The Builder Pattern in Java, and Dart Cascades

dev.to/jvarness/the-builder-pattern-in-java-and-dart-cascades-5l7

Object construction is something that everyone will have to do in a language that has object-oriented paradigms. However, when your objects have a lot of members and are subject to change, how do you create a robust API that protects your consumers from non-passive changes? How do you avoid having multiple constructors that allow your users to construct the objects differently (what I refer to as *constructor hell*)?

An Example

Let's say that you're opening up your own pizza chain, and you want to write a Java application that allows users to create their own pizza.

The most logical thing for you to do is to create a `Pizza` class that allows you to encapsulate the concept of a pizza:

```
public class Pizza {
   private Collection<String> toppings;
   private String sauce;
   private boolean hasExtraCheese;
}
```

This looks ok right? And we can create getter and setter methods for these members that allow us to alter the state of the object:

```
//... other code

public Collection<String> getToppings {
   return this.toppings;
}

public void setToppings(Collection<String> toppings) {
   this.toppings = toppings;
}

// ... other setters and getters
```

Not too bad... But how does one construct one of these things?? Well, the simplest answer would be to set everything by hand:

```
final Pizza pizza = new Pizza();
pizza.hasExtraCheese(true);
pizza.setSauce("garlic");
List<String> toppings = new ArrayList<String>();
toppings.add("pepperoni");
pizza.setToppings(toppings);
```

Which isn't too bad... But that's a lot of code... We could create a constructor:

```
public Pizza(Collection<String> toppings, String sauce, boolean hasExtraCheese) {
   // and then you set stuff...
}
```

Which makes the code look more like this:

```
List<String> toppings = new ArrayList<String>();
toppings.add("pepperoni");
final Pizza pizza = new Pizza(toppings, "marinara", false);
```

Which isn't too bad... But what if I don't care about specifying if I need extra cheese? And maybe it would be convenient to provide a means of constructing a pizza with a default sauce. At this point, you might be tempted to do the following:

```
public Pizza(Collection<String> toppings) {
   // default the sauce and extra cheese
}

public Pizza(String sauce, boolean hasExtraCheese) {
   // default toppings as empty
}

//... potentially many more constructors
```

You could make so many different constructors as a convenience (anybody who writes Swift get that one?).

Wanna know what makes this constructor hell not so great? *When people start wanting to customize their pizza crust.*

```
public class Pizza {
   private Collection<String> toppings;
   private String sauce;
   private boolean hasExtraCheese;
   private String crust; // OH NO, NEW THING I DIDN'T PLAN FOR!!! WE'RE DOOMED!
}
```

Who wants to go write a dozen constructors to support initializing a `Pizza` with an optional crust? Who wants to go create exponentially more after marketing tells you people want to customize their pizza with sauce drizzles and crust dust as a means to compete with Pizza Hut?

...

Nobody? Cool, let's write a Builder instead.

Builder Pattern

The Builder pattern allows you to *build* objects rather than *construct* them. You provide an API in your builder that allows you to set all of the properties of a `Pizza`, and then the builder will build the object for you:

```
public class PizzaBuilder() {
    private Collection<String> toppings;
    private String sauce;
    private boolean hasExtraCheese;
    private String crust;

    public PizzaBuilder withToppings(Collection<String> toppings) {
        this.toppings = toppings;
        return this;
    }

    // ... create a "with" method for each member you want to set

    public Pizza build() {
        final Pizza pizza = new Pizza();
        // set the pizza properties
        return pizza;
    }
}
```

This makes your `Pizza` creation **much easier**, it ends up **looking cleaner**, it can help make your `Pizza` s **immutable**, and your code is now much more **passive to changes**:

```
final Pizza pizza = new PizzaBuilder()
    .withHasExtraCheese(true)
    .withSauce("marinara")
    .withCrust("pan")
    .withToppings(new ArrayList<String>())
    .build();
```

Now, when people consume your `Pizza` -making API, if you add more functionality, then you won't need to create more constructors, and others won't need to be concerned about implementing the new functionality if they don't have to.

How Dart Addresses This

Dart has some excellent syntax that allows us to skip the creation of builders and prevents us from getting into constructor hell. Let's look at the same `Pizza` class in Dart:

```
class Pizza {
    List<String> toppings;
    String sauce;
    bool hasExtraCheese;
}
```

One cool thing about Dart is that instance variables implement implicit getters and setters. If the instances are final, setters don't get generated.

And we're done! Our consumers can create `Pizza` instances and are already guarded against non-passive changes!

...

No, I'm dead serious. Your job is done. You did the needful. You can go home.

Dart has an excellent feature called cascade notation that allows you to invoke getters, setters, and methods on object instances to instantiate them:

```
// don't mind me, just constructing a pizza...
var pizza = new Pizza()
    ..toppings = ['pepperoni', 'mushrooms']
    ..sauce = 'spaghetti'
    ..hasExtraCheese = true;
```

Looks a lot like a builder, but it really isn't. Now, if we add more instance variables, the above code still works correctly. Our consumers can add crust later if they want, *no reassembly required*.

I hope you enjoyed looking at the builder pattern, and I hope that this has sparked your interest in Dart!

