Lesson Plan – Introduction to Object-Oriented Programming (OOP)

Today's class will introduce the students to the basic principles of Object-Oriented Programming and Design. Further, students will learn how to write code for classes, objects, and association.

Learning Outcomes

- Students will identify the advantages and benefits of object orientation.
- Students will differentiate between classes and objects.
- Students will be able to draw a basic UML diagram including class, inheritance, and association.
- Students will write code for classes, object creation, and class inheritance.

Managing the class

- Today's class is fun because it is introducing the students to a new way of thinking about writing and organizing code. Stress the importance of why we are doing this. While there is a bit of theory, it is important to engage students in varieties of activities. Some of these activities are conceptual whereas others are hands-on implementation. It is important that students understand both the theory and practice, and are able to connect them.
- Some students may struggle a bit with the design aspect of object orientation. It is expected at first. This is why you want to engage the students with concrete examples they can relate to.

Activities

1. **Welcome the students** (5 minutes)

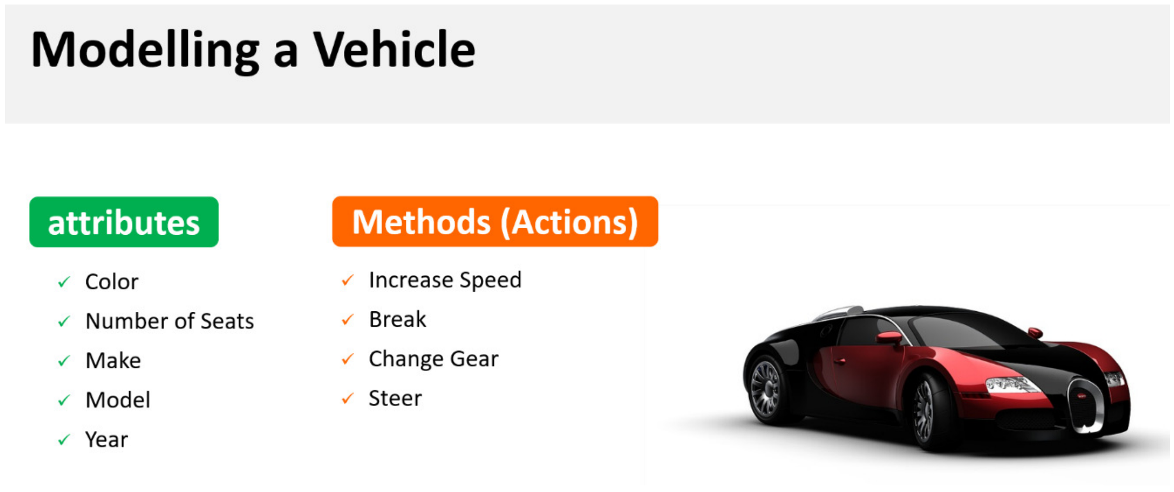
Welcome the students, and tell them shortly about today's class. Inform them that they will learn a new way of organizing their coding that is very useful in projects, as it improves readability and modularity. Explain that in object orientation, things are thought of as objects, which are an encapsulation of attributes and functions. Perhaps ask if some students are already familiar with it. This will give you quick feedback on your students.

2. Task (5 minutes)

- Ask the students to team up, and think about **modelling a vehicle**. What are some of the attributes that describe a vehicle? How can we operate a vehicle?

- As an instructor, you want to walk between students, and check on their progress. Perhaps engage in the discussion if you can.

- Quickly discuss a *possible* solution (Figure 1).



