THE SINGLETON PATTERN

“The reasoning behind the Singleton pattern is to provide a class that will have one and only one instance that will have a global point of access”

Singleton Theory!

So we do not want to allow other programmers to create instances of our Singleton class

```
1  package com.J1ggy;
2
3  public class SimpleSingleton {
4    //Fields - Note they are private and our instance of "SimpleSingleton" is static
5    private static SimpleSingleton instance;
6    private String someData;
7
8    //Private Constructor - only accessible by the class
9    private SimpleSingleton(String someData) {
10     this.someData = someData;
11   }
12
13   //A Public method that calls the Private Constructor only if an instance does not exist
14   public static SimpleSingleton getInstance(String someData) {
15     if(instance == null) {
16       instance = new SimpleSingleton(someData);
17     }
18     return instance;
19   }
20   //Public method to print out someData
21   public void printSomeData() {
22     System.out.println("NON THREAD SAFE: Message from SimpleSingleton: " + someData);
23   }
24 }
25
```

Here we see a simple implementation – The key is that the class “SimpleSingleton” will hold a private static instance of it’s self and the Constructor is also private. The only pubic way to create an instance is by using the getInstance() method which in turn calls the constructor.
The getinstance() method will only create an instance if no other instances exist.

This guarantees that there will only ever be one instance.

Remember -
private – Is only accessible within the declared class.
static – Can be accessed without creating an instance.

```java
package com.J1ggy;

public class SingletonClass {

// Fields
  private static volatile SingletonClass instance; // volatile indicates that a thread may not have finished instantiating
  private int someDataField;
//-----------------------------------------------------------------------------------------------------------

//Private Constructor - Can only be called by instances of the  Class SingletonClass
  private SingletonClass(int someData) {
    this.someDataField = someData;
  }
//-----------------------------------------------------------------------------------------------------------
//Public method to call Private Constructor
  public static SingletonClass getInstance(int someData) {
    SingletonClass theInstance = instance;   //get the existing instance if there is one
    if (theInstance == null) {
      synchronized(SingletonClass.class ){ //synchronised creates a lock allowing only one thread  to execute
        theInstance = instance;  // check if a thread has created an instance meantime
        if(theInstance == null) {//If still none
          instance = theInstance = new SingletonClass(someData); // Statically call private Constructor
        }
      }
    }
    System.out.println("THREAD SAFE: Message from SingletonClass I have successfully been instantiated! ");
  return theInstance;  //return the single instance
}
  //-----------------------------------------------------------------------------------------------------------

  //Public  method that accesses the data field
  public void getSomeDataField() {
    System.out.println("someDataField is: " + someDataField);
  }
  //-----------------------------------------------------------------------------------------------------------
}//End of Class

//Note: "theInstance is a local variable used to hold the single instance of SingletonClass that is created"
```

A problem arises in trying to use a Singleton Class in a multi-thread application. This is because two threads might try to instantiate a new instance at approximately the same time. Here we declare the private static field for the instance as also being volatile. (Potentially being written too by more than one thread)

We still follow the same idea of using a private Constructor which is called by a public getInstance() method.
We still check if instance is null
This time we use synchronized(Singleton.class) lock the class from access by other threads
Then we recheck that no instance has been created in the mean time
Finally we create and return the instance.

THE OBSERVER PATTERN

The reasoning behind the Observer Pattern is the creation of a "One to Many Dependancy" between objects where if an Object changes state the dependants of that Object will be of the state change and can perform some actions based on that change.

How would you like me to get in touch Sir?
We can do an e-mail message, an SMS text message directly to your phone - Sir or I could nip round to your office if you don't mind the intrusion Sir?
How is Lady Jean and the Grand Children if you don't mind me asking Sir?
Did I forget to salute Sir – Dreadfully Sorry!!!

More seriously – If an item is out of stock we want our Store system to automatically check the users preferences regarding contacting them an if we have their permission automatically generate an e-mail or sms message to inform them that the item they would like is back in stock.

So how do we accomplish this?
Lets consider the Dependant Classes
We will need one which listens for an event that will trigger an e-mail

```
1
2  public class EmailNotificationListener {
3    //Fields----------------------------------------------------------------------------------------------------
4    private String email;
5    private Customer cust;
6   //Constructor-----------------------------------------------------------------------------------------------
7      public EmailNotificationListener(String email, Customer cust) { // Stores the email and Customer instance that relate to this particular listener
8      this.email = email;
9      this.cust = cust;
10   }
11   // Methods--------------------------------------------------------------------------------------------------
12   public Customer getCust() {   // Lets us check who is listening
13     return cust;
14   }
15   public void update(Part part) { // Simulates sending an email to particular customer to say the stock has been replenished (Takes in the part that was out of stock)
16     if(cust.isEmlSubscriber()) {
17         System.out.println("E-Mail To: " + email + " Message: Hi " + cust.getForename() + " Just to let you know we now have " + part.getInStockQty() + " " + part.getPartName() + "'s in Stock\n")
18     }
19   }
20 }
21
```

...and one which listens for an event that will trigger an SMS message

```
1
2  public class SMSNotificationListener {
3    //Fields----------------------------------------------------------------------------------------------------
4    private String sms;
5      private Customer cust;
6  //Constructor-----------------------------------------------------------------------------------------------
7        public SMSNotificationListener(String sms, Customer cust) { //// Stores the mobile No and Customer instance that relate to this particular listener
8          this.sms = sms;
9          this.cust =  cust;
10       }
11   //Methods---------------------------------------------------------------------------------------------------
12     public Customer getCust() {    // Lets us check who is listening
13         return cust;
14       }
15
16     public void update(Part part) { // Simulates sending an SMS to particular customer to say the stock has been replenished (Takes in the part that was out of stock)
17       if(cust.isSmsSubscriber()) {
18       System.out.println("SMS To: " + sms + " Message: Hi " +  cust.getForename() + " Just to let you know we now have " + part.getInStockQty() + " " + part.getPartName() + "'s  in Stock\n");
19       }
20     }
21   }
```

The two classes in our exercise are relatively straight forward.
They each have two fields.

The first Class “EmailNotificationListener” has “email” and “Customer” fields.
The second Class “SMSNotificationListener” has “mobile” and “Customer” fields.

The Constructor initialises these fields – If you consider this each instance of a Listener is therefore very specific to one customer. (and we have a getCust() method if we need to know who that is)

We then have an update() method that takes in an “previously out of stock” Part and sends an e-mail or sms message to update the customer that we now have stock.
(For our purposes each is simulated as a simple syso).

Read through through the code above until you are reasonably happy with this step – (Not worrying too much about the syso just now)

So these Listener instances are particular to a customer and just sit an wait for an update(Part) to arrive at which point they contact the customer associated with them.

So EmailNotificationListener and SMSNotificationListener must be the dependants!

