Object-Orientated Programming

CMM24

RGU (2023)

Session 10 (Object Orientated Programming)

Return to enums

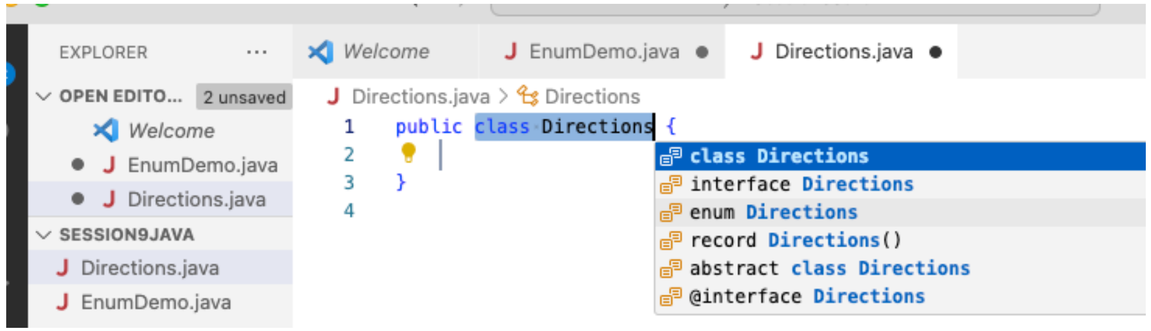
More on abstract classes

CMM024

Summary

- **Going over enums and finding out their uses**
- **Discussion abstract method further**

What learned last session




```java
public enum Directions {
    NORTH,
    SOUTH,
    EAST,
    WEST
}
```

```java
public enum DaysOfWeek {
    MONDAY, TUESDAY, WEDNESDAY,THURSDAY,FRIDAY,SATURDAY,SUNDAY
}
```

More on enums

- What we also learned is that you use them in the same manner as you do for any other data type but there are a two more things!
  - To set the value you use the Datatype i.e. Gender.MALE or Gender.FEMALE here
  - To obtain the index you must use the ordinal() method of the Datatype!
  - Index, like arrays starts at 0

```
public enum Gender {
    FEMALE,
    MALE
}
```

```
public class Person {

    private String name;
    private Gender gender;

    public Person(String name, Gender gender ){
        this.name = name;
        this.gender = gender;
    }

    public Gender getGender(){
        return gender;
    }

    public String getName(){
        return name;
    }

    public String toString(){
        String st = "";
        st += "Name: " + this.name;
        st += "\tGender: " + this.gender.toString();
        return st;
    }
}
```

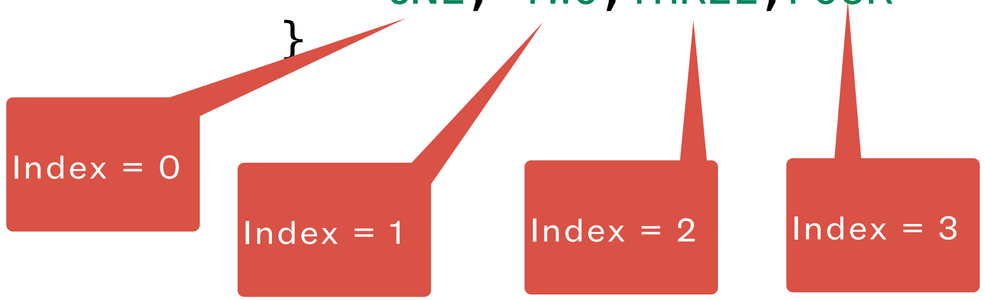
```
Person p1 = new Child("David", Gender.MALE);
Person p2 = new Child("Mary", Gender.FEMALE);
Person p3 = new Parent("Robert", Gender.MALE);
Person p4 = new Parent("Suzy", Gender.FEMALE);
```

More on enums

```
public enum CustomerType {
    CUSTOMER, COMPANY
}
```

- Here we use the index to compute the discount. The higher the index, more discount is given. Hence a company will have higher discount here
- Here we increase the index by 1 to avoid 0 as anything multiplied by 0 is 0

```
public enum DiscountLevel {
    ONE, TWO,THREE,FOUR
}
```




```
double originalPrice = 20.0;
double discountMultiplier = 0.3; //30%
CustomerType customer = CustomerType.COMPANY;
DiscountLevel discountLevel = DiscountLevel.FOUR;
double totalDiscountLevel =  (customer.ordinal()+1) * (discountLevel.ordinal()+1);
double newCost  = originalPrice  - (totalDiscountLevel* discountMultiplier);
System.out.println("Customer Type: " + customer.toString() + " Discount Level: " + discountLevel);
System.out.println("Original Price: " + originalPrice + " new Price: " + newCost);
```

```
/bin EnumDemo
Customer Type: COMPANY Discount Level: FOUR
Original Price: 20.0 new Price: 17.6
jean-claude@MacBook-Pro-2 Session9Java %
```

```
/bin EnumDemo
Customer Type: CUSTOMER Discount Level: ONE
Original Price: 20.0 new Price: 19.7
jean-claude@MacBook-Pro-2 Session9Java %
```

