# Tutorial for Introduction to Classes and Encapsulation

In this tutorial, you'll build a bookstore application using some of the concepts and principles of encapsulation. You'll create two classes, `Book` and `ShoppingCart`, and use them in the application.

## Step One: Review and run the `Bookstore` class

To get started, import this project into IntelliJ.

Next, open `src\main\java\com\techelevator\Bookstore.java` in the Project Explorer. This is the starting point of the command line bookstore application.

Inside the `main()` method, you'll see code that displays a welcome message, and some "step" comment lines:

```
System.out.println("Welcome to the Tech Elevator Bookstore");
System.out.println();

// Step Three: Test the getters and setters

// Step Five: Test the Book constructor

// Step Nine: Test the ShoppingCart class
```

Before moving on to step two, you'll need to run the application to make sure it works. Right-click on the project in the Project Explorer and select **Run As > Java Application**. You'll see the welcome message:

```
Welcome to the Tech Elevator Bookstore
```

The application terminates once it displays the welcome message.

## Step Two: Create the `Book` class

Now that you've reviewed the starting code, and confirmed the application runs, it's time to create your first class.

You'll start with creating a `Book` class. Each instance of the `Book` class holds the information for one book. Each book has a title, an author's name, and a price.

Remember classes combine, or *encapsulate*, *state* (instance variables) and *behavior* (instance methods). Instead of creating separate variables in the program to hold a book's title, author, and price, you keep all details of a book in one place—in a `Book` object.

Open the `src/main/java` folders, and then right-click on the `com.techelevator` package and select **New > Class**.

Next, type in `Book` as the name. You don't need to modify the default values. Then, press Enter or select "Finish" to create the class.

### Create the instance variables

The new `Book.java` file opens in the editor.

Typically, you begin writing the class with the *instance variables*. Instance variables are the *state* that each `Book` object holds; for this application, that's title, author, and price.

Type the following into `Book.java` within the `class` block to add the instance variables:

```
package com.techelevator;

public class Book {

    // Add the instance variables
    private String title;
    private String author;
    private double price;

}
```

> Note: `package com.techelevator;` was automatically added for you when the class file was created.

Instance variables are declared `private` to safely *encapsulate* state. This restricts access to the internal state of the class by hiding instance variables and only allowing them to be changed through methods that the class makes publicly available.

In other words, a class controls its internal state, and ensures it remains consistent, by declaring its instance variables `private`.

### Create getters and setters

The next step in safely encapsulating state is to provide the `public` methods which allows code external to the class to retrieve the current state and to request changes to it.

This is typically done through *getters* and *setters* which allows external code to get and set state under the control of the class.

Create a getter and setter for the `title` instance variable:

```
public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}
```

Now all access to `title` happens through the getter, `getTitle()`, and the setter, `setTitle()`.

Here are some rules for writing well-formed getters and setters:

- Getters
  - Start with `get` followed by the name of the instance variable.
  - Have no parameters.
  - Always return the same type as the instance variable.
- Setters
  - Start with `set` followed by the name of the instance variable.
  - Always have a single parameter of the same type as the instance variable.
  - Always return `void`.

Finish adding the remaining getters and setters for `author` and `price`:

```
public String getAuthor() {
    return author;
}

public void setAuthor(String author) {
    this.author = author;
}

public double getPrice() {
    return price;
}

public void setPrice(double price) {
    this.price = price;
}
```

The basic `Book` class is now complete.

## Step Three: Test the `Book` getters and setters

Test your getters and setters by creating an instance of `Book`, setting values for `title`, `author`, and `price`, and then getting and displaying the values.

Open `Bookstore.java`, if not already open, and enter the following code below the "Step Three" comment line as shown.

```
// Step Three: Test the getters and setters
Book twoCities = new Book();
twoCities.setTitle("A Tale of Two Cities");
twoCities.setAuthor("Charles Dickens");
twoCities.setPrice(14.99);
System.out.println("Title: " + twoCities.getTitle() + ", Author: " +
twoCities.getAuthor() + ", Price: $" + twoCities.getPrice());
```

Once you've entered and saved the test code, re-run `Bookstore`. You'll see this:

```
Welcome to the Tech Elevator Bookstore

Title: A Tale of Two Cities, Author: Charles Dickens, Price: $14.99
```

## Step Four: Add constructors to the `Book` class

Getters and setters are great, but sometimes they can be awkward to use. It might be better to include the title, author, and price when creating an instance of a `Book` instead of creating the instance and assigning a value to each instance variable one at a time.

