Builder Pattern

Michal Moravik, SD20w2

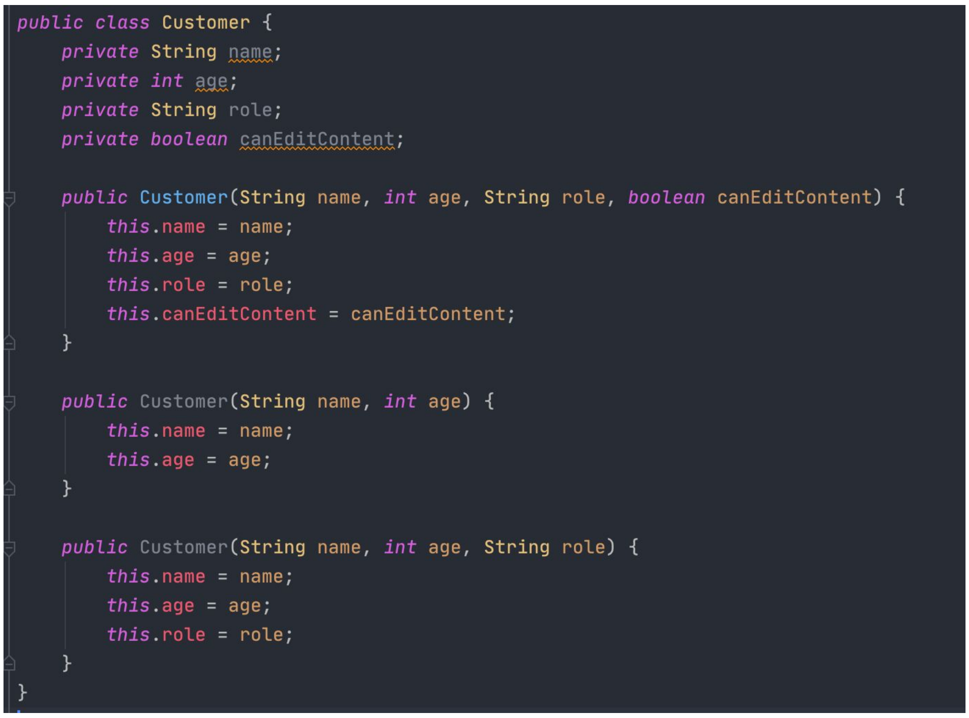
Name: Builder Pattern

Category: Creational pattern

Intent: You want to construct your object by defining the parts that are required (core) and all the rest you want to add by using **method chaining**.

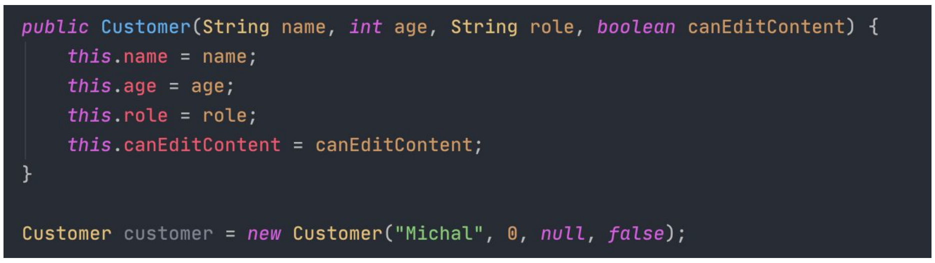
Motivation: Consider the traditional approach based on the following example: We want to create a new customer. But the number of customer's properties should be variable. Sometimes we want to have a customer only with a name, sometimes we require age as well, etc.

