FUNCTIONAL PROGRAMMING IN JAVA

June 3 2024

Lambda Expressions

- ***A well-known problem in software engineering is that no matter what you do, user requirement changes!***

- *A well-known problem in software engineering is that no matter what you do, user requirement changes!*
- *Let's walk through an example that we'll gradually improve, showing some best practice for making your code more flexible for changing requirements.*

Lambda Expressions

- *For example, imagine an application to help a farmer understand his inventory*

Lambda Expressions

- *For example, imagine an application to help a farmer understand his inventory*
- *the farmer might want a functionality to find all* ***green*** *apples in his inventory*

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory){
        if( "green".equals(apple.getColor() ) {
            result.add(apple);
        }
    }
    return result;
}
```

An accumulator list for apples.

Select only green apples.

Lambda Expressions

- *But now the farmer changes his mind and wants to also filter* ***red*** *apples*

Lambda Expressions

- *But now the farmer changes his mind and wants to also filter **red** apples*
- *A naive solution would be to duplicate our method, rename it as **filterRedApples** and change the if condition to match red apples.*

Lambda Expressions

- *But now the farmer changes his mind and wants to also filter* ***red*** *apples*
- *A naive solution would be to duplicate our method, rename it as* ***filterRedApples*** *and change the if condition to match red apples.*
- *what if the farmer wants multiple colors: light green, dark red, yellow and so on ?*

Lambda Expressions

- *But now the farmer changes his mind and wants to also filter **red** apples*
- *A naive solution would be to duplicate our method, rename it as **filterRedApples** and change the if condition to match red apples.*
- *what if the farmer wants multiple colors: light green, dark red, yellow and so on ?*

try to abstract!

- ***What we could do is add a parameter to your method to parameterize the color and be more flexible to such changes:***

- ***What we could do is add a parameter to your method to parameterize the color and be more flexible to such changes:***

```
public static List<Apple> filterApplesByColor(List<Apple> inventory,
                                              String color) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if ( apple.getColor().equals(color) ) {
            result.add(apple);
        }
    }
    return result;
}
```

- ***Now the farmer comes back to you and says,***

Lambda Expressions

- *Now the farmer comes back to you and says,*
- *"It would be really cool to differentiate between light apples and heavy apples.*

```
public static List<Apple> filterApplesByWeight(List<Apple> inventory,
                                                          int weight) {
    List<Apple> result = new ArrayList<>();
    For (Apple apple: inventory){
        if ( apple.getWeight() > weight ){
            result.add(apple);
        }
```

Lambda Expressions

It breaks DRY! (don't repeat yourself)

It breaks DRY! (don't repeat yourself)

- ***What if we want to alter the filter traversing to enhance performance?***

It breaks DRY! (don't repeat yourself)

- ***What if we want to alter the filter traversing to enhance performance?***
- ***We now have to modify the implementation of all of your methods!***

Lambda Expressions

- *We could combine the color and weight into one method*

Lambda Expressions

- *We could combine the color and weight into one method*

```java
public static List<Apple> filterApples(List<Apple> inventory, String color,
                                       int weight, boolean flag) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if ( (flag && apple.getColor().equals(color)) ||
           (!flag && apple.getWeight() > weight) ){
            result.add(apple);
        }
    }
    return result;
}
```

A really ugly way to select color or weight

Lambda Expressions

- *We could combine the color and weight into one method*

```
public static List<Apple> filterApples(List<Apple> inventory, String color,
                                        int weight, boolean flag) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if ( (flag && apple.getColor().equals(color)) ||
           (!flag && apple.getWeight() > weight) ){
            result.add(apple);
        }
    }
    return result;
}
```

A really ugly way to select color or weight

- *What do true and false mean?*

Lambda Expressions

- *We could combine the color and weight into one method*

```
public static List<Apple> filterApples(List<Apple> inventory, String color,
                                        int weight, boolean flag) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if ( (flag && apple.getColor().equals(color)) ||
           (!flag && apple.getWeight() > weight) ){
            result.add(apple);
        }
    }
    return result;
}
```

A really ugly way to select color or weight

- *What do true and false mean?*
- *What if the farmer asks you to filter with different attributes of an apple, e.g, its size, its shape, and so on?*

Lambda Expressions

- *what if the farmer asks you for more complicated queries that combine attributes, such as green apples that are also heavy?*

- *what if the farmer asks you for more complicated queries that combine attributes, such as green apples that are also heavy?*
- *We need a better way to tell your **filterApples** method the selection criteria for apples.*

- *what if the farmer asks you for more complicated queries that combine attributes, such as green apples that are also heavy?*
- *We need a better way to tell your **filterApples** method the selection criteria for apples.*

Strategy Pattern

- ***Let's therefore define an interface to model the selection criteria:***

