9

Using Encapsulation

ORACLE

Interactive Quizzes

Before you start today's lessons, test your knowledge by answering some quiz questions that relate to yesterday's lessons. Open your quiz file from labs > Quizzes > Java SE 8 Fundamentals Quiz.html. Click the links for the lessons titled "Describing Objects and Classes," "Manipulating and Formatting the Data in Your Program," and "Creating and Using Methods."

Objectives

After completing this lesson, you should be able to:

- Use an access modifier to make fields and methods private
- Create get and set methods to control access to private fields
- Define encapsulation as “information hiding”
- Implement encapsulation in a class using the NetBeans refactor feature
- Create an overloaded constructor and use it to instantiate an object

Topics

- Access control
- Encapsulation
- Overloading constructors

What Is Access Control?

Access control allows you to:

- Hide fields and methods from other classes
- Determine how internal data gets changed
- Keep the implementation separate from the public interface
  - Public interface:

```
setPrice( Customer cust)
```

  - Implementation:

```
public void setPrice(Customer cust){
   // set price discount relative to customer
}
```


Access control allows you to hide internal data and functionality in a class. In this lesson, you distinguish between the public interface of a class and the actual implementation of that interface.

- The public interface is what you see when you look up a class in the JDK API documentation. You get just the information you need in order to use a class. That is, the signatures for public methods, and data types of any public fields.
- The implementation of a class is the code itself, and also any private methods or fields that are used by that class. These are the internal workings of a class and it is not necessary to expose them to another class.

Access Modifiers

- `public`: Accessible by anyone
- `private`: Accessible only within the class

```
1 public class Item {
2    // Base price
3   private double price = 15.50;
4
5   public void setPrice(Customer cust){
6      if (cust.hasLoyaltyDiscount()){
7         price = price*.85; }
8   }
9 }
```

When a field is declared as public, any other class can access and potentially change the field's value. This is often problematic. It could be that the field represents sensitive data, such as a social security number, or that some type of logic or manipulation of the data may be required in order to safely modify the data. In the code example, the shirt price is declared in a private method. You would not want outside objects, such as a customer, to be able to freely manipulate the price of an item.

Access from Another Class

```
1 public class Item {
2    private double price = 15.50;
3    public void setPrice(Customer cust){
4      if (cust.hasLoyaltyDiscount()){
5         price = price*.85; }
6   }
7 }
8 public class Order{
9    public static void main(String args[]){
10     Customer cust = new Customer(int ID);
11     Item item = new Item();
12     item.price = 10.00;
13     item.setPrice(cust);
14  }
15 }
```

Won't compile

You don't need to know how `setPrice` works in order to use it.

ORACLE

Another Example

The data type of the field does not match the data type of the data used to set the field.

```
1 private int phone;
2 public void setPhoneNumber(String s_num){
3     //  parse out the dashes and parentheses from the
4     //  String first
5     this.phone = Integer.parseInt(s_num);
6 }
```


It may be that the data representing someone's phone number may be collected as a string, including spaces, dashes, and parentheses. If the phone number is represented internally as an `int`, then the setter method for the phone number will need to parse out spaces, dashes, and parentheses first, and then convert the `String` to an `int`. The `parseInt` method of `Integer` is covered in the "Using Encapsulation" lesson.

Using Access Control on Methods

```
1  public class Item {
2      private int id;
3      private String desc;
4      private double price;
5      private static int nextId = 1;
6
7      public Item(){
8          setId();
9          desc = "--description required--";
10         price = 0.00;
11     }
12
13     private void setId() {
14         id = Item.nextId++;
15     }
```

Called from within a public method

Private method

Here you see a private method that sets a new unique ID for an item. It is not necessary to expose this functionality to another class. The `setId` method is called from the public constructor method as part of its implementation.

Topics

- Access control
- Encapsulation
- Overloading constructors

Encapsulation

- Encapsulation means hiding object fields. It uses access control to hide the fields.
  - Safe access is provided by getter and setter methods.
  - In setter methods, use code to ensure that values are valid.
- Encapsulation mandates programming to the interface:
  - A method can change the data type to match the field.
  - A class can be changed as long as interface remains same.
- Encapsulation encourages good object-oriented (OO) design.

Get and Set Methods

```
1  public class Shirt {
2     private int shirtID = 0;       // Default ID for the shirt
3     private String description = "-description required-"; // default
4     private char colorCode = 'U'; //R=Red, B=Blue, G=Green, U=Unset
5     private double price = 0.0;    // Default price for all items
6
7     public char getColorCode() {
8         return colorCode;
9     }
10    public void setColorCode(char newCode) {
11        colorCode = newCode;
12    }
13        // Additional get and set methods for shirtID, description,
14        // and price would follow
15
16 } // end of class
```


If you make attributes private, how can another object access them? One object can access the private attributes of a second object if the second object provides public methods for each of the operations that are to be performed on the value of an attribute.