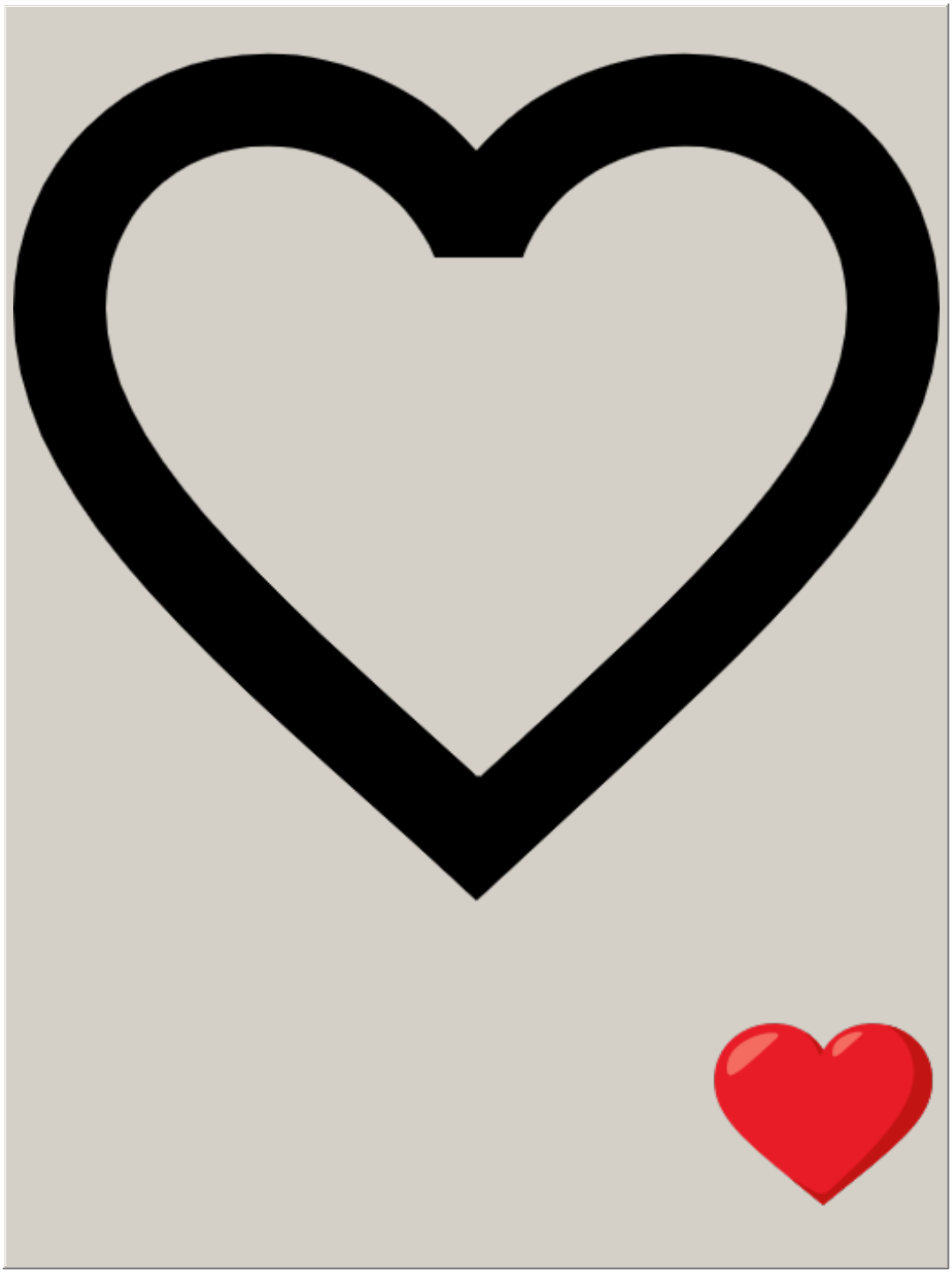
Posted on Dec 6 '17 by:

▼

I probably missed something but how do you make your Pizza immutable in Java? To make all the private attributes of Pizza accessible from PizzaBuilder, they need public setters which in my understanding contradicts immutability.

Well, there is a way to resolve this, which you did not mention in your article: You can make PizzaBuilder an inner class of Pizza. Then is can access the private attributes. It works but I am afraid it often makes your code harder to comprehend.

Another thing that should be mentioned: What you describe is the Builder pattern by Joshua Bloch, not the one by the Gang of Four. There are some similarities between the two as both of them separate object construction from object usage. But basically they are different things.

