Lesson 13: OOP #3, Safe Data Access & Mutator Methods (W04D1)

Balboa High School

Michael Ferraro

September 8, 2015

Do Now

Inside a new Eclipse Java project called `Closet`:

- Create a blueprint class, `Shirt`, and include these fields:
  - `color`, a `String`
  - `material`, a `String`
  - `size`, an `int`
- Create a driver class called `ShirtDriver`
  - include a `main()` method
  - make two different shirts and store values for their color, material, and size
  - retrieve the stored values and print them out

Aim

Students will learn about how to make objects “safe” by hiding them from the outside world and making them available by proxy.

Do Now Solution

In case you need to refer to it, working versions of the Do Now files are here.

Shirt Shenanigans

Let's say a troll gets their hands on your driver code and adds a line:

```
Shirt shirt2 = new Shirt();
shirt2.color = "green";
shirt2.material = "nylon";
shirt2.size = 12;                    //smallest shirt
                                     //this brand makes!

//TROLL ADDS THIS LINE:
shirt2.size--;
```

Shirt Shenanigans

Let's say a troll gets their hands on your driver code and adds a line:

```
Shirt shirt2 = new Shirt();
shirt2.color = "green";
shirt2.material = "nylon";
shirt2.size = 12;                    //smallest shirt
                                     //this brand makes!


//TROLL ADDS THIS LINE:
shirt2.size--;
```

1. What is the new size of `shirt2`?
2. According to the comment near the original size, why is the new size bad?

Protect Your Objects!

The author of the `Shirt` class has a way to protect the fields so that another developer — e.g., the writer of the driver class (or a troll) — cannot violate his/her intended object state.

Protect Your Objects!

The author of the `Shirt` class has a way to protect the fields so that another developer — e.g., the writer of the driver class (or a troll) — cannot violate his/her intended object state.

We'll return to this idea shortly. First, let's write an *accessor method* for the `size` field:

```
public _____________ getSize() {

    ____________________________ ;
}
```

Protect Your Objects!

We haven't yet done anything to prevent a troll from changing our shirt size! Let's modify the `size` field, marking it `private`:

```
String color
String material
private int size
```

Protect Your Objects!

We haven't yet done anything to prevent a troll from changing our shirt size! Let's modify the `size` field, marking it `private`:

```
String color
String material
private int size
```

Now what do you notice in the driver class when setting `Shirt` sizes? Is our troll affected?

Mutator Methods

- You already have a way to get `size` details out of a `Shirt` object: `getSize()`, an accessor method

Mutator Methods

- You already have a way to get `size` details out of a `Shirt` object: `getSize()`, an accessor method
- How do we save a `size` now? We need a new kind of method, a *mutator method*

Mutator Methods

- You already have a way to get `size` details out of a `Shirt` object: `getSize()`, an accessor method
- How do we save a `size` now? We need a new kind of method, a *mutator method*
- mutator $\approx$ mutate $\approx$ change/modify/set

Inputs and Outputs

- We say an accessor method take no input and return something as output

Inputs and Outputs

- We say an accessor method take no input and return something as output
- For example, `getSize()` takes no info from us when we call it and return an `int`:

```
public int getSize() {
    return size;
}
```

Inputs and Outputs

- We say an accessor method take no input and return something as output
- For example, `getSize()` takes no info from us when we call it and return an `int`:

```
public int getSize() {
    return size;
}
```

- Think of a mutator method as the opposite: It takes in a value and returns/outputs nothing.

Inputs and Outputs

- We say an accessor method take no input and return something as output
- For example, `getSize()` takes no info from us when we call it and return an `int`:

```
public int getSize() {
    return size;
}
```

- Think of a mutator method as the opposite: It takes in a value and returns/outputs nothing.
- Let's write mutator method `setSize()`...

`setSize()` Mutator Method

Solution for the `setSize()` mutator method:

```
public void setSize(int s) {
    size = s;
}
```

Mutators as Guardians

- Recall that we cannot have a shirt with a size less than 12.