```
public interface ApplePredicate {

boolean test (Apple apple);

}
```

- *Let's therefore define an interface to model the **selection criteria:***

```
public interface ApplePredicate {

boolean test (Apple apple);

}
```

```
public static List<Apple> filterApples( List<Apple> inventory,
ApplePredicate p) {

List<Apple> result = new ArrayList<>();
for(Apple apple : inventory) {
if (p.test(apple)) {
result.add(apple);
} }
return result;
}
```

Lambda Expressions

- ***The only code that really matters is the implementation of the test method.***

Lambda Expressions

- *The only code that really matters is the implementation of the* ***test*** *method.*
- *This is what defines the new behaviors for the* ***filterApples*** *method.*

- *The only code that really matters is the implementation of the **test** method.*
- *This is what defines the new behaviors for the **filterApples** method.*
- *Unfortunately, because the **filterApples** method can only take objects, we have to wrap that code inside an **ApplePredicate** object.*

Lambda Expressions

- *The only code that really matters is the implementation of the* ***test*** *method.*
- *This is what defines the new behaviors for the* ***filterApples*** *method.*
- *Unfortunately, because the* ***filterApples*** *method can only take objects, we have to wrap that code inside an* ***ApplePredicate*** *object.*

Lambda Expressions

- *At the moment, when we want to pass new behavior to your filterApples method,*

- *At the moment, when we want to pass new behavior to your filterApples method,*
  - *you're forced to declare several classes that implement the **ApplePredicate interface***

Lambda Expressions

- *At the moment, when we want to pass new behavior to your filterApples method,*
  - *you're forced to declare several classes that implement the **ApplePredicate interface***
  - *and then instantiate several **ApplePredicate objects** that you allocate only once.*

Lambda Expressions

- *At the moment, when we want to pass new behavior to your filterApples method,*
  - *you're forced to declare several classes that implement the **ApplePredicate interface***
  - *and then instantiate several **ApplePredicate objects** that you allocate only once.*
- *Can we do better?*

Lambda Expressions

- *At the moment, when we want to pass new behavior to your filterApples method,*
  - *you're forced to declare several classes that implement the **ApplePredicate interface***
  - *and then instantiate several **ApplePredicate objects** that you allocate only once.*
- *Can we do better?*

anonymous classes

- ***Anonymous classes*** *allow us to declare and istantiate a class a the same time*

```
List<Apple> redApples = filterApples( inventory,
new ApplePredicate() {
public boolean test(Apple apple) {
return "red".equals(apple.getColor());
}
} ) ;
```

- *They are still unsatisfactory*

Lambda Expressions

- ***Anonymous classes*** *allow us to declare and istantiate a class a the same time*

```
List<Apple> redApples = filterApples( inventory,
new ApplePredicate() {
public boolean test(Apple apple) {
return "red".equals(apple.getColor());
}
} ) ;
```

- *They are still unsatisfactory*
- *In the context of passing a simple piece of code, we still have to create an object and explicity implement a method to define the new behavior*

- *Java 8 language designers solved this problem by introducing*

Lambda Expressions

- *Java 8 language designers solved this problem by introducing*

lambda expressions

Lambda Expressions

- ***Java 8 language designers solved this problem by introducing***

lambda expressions

- ***a more concise way to pass code***

Lambda Expressions

- *Java 8 language designers solved this problem by introducing*

lambda expressions

- *a more concise way to pass code*
- *The previous example can be rewritten in Java 8 using lambda expression:*

