In the last video topic, *Introduction to abstract classes*, you learned some of the fundamentals of abstract classes and saw why they are necessary in Java. Now, keep this in mind as you put what you've learned into practice!

In this lab, you'll help Toni program two robots, *SideKick* and *WatADriver*, to function as required. Toni has a tendency to repeat code across programs. To help him, you'll create a superclass to consolidate common properties and methods shared by *SideKick* and WatADriver. This class will be made abstract to prevent its object from being created, ensuring it only serves to provide shared functionality. *SideKick* and *WatADriver* will inherit the class' properties and implement any abstract methods defined in it. This way, you'll promote code reuse and enforce the implementation of essential functionalities.

Goal

Help Toni use abstract and child classes in a Java program to promote code reuse and enforce functionality. This involves creating a superclass *Robot* to consolidate common properties and methods for two robots, *SideKick* and *WatADriver*. The superclass must be made abstract to prevent objects from being created in the class Robot since the Robot *class* has no functionality other than to contain common code. You must ensure that *SideKick* and *WatADriver* inherit from this abstract class and implement any abstract methods defined within the *Robot* class.

This practice will move common code to the abstract class, add child classes, and ensure proper implementation of essential functionalities.

🖥️ 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.

To help Toni, you'll need to write the necessary code to accomplish the requested tasks.

It sounds tough, but you can do it! Here's the plan:

- Starter classes: You have four classes in the starter code:
    - `SideKick`: Contains predefined properties and methods.
    - `WatADriver`: Initially empty, to be implemented.
    - `Main`: Contains the `main()` method for user interaction.
    - `Robot`: Target for moving common properties and method.
- Superclass creation:
    - Move common properties and methods to a new parent `Robot` class.
    - Make `Robot` an `abstract` class to prevent instantiation.
- Subclass updates:
    - `SideKick`: Update to extend `Robot` and use inherited functionality.

    - `WatADriver`: Create and extend `Robot`, implementing necessary methods.
- Implementing abstract methods:
    - Declare abstract methods in `Robot`.
    - Implement these methods in `SideKick` and `WatADriver`.
- Exploring reusability:
    - Reflect on how inherited methods and properties streamline and enhance code reusability and maintainability.

🖥️ 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.

Tip

Remember, the first thing you want to do when coding is to cut down on code repetition. One effective way to achieve this is by refactoring common code into a parent or superclass.

Properties like **`modeOfOperation`**, **`batteryCharge`**, and even the **`Scanner`** object can be shared. Similarly, the **`accessor`** and **`mutator`** methods for these properties can also be moved to the superclass.

The Main method of the Main class is placed in a file named *Main.java.*

As soon as you open Intellij, you will be presented with four files:

- *SideKick.java*
- *WatADriver.java*
- *Main.java*
- *Robot.java*

Let’s explore one of the files. When you open *Robot.java* you will notice your TODOs. Remember that the term TODO indicates the position within the program where you have to type in your own code.

```
/** TODO 15: Set the value of the property "modeOfOperation" of the object

 * "sideKickObject to a value of 2

 * using the setter method you created in TODO 7 and TODO 8.

 * After that is done, call the method takeAction() using object

 * "sideKickObject" and the dot operator.

 **/

}

}
```

Each Java file will have its own TODOs. Please follow the instructions and order of the TODOs to ensure that you’re coding in the right place at the right time!

You might wonder, "With so many files for this lab, how do I know which one to open and code in?" For each task, you will receive specific instructions indicating whether to work in the *SideKick.java*, *Main.java*, *WatADriver.java*, or *Robot.java* file.

Now, let's get into the thick of it and help Toni with his robots!

In this task, you will move the properties and functionality common to both SideKick and WatADriver to a new class called the **`Robot`** class. You'll also make sure that no objects of the **`Robot`** class can be created or instantiated.

Before you start coding, open the file *Main.java* and run the **`Main`** method. When prompted, enter the value 2 (which represents cooking) as your choice.

