Setters

Yegor Bugayenko

Lecture #4 out of 8
80 minutes

The slidedeck was presented by the author in this YouTube Video

- Pre-Test
- Mutability
- Drawbacks of Mutability
- ORM
- Apache Commons Email
- What About Performance?

Chapter #1:

Pre-Test

How would you do this?

```
1 Message m = new Message();
2 m.setName("Sarah");
3 m.print(); // Hello, Sarah!
4
5 m.setName("Victor");
6 m.print(); // Hello, Victor!
7
8 m.setName("Leyla");
9 m.print(); // Hello, Leyla!
```

```
1 Message m1 = new Message("Sarah");
2 m1.print(); // Hello, Sarah!
3
4 Message m2 = new Message("Victor");
5 m2.print(); // Hello, Victor!
6
7 Message m3 = new Message("Leyla");
8 m3.print(); // Hello, Leyla!
```

Chapter #2:

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;
  }
}
b = new Book();
b.setTitle("Object Thinking");
```

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

There are four gradients of immutability

I. Constant

II. Not a Constant

III. Represented Mutability

IV. Encapsulated Mutability

You may read my blog about this [Bugayenko, 2016].

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()
```

The `title()` method returns exactly the same data on each call. This object is definitely immutable.

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, return new Date()
    );
  }
}
```

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

The `title()` method returns different data on each new call, depending on system timer. Does it make the object mutable or not?

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()
```

The `title()` method returns different data on each new call, depending on the content of the file in the file system. Does it make the object mutable or not?

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()
```

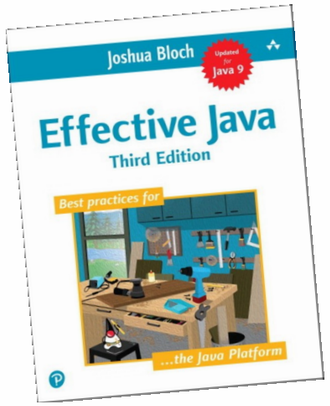
The `title()` method returns different data on each new call, depending on the content of the memory block. Does it make the object mutable or not?

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

You may want to read my blog about immutability [Bugayenko, 2014a,e,d,b].

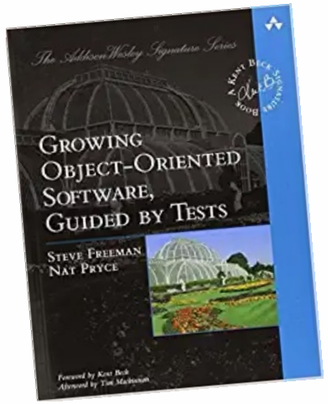
Chapter #3:

Drawbacks of Mutability



“Immutable classes are easier to design, implement, and use than mutable classes. They are less prone to error and are more secure.”

— Joshua Bloch. *Effective Java.* Prentice Hall PTR, 2008. doi:10.5555/1377533



Steve Freeman

"Writing large-scale functional programs is a topic for a different book, but we find that a little immutability within the implementation of a class leads to much safer code and that, if we do a good job, the code reads well too."

— Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Pearson Education, 2009. doi:10.5555/1655852

1) Side effects

With a side effect:

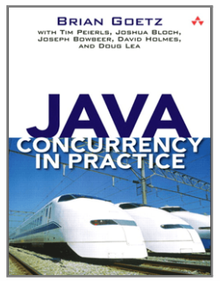
```
1  public String post(Request request) {
2    request.setMethod("POST");
3    return request.fetch();
4  }
5
6  r = new Request("x.com");
7  r.setMethod("GET");
8  String first = this.post(r);
9
10 String second = r.fetch();
```

Without a side effect:

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

2) Thread (un-)safety

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



Goetz [2006] explained the advantages of immutable objects in more details in their very famous book “Java Concurrency in Practice” (highly recommended!)

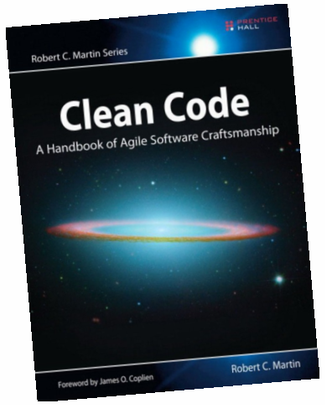
```
ExecutorService e =
  Executors.newCachedThreadPool();
final Books books = new Books();
for (int i = 0; i < 1000; i++) {
  e.execute(
    new Thread(
      () -> {
        books.add();
      }
    )
  );
}
// What is the value of "books.c"?
```

3) Temporal Coupling

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