Okay so who does the “Notifying” for these Dependants?

```java
import java.util.ArrayList;

public class MailService {              // Define the class
  //Fields-------------------------------------------------------------------
  List<EmailNotificationListener>emlsubscribers ;          // A List to hold the EmailNotificationListeners  --- Who wants an email when stock arrives
  List<SMSNotificationListener>smssubscribers ;            // A List to hold the SMSNotificationListeners  --- Who wants an sms when stock arrives
                                                           // Note it is the Listeners containing the customer request not the customer that is kept in these lists

  //Constructor--------------------------------------------------------------
  MailService() {
    emlsubscribers = new ArrayList<>();                    // Creates and assigns two ArrayLists for email and sms subscriptions respectfully
    smssubscribers = new ArrayList<>();
  }

  public void emlsubscribe( EmailNotificationListener listener) {      // Add an email subscriber
    emlsubscribers.add(listener);
  }
  public void emlunSubscribe( EmailNotificationListener listener) {   // remove an email subscriber
    emlsubscribers.remove(listener);
  }
  public void smssubscribe( SMSNotificationListener listener) {   // Add an SMS subscriber
    smssubscribers.add(listener);
  }
  public void smsunSubscribe( SMSNotificationListener listener) {  // remove an SMS subscriber
    smssubscribers.remove(listener);
  }
  public void notifyCustomers(Part part) {    // Iterates through each subscriber list and calls the update() method of the Listener (both email & sms lists) - passes in the out of stock part

    for(EmailNotificationListener eml : emlsubscribers) { if(eml.getCust().isEmlSubscriber()) {eml.update(part);}}
    for(SMSNotificationListener sms : smssubscribers) { if(sms.getCust().isSmsSubscriber()) {sms.update(part);}}

  }
}
```

The MailService Class
The MailService Class has two List fields emailsubscribers and smssubscribers

These are of type EmailNotificationListener and SMSNotificationListener respectfully.

The Constructor just initialises each List field with an empty ArrayList of each type.

We then have subscribe()and unsubscribe() methods allowing us to add Listener instances for individual customers as per there preference as to (how/if at all) they wish to be contacted.

The Store will therefore be able to ask customers if they wish to be on the email subscribers List the SMS subscribers List or both!

Notice that the Customer does not have to specify an “out of stock” Part that they would like to be informed about to be on either mailing List. This decision is deferred.

This means that we could “easily” adjust the classes later to perhaps send out marketing information or offers. ie: The hard bit is done we have Listeners and a notifier.

Note: Cont’d on next page – I have repeated the same code screenshot for your convenience

Repeat of screenshot from previous page-

```
1  import java.util.ArrayList;
3
4  public class MailService {            // Define the class
5    //Fields----------------------------------------------------------------------
6    List<EmailNotificationListener>emlsubscribers ;        // A List to hold the EmailNotificationListeners  --- Who wants an email when stock arrives
7    List<SMSNotificationListener>smssubscribers ;          // A List to hold the SMSNotificationListeners  --- Who wants an sms when stock arrives
8                                                           // Note it is the Listeners containing the customer request not the customer that is kept in these lists
9
10   //Constructor-----------------------------------------------------------------
11   MailService() {
12     emlsubscribers = new ArrayList<>();                  // Creates and assigns two ArrayLists for email and sms subscriptions respectfully
13     smssubscribers = new ArrayList<>();
14   }
15
16   public void emlsubscribe( EmailNotificationListener listener) {      // Add an email subscriber
17     emlsubscribers.add(listener);
18   }
19   public void emlunSubscribe( EmailNotificationListener listener) {   // remove an email subscriber
20     emlsubscribers.remove(listener);
21   }
22   public void smssubscribe( SMSNotificationListener listener) {   // Add an SMS subscriber
23     smssubscribers.add(listener);
24   }
25   public void smsunSubscribe( SMSNotificationListener listener) {  // remove an SMS subscriber
26     smssubscribers.remove(listener);
27   }
28   public void notifyCustomers(Part part) {   // Iterates through each subscriber list and calls the update() method of the Listener (both email & sms lists) - passes in the out of stock part
29
30     for(EmailNotificationListener eml : emlsubscribers) { if(eml.getCust().isEmlSubscriber()) {eml.update(part);}}
31     for(SMSNotificationListener sms : smssubscribers) { if(sms.getCust().isSmsSubscriber()) {sms.update(part);}}
32
33   }
34 }
35
```

MailService Class Cont'd – The Important Bit!

Let's look at notifyCustomers(Part part) method

```
public void notifyCustomers(Part part) {   // Iterates through each subscriber list and calls the update() method of the Listener (both email & sms lists) - passes in the out of stock part

  for(EmailNotificationListener eml : emlsubscribers) { if(eml.getCust().isEmlSubscriber()) {eml.update(part);}}
  for(SMSNotificationListener sms : smssubscribers) { if(sms.getCust().isSmsSubscriber()) {sms.update(part);}}

}
```

This is the method that carries out the notifying of the Listeners.

Notice: It takes in a "Part" - This will be the "out of stock" Part "That has just been replenished" (We'll see how that works shortly)

```
for(EmailNotificationListener eml : emlsubscribers) { //First we Iterate through our email subscribers
        if(eml.getCust().isEmlSubscriber()) { // Check each Listener is definitely for an email subscriber
                eml.update(part); //NOW WE CALL THE LISTENERS update(part) and pass in part
        }
}
```

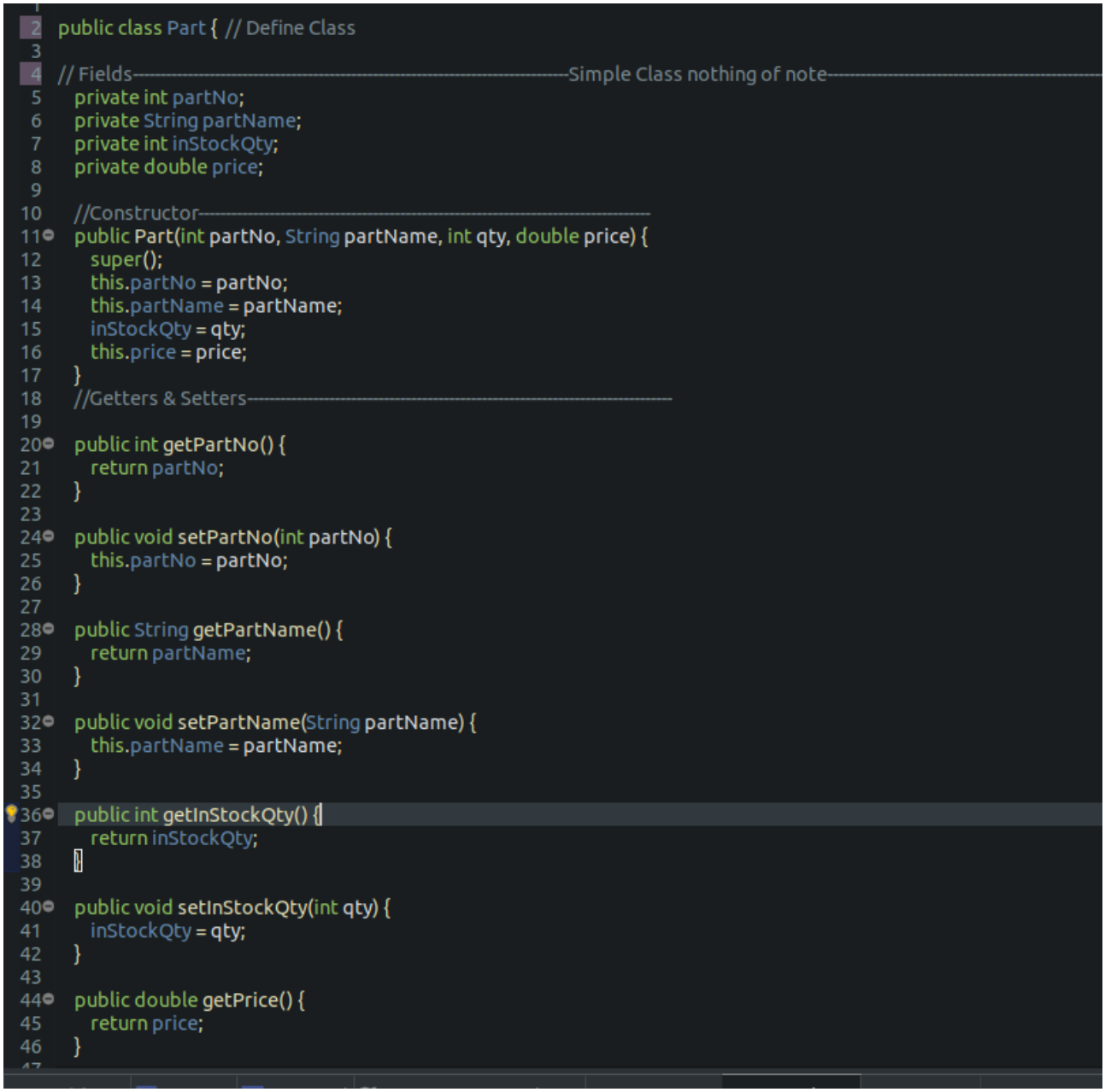
REPEAT BUT THIS TIME FOR THE SMS SUBSCRIBERS LIST

```
for(SMSNotificationListener sms : smssubscribers) { if(sms.getCust().isSmsSubscriber())
{sms.update(part);}}
```