```
List<Apple> result =
filterApples( inventory, apple -> "red".equals(apple.getColor() )
);
```

- *Notice that, we haven't provided the type at all, yet this example still compiles!*

Lambda Expressions

- *Notice that, we haven't provided the type at all, yet this example still compiles!*
- *what's going on under the hood?*

Lambda Expressions

- *Notice that, we haven't provided the type at all, yet this example still compiles!*
- *what's going on under the hood?*

  javac *is inferring the type of the variable* ***apple*** *from its context, i.e., from the signature of* ***ApplePredicate***

- *Lambda expressions are* ***statically typed!***

Functional Interfaces

Functional Interfaces

- *In java, all method parameters have types*

Functional Interfaces

- *In java, all method parameters have types*
- *So what's the type of a lambda expression?*

Functional Interfaces

- *In java, all method parameters have types*
- *So what's the type of a lambda expression?*

Functional Interface

Functional Interfaces

- *In java, all method parameters have types*
- *So what's the type of a lambda expression?*

Functional Interface

Definition *A functional interface is an interface with a single abstract method that is used as the type of a lambda expression.*

Functional Interfaces

- *Remember the **interface ApplePredicate** we created so we could parameterize the behavior of the filter method*

Functional Interfaces

- *Remember the **interface ApplePredicate** we created so we could parameterize the behavior of the filter method*
- *It's a functional interface!*

```
public interface ApplePredicate {
boolean test (Apple apple);
}
```

Functional Interfaces

- *Why functional interfaces?*

Functional Interfaces

- *Why functional interfaces?*
- *lambda expressions let us provide the implementation of the abstract method of a functional interface directly **inline** and,*
- *treat the whole expression as an instance of a concrete implementation of the functional interface*

Functional Interfaces

- *Why functional interfaces?*
- *lambda expressions let us provide the implementation of the abstract method of a functional interface directly **inline** and,*
- *treat the whole expression as an instance of a concrete implementation of the functional interface*
- *The signature of the abstract method of the functional interface describes the signature of the lambda expression*

Common functional interfaces in Java 8 19

Table 1: Java's predefined functional interfaces

Functional Interface	Parameter Types	Return Type	Abstract Method	Other Methods
`Runnable`	none	**void**	`run`	
`Supplier<T>`	none	`T`	`get`	
`Consumer<T>`	`T`	**void**	`accept`	`andThen`
`BiConsumer<T, U>`	`T, U`	**void**	`accept`	`andThen`
`Function<T, R>`	`T`	`R`	`apply`	`compose, andThen, identity`
`BiFunction<T, U, R>`	`T, U`	`R`	`apply`	`andThen`
`UnaryOperator<T>`	`T`	`T`	`apply`	`compose, andThen, identity`
`BinaryOperator<T>`	`T, T`	`T`	`apply`	`andThen`
`Predicate<T>`	`T`	**boolean**	`test`	`and, or, negate, isEqual`
`BiPredicate<T, U>`	`T, U`	**boolean**	`test`	`and, or, negate`

examples of lamba expressions

```
Runnable noArguments = () -> System.out.println("Hello World");

ActionListener oneArguments = event ->
System.out.println("button clicked");

Runnable multiStatement = () -> {
System.out.print("Hello World");
System.out.println("Hello World");
}

BinaryOperator<Long> add = (x,y) -> x + y;

BinaryOperator<Long> addExplicit = (Long x,Long y) -> x + y;

Predicate<Integer> atleast5 = x -> x > 5;
```

Streams

Streams

- ***Streams*** *are an update to the Java API that lets you manipulate collections of data in a declarative way.*

Streams

- ***Streams*** *are an update to the Java API that lets you manipulate collections of data in a declarative way.*
- *We can think them as fancy iterators over a collection of data.*

Streams

- *Let's get started from an example:*

- *Let's get started from an example:*

```
int count = 0;
Iterator<Artist> iterator = allArtists.iterator();
while(iterator.hasNext()) {
    Artist artist = iterator.next();
    if (artist.isFrom("London")) {
        count++;
    }
}
```

- ***Let's get started from an example:***

```java
int count = 0;
Iterator<Artist> iterator = allArtists.iterator();
while(iterator.hasNext()) {
    Artist artist = iterator.next();
    if (artist.isFrom("London")) {
        count++;
    }
}
```

Streams

- ***Let's get started from an example:***

```java
int count = 0;
Iterator<Artist> iterator = allArtists.iterator();
while(iterator.hasNext()) {
    Artist artist = iterator.next();
    if (artist.isFrom("London")) {
        count++;
    }
}
```


