Functional Programming Patterns With Java 8

by Victor Rentea · Jun. 17, 18 · Java Zone · Tutorial

It's been four years since functional programming became feasible in Java. The means we have had four years to play with Java 8.

And we've played... and played. After developing several large enterprise projects that made heavy use of Lambdas and Streams, consulted many other projects in doing so, and taught these concepts to hundreds of developers as an independent trainer, I think it's time to summarize patterns, best practices, and anti-patterns.

I wrote this article due to the level of enthusiasm received at the talks I gave this year at Devoxx. If you are interested in learning more from a passionate, entertaining, and lightning-fast, live-coding video, check out this recording of my talk.

This article will walk you through a series of simplified refactoring exercises from traditional imperative-style code to functional-style code in Java 8, continuously aiming for Simplicity Clean Code. To gain the maximum benefits from this article, you should have *some* practical experience with Java 8 features. I submitted each phase of the exercises on this GitHub repository, so feel free to walk through the repository yourself to see it all.

1) Prefer Named Functions Over Anonymous Lambdas

To warm up, let's start with the simple task of bringing the details of some users to the UI. We'll start with a traditional review of the entities list to convert each `User` to a `UserDto`:

```
public List<UserDto> getAllUsers() {
        List<User> users = userRepo.findAll();
        List<UserDto> dtos = new ArrayList<>();
        for (User user : users) {
                UserDto dto = new UserDto();
                dto.setUsername(user.getUsername());
                dto.setFullName(user.getFirstName() + " " + user.getLastName().toUpperCase(
                dto.setActive(user.getDeactivationDate() == null);
                dtos.add(dto);
        }
```

```
        }
        return dtos;
}
```

However, I'm not very proud of this code, because it's likely that I'll be writing similar code repeatedly for many use cases. So, let's cut the boilerplate using Java 8:

```
public List<UserDto> getAllUsers() {
        return userRepo.findAll().stream()
                .map(user -> {
                        UserDto dto = new UserDto();
                        dto.setUsername(user.getUsername());
                        dto.setFullName(user.getFirstName() + " " + user.getLastName().toUp
                        dto.setActive(user.getDeactivationDate() == null);
                        return dto;
                })
                .collect(toList());
}
```

That's better. But, I'm still not happy with it. This lambda I wrote demonstrates an "anonymous function." As a clean code maniac, I have a problem with that – I want expressive names. So, I quickly extracted the lambda content into a separate method:

```
public List<UserDto> getAllUsers() {
        return userRepo.findAll().stream().map(this::toDto).collect(toList());
}

private UserDto toDto(User user) {
        UserDto dto = new UserDto();
        dto.setUsername(user.getUsername());
        dto.setFullName(user.getFirstName() + " " + user.getLastName().toUpperCase());
        dto.setActive(user.getDeactivationDate() == null);
        return dto;
}
```

Nice. The code was simple enough in the previous version, but now it's slightly better. It only took me three seconds with my IDE. I always advise my trainees to master those refactoring shortcuts!

Sometimes, this conversion of logic is so trivial, as seen in this example, that we could put it directly in the DTO constructor. Please note that I would require DTO's to depend on entities, but not vice-versa (the Dependency Inversion Principle):

```
public class UserFacade {
        private UserRepo userRepo;

        public List<UserDto> getAllUsers() {
```

```
                return userRepo.findAll().stream().map(UserDto::new).collect(toList());
        }
}

public class UserDto {
        private String username;
        private String fullName;
        private boolean active;
        public UserDto(User user) {
                username = user.getUsername();
                fullName = user.getFirstName() + " " + user.getLastName().toUpperCase();
                active = user.getDeactivationDate() == null;
        }
        ...
}
```

Now, let's imagine that this conversion requires the help of some other component, which we would like to inject using Spring, Guice, CDI, etc. But, injecting a dependency in a class we instantiate would require very complex code. Instead, I would come back to the previous version, but if this conversion does grow too complex, we should move it to a separate `UserMapper` class and reference it from there:

```
@Service
public class UserFacade {
          @Autowired
          private UserRepo userRepo;
          @Autowired
          private UserMapper mapper;

          public List<UserDto> getAllUsers() {
                   return userRepo.findAll().stream().map(mapper::toDto).collect(toList());
          }
}

@Component
public class UserMapper {
          @Autowired
          private OtherClass otherClass;

          public UserDto toDto(User user) {
                   UserDto dto = new UserDto();
                   dto.setUsername(user.getUsername());
                   ... // code using otherClass
                   return dto;
          }
}
```