For example, it is recommended that all fields of a class should be private, and those that need to be accessed should have public methods for setting and getting their values.

This ensures that, at some future time, the actual field type itself could be changed, if that were advantageous. Or the getter or setter methods could be modified to control how the value could be changed, such as the value of the `colorCode`.

Why Use Setter and Getter Methods?

```
1  public class ShirtTest {
2      public static void main (String[] args) {
3          Shirt theShirt = new Shirt();
4          char colorCode;
5         // Set a valid colorCode
6          theShirt.setColorCode('R');
7          colorCode = theShirt.getColorCode();
8          System.out.println("Color Code: " + colorCode);
9         // Set an invalid color code
10         theShirt.setColorCode('Z');
11         colorCode = theShirt.getColorCode();
12         System.out.println("Color Code: " + colorCode);
13     }
14 …
```

Not a valid color code

Output:

```
Color Code: R
Color Code: Z
```


Though the code for the `Shirt` class is syntactically correct, the `setcolorCode` method does not contain any logic to ensure that the correct values are set.

The code example in the slide successfully sets an invalid color code in the `Shirt` object.

However, because `ShirtTest` accesses a private field on `Shirt` using a setter method, `Shirt` can now be recoded without modifying any of the classes that depend on it.

In the code example above, starting with line 6, the `ShirtTest` class is setting and getting a valid `colorCode`. Starting with line 10, the `ShirtTest` class is setting an invalid `colorCode` and confirming that invalid setting.

Setter Method with Checking

```
15   public void setColorCode(char newCode) {
16       if (newCode == 'R'){
17         colorCode = newCode;
18         return;
19       }
16       if (newCode == 'G') {
17         colorCode = newCode;
18         return;
19       }
16       if (newCode == 'B') {
17         colorCode = newCode;
18         return;
19       }
19       System.out.println("Invalid colorCode. Use R, G, or B");
20   }
21}
```


In the slide is another version of the `Shirt` class. However, in this class, before setting the value, the setter method ensures that the value is valid. If it is not valid, the `colorCode` field remains unchanged and an error message is printed.

Note: Void type methods can have return statements. They just cannot return any values.

Using Setter and Getter Methods

```
1 public class ShirtTest {
2           public static void main (String[] args) {
3           Shirt theShirt = new Shirt();
4           System.out.println("Color Code: " + theShirt.getColorCode());
5
6           // Try to set an invalid color code
7           Shirt1.setColorCode('Z');
8           System.out.println("Color Code: " + theShirt.getColorCode());
9  }
```

Not a valid color code

Output:

```
Color Code: U
Invalid colorCode. Use R, G, or B
Color Code: U
```

Before call to setColorCode() – shows default value

call to setColorCode prints error message

colorCode not modified by invalid argument passed to setColorCode()

Building on the previous slides, before the call to `setColorCode`, the default color value of U (unset) is printed. If you call `setColorCode` with an invalid code, the color code is not modified and the default value, U, is still the value. Additionally, you receive an error message that tells you to use the valid color codes, which are R, G, and B.

Exercise 9-1: Encapsulate a Class

In this exercise, you encapsulate the `Customer` class.

- Change access modifiers so that fields can be read or changed only through public methods.
- Allow the `ssn` field to be read but not modified.

- Open the Java Code Console and access 09-Encaps > Exercise1.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Topics

- Access control
- Encapsulation
- Overloading constructors

Initializing a `Shirt` Object

Explicitly:

```
 1 public class ShirtTest {
 2    public static void main (String[] args) {
 3        Shirt theShirt = new Shirt();
 4
 5        // Set values for the Shirt
 6        theShirt.setColorCode('R');
 7        theShirt.setDescription("Outdoors shirt");
 8        theShirt.price(39.99);
 9    }
10 }
```

Using a constructor:

```
Shirt theShirt = new Shirt('R', "Outdoors shirt", 39.99);
```


Assuming that you now have setters for all the private fields of `Shirt`, you could now instantiate and initialize a `Shirt` object by instantiating it and then setting the various fields through the setter methods.

However, Java provides a much more convenient way to instantiate and initialize an object by using a special method called a *constructor*.

Constructors

- Constructors are usually used to initialize fields in an object.
  - They can receive arguments.
  - When you create a constructor with arguments, it removes the default no-argument constructor.

All classes have at least one constructor.

If the code does not include an explicit constructor, the Java compiler automatically supplies a no-argument constructor. This is called the default constructor.