You've run the **`main`** method and seen the initial output. Why is it important to move the properties **`modeOfOperation`** and **`batteryCharge`** to a superclass? This change will promote code reuse and improve maintainability.

Now, we might need another robot later. So, why not move those properties and methods common to all the robots to a class **`Robot`**? If the child classes inherit from the **`Robot`** class, the code gets re-used.

🖥️ It's time to get coding!

- 🖥️ TODO 1: Open the *SideKick.java* file and select and cut the property declaration for the property **`modeOfOperation`**. Now, open the *Robot.java* file and paste the declaration where indicated.

```
/** TODO 1: Cut the of "modeOfOperation" property declaration * from

 * SideKick.java and paste it here.

 * From now on we will use

 * the term "move" for cut and paste from one class to

 * another. Cut out the comments for the property too

 * and put it here.

 **/
```

- 🖥️ TODO 2: In *SideKick.java*, cut the property declaration for the property **`batteryCharge`**. Then, go to the file *Robot.java* and paste it in the correct spot.

```
/** TODO 2: Move the declaration of "batteryCharge" to this class

 * from SideKick.java to here

 **/
```

- 🖥️ TODO 3: In the *Robot.java* file create a default constructor. Next, go to the *SideKick.java* file and cut the initialization of the properties **`modeOfOperation`** and **`batteryCharge`** from the default constructor. Then, paste this initialization inside the constructor of the *Robot.java* file.

The code was moved from the SideKick class to the Robot class to promote code reusability and reduce redundancy. Both SideKick and WatADriver are robots and share common properties and methods. By moving the shared code to the parent class Robot, both child classes can inherit and reuse this code.

```
/** TODO 3: Create a default constructor and move the initialization

 * of the property "modeOfOperation" and "batteryCharge"

 * from the constructor of SideKick.java to here.

 **/
```

- 🖥️ TODO 4: Go to the *SideKick.java* file and cut out the setters and getters of the property **`modeOfOperation`**. If you remember, each property, by convention, should have two methods: one **`setter`** and one **`getter`**. So, the methods you are cutting are:
  - **`setModeOfOperation`** – **`setter`** for property **`modeOfOperation`**.
  - **`getModeOfOperation`** – **`getter`** for property **`modeOfOperation`**.

```
/** TODO 4: Move the setters and getters for property

 * "modeOfOperation"

 * from SideKick.java to here.

 **/
```

Now, go to *Robot.java* and paste these two methods in the spot below the TODO 5.

- 🖥️ TODO 5: Go to the *SideKick.java* file and cut out the setters and getters of the property **`batteryCharge`**. The specific methods you are cutting are:
    - **`setBatteryCharge`** – **`setter`** for property **`batteryCharge`**.
    - **`getBatteryCharge`** – **`getter`** for property **`batteryCharge`**.

Now, go to the file *Robot.java* and paste these two methods there.

```
/** TODO 5: Move the setters and getters for property
 * "batterCharge"
 * from SideKick.java to here.
 **/
```

You've moved the properties and their accessor methods to the **`Robot`** class. How does consolidating these methods in a superclass benefit the code? It centralizes shared functionality, making the codebase easier to manage and extend. Plus, the child classes can re-use the code and properties.

- 🖥️ TODO 6: Go to the file *SideKick.java* and cut the method **`rechargeBattery`**. Then, paste it in the *Robot.java* file.

```
/** TODO 6: Move the method "rechargeBattery()" for
* recharging battery
* from SideKick.java to here.
**/
```

- 🖥️ TODO 7: Go to the file *SideKick.java* and cut the method **`displayBatteryLevel`**. Then, paste it in the *Robot.java* file.

```
/** TODO 7: Move the method "displayBatteryLevel()"

 * for displaying battery

 * charge level from SideKick.java to here.

 **/
```

- 🖥️ TODO 8: Go to the file *Robot.java* and ensure that no objects of the class **`Robot`** can be created by adding the keyword **`abstract`** before the keyword **`class`**.