The key point is: **always extract complex lambdas into functions with an expressive name** that you can then reference using four-dots (`::`) from:

- the same class (`this::`);
- another class (`mapper::`);
- some static helper method (`SomeClass::`);
- the Stream *item* type (`Item::`);
- even some constructor (`UserDto::new`), if it's simple enough;

In short: **never type** `-> {` .

2) Stream Wrecks

Suppose you've worked with lambdas and streams ever since they were added to the language. And, you need to prove that, right? So, you implement a use case:

```
public List<Product> getFrequentOrderedProducts(List<Order> orders) {
        return orders.stream()
                        .filter(o -> o.getCreationDate().isAfter(LocalDate.now().minusYears
                        .flatMap(o -> o.getOrderLines().stream())
                        .collect(groupingBy(OrderLine::getProduct, summingInt(OrderLine::ge
                        .entrySet()
                        .stream()
                        .filter(e -> e.getValue() >= 10)
                        .map(Entry::getKey)
                        .filter(p -> !p.isDeleted())
                        .filter(p -> !productRepo.getHiddenProductIds().contains(p.getId())
                        .collect(toList());
}
```

You count how many times products were ordered during the previous year. Now, take only the frequent ordered Products (>=10) and return them, if they were not logically deleted or explicitly hidden from database.

And you go home happy...

But we will find you! Management won't be able to fire you — I agree – but, who can read that stuff?! Admit it... who will ever want to work with you?

The worst thing about this code is that each line returns a different type. You won't see those types unless you hover the methods with your mouse in your IDE (detective work).

One of the most important rules of clean code is: **small methods**. So, let's break this long chain into two methods by looking at the code we see `.collect(..)` immediately followed by `.stream()` . Since you gathered the items in a collection anyway, why don't we explain what that collection is by extracting a method with a nice

name? Furthermore, let's replace `!order.isDeleted()` with `order.isNotDeleted()`, to be able to use `::`.

```
1  public List<Product> getFrequentOrderedProducts(List<Order> orders) {
2            return getProductCountsOverTheLastYear(orders).entrySet().stream()
3                                .filter(e -> e.getValue() >= 10)
4                                .map(Entry::getKey)
5                                .filter(Product::isNotDeleted)
6                                .filter(p -> !productRepo.getHiddenProductIds().contains(p.get
7                                .collect(toList());
8  }
9
10 private Map<Product, Integer> getProductCountsOverTheLastYear(List<Order> orders) {
11           return orders.stream()
12                               .filter(o -> o.getCreationDate().isAfter(LocalDate.now().minus
13                               .flatMap(o -> o.getOrderLines().stream())
14                               .collect(groupingBy(OrderLine::getProduct, summingInt(OrderLin
15 }
```

But, only then do we notice that at line #6 we might be querying an external system in a loop!! OMG! And that's something you should never-ever do. So, let's get that `hiddenProductIds` list before we start streaming. We may even go further and extract the check whether the product is hidden in a `Predicate` local variable. It could help the reader, if he's comfortable keeping a function in a variable :).

```
1  public List<Product> getFrequentOrderedProducts(List<Order> orders) {
2          List<Long> hiddenProductIds = productRepo.getHiddenProductIds();
3          Predicate<Product> productIsNotHidden = p -> !hiddenProductIds.contains(p.getId());
4          return getProductCountsOverTheLastYear(orders).entrySet().stream()
5                          .filter(e -> e.getValue() >= 10)
6                          .map(Entry::getKey)
7                          .filter(Product::isNotDeleted)
8                          .filter(productIsNotHidden)
9                          .collect(toList());
10 }
```