```java
long count = allArtists.stream()
                       .filter(artist -> artist.isFrom("London"))
                       .count();
```

Streams

- *Collections in Java 8 support a new stream method that returns a stream.*

`default Stream<E>`	`stream()` Returns a sequential `Stream` with this collection as its source.

Streams

- *Collections in Java 8 support a new stream method that returns a stream.*

`default Stream<E>`	`stream()` Returns a sequential `Stream` with this collection as its source.

- *The **Stream interface** contains a series of functions each of which corresponds to a common operation you might perform on a **Collection** (**using functional approach!**)*

- *Streams are about expressing computations such as* ***filter****,* ***map****.*

Streams

- *Streams are about expressing computations such as **filter**, **map**.*
- *Streams consume from a data-providing source such as collections and arrays.*

Streams

- *Streams are about expressing computations such as* ***filter***, ***map***.
- *Streams consume from a data-providing source such as collections and arrays.*
- *Streams allow operations to be chained and form a larger pipeline.*

Streams

- *Streams are about expressing computations such as* ***filter****,* ***map****.*
- *Streams consume from a data-providing source such as collections and arrays.*
- *Streams allow operations to be chained and form a larger pipeline.*
- *In contrast to collections, which are iterated explicitly using an iterator, stream operations do the iteration behind the scene.*

Stream operations

Table 5.1. Intermediate and terminal operations

Operation	Type	Return type	Type/functional interface used	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
distinct	Intermediate (stateful-unbounded)	Stream<T>		
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>

Stream operations

Operation	Type	Return type	Type/functional interface used	Function descriptor
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		

- *Notice that, since we are practicing functional programming when using the Streams API,* ***we aren't changing the content of the Collection!***

What's actually going on

- *We can actually break the previous example into two simpler operations:*

What's actually going on

- *We can actually break the previous example into two simpler operations:*
  - *Finding all the artists from London*

What's actually going on

- *We can actually break the previous example into two simpler operations:*
  - *Finding all the artists from London*
  - *Counting a list of artists*

What's actually going on

- *We can actually break the previous example into two simpler operations:*
  - *Finding all the artists from London*
  - *Counting a list of artists*

You may think that we would need two loop, as there are two operations

What's actually going on

- *We can actually break the previous example into two simpler operations:*
  - *Finding all the artists from London*
  - *Counting a list of artists*

  You may think that we would need two loop, as there are two operations

  Wrong!

- *Why?*

What's actually going on

- *We can actually break the previous example into two simpler operations:*
  - *Finding all the artists from London*
  - *Counting a list of artists*

  You may think that we would need two loop, as there are two operations

  Wrong!

- *Why?*

  Laziness

What's actually going on

- *Some of the methods of Stream interface are* ***lazy*** *others are* ***eager***

What's actually going on

- *Some of the methods of Stream interface are **lazy** others are **eager***
- *If it gives you back a Stream, it's lazy; if it gives you back another value or void, then it's eager.*

What's actually going on

- *Some of the methods of Stream interface are **lazy** others are **eager***
- *If it gives you back a Stream, it's lazy; if it gives you back another value or void, then it's eager.*
- *The **laziness** allow us to pipe together lots of different operations over our collection and iterate over it only once*

What's actually going on

- *Let's consider the following example:*

```
allArtists.stream()
          .filter(artist -> {
              System.out.println(artist.getName());
              return artist.isFrom("London");
          });
```

What's actually going on

- *Let's consider the following example:*

```
allArtists.stream()
          .filter(artist -> {
              System.out.println(artist.getName());
              return artist.isFrom("London");
          });
```

- *If we run this code, the program doesn't print anything*

What's actually going on

- *Let's consider the following example:*

```
allArtists.stream()
          .filter(artist -> {
              System.out.println(artist.getName());
              return artist.isFrom("London");
          });
```

- *If we run this code, the program doesn't print anything*
- *If we add the same printout to a stream that has a terminal step, such as the counting operation, then we will see the names of our artists printed out*

What's actually going on

- *Executing an intermediate operation such as filter() does not actually perform any filtering*

What's actually going on

- *Executing an intermediate operation such as filter() does not actually perform any filtering*
- *It stores the provided operation/function and return the new stream.*

What's actually going on

- *Executing an intermediate operation such as filter() does not actually perform any filtering*
- *It stores the provided operation/function and return the new stream.*
- *The pipeline accumulates these newly created streams.*

What's actually going on

- *Executing an intermediate operation such as filter() does not actually perform any filtering*
- *It stores the provided operation/function and return the new stream.*
- *The pipeline accumulates these newly created streams.*
- *The time when terminal operation is called, traversal of streams begins and the associated function is performed one by one.*

Example of Refactoring Code 3

Example of Refactoring Code

- *let's look at an example of some collections code that uses loops to perform a task and iteratively refactor it into a stream-based implementation*

Example of Refactoring Code

- *let's look at an example of some collections code that uses loops to perform a task and iteratively refactor it into a stream-based implementation*