The Robot class is made abstract because it represents a general concept of a robot and doesn't have its own specific functionality. An abstract class cannot be instantiated, ensuring that only concrete subclasses like SideKick and WatADriver can be created. This enforces that each robot type has its own implementation of essential methods.

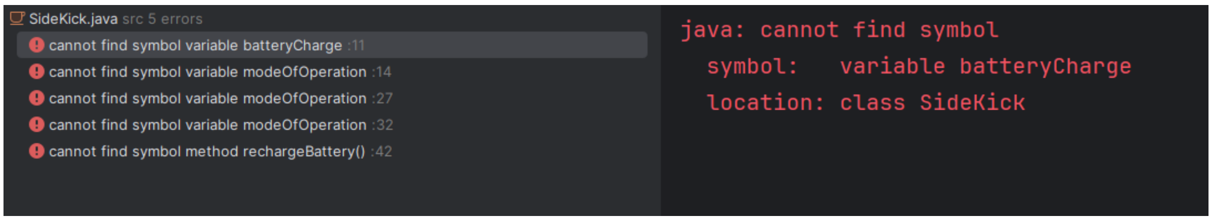
```
/** TODO 8: Prevent objects of this class from being

* created by making

* this class abstract.

 **/
```

Phew, a lot of cutting and pasting! Nonetheless, you have successfully moved the code common to both the **`SideKick`** and **`WatADriver`** classes to the **`Robot`** class.

You may be wondering why the **`Robot`** class was created? The main reason is to hold common code and promote code reuse. The **`Robot`** class itself doesn't have a practical use beyond sharing its code with its child classes. Since you don't want to create objects of this class accidentally, you'll add the keyword **`abstract`** before **`class`** to prevent this. You haven't done this yet, so let's do it now.

Well done, you've completed Task 1!

Now, let's say you were to open the *Main.java* file and run the **`Main`** method. Doing so would lead to many errors.



```
SideKick.java src 5 errors
  cannot find symbol variable batteryCharge :11
  cannot find symbol variable modeOfOperation :14
  cannot find symbol variable modeOfOperation :27
  cannot find symbol variable modeOfOperation :32
  cannot find symbol method rechargeBattery() :42
```

```
java: cannot find symbol
  symbol:   variable batteryCharge
  location: class SideKick
```

In the *SideKick.java* file you'll see that all the references to the properties **`modeOfOperation`** and **`batteryCharge`** are marked in red, meaning Java cannot understand the variable or property.

Why is this? Both the properties **`modeOfOperation`** and **`batteryCharge`** have been moved to the class **`Robot`**, and as such, they are not understood by the **`SideKick`** class.

In this task, you'll begin to fix this by making **`SideKick`** a child class of the **`Robot`** class. You'll adjust the code in **`SideKick`** to ensure it correctly utilizes the code moved to the **`Robot`** class and functions as it did before.

🖥️ It's time to get coding!

- 🖥️ TODO 9: Go to the file *SideKick.java* and make the class **`SideKick`** inherits from the class **`Robot`**, thus making the class **`SideKick`** a child class of the **`Robot`** class. You can do this by putting **`extends Robot`** after **`public class SideKick`**, before the curly braces of the class start.

```
/** TODO 9: Make this class a child class of

* the Robot class by using

 * "extends Robot" after the

 * "class SideKick" above.

 **/
```

Does this fix the issue? No, it does not. Why? Because the properties in the **`Robot`** class are private. If they were protected, they would be inaccessible from the child classes like the **`SideKick`** class.

Tip

Remember that setters and getters are made public or protected. We use them to access the private properties of the parent class.

So, how do you access the private properties of a parent class in the child class? Keep going to find out!

