In the previous lab, you laid the foundation for a virtual coffee machine by creating a hierarchy of coffee beverages. Now, it's time to elevate your barista skills by delving deeper into object-oriented programming concepts like polymorphism and separation of concerns.

The coffee machine you worked on needs to be upgraded to a sophisticated machine capable of brewing various coffee drinks with precision. Each coffee type must now have its unique preparation. As a star performer on your team, you are entrusted with creating and implementing this upgrade.

Goal

Upgrade your virtual coffee machine by implementing polymorphism and creating a utility class to handle different coffee types and their preparation processes efficiently. This will make your virtual coffee machine versatile for preparing different coffee types while maintaining code efficiency and flexibility.

By the end of this lab, you should have a deep understanding of polymorphism, method overriding, and separation of concerns. These skills are fundamental to building robust and scalable software applications. As a software developer, you'll encounter similar challenges when working on various projects during your career. This lab provides a solid starting point for developing essential problem-solving and coding abilities.

So, time to put on your coding hat and supercharge your coffee!

🖥️ Note: When encountering this icon, it's time to get into your IDE and start coding!

In your lab environment, open IntelliJ by double-clicking on the icon.

You'll create a more sophisticated coffee machine simulation by refining the provided code and applying advanced OOP concepts. Here's the plan:

1. **Review the starter code: The starter code contains the `Coffee` superclass, `Espresso`, and `Latte` subclasses and the main class `CoffeeMachine`. This starter code is the solution you coded for your previous lab, *Creating multiple classes*.**
2. **Refine subclasses to implement method overriding: To make coffee preparation more specific, override the `grindBeans()` and `brewCoffee()` methods in both subclasses. Also, override the `printInfo()` method in both subclasses to include the functionality of the `printEspressoDetails()` and `printLatteDetails()` methods.**
3. **Design a utility class to implement separation of concerns: To capture common coffee-making logic, you will create a utility class named `CoffeeMaker`. This class should have a method named**

`prepareCoffee()` that takes a `Coffee` object as an argument and calls the appropriate methods to prepare a specific type of coffee.

4. Demonstrate polymorphism: Use the `CoffeeMaker` class to demonstrate polymorphism. Pass the objects of `Espresso` and `Latte` to the `prepareCoffee()` method to determine how the correct methods are called based on the object type.

In your lab environment, open IntelliJ by double-clicking on the icon.

Expand the *src* folder and review the provided classes.

- `Coffee` superclass:
  - You already know that four attributes define `Coffee`: `name`, `roast`, `caffeineLevelInMg`, and `price`.
  - The parameterized constructor in the `Coffee` class assigns respective values to the `name`, `roast`, and `price` attributes and

then calls the `setCaffeineLevel()` method to set the caffeine level based on the chosen roast.
  - **The method `setCaffeineLevel()` checks the `roast` attribute value using an if-else-if statement and assigns the appropriate value to the `caffeineLevel` attribute.**
  - **The `grindBeans()` and `brewCoffee()` methods simulate the general coffee preparation, and the `printInfo()` method is defined to print the coffee's `name`, `roast`, and `caffeinLevelInMg` in a formatted way.**
- **`Espresso` subclass:**
  - **It inherits all the attributes of `Coffee` and has its unique int type attribute named `numberOfShots` to store the number of servings for the ordered espresso.**
  - **The parameterized constructor uses `super` to assign values to `name`, `roast`, and `price` and initializes `numberOfShots` using `this`.**
  - **The `printEspressoDetails()` method prints a message about the number of servings the user orders, the cost per serving, and their total bill.**
- **`Latte` subclass:**
  - **It inherits all the attributes of `Coffee` and has two unique String type attributes, `milkType` and `syrupFlavor`.**
  - **The parameterized constructor uses `super` to assign values to `name`, `roast`, and `price` and initializes `milkType` and `syrupFlavor` using `this`.**
  - **The `printLatteDetails()` method prints the milk type and syrup flavor used to prepare the latte and the total bill.**
- **`CoffeeMachine` class:**
  - **It contains the `main()` method that displays the menu to select the type of coffee you want to brew (`Espresso` or `Latte`) or exit the coffee machine and prompts for more information based on the coffee type chosen.**
  - **It also contains the already created objects of `Espresso` and `Latte` along with respective method calls.**

After reviewing the starter code, you can refine the **`Espresso`** subclass for specialized coffee preparation and printing the information.

