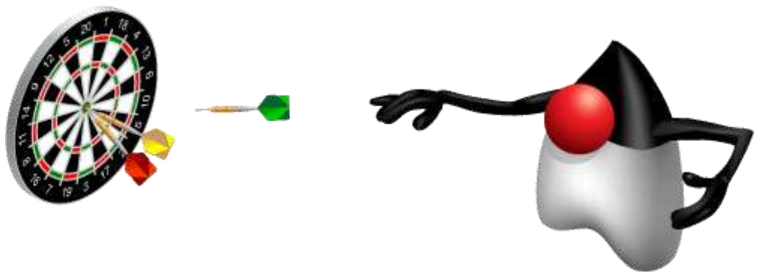
Using Encapsulation

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 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; }
    }
}
```

Access from Another Class

```
public class Item {
   private double price = 15.50;
   public void setPrice(Customer cust){
     if (cust.hasLoyaltyDiscount()){
        price = price*.85; }
  }
}
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.

```java
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);
}
```

Using Access Control on Methods

```java
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++;
    }
```

Called from within a public method

Private method

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
```

Why Use Setter and Getter Methods?

```java
public class ShirtTest {
        public static void main (String[] args) {
        Shirt theShirt = new Shirt();
        char colorCode;
     // Set a valid colorCode
        theShirt.setColorCode('R');
        colorCode = theShirt.getColorCode();
        System.out.println("Color Code: " + colorCode);
     // Set an invalid color code
        theShirt.setColorCode('Z');
        colorCode = theShirt.getColorCode();
        System.out.println("Color Code: " + colorCode);
   }
…
```

Not a valid color code

Output:

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

Setter Method with Checking

```java
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");
   }
}
```

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
            Shirt1.setColorCode('Z');
            System.out.println("Color Code: " + theShirt.getColorCode());
 }
```

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()

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.

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);
```

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.

`Shirt` Constructor with Arguments

```java
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);
  }
```

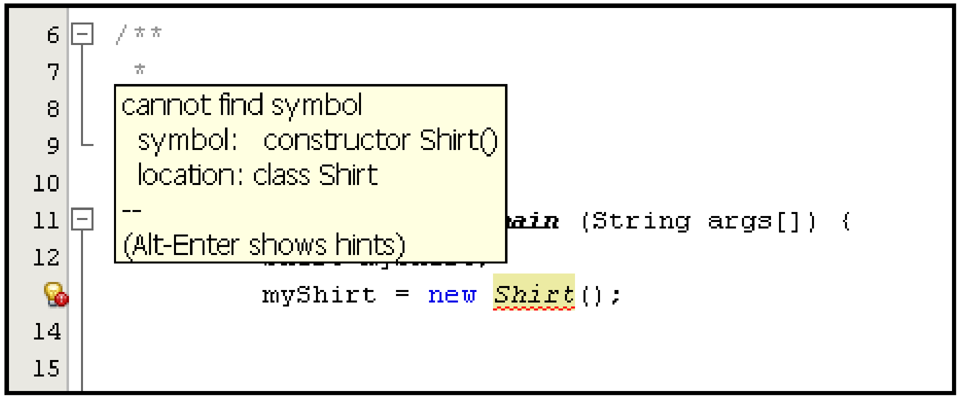
Default Constructor and Constructor with Args

•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)

Overloading Constructors

```java
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

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()`

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

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