Assigning a value to each instance variable one at a time involves additional typing, which might lead to errors. For example, forgetting to set one of the instance variables might put the object into an *inconsistent state*.

Constructors can address these concerns. Constructors allow you to create instances of a class, reduce excess code, and help eliminate inconsistent state.

By default, when no other constructor is defined, Java automatically adds a *default constructor* to your class. Default constructors set all instance variables to `null` for reference types, and `0` or `false` for primitives.

Because you didn't include a constructor in the `Book` class code, the Java generated default constructor for `Book` was called when you created an instance of the class to test the getters and setters:

```
Book twoCities = new Book();
```

Because all the instance variables were set to `null` or `0` by the default constructor, you needed to assign values to them using their setters:

```
twoCities.setTitle("A Tale of Two Cities");
twoCities.setAuthor("Charles Dickens");
twoCities.setPrice(14.99);
```

Fortunately, adding a custom constructor is no more difficult than writing a method. In fact, constructors are just *special* methods.

Like methods, constructors have a name, and optionally have parameters. The differences between constructors and methods are that constructors always have the same name as their class, and they never return anything, not even `void`.

Return to `Book.java`, and type in this new constructor:

```
public Book(String title, String author, double price) {
    this.title = title;
```

```
        this.author = author;
        this.price = price;
    }
```

Because once you provide a custom constructor, Java no longer generates the default one, and there is now only one way to create a new Book by giving the book a title, an author, and a price.

Consequently, you also need to provide the default constructor so the original getter and setter test code continues to work:

```
public Book() {
}
```

## Step Five: Test the Book constructors

To test the Book constructor, create two additional instances of Book using the custom constructor, and display the results:

```
// Step Five: Test the Book constructor
Book threeMusketeers = new Book("The Three Musketeers", "Alexandre Dumas", 12.95);

Book childhoodEnd = new Book("Childhood's End", "Arthur C. Clark", 5.99);

System.out.println("Title: " + threeMusketeers.getTitle() + ", Author: " +
threeMusketeers.getAuthor() + ", Price: $" + threeMusketeers.getPrice());

System.out.println("Title: " + childhoodEnd.getTitle() + ", Author: " +
childhoodEnd.getAuthor() + ", Price: $" + childhoodEnd.getPrice());
```

The following results are displayed after you've entered and saved the test code and re-run `Bookstore`:

```
Welcome to the Tech Elevator Bookstore

Title: A Tale of Two Cities, Author: Charles Dickens, Price: $14.99
Title: The Three Musketeers, Author: Alexandre Dumas, Price: $12.95
Title: Childhood's End, Author: Arthur C. Clark, Price: $5.99
```

## Step Six: Create String representation of Book object

Classes frequently have a method which builds a string representation of the values of the instance variables. This can be extremely useful for testing and debugging purposes, but it's also convenient to have a way to consistently display the contents of an object.

It also helps to eliminate duplicate code by building the string representation in a single method rather than repeatedly concatenating strings as you've seen in the testing code.

Add the following method to the `Book` class:

```
public String bookInfo() {
    return "Title: " + title + ", Author: " + author +
    ", Price: $" + price;
}
```

> Note: You'll learn the standard Java way of building string representations using the `toString()` method in the Inheritance chapter.

This completes the `Book` class. You now have a class that contains all the information needed to represent a book, has a convenient constructor, and a way to form a consistent string representation of it.

## Step Seven: Test the `bookInfo` method

Modify `Bookstore.java` by replacing the separate string concatenations in the three `System.out.println()` lines with the following code:

```
...
// System.out.println("Title: " + twoCities.getTitle() + ", Author: " +
twoCities.getAuthor() + ", Price: $" + twoCities.getPrice());
System.out.println(twoCities.bookInfo());
...
// System.out.println("Title: " + threeMusketeers.getTitle() + ", Author: " +
threeMusketeers.getAuthor() + ", Price: $" + threeMusketeers.getPrice());
//System.out.println("Title: " + childhoodEnd.getTitle() + ", Author: " +
childhoodEnd.getAuthor() + ", Price: $" + childhoodEnd.getPrice());
System.out.println(threeMusketeers.bookInfo());
System.out.println(childhoodEnd.bookInfo());
```

Save the replaced code, and re-run `Bookstore`. You'll see the same output you saw when testing the constructor:

```
Welcome to the Tech Elevator Bookstore

Title: A Tale of Two Cities, Author: Charles Dickens, Price: $14.99
Title: The Three Musketeers, Author: Alexandre Dumas, Price: $12.95
Title: Childhood's End, Author: Arthur C. Clark, Price: $5.99
```

## Step Eight: Create the `ShoppingCart` class

Now, you need to work on the Shopping Cart.