Note: We pass in part if you remember so that the NotificationListeners can format a sensible string giving the name and quantity of the part that is now in stock.

The Part Class

Note: incomplete or partial screenshot



```
2  public class Part { // Define Class
3
4  // Fields----------Simple Class nothing of note----------
5    private int partNo;
6    private String partName;
7    private int inStockQty;
8    private double price;
9
10   //Constructor----------
11   public Part(int partNo, String partName, int qty, double price) {
12     super();
13     this.partNo = partNo;
14     this.partName = partName;
15     inStockQty = qty;
16     this.price = price;
17   }
18   //Getters & Setters----------
19
20   public int getPartNo() {
21     return partNo;
22   }
23
24   public void setPartNo(int partNo) {
25     this.partNo = partNo;
26   }
27
28   public String getPartName() {
29     return partName;
30   }
31
32   public void setPartName(String partName) {
33     this.partName = partName;
34   }
35
36   public int getInStockQty() {
37     return inStockQty;
38   }
39
40   public void setInStockQty(int qty) {
41     inStockQty = qty;
42   }
43
44   public double getPrice() {
45     return price;
46   }
```

The Part Class is a simple Class (Nothing of note)

It has fields to hold “partNo”, “partName”, “inStockQty” and “price”
… a Constructor to create and initialise these fields
...and Getter and Setter methods for each field.

The Customer Class

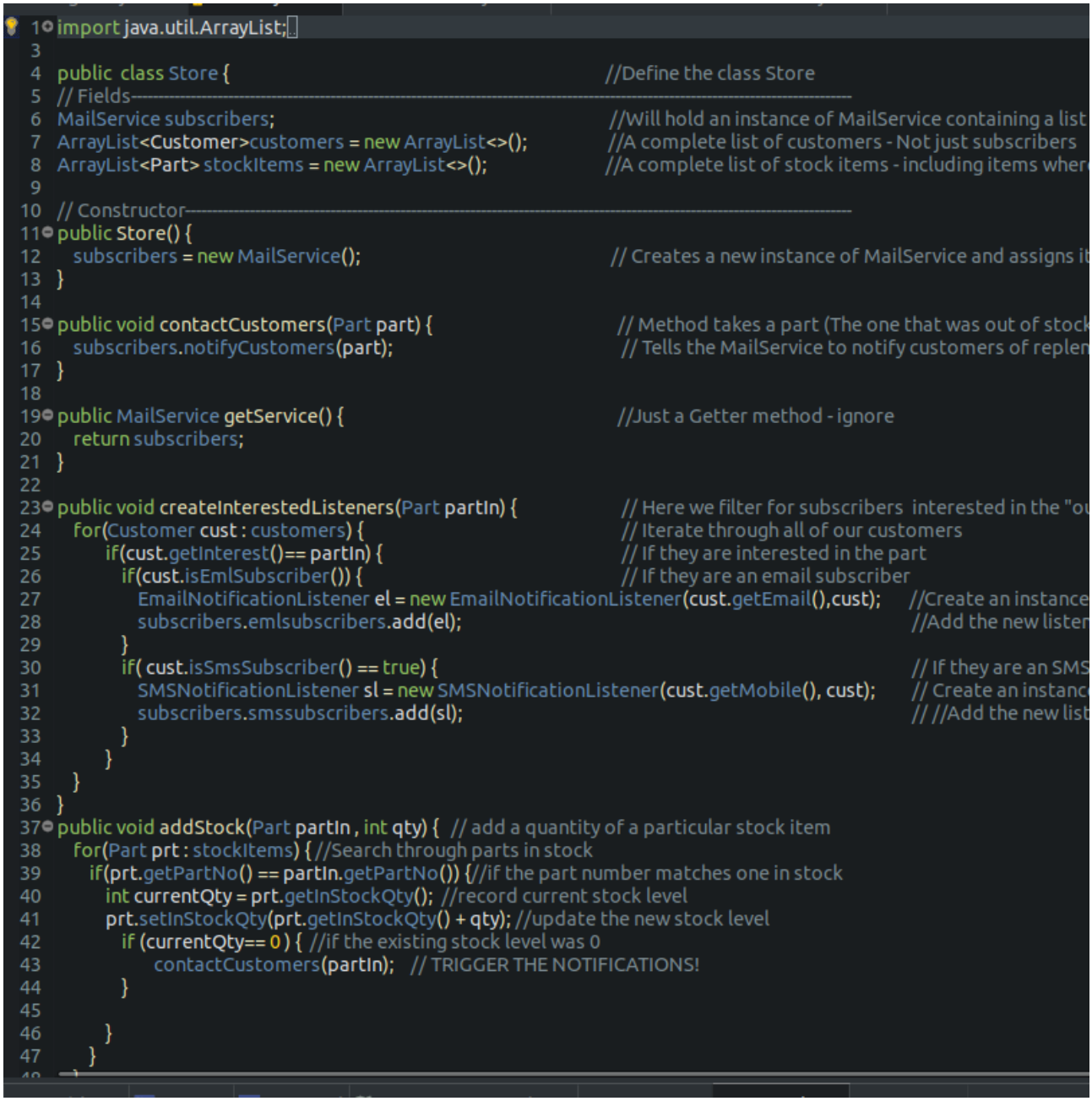
```
public class Customer {   //Class Def
  //Fields--------------------------------------------------------------------------------------------
  private String forename;
  private String surname;
  private String email;
  private String mobile;
  private Part interest;                          //Holds the part the customer has expressed an interest in (Could t
  private boolean isEmlSubscriber = false;        // Are they interested in receiving emails?
  private boolean isSmsSubscriber = false;        //Are they interested in receiving SMS messages?
  //Constructor---------------------------------------------------------------------------------------
  public Customer(String forename, String surname, String email, String mobile) { //Creates a basic customer witho
    super();
    this.forename = forename;
    this.surname = surname;
    this.email = email;
    this.mobile = mobile;
  }
//Methods ----------------------------------------------------------Just Getters and Setters-----------------------
  public boolean isEmlSubscriber() {
    return isEmlSubscriber;
  }
  public void setEmlSubscriber(boolean isEmlSubscriber) {
    this.isEmlSubscriber = isEmlSubscriber;
  }
  public boolean isSmsSubscriber() {
    return isSmsSubscriber;
  }
  public void setSmsSubscriber(boolean isSmsSubscriber) {
    this.isSmsSubscriber = isSmsSubscriber;
  }

  public void setInterest(Part part) {
    this.interest= part;
  }
  public Part getInterest(){
    return this.interest;
  }
  public String getForename() {
    return forename;
  }
  public void setForename(String forename) {
    this.forename = forename;
  }
  public String getSurname() {
    return surname;
```

The Customer Class is a similarly simple Class to hold the details.

