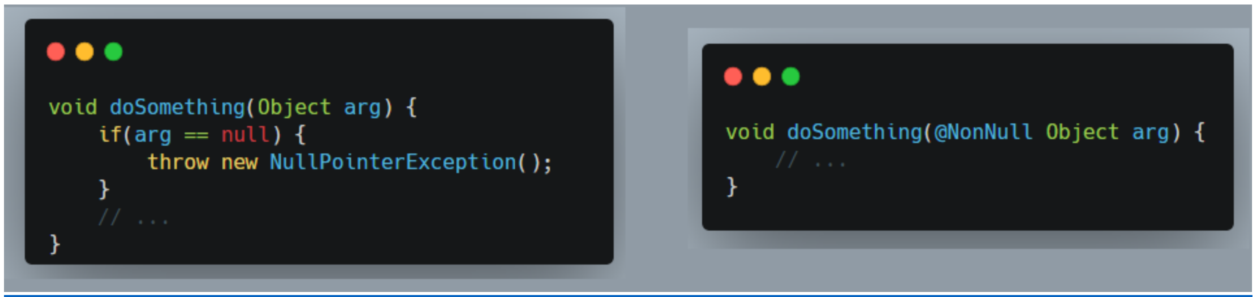
Lombok Annotations

Lombok is a Java library that can generate known patterns of code for us, allowing us to reduce the boilerplate code.

@NotNull

This annotation can be used to validate a constructor or a method parameter. Additionally, if we annotate a field with *@NonNull, the* validation will be added to the constructor and setter method. Let's compare the plain-java solution with Lombok's *@NonNull*:

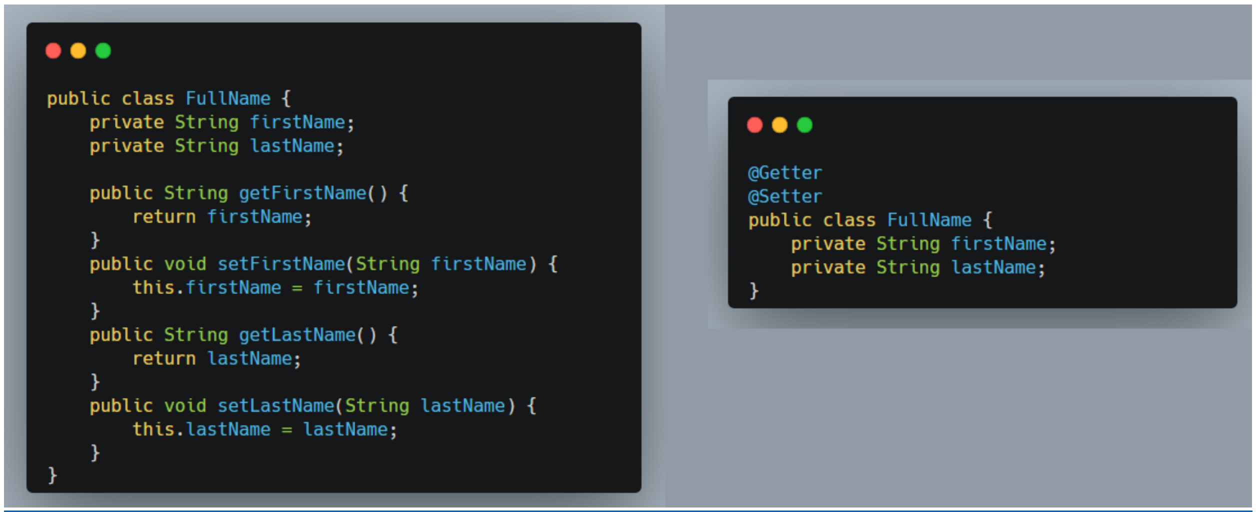


```java
void doSomething(Object arg) {
    if(arg == null) {
        throw new NullPointerException();
    }
    // ...
}
```

```java
void doSomething(@NonNull Object arg) {
    // ...
}
```

@Getter & @Setter

Perhaps the most popular Lombok annotations, @Getter and @Setter can be used to automatically generate getters and setters for all the fields:

