


In the last video, *Interfaces and multiple inheritance*, you learned about interfaces and how they can enforce the implementation of specific functionalities and support multiple inheritance. Keep this in mind now, as you apply what you have learned!

In this lab, you'll help Toni create a functionality that SideKick or WatADriver can optionally implement, depending on which version of the robot Toni wants to use for that feature. Specifically, Toni wants a protection module named **kungFuProtection**, where a robot can protect Toni using, you guessed it, Kung Fu. However, Toni has yet to decide which robot should implement this functionality.

Goal

Help Toni create a Kung Fu functionality that any robot can implement if needed. Do this by creating an interface to declare the **kungFuProtection** functionality. Then enable its implementation only in the robot Toni chooses, by having that class implement the interface.

 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 first create a menu program to allow Toni to choose between the two robots and then you will test the menu by running the main method and choosing actions for each robot. Next, you will create an interface named `KungFuSkills` to declare the `kungFuProtection` functionality and define the method inside the interface.

Thereafter, implement the `KungFuSkills` interface in one of the robot's classes and define the `kungFuProtection` method in that class. Finally, you will check if the chosen Robot's class implements the `KungFuSkills` interface using the `instanceof` operator. If implemented, you will call the `kungFuProtection` method; otherwise, you will display a message indicating the absence of Kung Fu skills.

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

- *SideKick.java*
- *WatADriver.java*


- *Main.java*
- *Robot.java*


Tip

In this lab, you'll build on the code you developed in the last lab you completed. However, this time the annotation `@Override` has been put before the methods `setChoice` and `takeAction` in the class `SideKick`.

In addition, the `setChoice` and `takeAction` methods in the class `WatADriver` have been fully implemented to represent what they do, giving options to drive or fly. Also, a menu system to choose the type of robot is provided.

Toni prefers to use only one robot at a time, so you must create a menu program on Toni's mobile that allows him to choose between `SideKick` and `WatADriver`. This menu program will call the appropriate robot based on Toni's choice. Consider this as the `main` method in *Main.java*.

 It's time to get coding!

-  **TODO 1:** First, open the file *Main.java* and update the code inside the `case 1` section of the `switch` statement to use the object `sideKickObject`, so the user can choose their action using the method `setChoice()`. Then, after calling `setChoice()`, call the method `takeAction()` to execute the user's chosen task.

Hint: Use the object name, followed by the dot operator, then the method name, and end with a semicolon.


```
/** TODO 1: Use the "sideKickObject", object of Sidekick class

* to display the menu and then take the appropriate

* action as required.

**/
```

Great job updating the `switch` statement for `SideKick`! Now, what about the `WatADriver` robot? How could you ensure it allows the user to choose and perform actions similarly? Think about how you could replicate the steps for `WatADriver`.

-  **TODO 2:** In the file *Main.java*, go to the `case 2` section of the `switch` statement. Use the `driverBot` object to let the user choose an action by

calling the method `setChoice()`. After calling `setChoice()`, call the method `takeAction()` to execute the chosen task.

Hint: Use the object name, followed by the dot operator, then the method name, and end with a semicolon.

```
/** TODO 2: Use the "driverBot" object of WatADriver class
```


```
 * to display the menu and then take the appropriate
```

```
 * action as required.
```

```
 **/
```

Nicely done, Task 1 is out of the way! Let's get stuck into task 2!

Before you create the interface, you need to test that your menu is working as expected by following these steps.

 It's time to get coding!

Step 1: Go to the file *Main.java*, and run the `main` method. You should see a menu allowing you to choose which robot to work with! When prompted, enter 1 for SideKick and press the Enter key on your keyboard. (Remember that from now on, whenever you need to enter a value, press the Enter key after typing it in).

```
***** MENU FOR ROBOT CHOICE *****
1. SideKick robot
2. WatADriver robot
*****
Enter your choice of robot(1 or 2):
1
```

Step 2: Now, you can choose what action SideKick should take! When you see the "SideKick Menu", select 'Cleaning' by entering the value 1.

```

***** MENU FOR ROBOT CHOICE *****
1. SideKick robot
2. WatADriver robot
*****
Enter your choice of robot(1 or 2):
1
***** SideKick Menu *****
1. Cleaning
2. Cooking
3. Re-charge
4. Check battery level
Enter choice(1-4):
1
Get the vacuum cleaner.....
Put the dust bag in vacuum.....
Go to Living room and clean.....
Go to bedroom and clean.....
Go to kitchen and clean.....
Go to bathroom and clean.....
Retrieve dust bag from vacuum cleaner and put in bin.....
Go back to Toni.....
SideKick cleaning completed.
Another run? (enter y for yes/ any value for no):

```

You'll observe that the `setChoice()` and `takeAction()` methods in `SideKick` are implementations of the `setChoice()` and `takeAction()` methods from the parent class `Robot`, in the class `SideKick`.

