Getters

Accessors, Encapsulation, Printers

Yegor Bugayenko

Lecture #3 out of 8
80 minutes

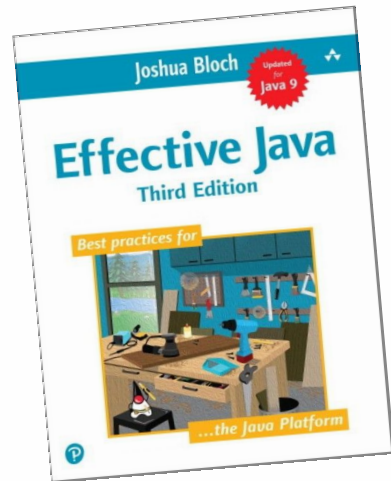The slidedeck was presented by the author in this YouTube Video

What Is a Getter?

Encapsulation

The Alternative

Chapter #1:
What Is a Getter?

"If a class is accessible outside the confines of its package, the prudent programmer will provide <u>accessor methods</u> to preserve the flexibility to change the class's internal representation."

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

```java
// Degenerate classes like this should not be public!
class Point {
    public float x;
    public float y;
}


// Encapsulated structure class
class Point {
    private float x;
    private float y;

    public Point(float x, float y) {
        this.x = x;
        this.y = y;
    }

    public float getX() { return x; }
    public float getY() { return y; }

    public void setX(float x) { this.x = x; }
    public void setY(float y) { this.y = y; }
}
```

"Because such classes are accessed by their data fields, they do not offer the benefits of encapsulation. You cannot change the representation of such a class without changing its API, enforce any invariants, and take any auxiliary action when a field is modified. [...] such classes are anathema and should always be replaced by classes with private fields and public accessor methods."

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

[What? Boilerplate Purpose DTO]

Which One Is a Getter?

```
1  class MiddleSquare {
2    private short seed;
3    MiddleSquare(short s) { seed = s; }
4
5    short getNextNumber() {
6      int s = (int) seed * seed;
7      seed = (short) (s >> 8);
8      return seed;
9    }
10 }
```

```
1  class MiddleSquare {
2    private short seed;
3    MiddleSquare(short s) { seed = s; }
4
5    short getSeed() {
6      return this.seed;
7    }
8  }
```

Boilerplate Getters

```ruby
class Document
  def initialize(n)
    @name = n
  end
  def name
    @name
  end
end
d = Document.new("/tmp/test.txt")
puts d.name
```

```ruby
class Document
  attr_reader :name
  def initialize(n)
    @name = n
  end
end

d = Document.new("/tmp/test.txt")
puts d.name
```

Some modern programming languages (e.g. Ruby) offer an ability to generate the boilerplate for mutators and accessors in a single line.

[What? Boilerplate Purpose DTO]

What Getters Are for?

Using getters in Java, rather than directly accessing fields, has several benefits:

1. **Encapsulation**: Getters and setters are fundamental to the object-oriented programming principle of encapsulation, also known as data hiding. This is the practice of keeping fields within a class private, then providing access to them via public methods (getters for read access and setters for write access). This means that the inner workings of the class can be hidden from outside classes.

2. **Validation**: By using setters, you can add validation logic before setting a value. This way, you can control what data gets set. For instance, if you have a `setAge` method, you can check if the age is not negative before setting the value. Directly exposing the age field could allow someone to set a negative age, which might not be appropriate.

3. **Flexibility & Maintenance**: When a field is accessed directly, you're tied to the field itself. If the field is removed or the type is changed, every access needs to be updated. By using getters and setters, you can remove a field and compute the value from other fields, or change the type and update the getter and setter, without any other code needing to be changed.

4. **Control over Field's Mutability**: Making class fields public exposes them to outside modification, which can lead to data inconsistency. Getters and setters help to maintain field's immutability by returning a defensive copy of the object (when the object is mutable) thereby preserving data consistency.

5. **Ease of Debugging**: If a field value is incorrect, it's easier to find out where the value was set if it's done through a setter, because you can add a breakpoint or logging in the setter. If fields are public, the incorrect value could be set from anywhere.

