ACTIVITY 4
Extending the Celebrity Class

At this point the Celebrity game is functional with a generic Celebrity class. While this is a perfectly valid implementation, it's also somewhat limited. Every celebrity that is created has the same attributes. But what if you wanted to create a more specific type of Celebrity? One that had additional attributes and behaviors apart from what all Celebrity objects have? By using inheritance to extend the Celebrity class, you can do just that.

With a partner, think about a subset or certain classification of celebrities that would have additional attributes or behaviors. If you were to incorporate a different type of celebrity into the Celebrity game, what kind of celebrity would you want to make?

1. Identify the attributes and behaviors that belong to this new type of celebrity that are separate from a "default" celebrity.

Class Name: **SportCelebrity**

- **Attributes**
    - **private int playerNumber;**
    - **private String sportPlayed;**

- **Type**
    - **SportCelebrity extends Celebrity**

- **Behaviors**
    - **setSport**
    - **setNumber**

- **Return type**
    - **getSport**
    - **getNumber**
    - **toString**

It's important to spend time on the design before implementation begins. This step cannot be overstated, as the use of inheritance in a program signifies a relationship between objects and has the goal of code reuse and sharing information.

Compare

Look at the supplied LiteratureCelebrity class. It has an ArrayList<String> for the clues that are associated with it. It also has a constructor with two parameters just like the Celebrity class. There is a method processClues that is called in the constructor and the overridden getClue method that maintains the integrity of the original data (clue). There is another overridden method toString that uses the super.getClue method to access the original data from the Celebrity class clue variable as well. This is one of the benefits of using inheritance in a design, since superclass methods can easily be called and used in the subclass. All subclasses need to have a clear relationship to the superclass and should provide additional, separate functionality from the superclass to differentiate it from the superclass.

The @Override prefix is provided as a cue to other developers to formally express that you as a developer are changing what is done in this subclass. It's not a requirement in Java but is an accepted practice to provide good documentation.

Implement

In this activity, you will create the Celebrity subclass you designed earlier.

2. Create the Java file for your Celebrity subclass.

3. Define the instance variables for the Celebrity subclass.

4. Write the constructor for your class.

5. Override the getClue and/or the getAnswer method(s).

Tip

When overriding methods in a subclass, method signatures must be the same. This includes the number, type, and order of any parameters of the overridden method.

6. Override the toString method.

7. Write any other methods that your design has indicated will be required. Look over your design plan and make sure that you have implemented all required components. Check that your code compiles, making sure to test any methods you write.

Tip

Methods that exist in a superclass can be accessed within the subclass or with subclass objects. When a method is called on a subclass object, the method that is executed is determined during runtime and if the subclass does not contain the called method, the superclass method will automatically be executed. From within the subclass, if you would like to call the superclass

version of an overridden method the keyword super must be used before the method call.

CelebrityGame Update

processGuess(String)

Looking at the method processGuess you can see that it only needs to interact with the String matching the name of the celebrity and that method getAnswer must have the same signature (without parameters) in all subclasses of Celebrity. This method is shown to illustrate that no code needs to be changed in order for it to work with the new subclass(es).

```
/** * Determines if the supplied guess is correct.
* * @param guess - The supplied String * @return Whether it matches regardless of case or extraneous
external * spaces. s */ public boolean processGuess(String guess) {
boolean matches = false; /* * Why use the .trim() method on the supplied String parameter? What
        * would need to be done to support a score? */ if
        (guess.trim().equalsIgnoreCase(gameCelebrity.getAnswer())) {
matches = true; celebGameList.remove(0); if
            (celebGameList.size() > 0) {
    gameCelebrity = celebGameList.get(0); } } return matches; }
```

8. Modify the addCelebrity method. As you see, the parameter type is used to tell the CelebrityGame which kind of Celebrity to make. This String will be sent from the GUI to the game so that the appropriate subclass constructor can be called. The provided code shows how the method was updated to work with the LiteratureCelebrity subclass as an example. Add another conditional branch using else if to handle your new subclass.

Since all instances of any subclass of Celebrity are a Celebrity, each can be added to the ArrayList<Celebrity> celebGameList by polymorphism. No changes are needed to that part of the program.