There is one more thing we could do: we could name the stream of frequent products and make it a variable of type `Stream`. As we know, the `Stream` items aren't actually evaluated at this point, but only at the end, when we `.collect()` it. However, working with `Stream<>` variables is sometimes discouraged, because careless developers might try to re-use it (re-traverse it), so before doing it, make sure your team is fully aware of this common occurrence.

```
1  public List<Product> getFrequentOrderedProducts(List<Order> orders) {
2          List<Long> hiddenProductIds = productRepo.getHiddenProductIds();
3          Predicate<Product> productIsNotHidden = p -> !hiddenProductIds.contains(p.getId());
4          Stream<Product> frequentProducts = getProductCountsOverTheLastYear(orders).entrySet
5                          .filter(e -> e.getValue() >= 10)
6                          .map(Entry::getKey);
```

```
        return frequentProducts
                        .filter(Product::isNotDeleted)
                        .filter(productIsNotHidden)
                        .collect(toList());
}
[...]
```

The idea here is to **avoid excessive method chaining by introducing explanatory variables.** This means extracting methods and even working with variables of a function or `Stream` type, in order to make the code as clear as possible to your reader.

3) Fighting the Greatest Beast of All: Null Pointer

Yes! Null Pointer wasn't always there! In fact, the cause of the most frequent bug of all was actually invented!

Let's get rid of it now, shall we?

The exercise is simple: we need to return a nicely formatted line to print the applicable discount for a customer based on the fidelity points he gathered:

```
public String getDiscountLine(Customer customer) {
        return "Discount%: " + getDiscountPercentage(customer.getMemberCard());
}

private Integer getDiscountPercentage(MemberCard card) {
        if (card.getFidelityPoints() >= 100) {
                return 5;
        }
        if (card.getFidelityPoints() >= 50) {
                return 3;
        }
        return null;
}
```

Let's see what it returns for 60 and then for 10 points:

```
System.out.println(discountService.getDiscountLine(new Customer(new MemberCard(60))));
System.out.println(discountService.getDiscountLine(new Customer(new MemberCard(10))));
```

Prints:

```
Discount%: 3
Discount%: null
```

But, I suppose showing your user a "null" is not something you would like to do every day. Obviously, the

problem is that we concatenate a potential `null` integer. Fixing it is trivial:

```
1  public String getDiscountLine(Customer customer) {
2          Integer discount = getDiscountPercentage(customer.getMemberCard());
3          if (discount != null) {
4                  return "Discount%: " + discount;
5          } else {
6                  return "";
7          }
8  }
```

Prints:

```
1  Discount%: 3
```

Next, we return an empty string, if there is no discount. But, to do that, we've polluted our code with a null-check. And, to make matters worse, we have to find the problem we had by peeking into the `getDiscountPercentage()` function to see when it might return a `null`. But, this technique does not scale in large code bases. Instead, your API should make it clear that the function might return nothing.

```
1  import java.util.Optional.*;
2
3  public String getDiscountLine(Customer customer) {
4          Optional<Integer> discount = getDiscountPercentage(customer.getMemberCard());
5          if (discount.isPresent()) {
6                  return "Discount%: " + discount.get();
7          } else {
8                  return "";
9          }
10 }
11
12 private Optional<Integer> getDiscountPercentage(MemberCard card) {
13         if (card.getFidelityPoints() >= 100) {
14                 return of(5);
15         }
16         if (card.getFidelityPoints() >= 50) {
17                 return of(3);
18         }
19         return empty();
20 }
```

Naturally, what I did was replace the null check with a call to `Optional.isPresent()` at line 3. But, sometimes, the first idea that pops into our head is NOT always the best one. When you play with `Optional`, you need to think of it the other way around. Whenever you try to change what's in the magic box, apply a function to that box using `.map()`, so that the contents of the box is transformed only in case there was something inside.

using `map()`, so that the contents of the box is transformed only in case there was something inside.

```
public String getDiscountLine(Customer customer) {
return getDiscountPercentage(customer.getMemberCard())
      .map(d -> "Discount%: " + d).orElse("");
}
```

Not only is the code more concise, but it's also easier to read once you get used to this style.