Remember, both the `setChoice()` and `takeAction()` methods are implemented and not overridden. Overriding means creating another method with the same signature in the child class as an implemented method in the parent class. Implementing means giving curly braces to an **abstract** (or declared method) of the parent class. What you write inside the curly braces is not important. This is because if a child class does not implement the abstract methods of a parent class, it becomes abstract.

Step 3: SideKick has finished cleaning - that was quick! When prompted to rerun, enter 'y' or 'Y' for "yes".

```

SideKick cleaning completed.
Another run? (enter y for yes/ any value for no):
y

```

Step 4: You're back at the "Menu for Robot Choice"! Enter 1 to pick SideKick again. This time, you'll ask SideKick to recharge its battery, so enter 3 beneath the "SideKick Menu".

```
Another run? (enter y for yes/ any value for no):
y
***** MENU FOR ROBOT CHOICE *****
1. SideKick robot
2. WatADriver robot
*****
Enter your choice of robot(1 or 2):
1
***** SideKick Menu *****
1. Cleaning
2. Cooking
3. Re-charge
4. Check battery level
Enter choice(1-4):
3
Plug into socket.....
Charging .....
Unplug from socket.....
Fully charged.....
SideKick recharged battery.
Another run? (enter y for yes/ any value for no):
```

SideKick performs all the actions necessary to recharge their battery, and you can follow along in your IDE. Pretty neat!

Step 5: Next, let's test WatADriver. Before you begin, you need to stop the running program by clicking on the stop button, a red square, at the top of the IntelliJ window. If you do not see the red square, it means your program is not running. Then, in the *Main.java* file, run the `main` method again. You should see the "Menu for Robot Choice" appear again. This time, choose the WatADriver robot by entering 2.

In the “WatADriver Menu”, why don’t you try out what happens for each action option? WatADriver is quite a talented robot and is probably keen to show off! Remember to check out which class contains the method being called.

Note

From the last ungraded lab, recall that the `rechargeBattery()` method is not part of the `SideKick` class but belongs to the parent class, `Robot`.


Keep in mind that when a method is not found in a class, Java looks up the inheritance chain, checking the parent classes until it finds the method. It never moves down the inheritance chain from the current class to its child classes to find a missing method.

```
***** MENU FOR ROBOT CHOICE *****
1. SideKick robot
2. WatADriver robot
*****
Enter your choice of robot(1 or 2):
2
***** WatADriver Menu *****
1. Drive a car
2. Fly a plane
3. Re-charge
4. Check battery level
Enter choice(1-4):
```

Congratulations! You’ve demonstrated to Toni how to choose a robot and then get them to complete a specific action! Now, let’s take it a step further!

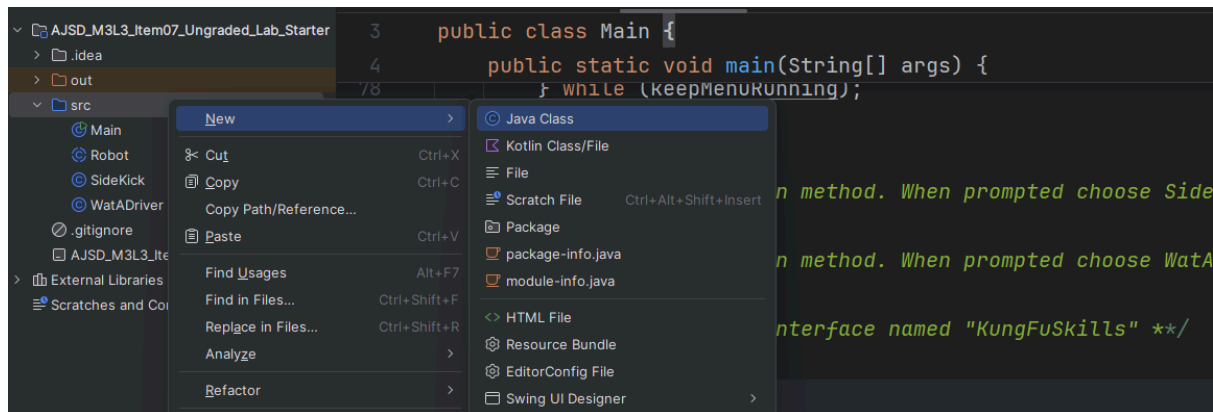
Toni is happy with the program’s progress so far but is eager to get one of his robots to acquire Kung Fu training! In this task, you will create an interface that declares the `kungFuProtection` functionality. You’ll name the interface `KungFuSkills`.

It's time to get coding!

