Java Fundamentals Day 2

- The methods that do not have the body and end with a semicolon are called as abstract methods.

- Abstract methods can only be inside abstract class.

- you cannot create an object of an abstract class. [Abstract classes cannot be inherited]

```
public abstract class Dictionary {

        public abstract void publishIndex();

        public abstract void authorInfo();

        public abstract void abstractInfo();

}
```

Dictionary d = new Dictionary(); -- CF Dictionary is abstract, cannot be instantiated(create an object).

- If a class extends an Abstract class, then it has to override all the abstract methods of that class.

```
Example: Dictionary

public abstract class Dictionary {    //super class : defines the template and tasks to be implemented
        public abstract void publishIndex();
        public abstract void authorInfo();
        public abstract void abstractInfo();
}
```

```
public class DictionaryE1 extends Dictionary{ //this sub class has to override the methods of super class which it does

        @Override
        public void publishIndex() {
                System.out.println("index published as of March '24 -- E1");
        }

        @Override
        public void authorInfo() {
                 System.out.println("Harry potter");

        }

        @Override
        public void abstractInfo() {
                 System.out.println("abstract info..");

        }
}
```

```
public class DictionaryE2 extends DictionaryE1{
//this class modifies the logic of publishIndex() method and retains other methods

        @Override
        public void publishIndex() {
                System.out.println("index published as of july '24 -- E2");
        }
}
```

```
IN App class
-------------
import com.service.Dictionary;
import com.service.DictionaryE1;
import com.service.DictionaryE2;

public class BookApp {
        public static void main(String[] args) {
                Dictionary dictionary = new DictionaryE2(); //polymorphic object
                dictionary.authorInfo();
                dictionary.abstractInfo();
                dictionary.publishIndex();

        }
}
```

Example 2: EmployeeDao

```
Case study:
You have an employee table in the DB. you want to perform certain ops on this DB.

Make a template and list out all the tasks/ops that are supposed to vbe performed on this table.
```

```
public abstract class EmployeeDao { //template class : define the tasks

        public abstract void insertEmployee(String name, String address)
        public abstract void deleteEmployee(int id);

}
```

```
public class EmployeeDaoImpl extends EmployeeDao{

        @Override
        public void insertEmployee(String name, String address) {
                // insert logic goes here...

        }

        @Override
        public void deleteEmployee(int id) {
                // delete by id goes here..

        }
}
```

```
EmployeeDao
    |
EmployeeDaoImpl


Note: We can create Non-Abstract normal methods inside abstract class.
This allows programmers to pass on implemented methods to the sub-classes.
```

```
Interface
========
- I will allow only abstract methods. No implementation of methods.
- Variables in interface are final and static.
```

Lets convert Dictionary abstract class from above example to Dictionary interface

```
public interface DictionaryInterface {

        public void publishIndex(); //by default, all methods in interface are abstract
        public void authorInfo();
        public void abstractInfo();
}
```

Rest of the code remains same, except that the class implements the interface instead of extends.

```
public class DictionaryE1 implements DictionaryInterface{ //class must implement the interface

        @Override
        public void publishIndex() {
                System.out.println("index published as of March '24 -- E1");
        }

        @Override
        public void abstractInfo() {
                 System.out.println("abstract info..");

        }

        @Override
        public void authorInfo() {
                // TODO Auto-generated method stub

        }
}
```

```
final & static keywords
=======================
Final Variable
===============
- if a variable is marked as final, we must initialize it as JVM won't do it for us
- if a variable is marked as final, we cannot assign any value to it. We must use only initialized value.

example:

class A{
        final int x=8; //instance variable -- x=0

        void m1(int x) { //local variable
                this.x = x; //CF - this.x is final, cannot be reassigned a value
        }
}
```

Tip: Never mark instance variables as final. You can declare local variables (arguments) as final as shown below:

```
Note: final can be used in arguments if the developer wants to take precaution.

public String concatName(final String fname, final String lname) { //final used in arguments
                 String name = fname + " " + lname;
                return name;
        }
```

```
Final Methods
=============
- if a method is marked as final, it cannot be overrided.

Ex:-
public final void authorInfo() { System.out.println("Author info... "); }

Now this method cannot be overrided in the sub-class.
```

Tip: Do not mark any method as final unless explicitly mentioned.