```java
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    for(Album album : albums) {
        for (Track track : album.getTrackList()) {
            if (track.getLength() > 60) {
                String name = track.getName();
                trackNames.add(name);
            }
        }
    }
    return trackNames;
}
```

Example of Refactoring Code 3

- *The first thing that we're going to change is the for loops. We'll keep their bodies in the existing Java coding style for now and move to using the* ***forEach*** *method on* ***Stream****.*

Example of Refactoring Code 3

- *The first thing that we're going to change is the for loops. We'll keep their bodies in the existing Java coding style for now and move to using the **forEach** method on **Stream**.*

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    albums.stream()
          .forEach(album -> {
              album.getTracks()
                   .forEach(track -> {

                       if (track.getLength() > 60) {
                           String name = track.getName();
                           trackNames.add(name);
                       }
                   });
          });
      return trackNames;
  }
```

Example of Refactoring Code 3

- *The inner* ***forEach*** *call does three things here:*

Example of Refactoring Code 3

- *The inner **forEach** call does three things here:*
  - *finding only tracks over a minute in length,*

Example of Refactoring Code 3

- *The inner **forEach** call does three things here:*
  - *finding only tracks over a minute in length,*
  - *getting their names,*

Example of Refactoring Code 3

- *The inner* ***forEach*** *call does three things here:*
  - *finding only tracks over a minute in length,*
  - *getting their names,*
  - *and adding their names into our name Set.*

```java
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    albums.stream()
          .forEach(album -> {
              album.getTracks()
                   .filter(track -> track.getLength() > 60)
                   .map(track -> track.getName())
                   .forEach(name -> trackNames.add(name));
          });
    return trackNames;
}
```

Example of Refactoring Code 3

- ***What we really want to do is find a way of transforming our album into a stream of tracks.***

Example of Refactoring Code 3

- *What we really want to do is find a way of transforming our album into a stream of tracks.*
- *We know that whenever we want to transform or replace code, the operation to use is **map***

Example of Refactoring Code 3

- *What we really want to do is find a way of transforming our album into a stream of tracks.*
- *We know that whenever we want to transform or replace code, the operation to use is **map***
- *This is the more complex case of map, **flatMap**, for which the output value is also a Stream and we want them merged together.*

Example of Refactoring Code 3

```java
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();

    albums.stream()
          .flatMap(album -> album.getTracks())
          .filter(track -> track.getLength() > 60)
          .map(track -> track.getName())
          .forEach(name -> trackNames.add(name));

    return trackNames;
}
```

Example of Refactoring Code 3

- *It's not enough yet*

Example of Refactoring Code 3

- *It's not enough yet*
- *We're still creating a* ***Set*** *by hand and adding every element in at the end.*

Example of Refactoring Code 3

- *It's not enough yet*
- *We're still creating a* ***Set*** *by hand and adding every element in at the end.*

```
public Set<String> findLongTracks(List<Album> albums) {
    return albums.stream()
                 .flatMap(album -> album.getTracks())
                 .filter(track -> track.getLength() > 60)
                 .map(track -> track.getName())
                 .collect(toSet());
}
```

- *Many existing object-oriented design patterns can be written in a more concise way using lambda expressions*

- *Many existing object-oriented design patterns can be written in a more concise way using lambda expressions*
- *In this section, we explore the following design patterns:*

- *Many existing object-oriented design patterns can be written in a more concise way using lambda expressions*
- *In this section, we explore the following design patterns:*
  - *Composite*

- *Many existing object-oriented design patterns can be written in a more concise way using lambda expressions*
- *In this section, we explore the following design patterns:*
  - *Composite*
  - *Decorator*

Composite

- *Consider the following example:*

- *Consider the following example:*
  - *We would like to validate whether a text input is properly formatted for different criteria (e.g, it consists of only lowercase letters or is numeric).*