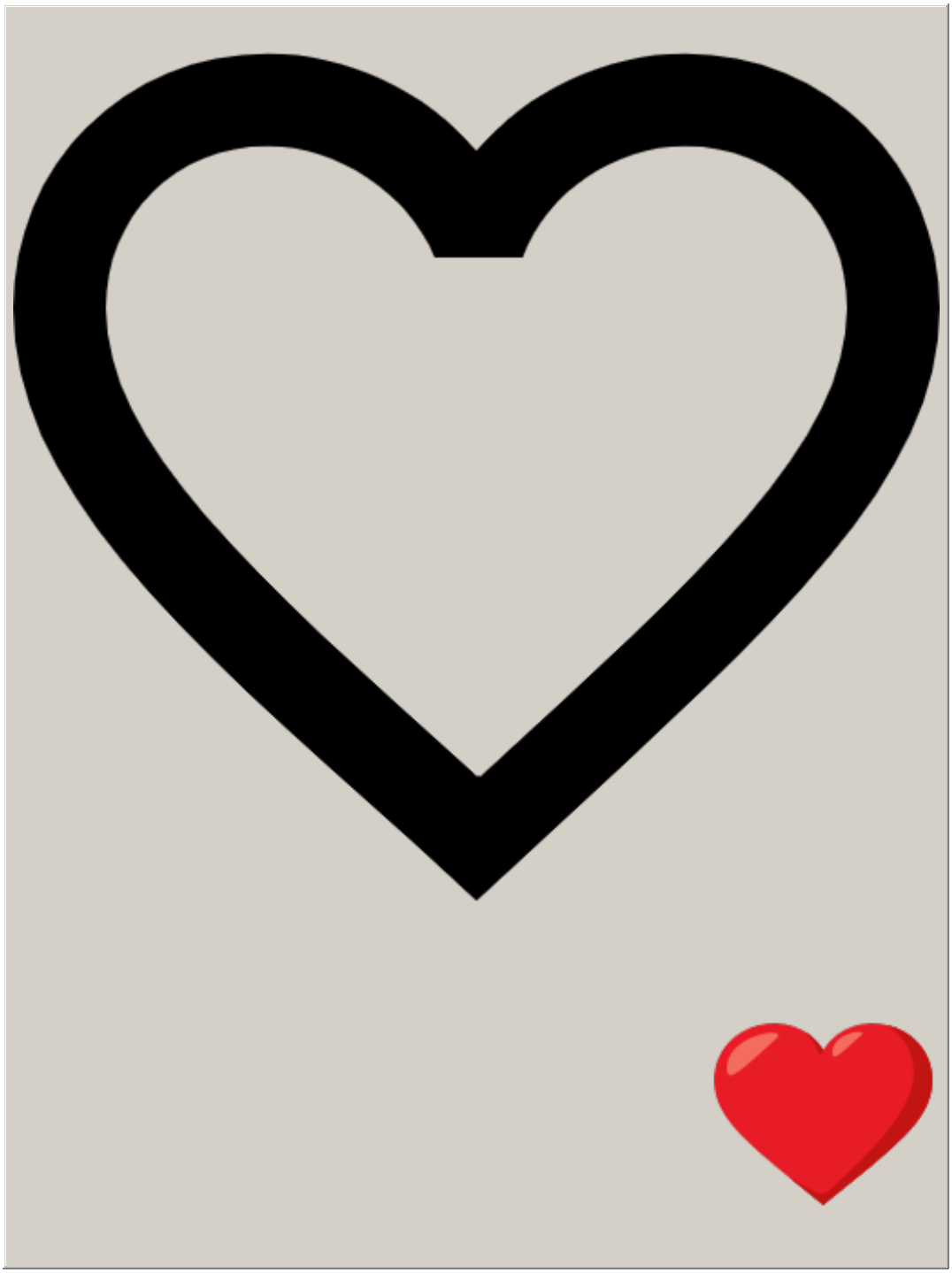


REPLY

▼

Good example. I like the idea but you point out 2 advantages of the builder pattern: First, it is easier to instantiate (or build) objects and second, is object immutability. With the dart cascade option, you lost the object immutability you had in the Java version.

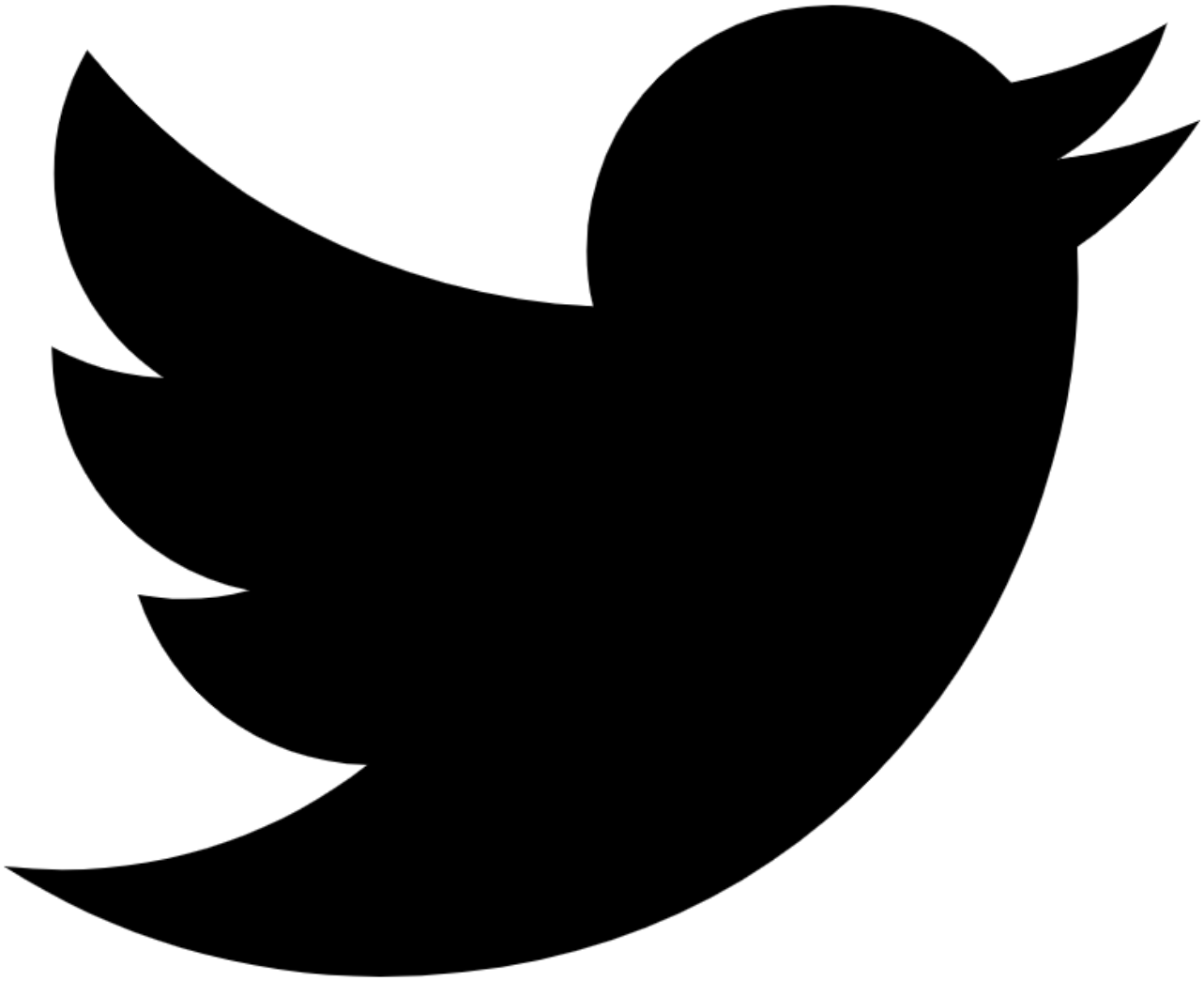
I guess if we want that we have two option: make a builder in Dart or have a mutable class (with setters) subclass the immutable class (with getters) and use the mutable class with the cascade idea but once constructed pass around the superclass. I personally prefer the builder option in this case.