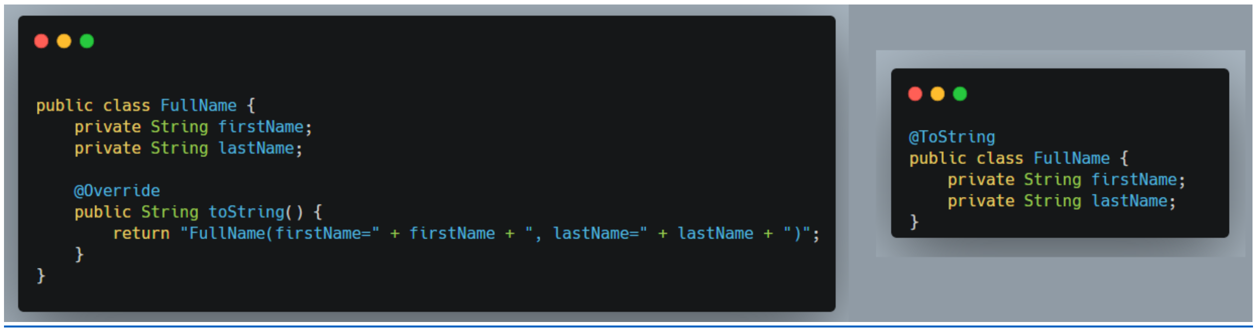


```java
public class FullName {
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

```java
@Getter
@Setter
public class FullName {
    private String firstName;
    private String lastName;
}
```

@ToString

This annotation is going to override the *toString()* method for easy debugging. As a result, the current state of the object will be returned as a *String* when *toString()* will be called:

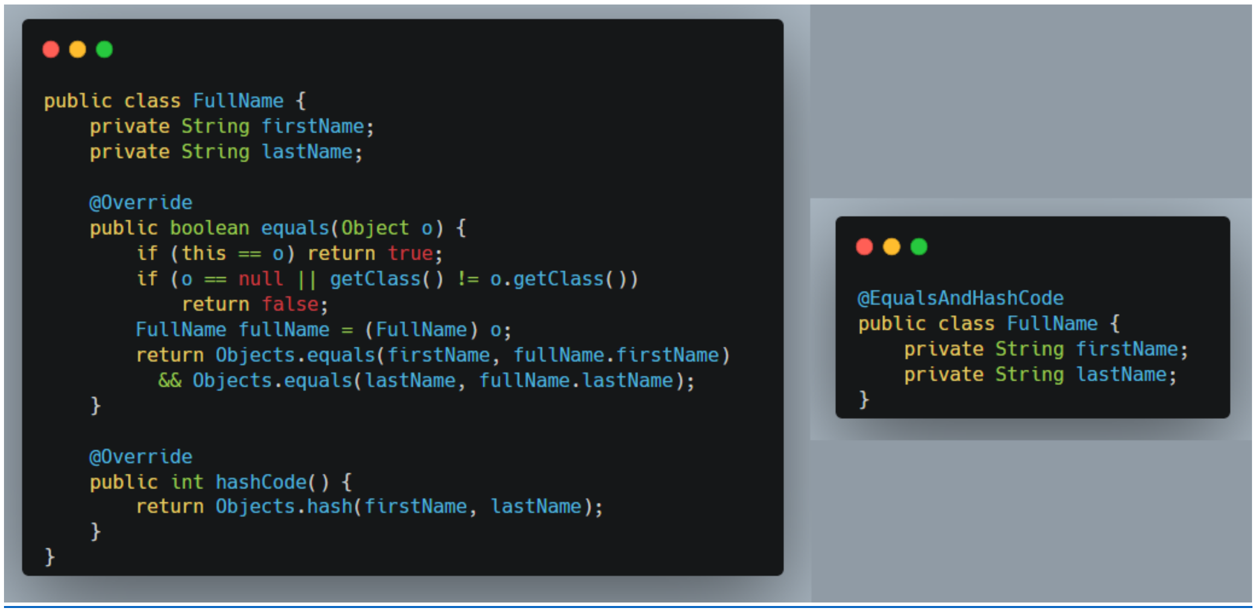


```
public class FullName {
    private String firstName;
    private String lastName;

    @Override
    public String toString() {
        return "FullName(firstName=" + firstName + ", lastName=" + lastName + ")";
    }
}
```