6. **Abstraction**: Getters and setters allow the class to change how the data is stored internally without affecting any classes that use it. For example, you might start with a simple class that just stores a value, but later you might store that value in a database. If you used public fields, you'd need to change all the classes that use it. But if you used getters and setters, you can just change them and the rest of your code remains unaffected.

7. **Enabling JavaBeans conventions**: JavaBeans are classes that encapsulate many objects into a single object (the bean). They are serializable, have a no-argument constructor, and allow access to properties using getter and setter methods. By following these conventions, your classes can interact with a lot of frameworks and libraries which expect these conventions.

Remember, while these are general best practices, there can be exceptions depending on the context. For example, in certain situations like data classes used for data transfer, directly accessible fields might be acceptable. As with all design decisions, the right approach depends on the specific needs of your project.

This is what GPT-4 replied to my question: "Why should we use getters in Java instead of making class fields public? What are the benefits?"

Data Transfer Objects (DTO)

```
1  class BookDTO {
2    private int id;
3    private String author;
4    private String title;
5    BookDTO(int i, String a, String t)
6        { id = i; author = a; title = t; }
7    int getId() { return id; }
8    String getAuthor() { return author; }
9    String getTitle() { return title; }
10 }
```

```
1  class JsonApi {
2    BookDTO getById(int id) { /* ... */ }
3  }
4
5  BookDTO dto = api.getById(42);
6
7  print(dto.getTitle());
8  print(dto.getAuthor());
```

There is no excuse for the use of DTO [Bugayenko, 2016b]. Almost...

Chapter #2:
Encapsulation
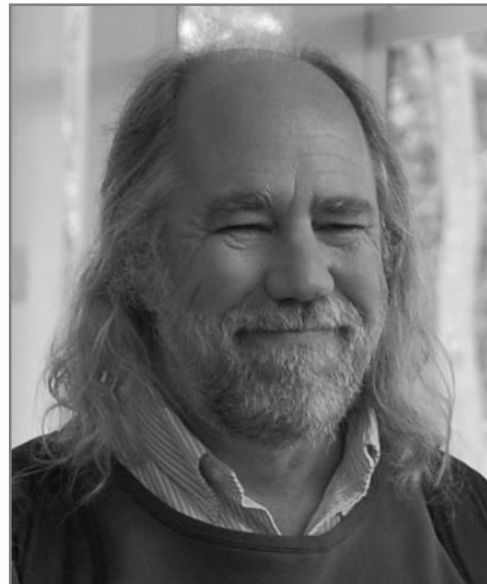
"Encapsulation is most often achieved through information hiding, which is the process of hiding all the secrets of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the implementation of its methods... Encapsulation provides explicit barriers among different abstractions and thus leads to a clear separation of concerns."

— Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Connallen, and Kelli A. Houston. *Object-Oriented Analysis and Design With Applications.* Addison-Wesley, 1994. doi:10.5555/1407387

"In most ways, encapsulation is a discipline more than a real barrier; seldom is the integrity of an object protected in any absolute sense, and this is especially true of software objects, so it is up to the user of an object to respect that object's encapsulation."

— David West. *Object Thinking*. Pearson Education, 2004. doi:10.5555/984130

[Integrity Semantic]

Who Is Protecting the Integrity Better?

```
1  class User {
2    private String name;
3    User(String n) { name = n; }
4    int getName() {
5      return this.name;
6    }
7  }
8  if (employees.contains(user.getName())) {
9    /* pay a salary */
10 }
```

```
1  class User {
2    private String name;
3    User(String n) { name = n; }
4    bool isEmployee() {
5      return employees.contains(this.name);
6    }
7  }
8  if (user.isEmployee()) {
9    /* pay a salary */
10 }
```

Which class hides its data better?

[Integrity Semantic]

Who Knows About Data Semantic?

```
1  class Box {
2    private int weight;
3    Box(int kg) { weight = kg; }
4    int getWeight() {
5      return this.weight;
6    }
7  }
8  int w = box.getWeight();
9  int lbs = w / 0.454;
10 printf("The weight is \%d lbs\n");
```

```
1  class Box {
2    private int weight;
3    Box(int kg) { weight = kg; }
4    int getLbs() {
5      return this.weight / 0.454;
6    }
7  }
8  int lbs = box.getLbs();
9  printf("The weight is \%d lbs\n");
```

What happens if the `Box` decides to store the `weight` in pounds instead of kilograms? How will it know how many of its clients still assume that the `weight` is in kilograms? [Bugayenko, 2014]

Chapter #3:
The Alternative

"**Tell-Don't-Ask** principle: Rather than asking an object for data and acting on that data, we should instead tell an object what to do. This encourages to move behavior into an object to go with the data."

