Using Encapsulation

Interactive Quizzes

Before you start today's lessons, test your knowledge by answering some quiz questions that relate to yesterday's lessons. Open the Quiz files by clicking the quizzes.html shortcut from the desktop of your VM. In the welcome page, JavaSEProgrammingI.html, click the links for Lessons 6, 7, and 8.

Objectives

After completing this lesson, you should be able 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

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.

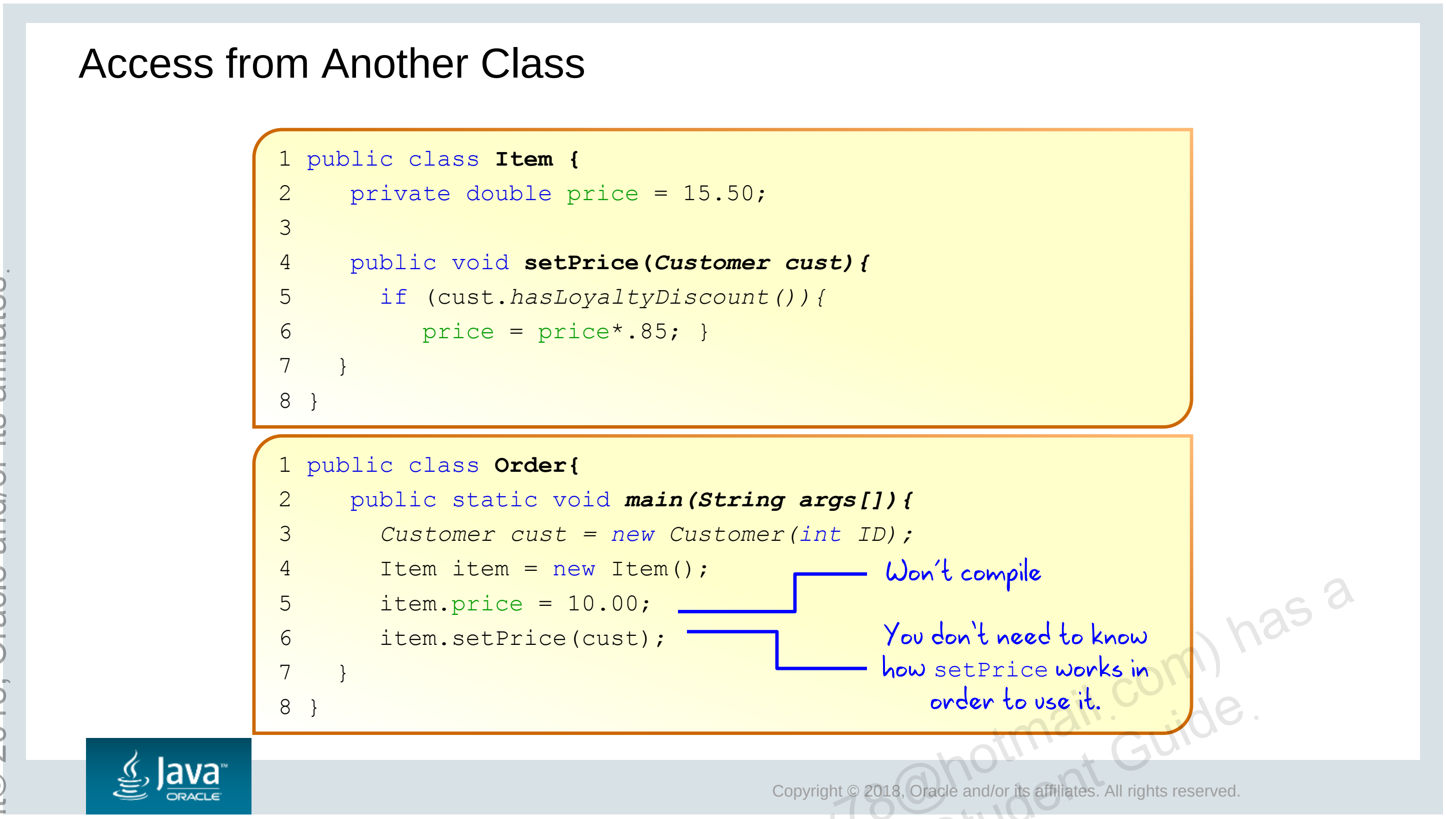
- The public interface is what you see when you look up a class in the JDK API documentation. You get just the information you need in order to use a class. That is, the signatures for public methods, and data types of any public fields.
- The implementation of a class is the code itself, and also any 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

```
public class Item {
   // Base price
  private double price = 15.50;

  public void setPrice(Customer cust){
      if (cust.hasLoyaltyDiscount()){
         price = price*.85; }
  }
}
```

