Question 1

(a) The object-oriented programming model provides **encapsulation** and **abstraction** benefits to programmers. Briefly explain this statement.

(b) Binding is the process of connecting a method call to a method body. Using the concepts of overloading and overriding explain the difference between static binding and dynamic binding.

Question 2

The Filao hotel is planning to implement a hotel room reservation system in order to manage room bookings. You have been requested to perform the following tasks.

(a) Implement a class named RoomBooking with the following **private** fields:

room_type	<*String*>	Room type (Standard, Deluxe, Family)
num_days	<int>	Number of days room is booked

The class contains

- a constructor that requires arguments for *room_type* and *num_days*.
- a method that displays the details of the room booking
- a method that check the room type and calculates and returns the booking charges, based on the following daily room charges

	Standard	Deluxe	Family
US $ (per day)	100	200	400

```
public class RoomBooking{
        private String room_type;
        private int num_days;

        public RoomBooking(String room_type, int num_days){
                this.room_type = room_type;
                this.num_days = num_days;
        }

        public void display(){
                System.out.println("ROOM TYPE:" + room_type);
                System.out.println("NUMBER OF DAYS:" + num_days);
        }
        public float charges(){
                if (room_type.equals("Standard"))
                        return ((float)(100 *num_days));
                else if (room_type.equals("Deluxe"))
                        return ((float)(200*num_days));
                else if (room_type.equals("Family"))
                        return ((float)(400*num_days));
                else
                        return(0.0);

        }
}
```

(b) Implement another class named PanoramicRoomBooking, which inherits from the RoomBooking class. A Panoramic room carries an additional charge on top of the room charges and the additional charge is based on the type of view from the room as given below:

	Garden View	Pool View	Sea View
US $	20	30	50

The class contains one **private** field

view_type *<String>* Type of view from room (Garden, Pool, Sea)

The class contains the following methods:

- a constructor that requires arguments for *room_type, num_days,* and *view_type*
- a method that displays the details of the room booking
- a method that check the view type and calculates and returns the room charges

```
public class PanoramicRoomBooking extends RoomBooking {
    private String view_type;

    public PanoramicRoomBooking (String room_type, int num_days, String view_type){
        super(room_type, num_days);
        this.view_type = view_type;
    }

    public void display(){
        super.display();
        System.out.println("VIEW TYPE:" + view_type);
    }
    public float charges(){
        if  (view_type.equals("Garden view"))
            return (super.charges() + 20;
        else if  (view_type.equals("Pool view"))
            return (super.charges() + 30);
        else if (view_type.equals("Sea view"))
            return (super.charges() + 50);
        else
            return(0.0);

    }

}
```

(c) Implement a class CreateBookings that will create an array of 5 room bookings as follows:

Bookings Number	Room Type	No of days	
1	Standard	2	
2	Family	3	Garden view
3	Deluxe	1	

