OOPs Continuation

Any code in java need to be inside the class (OOP)

Class can have state alone

Class can have behaviour alone

Class can have both the state and behaviour (Encapsulation)

Classes will be having private variables and public methods

POJO Plain Old Java Object

- Private variables
- Public void set(<datatype> argument) // won't return
- Public <datatype> get() // return the value of the declared data type

When the local variable and method argument names are same we will use "this" keyword before the local variable. This will refers to the current class.

Ex:
```
public void changeEmpSalary(float salary) {
	this.salary = salary;
}
```

Here the both the local variable name and the argument passing name is salary. There will be the conflict with the same names. Hence we used this keyword before the variable name. This will refers to the classes' local variables.

Multiple classes

The good practice of coding is to write the code in multiple java files.

By using multiple files we can achieve the layered approach

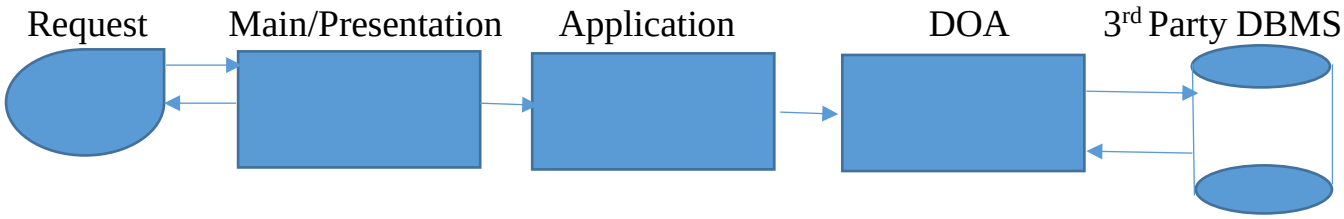
Modules – small code – loose binding – layered approach

Folders

Folders are the logically grouped multiple files. In java w call these as the packages.

Layers

1) Main
2) Service
3) DAO (Data Access Object)



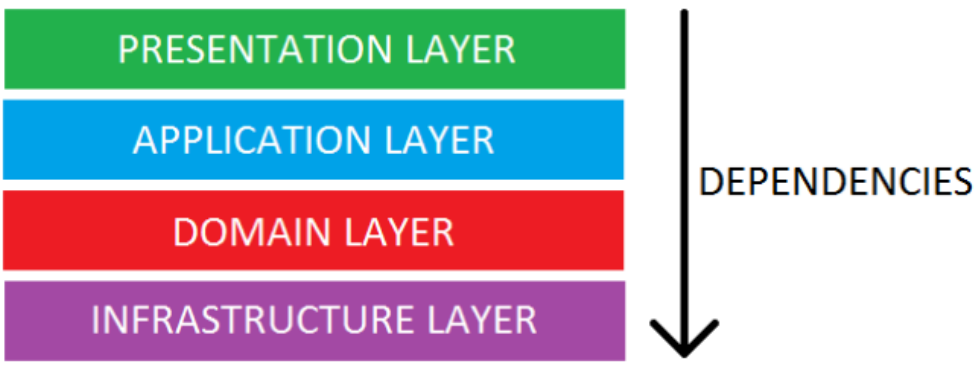
What Is Layered Architecture?

A Layered Architecture, as I understand it, is the organization of the project structure into four main categories: **presentation, application, domain, and infrastructure**. Each of the layers contains objects related to the particular concern it represents.

- **The presentation layer** contains all of the classes responsible for presenting the UI to the end-user or sending the response back to the client (in case we're operating deep in the back-end).
- **The application layer** contains all the logic that is required by the application to meet its functional requirements and, at the same time, is not a part of the domain rules. In most systems that I've worked with, the application layer consisted of **services orchestrating the domain objects to fulfil a use case scenario.**
- **The domain layer** represents the underlying domain, mostly consisting of domain entities and, in some cases, services. Business rules, like invariants and algorithms, should all stay in this layer.
- **The infrastructure layer (also known as the persistence layer)** contains all the classes responsible for doing the technical stuff, like persisting the data in the database, like DAOs, repositories, or whatever else you're using.

