Converter pattern in Java 8

by Anna Skawińska | FULL BIO

Our Java team has recently prepared a hands-on workshop on functional programming in Java 8. The participants solved our coding tasks, trying out the Java 8's features in separation, and now it's time we show off how we employ the full power of Java 8 in our real-life projects.

This is a common problem of converting pairs of similar objects one to another (in our case – domain classes to DTOs, which are then sent to frontend as JSON objects) and the other way round. In this case, what we need is a one-shot conversion, just once in the object's lifetime. What we do not want is coupling between the two kinds of objects: DTO classes shouldn't "know" about domain classes and the other way round.

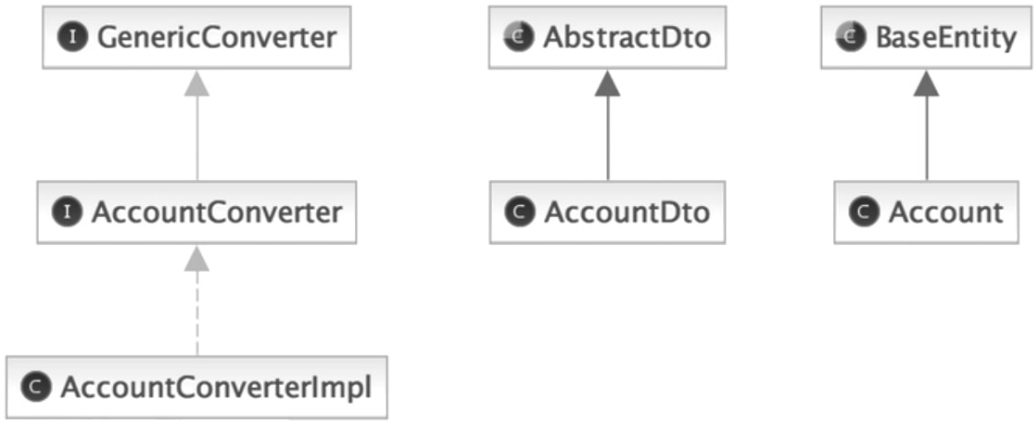
Converter class in Java

As changes in the source object do not have to affect the destination object, the classic solution would be to create a single "Mapper" (or "Converter") class for each pair. There are also tools which are able to map fields of similar classes, based on field names. But how about "producing" whole collections of

What addresses our needs, is Java 8 and its three core features:

- default method implementation in interfaces
- streams
- lambdas (here in the form of a method reference)

Default method implementation is what saves us from boilerplate code, creating collections of objects. Streams and lambdas build a beautiful code transforming our collections. Let's have a look at the final class hierarchy and the code itself:




```
public interface GenericConverter {

   E createFrom(D dto);

   D createFrom(E entity);

   E updateEntity(E entity, D dto);

   default List createFromEntities(final Collection entities) {
       return entities.stream()
               .map(this::createFrom)
               .collect(Collectors.toList());
   }

   default List createFromDtos(final Collection dtos) {
       return dtos.stream()
```

```
    }
}
```

Having implemented the default method that converts a collection of data transfer objects (D) into entities (E), as well as another one that does the opposite, we don't need to implement this in concrete implementations of the converter any more. Creating a converter for a single DTO/domain class is as simple as that:

```
@Component
public class AccountConverterImpl implements AccountConverter {

   @Override
   public Account createFrom(final AccountDto dto) {
       return updateEntity(new Account(), dto);
   }

   @Override
   public AccountDto createFrom(final Account entity) {
       AccountDto accountDto = new AccountDto();
       accountDto.setAccountType(entity.getAccountType());
       accountDto.setActive(entity.getActive());
       accountDto.setEmail(entity.getUserId());
       ClassUtils.setIfNotNull(
              entity::getPassword, accountDto::setPassword);
       return accountDto;
   }

   @Override
   public Account updateEntity(final Account entity,
        final AccountDto dto) {
       entity.setUserId(dto.getEmail());
       entity.setActive(dto.getActive());
       ClassUtils.setIfNotNull(
              dto::getAccountType, entity::setAccountType);
       return entity;
   }
}
```

```
public class ClassUtils {

   protected ClassUtils() { }

   public static  void setIfNotNull(final Supplier getter, final Consume

       T t = getter.get();

       if (null != t) {
           setter.accept(t);
       }
   }
}
```

So there we have a complete Converter structure using all the Java 8 goodness. Adding a new Converter for another entity-DTO pair (like User, Address, etc.) needs just creating a new UserConverterImpl class, implementing its own UserConverter, which in turn should implement GenericConverter. This way the new class will be capable of converting collections of objects out-of-the box. This is possible thanks to the default method implementations in GenericConverter interface, which is a very handy Java 8's feature.

Let us know your thoughts on this solution!

TAGS

Java Java 8