Tutorial for inheritance

In this tutorial, you'll continue to build a bookstore application using inheritance. Due to the success of your previous bookstore application, the store is growing and looking to expand its product line. In addition to selling books, the store is going to start selling movies. In the future, the owners hope to expand into music and audio books, among other media.

You need to modify your application to handle movies, with the flexibility to add more product lines in the future.

To get started, import this project into IntelliJ. You'll start your work in the file `Book.java`.

Design

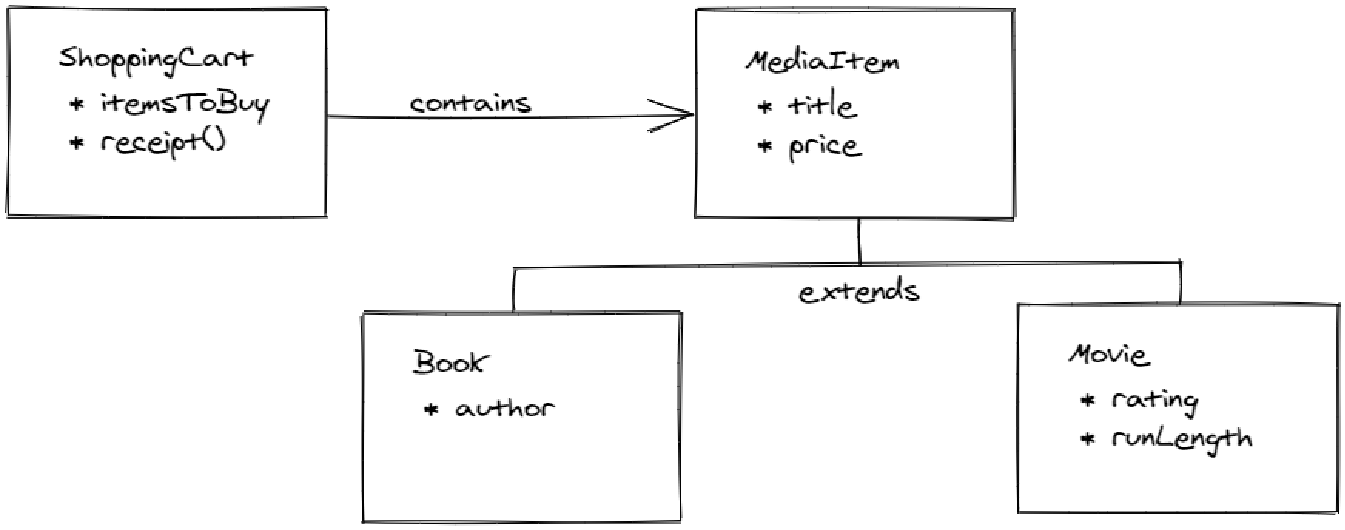
You need a new class to represent a `Movie`. A movie has properties for `title`, `rating`, `runLength`, and `price`. Recall that the `Book` class has properties `title`, `author`, and `price`.

An author doesn't apply to a movie, nor does a run-length apply to a book. Books and movies are similar in that they have a *title* and *price*. Any item the store sells needs to have these attributes.

Your job is to create a new *superclass* called `MediaItem`, which has `title` and `price` properties. `Book` *extends* `MediaItem` and adds an `author` property. A new class called `Movie` also extends `MediaItem`, adding all the movie-specific attributes.

Because `MediaItem` now represents a "thing to buy", the collection in your `ShoppingCart` class must also change from a list of `Book`s to a list of `MediaItem`s.

Your new class hierarchy looks like this:



Whenever you write code that uses inheritance, use the "is-a" test. Can you say that a book IS A media item, and that a movie IS A media item? Considering what the bookstore sells, these statements are true.

Step One: Use `toString()` for a string representation of `Book`

In the previous tutorial, you needed to represent a `Book` object in a printed receipt, so you created a method called `bookInfo()`, which returns details about a book as a `String`. The `ShoppingCart` class calls this when printing a receipt.

The `Object` class already defines a method for this purpose. The `toString()` method exists to provide a string representation of any object, which is exactly what you did in the `bookInfo()` method.

Every class either directly or indirectly extends `Object`. If the class definition doesn't contain the `extends` keyword, then the class directly extends `Object`. You can think of `Object` as being at the top of the class hierarchy, or the "ultimate superclass."