More on enums

- But we could use an array along side to provide us with real discount values!

```
double originalPrice = 20.0;
double[] discountValues = {0.1, 0.2, 0.3, 0.4};
CustomerType customerType = CustomerType.COMPANY;
DiscountLevel discountLevel = DiscountLevel.ONE;
System.out.println("Original Price: " + originalPrice);
double baseDiscountPercent = discountValues[discountLevel.ordinal()];
System.out.println("Base Discount(%): " + baseDiscountPercent);
double finalDiscountPercent =  baseDiscountPercent * (customerType.ordinal()+1);
System.out.println("Final Discount(%): " + finalDiscountPercent);
double discount = finalDiscountPercent* originalPrice;
System.out.println("Discount(%): " + discount);
double newCost  = originalPrice  - (finalDiscountPercent* originalPrice);
System.out.println("New Price: " + newCost);
```

We use the ordinal value of the DiscountLevel to find the real discount percent

Companies discount is twice the amount for customers

More on enums

- But we could use an array along side to provide us with real discount values, this time for the Customer Type. No extra discount for normal customer (1) and 20% for companies (1.2)

```
double originalPrice = 20.0;
double[] discountValues = {0.1, 0.2, 0.3, 0.4}; //10%, 20%, 30%, 40%
double[] custumerDiscountValues = {1, 1.2}; //customer:  NOTHING, company: 20%
CustomerType customerType = CustomerType.COMPANY;
DiscountLevel discountLevel = DiscountLevel.ONE;
System.out.println("Original Price: " + originalPrice);
double baseDiscountPercent = discountValues[discountLevel.ordinal()];
System.out.println("Base Discount(%): " + baseDiscountPercent);
double finalDiscountPercent =  baseDiscountPercent * (custumerDiscountValues[customerType.ordinal()]);
System.out.println("Final Discount(%): " + finalDiscountPercent);
double discount = finalDiscountPercent* originalPrice;
System.out.println("Discount(%): " + discount);
double newCost  = originalPrice  - (finalDiscountPercent* originalPrice);
System.out.println("New Price: " + newCost);
```

```
Customer Type: COMPANY Discount Level: ONE
Original Price: 20.0
Base Discount(%): 0.1
Final Discount(%): 0.12
Discount(%): 2.4
New Price: 17.6
```

```
Customer Type: CUSTOMER Discount Level: THREE
Original Price: 20.0
Base Discount(%): 0.3
Final Discount(%): 0.3
Discount(%): 6.0
New Price: 14.0
```

```
Customer Type: COMPANY Discount Level: FOUR
Original Price: 20.0
Base Discount(%): 0.4
Final Discount(%): 0.48
Discount(%): 9.6
New Price: 10.4
```

More on abstract classes

- Abstract superclasses and concrete subclasses are very good when some of their instance objects have specialised functionality.
- We have discussed employees, where you have part-time and full-time. The difference is in terms of how they get paid. When it came to store them in a collection, we stored them as Employee, because they both are, but when it came to calculate their monthly wage, we had issues. We resolved these issues by:
    - Creating a super class Employee, make it abstract, declare and abstract method getWages().
    - Creating two subclasses called PTEmployee, and FTEmployees, which were forced to implement that getWages() method.
    - This method implemented by the subclasses was used in the superclass to display their monthly wage regardless if they were part-time or full-time.

More on abstract classes

- Abstract superclasses and concrete subclasses are very good when the subclasses have a lot of fields and functionality in common. This can reside in the superclass
- Another example we came across is when we model graphical shapes. Some were triangles, some were rectangular or circular. They are still all shapes but their areas and circumferences are calculated differently. We solved that issues by:
    - Creating a super class Shape, make it abstract, declare and abstract methods getArea() and getPerimeter() .
    - Creating subclasses called Rectangle, and Circle (etc.), which were forced to implement these two abstract methods.
    - These two methods implemented by the subclasses were used in the Shape superclass to display these values regardless of their shapes!

More on abstract classes

- Remember that subclasses extending a superclass inherits all the fields and methods of the superclass.Hence subclasses can invoke superclass methods!
- You can put all the common behaviour in an abstract base class and having some abstract methods that will be implemented by classes extended the superclass to specifically implement the functionality that is needed
- Example: modelling a payment system. The superclass BasePayment define the main operation of charging money from customers, relying on concrete subclasses to implement specific methods of performing that action through an abstract method performPayment(). Hence some concrete subclasses like below can be programmed, all of which will implement that method
    - PaypalPayment
    - AlipayPayment
    - BankPayment (etc.)

Next week

- **Remedial Week**
-