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

```java
public class Item {
   private double price = 15.50;

   public void setPrice(Customer cust){
     if (cust.hasLoyaltyDiscount()){
        price = price*.85; }
   }
}
```

```java
public class Order{
   public static void main(String args[]){
     Customer cust = new Customer(int ID);
     Item item = new Item();
     item.price = 10.00;
     item.setPrice(cust);
   }
}
```

Won't compile

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

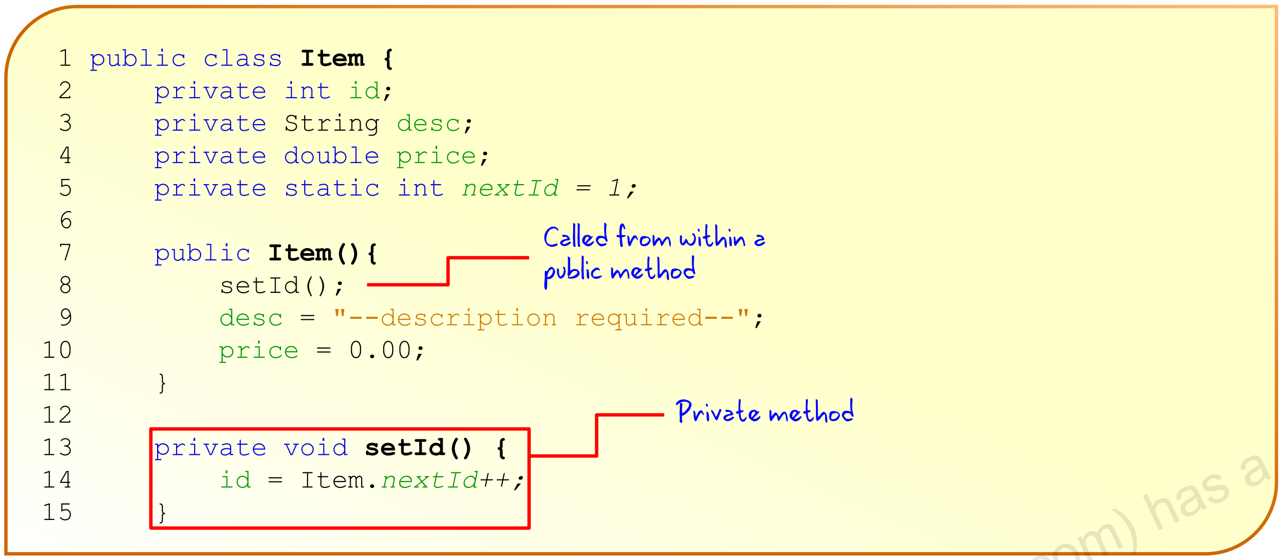
Another Example

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

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

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



```
public class Item {
    private int id;
    private String desc;
    private double price;
    private static int nextId = 1;

    public Item(){
        setId();
        desc = "--description required--";
        price = 0.00;
    }

    private void setId() {
        id = Item.nextId++;
    }
```

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.

