**Add Static Factory for this Code 🡪**

**public interface** IPasswordHasher **{**

public String hash**();** **}**

**class SHA256** **implements** IPasswordHasher **{**

private String \_password**;**

SHA256**(**String password**)** **{** \_password **=** password**;** **}**

public String hash**()** **{**

String salt **=** // create salt

**return** getHashedPassword**(**\_password**,** salt**);}**

public String getHashedPassword**(**String password**,** String salt**)**

**{** // SHA-256 Hashing Algorithm **}** **}**

**class** MD5 **implements** IPasswordHasher **{**

private String \_password**;**

MD5**(**String password**)** **{** \_password **=** password**;** **}**

public String hash**()** **{**

String salt **=** // create salt

**return** getHashedPassword**(**\_password**,** salt**);** **}**

public String getHashedPassword**(**String password**,** String salt**)**

**{** // SHA-256 Hashing Algorithm **}** **}**

**Static Factory**

**public interface** IPasswordHasher {

public String hash(); }

**class** SHA256 implements IPasswordHasher {

private String \_password;

SHA256(String password) { \_password = password; }

public String hash() {

String salt = // create salt

return getHashedPassword(\_password, salt);}

public String getHashedPassword(String password, String salt)

{ // SHA-256 Hashing Algorithm } }

**class** MD5 implements IPasswordHasher {

private String \_password;

MD5(String password) { \_password = password; }

public String hash() {

String salt = // create salt

return getHashedPassword(\_password, salt); }

public String getHashedPassword(String password, String salt)

{ // SHA-256 Hashing Algorithm } }

**public class** PasswordHasherFactory {

public static IPasswordHasher createMD5(String password){

return new MD5(password);}

public static IPasswordHasher createSHA256(String password){

return new SHA256(password);}}

**What does this code output Notes**

-Constructors will AUTOMATICALLY call the parent's 0 argument constructor. This continues all the way up the object hierarchy.

-If the method being called is STATIC, it will call the method for the DECLARED type (the type of the pointer). E.g. A a = new B(), A is the DECLARED type, and B is the ACTUAL type.

-NON-STATIC methods will be called based on the ACTUAL type. Period.

-SUPER method calls (e.g. super.someMethod()) rely upon the CURRENT CONTEXT. So if you see a super call inside of B, it will be calling a method inside of A, regardless of the actual type or declared type of the object.

**RefactorToStrategy**

public interface IPasswordHasher {

public String hash(); }

**class** PasswordHasher implements IPasswordHasher {

private String \_password;

private IHashingAlgorithm \_hashingAlgorithm;

PasswordHasher(String password, IHashingAlgorithm hashingAlgorithm) { \_password = password;

\_hashingAlgorithm = hashingAlgorithm; }

public String hash() {

String salt = // create salt

return \_hashingAlgorithm.getHash(password, salt;) }}

**public interface** IHashingAlgorithm{

String getHash(String password, String salt);}

**public** class MD5HashingAlgorithm implements IHashingAlgorithm{

public String getHash(String password, String salt){

// MD5 Hashing Algorithm}}

**public** class SHA256HashingAlgorithm implements IHashingAlgorithm {

public String getHash(String password, String salt){

// SHA-256 Hashing Algorithm }}

**public** class PasswordHasherFactory {

public static IPasswordHasher createMD5(String password){

return new PasswordHasher(password, new MD5HashingAlgorithm());}

public static IPasswordHasher createSHA256(String password){

return new PasswordHasher(password, new SHA256HashingAlgorithm());}}

**RefactorToTemplateMethod**

**public** interface IPasswordHasher {

public String hash(); }

**abstract** class PasswordHasher implements IPasswordHasher {

private String \_password;

PasswordHasher(String password) { \_password = password; }

public String hash() {

String salt = // create salt

return getHashedPassword(\_password, salt); }

public abstract String getHashedPassword(String password, String salt);}

**class** SHA256 extends PasswordHasher {

public String getHashedPassword(String password, String salt){ // SHA-256 Hashing Algorithm } }

**class** MD5 extends PasswordHasher {

public String getHashedPassword(String password, String salt){ // MD5 Hashing Algorithm } }