Phew! We only have one more test to do: a customer without a `MemberCard`:

```
System.out.println(discountService.getDiscountLine(new Customer()));
```

Prints:

```
Exception in thread "main" java.lang.NullPointerException...
```

KABOOM! There you go! We often rush in terror to see where the exception pops up from. Here, it's in the first line of the `getDiscountPercentage()` function. This is due to some boundary value (`null`) for the `MemberCard` parameter that we never handled. Let's fix that right away – hide that bug as quickly as possible and pretend we never saw it:

```
private Optional<Integer> getDiscountPercentage(MemberCard card) {
        if (card == null) {
                return empty();
        }
        if (card.getFidelityPoints() >= 100) {
                return of(5);
        }
        if (card.getFidelityPoints() >= 50) {
                return of(3);
        }
        return empty();
}
```

See, we rushed again. And, again, we missed one design insight (do you see a pattern?). We quickly applied *defensive programming* here and guarded our code against all other invalid data. But, it's said that the best defense is the offensive. What if, instead of guarding our code in fear, we said: "Wait a second. So, the member card can be absent for a customer? Then, the `Customer.getMemberCard()` should return an `Optional<MemberCard>`."

```
public class Customer {
          ...
          public Optional<MemberCard> getMemberCard() {
                  returnofNullable(memberCard);
          }
}
```

Yes, I'm touching on sacred things here. I dared to change the domain entity! But, I believe I made it more expressive because the link between a customer and a member card was actually optional (Note: the field and the setter still use the 'raw' type `MemberCard`). Then, instead of testing for `null`, we test if `.isPresent()`:

```
1  private Optional<Integer> getDiscountPercentage(Optional<MemberCard> cardOpt) {
2          if (!cardOpt.isPresent()) {
3                  returnempty();
4          }
5  ...// Wait a bit!
```

STOP!

We gained nothing! I'd say this code gets even uglier! But, there was this clean code rule**: don't take nullable parameters**, because the first thing you need to do is check for null. In Java 8, this translates to **don't take Optional parameters**.

Let's twist our mind again and see that we should apply the `getDiscountPercentage()` to the `MemberCard`, only if there is a member card. So, let's undo our changes to `getDiscountPercentage()` go back to the `getDiscountLine()` and start there from the `Optional<MemberCard>`:

```
1  public String getDiscountLine(Customer customer) {
2            return customer.getMemberCard()
3                                .map(card -> getDiscountPercentage(card))
4                                .map(d -> "Discount%: " + d)
5                                .orElse("");
6  }
```

But, the output might surprise us:

```
1  Discount%: Optional[3]
2  Discount%: Optional.empty
```

It's because *d* at line #4 is no longer an `Integer`, but an `Optional<Integer>`. You'll see it yourself if you hover your mouse over the first `.map()` and look at the return type: `Optional<Optional<Integer>>`. Here, you got a number inside a box inside another box. It's like wrapping your kid a present for Christmas in multiple nested wraps just to increase the thrill of discovery. In our case, we will use the monadic nature of `Optional` and use `.flatMap()` instead of `.map()` to get rid of the extra wrapping. (To do on a cold windy night: read more about Monads). There is a lot more to using Optionals, but, in my experience as a trainer, the most difficult mental step is the one described here.

```
1  public String getDiscountLine(Customer customer) {
2              return customer.getMemberCard()
3                                    .flatMap(this::getDiscountPercentage)
4                                    .map(d -> "Discount%: " + d)
```

```
5                                       .orElse("");
6   }
```

So, **whenever null gives you problems in Java 8, don't hesitate to jump on Optional** and apply transformation functions on the, potentially empty, magic box. The clean code rule becomes: **don't take Optional params;** instead, **return an Optional whenever your function wants to signal to your caller that there might be NO return value in some cases**.

4) The Loan Pattern / Passing a block

For the following exercise, let's export the orders to a CSV file:

```
1   public File exportFile(String fileName) throws Exception {
2           File file = new File("export/" + fileName);
3           try (Writer writer = new FileWriter(file)) {
4                   writer.write("OrderID;Date\n");
5                   repo.findByActiveTrue()
6                           .map(o -> o.getId() + ";" + o.getCreationDate())
7                           .forEach(writer::write);
8                   return file;
9           } catch (Exception e) {
10                  // TODO send email notification
11                  log.debug("Gotcha!", e); // TERROR-Driven Development
12                  throw e;
13          }
14  }
```

I'll open a `Writer`, stream over all the orders, convert them, and then write each to the file. The vague odor of fear at the end of this example stems from the possibility that, perhaps, no one will ever catch my exception afterward. Ideally, you should trust your team with these exceptions, so that if they are thrown on any threads, they are gracefully caught and logged.