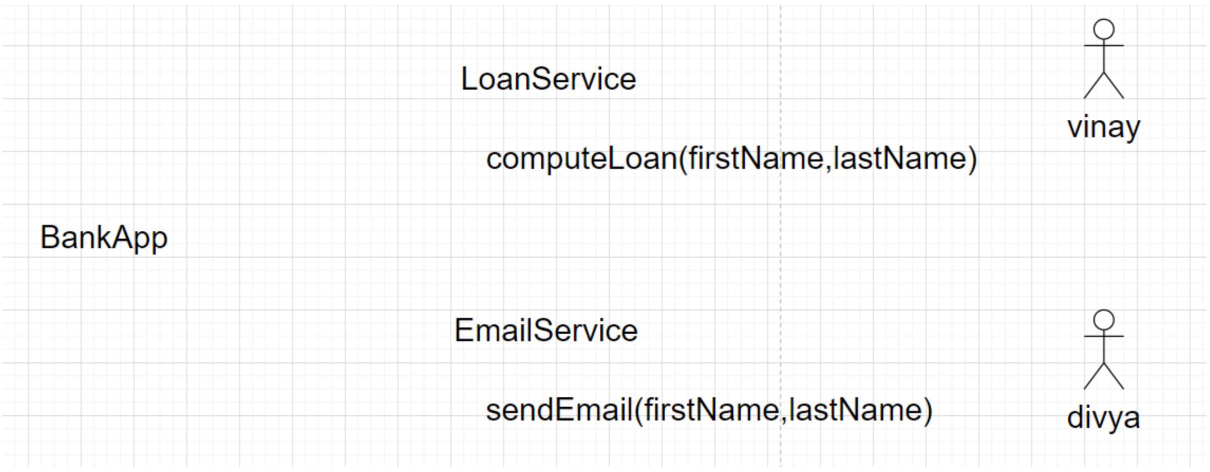
```
Final class
===========
- if a class is marked as final, it cannot be extended.

public final class DictionaryE1 extends Dictionary{ }

- Cannot extends DictionaryE1 class

public class DictionaryE2 extends DictionaryE1 {} -- CF

DictionaryE2 cannot extend DictionaryE1 as it is final.
```

Never mark a class as final unless explicitly mentioned in the JIRA.

```
Static Keyword
--------------
static method
=============

- static methods can be accessed from main class in 3 possible ways.

a. using object
b. using class name
c. directly.

from the above 3 ways, we must prefer accessing static methods using class name.

this offers convinience which calling the method and we don't have to create an object as well for method call.
```

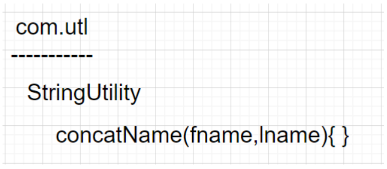
```
Ex: suppose in our application, we need to create a method that concats firstName and lastName of the user. This method can
be usefull in entire application to many programmers in their service class.

Solution is:
```

```
Create a Util class say StringUtil, and create a static method say.. concatName(String fname,String lname).

All Services class than want to perform this concat operation can then use this static method without creating the object.

Benifits:
---------
a. the code logic has to be written only once in single pace.
b. Any changes in the logic does not effect service class as it can be changed in Util class.
c. Common code can be centralized.
```


LoanService
computeLoan(firstName,lastName)
vinay
BankApp
EmailService
sendEmail(firstName,lastName)
divya

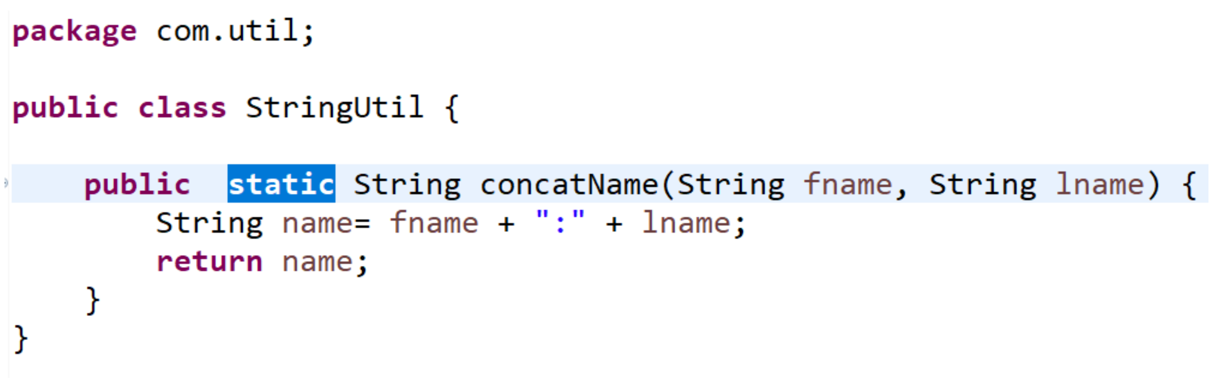
In the above scenario, Vinay needs to implement computeLoan method and divya needs to implement sendEmail method.
They both need to concat firstName and lastName as part of their logic.