The thing to note here:
We have three fields that are "NOT" initialised by the Constructor:
private Part interest; **// Holds an out of stock Part that the customer wishes to express interest in**
private boolean isEmlSubscriber = false;
private boolean isSmsSubscriber = false;

This is because these are changeable values that must be agreed with the customer.
By default we are not interested in an out of stock part and we are not on a mailing list.

The Store Class



```
1  import java.util.ArrayList;
3
4  public  class Store {                                          //Define the class Store
5  // Fields----------------------------------------------------------------------------------------------------
6  MailService subscribers;                                       //Will hold an instance of MailService containing a list
7  ArrayList<Customer>customers = new ArrayList<>();              //A complete list of customers - Not just subscribers
8  ArrayList<Part> stockItems = new ArrayList<>();                //A complete list of stock items - including items wher
9
10 // Constructor----------------------------------------------------------------------------------------------
11 public Store() {
12    subscribers = new MailService();                            // Creates a new instance of MailService and assigns it
13 }
14
15 public void contactCustomers(Part part) {                     // Method takes a part (The one that was out of stock
16    subscribers.notifyCustomers(part);                          // Tells the MailService to notify customers of replen
17 }
18
19 public MailService getService() {                             //Just a Getter method - ignore
20    return subscribers;
21 }
22
23 public void createInterestedListeners(Part partIn) {          // Here we filter for subscribers  interested in the "ou
24    for(Customer cust : customers) {                            // Iterate through all of our customers
25       if(cust.getInterest()== partIn) {                        // If they are interested in the part
26          if(cust.isEmlSubscriber()) {                          // If they are an email subscriber
27             EmailNotificationListener el = new EmailNotificationListener(cust.getEmail(),cust);    //Create an instance
28             subscribers.emlsubscribers.add(el);                                                     //Add the new lister
29          }
30          if( cust.isSmsSubscriber() == true) {                                                      // If they are an SMS
31             SMSNotificationListener sl = new SMSNotificationListener(cust.getMobile(), cust);      // Create an instanc
32             subscribers.smssubscribers.add(sl);                                                     // //Add the new list
33          }
34       }
35    }
36 }
37 public void addStock(Part partIn , int qty) {  // add a quantity of a particular stock item
38    for(Part prt : stockItems) {//Search through parts in stock
39       if(prt.getPartNo() == partIn.getPartNo()) {//if the part number matches one in stock
40          int currentQty = prt.getInStockQty();  //record current stock level
41          prt.setInStockQty(prt.getInStockQty() + qty); //update the new stock level
42          if (currentQty== 0 ) { //if the existing stock level was 0
43                contactCustomers(partIn);   // TRIGGER THE NOTIFICATIONS!
44          }
45
46       }
47    }
```

Field's

MailService – Will hold an instance of the MailService class giving us access to the
2 x ArrayLists containing our emlsubscribers and smssubscribers – These are confirmed subscribers giving their permission to be contacted

customers – An ArrayList that holds all customers (Suscribers & NonSubscribers)

stockItems – An ArrayList that holds all the parts that are stocked

Constructor

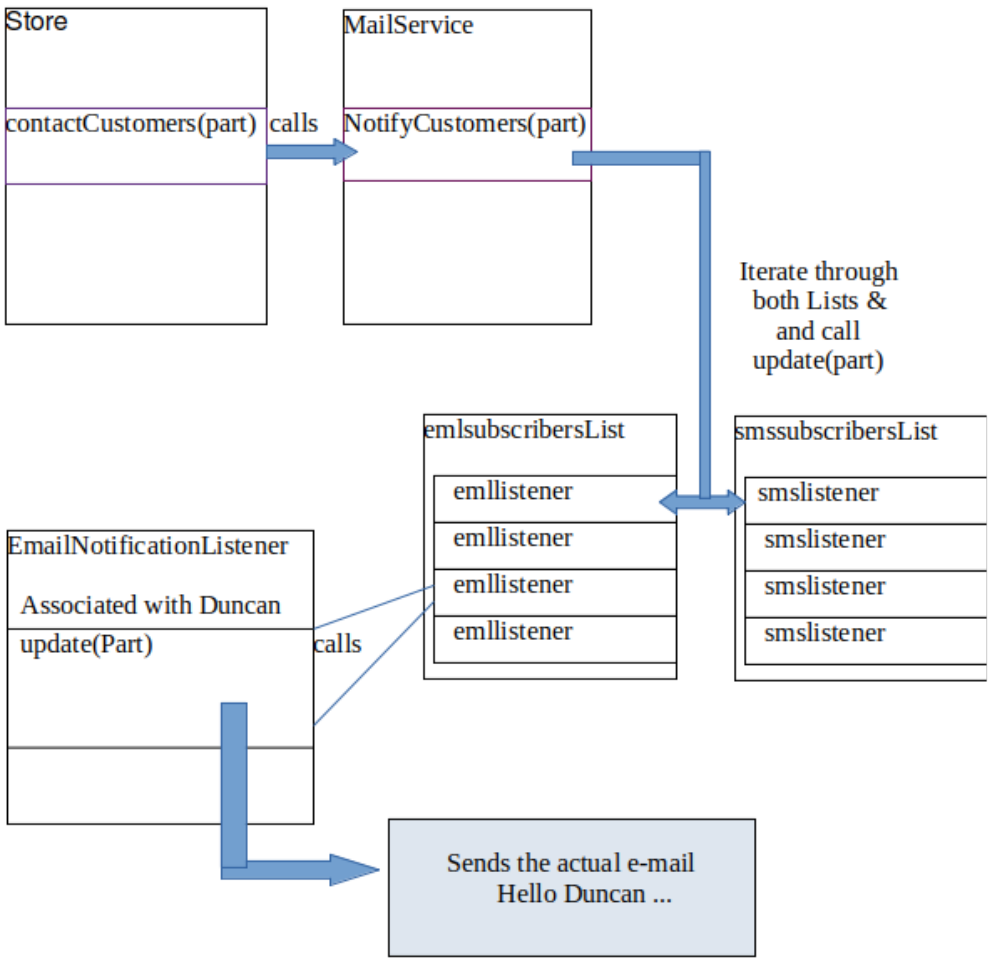
Creates the instance of MailService and assigns it to the field subscribers