In complex classes, this would mean a necessity of creating either multiple different constructors or writing null as a value for the properties we don't want to specify.



```java
public class Customer {
    private String name;
    private int age;
    private String role;
    private boolean canEditContent;

    public Customer(String name, int age, String role, boolean canEditContent) {
        this.name = name;
        this.age = age;
        this.role = role;
        this.canEditContent = canEditContent;
    }

    public Customer(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Customer(String name, int age, String role) {
        this.name = name;
        this.age = age;
        this.role = role;
    }
}
```

Figure 1 - creating lots of different constructors



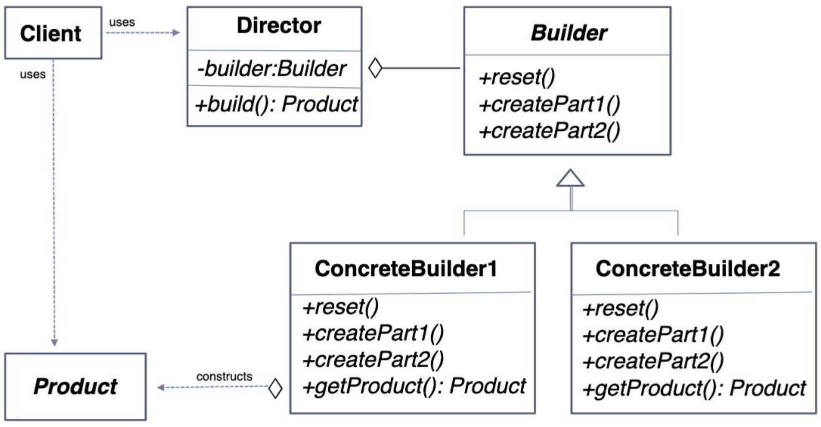
```java
public Customer(String name, int age, String role, boolean canEditContent) {
    this.name = name;
    this.age = age;
    this.role = role;
    this.canEditContent = canEditContent;
}

Customer customer = new Customer("Michal", 0, null, false);
```

Figure 2 - filling properties we don't need with the base values

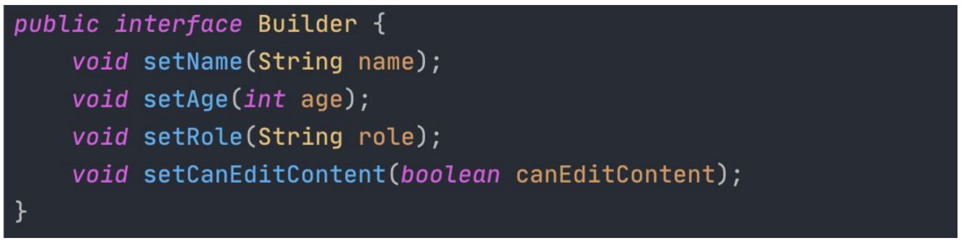
This is, of course, an ugly piece of code. Imagine that we would want to create an object which has a constructor with 20 or 30 different parameters. Then we would need to write nulls, zeros, etc. Moreover, we would use a lot of time just to figuring out which parameter has null and which has a value by looking to the class notation.

Instead, **there is a builder pattern** which provides an option to create an object with a variable number of properties during the instantiation process. (**see implementation part**).