Modelling a Vehicle

attributes

- ✓ Color
- ✓ Number of Seats
- ✓ Make
- ✓ Model
- ✓ Year

Methods (Actions)

- ✓ Increase Speed
- ✓ Break
- ✓ Change Gear
- ✓ Steer

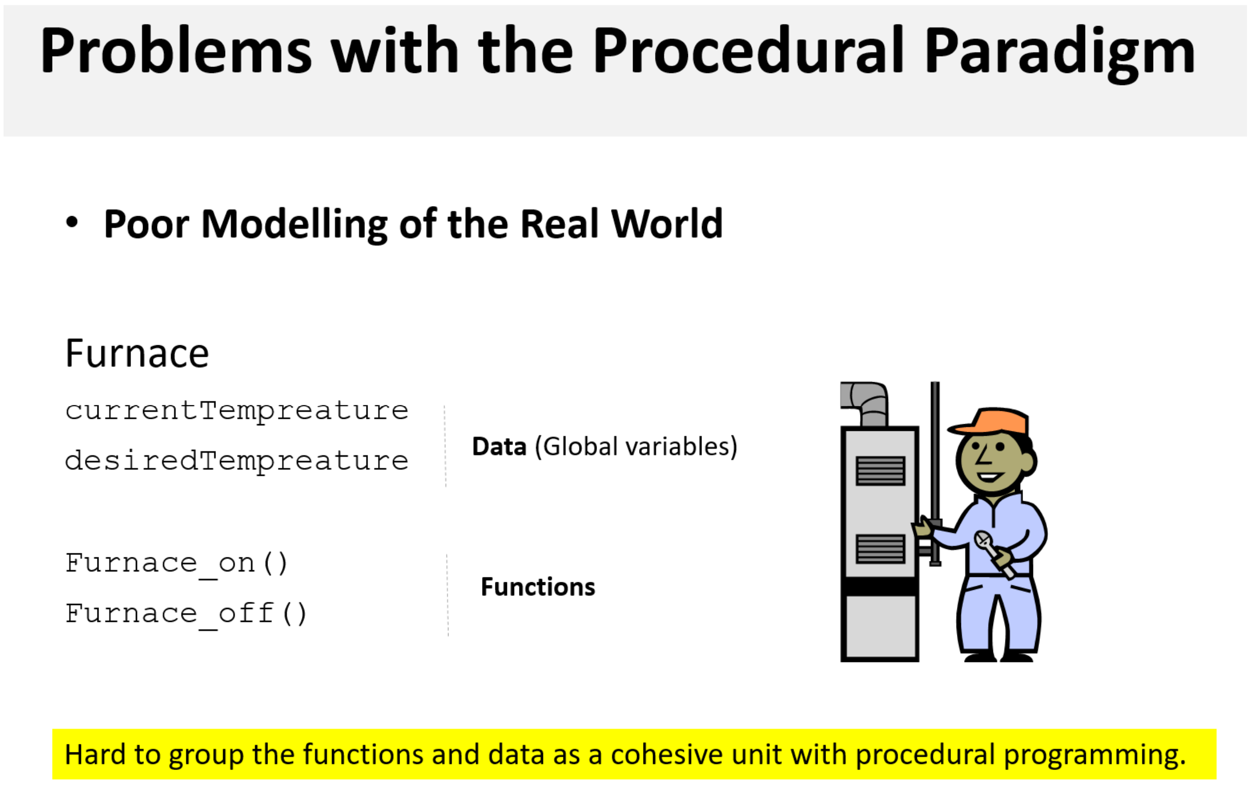
Figure 1

3. Why Object Orientation? (5 minutes)

- Tell the students why object orientation is important, and how it can improve the code quality in terms of readability, modularity, testability, and reusability. All of these are essentials in large team-based projects.

- Give the students a vivid scenario of badly-managed code. For instance, ask the students to imagine a group of coders working on a car rental system. A couple of coders are responsible for the business logic and interfacing with the database, and the other coder is responsible for the user interface. Imagine that their work is tightly coupled meaning that when the business logic coders change their code, it breaks/affects the user interface, and vice versa. Such a project will be a nightmare. That’s the opposite of object-oriented code where the code is structured into independent modules that can smoothly interact with one another.

- Give the students a quick example on why procedural programming (where code is organized around variables and functions) isn’t a good solution (variables are declared a global, and any function can change them. It is hard to group functions and attributes as a cohesive unit).