- 🖥️ TODO 10: In *SideKick.java* and in the **`setChoice`** method, replace the assignment of the value **`choice`** to the property **`modeOfOperation`** with a call to the **`setter setModeOfOperation()`**, passing the variable **`choice`** as a parameter to the method.

```
/** TODO 10: Replace the direct assignment of the

* property "modeOfOperation"

 * to use the setter method

*/
```

- 🖥️ TODO 11: In *SideKick.java* and in the **`takeAction`** method, replace the use of the value of the property **`modeOfOperation`** in the **`switch`** statement with a call to the **`getter`** method named **`getModeOfOperation()`** inside the **`switch`**.

```
/** TODO 11: Replace the direct use of

 * the property "modeOfOperation"

 * in the switch to use the

 * getter method

 */
```

Go to *Main.java* and run the **`Main`** method. Does it run? Of course, it does!

Super! You just corrected the code in the child class after making **`SideKick`** a child class of the **`Robot`** class, and resolved the inconsistencies that occurred when you moved code to the **`abstract`** parent class. How does it feel? Nice, right? Keep going, and you'll feel even better!

Great, SideKick is running nice and smoothly, but what about WatADriver? When you open the class **WatADriver**, you'll see that the class is empty, apart from the TODO instructions.

```
public class WhatADriver{



    // other code of class WhatADriver



}
```

You know that WatADriver is also a robot and shares the properties and functionalities you moved to the **Robot** class. To ensure **WatADriver** inherits all these properties and functionalities, you must make the **WatADriver** class a child class of the **Robot** class, just as you did with **SideKick**.

🖥️ It's time to get coding!

- 🖥️ TODO 12: Go to *WatADriver.java* and make the class **WatADriver** a child class of the **Robot** class.

```
/** TODO 12: Make this class "WatADriver"

 *  a child class of

 * the "Robot" class

 */
```

As soon as you do this, **WatADriver** inherits all the properties and functionalities of the **Robot** class (like a child inherits some of the traits of his or her parents, hence the name 'child class'). Let's verify this next.

- 🖥️ TODO 13: Now, you know that the code of SideKick is running fine. So, comment out the code of **SideKick** and focus on the correct functioning of **WatADriver**. To do this, go to the *Main.java* file and comment out the code on **SideKick**.

```
 /** TODO 13: Comment out the code on SideKick below  **/
```

- 🖥️ TODO 14: In *Main.java*, create an object of the class **`WatADriver`** naming the object **`driverBot`**.

```
/** TODO 14: Create an object of the

 * "WatADriver" class

 *  named "driverBot"

 **/
```

- 🖥️ TODO 15: Stay in *Main.java* and use the **`driverBot`** object you just created to call the methods **`rechargeBattery`** and **`displayBatteryLevel`**.

```
/** TODO 15: Call the method rechargeBattery()

 * and then

 * displayBatteryLevel()

 * with the object "driverBot"

 **/
```

When you run the **`Main`** method in *Main.java*, what do you notice? You should observe the battery charging and the battery charge. However, the **`WatADriver`** class doesn't have any code inside its curly braces. This is because it inherits everything from the **`Robot`** class.

Remember, a child class gets all the properties and methods of the parent class and can call the parent's public methods through its own objects. If a class does not have a method inside it, Java checks if it is available in its parent. If it is found in the parent, the method of the parent is called, which is what happens here.

Wow! Isn't that a neat trick? You saved yourself a lot of code repetition and time by simply extending the Robot class.

So far, you've ensured the robots can run code from the parent class. Now, an **`abstract`** class might declare functionality or methods that it never implements. The child class needs to implement the methods by giving body or curly braces to the methods that are abstract (or only declared). If the child class does not do this, it itself becomes abstract, and no methods of the child class can be created.

The **`WatADriver`** class inherits properties like **`modeOfOperation`** and **`batteryRecharge`** from the **`Robot`** class but needs additional features like options to drive a car, plane, or jet fighter, and the ability to perform those actions. Similarly, the **`SideKick`** class has methods to set user choice and execute tasks but needs similar enhancements.

To make sure all child classes provide the necessary functionality, you'll make it so any class that inherits from the **`Robot`** class must implement two methods: **`setChoice`** and **`takeAction`**. If a child class doesn't implement these methods, you won't be able to create an object of that class. This is easy to enforce – simply declare these methods as **`abstract`** in the **`Robot`** class. This way, every child class has the essential features it needs to function properly.