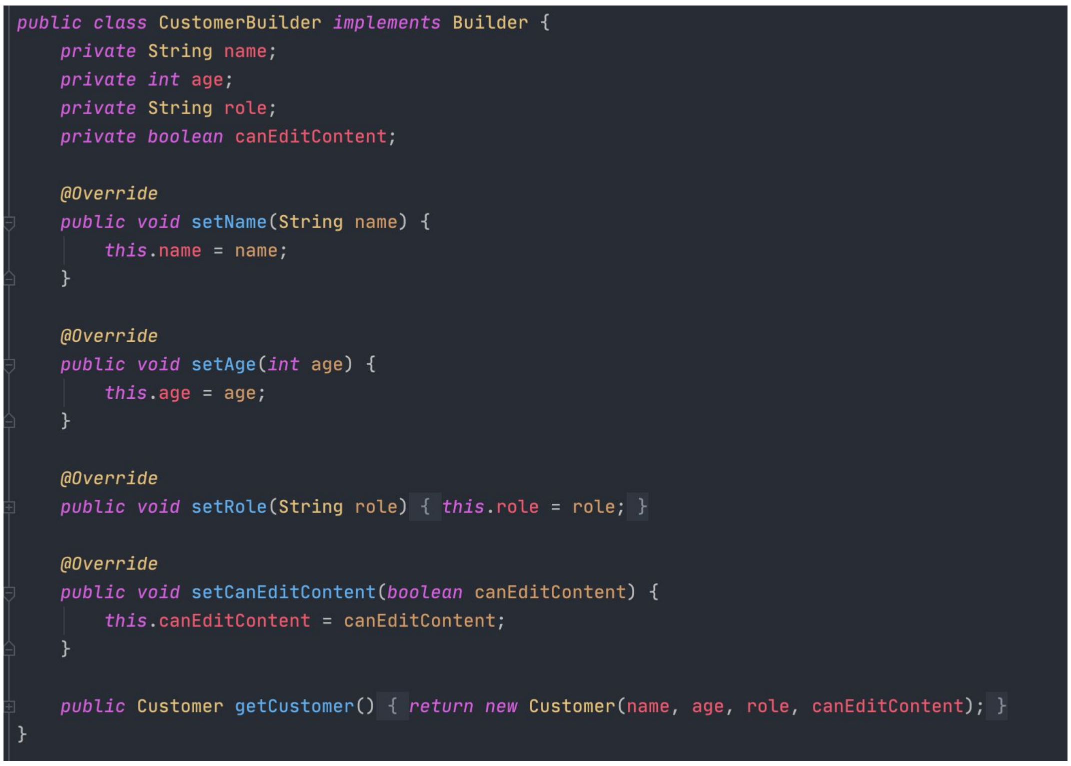
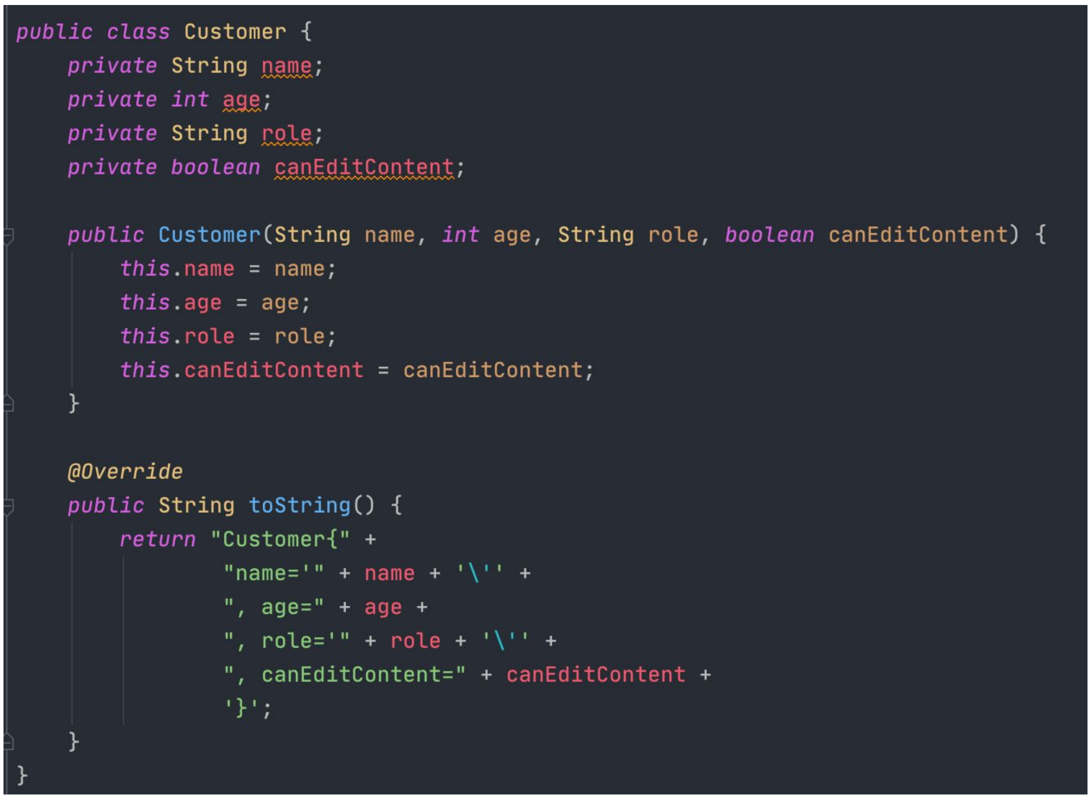
UML:

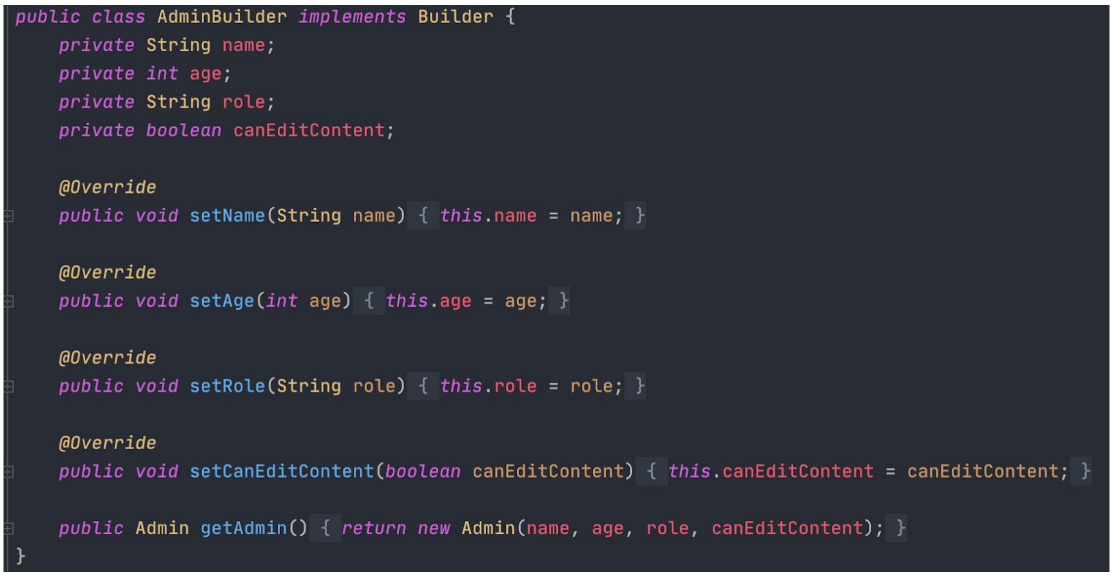


Implementation:

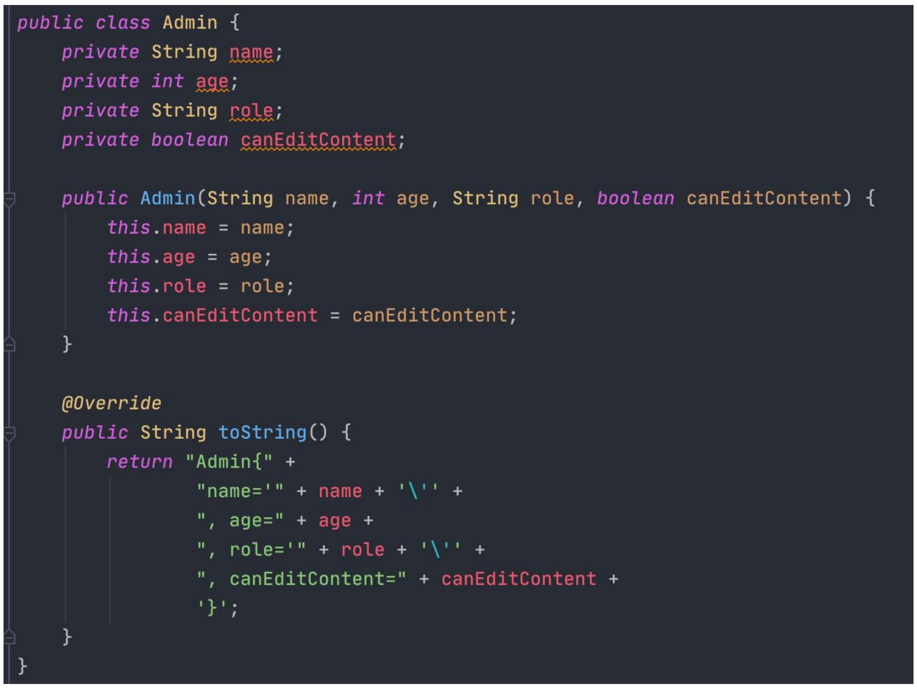


```java
public interface Builder {
    void setName(String name);
    void setAge(int age);
    void setRole(String role);
    void setCanEditContent(boolean canEditContent);
}
```