REPLY

▼

 Jake Varness



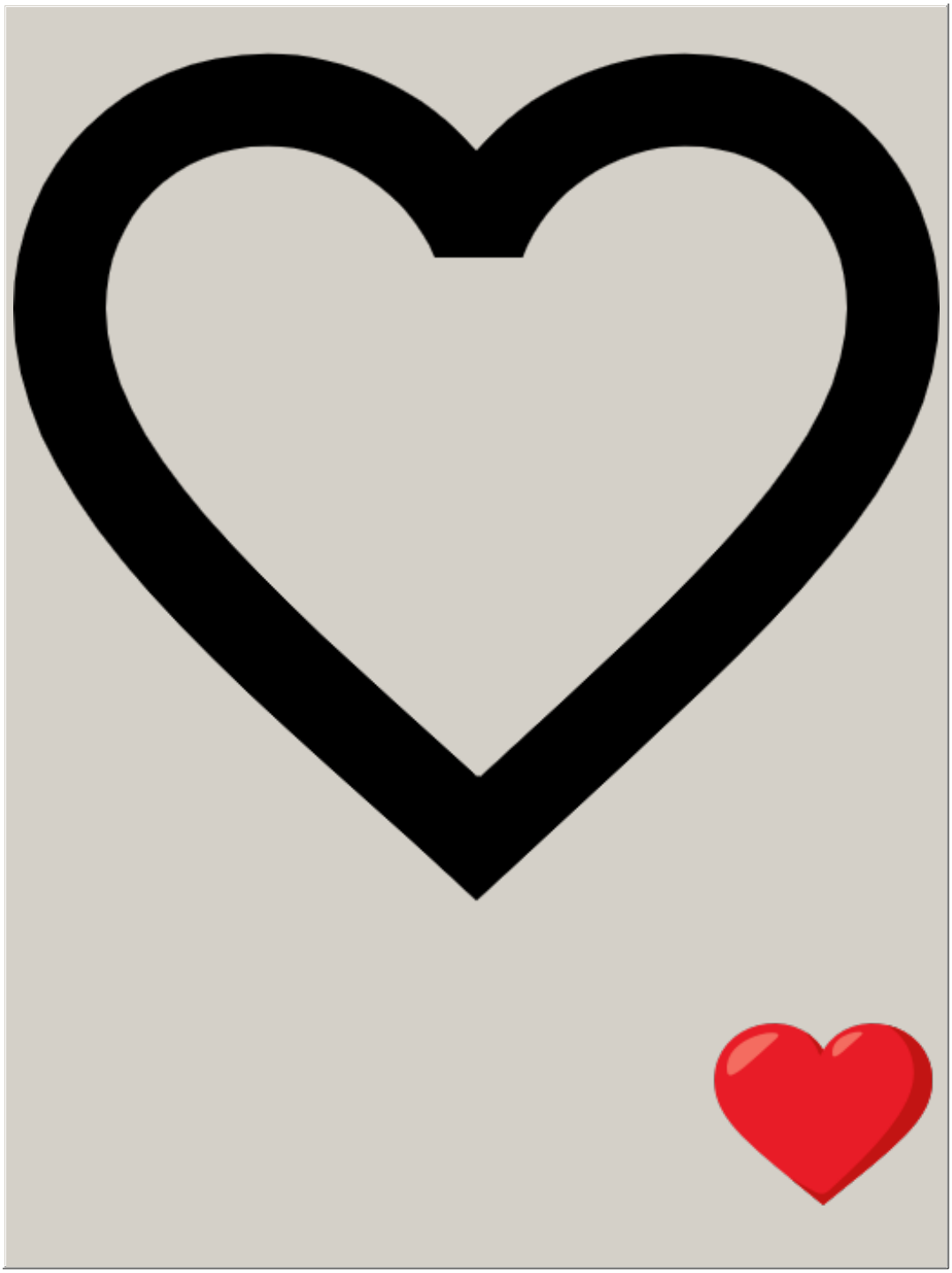
Author

Jun 1 '18

•••

Permalink

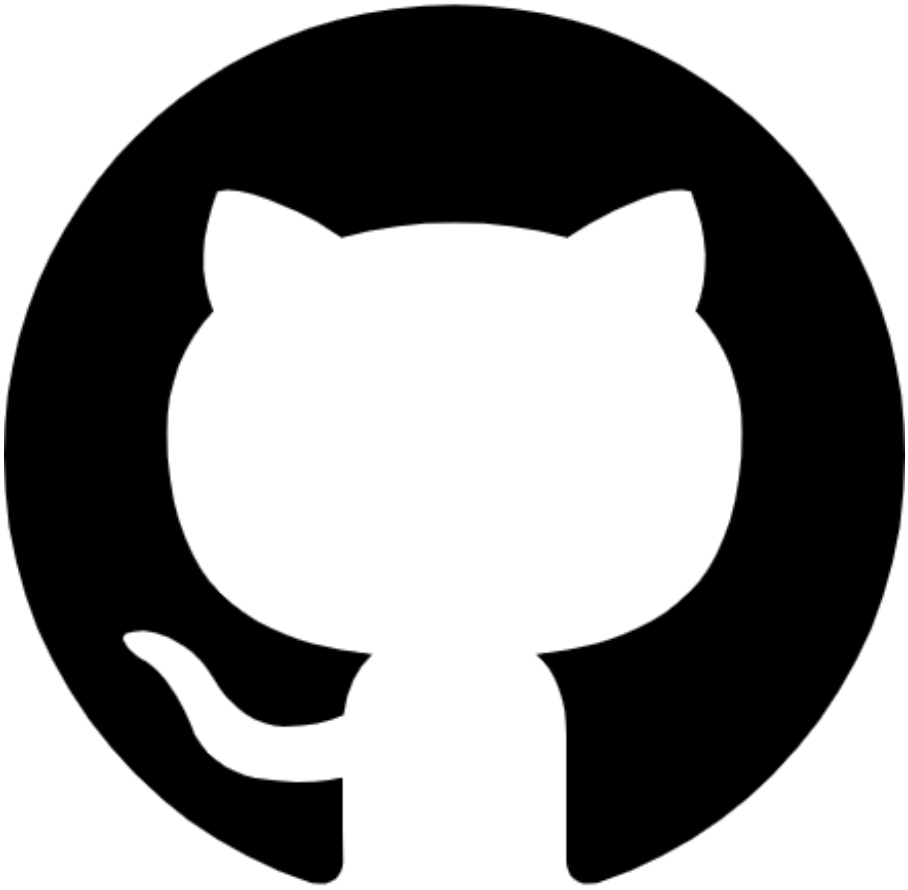
Excellent points Marc. I think with Dart it would be pretty easy to make a builder that creates an immutable object, and cascades can be easily used with a builder!