Methods

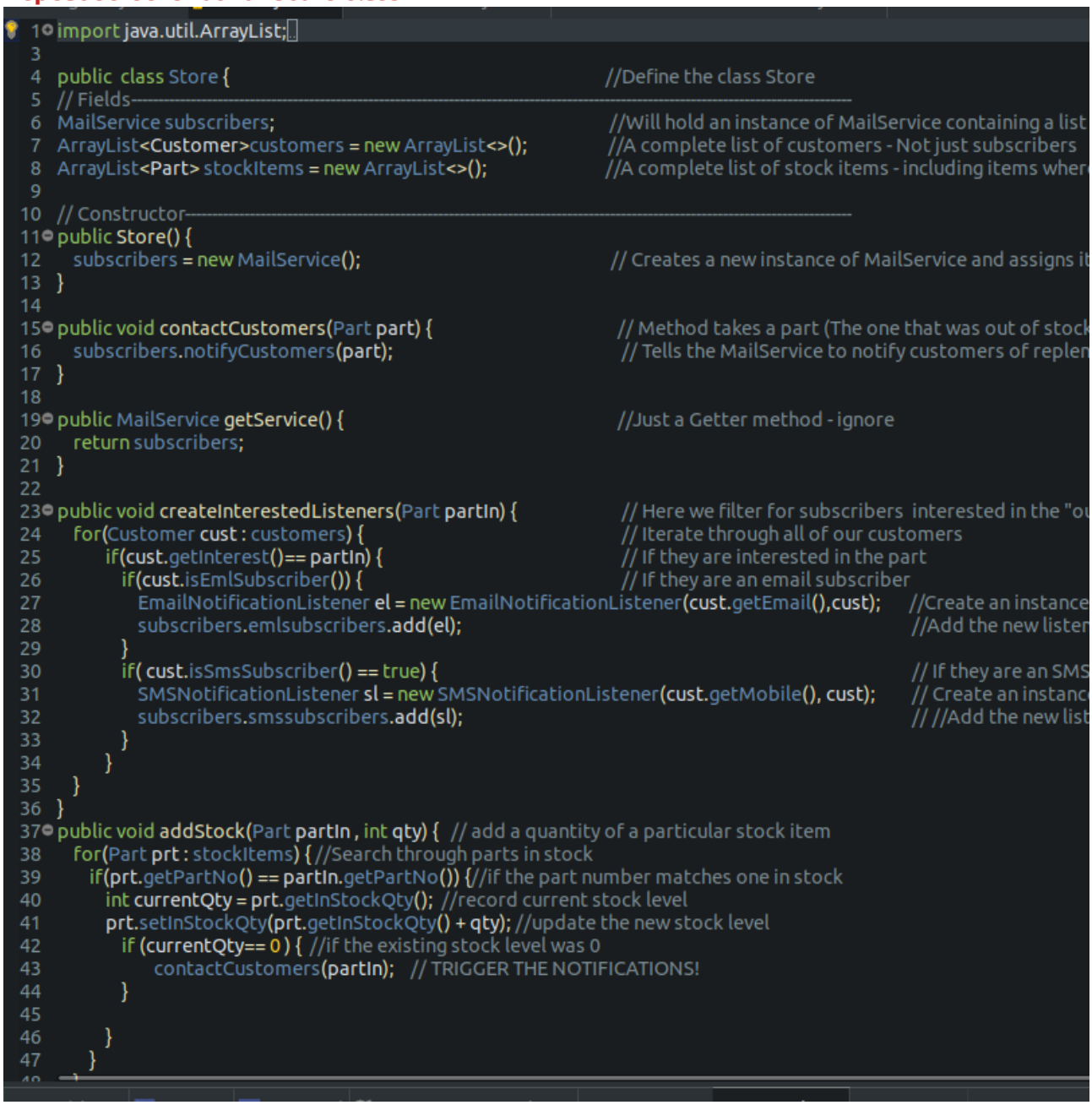
Important!

```
public void contactCustomers(Part part) {
        subscribers.notifyCustomers(part);

}
```

The method contactCustomers(Part part) seems like a simple method but is key to what we are doing here.
This method calls the MailService's "notifyCustomers(part)" method which if you remember, iterates through the two subscription Lists held by the MailService instance
(in this case subscriptions is the MailService instance whilst emailsubscribers & smssubscribers are the two Lists that contain the EmaiNotificationListeners & SMSNotificationListeners)
...and calls each Listeners update(part) method which in turn simulates the sending of either email or sms to those users who were interested in the part.



Repeat screenshot for Store class



```
1  import java.util.ArrayList;
3
4  public class Store {                                          //Define the class Store
5  // Fields----------
6  MailService subscribers;                                      //Will hold an instance of MailService containing a list
7  ArrayList<Customer>customers = new ArrayList<>();             //A complete list of customers - Not just subscribers
8  ArrayList<Part> stockItems = new ArrayList<>();               //A complete list of stock items - including items wher
9
10 // Constructor----------
11 public Store() {
12    subscribers = new MailService();                           // Creates a new instance of MailService and assigns it
13 }
14
15 public void contactCustomers(Part part) {                    // Method takes a part (The one that was out of stock
16    subscribers.notifyCustomers(part);                         // Tells the MailService to notify customers of replen
17 }
18
19 public MailService getService() {                            //Just a Getter method - ignore
20    return subscribers;
21 }
22
23 public void createInterestedListeners(Part partIn) {         // Here we filter for subscribers  interested in the "ou
24    for(Customer cust : customers) {                           // Iterate through all of our customers
25       if(cust.getInterest()== partIn) {                       // If they are interested in the part
26          if(cust.isEmlSubscriber()) {                         // If they are an email subscriber
27             EmailNotificationListener el = new EmailNotificationListener(cust.getEmail(),cust);    //Create an instance
28             subscribers.emlsubscribers.add(el);                                                     //Add the new lister
29          }
30          if( cust.isSmsSubscriber() == true) {                                                      // If they are an SMS
31             SMSNotificationListener sl = new SMSNotificationListener(cust.getMobile(), cust);      // Create an instanc
32             subscribers.smssubscribers.add(sl);                                                     // //Add the new list
33          }
34       }
35    }
36 }
37 public void addStock(Part partIn , int qty) {  // add a quantity of a particular stock item
38    for(Part prt : stockItems) {//Search through parts in stock
39       if(prt.getPartNo() == partIn.getPartNo()) {//if the part number matches one in stock
40          int currentQty = prt.getInStockQty();  //record current stock level
41          prt.setInStockQty(prt.getInStockQty() + qty); //update the new stock level
42             if (currentQty== 0 ) { //if the existing stock level was 0
43                contactCustomers(partIn);   // TRIGGER THE NOTIFICATIONS!
44             }
45
46       }
47    }
```

public Mailservice getService() - Just a “Getter” method not really used in our scenario – ignore

public void createInterestedListeners(partIn) – This is used to filter and create the EmailNotificationListeners and the SMSNotificationListeners.

The way we do this is to check firstly whether the item was “out of stock” before we replenish it. We then check all customers to see if they have this part held in the “interested” field of the customer instances. ie: Has the customer expressed an interest in this Part? Then we check the customer fields to see if they have subscribed to e-mail, SMS, both or neither

So where:

- The Part “WAS” previously “out of stock”
- The customer is Interested in the particular Part
- ...and the customer is a subscriber – either to e-mail or SMS

Then we can create the appropriate EmailNotificationListener
...or SMSNotificationListener
...and add them to the appropriate List in the MailService instance.

If we still “had stock” of the Part, they are not interested or are not subscribing they will not be added into the MailService Lists but will still be held in the customers List for the Store and could become subscribers at any time.

Public void addStock(Part partIn, int Qty) – with this method we add or replenish a “Part into stock” by the amount “Qty”

The Steps:

- Iterate through the current stock items to see if the part exists already?
- If it exists “in Stock” safe the amount currently in stock to a local variable “currentQty”
- Now update the stock levels to existing stock level + the Qty coming in
- Next check to see if the currentQty ie: pre update level is “0” zero.

If it “is zero” - this implies that a customer may have been unable to make a purchase but might have expressed an “interest” ?