Composite


```
public interface ValidationStrategy {
boolean execute(String s);
}

public class IsAllLowerCase
implements ValidationStrategy {

public boolean execute(String s)
return s.matches(”[a-z]+”);
}

public class IsNumeric
implements ValidationStrategy {

public boolean execute(String s)
return s.matches(”\\d+”);
```

Composite

- *We can then use these different validation strategies in our program:*

Composite

- ***We can then use these different validation strategies in our program:***

```
public class Validator{
    private final ValidationStrategy strategy;

    public Validator(ValidationStrategy v){
        this.strategy = v;
    }

    public boolean validate(String s){
        return strategy.execute(s);
    }
}

Validator numericValidator = new Validator(new IsNumeric());
boolean b1 = numericValidator.validate("aaaa");
Validator lowerCaseValidator = new Validator(new IsAllLowerCase ());
boolean b2 = lowerCaseValidator.validate("bbbb");
```

Returns `false`

Returns `true`

Composite

- ***what happens if we now want to validate the string based on***

Composite

- *what happens if we now want to validate the string based on*
  - *containing either only lowercase characters or being numeric?*

Composite

- *what happens if we now want to validate the string based on*
  - *containing either only lowercase characters or being numeric?*
  - *containing only uppercase characters?*

Composite

- *what happens if we now want to validate the string based on*
  - *containing either only lowercase characters or being numeric?*
  - *containing only uppercase characters?*
  - *containing only uppercase characters or only lowercase characters or being numeric?*

Composite

- *what happens if we now want to validate the string based on*
  - *containing either only lowercase characters or being numeric?*
  - *containing only uppercase characters?*
  - *containing only uppercase characters or only lowercase characters or being numeric?*

Composite

$\textit{Lambda expressions approach:}$

Lambda expressions approach:

- *Notice that* ***ValidationStrategy*** *is a functional interface (in addition, it has the same function signature as* ***Predicate<String>****).*

Composite


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

Predicate<String> IsAllLowerCase = s -> s.matches("[a-z]+");

Predicate<String> IsNumeric = s -> s.matches("\\d+");

Predicate<String> IsAllUpperCase = s -> s.matches("[A-Z]+");

Validator validator' = new Validator
(IsAllLowerCase.or(IsNumeric));

Validator validator" = new Validator
(IsAllLowerCase.or(IsNumeric).or(IsAllUppercase));

}
```

Decorator

Decorator


```
public class Signal {
private int magnitude;

public Signal (int theMagnitude) {
magnitude = theMagnitude;
}

public int getMagnitude() {
return magnitude;
}

}
```

Decorator


```
public abstract class SignalTransformer {
private  SignalTransformer next;

public SignalTransformer (SignalTransformer nextTransformer) {
next = nextTransformer;
}

public Signal transform(Signal signal) {
if (next != null) { return next.transform(signal); }
return signal;
}
```

Decorator


```
public class SignalAugmenter extends SignalTransformer {
public SignalAugmenter() {}

public SignalAugmenter(SignalTransformer next) { super(next); }

public Signal transform(Signal signal) {
return super.transform
(new Signal(signal.getMagnitude() + 1) ); }

}
```

Decorator


```
public class SignalDamper extends SignalTransformer {
public SignalDamper() {}

public SignalDamper(SignalTransformer next) { super(next); }

public Signal transform(Signal signal) {
return super.transform
(new Signal(signal.getMagnitude() - 1) ); }

}
```

Decorator


```
public class SignalAugmenter extends SignalTransformer {
private int factor

public SignalMultiplier(int multiplicationFactor) {
factor = multiplicationFactor; }

public SignalMultiplier(int multiplicationFactor,
SignalTransformer next) {

super(next);
factor = multiplicationFactor; }

public Signal transform(Signal signal) {
return super.transform
(new Signal(signal.getMagnitude() * factor) ); }

}
```

Decorator


```
public class UserTransformers {

public static void applyTransformation(Signal signal
SignalTransformer signalTransformer) {

signalTransformer.transform(signal).getMagnitude()); }

public static void main(String[] args) {
Signal signal = new Signal(10);
applyTransformation(signal,new SignalAugmenter());
applyTransformation(signal,new SignalMultiplier(2,new SignalAug-
menter()));