REPLY

▼

 Marc Guilera



Jun 1 '18

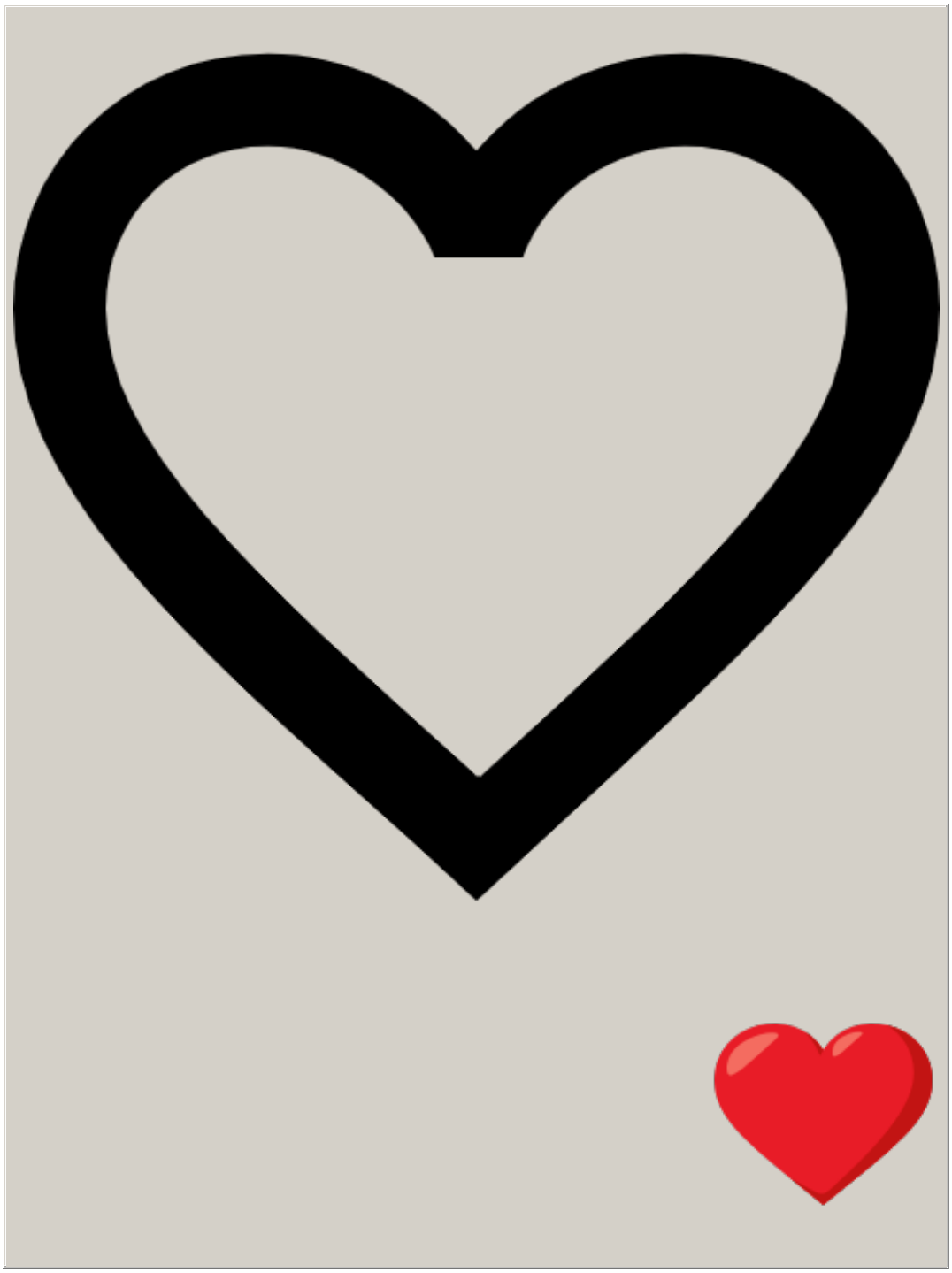
•••

Permalink

Absolutely! I usually create a builder that returns itself on each method to follow the usual builder pattern and make it easier for devs coming from other languages but the consumer can always choose to use cascading instead.

I really value immutability in my code so it's obvious from the API that after instantiation an object is not meant to be changed. Take a DI container for example, after setting it up with an InjectorBuilder (with registerFactory, registorSingleton, etc) I get an Injector that I pass around and only allows gets.