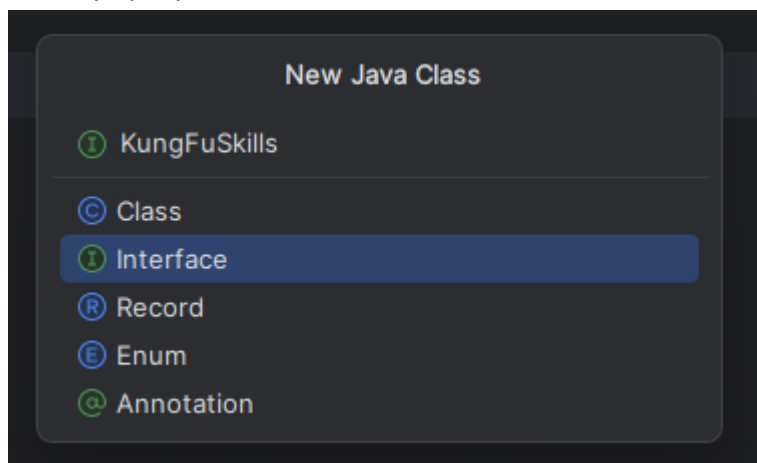
-  **TODO 3:** Here, you will use IntelliJ to create an interface that declares the **kungFuProtection** functionality in an easier way.

```
/** TODO 3: Create an interface named "KungFuSkills" */
```

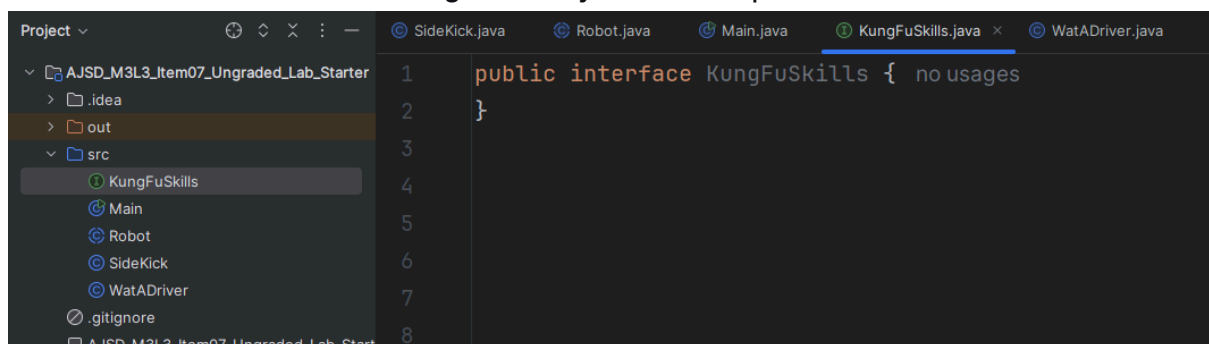
Navigate to the Project window in IntelliJ and right-click on the `src` folder. Choose the option New then Java Class.




In the pop-up menu, choose Interface then enter 'KungFuSkills' and press Enter.



This will create a new file - *KungFuSkills.java* - and open the file in the editor.



-  **TODO 4:** In the file *KungFuSkills.java*, you need to create the functionality for **kungFuProtection**. And how do you add functionality in an interface?

You declare a method inside it. So, declare a method named **kungFuProtection()**.

```
/** TODO 4: In the interface "KungFuSkills" created in TODO 3,  
  
 * get inside the curly braces of the interface in order  
  
 * to declare a method named "kungFuProtection()".  
  
 * Ensure that the return type of the method is "void".  
  
 **/
```


Excellent, just like that, you created an interface named **KungFuSkills** with a single functionality or method named **kungFuProtection**. One of Toni's robots is that much closer to becoming a Kung Fu master, but which robot will Toni choose? Move to the next task to find out.

Now, you'll implement the **KungFuSkills** interface in only one class, or, in other words, only one robot. And that robot is ... SideKick! A Kung Fu move is in SideKick's name, after all!

Implementing an interface in only one class demonstrates that implementing an interface is optional and up to the programmer. You can choose to implement it in another class, or not implement it at all, based on Toni's needs.



It's time to get coding!

-  TODO 5: In the file *SideKick.java*, you will implement the interface, created in TODO 3, in the class **SideKick** by entering implements **KungFuSkills** after **extends Robot**.

```
/** TODO 5: Implement the interface "KungFuSkills"  
  
 * created in TODO 3.  
  
 **/
```

Let's check what happened after you typed in **implements KungFuSkills** in the class **SideKick**.

```
public class SideKick extends Robot implements KungFuSkills {}
```

Hmm, there's a red line under the class name. This happens because when an interface is implemented, the class that implemented it must define the methods of the interface. This, in turn, means providing the method bodies with curly braces. Do

you remember what happens if the class does not define these methods? That's right, the implementing class is penalized and becomes abstract.