Perfect!

But, it doesn't compile!

This is because the `Writer.write()` method declares it throws `IOException`, even when the `Consumer` expected by the `.forEach` does not. You should *suffer* if you throw checked exceptions! But how to hide that checked exception thrown by the JDK class? We could expand line #7 into an inline lambda, and, then, within in that perform the following:

```
1   try {...} catch(IOException) {throw new RuntimeException(e);}
```

However, reading this would hurt our eyes. We should probably immediately bury it in some method, or we could let jOOL do that for us. Line #7 then becomes:

```
1   .forEach(Unchecked.consumer(writer::write));
```

And we go home happy ...

```
.forEach(Unchecked.consumer(writer::write));
```

There are many ways you could stray from the path of righteousness while doing that, including booleans, enum ExportType, and @Overriding concrete methods (I mention some of those in my talk, just for fun), but I will sketch here an application of the Template Method design pattern [GoF].

```
abstract class FileExporter {
	public File exportFile(String fileName) throws Exception {
		File file = new File("export/" + fileName);
		try (Writer writer = new FileWriter(file)) {
			writeContent(writer);
			return file;
		} catch (Exception e) {
			// TODO send email notification
			throw e;
		}
	}
	protected abstract void writeContent(Writer writer) throws IOException;
}

class OrderExporter extends FileExporter{
	private OrderRepo repo;
	@Override
	protected void writeContent(Writer writer) throws IOException {
		writer.write("OrderID;Date\n");
		repo.findByActiveTrue()
			.map(o -> o.getId() + ";" + o.getCreationDate())
			.forEach(Unchecked.consumer(writer::write));
	}
}
class UserExporter extends FileExporter {
	@Override
	protected void writeContent(Writer writer) throws IOException {
		...
	}
}
```

I want you to ask yourself: *why* did we use that dangerous word there? Why did we play with fire? What's the excuse for that awful `extends` in the code? To force me to provide the missing logic, there will be a function `f(Writer):void` whenever I subclass the `FileExporter`.

But, we can do that a lot easier in Java 8! We just need to take a `Consumer<Writer>` as a method parameter!

```
class FileExporter {
        public File exportFile(String fileName, Consumer<Writer> contentWriter) throws Ex
                File file = new File("export/" + fileName);
                try (Writer writer = new FileWriter(file)) {
                        contentWriter.accept(writer);
                        return file;
                } catch (Exception e) {
                        // TODO send email notification
                        throw e;
                }
        }
}

class OrderExportWriter extends FileExporter{
        private OrderRepo repo;
        public void writeOrders(Writer writer) throws IOException {
                writer.write("OrderID;Date\n");
                repo.findByActiveTrue()
                        .map(o -> o.getId() + ";" + o.getCreationDate())
                        .forEach(Unchecked.consumer(writer::write));
        }
}
class UserExportWriter extends FileExporter {
        protected void writeUsers(Writer writer) throws IOException {
                ...
        }
}
```

Wow, a lot of things changed here. Instead of `abstract` and `extends`, the `exportFile()` function got a new `Consumer<Writer>` parameter, which it calls to write the actual export content. To get the whole picture, let's sketch the client code:

```
fileExporter.exportFile("orders.csv", Unchecked.consumer(orderWriter::writeOrders));
fileExporter.exportFile("users.csv", Unchecked.consumer(userWriter::writeUsers));
```

Here, I had to use `Unchecked` again to make it to compile, because `writeOrders()` declared it throws an exception! Alternatively, if you were a Lombok fan, you could drop a `@SneakyThrows` on the `writeOrders()`, and, then, simply delete the throws clause from it. You can try it yourself. I pushed all the steps (including this one) on this dedicated GitHub repository. I won't debate if it's a good practice, I'm just playing around. Also, please note that, to try it out, you will have to configure Lombok on your IDE.

The fundamental idea is that **whenever you have some 'variable logic', you can consider taking it as a method parameter**. In my training, I call this the ***Passing-a-Block*** pattern. The above example, however, is a slight variation, in which the function given as a parameter works with a resource that is managed by the host function. In our example, `OrderExportWriter.writeOrders` receives a `Writer` as a parameter to write the content

to it. However, `writeOrders` is not concerned with creating, closing, or handling errors related to the `FileWriter`. That's why we call this **the Loan pattern. This is a function we pass in that is essentially borrowed so the Writer can do its job**.