- So we should call the contactCustomers(partIn) method to trigger automatic email and SMS messages to let them know the Part is now available

```
public void addStock(Part partIn , int qty) {
        for(Part prt : stockItems) {
                if(prt.getPartNo() == partIn.getPartNo()) {
                        int currentQty = prt.getInStockQty();
                        prt.setInStockQty(prt.getInStockQty() + qty);
                                if (currentQty== 0 ) {
                                        contactCustomers(partIn);    //...ignored closing brackets
```

The Program Class

```
/* JAVA DESIGN PATTERNS - THE OBSERVER PATTERN (A Responsibility Pattern)
The reasoning behind the Observer Pattern is the creation of a "One to Many Dependancy" between objects where if an Object changes state the dependants of that
Object will be of the state change and can perform some actions based on that change.
In our example I have a hardware store where replenishment of "empty or 0" stock triggers a notification - The dependants can then email or SMS the customer with an update.  */
public class Program {

   public static void main(String[] args) {
//Create a Store instance-------------------------------------------------------
Store JoesHardware = new Store();   // A simple Hardware Store that takes subscriptions so that if an item is out of stock we can email or sms the customer when stock arrives in

//Add some stock items (not necessarily in stock - see hammer)---------
Part screwdriver = new Part(1111,"Screwdriver", 30, 3.99);
JoesHardware .stockItems.add(screwdriver);
Part spanner = new Part(1112,"Spanner", 20, 4.99);
JoesHardware .stockItems.add(spanner);
Part hammer = new Part(1113,"Claw Hammer", 0, 6.99);  // Item out of Stock!!!
JoesHardware .stockItems.add(hammer);

//Create some Customers--------------------------------------------------------
Customer cust1 = new Customer("John", "Johnston", "john77johnston@gmail.com", "078260260");
Customer cust2= new Customer("Graeme", "Watson", "graeme977watson@yahoo.com", "078260261");
Customer cust3 = new Customer("Robin", "Menzies", "robm897@pccsystems.co.uk", "078260262");
Customer cust4 = new Customer("Duncan", "Dalgleish", "duncan.dalgleish@avcdirect.co.uk", "078260263");

// Set subscriptions-----------------------------------------------------------Remember we have stock on some items!
cust1.setEmlSubscriber(true);  // John is an E-Mail Subscriber
cust2.setEmlSubscriber(false);  // Graeme is not an E-Mail Subscriber
cust3.setEmlSubscriber(true);  // Robin is an E-Mail Subscriber
cust4.setEmlSubscriber(false);  // Duncan is not an E-Mail Subscriber
cust1.setSmsSubscriber(false); // John is not an SMS Subscriber
cust2.setSmsSubscriber(false);  // Graeme is not an SMS Subscriber either
cust3.setSmsSubscriber(true);  // Robin is an SMS Subscriber
cust4.setSmsSubscriber(true);  // Duncan is an SMS Subscriber

//Who needs a hammer-----------------------------------------------------------
cust1.setInterest(hammer); // John needs a hammer
cust2.setInterest(spanner); // Graeme needs a spanner (in store)
cust3.setInterest(screwdriver); // Rob needs a screwdriver (in store)
cust4.setInterest(hammer); // Duncan needs a hammer

//Add each Customer to customers list------------------------------------------
JoesHardware.customers.add(cust1);  //Adds each customer to the customers ArrayList of the Store
JoesHardware.customers.add(cust2);
JoesHardware.customers.add(cust3);
JoesHardware.customers.add(cust4);

//How many hammers do we have-------------------------------------------------//Unnecessary - Just confirm they are there zero hammers before we start
int item = JoesHardware.stockItems.indexOf(hammer);  //Unnecessary - get the index of "hammer"
System.out.println("We have " + JoesHardware.stockItems.get(item).getInStockQty()+ " hammers in Stock\n"); //Unnecessary - get and print out the quantity
//Load up the Listeners into the MailService ArrayLists for email and sms subscribers
JoesHardware.createInterestedListeners(hammer);
System.out.println(JoesHardware.subscribers.emlsubscribers.toString() + "\n");  //Unnecessary - Just confirm they are there
System.out.println(JoesHardware.subscribers.smssubscribers.toString() + "\n"); //Unnecessary - Just confirm they are there
JoesHardware.addStock(hammer, 10);// THIS IS THE TRIGGER - REMEMBER THE STOCK LEVEL FOR HAMMER IS CURRENTLY "0" ZERO
  }
}
```

So we need a “Program” class with a “main” method to try all this out

Let’s see how it works -

First we create an instance of the “Store” class – “JoesHardware”

Then we create some parts – “screwdriver”, “spanner” and “hammer”

The Part Constructor includes (partNo, description, quantity, price)

We have 30 x screwdrivers, 20 x spanners and zero hammers in stock

Ah! - So hammers are going to be “out of stock!!!”

We “Add()” our parts into the “stockItems” ArrayList field of the “Store” classes instance “JoesHardware”

“JoesHardware” now has a stock List (“0” being a valid qty for a Part)

Next we create some customers

The Customer Constructor takes (forename, surname, email and mobile No)

Nothing special here - we just create four instances (John, Graeme, Robin and Duncan)

Note: none of the Customer instances have an “Interest” or have “subscribed” to anything – These will be their choices.

So let’s ask them -?

```
cust1.setEmlSubscriber(true);   // John is an E-Mail Subscriber
cust2.setEmlSubscriber(false);  // Graeme is not an E-Mail Subscriber
cust3.setEmlSubscriber(true);   // Robin is an E-Mail Subscriber
cust4.setEmlSubscriber(false);  // Duncan is not an E-Mail Subscriber
cust1.setSmsSubscriber(false); // John is not an SMS Subscriber
cust2.setSmsSubscriber(false);  // Graeme is not an SMS Subscriber either
cust3.setSmsSubscriber(true);   // Robin is an SMS Subscriber
cust4.setSmsSubscriber(true);   // Duncan is an SMS Subscriber
```

...and have you found what you need?

John, “Actually I was looking for a hammer!”
Duncan, “By coincidence, I was also looking for a hammer!”

Joe, “My apologies we are out of hammers as you are both subscribed we will e-mail or sms message you automatically when they arrive in?
They should be in shortly!”

John, Duncan, “Perfect!”

Joe sets the Interest fields for his customers

```
cust1.setInterest(hammer); // John needs a hammer
cust2.setInterest(spanner); // Graeme needs a spanner (in store no need to set)-will be ignored*
cust3.setInterest(screwdriver); // Rob needs a screwdriver (in store no need to set)-also ignored*
cust4.setInterest(hammer); // Duncan needs a hammer
```

*__Aside__ -Remember the filter rules – stock must be zero before interest is checked
ie: Joe should only set an interest if the item is “out of stock”. The danger here is that the customers could at a later stage get emails telling them that we have stock of an item they have already purchased. From a design & testing point of view we want to know that the filtering is behaving properly so I have included these entries. This is good practice – We now know that we should not allow Joe to do this by mistake.

So we now have two customers who would like informed when the new hammers arrive in:
John – an e-mail subscriber and Duncan an SMS subscriber.

Lets add our Customers ArrayList - (All customers not just subscribers)

The customers ArrayList is a field of the Store class instance “JoesHardware”

```
JoesHardware.customers.add(cust1);
JoesHardware.customers.add(cust2);
JoesHardware.customers.add(cust3);
JoesHardware.customers.add(cust4);
```

That was easy

So – FINALLY !!! 

```
JoesHardware.addStock(hammer, 10);
```

THIS IS THE TRIGGER - REMEMBER THE STOCK LEVEL FOR HAMMER IS CURRENTLY "0" ZERO

All the conditions should be met to create to NotificationListeners

One EmailNotificationListener associated with John and one SMSNotificationListener Associated with Duncan.

These will be added into the MailService ArrayLists as email or SMS

The addStock(Part, Qty) method finishes by calling the Store method contactCustomers(Part) which in turn triggers the MailService notifyCustomers(Part) method which triggers the update(Part) method of the Listeners – phew!!!