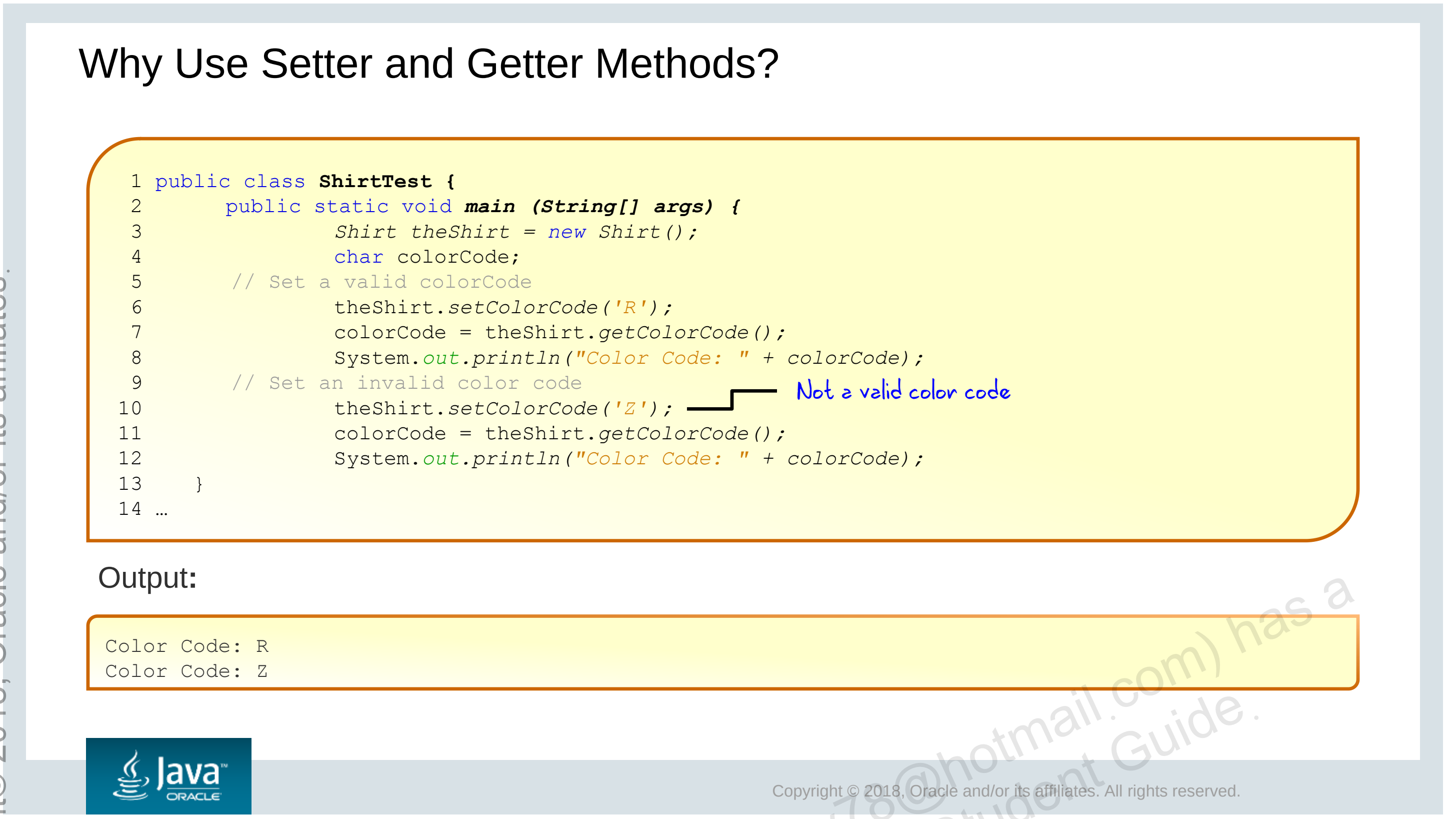
Get and Set Methods

```java
public class Shirt {
   private int shirtID = 0;            // Default ID for the shirt
   private String description = "-description required-"; // default
   private char colorCode = 'U';      //R=Red, B=Blue, G=Green, U=Unset
   private double price = 0.0;        // Default price for all items

   public char getColorCode() {
       return colorCode;
   }
   public void setColorCode(char newCode) {