One major benefit of the *Loan pattern* is that it decouples nicely with the infrastructural code (`FileExporter`) from the actual export format logic (`OrderExportWriter`). Through a better separation by layers of abstraction, this pattern enables a *Dependency Inversion*, i.e. you could keep the `OrderWriter` in a more interior layer. Because it decouples the code so nicely, the design becomes a lot easier to reason with and unit test. You can test `writeOrders()` by passing it a `StringWriter` and, afterward, see what was written to it. To test the `FileExporter`, you can pass simply a dummy `Consumer` that just writes "dummy", and then verify that the infra code did its job.

And, this is all done without any extends. Extending to reuse logic is known to doing long-term damage to the design. Therefore, the elegant way we can use the *Passing-a-Block* pattern in Java 8 could be witnessing the funeral of the template method design pattern, because it achieves mostly the same goal without forcing you to extend anything.

There is one more variation of the *Passing-a-Block* pattern, the *Execute Around* pattern. Syntactically, the code is very similar:

```
1  measure(() -> stuff());
2  executeInTransaction(() -> stuff());
```

However, the purpose is slightly different. Here, `stuff()` was already implemented, but, afterward, we wanted to execute some arbitrary code around it (before and after). **With *Execute Around,* we write this before/after code in some helper function and then wrap our original call within a call to this helper**. If we look closely at the second line, it smells like Aspect-Oriented Programming (AOP) right? With Spring, we normally just put `@Transactional` on a method to get a transaction for this method. For the TransactionInterceptor to come into play, however, you need to call the `@Transactional` method on a (proxied) reference to this class provided by Spring, which is not always desirable. So, both the above two lines refer to the case in which you require some ad-hoc weaving, that is, to run your function within some arbitrary utility function and to do so before and/or after your code.

Disclaimer: The pattern names are used interchangeably in articles you can read on the internet.

The key takeaway of this section is that **you should force yourself into thinking about handling bits of logic and juggling them as first-class citizens in Java 8**.

It will make your code more elegant, simple, and expressive.

5) Five Ways to Implement Type-Specific Logic

The task is simple: there are three movie types, each with its own formula for computing the price based on the loaned number of days.

```
1  class Movie {
2          enum Type {
3                  REGULAR, NEW_RELEASE, CHILDREN
4          }
```

```
5
6           private final Type type;
7
8           public Movie(Type type) {
9                   this.type = type;
10          }
11
12          public int computePrice(int days) {
13                  switch (type) {
14                  case REGULAR: return days + 1;
15                  case NEW_RELEASE: return days * 2;
16                  case CHILDREN: return 5;
17                  default: throw new IllegalArgumentException(); // Always have this here!
18                  }
19          }
20  }
```

If we test:

```
1   System.out.println(new Movie(Movie.Type.REGULAR).computePrice(2));
2   System.out.println(new Movie(Movie.Type.NEW_RELEASE).computePrice(2));
3   System.out.println(new Movie(Movie.Type.CHILDREN).computePrice(2));
```

we get:

```
1   3
2   4
3   5
```

The example is a distillation of the classic video store coding Kata of Uncle Bob. The problem in the code above could be the `switch`: whenever you add a new value to the enum, you need to hunt down all the switches and make sure you handle the new case. But this is fragile. The `IllegalArgumentException` will pop up, but only if you walk that path from tests/UI/API. In short, although anyone can read this code, it's a bit risky.

One way to avoid the risk would be an OOP solution:

```
1   abstract class Movie {
2           public abstract int computePrice(int days);
3   }
4
5   class RegularMovie extends Movie {
6           public int computePrice(int days) {
7                   return days+1;
8           }
9   }
```

```
10
11  class NewReleaseMovie extends Movie {
12          public int computePrice(int days) {
13                  return days*2;
14          }
15  }
16
17  class ChildrenMovie extends Movie {
18          public int computePrice(int days) {
19                  return 5;
20          }
21  }
```

If you create a new type of movie, a new subclass actually, the code won't compile unless you implement `computePrice()`. But, it `extends` again! What if you want to classify the movies by another criterion, say release year? Or how would you handle the 'downgrade' from a `Type.NEW_RELEASE` to a `Type.REGULAR` movie after several months?