Mutators as Guardians

- Recall that we cannot have a shirt with a size less than 12.
- Add these statements to your driver class' `main()`:

```
Shirt shirt3 = new Shirt();

//good size:
shirt3.size = 15;
System.out.println("a.  shirt3 is size " + shirt3.getSize());

//bad size:
shirt3.size = 2;
System.out.println("b.  shirt3 is size " + shirt3.getSize());
```

Mutators as Guardians

- Recall that we cannot have a shirt with a size less than 12.
- Add these statements to your driver class' `main()`:

```
Shirt shirt3 = new Shirt();

//good size:
shirt3.size = 15;
System.out.println("a.  shirt3 is size " + shirt3.getSize());

//bad size:
shirt3.size = 2;
System.out.println("b.  shirt3 is size " + shirt3.getSize());
```

- How can we protect `Shirts` so that their sizes cannot be specified incorrectly?

Mutators as Guardians

Solution: Only allow the mutator method to change the size if the given size is 12 or greater.

```
public void setSize(int s) {
    if( s >= 12 ) {           //valid size
        size = s;
    }
}
```

Methods for the Other Fields

Let's do for `material` and `color` what we did for `size`:

1. mark the field `private`
2. write an accessor method
3. write a mutator method
4. update driver to use the new methods

Recap

- In the beginning, there was **dot notation:**

```
shirt1.color = "purple";
shirt1.size = 16;
```

  Fields in `Shirt.java` were not `private`

Recap

- In the beginning, there was **dot notation:**

```
shirt1.color = "purple";
shirt1.size = 16;
```

  Fields in `Shirt.java` were not `private`

- Now with getters and setters, we have to use **methods:**

```
shirt1.setColor("purple");
shirt1.setSize(16);
```

  Dot notation no longer works — fields in `Shirt.java` are `private`

Recap

- In the beginning, there was **dot notation:**

```
shirt1.color = "purple";
shirt1.size = 16;
```

  Fields in `Shirt.java` were not `private`

- Now with getters and setters, we have to use **methods:**

```
shirt1.setColor("purple");
shirt1.setSize(16);
```

  Dot notation no longer works — fields in `Shirt.java` are `private`

- Which way is faster? Safer? Better?

Data Hiding/Encapsulating

- When `private` fields are available exclusively through methods, you have
  - retained control over how the variable is set/changed/accessed
  - hidden details about how fields are managed (more on this later)

Data Hiding/Encapsulating

- When `private` fields are available exclusively through methods, you have
  - retained control over how the variable is set/changed/accessed
  - hidden details about how fields are managed (more on this later)
- Your class now *encapsulates* its data

Data Hiding/Encapsulating

- When `private` fields are available exclusively through methods, you have
  - retained control over how the variable is set/changed/accessed
  - hidden details about how fields are managed (more on this later)
- Your class now *encapsulates* its data
- Other classes — like a driver — have to use a middleman/broker/proxy to get data, regarded as a *safer* approach to managing data.

Data Hiding/Encapsulating

- When `private` fields are available exclusively through methods, you have
  - retained control over how the variable is set/changed/accessed
  - hidden details about how fields are managed (more on this later)
- Your class now *encapsulates* its data
- Other classes — like a driver — have to use a middleman/broker/proxy to get data, regarded as a *safer* approach to managing data.
- Using accessor and mutator methods is considered a **best practice** in OOP

Why is data hiding good?

Imagine this situation: You go to the bank, wanting to deposit $100 from your account.

- Scenario #1: You fill out a deposit slip, bring to teller behind counter, and (s)he handles transaction
- Scenario #2: Bank is deserted, doors are unlocked, safe is open, and you deposit the money yourself

What are the pros and cons of these scenarios?

Rest of the Period

- For each field in the `Person` class:
  - mark the field `private`
  - write an accessor method: `get_____()`
  - write a mutator method: `set_____(?)`
  - update `PersonDriver` so there are no errors
- Follow instructions in §1.3 of PS #2 — preparing project `ps02` and pushing it to GitHub
- Done with time to spare? Find a classmate who can use your help!

HW

- You should now be able complete PS #2, §§1−5, inclusive.
- To stay on pace to finish the problem set on time, you *should* complete §§1−3 in time for next class.
- Post your questions for peers to answer as comments on the PS #2 posting on the course site.