       colorCode = newCode;
   }
       // Additional get and set methods for shirtID, description,
       // and price would follow

} // 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:

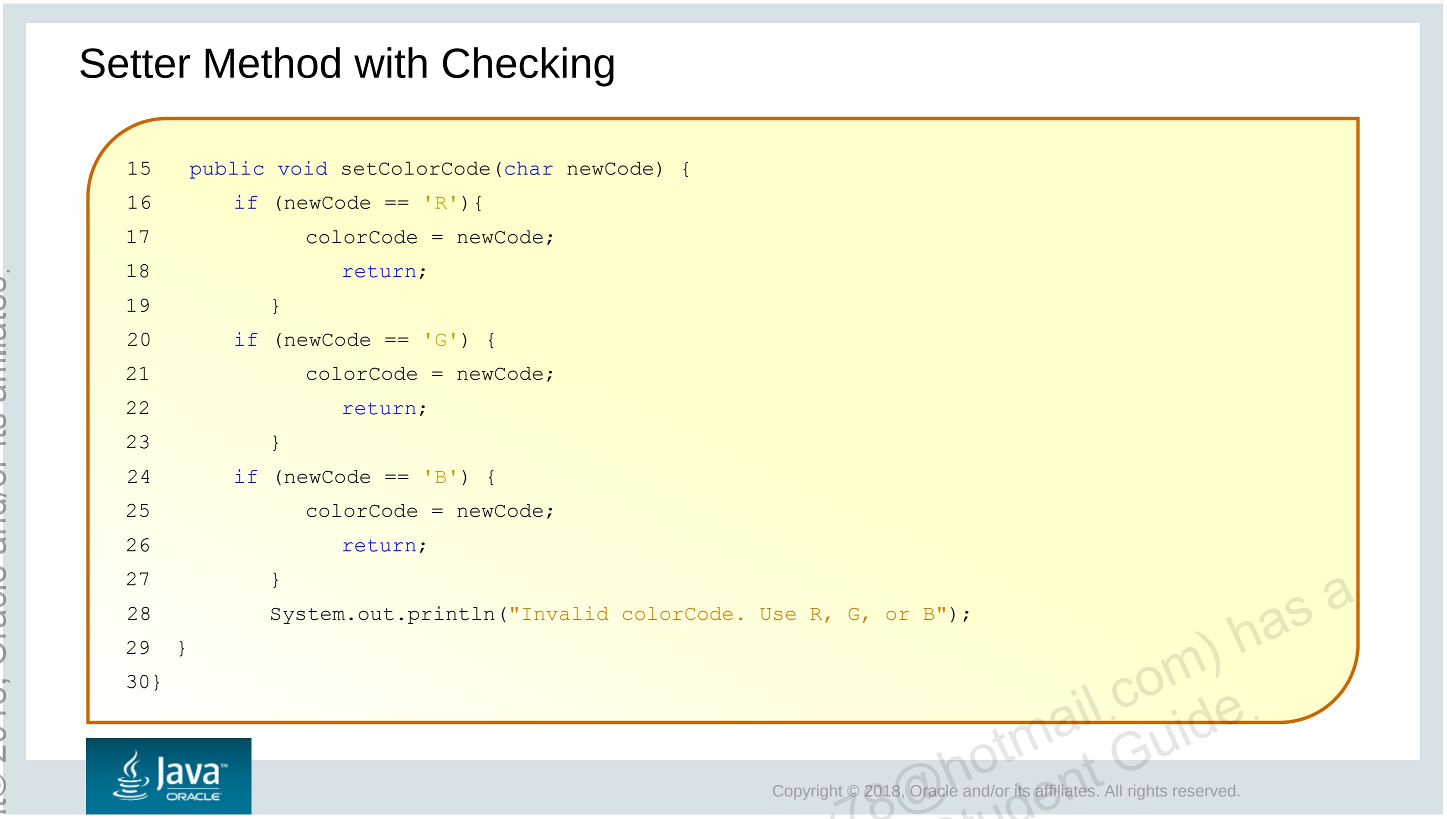
```
Color Code: R
Color Code: Z
```

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

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

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

