Assignment 2 Explanation

Brendan Tea

November 6, 2025

Abstract

This is an explanation of my code, design choices, and how it's implementation is efficient. Additionally, we will also discuss potential future features and classes for new behaviors.

Contents

Figure 1: CS 210 #26175 - Professor Taesik Kim
"gpt is not god"

1 Efficiencies

When we first were programming this assignment, it was important for it to follow a specific structure with super classes and subclasses inheriting methods. This would allow us to reuse code and override them to create our own behaviors and additions. In the end, we are able to create instances and run their methods for our purposes: specifically looking at `Chloe (Adult)`, `Ted (Senior)`, `Ed (SuperSenior)` fares. This is the whole idea of the OOP paradigm, allowing us to extend our code for many different purposes.

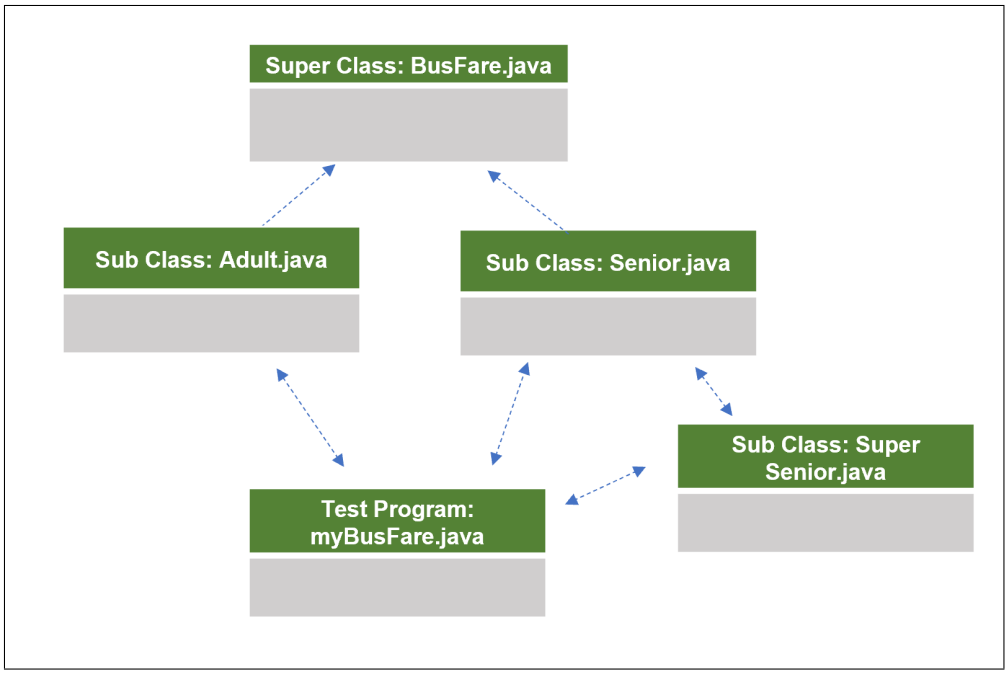


Figure 2: Inheritance Structure

1.1 myBusFare

In the end, this is how how our main myBusFare.java performs. We can simply create our instances and run its output method. This makes our code extremely simply with abstractions. Additionally, in the future, if anyone creates instances, we can update the value of our baseRate and it will cascade and be updated.

```
/*
 * Name: Brendan Tea
 * Date: November 11, 2025
 * Description: Main program runs here.
 *
 */

public class myBusFare {
    // We add our people and run our outputs.
    public static void main(String[] args) {
        // Senior Ted = new Senior();
        // SuperSenior Ed = new SuperSenior();
        // Adult Chloe = new Adult();

        persons p = new persons();
        p.add(new person("Ted", new Senior()));
        p.add(new person("Chloe", new Adult()));
        p.add(new person("Ed", new SuperSenior()));
```

```
19
20          p.output();
21
22      }
23  }
```

1.2 BusFare

Let's take a look at the super class BusFare.java and uncover how it is efficient. In this code snippet, we can encapsulate our class fields `baseRate` and `color` with our getter methods. All of these methods provide a structure for of all subclasses to inherit. The most efficient part of the code is the `fare(String name)` method as allows us to just write one singular method to output our instance data since subclasses have their own overriden methods.

```
1  /*
2   * Name: Brendan Tea
3   * Date: November 11, 2025
4   * Description: Main super class containing our structure and fields
5   *
6   */
7
8
9  public abstract class BusFare {
10     private static double baseRate = 5.0;
11     private static String color = "Red";
12
13     // Abstract method since each subclass handles it differently
14     public abstract double getFare();
15
16     // encapsulated accessor getter methods
17     public double getBaseRate() {
18         return BusFare.baseRate;
19     }
20
21     public String getColor() {
22         return BusFare.color;
23     }
24
25     public void fare(String name) {
26         // %s for printing out the double leaves out the trailing zeros
27         // 2.50 -> 2.5 and 1.2500 -> 1.25 :)
28         System.out.printf("%s.fare() prints \"Fare: %s, Color: %s\" %n",
29                 name,
30                 this.getFare(),
31                 this.getColor());
32     }
33
34 }
```