Rename `bookInfo()` to `toString()`

This means that your `Book` class IS A `Object`. So you can *override* `toString()` in the `Book` class to provide a string representation of a book. Since you already have the code for this, you just need to rename that method.

In `Book.java`, find the `bookInfo()` method near the bottom, and rename the method to `toString()`:

```java
// Book.java
@Override
public String toString() {
    return "Title: " + title + ", Author: " + author + ", Price: $" + price;
}
```

> Note the `@Override` annotation preceding the `toString()` method. Leaving it out won't prevent your code from compiling, but it's a Java best practice to include it. It indicates to the compiler and other developers that this class replaces the superclass method. If something about the superclass method were to change—such as changing parameters or renaming it—the compiler warns you about that so you can make the appropriate change to the overriding method.

Now in `ShoppingCart.java`, inside the `receipt()` method, change the call to the method `bookInfo()` into a call to the method `toString()`:

```java
// ShoppingCart.java
public String receipt() {
    ...
    for (Book book : booksToBuy) {
        receipt += book.toString();
        receipt += "\n";
    }
    ...
}
```

If you run the program, you get the same output as you saw in the previous tutorial:

```
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
```

Remove the explicit call to `toString()`

In the previous step, you renamed a method. However, now that you used the proper method, `toString()`, you get an added bonus. You don't have call that method explicitly. When your code needs a string representation of your object, the Java compiler calls the object's `toString()` method.

Try this—in the `ShoppingCart.receipt()` method, remove the explicit call to `toString()`:

```
// ShoppingCart.java
public String receipt() {
    ...
    for (Book book : booksToBuy) {
        receipt += book;        // toString() explicit call removed
        receipt += "\n";
    }
    ...
}
```

When you run the program again, you see the same output. That's because in the line `receipt += book;`, your code needed a string to represent `book`, so the Java compiler called `book.toString()` implicitly.

Step Two: Refactor the `Book` class into `MediaItem` and `Book`

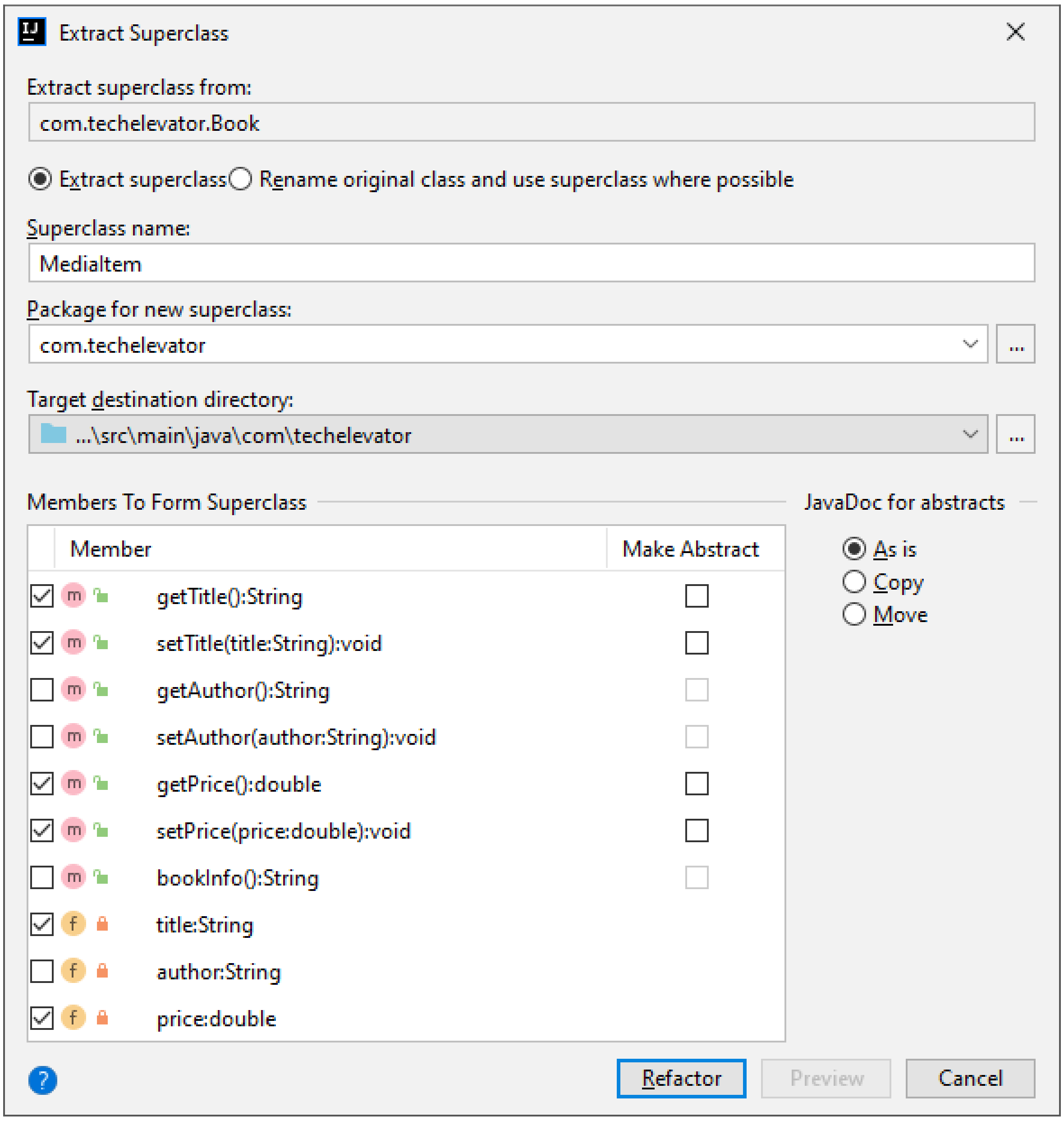
Now you must *refactor* your book code. Refactoring means that you change the form of the code, but not the functionality. To refactor `Book` into a superclass and subclass, you need to:

1. Create the `MediaItem` superclass, and move the properties `title` and `price` as well as their getters and setters from `Book` into `MediaItem`.
2. Change `Book` so that it *extends* `MediaItem`.
3. Remove the `title` and `price` and their getters and setters from `Book`, since `Book` now *inherits* them from the superclass.
4. Change `ShoppingCart` to hold a list of `MediaItem` objects instead of `Book`s.
5. Rename the variable `booksToBuy` to `itemsToBuy` to properly reflect that the bookstore sells more than books.

Split the classes

IntelliJ includes some *refactoring tools* that can help you do this type of work. To split `Book` into a superclass and subclass, right-click on the `Book` class in the Project window. Then select **Refactor > Extract Superclass...** from the menu.

In the dialog box that appears, type `MediaItem` as the Superclass name, and in the list, check the `title` and `price` properties and the `getTitle()`, `setTitle()`, `getPrice()`, and `setPrice()` methods. These are the members to remove from `Book` and add to the new superclass, `MediaItem`. Click the **Refactor** button:



Extract superclass from: com.techelevator.Book

[✓ Extract superclass] Rename original class and use superclass where possible

Superclass name: MediaItem

Package for new superclass: com.techelevator

Target destination directory: ...\src\main\java\com\techelevator

Members To Form Superclass

	Member	Make Abstract
✓	getTitle():String	☐
✓	setTitle(title:String):void	☐
☐	getAuthor():String	☐
☐	setAuthor(author:String):void	☐
✓	getPrice():double	☐
✓	setPrice(price:double):void	☐
☐	bookInfo():String	☐
✓	title:String	
☐	author:String	
✓	price:double	

JavaDoc for abstracts: [✓ As is] Copy Move

If IntelliJ asks about replacing usages, click **No**. If you're prompted to add the file to git, choose **Cancel**.

> Note: What `refactor` > `Extract Superclass` did
>
> You asked IntelliJ to refactor your code by extracting a superclass from the code so that you now have a superclass (`MediaItem`) and a subclass (`Book`). Look at your code to see the changes.
>
> - See the new file, `MediaItem.java`, which contains a new class, `MediaItem`. This class contains the members: `title`, `getTitle()`, `setTitle()`, `price`, `getPrice()`, and `setPrice()`.
> - Look in `Book.java` and notice that:
>   - It now *extends* `MediaItem`.
>   - `title`, `price` and their related methods no longer exist in the subclass.

Modify the collection in `ShoppingCart`

`ShoppingCart` has a private member variable called `booksToBuy`. To allow your shopping cart to hold more than just books, you must do two things:

1. Change the type of `booksToBuy` from `List<Book>` to `List<MediaItem>`.
2. Change the name of `booksToBuy`, since that name no longer accurately reflects what the list holds.

First, rename the variable using refactor tools. Right-click on the variable `booksToBuy` in its declaration, then select **Refactor > Rename...**. Then, without clicking anything else, type `itemsToBuy`.