```
@ToString
public class FullName {
    private String firstName;
    private String lastName;
}
```

@EqualsAndHashCode

If we need to override the equals method of a class and compare the fields instead of the reference, we can do it by adding the *@EqualsAndHashCode* annotation. This will also override the *hashcode()* method accordingly:

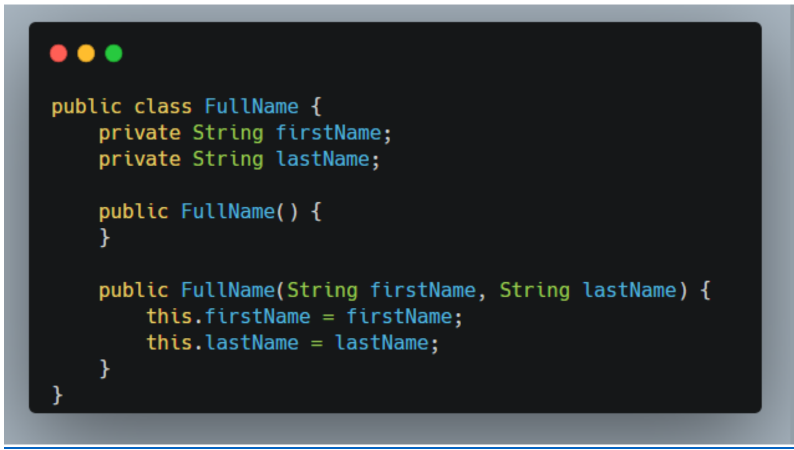


```
public class FullName {
    private String firstName;
    private String lastName;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass())
            return false;
        FullName fullName = (FullName) o;
        return Objects.equals(firstName, fullName.firstName)
          && Objects.equals(lastName, fullName.lastName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }
}
```

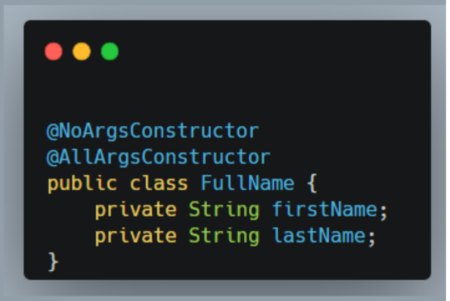
```
@EqualsAndHashCode
public class FullName {
    private String firstName;
    private String lastName;
}
```

@NoArgsConstructor & @AllArgsConstructor

We can easily generate the class constructors with Lombok: if we want a constructor that receives all the fields, we only need to annotate the class with *@AllArgsConstuructor.* On the other hand, if we need the default constructor with no arguments, we need to add *@NoArgsConstructor* as well:



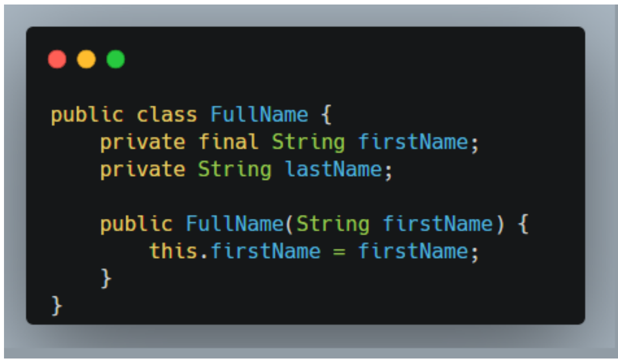
```
public class FullName {
    private String firstName;
    private String lastName;

    public FullName() {
    }

    public FullName(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```



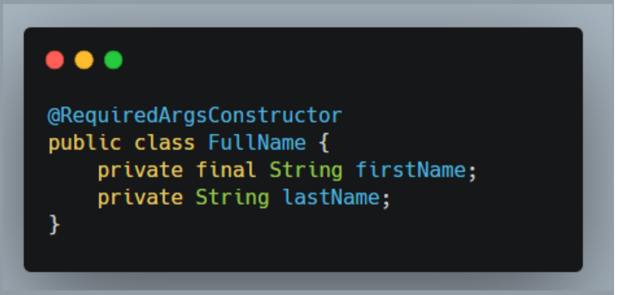
```
@NoArgsConstructor
@AllArgsConstructor
public class FullName {
    private String firstName;
    private String lastName;
}
```

@RequiredArgConstructor

Though, we might only need to define constructors for the fields that are final — after all, the non-final fields can be set later. If this is the case, we should add the *@RequreidArgsConstructor* annotation at a class level:



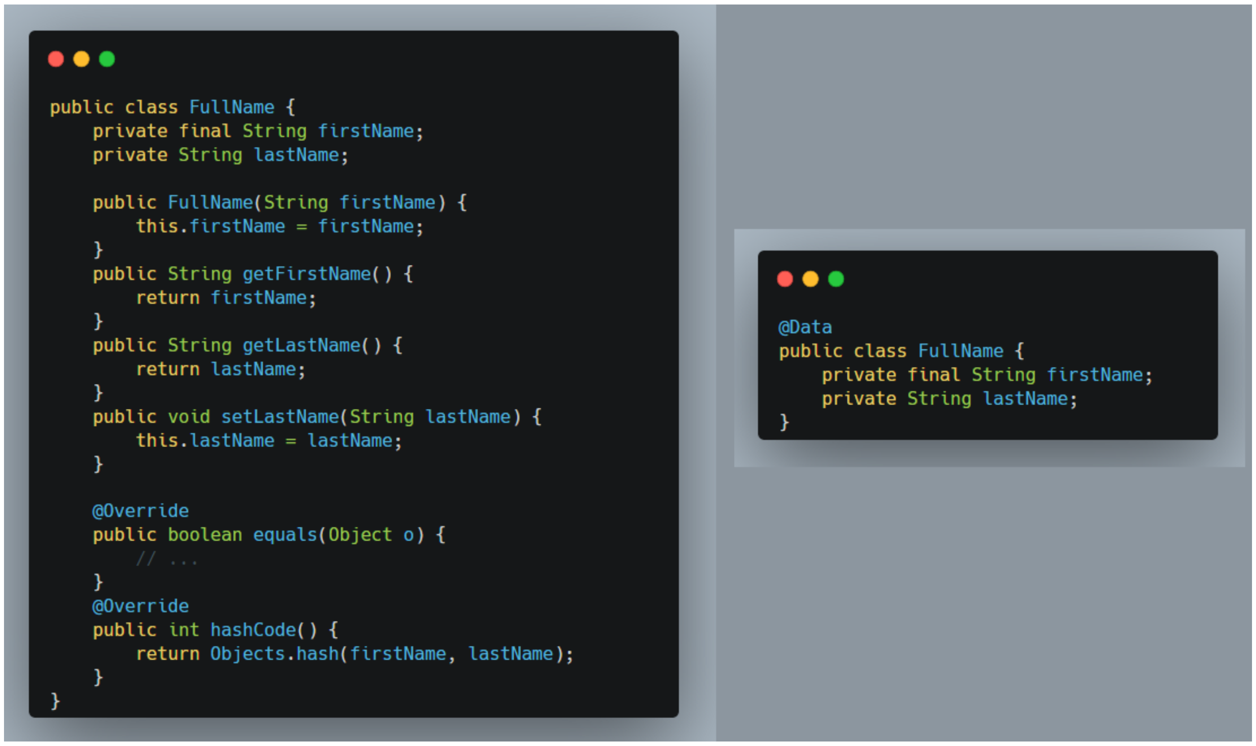
```
public class FullName {
    private final String firstName;
    private String lastName;

    public FullName(String firstName) {
        this.firstName = firstName;
    }
}
```



```
@RequiredArgsConstructor
public class FullName {
    private final String firstName;
    private String lastName;
}
```

@Data

If we have a class that exposes all its fields through getters and setters, we can annotate it with *@Data*. This will be the equivalent of *@Getters, @Setters, @ToString, @EqualsAndHashCode,* and *@ToString:*