There are two important rules for a classical Layered Architecture to be correctly implemented:

1. All the dependencies go in one direction, from presentation to infrastructure. (Well, handling persistence and domain are a bit tricky because the infrastructure layer often saves domain objects directly, so it actually knows about the classes in the domain)
2. No logic related to one layer's concern should be placed in another layer. For instance, no domain logic or database queries should be done in the UI.



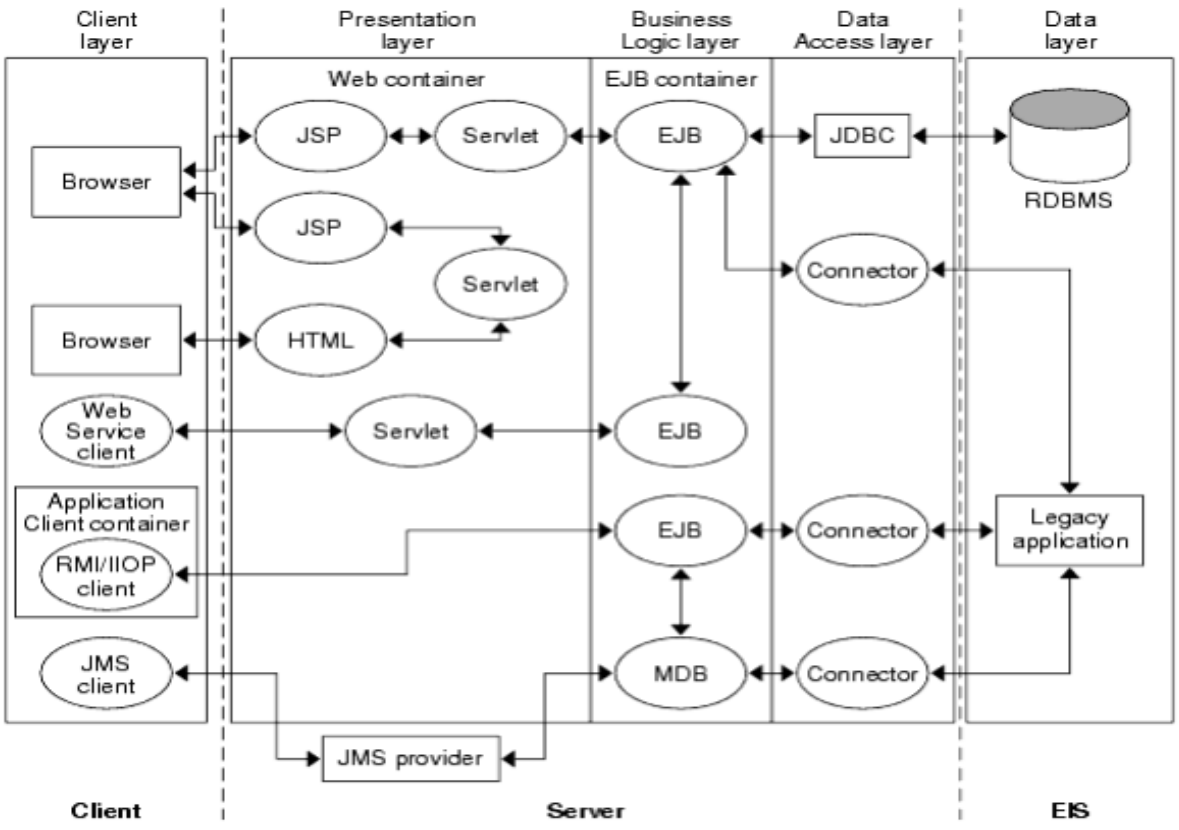
The Essence of Layered Architecture

Architecture is kind of an overloaded term, so we should probably dig deeper into what the term really means in the context of layers. The main idea behind Layered Architecture is a separation of concerns – as we said already, we want to avoid mixing domain or database code with the UI stuff, etc. The actual idea of separating a project into layers suggests that this separation of concerns should be achieved by source code organization. This means that apart from some guidance to what concerns we should separate, the Layered Architecture tells us nothing else about the design and implementation of the project. This implies that we should complement it with some other architectural processes, such as some upfront design, daily design sessions, or even full-blown Domain-Driven Design. Whichever option we choose doesn't matter, at least for the sake of layering, but we need to remember: **Layered Architecture gives us nothing apart from a guideline on how to organize the source code**.