```
1 r = new Request("x.com");
2
3 // 100 lines later:
4 // r.setMethod("POST");
5 // String first = r.fetch();
6
7 r.setBody("text=hello");
8 String second = r.fetch();
```

"Sequential coupling (also known as temporal coupling) is a form of coupling where a class requires its methods to be called in a particular sequence." — Wikipedia.

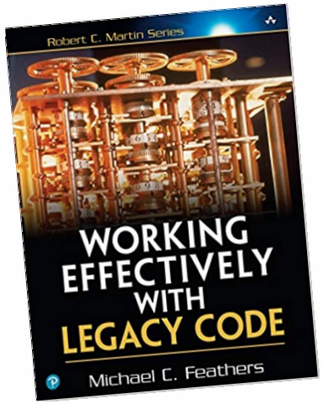


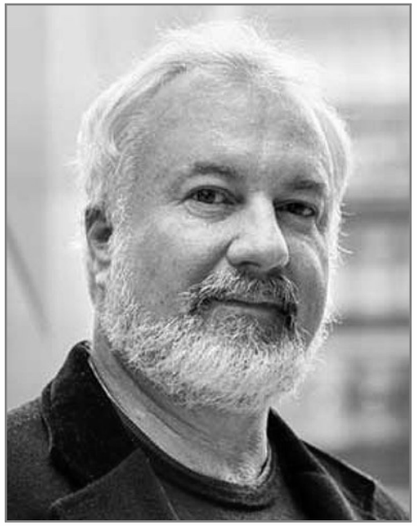
"Side effects are lies. Your function promises to do one thing, but it also does other hidden things. They are devious and damaging mistruths that often result in strange temporal couplings and order dependencies."

— Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson Education, 2008. doi:10.5555/1388398

"Sequential cohesion (considered to be less than ideal) exists when a routine contains operations that must be performed in a specific order, that share data from step to step, and that don't make up a complete function when done together."

— Steve McConnell. *Software Project Survival Guide.* Microsoft Press, 1998. doi:10.5555/270015





"Back in the early days of programming, this was named temporal coupling, and it is a pretty nasty thing when you do it excessively. When you group things together just because they have to happen at the same time, the relationship between them isn't very strong. Later you might find that you have to do one of those things without the other, but at that point they might have grown together. Without a seam, separating them can be hard work."

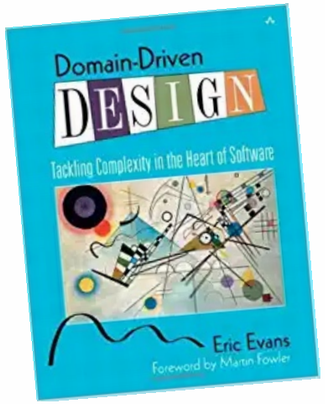
— Michael Feathers. *Working Effectively With Legacy Code.* Prentice Hall Professional, 2004. doi:10.5555/1050933

4) Identity Mutability

```
1  Map<Date, String> map = new HashMap<>();
2  Date date = new Date();
3  map.put(date, "hello, world!");
4
5  // This is TRUE:
6  assert map.containsKey(date);
7
8  date.setTime(12345L);
9  // Why this is FALSE??:
10 assert map.containsKey(date);
```

"In order for an object to be shared safely, it must be immutable: it cannot be changed except by full replacement."

Source: Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004. doi:10.5555/861502



"As long as a value object is immutable, change management is simple—there isn't any change except full replacement. Immutable objects can be freely shared."

— Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004. doi:10.5555/861502

Chapter #4:
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

You may want to read my blog about ORM [Bugayenko, 2014c].

"One of the first and most easily-recognizable problems in using objects as a front-end to a relational data store is that of how to map classes to tables."

— Ted Neward. The Vietnam of Computer Science. https://web.archive.org/web/20220823105749/http://blogs.tedneward.com/post/the-vietnam-of-computer-science/, jun 2006. [Online; accessed 19-09-2024]

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.rename("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 #5:

Apache Commons Email

org.apache.commons.mail.Email