**public class** PasswordHasherFactory {

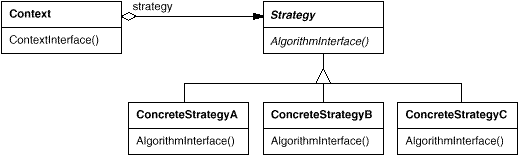
public static IPasswordHasher createMD5(String password){

return new MD5(password);}

public static IPasswordHasher createSHA256(String password){

return new SHA256(password);}}

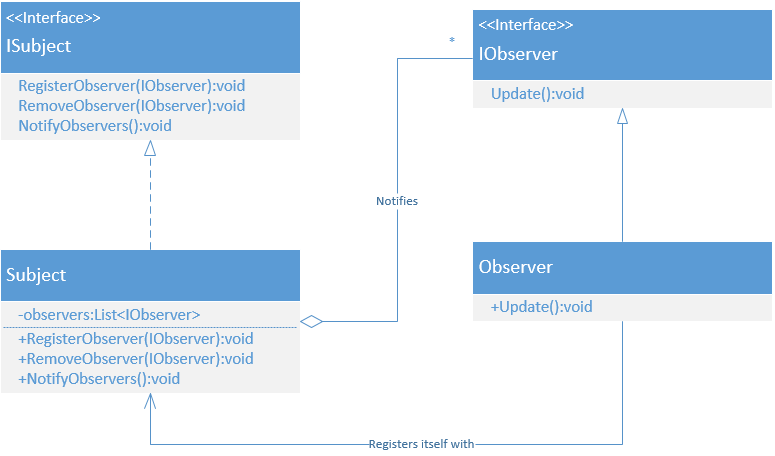
**Strategy Pattern(Behavioral)-** Put part of a method’s functionality in its own separate class

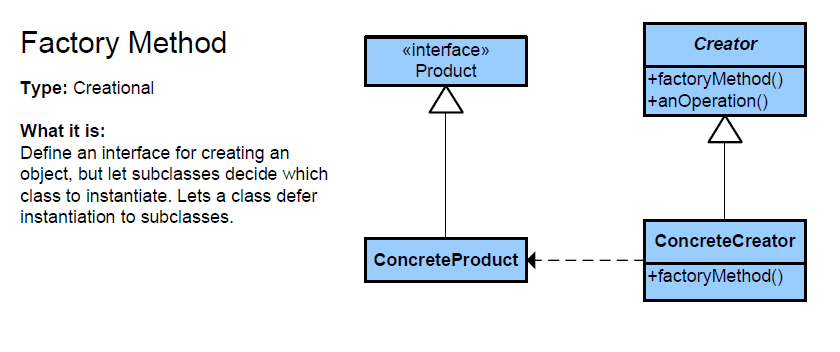


**Observer(behavioral)-** A subscribe/publish pattern

One or more objects subscribe to an object to be automatically notified when some event occurs

One common scenario is in a GUI application, multiple objects might subscribe to a button to be notified when the button is clicked





**Draw Class Diagram Example**

interface ISithLord **{** **}**

interface ILightSaber **{** **}**

interface ISithPower **{** **}**

interface IJedi **{**

void FightSithLord**(**ISithLord sithLord**);** **}**

class Jedi **implements** IJedi **{**

ILightSaber \_lightSaber**;** // lightsaber held exclusively **}** *Composition*

class SithLord **extends** Jedi**,** ISithLord **{**

List**<**ISithPower**>** powers**;** //List held exclusively, but not the individual powers. *Aggregation*

public void addPower**(**ISithPower power**)** **{** powers**.**add**(**power**);** **}**

public void removePower**(**ISithPower power**){** powers**.**remove**(**power**);}** **}**

**Observable Implementation Example**

import java.util.Observable;

public class ObservableDemo extends Observable

private String weather;

public ObservableDemo(String weather) {

this.weather = weather; }

public String getWeather() {

return weather;}

public void setWeather(String weather){

this.weather = weather;

setChanged();

notifyObservers();}}

import java.util.Observable;

import java.util.Observer;

public class ObserverExample implements Observer{

private ObservableDemo weatherUpdate ;

@Override

public void update(Observable observable, Object arg) {

weatherUpdate = (ObservableDemo) observable;