Implementing Layered Architecture

Equipped with the knowledge of the layers to create, the relationships between them and the essence of the architecture, we are ready to implement it. As most of you probably expect, we will slice the system into layers by creating a separate package for each of them. When it comes to applying the dependency and separation rules, things are not so obvious. One could try putting each layer in a separate Maven module, but then capturing the weird relationship between domain and persistence would not be easy. I usually stick with packages and use common sense along with code reviews to make sure that none of the rules are broken.

Figure 1-1 J2EE application layers



Access Modifiers

The access modifiers can be attached to class, variables and to methods

<access_modifier> <data_type> variable;

- Public
  - Public int a; // This variable can be accessed by the class, the classes in the package/folder and other classes in the other package/folder
  -
- Private
  - private int b; //Highest secured. This variable b can have only the access to the class where it has declared
- Protected (Inheritance)
  - Protected int a; // this variable can be accessed by that class (acts as a private). But, it can be accessed in the sub class, child class.
- Default
  - int a ; //default This mean, this a variable can be accessed in that class and in that package

Note: Avoid default access modifiers

Package == Folder

Steps to create package

1) eclipse→ menu → file → new → package
2) When we create a class, there we need to define the package

Rules for creating packages

- Package name and directory name should be the same.
- A Java package may contain Sub-Packages or sub-directories with unique names.
- A Java package may contain any number of class files.
- The package declaration statement should be the very first statement in a Java file.
- The package import statement can come after the package declaration statement and before any Class declarations.
- The same package-name or class-name can be reused in some other packages but not within the same package.
- The directories corresponding to the Java packages are hierarchical in nature.
- To refer to an inner-package, use DOT or PERIOD symbol in conjunction with all the names of parent-packages. For example, use **pkga.pkgb.pkgc.pkgd**.
- Java does not define the maximum depth of inner packages. But the OS may limit this number.

- You can declare only one package name in a CLASS file. In other words, there can only be one package (declaration) statement.
- You can import any number of Packages or specific Package-classes into one java CLASS. In other words, there can be any number of import statements.
- You can import only one specific class of a given package by directly mentioning the package name and the class name.
- You can import all classes of a given package with a START (*) symbol next to the package name separated by a DOT or PERIOD symbol.

Declaring a Java Package

The keyword used to declare a package is "**package**". A package declaration statement should be the very first statement in a Java class file. The line number may be anyone like 1, 2, 3 or any. All the classes in the java file belong to the mentioned PACKAGE-NAME.

Syntax:

- package PACKAGE_NAME; //very first statement in a class file

Importing a Java Package

- The keyword used to import a Java package or Java class is "**import**". Java import statements can only come after the PACKAGE declaration statement if any.

Syntax:

```
package PACKAGE_NAME2; //It may be absent also

import PACKAGE_NAME1.*; //STAR imports all classes
import PACKAGE_NAME2.MyClass; //import a specific class
import PKG2.PKG3.PKG.*;

//before class declaration
class CLASSNAME
{ }
//
```

Coding standards

- Variables must be private and avoid default

Array

- Derived data type
- Declared using []
- Accessed by the index position of the data in the array
- There are two ways to create the arrays
  - 1st Way
    - <datatype> [] variableName = {data1,data2,data3,……..}
    - <datatype> variableName[] = {data1,data2,data3,……..}
  - 2nd Way
    - <dataType> [] variableName = new <dataType>[size]
    - Int [] numsArray = new int[5];