Let's revert to the first form and let's look for other ways to implement this. In my Devoxx talk, I also live-code how to implement this logic using abstract methods on enums. But, here, I'd like to directly throw in the change request: "*the factor in the price formula for new release movies (in our example was 2) must be updatable via the database.*"

Auch! This means that I have to get this factor from some injected repository. But, since I can't inject repos in my `Movie` entity, let's move the logic to a separate class:

```
1   public class PriceService {
2
3           private final NewReleasePriceRepo repo;
4
5           public PriceService(NewReleasePriceRepo repo) {
6                   this.repo = repo;
7           }
8
9           public int computeNewReleasePrice(int days) {
10                  return (int) (days * repo.getFactor());
11          }
12          public int computeRegularPrice(int days) {
13                  return days + 1;
14          }
15          public int computeChildrenPrice(int days) {
16                  return 5;
17          }
18
19          public int computePrice(Movie.Type type, int days) {
20                  switch (type) {
21                  case REGULAR: return computeRegularPrice(days);
```

```
                case NEW_RELEASE: return computeNewReleasePrice(days);
                case CHILDREN: return computeChildrenPrice(days);
                default: thrownew IllegalArgumentException();
                }
        }
}
```

But, the `switch` is back with the inherent risks! Is there any way to make sure noone forgets to define the associated price formula?

And, now, for the grand finale:

```
public class Movie {

        public enum Type {
                REGULAR(PriceService::computeRegularPrice),
                NEW_RELEASE(PriceService::computeNewReleasePrice),
                CHILDREN(PriceService::computeChildrenPrice);

                public final BiFunction<PriceService, Integer, Integer> priceAlgo;

                private Type(BiFunction<PriceService, Integer, Integer> priceAlgo) {
                        this.priceAlgo = priceAlgo;
                }
        }
        ...

}
```

And, instead of the `switch`:

```
class PriceService {
        ...
        public int computePrice(Movie.Type type, int days) {
                return type.priceAlgo.apply(this, days);
        }
}
```

?!!?!

I am storing into each enum value a method reference to the corresponding instance method from `PriceService`. Since I refer to instance methods in a static way (from `PriceService::`), I will need to provide the `PriceService` instance as the first parameter at the invocation time. And I give it `this`. This way, I can effectively reference methods from any [Spring] bean from the static context of the definition of an enum value.

In the various Coding Dojos I held, developers also proposed `Map<Type,Function<>>`, but compared to an old-

In the various Coding Dojos I held, developers also proposed `map<type,function>`, but compared to an old school `switch`, it doesn't have any real benefit – the compilation still doesn't break if you add a new movie type, but the code becomes more esoteric.

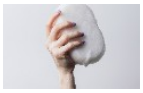
Conclusion

In short:

- Always extract complex lambdas into named functions that you reference using `::`. You should NEVER write `-> {`.
- Avoid excessive call chaining — break them up using explanatory methods and variables, especially if the return type varies across these calls.
- Whenever `null` annoys you, think about using the `Optional`. Twist your mind — you will have to apply functions to the magic box.
- Realize when the variable thing is a function, and you work with that explicitly, pass a function to another function.
- Loan Pattern means to have the function you give as the parameter that you use for a resource managed by the 'host' function. This leads to conceptually lighter, loosely coupled, and easy to test design.
- Sometimes, you might want to have some arbitrary code to execute around another function. If that is the case, pass that code to the function as a parameter.
- You can hook type-specific logic to your enums using method references to make sure each enum value is associated with a corresponding bit of logic.

Always aim for the simplest possible code and aggressively distill your code until it's trivial/"boring."

Like This Article? Read More From DZone



Lambdas and Clean Code

Java 8: The Bad Parts

Lambda Expressions in Java 8

Free DZone Refcard

Getting Started With Kotlin

Topics: JAVA 8 , FUNCTIONAL PROGRAMING , DESIGN PATTERNS , CLEAN CODE , REFACTORING , SIMPLICITY , LAMBDA EXPRESSION , STREAM API