So instead of both implementing the same logic, we can create a central Util class, implement the logic there, and sak both vinay and divya to use that.

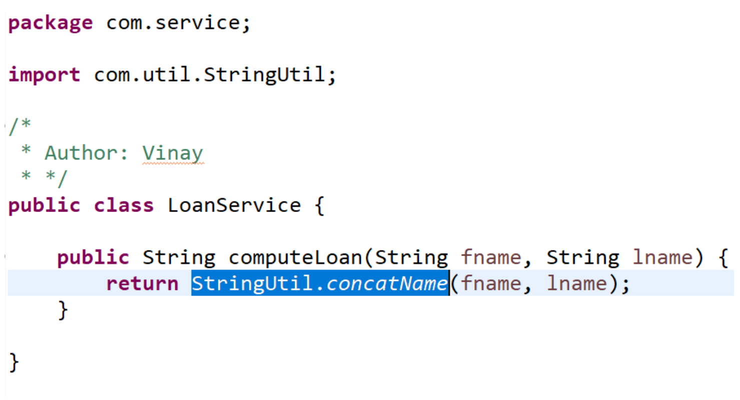


Implementation:

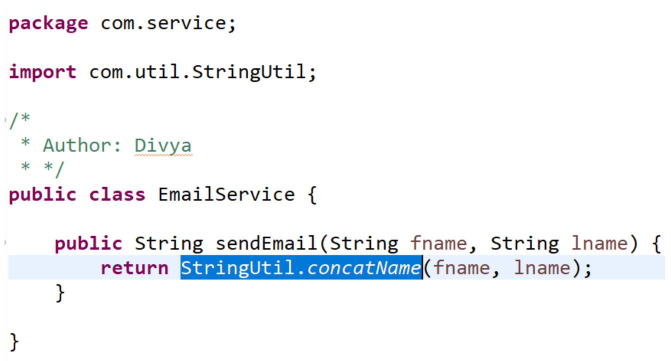
```java
package com.controller;

import com.service.EmailService;

public class BankApp {
    public static void main(String[] args) {
        LoanService loanService = new LoanService();
        String fname="";
        String lname= "";
        String name = loanService.computeLoan(fname,lname);
        System.out.println("Loan processed for " + name);

        EmailService emailService = new EmailService();
        name = emailService.sendEmail(fname,lname);
        System.out.println("Email sent to " + name);
    }
}
```



```
package com.util;

public class StringUtil {

    public  static String concatName(String fname, String lname) {
        String name= fname + ":" + lname;
        return name;
    }
}
```



```
package com.service;

import com.util.StringUtil;

/*
 * Author: Vinay
 * */
public class LoanService {

    public String computeLoan(String fname, String lname) {
        return StringUtil.concatName(fname, lname);
    }

}
```

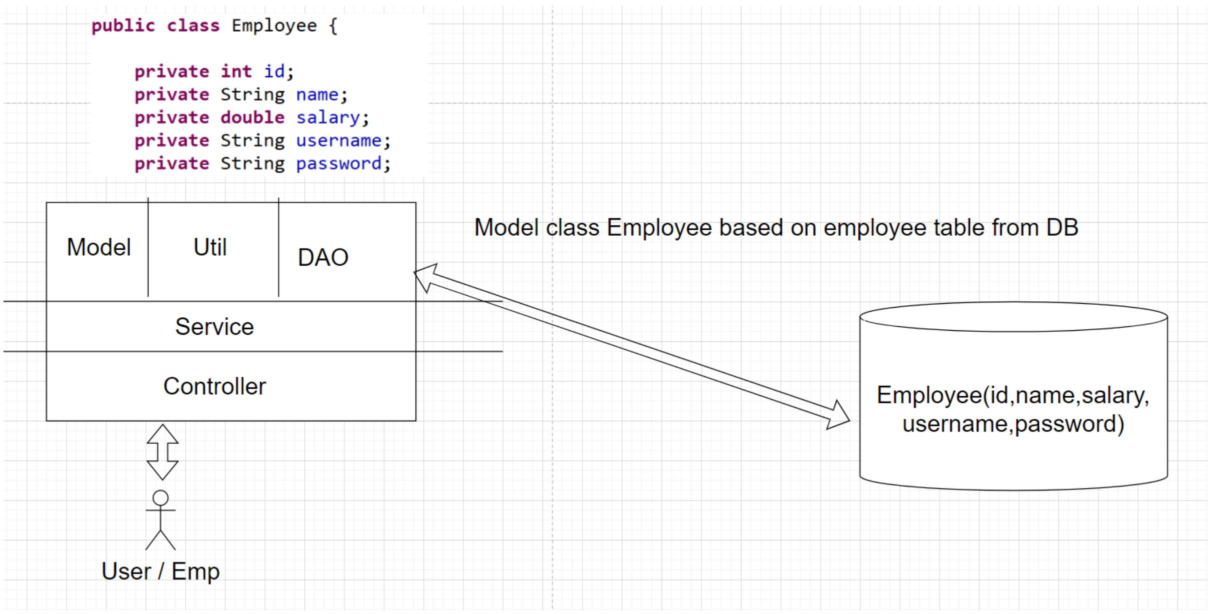


```
package com.service;

import com.util.StringUtil;

/*
 * Author: Divya
 * */
public class EmailService {

    public String sendEmail(String fname, String lname) {
        return StringUtil.concatName(fname, lname);
    }

}
```