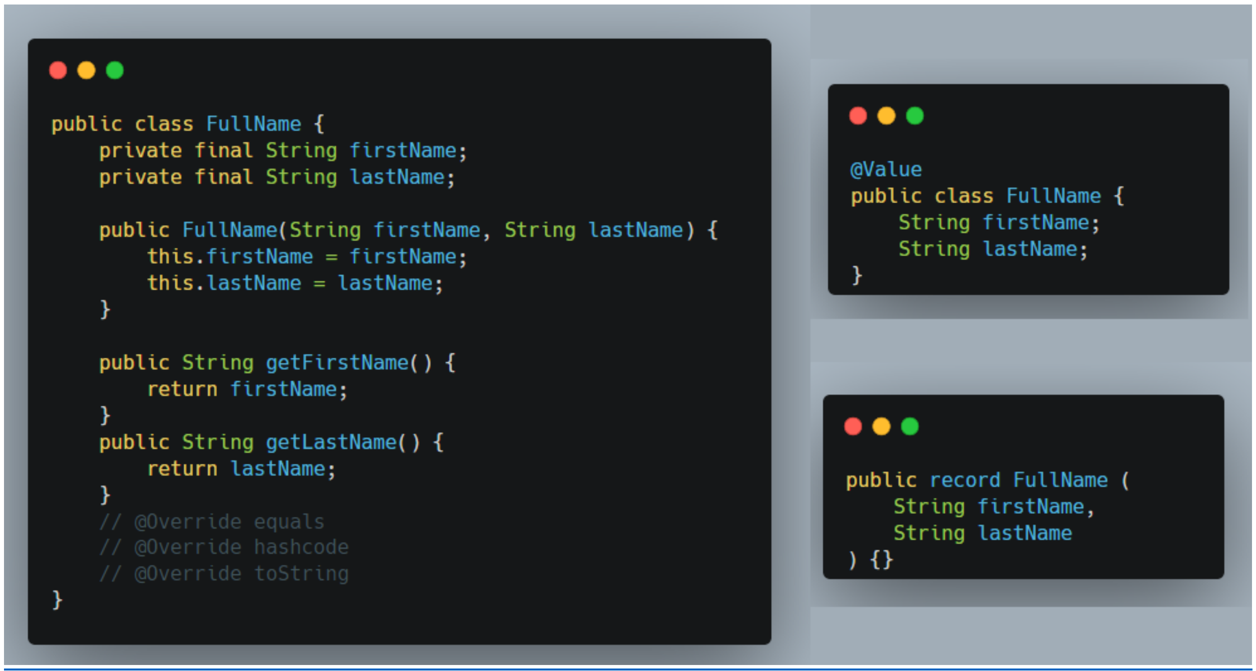


```java
public class FullName {
    private final String firstName;
    private String lastName;

    public FullName(String firstName) {
        this.firstName = firstName;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Override
    public boolean equals(Object o) {
        // ...
    }
    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }
}
```

```java
@Data
public class FullName {
    private final String firstName;
    private String lastName;
}
```

@Value

If we want to work with immutable classes, we can annotate them with *@Value.* As a result, all the fields will be declared *private* and *final,* and will be required by the constructor. Additionally, the *toString(), hascode()* and [*equals()* methods]() will be overridden. Though, if you are using Java17, the *record* type might be a better option:



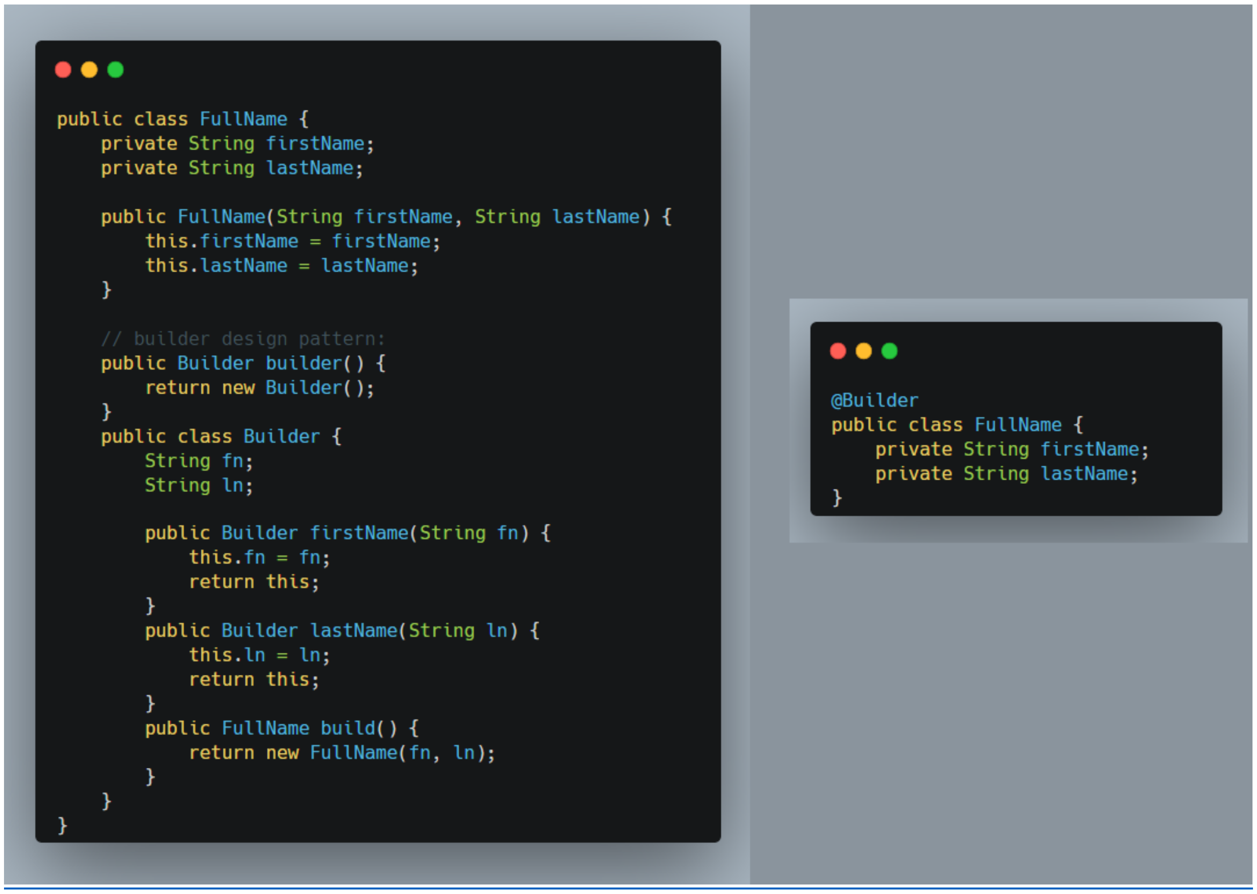
```java
public class FullName {
    private final String firstName;
    private final String lastName;

    public FullName(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    // @Override equals
    // @Override hashcode
    // @Override toString
}
```

```java
@Value
public class FullName {
    String firstName;
    String lastName;
}
```

```java
public record FullName (
    String firstName,
    String lastName
) {}
```

