DESIGN INTERVIEW QUESTIONS

Chapter-8

☼ ☼ ☼

8.1 Design Interview Questions

In this chapter, we will discuss few common interview questions. Along with them, we have presented few practice questions as well. Try designing classes for those problems so that you will get more confidence in object oriented designing.

For some design problems we have used $C++$ classes to explore $C++$ concepts. If you are not a $C++$ professional, feel free to download the *Java* code online or try designing your own classes.

Problem-1 Can you explain about access specifiers in *Java*?

Solution: In *Java*, classes, variables, methods and constructors can have *access specifiers*. There are 4 types of access specifiers:

- *private*
- *protected*
- *public*
- No specifier or none

It is important to remember that constructors in *Java* are treated differently than methods. Constructors are *not* considerd a class member. Class members are made of 2 things:

- Class's variables.
- Class's methods.

Access modifiers specify who can access them. Its effect is different when used on any of:

- Class
- Class variable
- Class method
- Class's constructor

Access Specifiers for Class Variables and Class Methods: Below is a table showing the effects of access specifiers for class members (i.e. class variable and class methods).

Yes = Can Access. No = No Access.

Specifier	Class	Subclass	Package	Other Packages
private	Yes	No	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes
No specifier	Yes	No	Yes	No

For example, if a variable is declared *protected*, then the same class can access it, its subclass can access it, and any class in the same package can also access it, but otherwise a class cannot access it.

If a class member doesn't have any access specifier (the *No specifier* row in above), its access level is sometimes known as *package*.

```
class Test {
   int x = 7;
}
public class Testing {
   public static void main(String[] args) {
      Test var = new Test ();
      System.out.println(var.x);
   }
}
```

The above code compiles and runs. But, if we add *private* in front of int *x*, then we will get a compiler error: “x has private access in Test”. This is because when a member variable is *private*, it can *only* be accessed within that class.

Access Specifiers for Constructors: Constructors can have the same access specifiers used for variables and methods. Their meaning is same. For example, when a constructor has *private* declared, then, only the class itself can create an instance of it. Other class in the same package *cannot* create an instance of that class *nor* any subclass of that class. Nor any other class outside of this package.

```
class Test {
   public int x;
   private Test (int n) {
      x=n;
      System.out.println("Test Message!");
   }
}
public class Testing {
   public static void main(String[] args) {
      Test t = new Test (3);
      System.out.println(t.x);
   }
}
```

In the above code, it won't compile because Test's constructor is *private* but it is being created outside of itself. If we delete the *private* keyword in front of *Test*'s constructor, then it compiles.

Constructors in Same Class Can Have Different Access Specifiers: Remember that a class can have more than one constructor, each with different parameters. Each constructor can have different access specifier. In the following example, the class *Test* has two constructors; one takes an *int* argument, the other takes a *double* argument. One is declared *private*, while the other with no access specifier (default package level access).

```
class Test {
   Test (int n) {
      System.out.println("Test Message-1”);
   }
   private Test (double d) {
        System.out.println("Test Message-2");
    }
}
public class Testing {
    public static void main(String[] args) {
        Test t1 = new Test (3);
        // Test t2 = new Test (3.3);
    }
}
```

The fact that there can be constructors with different access specifiers means that in Java, the ability to create an object also depends on which constructor is called to create the object.

Access Specifiers for Classes: For classes, only the *public* access specifier can be used on classes. That is, in every java source code file, only one class in the file is public accessible and that class must have the same name as the file. For example, if the file is *ABC.java*, then there must be a class named *ABC* in it, and that is the class that's public. Optionally, the class can be declared with *public* keyword. By convention, class's names should start with capital letter. So, a class named *ABC* really should be *ABC*, with the file named *ABC.java*. If we use any other access specifier on classes, or declare more than one class *public* in a file, the compiler will complain.

Problem-2 Given a database application, can you think of classes to generate unique sequence numbers?

Solution: We can use *Singleton* pattern to create a counter to provide unique sequential numbers. This can be used as *primary keys* in a database.

```
public class SequenceNumbers {
    private static SequenceNumber instance;
    private static int counter;
    private SequenceNumber(){
        counter = 0;                    // starting value
    }
    public static synchronized SequenceNumber getInstance(){
        if(instance == null){           // Lazy instantiation
            instance = new SequenceNumber();
        }
        return instance;
    }
    public synchronized int getNext(){
        return ++counter;
    }
}
```

Notes about this implementation:

- Synchronized methods are used to ensure that the class is thread-safe.
- This class cannot be subclassed because the constructor is private. This may or may not be a good thing depending on the resource being protected. To allow subclassing, the visibility of the constructor should be changed to protected.
- Object serialization can cause problems; if a Singleton is serialized and then de-serialized more than once, there will be multiple objects and not a singleton.

Problem-3 Simulate a supermarket (grocery store).