`Shirt` Constructor with Arguments

```
1 public class Shirt {
2   public int shirtID = 0;        // Default ID for the shirt
3   public String description = "-description required-"; // default
4   private char colorCode = 'U'; //R=Red, B=Blue, G=Green, U=Unset
5   public double price = 0.0;     // Default price all items
6
7  // This constructor takes three argument
8   public Shirt(char colorCode, String desc, double price ) {
9       setColorCode(colorCode);
10      setDescription(desc);
11      setPrice(price);
12  }
```


The `Shirt` example shown in the slide has a constructor that accepts three values to initialize three of the object's fields. Because `setColorCode` ensures that an invalid code cannot be set, the constructor can just call this method.

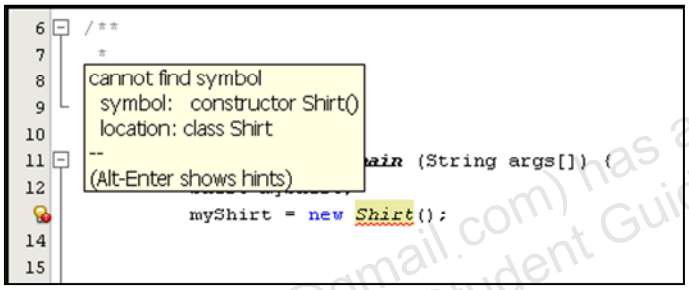
Default Constructor and Constructor with Args

When you create a constructor with arguments, the default constructor is no longer created by the compiler.

```
// default constructor
public Shirt()

// Constructor with args
public Shirt (char color, String desc, double price)
```

This constructor is not in the source code. It only exists if no constructor is explicitly defined.



When you explicitly create an overloaded constructor, it replaces the default no-argument constructor.

You may be wondering why you have been able to instantiate a Shirt object with `Shirt myShirt = new Shirt()` even if you did not actually create that no-argument constructor. If there is no explicit constructor in a class, Java assumes that you want to be able to instantiate the class, and gives you an *implicit* default no-argument constructor. Otherwise, how could you instantiate the class?

The example above shows a new constructor that takes arguments. When you do this, Java removes the implicit default constructor. Therefore, if you try to use `Shirt myShirt = new Shirt()`, the compiler cannot find this constructor because it no longer exists.

Overloading Constructors

```
1  public class Shirt {
2     ... //fields
3
4     // No-argument constructor
5     public Shirt() {
6          setColorCode('U');
7     }
8     // 1 argument constructor
9     public Shirt(char colorCode ) {
10         setColorCode(colorCode);
11    }
12    // 2 argument constructor
12    public Shirt(char colorCode, double price) {
14        this(colorCode);
15        setPrice(price);
16    }
```

If required, must be added explicitly

Calling the 1 argument constructor

The code in the slide shows three overloaded constructors:

- A default no-argument constructor
- A constructor with one parameter (a `char`)
- A constructor with two parameters (a `char` and a `double`)

This third constructor sets both the `colorCode` field and the `price` field. Notice, however, that the syntax where it sets the `colorCode` field is one that you have not seen yet. It would be possible to set `colorCode` with a simple call to `setColorCode()` just as the previous constructor does, but there is another option, as shown here.

You can chain the constructors by calling the second constructor in the first line of the third constructor using the following syntax:

```
this(argument);
```

The keyword `this` is a reference to the current object. In this case, it references the constructor method from this class whose signature matches.

This technique of chaining constructors is especially useful when one constructor has some (perhaps quite complex) code associated with setting fields. You would not want to duplicate this code in another constructor and so you would chain the constructors.

Quiz

What is the default constructor for the following class?

```
public class Penny {
    String name = "lane";
}
```

a. `public Penny(String name)`
b. `public Penny()`
c. `class()`
d. `String()`
e. `private Penny()`

Answer: b

Exercise 9-2: Create an Overloaded Constructor

In this exercise, you:

- Add an overloaded constructor to the `Customer` class
- Create a new `Customer` object by calling the overloaded constructor

ORACLE

- Open the Java Code Console and access 09-ManipulateFormat > Exercise2.
- Follow the instructions below the code editor to modify the `Customer` class and then instantiate a Customer object using the new constructor from the `ShoppingCart` class.
- If you need help, click the Solution link. To go back to your code, click the Exercise link again. Any changes that you have made will have been saved.

Summary

In this lesson, you should have learned how to:

- Use public and private access modifiers
- Restrict access to fields and methods using encapsulation
- Implement encapsulation in a class
- Overload a constructor by adding method parameters to a constructor

Play Time!

Play **Basic Puzzle 12** before the next lesson titled “More on Conditionals.”

Consider the following:

What happens if the ball strikes the blade?

You will be asked this question in the lesson titled “More on Conditionals.”

Practice 9-1 Overview: Encapsulating Fields

This practice covers using the NetBeans refactor feature to encapsulate the fields of several classes from the Soccer application.

Practice 9-2 Overview: Creating Overloaded Constructors

This practice covers the following topics:

- Creating overloaded constructors for several classes of the Soccer application
- Initializing fields within the custom constructor methods