🖥️ It's time to get coding!

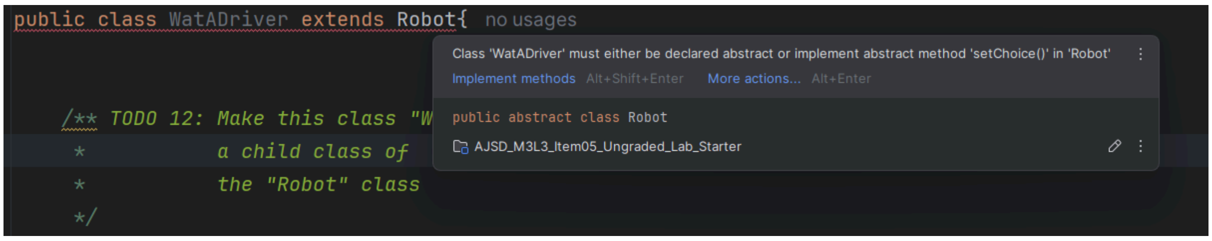
- 🖥️ TODO 16: Go to the file *Robot.java* and declare the two **`abstract`** methods **`setChoice`** and **`takeAction`**.

```
/** TODO 16: Declare two abstract methods

 * "setChoice" and "takeAction"

 *  with the return type "void"

 **/
```

Go to the *Main.java* and run the **`Main`** method. Hold on, something's not right - running the **`Main`** method resulted in an error.

```
java: WatADriver is not abstract and does not override abstrct method

 takeAction() in Robot
```

This is because all **`abstract`** methods have to be implemented by the child class. You can confirm this by opening *WatADriver.java*, where there will be a red line beneath the class name. Hover the cursor over the red line and you will see the reason why it's there.



```
public class WatADriver extends Robot{  no usages


    /** TODO 12: Make this class "W
     *          a child class of
     *          the "Robot" class
     */
```

Class 'WatADriver' must either be declared abstract or implement abstract method 'setChoice()' in 'Robot'

Implement methods Alt+Shift+Enter   More actions... Alt+Enter

```
public abstract class Robot
```

AJSD_M3L3_Item05_Ungraded_Lab_Starter

Abstract methods of an **`abstract`** class must be implemented in the child class, not the parent. This is the aim of your final task.

Almost there! Keep up the great work, you are becoming a better programmer with every task you complete.

Your last move resulted in an error. However, you know what the problem is, so let's fix it! In this task, you will fix the error of not implementing the **`abstract`** methods from the **`abstract`** class **`Robot`** in the child class **`WatADriver`**.

🖥️ It's time to get coding!

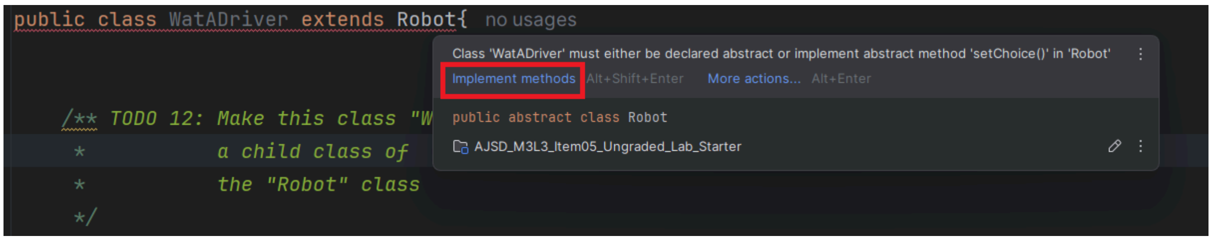
- 🖥️ TODO 17: Open *WatADriver.java* and implement the **`abstract`** methods of the parent class **`Robot`** in the class **`WatADriver`**. The *WatADriver* robot is designed to be capable of both driving and flying. Therefore, the **`setChoice()`** method should be capable of providing a menu with options for driving or flying. Similarly, the **`takeAction()`** method should simulate driving or flying with an appropriate message. For now, you will just display "Inside choice setting of WatADriver" in the **`setChoice()`** method, and "WatADriver will drive or fly here" in the takeAction() method.

```
/** TODO 17: Implement the abstract methods

     *           methods setChoice and takeAction

     *           of the parent class "Robot"

     */
```

Remember the 1st step in implementing a method is to add curly braces to the method. It does not matter whether you write code inside the method definition (or curly) braces. As long as the curly braces are given, the method is defined.