🖥️ It's time to get coding!

Step 1: Override methods in the Espresso subclass for specific behaviors

Espresso is a concentrated coffee shot for which the coffee beans are finely grounded and then brewed with hot water under high pressure. You must display this specialized grinding and brewing behavior in your **`Espresso`** class. To begin, open the **`Espresso`** class.

- 🖥️ TODO 1: Override the **`grindBeans()`** method to simulate grinding the espresso beans finely. For example, the method may print “Grinding the espresso beans finely…”.
- 🖥️ TODO 2: Override the **`brewCoffee()`** method to simulate brewing under high pressure. For example, the method may print “Brewing the espresso under high pressure…”.
- 🖥️ TODO3: Override the **`printInfo()`** method to print the common and specific details of espresso including the bill.
- 🖥️ TODO 4: The **`printInfo()`** method in the **`Coffee`** class already prints the common coffee details in a formatted way, so call it using **`super`**.
- 🖥️ TODO 5: Cut the print statements from the **`printEspressoDetails()`** method and paste them here.
- 🖥️ TODO 6: Delete the **`printEspressoDetails()`** method because it is not needed anymore.

Step 2: Grind and brew an espresso

To verify that your **`Espresso`** class works as expected, complete the following TODOs in the **`main()`** method inside the **`CoffeeMachine`** class:

- 🖥️ TODO 7: Create an **`Espresso`** object using the parameterized constructor and pass the already declared variables as arguments (in the precise order).
    - For example, **`Espresso myFavoriteEspresso = new Espresso(espressoName, espressoRoast, espressoPrice, numberOfShots);`**
- 🖥️ TODO 8: Call the **`grindBeans()`** method on the **`Espresso`** object using the dot operator.
- 🖥️ TODO 9: Call the **`brewCoffee()`** method on the **`Espresso`** object using the dot operator.
- 🖥️ TODO 10: Call the **`printInfo()`** method on the **`Espresso`** object using the dot operator.

Great! So far, you’ve refined your **`Espresso`** class and created its object to call all the overridden methods. Run your code in the IDE to confirm that the overridden methods work fine. The program should give you a menu as before, and since you’ve only modified the **`Espresso`** class, choose the first option, Espresso, to observe the modified output.

Now, refine the Latte subclass to reflect the specific behaviors.

🖥️ It’s time to get coding!

Step 1: Override methods in the Latte subclass for specific behaviors

A latte is a coffee drink for which the coffee beans are coarsely ground (medium grind, as they call it) and then brewed with hot water under no pressure. It is prepared with steamed milk, a layer of milk foam, and optionally flavored with syrup (for example, vanilla or caramel).

You must represent this specialized grinding and brewing behavior in your **`Latte`** class.

- 🖥️ TODO 11:Open the **`Latte`** class and override the **`grindBeans()`** method to simulate the coarse grinding of beans for a latte. For example, the method may print “Grinding coffee beans coarsely for a latte (medium grind)”.
- 🖥️ TODO 12: Override the **`brewCoffee()`** method to:
  - Simulate brewing coffee for a latte.
  - Check if **`syrupFlavor`** is selected.
  - If yes, add syrup flavor,
  - steam milk,
  - combine coffee with steamed milk, and
  - add a layer of foam.

Here is the method skeleton for **`brewCoffee()`** with comments to guide you:

```
@Override

public void brewCoffee() {

// TODO 12a: simulate brewing coffee for a latte

System.out.println("Brewing coffee for a latte...");

// TODO 12b: condition to check if syrupFlavor is selected

if (!syrupFlavor.equals("no)")) {

// TODO 12c: simulate adding the syrup

System.out.println("Adding " + syrupFlavor + " syrup to the cup...");

}



// TODO 12d: simulate steaming milk
// TODO 12e: simulate combining coffee and steamed milk



// TODO 12f: simulate adding a layer of foam on top

}
```

- 🖥️ TODO 13: Override the `printInfo()` method to print the common and specific details of the latte, including the bill:
  - 🖥️ TODO 14: The `printInfo()` method in the `Coffee` class already prints the common coffee details in a formatted way, so call it using `super`.
  - 🖥️ TODO 15: Cut the print statements from the `printLatteDetails()` method and paste them here.
- 🖥️ TODO 16: Delete the `printLatteDetails()` method because it is not needed anymore.