Click on the line before the closing curly brace of the `SideKick` class. Implement the method as you learned in the previous lab for implementing unimplemented methods of abstract classes (or type it out if you prefer). This will place the method at the position where the cursor is located.

Tip

Revisit Task 6 in the previous lab, *Activity: Creating abstract classes*, to review how to fix this error in the class.

```
@Override no usages

public void kungFuProtection() {

}
```

Let's explore an important concept about interfaces and inheritance in Java. Java does not allow multiple inheritance, meaning a class cannot extend more than one class or have more than one parent class. However, Java does allow a class to implement multiple interfaces.

For example, if the `SideKick` class wants to implement both the `KungFuSkills` interface and another interface named `KarateSkills`, it can do so like this:

```
public class SideKick extends Robot implements KungFuSkills, KarateSkills {

    // code of the interface

}
```

The interface names are separated by a comma. This way, a class can implement functionalities from different interfaces, achieving a form of multiple inheritance in Java.

When a class implements multiple interfaces, it must define all the **abstract** methods from all the interfaces it implements. Otherwise, the implementing class becomes abstract, meaning objects of that class cannot be created.

This may seem like a lot to take in, but keep going, you're nearly there!

-  **TODO 6:** In the file *SideKick.java*, write the code for the implemented method **kungFuProtection**.

```
/** TODO 6: Give the functionality inside the the methods  
  
 * of the interface "KungFuSkills" which is  
  
 * inside the method "kungFuProtection"  
  
 **/
```

Get inside the curly braces of the implemented **kungFuProtection** method, and create the output you want to see when the method is executed. For example:

```
@Override  
  
public void kungFuProtection() {  
  
    System.out.println("Find a Taolu...");  
  
    System.out.println("Execute the movements....");  
  
    System.out.println("Get in a defensive position...");  
  
}
```

Don't worry about Taolu. This is not a new Java method or interface—these are the steps in Kung Fu to attack or defend!

You don't need to type the exact same output inside the **kungFuProtection** method. Feel free to display whatever message you prefer, or use the example. Now, SideKick is ready to execute Kung Fu and can be called when required.

Toni wants to observe different Kung Fu moves every time he uses SideKick. So, he has requested you to integrate the display of Kung Fu moves, if available, with the robot he chooses to entrust the Kung Fu skills to. You'll need to check if the child class can execute the **kungFuProtection** functionality when needed and then either execute the functionality or display "Sorry, no Kung Fu skills available". As you know, Toni has decided to implement Kung Fu knowledge in SideKick.

Now the question arises: how can you know if a class has implemented an interface? You can use the **instanceOf** operator for that. It returns "true" if the object is an instance of the subclass or a class that has implemented the interface.

For example:

```

if(sideKick instanceof KungFuSkills) {
}

//or

if(driverBot instanceof KungFuSkills) {

}


```

If the class has implemented the interface, you can call the methods defined by the interface.

Now, let's finish the task for Toni!



It's time to get coding!

-  **TODO 7:** In the file *Main.java*, inside the **switch** case, check if the **SideKick** class object named **sideKickObject** is an instance of a class that has implemented the **KungFuSkills** interface. If it has been implemented, then call the method **kungFuProtection**.

```

/** TODO 7: Check if the "SideKick" class has implemented the

* "KungFuSkills" interface which you created in

* TODO 5. If it has implemented it, then call the

* method "kungFuProtection".

**/

```

Hint: Here is how you can do it:

```

if(sideKickObject instanceof KungFuSkills) {

sideKickObject.kungFuProtection();

} else {

System.out.println("Sorry no KungFu skills available.");
}

```

Excellent, you've checked if SideKick implements **KungFuSkills** and called the method if it does. How will you test this functionality?

In *Main.java*, run the **main** method and, when the "Menu for Robot Choice" appears, choose SideKick.

You can run it in the same way as you did in Task 2. But now, you should see the functionality of the `kungFuProtection()` method executed whenever you choose any SideKick functionality.

Great job! You've completed the task and now know how to check if an object of a parent class is pointing to an instance of a subclass that has implemented a certain interface. You also know how to optionally execute functionality or methods of such an implemented interface of a class.

Congratulations on completing this ungraded lab! You have successfully learned how to use interfaces, implement them, and check if an object is pointing to an instance of a subclass. Additionally, you've explored how interfaces can mimic some aspects of multiple inheritance, especially for adding functionality.

Now, Toni's robots, specifically SideKick, can perform Kung Fu if they have implemented the `KungFuSkills` interface. Toni feels safer and more comfortable knowing his robots have added more skills to their list of talents. Plus, you've added another valuable Java programming skill to your growing expertise! Great work, keep it up!