— Martin Fowler. TellDontAsk.
https://martinfowler.com/bliki/TellDontAsk.html, 9 2013. [Online; accessed 12-09-2024]

"One of the fundamental principles of object-oriented design is to combine data and behavior, so that the basic elements of our system (objects) combine both together. This is often a good thing because this data and the behavior that manipulates them are tightly coupled."

Source: Martin Fowler. TellDontAsk. `https://martinfowler.com/bliki/TellDontAsk.html`, 9 2013. [Online; accessed 12-09-2024]

[TellDontAsk NoPrefix Public]

1) Tell Don't Ask, an Example

```
1  class BookDTO {
2    private int id;
3    private String author;
4    private String title;
5    BookDTO(int i, String a, String t)
6       { id = i; author = a; title = t; }
7    void print() { /* ... */ }
8  }
```

```
1  class JsonApi {
2    BookDTO getById(int id) { /* ... */ }
3  }
4
5  BookDTO dto = api.getById(42);
6  dto.print();
```

The `print()` method may also be called a "printer" [Bugayenko, 2016a].

[TellDontAsk NoPrefix Public]

"Though getter/setter methods are commonplace in Java, they are not particularly object-oriented. In fact, they can <u>damage</u> your code's maintainability. Moreover, the presence of numerous getter and setter methods is a red flag that the program isn't necessarily well designed from an OO perspective."

— Allen Holub. Why Getter and Setter Methods Are Evil. https://jttu.net/holub2003why, 9 2003. [Online; accessed 12-09-2024]

[TellDontAsk NoPrefix Public]

2) Get Rid of the "Get" Prefix

```
1  class Book {
2    private int id;
3    private String author;
4    private String title;
5    Book(int i, String a, String t)
6      { id = i; author = a; title = t; }
7    int id() { return id; }
8    String author() { return author; }
9    String title() { return title; }
10 }
```

```
1  class JsonApi {
2    Book getById(int id) { /* ... */ }
3  }
4
5  Book book = api.getById(42);
6
7  print(book.author());
8  print(book.title());
```

[TellDontAsk NoPrefix Public]

3) Make Fields Public

```
1  class Book {
2    public final int id;
3    public final String author;
4    public final String title;
5    Book(int i, String a, String t)
6      { id = i; author = a; title = t; }
7  }
```

```
1  class JsonApi {
2    Book getById(int id) { /* ... */ }
3  }
4
5  Book book = api.getById(42);
6
7  print(book.author);
8  print(book.title);
```

[TellDontAsk NoPrefix Public]

Bibliography

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

Grady Booch, Robert A. Maksimchuk, Michael W. Engle,
Bobbi J. Young, Jim Connallen, and Kelli A. Houston.
Object-Oriented Analysis and Design With Applications.

Addison-Wesley, 1994. doi:10.5555/1407387.

Yegor Bugayenko. Getters/Setters. Evil. Period.
https://www.yegor256.com/140916.html, 9 2014.
[Online; accessed 08-07-2024].

Yegor Bugayenko. Printers Instead of Getters.
https://www.yegor256.com/160405.html, 4 2016a.
[Online; accessed 08-07-2024].

Yegor Bugayenko. Data Transfer Object Is a Shame.
https://www.yegor256.com/160706.html, 7 2016b.

[Online; accessed 08-07-2024].

Martin Fowler. TellDontAsk. https:
//martinfowler.com/bliki/TellDontAsk.html, 9
2013. [Online; accessed 12-09-2024].

Allen Holub. Why Getter and Setter Methods Are Evil.
https://jttu.net/holub2003why, 9 2003. [Online;
accessed 12-09-2024].

David West. *Object Thinking*. Pearson Education, 2004.
doi:10.5555/984130.