Problems with the Procedural Paradigm

- **Poor Modelling of the Real World**

Furnace

```
currentTempreature
desiredTempreature
```

Data (Global variables)

```
Furnace_on()
Furnace_off()
```

Functions

Hard to group the functions and data as a cohesive unit with procedural programming.

Figure 2

4. **Explaining Objects** (10 minutes)

- Objects can be physical (e.g. furnace, car) and abstract such as bank accounts. Break down each object in terms of attributes and methods (Figure 3).

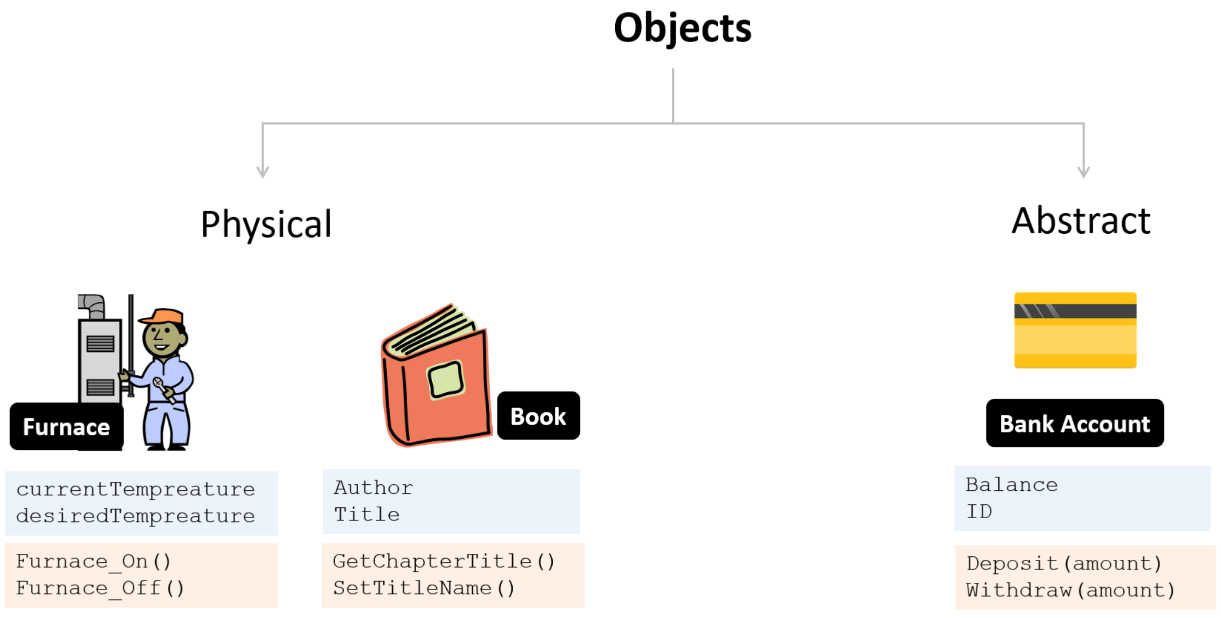


Figure 3

- Objects can also consist of other objects, and interact with other objects. Further, events may happen to objects (e.g. a button is clicked).

- Explain to the students the difference between classes and objects. Classes are the blueprints, they define the variables and methods. Objects are the instances of the classes. Give the students an example like this one (Figure 4). Also explain that these boxes are basic building blocks of UML (Unified Modelling Language) diagrams that illustrate code concepts visually.