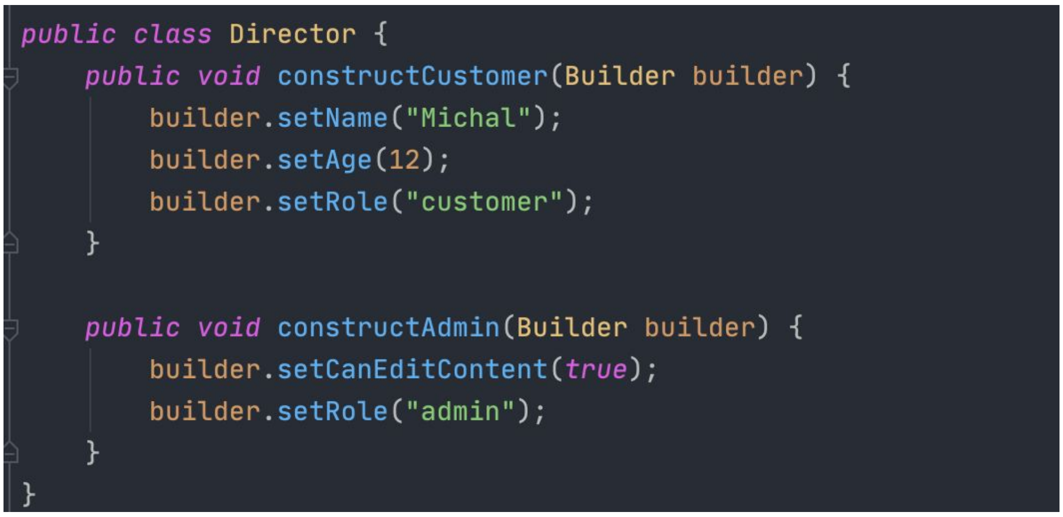

```java
public class CustomerBuilder implements Builder {
    private String name;
    private int age;
    private String role;
    private boolean canEditContent;

    @Override
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public void setRole(String role) { this.role = role; }

    @Override
    public void setCanEditContent(boolean canEditContent) {
        this.canEditContent = canEditContent;
    }

    public Customer getCustomer() { return new Customer(name, age, role, canEditContent); }
}
```

```java
public class Customer {
    private String name;
    private int age;
    private String role;
    private boolean canEditContent;

    public Customer(String name, int age, String role, boolean canEditContent) {
        this.name = name;
        this.age = age;
        this.role = role;
        this.canEditContent = canEditContent;
    }

    @Override
    public String toString() {
        return "Customer{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", role='" + role + '\'' +
                ", canEditContent=" + canEditContent +
                '}';
    }
}
```