Now, lucky for you, IntelliJ can help you automatically implement these methods (you can, of course, type them out too). Follow the steps to implement the methods with IntelliJ's guidance.

- Step A: Place the cursor below the TODO by clicking on the line below it.
- Step B: Hover the cursor over the red line under the class name. In the popup menu, click Implement methods to automatically implement abstract methods at the position where the cursor was placed before in Step A.



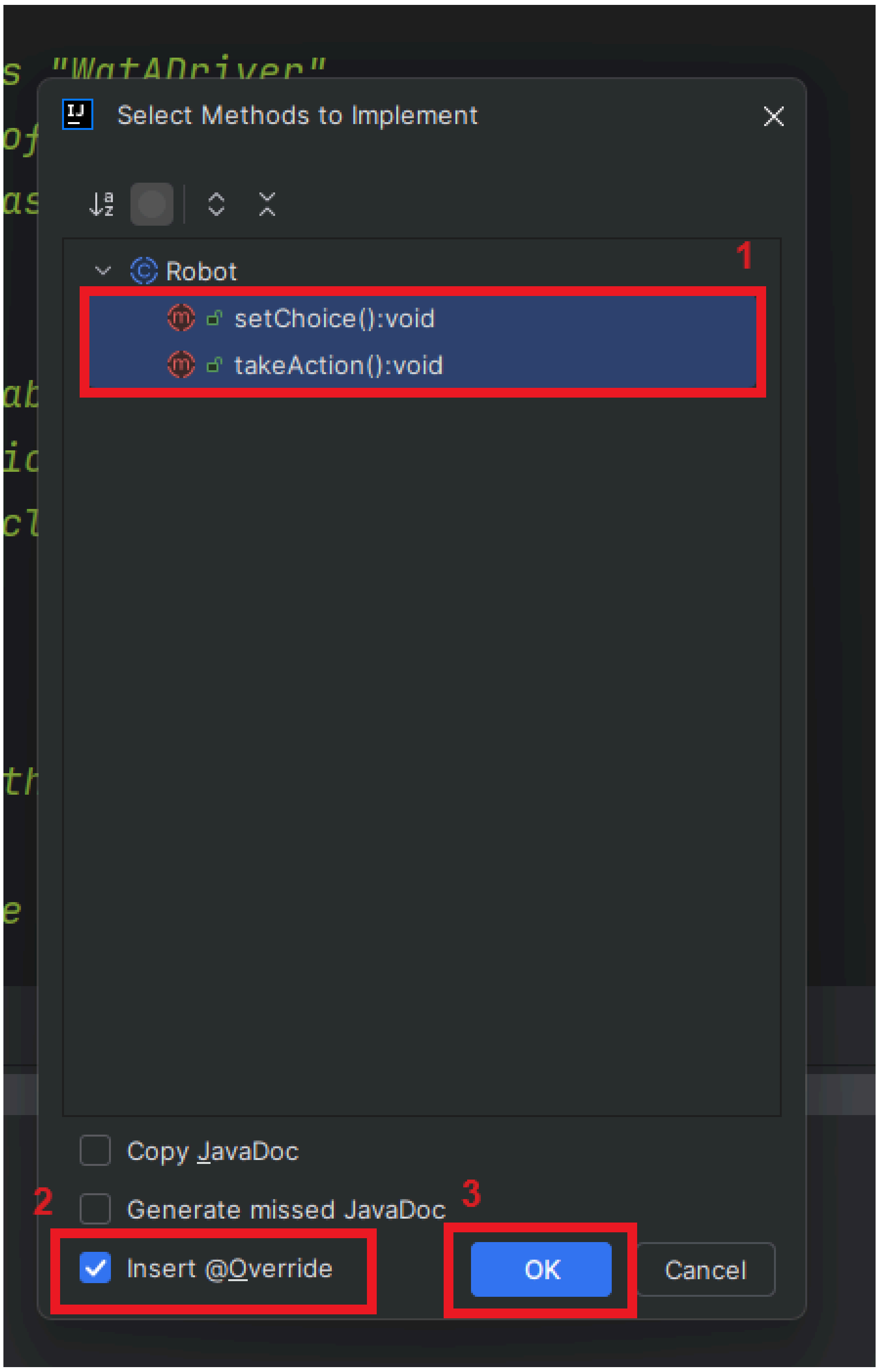
```
public class WatADriver extends Robot{  no usages

    /** TODO 12: Make this class "W
     *           a child class of
     *           the "Robot" class
     */
```