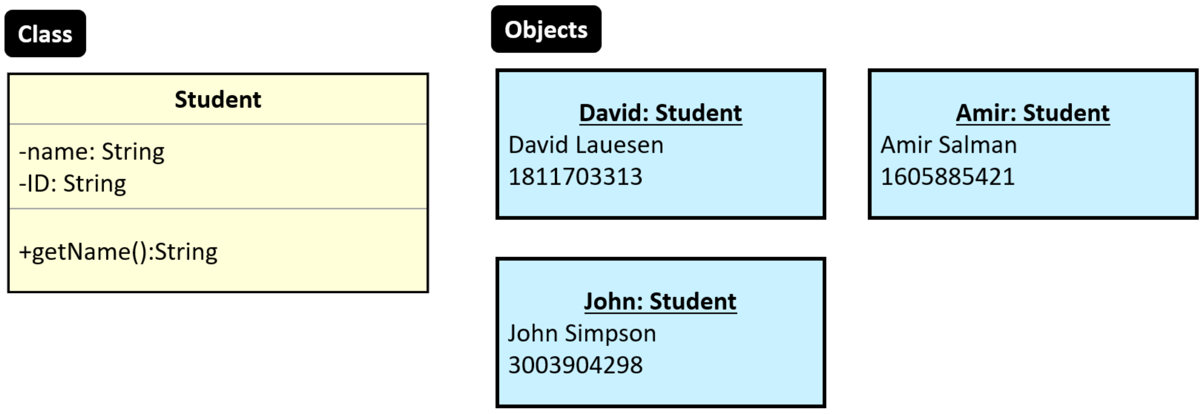


Figure 4

Implementation

- It is time to get started on implementing the concepts we talked about. As an instructor, go ahead and write code for the `Rectangle` class (Figure 5). It is important that you write down the code line by line in front of the students, and also instruct them to write the code with you (Listing 1). Save it as a file named as `Rectangle.java`

- It is important as you write the code, you emphasize these things:

  - ✓ Class names start with an uppercase letter.
  - ✓ Class attributes are typically private (for protection), and they start with a lowercase letter.
  - ✓ Classes have a default constructor, but you typically want to define your own. Constructors are special methods that are called when an object of that class is created. They are used for initializing the attributes.
  - ✓ The `this` keyword refers to this particular object. Use the constructor to explain it.

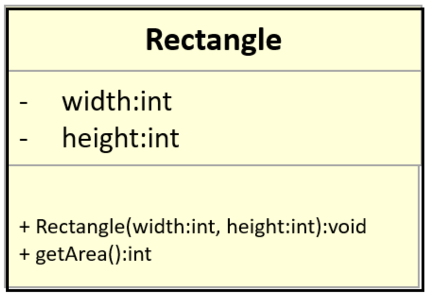


Figure 5

```
public class Rectangle {
  private int width;
  private int height;
public Rectangle(int width, int height) {
  this.width=width;
  this.height=height;
}

public int getArea() {
  return this.width*this.height;
 }
 }
```

Listing 1

- Next, you want to test the `Rectangle` class by creating a `Rectangle` object in a `Test` class (see the code below). We create a 5X4 `Rectangle`. We calculate the area of it by calling the `getArea` method. We store the result in a variable named `area`, and then display the result on the screen (Listing 2).

```java
public class Test{
  public static void main(String[] args) {
   Rectangle rec=new Rectangle(5,4);
   int area=rec.getArea();
   //area is 20
   System.out.println("Area of Rectangle is: "+area);
  }
}
```

Listing 2

5. **Task** (5 minutes)

- Ask the students to individually create several rectangles with different heights and widths, and print out their areas on the screen. Walk between the students and help on the way if needed.

- Next ask the students to write the method isSquare which returns a boolean. This method checks if this rectangle is a square.

- In the last minute, you can walk the students through the solution (Listing 3).

```java
public boolean isSquare() {
return this.width==this.height;
}
```

Listing 3

6. **Task** (5 minutes)

- Ask the students to create the shown `Student` class as well as create two Student objects with the given information (Figure 6).