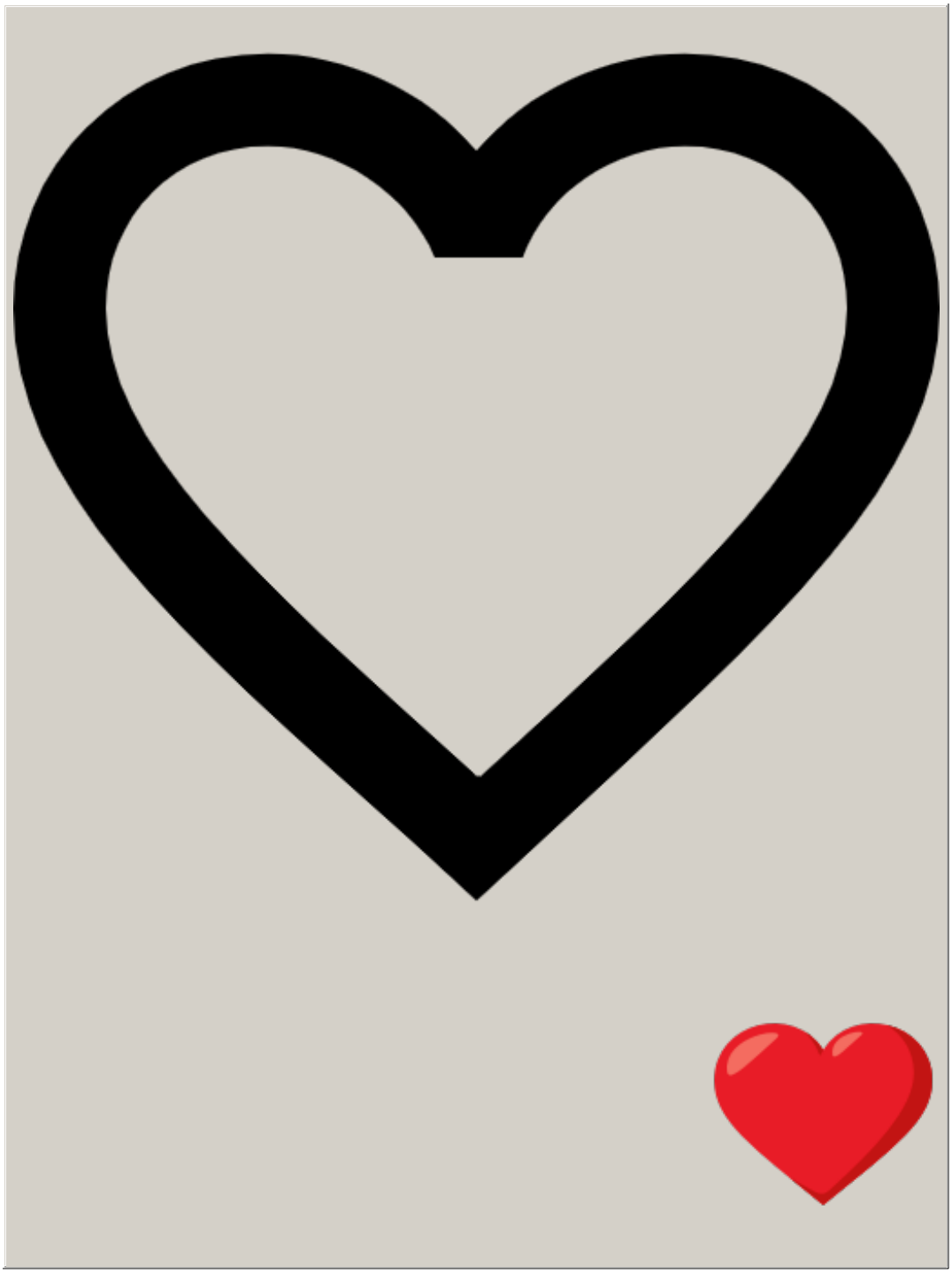
To your point of not having to copy the object to instantiate it from the builder and continuing with the Injector example, I have an Injector interface (or abstract class in Dart) and an InjectorImpl (which implements Injector + the setter methods). Then from the builder, I create a private instance of InjectorImpl which I modify through the builder and then on build return it as an Injector (basically a getter). This is an option if instantiating that object is not that expansive. I do not expose InjectorImpl to the outside world.