```
public class ShirtTest {
    public static void main (String[] args) {
        Shirt theShirt = new Shirt();
        System.out.println("Color Code: " + theShirt.getColorCode());

        // Try to set an invalid color code
        theShirt.setColorCode('Z');        Not a valid color code
        System.out.println("Color Code: " + theShirt.getColorCode());
    }
```

Output:

```
Color Code: U          Before call to setColorCode() – shows default value
Invalid colorCode. Use R, G, or B       call to setColorCode prints error message
Color Code: U     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.

1. Open **`Exercise_09-1`** project in NetBeans.
2. Change access modifiers so that fields must be read or modified through public methods.
3. Allow the `name` field to be read and modified.
4. Allow the `ssn` field to be read but not modified (read only).

In this exercise, you encapsulate the Customer class.

Topics

- Access control
- Encapsulation
- Overloading constructors

Initializing a Shirt Object

Explicitly:

```
public class ShirtTest {
    public static void main (String[] args) {
        Shirt theShirt = new Shirt();

        // Set values for the Shirt
        theShirt.setColorCode('R');
        theShirt.setDescription("Outdoors shirt");
        theShirt.price(39.99);
    }
}
```

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

```
public class Shirt {
  public int shirtID = 0;                      // Default ID for the shirt
  public String description = "-description required-"; // default
  private char colorCode = 'U';                // R=Red, B=Blue, G=Green, U=Unset
  public double price = 0.0;                   // Default price all items

 // This constructor takes three argument
  public Shirt(char colorCode, String desc, double price ) {
      setColorCode(colorCode);
      setDescription(desc);
      setPrice(price);
  }
```

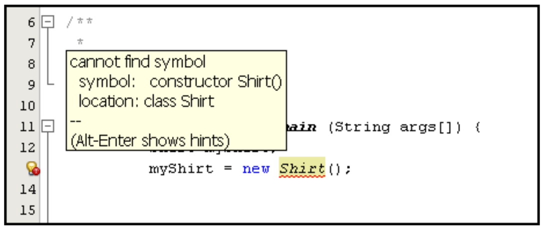
The Shirt example shown in the slide has a constructor that accepts three values to initialize three of the object's fields. Because setColorCode ensures that an invalid code cannot be set, the constructor can just call this method.

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.