4	Standard	2	Pool view
5	Deluxe	5	Sea view

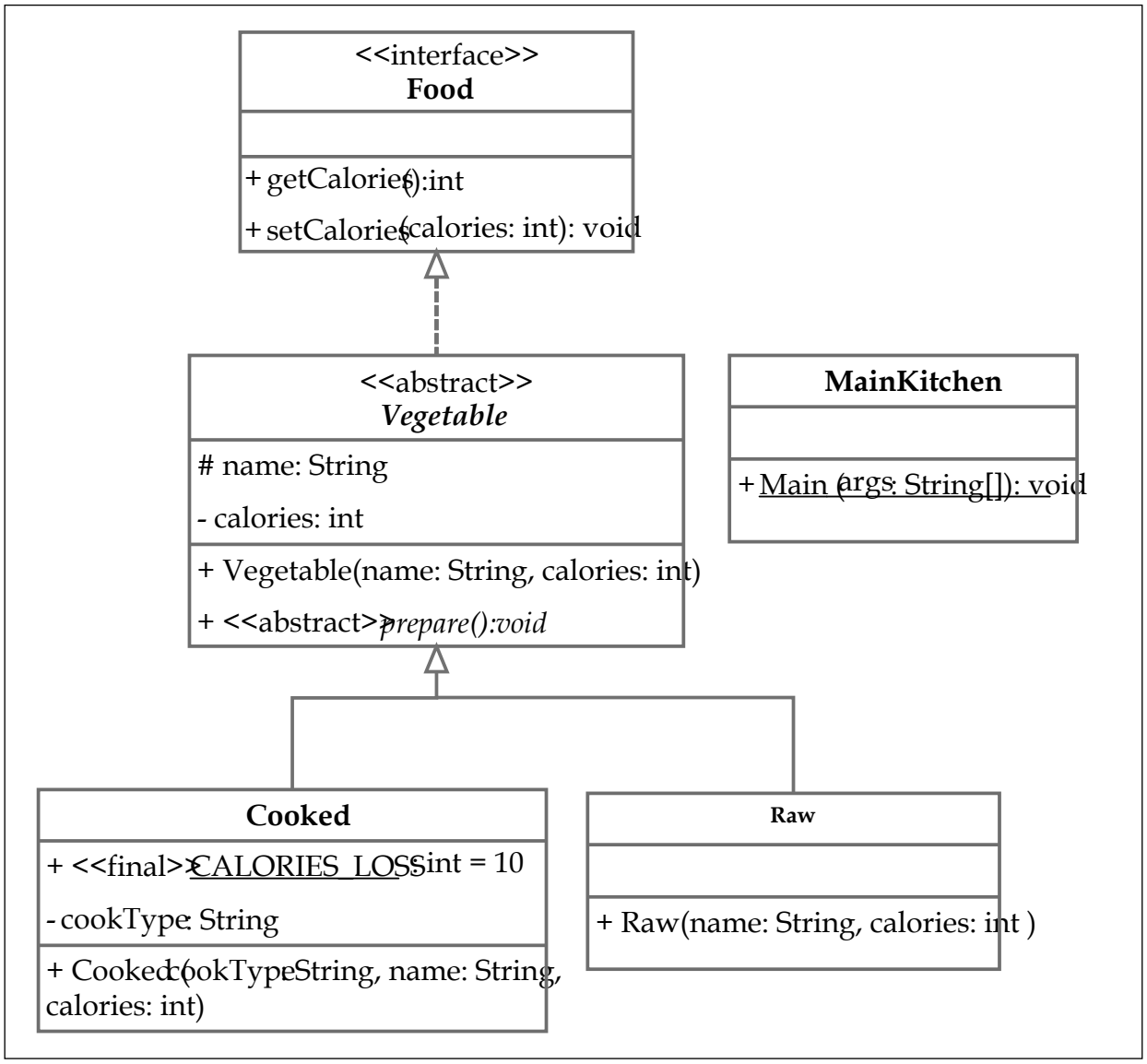
Your program should then display

- **The room details of each booking**
- **The total charges of all the bookings**

```
public class CreateBookings{

public static void main (String args[]){
        RoomBooking[] roomlist = new RoomBooking[5];

        roomlist[1] = new RoomBooking ("Standard",2);
        roomlist[1] = new PanoramicRoomBooking ("Family",3, ""Garden view");
        roomlist[2] = new RoomBooking ("Deluxe",1);
        roomlist[3] = new PanoramicRoomBooking ("Standard",2,"Pool view");
        roomlist[4] = new PanoramicRoomBooking ("Deluxe",5, "Sea view");


                //Displaying  and calculating total charges
                float tot_charges=0.0;
                for (int i = 0, i < roomlist.length; i++){
                        roomlist[i].display();
                        tot_charges +=roomlist[i].charges();
                }
                System.out.println("TOTAL CHARGES:" + tot_charges);
        }

}
```

Question 3

The class diagram below shows the interface *Food,* the abstract class *Vegetable* and its subclasses *Cooked* and *Raw*, denoting cooked vegetables and raw vegetables respectively.



The class *Vegetable* contains:

- One protected variable *name (String)* and one private variable *calories (Integer)*
- The protected variable can be accessed by its subclasses and classes in the same package.
- One constructor that initializes the *name* and *calories* to the given values.
- One abstract method *prepare()*.
- The class *Vegetable* should implement the interface *Food*.
- The above class diagram gives all the data and methods.

The *Cooked* class contains:

- One constant variable CALORIES_LOSS *(Integer)* and an instance variable *cookType (String)*
- One constructor that initializes the *cookType, name* and *calories* to the given values.

- **Note that the *Cooked* class only implements methods specified and implied by the relationship diagram.** Thus, the *Cooked* class displays information based on the information provided below:

  The cooking type can be *microwave-cook*, *boil*, etc., and allows instantiation of different cooked vegetables. When a vegetable is cooked, the number of calories decreases by 10 calories as opposed to a raw vegetable, which retains all of its calories.

- Sample output from the **Cooked** class having specified *cookType* as '*Boil*', *name* as '*Beetroot*' and *calories* as *60*.

```
Boil Beetroot. Calories left: 50
```

The *Raw* class contains:

- One constructor that initializes the *name* and *calories* to the given values.
- **Note that the *Raw* class only implements methods specified and implied by the relationship diagram.** Thus, the *Raw* class displays information based on the information provided below:

  Sample output from the **Raw** class having specified *name* as '*Onion*' and *calories* as *20*.

```
Slice Onion. Calories: 20
```

(i) Write the Java code(s) for interface ***Food***.

```
public interface Food
{
   public int getCalories();
   public void setCalories(int calories);
}
```

(ii) Write Java code(s) for the classes ***Vegetable***, ***Cooked*** *and* ***Raw*** using the information provided in the class diagram and description.

Vegetables

```
public abstract class Vegetable implements Food {
   protected String name;
   private int calories;
   public Vegetable(String name, int calories){
      this.name=name;
      this.calories = calories;
   }
   public int getCalories(){
```

```
      return this.calories;
   }
   public void setCalories(int calories){
      this.calories = calories;
   }
   public abstract void prepare();
}
```

Cook

```
public class Cooked extends Vegetable {
   public static final int CALORIES_LOSS = 10;
   private String cookType;

   public Cooked(String cookType, String name, int calories){
      super(name, calories);
      this.cookType = cookType;
   }

   public void prepare(){
      setCalories(getCalories()-CALORIES_LOSS);
      System.out.println(cookType+" "+super.name+". Calories left: "+getCalories());
   }
}
```

Raw

```
public class Raw extends Vegetable{
     public Raw(String name, int calories){
      super(name, calories);
   }

   public void prepare(){
      System.out.println("Slice "+super.name+". Calories: "+getCalories());
   }
}
```