The `ShoppingCart` class represents a collection of books that a customer wants to buy.

Once again, open the `src/main/java` folders in the Project Explorer. Right-click on the `com.techelevator` package and select **New > Class**.

Next, type in `ShoppingCart` as the name. You don't need to modify the default values. Then, press Enter or select "Finish" to create the class.

You need a single instance variable to store the book objects in the shopping cart. Enter the following code into `ShoppingCart.java`:

```
private List<Book> booksToBuy = new ArrayList<>();
```

> Note: You'll need to import `java.util.List` and `java.util.ArrayList`.

You may've noticed that `Book` is missing from `ArrayList<>`. Recall the *diamond operator* `<>` is a bit of *syntactic sugar* that says, "Use the data type given in the `List<T>`."

Since there is only the one instance variable, and it's automatically instantiated as an empty list of books, you don't need a custom constructor. The default constructor works fine.

However, books do need to be added to the list, so the `ShoppingCart` needs an `add()` method. The method takes an instance of `Book` that's the book to add to the list.

Type the following code in `ShoppingCart.java`:

```
public void add(Book bookToAdd) {
    booksToBuy.add(bookToAdd);
}
```

The `ShoppingCart` also needs to get the total price for all the books currently in the cart. This is a derived property calculated by looping through the list of books and adding each book's price to the total purchase amount.

Add the following code to create a new method in the `ShoppingCart` class called `getTotalPrice()`:

```
public double getTotalPrice() {
    double total = 0.0;
    for(Book book : booksToBuy) {
        total += book.getPrice();
    }
    return total;
}
```

Finally, the `ShoppingCart` needs a `receipt()` method that returns a receipt-like string representation of the shopping cart, with a listing of all the books in the cart and, at the end, the total price of everything in the cart:

```
public String receipt() {
    String receipt = "\nReceipt\n";
    for(Book book : booksToBuy) {
        receipt += book.bookInfo();
        receipt += "\n";
    }

    receipt += "\nTotal: $" + getTotalPrice();

    return receipt;
}
```

> Note the use of `book.bookInfo()` to build the string representation of the `book` object, and `getTotalPrice()` to calculate the derived property for the receipt.

The `ShoppingCart` is complete.

Take a minute to review how the state of `ShoppingCart` is encapsulated. The instance variable `booksToBuy` is declared `private` with the only way to alter it through the public `add()` method.

You saw this same pattern of `private` instance variables and `public` methods in the `Book` class, and you'll see it in almost every class you write.

The `ShoppingCart`'s state is further encapsulated with the `public` derived property, `getTotalPrice()`. Remember, safely encapsulating an object is not just hiding data, but also keeping implementation details firmly embedded in the class.

## Step Nine: Test the `ShoppingCart` class

To test the `ShoppingCart`, edit `Bookstore.java` to add the three instances of the `Book` already created to a new `ShoppingCart`, then call `receipt()` and display the returned string representation of the shopping cart:

```
// Step Nine: Test the ShoppingCart class
ShoppingCart shoppingCart = new ShoppingCart();
shoppingCart.add(twoCities);
shoppingCart.add(threeMusketeers);
shoppingCart.add(childhoodEnd);
System.out.println(shoppingCart.receipt());
```

Re-run `Bookstore` after adding and saving the test code. You'll see the following output:

```
Welcome to the Tech Elevator Bookstore

Title: A Tale of Two Cities, Author: Charles Dickens, Price: $14.99
Title: The Three Musketeers, Author: Alexandre Dumas, Price: $12.95
Title: Childhood's End, Author: Arthur C. Clark, Price: $5.99

Receipt
```

```
Title: A Tale of Two Cities, Author: Charles Dickens, Price: $14.99
Title: The Three Musketeers, Author: Alexandre Dumas, Price: $12.95
Title: Childhood's End, Author: Arthur C. Clark, Price: $5.99

Total: $33.93
```

## Summary

After completing this tutorial, you should understand:

- How to create a basic Java class.
- How to safely encapsulate state through the use of `private` and `public` access modifiers.
- How to write constructors, and their role in furthering encapsulation.
- How to create instances of a class, and make use of their `public` methods.