Kevin Barone, Ben Burkhalter

CSCI 4448 - OOAD

03 February 2023

# Project 2.1

1. **A. *Default Method*** - the *default* keyword makes it so that it is not mandatory to provide implementation for these methods of the interface when it's implemented by a class

   Example:

   ```
   default void log(String str){
           System.out.println("I1 logging::"+str);
   }
   ```

   **B. *Static Method*** - the *static* keyword makes it so that the method cannot be overridden by the class that implements the interface to avoid poor method implementation in classes

   Example:

   ```
   static boolean isNull(String str) {
           System.out.println("Interface Null Check");
           return str == null ? true : "".equals(str) ? true : false;
   }
   ```

   **C. *Lambda Expressions* -** these allow you to define functional interfaces inline and only implement the one abstract function, they also give way to treating code as data and passing in functions as arguments, and lambdas can be passed around as if it was an object and executed on demand

   Example:

   ```
   FuncInter1 add = (int x, int y) -> x + y;
   ```

https://www.geeksforgeeks.org/lambda-expressions-java-8/

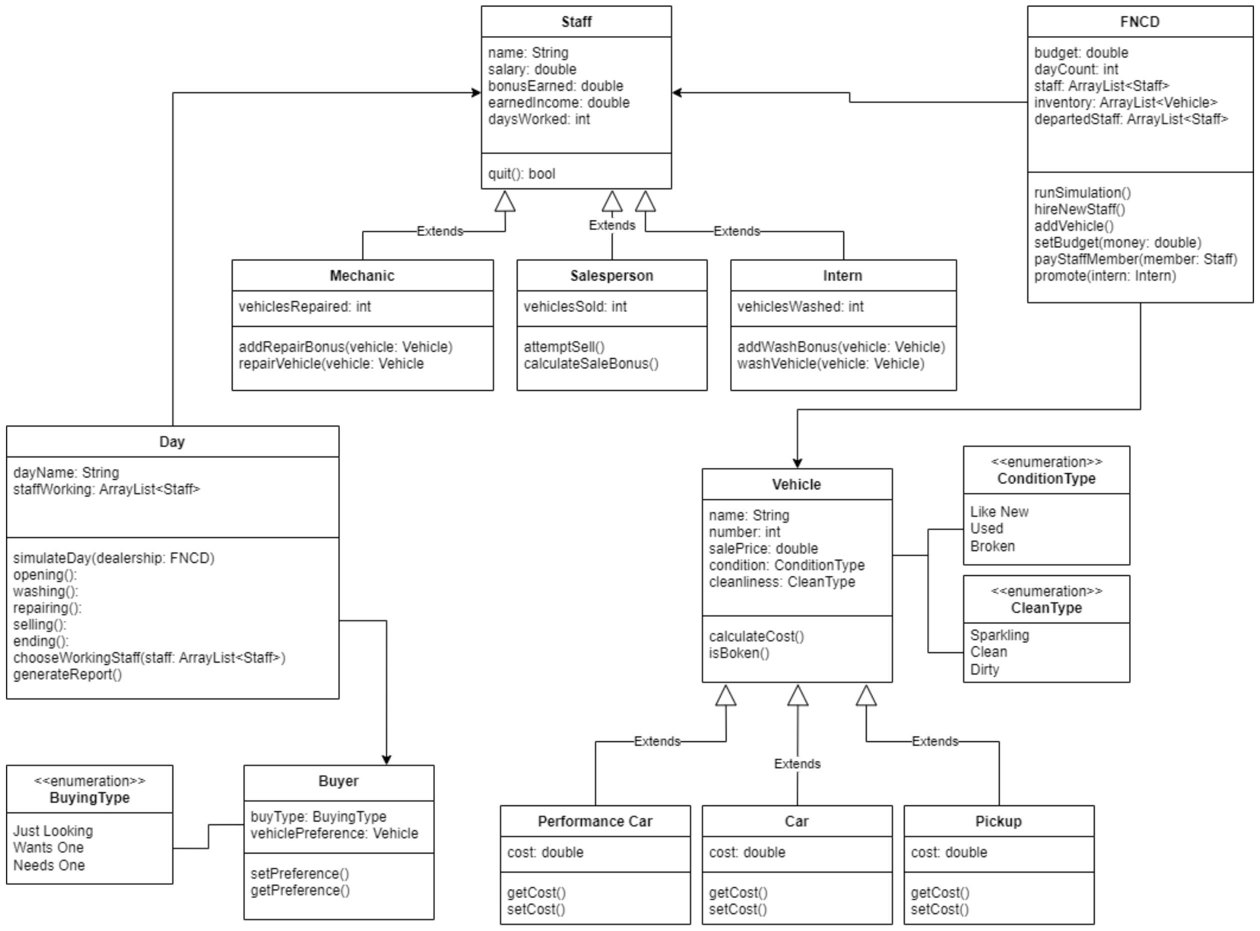
https://www.digitalocean.com/community/tutorials/java-8-interface-changes-static-method-default-method