Step 2: Grind and brew a latte

To verify that your `Latte` class works as expected, complete the following TODOs in the `main()` method inside the `CoffeeMachine` class.

- 🖥️ TODO 17: Create a `Latte` object using the parameterized constructor and pass the already declared variables as arguments (in the precise order).
  - For example, `Latte myFavoriteLatte = new Latte(LatteName, LatteRoast, LattePrice, milkType, syrupFlavor);`
- 🖥️ TODO 18: Call the `grindBeans()` method on the `Latte` object using the dot operator.
- 🖥️ TODO 19: Call the `brewCoffee()` method on the `Latte` object using the dot operator.
- 🖥️ TODO 20: Call the `printInfo()` method on the `Latte` object using the dot operator.

Awesome! You've also refined your `Latte` class and created its object to call all the overridden methods. Run your code in the IDE to confirm that the overridden methods work fine. The program should give you a menu as before, and since you've modified the Latte class, choose the second option, `Latte`, to observe the modified output.

Before diving into Task 4, take a moment to consider the principle of separation of concerns. How can you ensure that each part of your coffee machine program has a single responsibility? You can create more maintainable and modular code by isolating the coffee preparation logic in a dedicated utility class.

Tip

The video *Grouping methods* will help you to recall the purpose of creating a utility class. The video *Using polymorphic design* will be useful if you need to revisit the importance of the principle of separation of concerns while designing an application.

Reflect on how this approach can simplify future modifications and enhance code reusability.

To implement the separation of concerns, create a utility class named **`CoffeeMaker`**. This class will contain the common coffee-making logic that can be used to perform the coffee-making process for any type of coffee. Creating a utility class helps you implement code reusability and separation of concerns.

🖥️ It's time to get coding!

- 🖥️ TODO 21: Create a new **`CoffeeMaker`** class.
- 🖥️ TODO 22: Create a method called **`prepareCoffee`** that takes a **`Coffee`** object as an argument and executes the proper coffee-making process. This method must be of type **`void`** because it won't return any value.
- 🖥️ TODO 23: Call the **`grindBeans()`** method on the **`Coffee`** object using the dot operator.
- 🖥️ TODO 24: Call the **`brewCoffee()`** method on the **`Coffee`** object using the dot operator.

Here is the skeleton of the class with comments to guide you:

```java
// TODO 21: create the CoffeeMaker class

public class CoffeeMaker {



    // TODO 22: create a void method that takes a Coffee object as an argument

    public void prepareCoffee(Coffee myFavoriteCoffee) {



        // TODO 23: call the grindBeans() method on the Coffee object
        // TODO 24: call the brewCoffee() method on the Coffee object



    }

}
```

It's time to finally observe how the `CoffeeMaker` class utilizes polymorphism to handle different coffee types. You've already created the objects of the `Espresso` and `Latte` subclasses. However, before starting on the next coding steps mentioned below, you must complete an important task.

You must comment out the code under TODO 8, TODO 9, TODO 10, TODO 18, TODO 19, and TODO 20 because the `grindBeans()` and `brewCoffee()` methods are now being called by the `prepareCoffee()` method in the `CoffeeMaker` class. So, you do not need to call them individually for your `Espresso` and `Latte` objects. The `printInfo()` method also does not need to be called here.

Tip

How do you comment out code?

To comment out a single line of code, you can simply type // (double slash) characters at the beginning of the line of code. For example,

```
// myFavoriteEspresso.grindBeans();
```

The comments tell Java that this particular line in the code must not be executed.

Why not delete the code, you ask? Sometimes, you'd like to comment out code instead of deleting it so that you can verify that the rest of the code works well.