@Builder

We can annotate a class with many fields with *@Builder* and Lombok will inject an implementation of the builder design pattern for us. This can be extremely useful sometimes, especially if some of the fields are nullable:

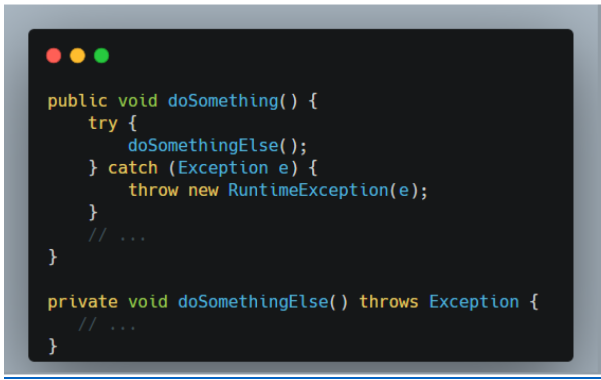


```java
public class FullName {
    private String firstName;
    private String lastName;

    public FullName(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    // builder design pattern:
    public Builder builder() {
        return new Builder();
    }
    public class Builder {
        String fn;
        String ln;

        public Builder firstName(String fn) {
            this.fn = fn;
            return this;
        }
        public Builder lastName(String ln) {
            this.ln = ln;
            return this;
        }
        public FullName build() {
            return new FullName(fn, ln);
        }
    }
}
```

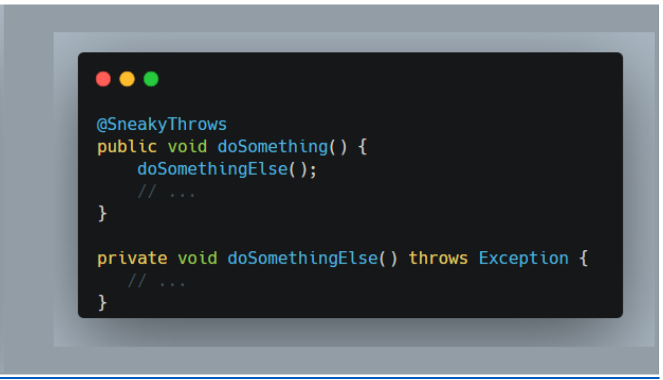
```java
@Builder
public class FullName {
    private String firstName;
    private String lastName;
}
```

@SneakyThrows

One of my favorite Lombok annotations is *@SneakyThrows.* If we annotate a method with this, the code will be surrounded by a try-catch block and the checked exception will be wrapped in a *RuntimeException*. This can be useful when working with Java streams where checked exceptions are not allowed.