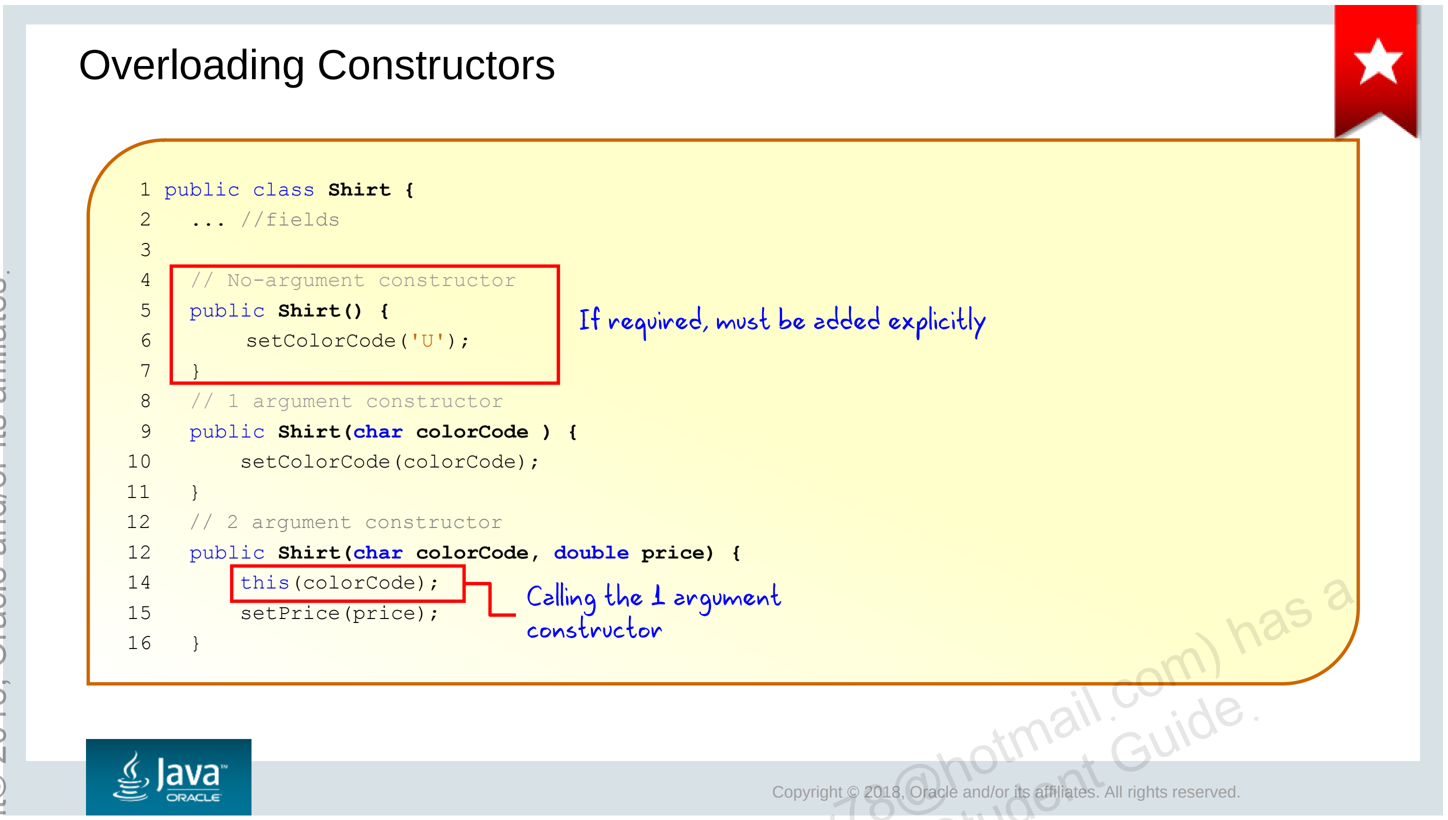
```
/**
 *
       ain (String args[]) {
        myShirt = new Shirt();
```

```
cannot find symbol
  symbol:   constructor Shirt()
  location: class Shirt
--
(Alt-Enter shows hints)
```

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

```
public class Shirt {
  ... //fields

  // No-argument constructor
  public Shirt() {
      setColorCode('U');
  }
  // 1 argument constructor
  public Shirt(char colorCode ) {
      setColorCode(colorCode);
  }
  // 2 argument constructor
  public Shirt(char colorCode, double price) {
      this(colorCode);
      setPrice(price);
  }
```

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

1. Continue editing **`Exercise_09-1`** or open **`Exercise_09-2`** in NetBeans.

In the `Customer` class:

2. Add a custom constructor that initializes the fields.

In the `ShoppingCart` class:

3. Declare, instantiate, and initialize a new `Customer` object by calling the custom constructor.
4. Test it by printing the `Customer` object name (call the `getName` method).

In this exercise, you add a constructor to the Customer class and create a new Customer object by calling the constructor.

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

Practices Overview

- 9-1: Encapsulating Fields
- 9-2: Creating Overloaded Constructors