Getters

Accessors, Encapsulation, Printers

Yegor Bugayenko

Lecture #3 out of 8
90 minutes

- What is a Getter?
- Encapsulation
- The Alternative
- Read and Watch

Chapter #1:

What is a Getter?

Which one is a getter?

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

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

A getter method may also be known as an “accessor.”

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
```

Modern programming languages often offer the ability to generate the boilerplate for mutators and accessors in a single line.

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?”

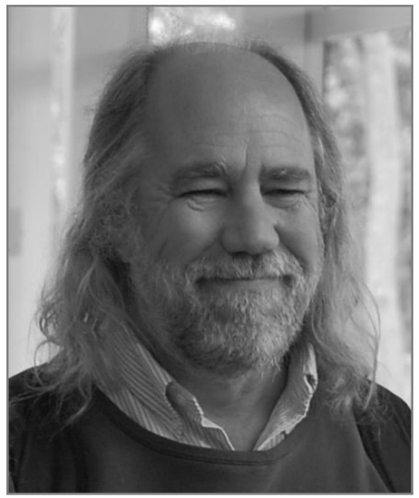
Data Transfer Objects (DTO)

```
class BookDTO {
  private int id;
  private String author;
  private String title;
  BookDTO(int i, String a, String t)
    { id = i; author = a; title = t; }
  int getId() { return id; }
  String getAuthor() { return author; }
  String getTitle() { return title; }
}
```

```
class JsonApi {
  BookDTO getById(int id) { /* ... */ }
}
BookDTO dto = api.getById(42);
print(dto.getTitle());
print(dto.getAuthor());
```

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 et al. (2007)
Object-Oriented Analysis and Design with Applications

"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 et al. (2006)
Object Thinking

Who is protecting the integrity better?

```
class User {
  private String name;
  User(String n) { name = n; }
  int getName() {
    return this.name;
  }
}
if (employees.contains(user.getName())) {
  /* pay a salary */
}
```

```
class User {
  private String name;
  User(String n) { name = n; }
  bool isEmployee() {
    return employees.contains(this.name);
  }
}
if (user.isEmployee()) {
  /* pay a salary */
}
```

Which class hides data better?

Who knows about data semantic?

```
class Box {
  private int weight;
  Box(int kg) { weight = kg; }
  int getWeight() {
    return this.weight;
  }
}
int w = box.getWeight();
int lbs = w / 0.454;
printf("The weight is \%d lbs\n");
```

```
class Box {
  private int weight;
  Box(int kg) { weight = kg; }
  int getLbs() {
    return this.weight / 0.454;
  }
}
int lbs = box.getLbs();
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?

Chapter #3:

The Alternative

Tell Don't Ask

```
class BookDTO {
  private int id;
  private String author;
  private String title;
  BookDTO(int i, String a, String t)
    { id = i; author = a; title = t; }
  void print() { /* ... */ }
}
```

```
class JsonApi {
  BookDTO getById(int id) { /* ... */ }
}
BookDTO dto = api.getById(42);
dto.print();
```

Get rid of the “get” prefix

```
class Book {
  private int id;
  private String author;
  private String title;
  Book(int i, String a, String t)
    { id = i; author = a; title = t; }
  int id() { return id; }
  String author() { return author; }
  String title() { return title; }
}
```

```
class JsonApi {
  Book getById(int id) { /* ... */ }
}
Book book = api.getById(42);
print(book.author());
print(book.title());
```

Make fields public

```
class Book {
  public final int id;
  public final String author;
  public final String title;
  Book(int i, String a, String t)
    { id = i; author = a; title = t; }
}
```

```
class JsonApi {
  Book getById(int id) { /* ... */ }
}
Book book = api.getById(42);
print(book.author);
print(book.title);
```

Chapter #4:

Read and Watch

Why getter and setter methods are evil by Allen Holub (2003)

GetterEradicator by Martin Fowler (2006)

https://martinfowler.com/bliki/TellDontAsk.html by Martin Fowler (2013)

Getters/Setters. Evil. Period. by me (2014)