System.out.println("Weather Report Live. Its "+weatherUpdate.getWeather());}

public static void main(String[] args){

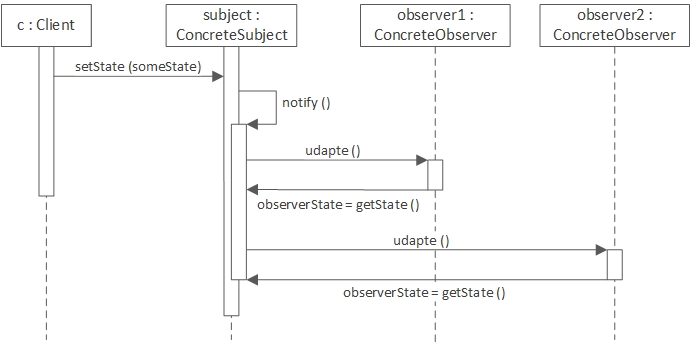
ObservableDemo observable = new ObservableDemo(null);

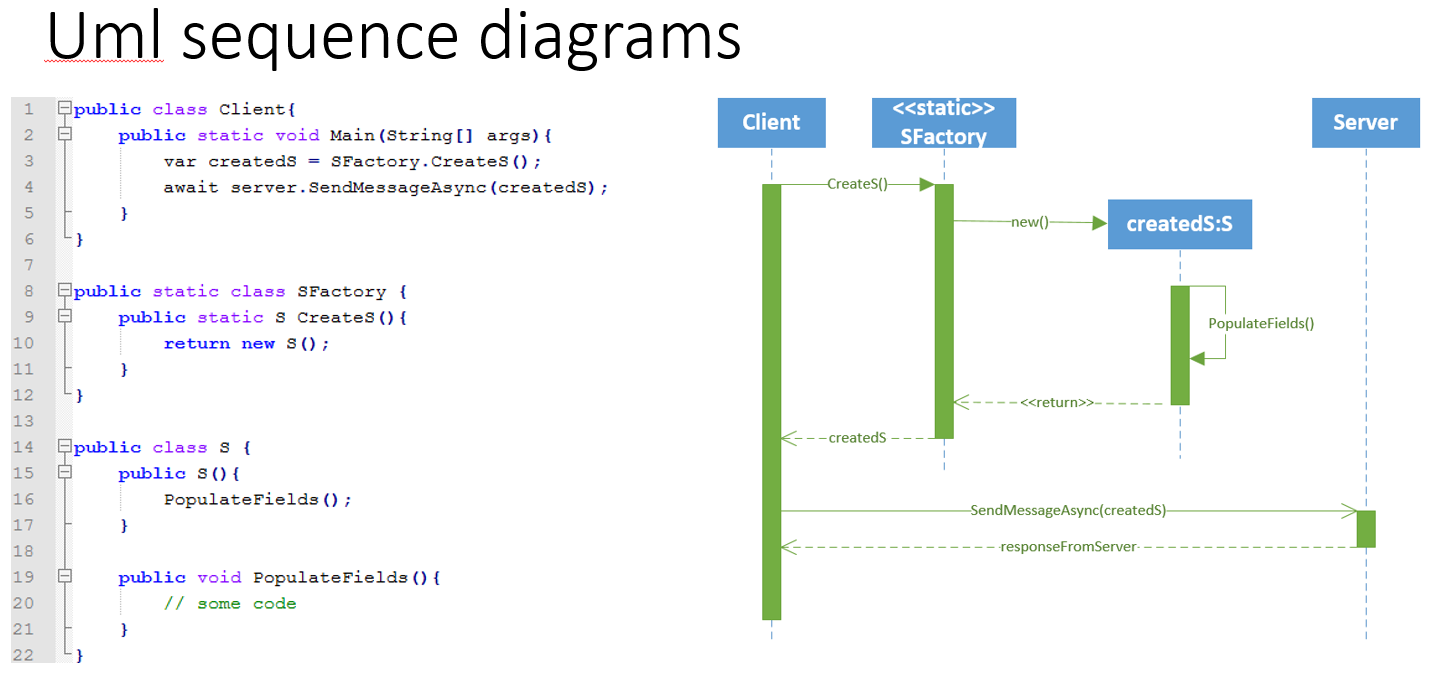
ObserverExample observer = new ObserverExample();