1.3 Adult

This `Adult` subclass inherits all of the super methods such as `getBaseRate()` and `getColor()`. We can overrides these methods, `getFare()` & `getColor()`, with our own class behavior. `getColor()` again is basically one-to-one because color is static to the class. —

```
/*
 * Name: Brendan Tea
 * Date: November 11, 2025
 * Description: Adult subclass with overriden methods
 *
 */

public class Adult extends BusFare {

    // discount is 1 because there is no discount.
    private static double discount = 1.0;
    private static String color = "Red";

    // uses our super method and applies our class discount value.
    @Override
    public double getFare() {
        return super.getBaseRate() * Adult.discount;
    }

    @Override
    public String getColor() {
        return Adult.color;
    }
}
```

1.4 Senior

This `Senior` subclass is basically the same as Adult and inherits all of the super methods such as `getBaseRate()` and `getColor()`. We can overrides these methods, `getFare()` & `getColor()`, with our own class behavior. `getColor()` again is basically one-to-one because color is static to the class.

```
/*
 * Name: Brendan Tea
 * Date: November 11, 2025
 * Description: Senior subclass extends BusFare and uses its methods
 *
 */

public class Senior extends BusFare {
    // discount is 0.5 as in 50% of the total value equating to 50% off.
    // if we want 30% off, we would write 0.7
    private static double discount = 0.5;
    private static String color = "Grey";

    @Override
    public double getFare() {
        return super.getBaseRate() * Senior.discount;
    }

    @Override
```

```
20      public String getColor() {
21          return Senior.color;
22      }
23  }
```

1.5 SuperSenior

This `SuperSenior` subclass extends `Senior` and inherits all of the super methods such as `getBaseRate()` and `getColor()`. We can overrides these methods, `getFare()` & `getColor()`, with our own class behavior. Something special now is the way we can calculate our `getFare`. Instead of grabbing `BusFare` base rate and applying a discount, we can instead just take 50% of the `Senior` fare cost. `getColor()` again is basically one-to-one because color is static to the class.

```
/*
 * Name: Brendan Tea
 * Date: November 11, 2025
 * Description: SuperSenior subclass extends Senior and uses its methods
 *
 */


public class SuperSenior extends Senior {
    // discount is 0.5 as in 50% of the total value equating to 50% off.
    // if we want 30% off, we would write 0.7
    private static double discount = 0.5;
    private static String color = "White";

    // Slightly confusing. We are taking the senior's rate and halving it again
    // Essentially our super.BaseRate() * 0.25;
    @Override
    public double getFare() {

        // return super.getBaseRate() * this.discount (change to 0.25 ?);
        return super.getFare() * SuperSenior.discount;
    }

    @Override
    public String getColor() {
        return SuperSenior.color;
    }
}
```

2 Future potentials

2.1 myBusFare improvements

Usually without creating specific person struct for the with names and their specific BusFare, you'll get something like this. It's dry and honestly redundant. It would be better if we could have an array. This is the bare minimum solution.

```
1
2  public class v1myBusFare {
3
4      public static void main(String[] args) {
5          Senior Ted = new Senior();
6          Ted.fare("Ted");
7
8          SuperSenior Ed = new SuperSenior();
9          Ed.fare("Ed");
10         Adult Chloe = new Adult();
11         Chloe.fare("Chloe");
12
13     }
14 }
```

This is what we get but with two arrays with same elements correlating to the same object, you can see some improvement. Later on you can improve upon this but creating a struct.

```
1  public class v2myBusFare {
2
3      public static void main(String[] args) {
4
5          BusFare[] people = {
6              new Adult(),
7              new Senior(),
8              new SuperSenior()
9          };
10
11         String[] names = {
12             "Chloe",
13             "Ted",
14             "Ed"
15         };
16
17         for (int i = 0; i < people.length; i++) {
18             people[i].fare(names[i]);
19         }
20
21     }
22 }
```

Afterwards, you can should instead make the person a seperate file.

```
1
2  class person {
3
4      String name;
5      BusFare type;
6
7      person(String name, BusFare type) {
8          this.name = name;
9          this.type = type;
10     }
11
```

```
}

public class v3myBusFare {

    public static void main(String[] args) {
        // Senior Ted = new Senior();
        // SuperSenior Ed = new SuperSenior();
        // Adult Chloe = new Adult();

        person[] peoples = {
            new person("Chloe", new Adult()),
            new person("Ted", new Senior()),
            new person("Ed", new SuperSenior())
        };

        for (person p : peoples) {
            p.type.fare(p.name);
        }

    }
}
```

This is where we arrive upon with extending classes.