Tip

In addition to parameters and local variables declared in a method, a method always has access to any instance variables that are declared within the enclosing class. These instance variables are often required to complete the goal of the method. If any of the available variables are objects (reference data), then those objects may have their own methods, variables, and

constants that would be accessible within the given method as well.

```
/** * Adds a Celebrity of specified type to the game list

* * @param name - The name of the celebrity * @param guess * The clue(s) for the celebrity
* @param type * What type of celebrity */ public void addCelebrity(String name, String guess,
String type) {

    /* * How would you add other subclasses to this CelebrityGame?
*/ Celebrity currentCelebrity; if (type.equals("Literature"))
        {
    currentCelebrity = new LiteratureCelebrity(name, guess); } else //Add an else if here {
    currentCelebrity = new Celebrity(name, guess); }
this.celebGameList.add(currentCelebrity); }
```

9. The validateClue and validateCelebrity methods may need to be updated for the subclass to check that the supplied String values match the requirements for the subclass. The provided code shows how the validateClue method was updated to work with the LiteratureCelebrity subclass as an example. Add another conditional branch using else if to handle your new subclass.

```
public boolean validateClue(String clue, String type) {
booolean validClue = false; if (clue.trim().length() >= 10) {

            validClue = true;

            if (type.equalsIgnoreCase("lit terature")) { String[] temp =

            clue.split(",");

    if (temp.length > 1) { validClue = true; } else { validClue = false; } } //You will need to add an else if condition
            here fo or your subclass } return validClue; }
```

GUI Update

You'll need to add to the GUI class to support the new addition to the project. The framework for the project has been set so that **you only need to make changes to the StartPanel class**. Don't start this until you have completed your new Celebrity subclass.

Again, the CelebrityFrame, StartPanel, and CelebrityPanel classes have no knowledge of the Celebrity class hierarchy. They only understand String and boolean values (JTextField and JRadioButton).

The StartPanel class needs to be updated to allow for the selection and creation of multiple types of celebrities. The following instructions assume that all input can be parsed from the clue and name instance variables. If you want to add more instance variables then you'll need to define, initialize, and setVisible additional GUI components based on the selection of the associated JRadioButton. This is to make the GUI as easy to use with the least amount of configuration. **All of the following code should be added to the StartPanel class**.

Celebrity Lab: Extending the Celebrity Class

10. Add private String and JRadioButton variables to the StartPanel class to match the custom subclass in the instance variable section. Throughout the instructions the placeholders yourRadioButtonName and nameOfCelebrity are used, however you should use more meaningful variable names for both of these when completing your implementation.

private JRadioButton yourRadioButtonName;

private String nameOfCelebrity;

11. Initialize the yourRadioButtonName and nameOfCelebrity variables in the constructor after the call to super. As a practice, it's often a good idea to put variables of the same type together. This makes it easier to identify when other components are added.

yourRadioButtonName = new JRadioButton("Your Celebrity Type");

nameOfCelebrity = "Your celebrity type clue format hint";

12. In the setupPanel method add the following lines. Remember to change yourRadioButtonName to match your variable name. This method is used to add the component to the JPanel and to be part of the RadioButton group.

this.add(yourRadioButtonName);

typeGroup.add(yourRadioButtonName);

13. Change the setupLayout method. This method is used to arrange all the components into the panel at the specified locations. The values are used to show the relationship between the different components. To help identify specific locations, look for the bolded words. Change the line

panelLayout.putConstraint(SpringLayout.NORTH, literatureRadio,

    10, SpringLayout.SOUTH, **celebrityRadio**);

to be

panelLayout.putConstraint(SpringLayout.NORTH, literatureRadio,

    10, SpringLayout.SOUTH, **yourRadioButtonName**);

and add the following lines below the line you just changed.

panelLayout.putConstraint(SpringLayout.WEST,

    yourRadioButtonName, 0, SpringLayout.WEST, celebrityRadio);

panelLayout.putConstraint(SpringLayout.NORTH,

    yourRadioButtonName, 10, SpringLayout.SOUTH, celebrityRadio);

14. The setupListeners method is the method that links all the GUI components that allow user input to the ActionListeners to be processed by the program. In the radio button section of the setupListeners method, add this line:

yourRadioButtonName.addActionListener(select ->

    clueLabel.setText(yourCelebrityClue));

15. In the validate method:

    › add an else-if after the literature section based on the yourRadioButtonName. isSelected result › Call the validateClue method in the Celebrity class, passing the associated values

16. In the addToGame method, add an else-if yourRadioButtonName.isSelected to provide the correct string to the type variable that is used in the controller. addCelebrity(answer, clue, type) call.

At this point you should execute the CelebrityRunner class to verify game functionality.

Check Your Understanding

17. How do we identify if a method is an overridden method?

If it is method header is written the exact same in the
subclass, it is overwritten OR if it has @Override on top of the
method

18. How do we send information from the subclass to the superclass?
By calling super and then calling the appropriate accessor method or
setter method

19. What keyword is used in Java to identify inheritance?
Extends

20. What method is executed when an ArrayList is made of the supe rclass but a
subclass instance is stored in it?