🖥️ **It's time to get coding!**

Now, complete the following steps inside the `main()` method in the provided `CoffeeMachine` class:

- 🖥️ TODO 25: Create the object of `CoffeeMaker` class. For example: `CoffeeMaker cafeCoffeeMaker = new CoffeeMaker();`
    - This comment is at the beginning of the `main()` method after the `Scanner` class object has been created. The creation of the `CoffeeMaker` object is not condition-dependent, and it will be required inside both cases in the `switch` statement; therefore, you should create it outside the `switch` statement and the while loop.
- 🖥️ TODO 26: Call the `prepareCoffee()` method on your `CoffeeMaker` object and pass the `Espresso` object as an argument.
- 🖥️ TODO 27: Call the `printInfo()` method on your `Espresso` object using the dot operator.
- 🖥️ TODO 28: Call the `prepareCoffee()` method on your `CoffeeMaker` object and pass the Latte object as an argument.
- 🖥️ TODO 29: Call the `printInfo()` method on your `Latte` object using the dot operator.

Remember
- When you create Espresso and Latte objects, they are essentially Coffee objects due to inheritance.
- When you pass an Espresso or Latte object to the prepareCoffee() method, the correct overridden grindBeans() and brewCoffee() methods are invoked based on the actual type of object at runtime.
- Although the prepareCoffee() method is called twice for different coffee options in the switch case statement, it is actually executed only once based on your coffee selection and using the appropriate Coffee type object as the argument. This ensures proper implementation of polymorphism in the code.

You've performed all the required steps to enhance your coffee machine by implementing polymorphism and separation of concerns, so it's time to put your application to the test.

🖥️ It's time to get coding!
1. Run the `CoffeeMachine` using the IDE to start the simulation. You are already familiar with the menu and prompts that ask for more information about your favorite coffee beverage.
2. You must try all the options, including the exit.
3. Verify that the correct overridden methods are called for each coffee type.
4. Check the output to ensure the coffee preparation steps (grinding and brewing) are displayed accurately.
5. Test different combinations of coffee options, such as different roasts, milk types, and syrup flavors, to verify the system's flexibility.

6. **Compare the output of your program to the expected results. A few expected results are given below.**

   a) **Selecting Espresso with medium roast and 2 servings:**

```
Welcome to the Coffee Machine!

Select an option to continue:

1. Espresso

2. Latte

3. Exit

Enter your choice (1, 2, or 3): 1

What Roast would you like? (light, medium, dark): medium

How many servings would you like? (a number please): 2



Grinding the espresso beans finely...

Brewing the espresso under high pressure...



You ordered a Espresso with a medium roast.

The caffeine level in your coffee is 100 mg.

You asked for 2 servings!

Every serving of Espresso costs 2.5$. Your total bill is 5.0$.
```

   b) **Selecting Latte with light roast, skim milk, and caramel syrup:**

```
Welcome to the Coffee Machine!

Select an option to continue:

1. Espresso

2. Latte

3. Exit
```

```
Enter your choice (1, 2, or 3): 2

What Roast would you like? (light, medium, dark): light

What milk type would you like? (whole, skim, almond, oat): skim

Would you like syrup? (yes/ no): yes

Which flavor would you like? (vanilla, caramel, hazelnut): caramel



Grinding coffee beans coarsely for a latte (medium grind).

Brewing coffee for a latte...

Adding caramel syrup to the cup...

Steaming skim milk...

Combining coffee and steamed milk...

Adding a layer of foam on top...



You ordered a Latte with a light roast.

The caffeine level in your coffee is 50 mg.

Your latte has skim milk and caramel flavor.

Your total bill is 3.5$.
```

Carefully observe the grinding and brewing messages in the output for Espresso and Latte. They are specific to the ordered coffee type, demonstrating polymorphism's power in action. This flexibility allows the system to handle different coffee preparations without explicitly coding for each type, making the code more maintainable and adaptable to new coffee variations.

If you'd like to challenge yourself further, why not explore customization options for your coffee machine, such as implementing different coffee sizes or additional toppings?