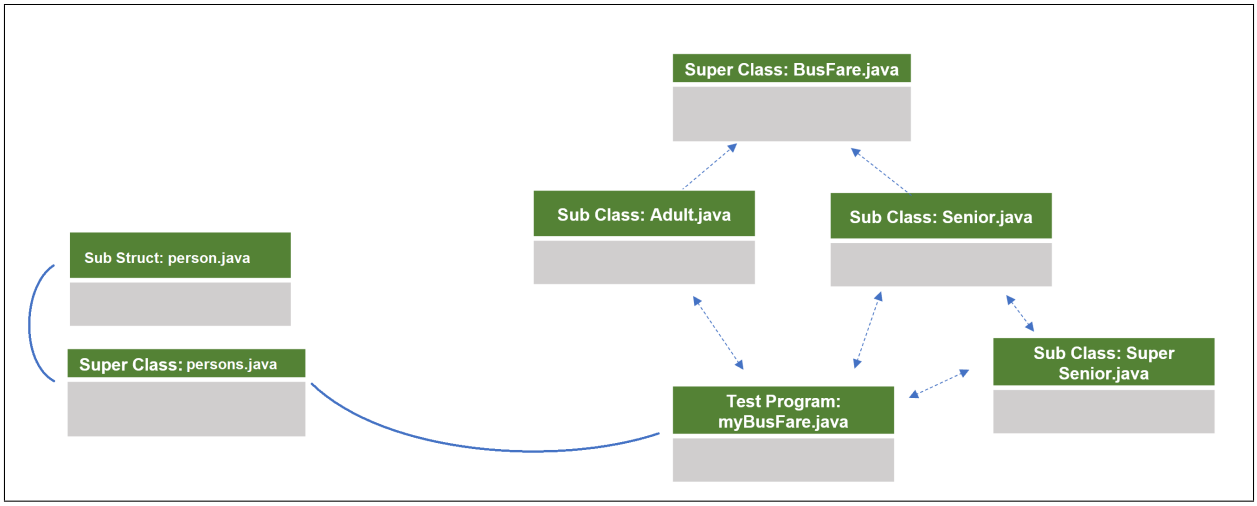


Figure 3: Inheritance Structure with new object types

2.2 persons

This `persons` class utilizes ArrayList so that people in the future can easily add more passengers. This class gives us the ability to clearly see who our people and basically have full control.

```
/*
 * Name: Brendan Tea
 * Date: November 11, 2025
 * Description: Persons class contains all singular person.
 *
 */

import java.util.ArrayList;

// We are using Arraylist and storing a class variable with all our people
public class persons {

    private static ArrayList<person> people;
```

```
14
15      // if no parameters in the constructor, at least set some value to people
16      persons() {
17          this.people = new ArrayList<>();
18      }
19
20      persons(ArrayList<person> people) {
21          persons.people = people;
22      }
23
24      public void add(person p) {
25          persons.people.add(p);
26      }
27
28      // Main output function that allows to see how many people there are and their
        individual info
29      public void output() {
30          System.out.printf("%s total people %n", people.size());
31          for (person p : people) {
32              p.output();
33          }
34      }
35  }
```

2.3 person

This `person` class handles behavior for individual people to accodomoate for their `name` and `BusFare` type. It uses runtime polymorphism so that we can call the output() method in every `BusFare` object which makes it very efficient. It's almost like a struct, providing us with framework of fields and methods to display.

```
1   /*
2    * Name: Brendan Tea
3    * Date: November 11, 2025
4    * Description: Person "struct" class. Contains our fields for a person.
5    *
6    */
7
8   // person struct
9   public class person {
10      private String name;
11      private BusFare type;
12
13      // constructor everytime we make someone
14      // fields are encapsulated so we must assign values when creating instances
15      person(String name, BusFare type) {
16          this.name = name;
17          this.type = type;
18      }
19
20      // runtime polymorphism. Very efficient way to write this.
21      // it actually runs the overriden subclass's fare method of the BusFare.
22      public void output() {
23          this.type.fare(this.name);
24      }
25  }
```

3 Conclusion

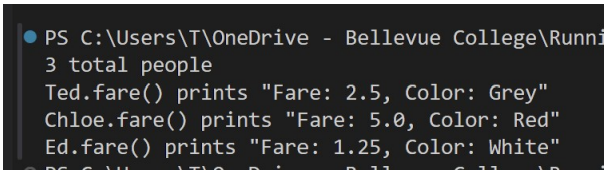
In whole, we've used OOP to help us efficiently accodomoate for all bus users. In the future, we can also add our own subclasses with their own individual behavior. Additionally, others may add users efficiently and view their information.

3.1 Output

[Expected output]

Ted.fare() prints "Fare: 2.5, color=grey"
Chloe.fare() prints "Fare: 5.0, color=red"
Ed.fare() prints "Fare: 1.25, color=white"

(a)



(b)

Figure 4: Side by side comparison