I.BA_OOP.H22 - Eldar Omerovic - 2023

Commands to export mvn project

```
mvn clean package
```

Data structures

Using a datastructure that enables us to get data by the identifier.

```
public Map<String, Car> carList = new HashMap<>();
```

Usage

```
// Counts the entries in the HashMap
public Integer getAmountOfCarTypes(){
    return carList.size();
}
// Gets a specific entry in the HashMap by its key (identifier) and returns the value
// for .getAmount() on the object
public Integer getAmountForCar(String identifier){
    return carList.get(identifier).getAmount();
}
```

`.toString()` implementation

```
@Override
public String toString(){
    return String.format("The car with id %id, amount: %amount",
         this.id, this.amount);
}
```

Java Streams

Stream In Java - GeeksforGeeks

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.The

https://www.geeksforgeeks.org/stream-in-java/

Event and Eventhandling

To create an event handler we first need to create the listener interface.

```
import java.util.EventListener;

public interface ReorderCarListener extends EventListener {
    void onNotEnoughCarAmount(ReorderCarEvent e);
}
```

Now create the Event.

```
import java.util.EventObject;

public class ReorderCarEvent extends EventObject {
    private final String identifier;
    /**
     * Constructs a prototypical Event.
     * @param source the object on which the Event initially occurred
     * @param identifier the identifier of the car to reorder
     * @throws IllegalArgumentException if source is null
     */
    public ReorderCarEvent(Object source, String identifier) {
        super(source);
        this.identifier = identifier;
    }

    public String getIdentifier(){
        return this.identifier;
    }
}
```

Now we need to prepare the Garage class for publishing the event and publish the event if not enough cars of a type are in the garage.

```
private List<ReorderCarListener> listeners = new ArrayList<ReorderCarListener>();

public void addListener(ReorderCarListener listenerToAdd){
    listeners.add(listenerToAdd);
}

public void takeCarById(String identifier, int amountToTake){
    Car car = carList.get(identifier);
    car.takeCars(amountToTake);
    if(car.getAmount() < MIN_VALUE_PER_CAR){
        fireEvent(car.getId());
    }
}
```

Methods and attributes for event handling

```
private List<ReorderCarListener> listeners = new ArrayList<ReorderCarListener>();

public void addListener(ReorderCarListener listenerToAdd){
    listeners.add(listenerToAdd);
}

public void removeListener(ReorderCarListener listenerToRemove){
    listeners.remove(listenerToRemove);
}

public void fireEvent(String identifier){
    ReorderCarEvent reorderCar = new ReorderCarEvent(this, identifier);
    for(ReorderCarListener listener : listeners)
        listener.onNotEnoughCarAmount(reorderCar);
}
```

Event Testing

```
@Test
void TestIfEventGetsFiredWhenAmountOfCars_UnderFour(){
    Garage garage = new Garage();
    garage.addListener(e -> Assertions.assertEquals("aaaaa", e.getIdentifier()));
    garage.takeCarById("aaaaa", 7);
}
```

Equals and Hashcode

Implementing the `equals()` method

```
/**
 * Zuerst wird überprüft ob das zu vergleichende Objekt, die selbe ist wie die ursprüngliche Instanz.
 * Dann wird sofort false zurückgegeben, wenn das zu überprüfende Objekt nicht vom gleichen Typ ist
 * Nun wird das zu vergleichende Objekt als Car geparsed.
 * Jetzt wird verglichen ob der String derselbe ist(mit null checks vorher)
 * Nun wird true zurückgegeben, wenn der amount derselbe ist sowie auch der identifier derselbe ist.
 * @param o ist das zu vergleichende Objekt.
 * @returns true, wenn die Cars gleich sind.
 */
@Override
public boolean equals(Object o) {
    if (o == this)
        return true;
    if (!(o instanceof Car))
        return false;
    Car other = (Car)o;
    boolean sameIdentifier = (this.id == null && other.id == null)
            || (this.id != null && this.id.equals(other.id));
    return sameIdentifier;
}
```

Implementing / Overriding the hashcode method

```
@Override
public int hashCode() {
    return (Objects.hash(this.id));
}
```

Testing the `equals()` and `hashcode()` method

```
@Test
void CarEqualsVerifier() {
    EqualsVerifier.simple().forClass(Car.class)
            .suppress(Warning.ALL_FIELDS_SHOULD_BE_USED).verify();
}
```

Testing

```
Car car = new Car("abcde");
Assertions.assertEquals( 0, car.getAmount());
```

⚠ Testing with the `Assertions` library requires it to import it.
Here an example of what to import on the test classes.

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
```

Testing → Exception Handling

What the class looks like

```
public Car(String identifier) {
    if (identifier.length() != 5){
        throw new IllegalArgumentException("Identifier needs to be 5 characters long");
    }
    this.id = identifier;
    this.setAmount(0);
}
```

What the test looks like

```
@Test
void CarExceptionOnInvalidIdentifier_TooLong() {
    final Exception e =
            Assertions.assertThrows(IllegalArgumentException.class, () -> {
                new Car("abcdefg");
            });
    Assertions.assertEquals("Identifier needs to be 5 characters long", e.getMessage());
}
```