# Setters

## Mutability, Problems, DTO and ORM

Yegor Bugayenko

Lecture #4 out of 8
90 minutes

- Mutability
- Problems
- ORM
- Apache Commons Email
- Read and Watch

Chapter #1:

# Mutability

# Which object is immutable?

```
class Book {
  private String title;
  Book(String t) { title = t; }
  void setTitle(String t) {
    this.title = t;
  }
  String getTitle() {
    return this.title;
  }
}
Book b = new Book("Object Thinking");
b.setTitle("It");
```

```
class Book {
  private final String title;
  Book(String t) { title = t; }
  void withTitle(String t) {
    return new Book(t);
  }
  String getTitle() {
    return this.title;
  }
}
Book b1 = new Book("Object Thinking");
Book b2 = b1.withTitle("It");
```

# There are four gradients of immutability

I. Constant

II. Not a Constant

III. Represented Mutability

IV. Encapsulated Mutability

# Gradient I: Constant

```
class Book {
  private final String t;
  Book(String t) { this.t = t; }
  String title() {
    return this.t;
  }
}
```

```
Book b = new Book("Object Thinking");
String t1 = b.title();
String t2 = b.title()
```

# Gradient II: Not a Constant

```
class Book {
  private final String t;
  Book(String t) { this.t = t; }
  String title() {
    return String.format(
      "%s / %s", title, new Date()
    );
  }
}
```

```
Book b = new Book("Object Thinking");
String t1 = b.title();
String t2 = b.title()
```

# Gradient III: Represented Mutability

```
class Book {
  private final Path path;
  Book(Path p) { this.path = p; }
  Book rename(String title) {
    Files.write(
      this.path,
      title.getBytes(),
      StandardOpenOption.CREATE
    );
    return this;
  }
  String title() {
    return new String(
      Files.readAllBytes(this.path)
    );
  }
}
```

```
Book b = new Book("Object Thinking");
String t1 = b.title();
b.rename("Elegant Objects");
String t2 = b.title()
```

# Gradient IV: Encapsulated Mutability

```
class Book {
  private final StringBuffer buffer;
  Book rename(String t) {
    this.buffer.setLength(0);
    this.buffer.append(t);
    return this;
  }
  String title() {
    return this.buffer.toString();
  }
}
```

```
Book b = new Book("Object Thinking");
String t1 = b.title();
b.rename("Elegant Objects");
String t2 = b.title()
```

Chapter #2:

# Problems

> Only gradients III and IV cause problems, while “Constant” and ‘Not a Constant” objects are harmless

# Side effects

With a side effect:

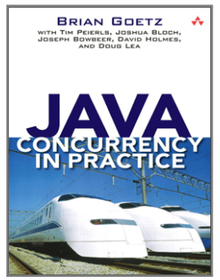
```
1 public String post(Request request) {
2   request.setMethod("POST");
3   return request.fetch();
4 }
5
6 Request r = new Request("http://...");
7 r.setMethod("GET");
8 String first = this.post(r);
9 String second = r.fetch();
```

Without a side effect:

```
1 public String post(Request request) {
2   return request
3     .withMethod("POST")
4     .fetch();
5 }
6
7 Request r = new Request("http://...")
8   .withMethod("GET");
9 String first = this.post(r);
10 String second = r.fetch();
```

# Thread (un-)safety

```
class Books {
  private int c = 0;
  void add() {
    this.c = this.c + 1;
  }
}
```




Goetz et al. explained the advantages of immutable objects in more details in their very famous book “Java Concurrency in Practice” (highly recommended!)

```
ExecutorService e =
  Executors.newCachedThreadPool();
final Books books = new Books();
for (int i = 0; i < 1000; i++) {
  e.execute(
    new Thread(
      () -> {
        books.add();
      }
    )
  );
}
// What is the value of "books.c"?
```

# Temporal Coupling

```
1 Request r = new Request("http://...");
2 r.setMethod("POST");
3 String first = r.fetch();
4 r.setBody("text=hello");
5 String second = r.fetch();
```

```
1 Request r = new Request("http://...");
2 // r.setMethod("POST");
3 // String first = r.fetch();
4 r.setBody("text=hello");
5 String second = r.fetch();
```

# Identity Mutability

```
1 Date first = new Date(1L);
2 Date second = new Date(1L);
3 first.setTime(2L);
4 assert first.equals(second); // false
```

```
1 Map<Date, String> map = new HashMap<>();
2 Date date = new Date();
3 map.put(date, "hello, world!");
4 date.setTime(12345L);
5 assert map.containsKey(date); // false
```

# Chapter #3:
# ORM

> ORM stands for “Object Relational Mapping,” which is an attempt to represent a relational data model in objects and relations between them, such as attributes, methods, and inheritance

# Java Persistence API

```java
@Entity
@Table(name = "movie")
public class Movie {
  @Id
  private Long id;
  private String name;
  private Integer year;
  // ctors
  // getters
  // setters
}
```

```java
EntityManager em = getEntityManager();
em.getTransaction().begin();
Movie movie = em.findById(1L);
movie.setName("The Godfather");
em.persist(movie);
em.getTransaction().commit();
```

# SQL speaking objects

```
interface Movie {
  int id();
  String title();
  String author();
}
Movie m = new PgMovie(ds, 1L);
m.update("The Godfather");
```

Here I'm using jcabi-jdbc, an object-oriented wrapper around JDBC data source.

```
final class PgMovie implements Movie
  private final Source dbase;
  private final int number;
  public PgMovie(DataSource data, int id)
    this.dbase = data;
    this.number = id;
  public String title()
    return new JdbcSession(this.dbase)
      .sql("SELECT title FROM movie WHERE id = ?")
      .set(this.number)
      .select(new SingleOutcome<String>(String.class));
  public void rename(String n)
    new JdbcSession(this.dbase)
      .sql("UPDATE movie SET name = ? WHERE id = ?")
      .set(n)
      .set(this.number)
      .execute();
```

# Complex SQL queries

```
final class PgMovies
  private final Source dbase;
  public PgMovies(DataSource data)
    this.dbase = data;
  public Movie movie(Long id)
    return new PgMovie(this.dbase, id);
```

```
final class PgMovie implements Movie
  private final Source dbase;
  private final int number;
  public PgMovie(DataSource data, int id)
    this.dbase = data;
    this.number = id;
  public String title()
    return new JdbcSession(this.dbase)
      .sql("SELECT title FROM movie WHERE id = ?")
      .set(this.number)
      .select(new SingleOutcome<String>(String.class));
  public String author()
    return new JdbcSession(this.dbase)
      .sql("SELECT name FROM movie JOIN author ON
      ↪  author.id = movie.author WHERE movie.id = ?")
      .set(this.number)
      .select(new SingleOutcome<String>(String.class));
```

Chapter #4:

# Apache Commons Email

Chapter #5:

# Read and Watch

Why use getters and setters/accessors? in Stack Overflow

Objects Should Be Immutable by me

Gradients of Immutability by me

Immutable Objects Are Not Dumb by me

How an Immutable Object Can Have State and Behavior? by me

How Immutability Helps by me

ORM Is an Offensive Anti-Pattern by me