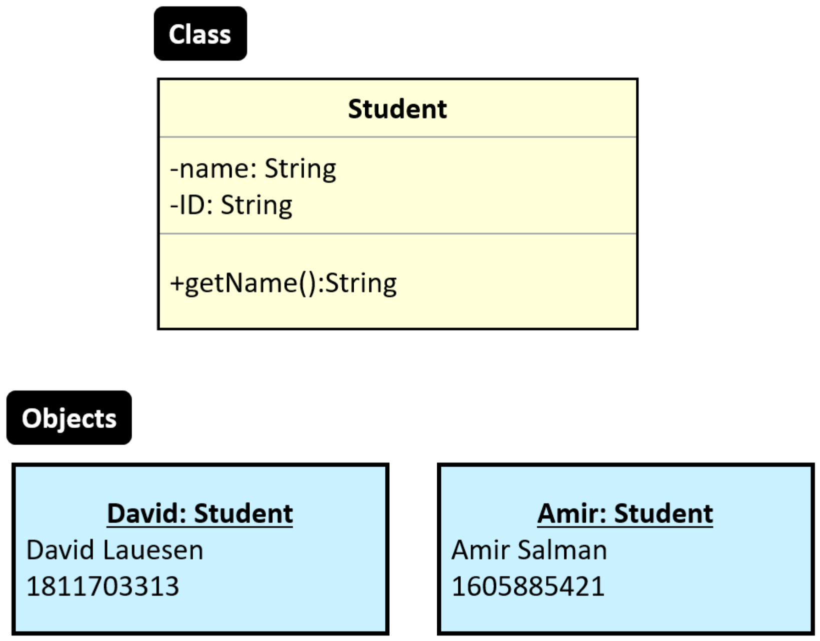


Figure 6

- The solution is as follows (walk the students through the solution. It shouldn't take more than a minute). See listing 4.

```
public class Student {
  private String name;
  private String ID;
  public Student(String name, String ID) {
    this.name=name;
    this.ID=ID;
  }
 public String getName() {
    return this.name;
  }
  }
```

Listing 4

- You can test the solution by creating two objects like this:

```java
public class Test{
  public static void main(String[] args) {
    Student Amir=new Student("Amir Salman","1605885421");
      System.out.println("First Student’s Name is: "+Amir.getName());
      Student David=new Student("David Lauesen","1811703313");
      System.out.println("Second Student’s Name is: "+David.getName());
}
}
```

Listing 5

7. **Features of Object Orientation** (10 minutes)

- Next, we talk about important features of object orientation such as encapsulation, inheritance, and polymorphism.
- **Encapsulation** simply means a class encapsulates its data (attributes) and behavior (methods). The inner workings of the class are hidden from the user. The user only accesses what’s needed. Each class is thought of a cohesive unit by itself. It can interact with other classes, but it ideally only knows about its own data and behavior.
- Class attributes are typically private so that outside users can’t tamper with them. Typically users can read them, sometimes users are not allowed to update them at all, and in some instances users can update them but according to the constraints imposed by the class.
- For example, the `Account` class (Figure 7) has a private `balance` attribute. You can read it by means of `checkBalance`. You can modify `balance` by means of calling `withdraw`. However, `withdraw` has constraints (e.g. you can’t withdraw more than a certain amount, you can’t withdraw money if you don’t have sufficient funds, etc.)

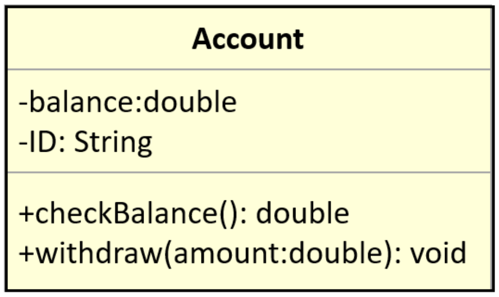


Figure 7

- **Inheritance** simply means a class inherits public and protected attributes and methods of a parent class (protected is a way to label some attributes and methods, which means they are only accessible to child classes). One important benefit of inheritance is *code reusability*, because you don't have to re-write code that already exists in the parent class.
- **Example:** Imagine that you work for a retail company that sells different kinds of items to customers. The company would like to keep track of customer and employee information. Customers and employees have some data in common (e.g. name, ID, e-mail), but they also have their own data. Customers have type (e.g. regular customer, premium customer). Further, customers can purchase items. Employees have different attributes such as salary and position type.
- To model these concepts, we make a parent class (`Person`) that has the common attributes (Name, Email, ID), and then we create child classes (`Customer` and `Employee`) that have their own added-on attributes and methods (see Figure 8).

