Holly Calloway
COMP 311 – R1WW (W18)
Homework 1
January 24, 2018

Problem 1 [8 points]

Write a simple adapter class to translate between these two interfaces:

```
public interface Student {
  int getId();
  void setId(int id);
  String getFirstName();
  void setFirstName(String firstName);
  String getLastName();
  void setLastName(String lastName);
}
public interface LegacyStudent {
  String getId();
  void setId(String id);
  String getFullName();
  void setFullName(String name);
}
```

```
public class LegacyStudentToStudentAdapter implements Student
{
    private LegacyStudent student;
    private String firstName;
    private String lastName;

    public LegacyStudentToStudentAdapter(LegacyStudent student)
    {
        this.student = student;
    }
    @Override
    public int getId()
    {
        return Integer.parseInt(student.getId());
    }

    @Override
    public void setId(int id)
    {
        String idString = Integer.toString(id);
        student.setId(idString);
    }

    @Override
    public String getFirstName()
    {
        return firstName;
    }
    @Override
    public void setFirstName(String firstName)
    {
        this.firstName = firstName;
    }

    @Override
    public String getLastName()
    {
        return lastName;
    }

    @Override
    public void setLastName(String lastName)
    {
        this.lastName = lastName;
    }

    public String getFullName()
    {
        return student.getFullName();
    }

    public void setFullName()
    {
        student.setFullName(getFirstName() + " " + getLastName());
    }
}
```

Problem 2 [4 points]

There are two simple ways to traverse a list: using an iterator and using indexing and the get() method. For example the following two methods will produce the same output regardless of the dynamic type of the list passed in as a parameter:

```
public static <E> void printListOne(List<E> list) {
    for (ListIterator<E> iterator = list.listIterator();
            iterator.hasNext(); ) {
        System.out.println(iterator.next());
    }
}

public static <E> void printListTwo(List<E> list) {
    for (int i = 0; i < list.size(); ++i) {
        System.out.println(list.get(i));
    }
}
```

Two claims are made:

1. printListOne() is more efficient for LinkedList than printListTwo().
    a. LinkedLists can not be randomly searched, they have to be traversed to search for data.
    b. LinkedLists only know the location of the most recent element.
        i. That element has a reference to the element added before that one.

        ii. So by calling .get(i) on the linked list, it has to intuitively calling next() on each element. Plus you have the loop of .size().
        iii. This has a performance of O(n^2)
    c. By using an iterator and next() – you can directly reference the intended element – “next”
        i. It’s also has a quick boolean method to verify if there are more elements for the iterator to advance too – hasNext();

2. Both algorithms perform the same for ArrayList objects.
    a. ArrayLists can be searched dynamically. When you call “get(i)” on an ArrayList, it immediately goes and finds the element at that index (i).
    b. The performance of searching through an ArrayList is O(1); since you don’t have to traverse the list to find an element.

Provide an explanation for the two claims above.

Problem 3 [8 points]

(Adapted from Programming Project 1, page 253, in Data Structures)

If we insert JDK classes into a `HashSet`, then the `Set` will not accept duplicates:

```
HashSet<String> set = new HashSet<>();
set.add("Hi"); // returns true
set.add("Hi"); // returns false
set.size(); // returns 1
```

If we try the same thing with a class we created, we don’t see the same results:

```
public class Student {
  private String firstName;
  private String lastName;
  // constructor and accessors omitted
}

HashSet<Student> set = new HashSet<>();
set.add(new Student(“John”, “Doe”));
set.add(new Student(“John”, “Doe”));
set.size();

@Override
public int hashCode() {
       return this.hashCode();
}

@Override
public boolean equals(Object obj) {
       if ((obj !=null) && (obj instanceof Student)) {
              this.firstName = ((Student)o).firstName;
              this.lastName = ((Student)o).lastName;
              this.firstName.hashCode() = ((Student)o).firstName.hashCode();
              this.lastName.hashCode() = ((Student)o).lastName.hashCode();
       }
```

```
        if ((firstName != null && lastName != null) &&
        ((firstName.equals(this.firstName) && (lastName.equals(this.lastName))
        {
                return true;
        }

        return false;
}
```

Explain why this isn't working as expected, and fix the `Student` class so that it works correctly with `HashSet`. Hint: Look into the contract for `hashCode()`.

HashCode JavaDoc:

"Whenever it (hashCode()) is invoked on the same object (such as Student) more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer **need not remain consistent from one execution of an application to another** execution of the same application."

Basically when Hash's are used on objects, there is no way to identify that if two objects are the same – they must have different hashCodes; since hashCodes treat each new object as a "new" object. When used with data types/JDK classes, the value is more easily compared with equals() and hashCode() methods.

The equals() methods looks for the memory location, and if the hashCodes are different – then it will return "false" each time. To fix this, you need to use the equals() method. By overriding this, you are explicitly declaring what should return "true" and what should return "false" based on actual data and not memory locations.

https://eclipsesource.com/blogs/2012/09/04/the-3-things-you-should-know-about-hashcode/