2. The key difference between abstraction and encapsulation is that abstraction is the general concept of representing essential features of objects through a conceptual process, while encapsulation is the specific way in which information is being hidden, ie. using classes that encapsulate methods and private attributes. Abstraction can be the act of hiding implementation details from the user, but encapsulation is combining data under a single object or unit and solely involves information hiding.
    a. Example:

```
public class Encapsulate {

    // private variables declared
    // these can only be accessed by
    // public methods of class - encapsulation
    private String geekName;

    // get method for name to access
    // private variable geekName
    public String getName() { return geekName; }

    // set method for name to access
    // private variable geekName
    public void setName(String newName) { geekName = newName; }

    // method used to hide how area is calculated - abstraction
    public double calculateArea(double height, double width) {
            return height * width;
    }
}
```

[https://www.geeksforgeeks.org/difference-between-abstraction-and-encapsulation-in-java-with-examples/](https://www.geeksforgeeks.org/difference-between-abstraction-and-encapsulation-in-java-with-examples/)

## 3. UML Diagram of FNCD



Staff
name: String
salary: double
bonusEarned: double
earnedIncome: double
daysWorked: int
quit(): bool
FNCD
budget: double
dayCount: int
staff: ArrayList<Staff>
inventory: ArrayList<Vehicle>
departedStaff: ArrayList<Staff>
runSimulation()
hireNewStaff()
addVehicle()
setBudget(money: double)
payStaffMember(member: Staff)
promote(intern: Intern)
Extends
Extends
Extends
Mechanic
vehiclesRepaired: int
addRepairBonus(vehicle: Vehicle)
repairVehicle(vehicle: Vehicle
Salesperson
vehiclesSold: int
attemptSell()
calculateSaleBonus()
Intern
vehiclesWashed: int
addWashBonus(vehicle: Vehicle)
washVehicle(vehicle: Vehicle)
Day
dayName: String
staffWorking: ArrayList<Staff>
simulateDay(dealership: FNCD)
opening():
washing():
repairing():
selling():
ending():
chooseWorkingStaff(staff: ArrayList<Staff>)
generateReport()
Vehicle
name: String
number: int
salePrice: double
condition: ConditionType
cleanliness: CleanType
calculateCost()
isBoken()
<<enumeration>>
ConditionType
Like New
Used
Broken
<<enumeration>>
CleanType
Sparkling
Clean
Dirty
Extends
Extends
Extends
<<enumeration>>
BuyingType
Just Looking
Wants One
Needs One
Buyer
buyType: BuyingType
vehiclePreference: Vehicle
setPreference()
getPreference()
Performance Car
cost: double
getCost()
setCost()
Car
cost: double
getCost()
setCost()
Pickup
cost: double
getCost()
setCost()