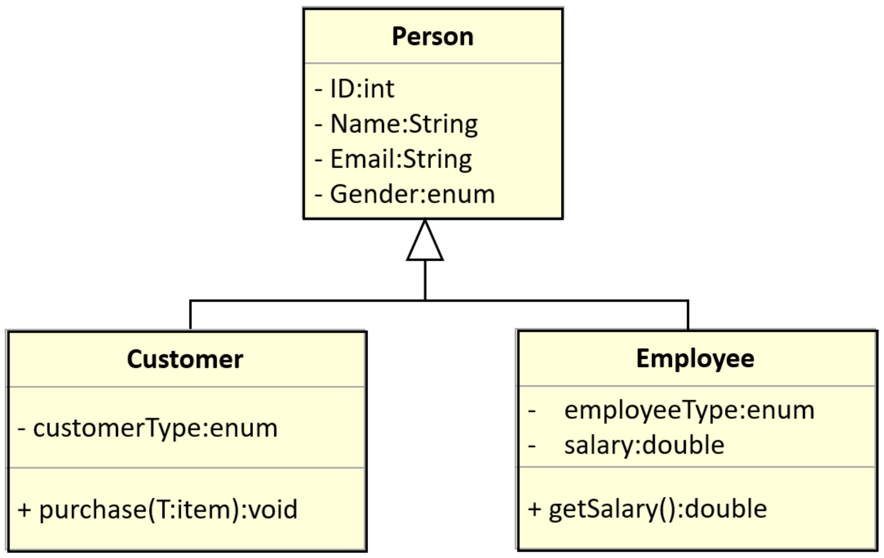


Figure 8

Implementation

- We would like to implement an *abstract* `BankAccount` class that has basic information such as account holder's name and the balance. Explain to the students that an abstract class is a class that can't be instantiated because it may not have full implementation for all methods. However, it serves as a basis for inheriting classes.
- It also has a `deposit` method that allows the user to deposit an amount of money into the account. The `widthdraw` method is only declared. The child classes must provide an implementation for it.
- The child class `CheckingAccount` inherits the attributes and methods from a `BankAccount` (Instructor: note that private attributes can't be inherited but `BankAccount` only has protected and public attributes and methods). `CheckingAccount` has a maximum withdrawal attribute (that imposes a constraint on how much money one can withdraw). `CheckingAccount` also provides an implementation for the withdraw method that it inherited from `BankAccount`. Notice the optional @Override annotation. While it is

not required, it is a form of code documentation that tells the developer that the method has been overriden by the child class.

- Write the code for both of the classes, and ask the students to write it with you (See Figure 9 and Listings 6 and 7).

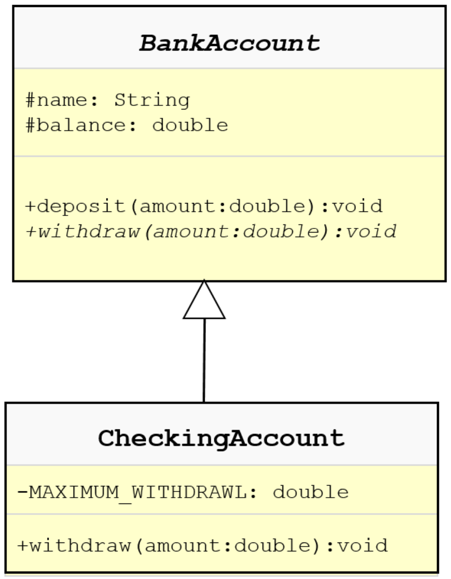

BankAccount
#name: String
#balance: double
+deposit(amount:double):void
+withdraw(amount:double):void
CheckingAccount
-MAXIMUM_WITHDRAWL: double
+withdraw(amount:double):void