The listeners send John an e-mail and Duncan an SMS message.

Your turn – Have a Go! ...or at least play around with & test the code.

```
E-Mail To: john77johnston@gmail.com Message: Hi John Just to let you know we now have 10 Claw Hammer's in Stock
SMS To: 078260263 Message: Hi Duncan Just to let you know we now have 10 Claw Hammer's  in Stock
```

Just to prove it works – lol

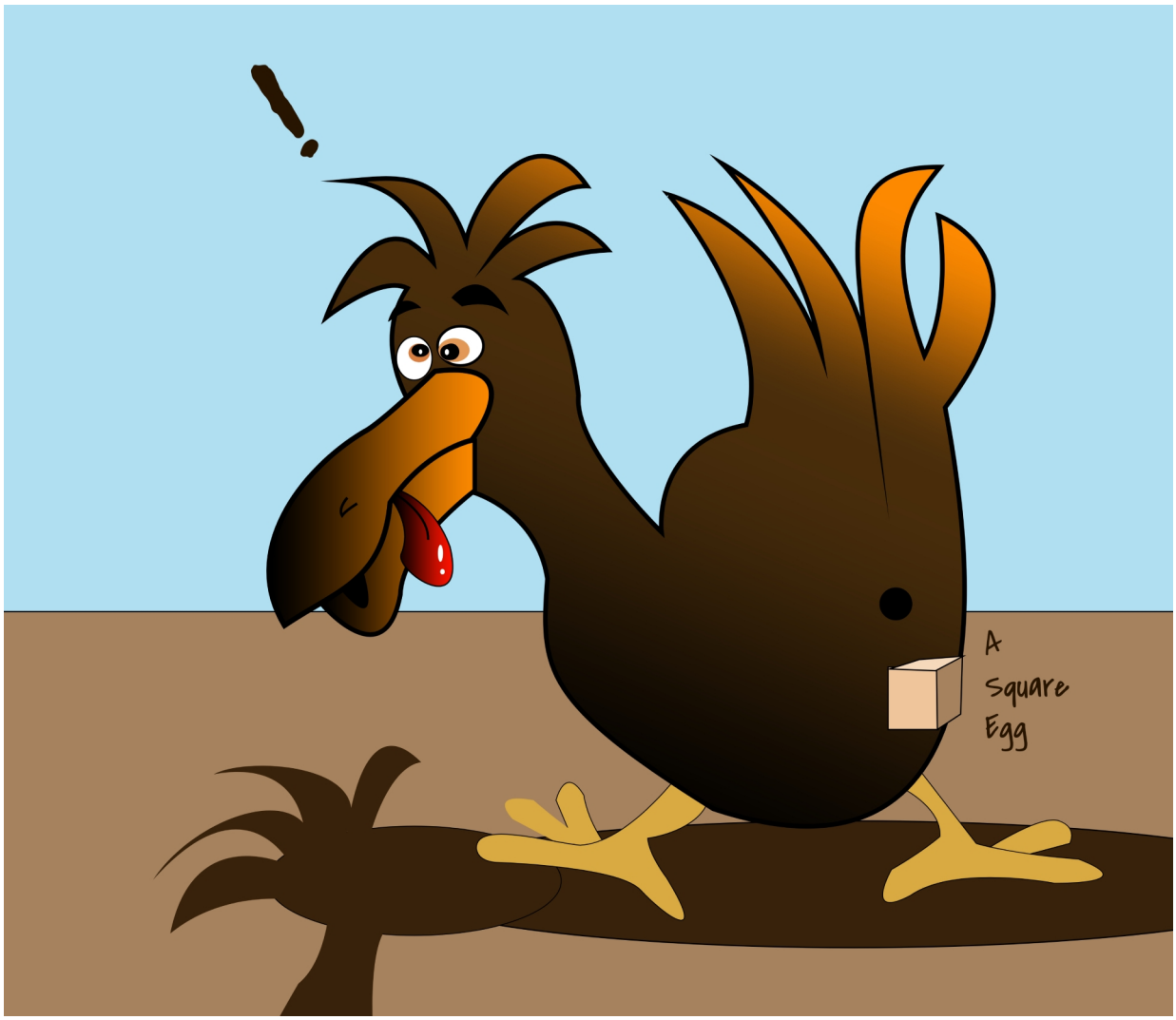
Recap:

JAVA DESIGN PATTERNS - “THE OBSERVER PATTERN” (A Responsibility Pattern)
The reasoning behind the Observer Pattern is the creation of a "One to Many Dependency" between objects where if an Object changes state the dependants of that
Object will be aware of the state change and can perform some actions based on that change.
In our example I have a hardware store where replenishment of "empty or 0" stock triggers a notification - The dependants can then email or SMS the customer with an update.

THE ADAPTER PATTERN

“The reasoning behind the Adapter is to provide a way in which two objects with different interfaces can be used In a similar way. Just like an adapter to allow a European plug sit next to a UK plug with both providing some light. The adapter simply allows the Euro plug access to the UK socket”.

So “Square peg in a round hole”



Peg! Not Egg???

So let's have a simple Class Round_Hole

```
package com.J1ggy;

public class Round_Hole { //Class for a round hole
  private double diameter; // field for it's diameter

//Constructor_____________________________________________-
  public Round_Hole(double diameter) {  //Constructor simply sets the diameter
    this.diameter = diameter;
  }

  public double getDiameter() { // Get the Diameter
    return diameter;
  }

  public boolean fits(Round_Peg peg) { // fits method takes an instance of Round_Peg  and returns a boolean of true if it is a snug fit to the hole
    boolean result = false;
    result = ((peg.getDiameter() >= this.getDiameter() -0.5) &&( peg.getDiameter()<=this.getDiameter()-0.) );
    return result;
  }
}
```

It has one field - "diameter"
A Constructor which sets the "diameter"
& a single method "fits()" which takes in an instance of a "Round_Peg" class and checks whether it is a snug fit to the "Round_Hole".
The return is a boolean true/false (does the peg fit the hole?)

Let's look at the Round_Peg class

```
package com.J1ggy;

public class Round_Peg {//Class Round_Peg
  private double diameter;

  public Round_Peg() {// This Constructor will be inherited by the adapter class "Lathe" extends "Round_Peg"
    super();
    }

  public Round_Peg(double diameter) {//Constructor normally used for Round_Peg - Sets the diameter
    super();
    this.diameter = diameter;
  }

  public double getDiameter() {// Get method to return the diameter
    return diameter;
}
}
```

A simple class with a field "diameter" (The diameter of the peg)
Two alternative Constructors:

- A Constructor which just creates a "Round_Peg" instance.
- A Constructor creates a "Round_Peg" instance and sets it's "diameter".

A method "getDiameter()" which returns the diameter of the peg instance.

So How do we define a Square_Peg

```
package com.J1ggy;

public class Square_Peg {
   private double width;


public Square_Peg(double width) {
   this.width = width;
}

public double getWidth() {
   return width;
}

}
```

Again a simple class – This time with a “width” rather than a “diameter”
So one field called “width”
A Constructor which sets the width
and a “getWidth()” method to return the width.

The Lathe Class (This is the Adapter Class)

```
package com.J1ggy;
// THIS IS THE ADAPTER CLASS - WE EXTEND THE BEHAVIOUR OF Round_Peg TO ACCEPT A SQUARE PEG (For the Lathe -  LOL)
public class Lathe  extends Round_Peg{
    private Square_Peg peg;

    public Lathe (Square_Peg peg) {
       this.peg = peg;
    }
    @Override
    public double getDiameter() {
       return this.peg.getWidth();
    }
}
```

The trick here is to inherit a class from the “Round_Peg” class and encapsulate an instance of the “Square_Peg” class.