IntelliJ replaces the variable name not only on the line you're typing in, but for every usage of the variable `booksToBuy`. Press `Enter` when you finish typing to accept the change.

Now, change the type in the declaration from `List<Book>` to `List<MediaItem>`:

```
// ShoppingCart.java
private List<MediaItem> itemsToBuy = new ArrayList<>();
```

Change the `add()` method to reflect the type of item to add:

```
// ShoppingCart.java
public void add(MediaItem itemToAdd) {
    itemsToBuy.add(itemToAdd);
}
```

Modify the two foreach loops to reflect the type change. First, modify the method `getTotalPrice()`:

```
// ShoppingCart.java
for (MediaItem item : itemsToBuy) {
    total += item.getPrice();
}
```

Next, modify the `receipt()` method:

```
// ShoppingCart.java
for (MediaItem item : itemsToBuy) {
    receipt += item;
    receipt += "\n";
}
```

Now run your program. You see the exact same output you saw originally.

You've done a lot of work here, and to an outside observer, nothing has changed. This is *exactly* what refactoring is about. You haven't changed what the code *does*. But you have changed *how it's done*. You've made your code more adaptable.

Step Three: Add the `Movie` class

This adaptability becomes apparent when you add the `Movie` class. `Movie` is another subclass of `MediaItem`.

Add a class

Add a new class by right-clicking on the `com.techelevator` package in Project window and selecting **New > Java Class**. In the dialog, type "Movie" and press **Enter**. IntelliJ creates a new file called `Movie.java` which contains the new class.

Extend `MediaItem`

In `Movie.java`, update the class definition so `Movie` extends `MediaItem`:

```java
// Movie.java
public class Movie extends MediaItem {
}
```

Add properties and methods

In addition to a title and a price—which `Movie` inherits from `MediaItem`—movies have a rating and run-length. Add these properties with their getters and setters:

```java
// Movie.java
public class Movie extends MediaItem {
    private String rating;
    private int runLength;

    public String getRating() {
        return rating;
    }
    public void setRating(String rating) {
        this.rating = rating;
    }
    public int getRunLength() {
        return runLength;
    }
    public void setRunLength(int runLength) {
        this.runLength = runLength;
    }
}
```

Add a constructor

While not necessary, constructors make it convenient to create and initialize an object in a single line of code. It's a good idea to create one or more.

Add a constructor to the `Movie` class to initialize a movie at creation time:

```
// Movie.java
public Movie(String title, String rating, int runLength, double price) {
    this.title = title;
    this.rating = rating;
    this.runLength = runLength;
    this.price = price;
}
```

Override `toString()`

Remember that `toString()` is the conventional way to create a string representation of any object. You need this so that on the receipt, it's clear what movie is in the cart.

In `Movie.java`, override the `toString()` method:

```
// Movie.java
@Override
public String toString() {
    return "Title: " + title + ", Rating: " + rating + ", Time: " + runLength + "
minutes, Price: $" + price;
}
```

That's all you need to do to sell movies in your bookstore. Because you created a general class to represent any purchasable item in the store, adding a specific type of item such as a movie doesn't take a lot of effort.

Buy some movies

Now you can create a new movie and add it to the shopping cart. Do this in `Bookstore.java`:

```
// Bookstore.java
// Add some new movies and purchase them
Movie toyStory = new Movie("Toy Story", "G", 81, 19.99);
shoppingCart.add(toyStory);

// Shirley, you can't be serious!
Movie airplane = new Movie("Airplane!", "PG", 88, 14.99);
shoppingCart.add(airplane);

System.out.println(shoppingCart.receipt());
```

In the output window, you see a receipt that shows *both* books and movies:

```
Welcome to the Tech Elevator Bookstore

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
Title: Toy Story, Rating: G, Time: 81 minutes, Price: $19.99
Title: Airplane!, Rating: PG, Time: 88 minutes, Price: $14.99

Total: $68.91
```

Next steps

If you want further practice, implement another product line in your bookstore. For example, add music as a new type of media item. Then open `Bookstore.java`, and add some music to your shopping cart.

> Hint: repeat step three, substituting your new class and its properties for `Movie`.

Summary

In this tutorial, you:

- Used inheritance by refactoring a class into a superclass and subclass.
- Created a new class that extends an existing class.
- Implemented `toString()` to override the method in `Object`.