Figure 9

```
public abstract class BankAccount {
  private String name;
  protected double balance;

  public BankAccount(String owner,double balance){
    this.name=owner;
    this.balance=balance;
  }
  public void deposit(double amount){
    this.balance+=amount;
   }
    public abstract void withdraw(double amount) throws Exception;
   }
```

Listing 6

```
public class CheckingAccount extends BankAccount {

  private final double MAXIMUM_WITHDRAWL=1000;

  public CheckingAccount(String name,double balance){
      super(name,balance);
  }

@Override
  public void withdraw(double amount) throws Exception{
    if(amount>MAXIMUM_WITHDRAWL)
        throw new Exception("You can't withdraw more than "+MAXIMUM_WITHDRAWL);
    if(amount> super.balance)
       throw new Exception("You can't withdraw more than the balance");
      super.balance-=amount;
     }

   }
```

Listing 7

- It is important that you empahsize the meaning of super, which is a reference to the parent class. It allows you to access the parent's class attributes and methods (See Listing 7.)
- To test the code, We create a checking account, and withdraw money from it. Trace the code step by step to show the students how it works. Show the students what happens when the user attempts to withdraw more than the balance or allowed amount. (Don't talk so much about exception handling. Just cover it briefly).

```
public class Test{
  public static void main(String[] args) {
    BankAccount account=new CheckingAccount("Adam Green",500.00);
    try{
     account.withdraw(200);
     account. withdraw (400);
    }
    catch(Exception e){
      System.out.println(e.getMessage());
    }
    }
   }
```

Listing 8

8. **Task** (10 minutes)

- Let's build upon the previous example. Write a `SavingsAccount` class that inherits from `BankAccount` and has these specifications. (1) You can't withdraw money from a `SavingsAccount`. Exception happens if that was the case. (2) Whenever you deposit money, an interest rate is applied to the added amount based on this formula: New balance= old balance + interest rate * amount
- At this point the students are a bit tired, but this should be a doable exercise. Walk the students through the solution. Emphasize things like calling the parent class's constructor by means of "`super`".
- Also, emphasize the fact that the deposit method has been overridden based on the logic required in the task.

```java
public class SavingsAccount extends BankAccount{
private float interest_rate;
public SavingsAccount(float interest_rate,String name, double balance){
super(name,balance);
this.interest_rate=interest_rate;
}
@Override
public void deposit(double amount){
super.deposit(amount*interest_rate+amount);
  }
  @Override
  public void withdraw(double amount) throws Exception {
  throw new Exception("Not allowed to withdraw from a savings account.");
  }
  }
```

Listing 9

9. **Association** (5 minutes)

- Classes can be associated with other classes. **An association** implies that an object of one class is making use of an object of another class is indicated by a solid line connecting the two class icons
- Take a look at the following UML diagram. On the right-hand side, we see two classes: Guest and Room. They are connected via a line. On the line are two asterisks. An asterisk means zero or many. When we read this visual relationship, we can say a guest stays in zero or many hotel rooms. A room may have zero or many guests. On the left-hand side is an example of different Guest and Room objects that indicate the relationship. For instance, Kim stayed in two rooms (S101 and M13) at different times. Nancy stayed in room M13.