So as you can see from above example, both vinay and divya are reaching out to StringUtil class and calling static method using class name to fetch the logic of string concat.

This prevents multiple implementation of logic. This is the primary use of static

Mapping DB with Java App



```java
public class Employee {

    private int id;
    private String name;
    private double salary;
    private String username;
    private String password;
```

Model
Util
DAO
Service
Controller
Model class Employee based on employee table from DB
Employee(id,name,salary, username,password)
User / Emp

For every DB table, we must create a corresponding matching Java model class as shown above.

Constructors & Encapsulation

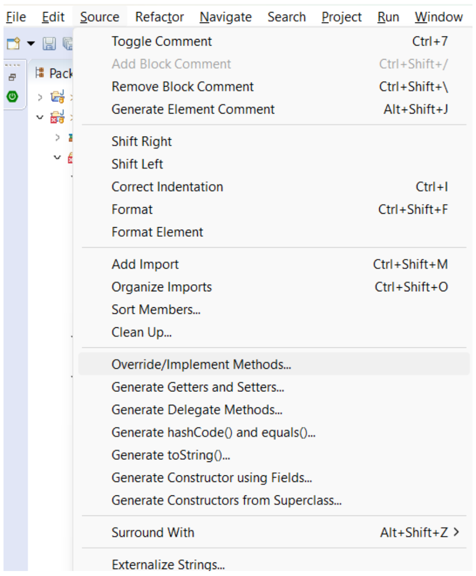
```
Encapsulation means hiding your instance variables from direct access.
- Mark your instance variables as private
- to make them accessible create getters and setters.
- To allow the programmers to attach values to the fields, create custructors.
- You can also override toString() method, so that you get meaningful info instead of memory location.
```

```
Constructor
------------
- Name of the Constructor is similar to that of the class.
- Every class has a default No-Arg Constructor
- Constructor gets called when the object of the class is created.
- COnstructors are used to initialize instance variables.
- We can create multiple constructors in the class.
```

Employee class using Encapsulation Rules

```java
public class Employee {

    private int id;
    private String name;
    private double salary;
    private String username;
    private String password;

    //add relevant constructors depending upon object signatures
    //add getters and setters
    //override toString method
```

You can auto-generate this as shown below in eclipse:


File
Edit
Source
Refactor
Navigate
Search
Project
Run
Window
Pack
Toggle Comment Ctrl+7
Add Block Comment Ctrl+Shift+/
Remove Block Comment Ctrl+Shift+\
Generate Element Comment Alt+Shift+J
Shift Right
Shift Left
Correct Indentation Ctrl+I
Format Ctrl+Shift+F
Format Element
Add Import Ctrl+Shift+M
Organize Imports Ctrl+Shift+O
Sort Members...
Clean Up...
Override/Implement Methods...
Generate Getters and Setters...
Generate Delegate Methods...
Generate hashCode() and equals()...
Generate toString()...
Generate Constructor using Fields...
Generate Constructors from Superclass...
Surround With Alt+Shift+Z
Externalize Strings...

Example of Constructor being called as per Object Signature:

Case 1:

```
Employee e1 = new Employee(1,"harry", 34000, "harry@gmail.com", "1234");
```

```
public Employee(int id, String name, double salary, String username, String password) {
    this.id = id;
    this.name = name;
    this.salary = salary;
    this.username = username;
    this.password = password;
}
```

Case 2:

```
Employee e2 = new Employee();

e2.setName("ronald");
e2.setSalary(40000);
e2.setUsername("ron@gmail.com");
e2.setPassword("123");
```

```
public Employee() { }
```

Case 3:

```
Employee e3 = new Employee("hermione", 54000, "her@gmail.com", "12345");
```

Note: no id given

```
public Employee(String name, double salary, String username, String password) {
    this.name = name;
    this.salary = salary;
    this.username = username;
    this.password = password;
}
```

Use of Constructor

==================

Constructors are used to initialize instance variables.