```
1  public abstract class Email {
2    protected MimeMessage message;
3    protected String charset;
4    protected InternetAddress fromAddress;
5    protected String subject;
6    protected MimeMultipart emailBody;
7    protected Object content;
8    protected String contentType;
9    protected boolean debug;
10   protected Date sentDate;
11   protected Authenticator authenticator;
12   protected String hostName;
13   protected String smtpPort;
14   protected String sslSmtpPort;
15   protected List<InternetAddress> toList;
16   protected List<InternetAddress> ccList;
17   protected List<InternetAddress> bccList;
18   protected List<InternetAddress> replyList;
19   protected String bounceAddress;
20   protected Map<String, String> headers;
21   protected boolean popBeforeSmtp;
22   protected String popHost;
23   protected String popUsername;
24   protected String popPassword;
25   protected boolean tls;
26   protected boolean ssl;
27   protected int socketTimeout;
28   protected int socketConnectionTimeout;
29   private boolean startTlsEnabled;
30   private boolean startTlsRequired;
31   private boolean sslOnConnect;
32   private boolean sslCheckServerIdentity;
33   private boolean sendPartial;
34   private Session session;
35 }
```

https://github.com/apache/commons-email/blob/EMAIL_1_5/src/main/java/org/apache/commons/mail/Email.java

Chapter #6:

What About Performance?

Zoran Budimlić

"Although Java implementations have been made great strides, they still fall short on programs that use the full power of Java's object-oriented features. Ideally, future compiler technologies will be able to automatically transform the [OO style code] into something that approaches the [procedural style] in performance."

— Zoran Budimlić, Ken Kennedy, and Jeff Piper. The Cost of Being Object-Oriented: A Preliminary Study. *Scientific Programming*, 7(2):87–95, 1999. doi:10.1155/1999/464598

References

Joshua Bloch. *Effective Java*. Prentice Hall PTR, 2008. doi:10.5555/1377533.

Zoran Budimlić, Ken Kennedy, and Jeff Piper. The Cost of Being Object-Oriented: A Preliminary Study. *Scientific Programming*, 7(2):87–95, 1999. doi:10.1155/1999/464598.

Yegor Bugayenko. Objects Should Be Immutable. https://www.yegor256.com/140609.html, jun 2014a. [Online; accessed 08-07-2024].

Yegor Bugayenko. How Immutability Helps. https://www.yegor256.com/141107.html, nov 2014b. [Online; accessed 08-07-2024].

Yegor Bugayenko. ORM Is an Offensive Anti-Pattern. https://www.yegor256.com/141201.html, dec 2014c. [Online; accessed 08-07-2024].

Yegor Bugayenko. How an Immutable Object Can Have State and Behavior? https://www.yegor256.com/141209.html, dec 2014d. [Online; accessed 08-07-2024].

Yegor Bugayenko. Immutable Objects Are Not Dumb. https://www.yegor256.com/141222.html, dec 2014e. [Online; accessed 08-07-2024].

Yegor Bugayenko. Gradients of Immutability. https://www.yegor256.com/160907.html, sep 2016. [Online; accessed 08-07-2024].

Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004. doi:10.5555/861502.

Michael Feathers. *Working Effectively With Legacy Code*. Prentice Hall Professional, 2004. doi:10.5555/1050933.

Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Pearson Education, 2009. doi:10.5555/1655852.

Brian Goetz. *Java Concurrency in Practice*. Pearson Education, 2006. doi:10.5555/1076522.

Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2008.

doi:10.5555/1388398.

Steve McConnell. *Software Project Survival Guide*. Microsoft Press, 1998. doi:10.5555/270015.

Ted Neward. The Vietnam of Computer Science. https://web.archive.org/web/20220823105749/http://blogs.tedneward.com/post/the-vietnam-of-computer-science/, jun 2006. [Online; accessed 19-09-2024].