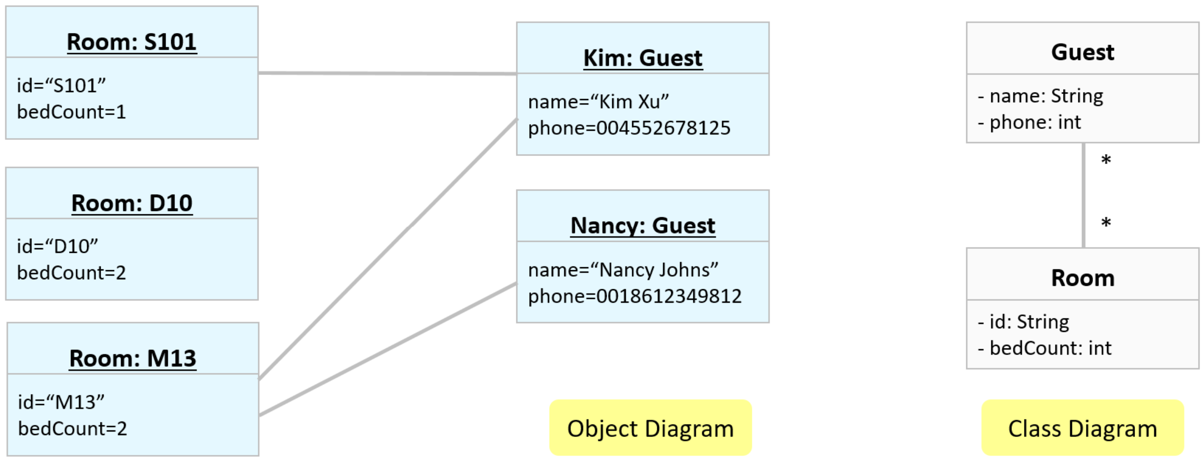


Figure 10

10. **Assignment** (5 minutes)

- At this point in time, you should be ready to wrap up the class. You may want to give the students an assignment to work on in the lab, or perhaps at home, and they can discuss it with you during office hours or the TA.
- The assignment is: We would like to build a system that supports a class enrollment system. The system allows students enroll in courses, and teachers to teach courses. A course could have multiple sections. The student receives a score after the course finishes.
- Solution: First, we need to draw the concepts mentioned in the task: Student, Course, Section, and Professor. We also an enrollment class to keep track of when students enrolled in a class and what score they got.
- Also, highlight the difference between Course and Section. Course is just the information of a course in a catalog. Section is an offered course. It has a classroom and a schedule. A course may be offered many times. This is why there is an asterisk on the Section's side and one on the course side (one course has many sections). A professor may teach many sections, and a section is taught by one professor. A student may have many enrollments, and a section also may have many enrollments.

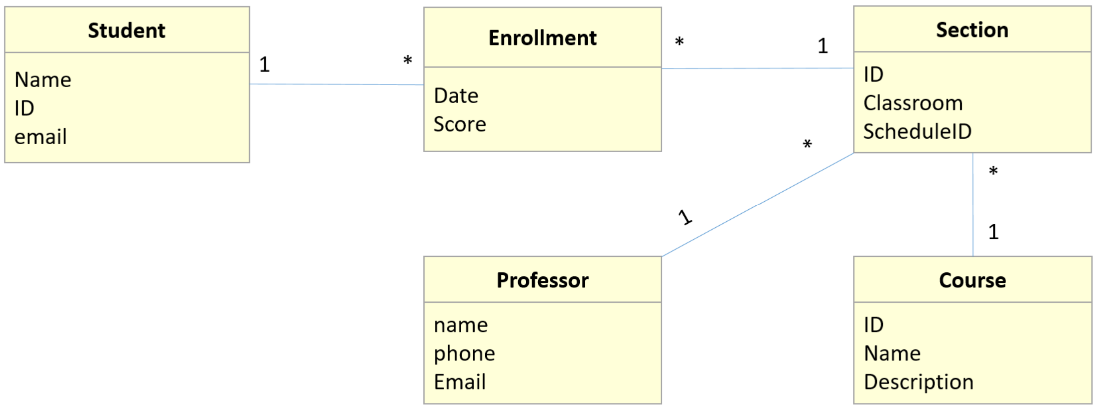


Figure 11