Meal Master

OOP Concepts:

1. Inheritance

When a class inherits or extends to the properties of another class, this is known as inheritance. The class being inherited becomes the super class and the class that inherits becomes the child class.

Multiple Inheritance is when a class inherits multiple classes. Java doesn't support multiple inheritances, but we can achieve it through interfaces, by extending a class and implementing interfaces. A class can implement multiple interfaces

Inheritances in the project:

- MainWindow extends JFrame
- ScreenManager extends JPanel
- Screen extends JPanel
- LoginScreen extends Screen
- SignUpScreen extends Screen
- HomeScreen extends Screen
- AddFoods extends Screen
- User implements Serializable
- FoodItem extends JPanel
- FoodCard extends JPanel implements MouseListener (Multiple Inheritance)

2. Abstraction

When we hide the implementation details but only provide functionality, we are achieving abstraction. Abstraction in simple means to hide how the work is done, and instead the focus is only on getting the work done. For example, we use a calculator for calculation, but we don't know how the calculator does the calculation, because its of no use for us. Therefore, the calculator is providing functionality, but hiding its implementation.

There are two ways of achieving abstraction (in java):

- By setting class members private and providing the functionality of those members in some other way
- By using abstract classes or interfaces

Abstract Classes in the project:

- UserManager
- DietPlan
- Screen

3. Encapsulation

Encapsulation is a concept of “binding” or “compiling” or “bundling” or “Wrapping” details and functionalities inside a single unit. We wrap all the relevant and related data into a single unit and provide alternate ways to access those members. Encapsulation can be achieved by setting all the attributes private and creating public getter and setter methods to access those members.

Pure Encapsulated Class in the project: User (it has all attributes marked as private and public getter methods are created to access them)

4. Polymorphism

Polymorphism in literal means “Many Forms”. When the same functionality can be achieved in different ways or forms, it is known as polymorphism.
Polymorphism can be achieved in two ways:

- Method Overloading
- Method Overriding

Polymorphism is of two different types:

- Compile-Time Polymorphism (Overloading)
- Runtime Polymorphism (Overriding)

Overloading is when a function is created more than once in the same class with the same name but different parameters or arguments. This polymorphism happens during compile time, as the compiler binds them before execution.

Overriding occurs with inheritance. When a sub-class re-implements or re-defines a method of the super class and changes its functionality (code inside the body not the parameters), this is overriding. And this type of polymorphism happens during runtime, when the objects are created and how the methods are being invoked or called.

This project has used Overriding.

Keywords and their Usages in project:

1. static (attributes and functions)

The static keyword is used to ensure that the attributes or methods are ‘singular’, they are bind at compile time, and they do not belong to any class, instead they are accessed by the class name and they are shared by all the objects of the class. If one object “obj1” changes the value of a static variable, the change will be done for the other object “obj2” or objects (no matter how many) of the same class. If a method is static, it does not belong to the class, instead it is shared among all the objects.
Note that if a method is marked as static, you cannot override it, as it is bind during compile time and it does not belong to the class. All static members are “statically”

bonded during compile time, whereas method overloading is bonded “dynamically” during runtime.

The static keyword is also used for creating static blocks, which are used for initializing members only once while the object is being assigned memory during runtime. It provides an option for initializing all objects at once during runtime.

2. **abstract**

The abstract keyword is used in the project to create abstract classes (mentioned above) and an abstract method (init()). This keyword is used to declare a class or method abstract. Making a class abstract ensures that no one can create an object of the class and allows the creation of abstract methods. If a class is abstract it is not necessary that it must have an abstract method, but if the class is marked abstract and it does not have any abstract method, then it is useless to mark the class abstract. The project contains two abstract classes (UserManager and DietPlan) without any abstract method, this does not mean that making them abstract is useless. We marked them abstract to ensure no one can make an object of the class, and it has all members as static, so that they can only be accessed by the class name. This is done due to the limitation in java to create “Static Classes”. Java does not support static classes, and the only way to achieve the behavior of a static class in java is to create an abstract class and mark all of its members as static. The reason to create static classes was that creating objects for these two classes was useless and a waste of memory, and a static class here would be helpful, but since java does not support static classes, we opted for this alternate, kind of static class.

Note that to create an abstract method you must declare its class abstract as well, otherwise it won’t allow you to create abstract method.

3. **final**

The final keyword is used to mark members as “constant” or “unchangeable”. If you mark an attribute as final its value cannot be changed once assigned. Note that it is similar to constants but it’s not exactly like a constant. Java has both const and final keyword and the main difference is just the type of binding. ‘const’ keyword is used when you want to bind the value during compile time and its value MUST be assigned during creation. ‘final’ keyword is used when you want to bind the value during runtime and its not necessary to assign it a value during creation. However, if you mark the attribute as static as well as final, then you MUST assign it a value during creation as static variables are bind during compile time, and if you try not to assign it a value, it will generate a compile-time error as the value of the uninitialized static final variable is always null, that means it’s useless to create the variable if it will remain as null, since final attributes cannot be changed once they are assigned a value.

If you mark a method as final, you cannot override or overload it since its implementation is “FINAL” or in other words unchangeable.

4. **Super**

The super keyword is used in inheritance, when a child class wants to invoke the members of the super/parent class. If a parent class has an attribute “name” with a

value “Tahira Tufail”, and a child class has the same attribute “name” with a value “Syed Dawood”, if you try to print the value of name in the child class, you will get an output “Syed Dawood”. But if you wanted the value of the parent class, you can access it using super keyword like so: super.name, which will give an output “Tahira Tufail”.

You can also use the super keyword to invoke methods of super class, for example if you override the method of the super class and change its functionality, you can still access the functionality implemented in the parent class using super keyword.

You can also invoke the parent class constructor using the super keyword. If the parent has no constructor, it will invoke the default constructor, and if the parent has any constructor, it will invoke that constructor. To use super to access constructor, just add round brackets in front of super like so: super().

5. **This**

   The ‘this’ keyword is used when a class wants to invoke its own members. It is commonly used to avoid ambiguity, like if you have a class with attribute “name” and you pass the same variable name inside constructor and try to assign the attribute this variable like so: “name = name”, it will cause confusion as the compiler doesn’t know which “name” is it referring to, the local variable or the attribute. Here we can use ‘this’ keyword to differentiate the attribute with the local variable like so: ‘this.name = name’.

   You can also invoke the class constructor using the ‘this’ keyword. If the class has no constructor, it will invoke the default constructor, and if the class has any constructor, it will invoke that constructor. To use ‘this’ to access constructor, just add round brackets in front of ‘this’ like so: this().

6. **@Override**

   This is an annotation, and is optionally used to enhance code performance. This is used to mark any method as override, so that it becomes easier for the compiler to recognize it and makes execution faster.