + input);
        this.input = input;
    }

    public String getInput() {
        return input;
    }
}
```

Tegyük fel, hogy adott az alábbi funkcionális IF:

```
@FunctionalInterface
public interface MyFunction {
    MyString process(String input);
}
```

Itt a 'process(..)' implementáció egy string-et vár és egy MyString-et ad vissza.

Tekintsük az alábbi ENUM-ot. Az enum-nak van egy MyFunction típusú funkcionális if osztály változója, amit a konstruktor inicializál. Tehát minden enum értéknek meg kell adni egy MyFnction implementációt.

```
public enum MyEnum2 {

    ENUMV1(..<MyFunction implementáció>...),
    ;

    private final MyFunction function;

    private MyEnum2(MyFunction function) {

        System.out.println("ENUM constructor is called with input");
        this.function = function;
    }

    public MyFunction getFunction() {
        return function;
    }
}
```

Láttuk már, hogy elvi szinten egy funkcionális if megadására 4 lehet?ségünk van, azonban itt most nem használhatjuk mindent:

- Implementációt készítünk a MyFunction-hez: Ez nem fog m?ködni, mert az enum példányosításakor statikus kontextusban vagyunk, nincs hol példányosítsuk a funkcionális if implementációnkat, nem olyan mint amikor enum kívül, normál class metódusban használjuk.
- lambda kifejezésben adjuk meg inline
- Metódus referenciát használunk egy olyan osztály metódusával, ami String-et vár és gyárt bel?le egy MyString-et
- Konstruktor referenciát használunk

lambda kifejezésben adjuk meg inline

Az vet?dhet fel kapásból bennünk, hogy az itt lambda-val definiált funkcionális if implementáció (vagyis a 'MyString process(String) impl') mikor fog lefutni, ki fogja meghívni, és mi lesz az a String érték amin dolgozni fog. A korábbi példákban mindig volt egy akármilyen metódus, ami várta paraméterül a funkcionális if-et, majd meghívta rajta a funkcionális metódusát (jelen esetben a process(Stringt)-et), úgy, hogy az a bizonyos metódus töltötte fel az összes input paraméterrel a funkcionális if hívást. Pszeudó kóddal ez elvi szinten így néz ki amir?l itt beszélek:

```
class MyClassUsingFunctionIF {

  void myMethodUsingFunctionIF(MyFunction mf) {

        String inputForFuncion = "this is the input";
        MyString result = mf.process(inputForFunction);
  }
}
```

Itt a 'myMethodUsingFunctionIF(..)' meghívásával tudjuk implicit lefuttatni a funkcionális if implementációt. A process(..) input paraméterét saját bels? változójából tölti fel.

ENUM konstruktor referencia estében el kell kérni az enum-tol a konstruktorban inicializált funkcionális függvényt, és meg kell hívni a funkcionális interfészét úgy hogy feltöltjük azt a megfelel? paraméterekkel. Nézzük ezt részletesen. Az alábbi példában az enum konstruktorában egy 'MyFunction' funkcionális interfészt kell átadni, amit egy inline lambda kifejezéssel definiáltunk (mármint a process metódusát).

```
public enum MyEnum2 {

    ENUMV1(var1 -> {
        System.out.println("ENUM lambda is called with input: " + var1);
        return new MyString(var1);
    }),
    ;
```

{note||Az itt megadott kód ugyebár nem fog lefutni a a MyEnum2 példányosításakor. A példányosításkor a konstruktorban ezzel a lambda kifejezéssel csak megadtuk a MyFunction if tervrajzát, vagyis hogy a MyFunction osztály változó rakjon bele egy olyan "virtuális" implementációs példányt, amiben az itt megadott 'MyString process(String)' implementáció található. }}

Ahhoz hogy a 'process(String)'-et implementáló lambda kifejezésünk lefusson, els?ként példányosítani kell az ENUM-ot a megfelel? típusra, arra, ami a lambda kifejezést tartalmazza, ami nem más mint az **ENUMV1** (lehetne több értéke is az enum-nak, és mindegyikben más és más implementációt is megadhattunk volna). Majd a példányosított enmum-tól a getter-el el kell kérni a funkcionális IF osztály változót, aminek meg kell hívni a process(..) metódusát, úgy hogy megadjuk az if-en kötelez? egy darab String input paramétert.

```
System.out.println("START");

        MyEnum2 myEnum21 = MyEnum2.ENUMV1;
        System.out.println("Enum was created");
        MyString result = myEnum21.getFunction().process("process input"); <<---- itt fut csak le a konstruktorban megadott lambda
```

Fontos, hogy az ENUM példányosításakor tudunk semmit "kívülr?l" megadni, tehát, nem lehet így példányosítani az ENUM-ot:

```
MyEnum2.ENUMV1(..constructor values...); <------WRONG
```

Nézzük meg futási sorrendet, ezt logolja az stdout-ra. Látható, hogy a lambda-ba megadott kifejezés csak a 'process(..)' metódus meghívásakor futott le. A lambda kifejezés tartalmazott egy 'MyString' példányosítást, az futott le utoljára.

```
START
ENUM constructor is called with input
Enum was created
ENUM lambda is called with input: process input
MyString consturctor is called with input:process input
```

Metódus referenciát használunk

Metódus referenciát használunk egy olyan osztály metódusával, ami String-et vár és gyárt bel?le egy MyString-et:

Konstruktor referenciát használunk

Emlékezzünk rá, hogy funkcionális IF definíciókat meg kelhet adni azon osztály konstruktor referenciájával is, amikkel a funkcionális metódus visszatér, abban az esetben ha ez az osztály rendelkezik olyan konstruktorral, aminek a paraméterei megegyeznek a funkcionális metódus paramétereivel.

Pl. ha adott a már ismert Funkcionális if:

```
@FunctionalInterface
public interface MyFunction {
    MyString process(String input);
}
</sring>
Akkor a funkcionális if definiálható a '''MyString::new''' konstruktor referenciával, ha rendelkezik String paraméter? konstruktorral:
<source lang="java">
public class MyString {

    private String input;

    public MyString(String input) {   <<<<---------- ez itt a lényeg!!!!
        System.out.println("MyString consturctor is called with input:" + input);
        this.input = input;
    }

    public String getInput() {
        return input;
    }
}
```

Nézzük a már ismert java ENUM-ot, ezt egészítsük ki egy új értékkel: **ENUMV2**, ahol a funkcionális if-et a konstruktorban a 'MyString' konstruktor referenciájával adjuk meg:

```
public enum MyEnum2 {
    ENUMV1(var1 -> {
        System.out.println("ENUM lambda is called with input: " + var1);
        return new MyString(var1);
    }),
    ENUMV2(MyString::new)   <<<-------------- ez itt a lényeg!!!
    ;
    private final MyFunction function;

    private MyEnum2(MyFunction function) {
```

```
        System.out.println("ENUM constructor is called with input");
        this.function = function;
    }

    public MyFunction getFunction() {
        return function;
    }
}
```

Ha példányosítjuk az enum-ot a ENUMV2 értékkel, majd azon meghívjuk a funkcionális if process(..) metódusát, akkor fog lefutni a MyString konstruktora:

```
System.out.println("START");
        MyEnum2 myEnum22 = MyEnum2.ENUMV2;
        System.out.println("Enum was created");
        myEnum22.getFunction().process("prcoess again");
```

Az output az alábbi lesz:

```
START
Enum was created
MyString consturctor is called with input:prcoess again
```

Mire használható (komplex példa)

Stream

https://stackify.com/streams-guide-java-8/ ..TODO..