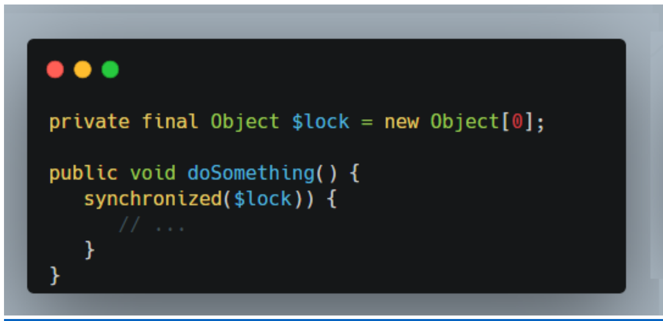
```java
public void doSomething() {
    try {
        doSomethingElse();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
    // ...
}

private void doSomethingElse() throws Exception {
   // ...
}
```



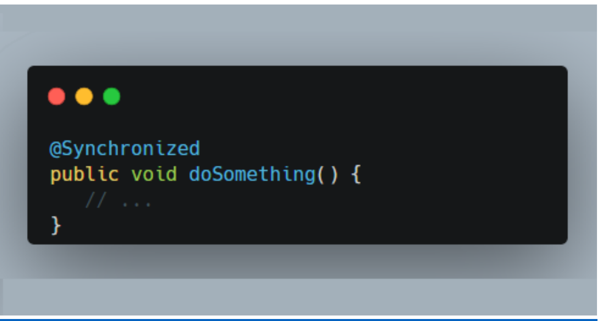
```java
@SneakyThrows
public void doSomething() {
    doSomethingElse();
    // ...
}

private void doSomethingElse() throws Exception {
   // ...
}
```

@Synchronized

If we need to synchronize method calls, we can use Lombok's annotation instead of the *synchronized* Java keyword for a safer implementation:



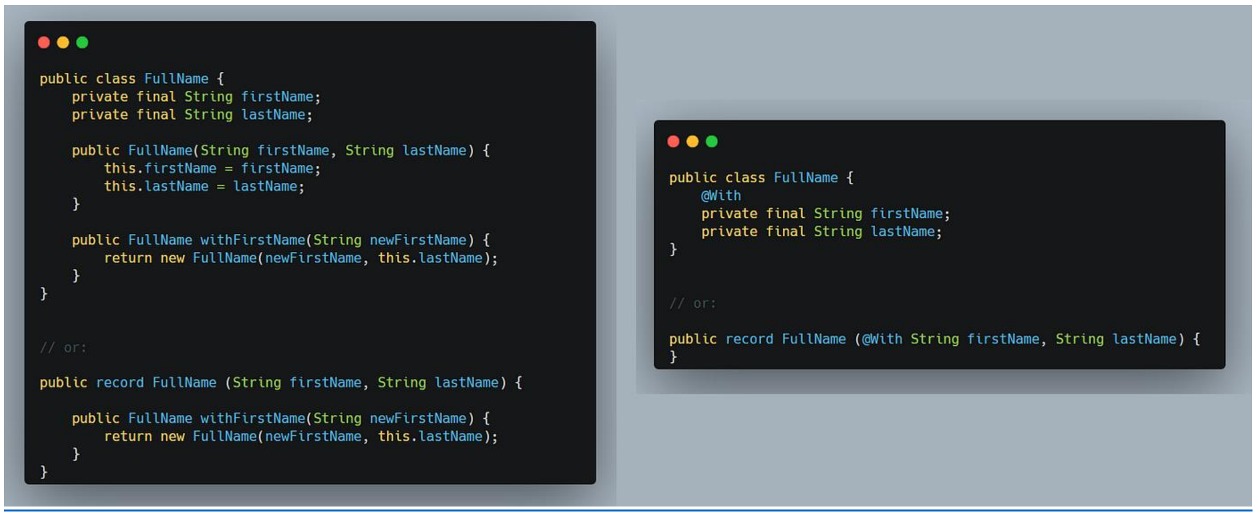
```java
private final Object $lock = new Object[0];

public void doSomething() {
    synchronized($lock)) {
        // ...
    }
}
```



```java
@Synchronized
public void doSomething() {
    // ...
}
```

@With

If we use immutable classes, we can annotate their fields with *@With* and the equivalent of a setter will be generated. The difference is that the *with-er* method will return a completely new instance of the object, with one of the fields updated. This annotation can be extremely useful for Java 17's records:

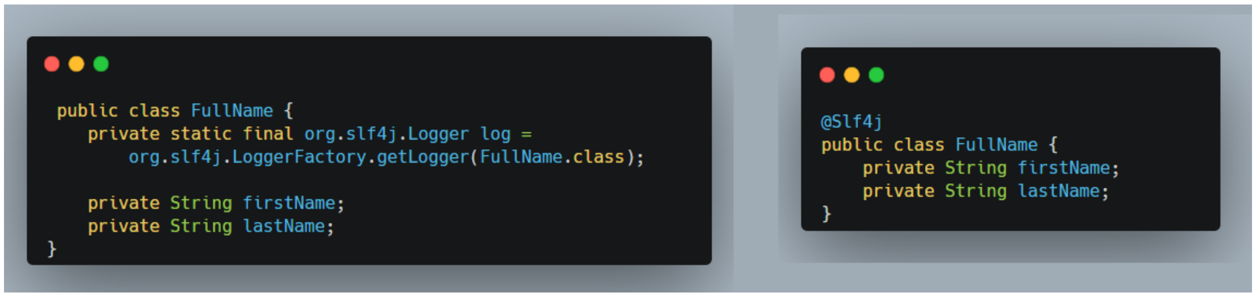


```java
public class FullName {
    private final String firstName;
    private final String lastName;

    public FullName(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public FullName withFirstName(String newFirstName) {
        return new FullName(newFirstName, this.lastName);
    }
}


// or:

public record FullName (String firstName, String lastName) {

    public FullName withFirstName(String newFirstName) {
        return new FullName(newFirstName, this.lastName);
    }
}
```

```java
public class FullName {
    @With
    private final String firstName;
    private final String lastName;
}


// or:

public record FullName (@With String firstName, String lastName) {
}
```

@Slf4j

We can add an annotation at the class level and Lombok will create a log instance for us. Depending on what logging library we use, there are different implementations available, some popular ones are *@Slf4j* and *@Log4j2:*

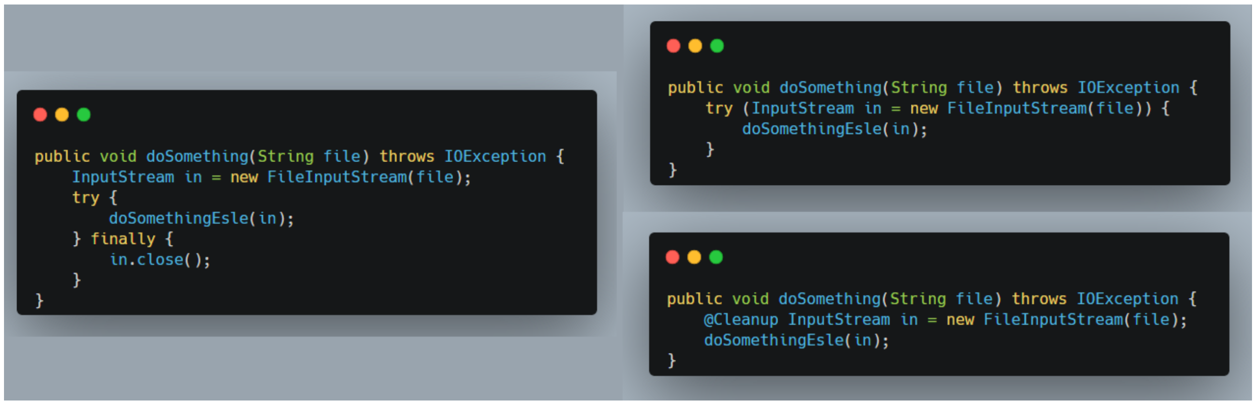


```java
public class FullName {
    private static final org.slf4j.Logger log =
        org.slf4j.LoggerFactory.getLogger(FullName.class);

    private String firstName;
    private String lastName;
}
```

```java
@Slf4j
public class FullName {
    private String firstName;
    private String lastName;
}
```

@Cleanup

This is, perhaps, one of the least used Lombok annotations. This can be used on the classes that do implement the *Closable* interface and it will be similar to the try-with-resources block. Let's compare all three solutions:



```java
public void doSomething(String file) throws IOException {
    InputStream in = new FileInputStream(file);
    try {
        doSomethingEsle(in);
    } finally {
        in.close();
    }
}
```

```java
public void doSomething(String file) throws IOException {
    try (InputStream in = new FileInputStream(file)) {
        doSomethingEsle(in);
    }
}
```

```java
public void doSomething(String file) throws IOException {
    @Cleanup InputStream in = new FileInputStream(file);
    doSomethingEsle(in);
}
```