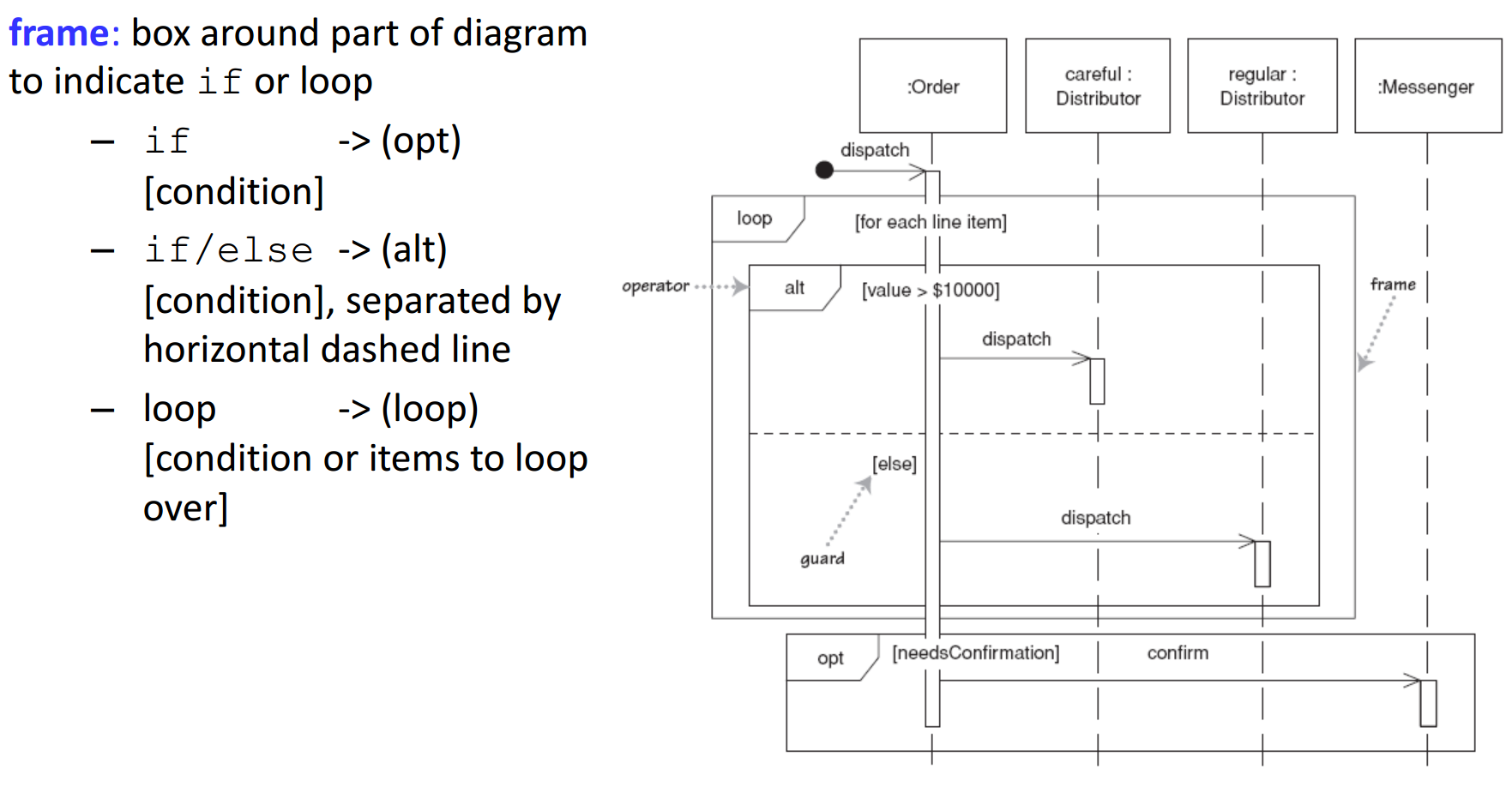
observable.addObserver(observer);

observable.setWeather("Bright and sunny...Let's play cricket!! ");

observable.setWeather("Raining Heavily!..Let's have hot Pakodas!!"); }}







**Template method(behavioral)-** Put part of a method’ s functionality in a own subclass