REPLY

▼

Even better solution: Allow cascade operators on final parameters during object instantiation as shown in the example and map that to the initialized and final value.



REPLY

▼

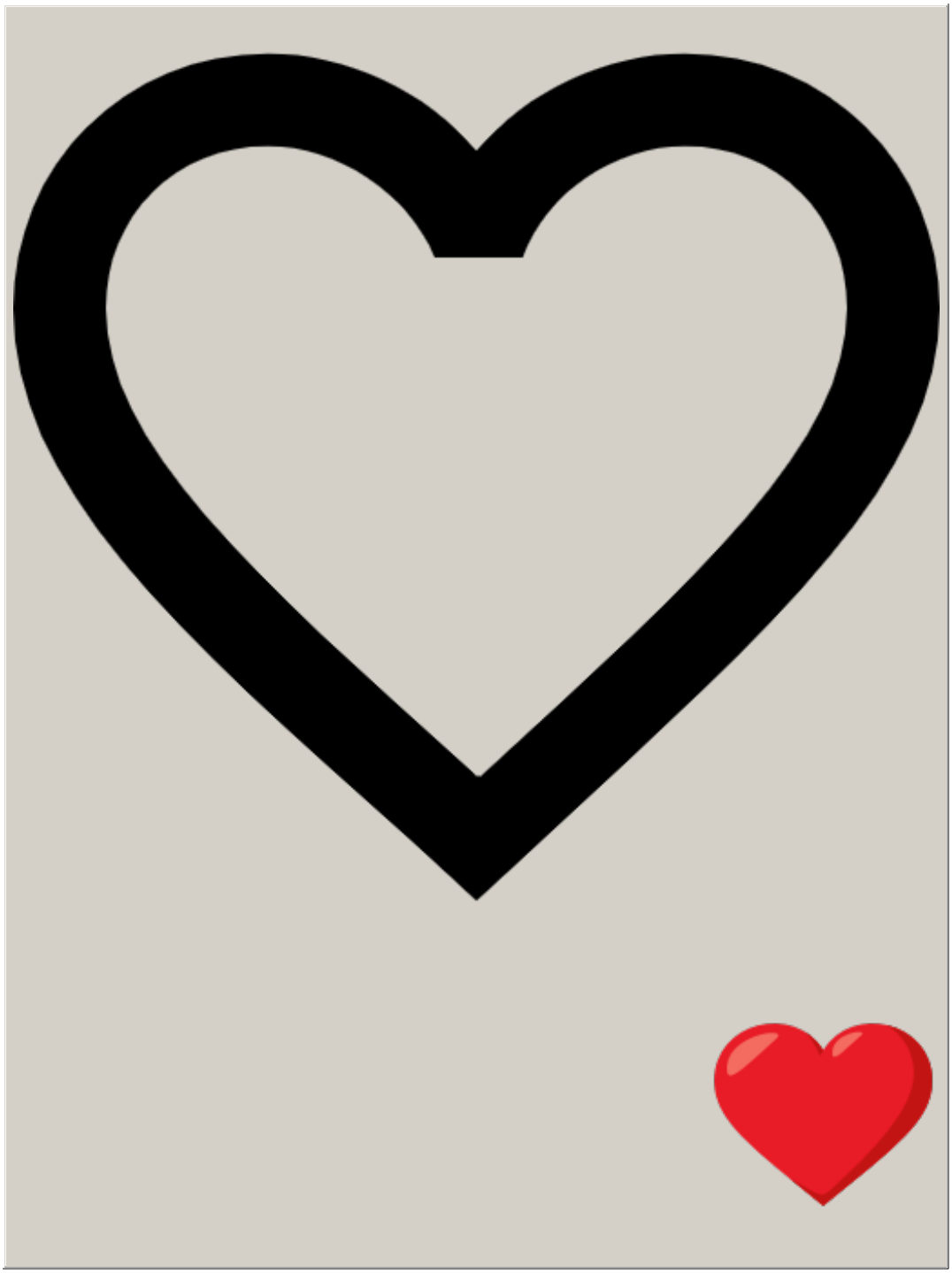
Just a little detail:

Personally, I prefer the following:

Instead of letting the builder call all the getters/setters (or even set the fields) directly, you can create a Pizza constructor that takes a builder instance and gets the values from there (personally, I tend to make the builder a static inner class of the class it instantiates):

private Pizza(PizzaBuilder builder) {...}

This has some slight advantages, most of all that you can keep your Pizza object immutable, if you want to (why would you change the crust after baking it?), because all your members can be final and all your collections can be immutableList/Set/etc.



REPLY

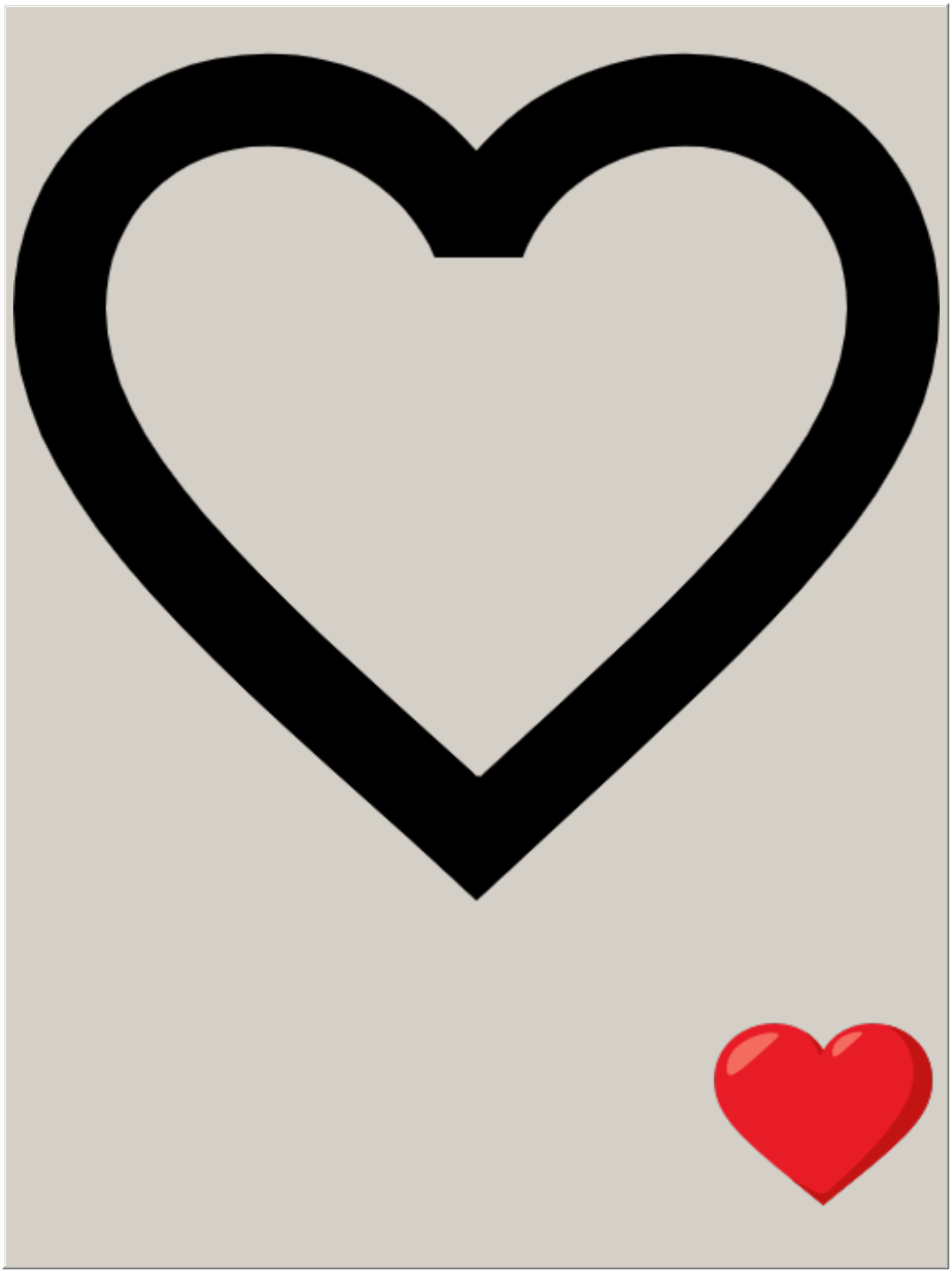
▼

A possible improvement would be adding a named constructor 'Pizza.builder()' which not allow consumers to directly create an instance through the default constructor, giving to the consumer the context that the class is following the builder pattern.

Pizza pizza = Pizza.builder()
..toppings = ['pepperoni', 'mushrooms']
..sauce = 'spaghetti'

..hasExtraCheese = true;

I've written a post following your Pizza example about my own implementation of the Builder pattern :)



REPLY