Think about real life – If we have a Square_Peg (or for that matter a Round_Peg that is too thick) – Provided we have sufficient thickness we can turn the peg down on the lathe until we get a perfect fit for the hole.

If we “extend” the Round_Hole class we inherit all its methods – including “getDiameter()”.
If we have a “Square_Peg” encapsulated within the “Lathe” class then the width of that peg can become it’s diameter. All we are doing is shaving off the corners.
To keep it simple I have deferred the size checking to the main program.

So all we are doing is holding an instance of “Square_Peg” and providing a method to retrieve its “diameter”

We could of course have implemented checks on the Square_Pegs width before deciding whether the peg is of a suitable thickness to be put in the Lathe.

The Program

```
package com.J1ggy;

public class Program {  // Class for the main Program

   public static void main(String[] args) {  // Normal main() method
      // Round fits round, no surprise.
      Round_Hole hole = new Round_Hole(5.0);  // Create an instance of "Round_Hole" called simply "hole" and set its diameter to 5.0mm
      Round_Peg roundPeg = new Round_Peg(4.9);  // Create an instance of "Round_Peg" called simply "roundPeg" and set its diameter to 4.9mm ( a snug fit!!! )
      if (hole.fits(roundPeg)) { // pass our roundpeg to the "fits(roundpeg)" method (Which of course should return true as 4.9mm is a snug fit to 5mm)
         System.out.println("The round peg fits round hole ."); //Unsurprisingly - Print out a positive result - "The round peg fits round hole ."
      }else if(roundPeg.getDiameter() > hole.getDiameter()) {System.out.println("We'll turn that down on the lathe!");} // else if we have a thicker peg - "Turn it down on the Lathe"
      else{System.out.println("Scrap wood i'm afraid");} //else if the peg is thinner than the hole - Report that it is scrap.

      Square_Peg SquarePeg_1 = new Square_Peg(2.5);  //Create an instance of Square_Peg and set its width to 2.5mm (ie: This will be too thin to turn a Round_Peg)
      Square_Peg SquarePeg_2 = new Square_Peg(8.0); //Create an instance of Square_Peg and set its width to 8.0mm (ie: This will be thick enough to turn a Round_Peg)
      // hole.fits(An instance of Squre_Peg); // Won't compile. - Remember the "fits(Round_Peg roundPeg)" method of "hole" expects round pegs - not square pegs!

      // Adapter solves the problem.
      Lathe newPeg_1 = new Lathe (SquarePeg_1 );// Now we wrap the "Square_Peg" instances in instances of "Lathe" - which will provide them with a "getDiameter()" method
      Lathe  newPeg_2 = new Lathe (SquarePeg_2);// Because "Lathe" extends from "Round_Peg"  - The "fits(Round_Peg Lathe instance)" method of "hole" will now compile
      if (hole.fits(newPeg_1)){ // If the first new instance of "Lathe" fits the "hole" -  (Actually it will be too thin) - So returns false
         System.out.println("The first new peg  now fits the round hole"); // Not actioned as false result
      }else if (newPeg_1.getDiameter() > hole.getDiameter()){System.out.println("We'll turn that down on the lathe!");}// else if thick enough (again false) -  "We'll turn that down on the lathe!")
      else {System.out.println("Scrap wood i'm afraid");} // else if still false (Which we are in this case) - Report "Scrap wood i'm afraid"
      if (hole.fits(newPeg_2)){ System.out.println("The second new  peg now  fits the round hole");}// If the second "Lathe" instance 8.0mm fits the hole - (false too thick) - "The second new  peg now  fits the round hole"
      else if (newPeg_2.getDiameter() > hole.getDiameter()){System.out.println("We'll turn that down on the lathe!");} // If the second "Lathe" instance 8.0mm is thicker(true) -  "We'll turn that down on the lathe!"
      else {System.out.println("Scrap wood i'm afraid");} //else - never reached as 2nd "Lathe" instance was thick enough - "Scrap wood i'm afraid"
   }
}
```

By wrapping the instances of Square_Peg inside an Adapter class (in this case Lathe) we can give them the ability to act like the Round_Peg instances. This is achieved through use of polymorphism, inheritance and encapsulation.

Polymorphism – many forms - a Round_Peg instance can also be a Lathe instance which extends or is a type of Round_Peg.

Inheritance – is a type of – A Lathe instance is a type of Round_Peg and inherits its fields and methods. Which it can then over-ride with its own implementations.

Encapsulation – The Lathe instances contain or encapsulate an instance of Square_Peg but additionally inherit the properties of a Round_Peg. By encapsulating an instance of Square_Peg we can access its methods and fields from within our Lathe instance and “Adapt” the behaviour to suit the expected interface of a Round_Peg.

Listed below is the Comments or “Pseudo Code” version of our program – Please read through this then each code listing and confirm to yourself that you fully understand what is happening.

Ask your Tutor if you have any questions – Don’t sit quietly if you are confused.

Program Comments “Pseudo Code”:

Create an instance of "Round_Hole" called simply "hole" and set its diameter to 5.0mm

Create an instance of "Round_Peg" called simply "roundPeg" and set its diameter to 4.9mm (a snug fit!!!)

pass our roundpeg instance to the "fits(roundpeg)" method of the “hole” instance - (Which of course should return “true” as 4.9mm is a snug fit to 5mm)

Unsurprisingly a 4.9mm peg fits a 5mm hole - Print out a positive result - *"The round peg fits round hole ."*

Had it been thicker:
else if we have a thicker peg - *"Turn it down on the Lathe"*

Had it been too thin:
else if the peg is thinner than the hole - Report - *Scrap wood i'm afraid"*

Create an instance of Square_Peg and set its width to 2.5mm (ie: This will be too thin to turn a Round_Peg)

Create an instance of Square_Peg and set its width to 8.0mm (ie: This will be thick enough to turn a Round_Peg)

IMPORTANT CONCEPT:

hole.fits(An instance of Square_Peg); Won't compile. - Remember the "fits(Round_Peg roundPeg)" method of "hole" expects round pegs - not square pegs!

Adapter solves the problem:

Now we wrap the "Square_Peg" instances into instances of "Lathe" - which will provide them with a "getDiameter()" method

Because "Lathe" extends from "Round_Peg" - The "fits(Round_Peg Lathe instance)" method of "hole" will now compile

If the first new instance of "Lathe" fits the "hole" - (Actually it will be too thin) - So returns “false” - *"The first new peg now fits the round hole"*

else if thick enough (again “false”) - "We'll turn that down on the lathe!")

else if still “false” (Which we are in this case) - Report *"Scrap wood i'm afraid"*

If the second "Lathe" instance 8.0mm fits the hole - (“false” too thick) - *"The second new peg now fits the round hole"*

If the second "Lathe" instance 8.0mm is thicker(“true”) - *"We'll turn that down on the lathe!"*

else - never reached as 2nd "Lathe" instance was thick enough - *"Scrap wood i'm afraid"*

Summary:
So an Adapter class is just like an adapter plug. The trick is to create an Adapter class which inherits the properties of the interface that the program “expects” and to encapsulate within the Adapter class an instance of the class you wish to adapt.

The Inheritance or extension of the base class facilitates the interactions with the expected interface.

The encapsulation facilitates the adaption of the alternative class to seamlessly carry out the required implementation whilst hiding the complexity from the user.

“Poor Chicken...”