Class 'WatADriver' must either be declared abstract or implement abstract method 'setChoice()' in 'Robot'

```
public abstract class Robot
```

AJSD_M3L3_Item05_Ungraded_Lab_Starter

- Step C: A pop-up dialog box will appear. In it, select the methods of the parent class that must be implemented. Note that all methods are selected by default. Then, make sure the box next to Insert @Override is checked and click OK.



- Step D: The methods are generated in the IDE where the cursor was placed in Step A. Note that the implemented methods are empty.

```
@Override no usages

public void setChoice() {



}



@Override no usages

public void takeAction() {



}
```

- Step E: Inside the empty methods generated in Step D, display "Inside choice setting of WatADriver" in the **`setChoice()`** method, and "WatADriver will drive or fly here" in the **`takeAction()`** method.

Tip

Remember, to implement a method in Java means to place curly braces after the method declaration. Java doesn't care if there's any code inside those braces or not.

- 🖥️ TODO 18: Open the file *WatAdriver.java*, and inside the method **`setChoice`**, display “Inside choice setting of WatADriver”. Inside the method **`takeAction()`** display “WatADriver will drive or fly here”.

```
 *            inside "takeAction" display “WatADriver will

 *            drive or fly here”

 */

 *             display

 *            "Inside choice setting of WatADriver”
```

```
    *           and

/** TODO 18:  Inside the method "setChoice"
```

- 🖥️ TODO 19: Return to *Main.java* and call the methods **`setChoice`** and **`takeAction`** using the **`driverBot`** object that you created in TODO 16.

```
/** TODO 19: Call the methods "setChoice"
         *           and "takeAction"
         *           with the object "driverBot"
         *           which you created in
         *           "TODO 16".
         **/
```

After doing that, run the **`Main`** method to ensure everything works correctly. Reflect on the entire process: How have you used abstract classes and inheritance to improve code organization and enforce necessary functionalities? What are the key takeaways from this lab? If all goes well, you should get the following output.

```
Inside choice setting of WatADriver
WatADriver will drive or fly here
```

Wasn't **`WatADriver`** supposed to input the user choice in **`setChoice`**? Yes, it was. However, Java only requires that the abstract methods of the parent class are implemented in the child class, meaning that curly braces are provided in the child class. It doesn't care what code is inside those braces, just that the method signatures are there.

But don't stress, you will implement full functionality in your next ungraded lab!
In this lab, you've successfully created an abstract class, `Robot`, by consolidating common code for the `SideKick` and `WatADriver` classes. By doing this, you ensure that they reuse common properties and methods, while also being required to run the abstract methods defined in `Robot`.

But that’s not all! You’ve also effectively shown Toni a way to reuse code and enforce essential functionalities within the child classes. This streamlined the code and reduced redundancy and potential errors. Throughout this process, you've helped Toni avoid code repetition, save time, and maintain cleaner, more efficient programs.

Great job on making these improvements! You've helped Toni enhance his robots' functionality while keeping his codebase organized. Keep up the good work, and you'll continue to see how these concepts can simplify and strengthen your coding practices.