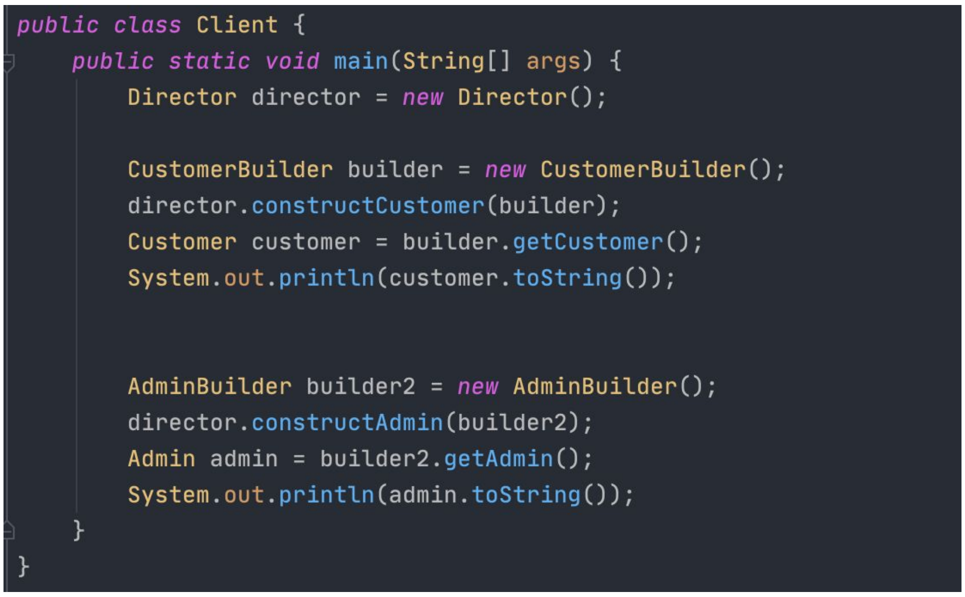
```
public class AdminBuilder implements Builder {
    private String name;
    private int age;
    private String role;
    private boolean canEditContent;

    @Override
    public void setName(String name) { this.name = name; }

    @Override
    public void setAge(int age) { this.age = age; }

    @Override
    public void setRole(String role) { this.role = role; }

    @Override
    public void setCanEditContent(boolean canEditContent) { this.canEditContent = canEditContent; }

    public Admin getAdmin() { return new Admin(name, age, role, canEditContent); }
}
```



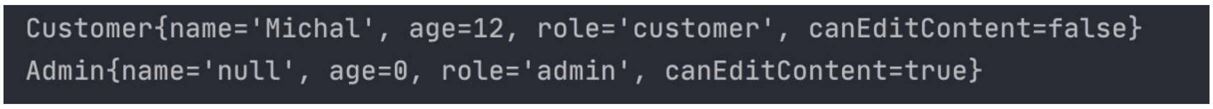
```
public class Admin {
    private String name;
    private int age;
    private String role;
    private boolean canEditContent;

    public Admin(String name, int age, String role, boolean canEditContent) {
        this.name = name;
        this.age = age;
        this.role = role;
        this.canEditContent = canEditContent;
    }

    @Override
    public String toString() {
        return "Admin{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", role='" + role + '\'' +
                ", canEditContent=" + canEditContent +
                '}';
    }
}
```



```
public class Director {
    public void constructCustomer(Builder builder) {
        builder.setName("Michal");
        builder.setAge(12);
        builder.setRole("customer");
    }

    public void constructAdmin(Builder builder) {
        builder.setCanEditContent(true);
        builder.setRole("admin");
    }
}
```



```java
public class Client {
    public static void main(String[] args) {
        Director director = new Director();

        CustomerBuilder builder = new CustomerBuilder();
        director.constructCustomer(builder);
        Customer customer = builder.getCustomer();
        System.out.println(customer.toString());


        AdminBuilder builder2 = new AdminBuilder();
        director.constructAdmin(builder2);
        Admin admin = builder2.getAdmin();
        System.out.println(admin.toString());
    }
}
```



```
Customer{name='Michal', age=12, role='customer', canEditContent=false}
Admin{name='null', age=0, role='admin', canEditContent=true}
```

Consequences:
Pros:
- Straightforward creation of an object - programmers know exactly what they are passing as values
- Minimizing the number of constructors needed
- Helps to understand the object’s properties

Cons:
- Requires code duplication (e.g. Customer and CustomerBuilder)
- It introduces complexity to the system

Known uses:
- In Java core libraries - java.lang.Appendable, java.lang.StringBuffer, ...

Related patterns:
- We could use it for example with the factory and based on a condition, we could call different builders