}
```

Lambda expressions approach:

- *Let's consider the* ***Function*** *functional interface*

Lambda expressions approach:

- ***Let's consider the Function functional interface***
- ***it turns a value or object into another***

Lambda expressions approach:

- *Let's consider the **Function** functional interface*
- *it turns a value or object into another*
- *The **map** method of **Stream** make use of this*

Decorator

Lambda expressions approach:

- *Let's consider the **Function** functional interface*
- *it turns a value or object into another*
- *The **map** method of **Stream** make use of this*
- *but it may be used in other contexts as well.*

Decorator

- ***Let's write a method that takes an integer value and applies a given Function, like so:***

```
public static void applyFunction(int value, String message,
Function<Integer,Integer> mapper) {
System.out.println(value + "" + message + ":" +
mapper.apply(value) ); }

Examples of using the applyFunction method:

Function<Integer,Integer> increment = value -> value + 1;
applyFunction(5, "incremented", incrtement);

Function<Integer,Integer> square = value -> value * value;
applyFunction(5, "square", square);
```

Decorator

- ***What if we want to perform a combined operation, e.g, increment and then square?***

- *What if we want to perform a combined operation, e.g, increment and then square?*
- *The **applyFunction** receives only one **Function***

Decorator

- *What if we want to perform a combined operation, e.g, increment and then square?*
- *The **applyFunction** receives only one **Function***
- *We want a Function that will perform increment **and then** perform the square operation*

- *What if we want to perform a combined operation, e.g, increment and then square?*
- *The **applyFunction** receives only one **Function***
- *We want a Function that will perform increment **and then** perform the square operation*

andThen default method in Function

Decorator

- *The* ***andThen*** *method creates a* ***Function*** *that will perform*

Decorator

- *The **andThen** method creates a **Function** that will perform*
  - *the mapping function to the left of **andThen** first*

Decorator

- *The **andThen** method creates a **Function** that will perform*
  - *the mapping function to the left of **andThen** first*
  - *and pass the result of that operation to the mapping function given as argument to the **andThen** method.*

Decorator

- *The* ***andThen*** *method creates a* ***Function*** *that will perform*
  - *the mapping function to the left of* ***andThen*** *first*
  - *and pass the result of that operation to the mapping function given as argument to the* ***andThen*** *method.*
- *We can use the* ***andThen*** *method to create a highly expressive and easy-to-use decorator pattern implementation!*

- *The **Signal** class stays the same as before*
- *Let's get rid of **SignalTransformer**, **SignalAugmenter**, **SignalDamper**, and **SignalMultiplier**.*
- *We will rewrite the **applyTransformation** function of **UserTransformer**, like so:*

```
public static void applyTransformation(Signal signal,
Function<Signal, Signal> signalTransformer) {

signalTransformer.apply(signal).getMagnitude()); }
```

Decorator


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

Function<Signal, Signal> augmentMagnitude = signal -> new
Signal(signal.getMagnitude() + 1);

Function<Signal, Signal> dampenMagnitude = signal -> new
Signal(signal.getMagnitude() - 1);

Function<Signal, Signal> doubleMagnitude = signal -> new
Signal(signal.getMagnitude() * 2);

Signal signal = new Signal(10);
applyTransformation(signal,
dampenMagnitude.andThen(doubleMagnitude).
andThen(augmentMagnitude));

}
```

Conclusion

Conclusion

- *Should we use lambda expressions all the time?*

Conclusion

- *Should we use lambda expressions all the time?*

No!

Conclusion

- *Should we use lambda expressions all the time?*

No!

- *They work great when the behavior to execute is simple*

Conclusion

- *Should we use lambda expressions all the time?*

No!

- *They work great when the behavior to execute is simple*
- *In some case may be more complex:*
  - *it could have state*

Conclusion

- *Should we use lambda expressions all the time?*

No!

- *They work great when the behavior to execute is simple*
- *In some case may be more complex:*
  - *it could have state*
  - *it needs several methods*

Conclusion

- *Should we use lambda expressions all the time?*

No!

- *They work great when the behavior to execute is simple*
- *In some case may be more complex:*
  - *it could have state*
  - *it needs several methods*
- *In those cases, the object oriented solution remains the best choice!*

References

References

- *Richard Warburton, Java 8 Lambdas, 2014*
- *Raoul-Gabriel Urma, Mario Fusco, and Alan Mycro, Java 8 in Action: Lambdas, streams, and functional-style programming*
- *Refactoring to functional Style in Java 8: Applying the Decorator pattern, Venkat Subramaniam, Medium*

Fine