Solution: Before creating the objects let's understand the use-cases for simulating a grocery store. At very top-level, a customer walks into a grocery store, picks up a few items, pays for them, and leaves. To make the problem simple we will discard the corner cases (e.g., what if customer forgets to bring money after billing). The obvious objects of any grocery store are:

- A customer
- A store
- Grocery items of various sorts
- A cashier

One doubt that may arise here is: Do we need to subclass the grocery items (beauty, health, cook etc.)? The answer is no, because these items don't have different behaviors. There is no obvious reason for any of these to be a superclass (or subclass) of any of the others. Another doubt that may arise here is: Should we make a class Person and use it as a superclass of *Customer* and *Cashier*? To make it simple let us not do that. We can always do it later, if we find some data or actions that *Customer* and *Cashier* should both have.

Now, let us start creating classes. The grocery items should be in the *store*, and initially only the store needs to know about them, so we will let the *store* create those. The *Customer* and the *Cashier* both need to know about the *store* (but the *store* kind of just sits there). So, we probably should create the store first. This is because, when we create the *Cashier* and the *Customer*, we want to create them with knowledge of the *store*.

The store is the central idea of this program, so let's put a main method in the *store* to kick things off. Let us call our main class as *GroceryStore*. What does our main method need to do?

- Create a GroceryStore
- Create a Cashier for the store
- Create a Customer who knows about the store
- Tell the Customer to "shop"

```
class GroceryStore {
    public static void main(String args[]) {
        GroceryStore store = new GroceryStore();
        Cashier cashier = new Cashier(store);
        Customer customer = new Customer(store);
        customer.shop();
    }
}
```

With the above code, the customer only shops in this one store. This is adequate for our program, but it's very restrictive and it's trivial to fix. So, the Customer needs to shop: this includes selecting groceries and paying for them.

```
class Customer {
    public void shop(GroceryStore store) {
       selectGroceries(store); // because the store holds the groceries
       checkOut(???); // who or what do we pay?
}
```

Obviously, the *Customer* should pay the *Cashier*. But how does the *Customer* know about the *Cashier*? The Customer knows about (can reference) the GroceryStore and the Cashier knows about the *GroceryStore*. Neither the GroceryStore nor the Customer knows about the Cashier. To fix this, the *Customer* do not know about the *Cashier* as they don't know any clerks personally. Buy the GroceryStore know about the Cashier and the Cashier still needs to know about the GroceryStore. So, the customer class can be changed as:

```
class Customer {
    public void shop(GroceryStore store) {
       selectGroceries(store);
       checkOut(store);
}
```

At this point, we understand that the Customer knows about the GroceryStore, the GroceryStore knows about the Cashier and hence, the Customer can ask the GroceryStore about the Cashier.

```
class GroceryStore {
          GroceryStore() {...} // Constructor
          public static void main(String args[]) {...}
          public void hire(Clerk clerk) {...}
          public Clerk getClerk() {...}
}
class Customer {
          public void shop(GroceryStore store) {...}
          public void selectGroceries(GroceryStore  store) {...}
          checkOut(GroceryStore  store) {...}
}
```

There's just one Cashier, whom we hired like this:

```
GroceryStore store = new GroceryStore();
Cashier cashier = new Cashier();
store.hire(cashier);
```

So we need to write the hire method. Also, don't forget the store and Cashier need to know about each other.

```
class GroceryStore {
          Cashier myCashier;
          public void hire(Cashier cashier) {
                    myCashier = Cashier;
                    cashier.takePosition(this);
          }
}
class Cashier {
          GroceryStore myStore;
          public void takePosition(GroceryStore store) {
                    myStore = store;
          }
}
```

The Customer call gets the Cashier from GroceryStore as:

```
class Store {
          Cashier myCashier;
          ...
          public Cashier getCashier() {
                    return myCashier;
          }
```

```
        ...
}
```

Next, construct a Store containing an array of GroceryItems (along with how many of each).

```
public int KINDS_OF_ITEMS = 4;
public GroceryItem[ ] item = new GroceryItem[KINDS_OF_ITEMS];
public int[ ] itemCount = new int[KINDS_OF_ITEMS];

GroceryStore() {
    item[0] = new GroceryItem("milk", 2.12);
    item[1] = new GroceryItem("butter", 2.50);
    item[2] = new GroceryItem("eggs", 0.89);
    item[3] = new GroceryItem("bread", 1.59);
    for (int i = 0; i < KINDS_OF_ITEMS; i++) {
        itemCount[i] = 50;  // the store has lots of everything
    }
}
```

Customer selects the items as:

```
GroceryItem[ ] myShoppingBasket = new GroceryItem[20];
Random random = new Random();
public void selectGroceries(GroceryStore store) {
    int itemsInMyBasket = 0;
    for (int i = 0; i < store.KINDS_OF_ITEMS; i++) {
        for (int j = 0; j < 3; j++) {
            if (random.nextInt(2) == 1) {                    // choose up to 3 of it
                myShoppingBasket[itemsInMyBasket] = store.item[i];
                store.itemCount[i] = store.itemCount[i] - 1;
                itemsInMyBasket = itemsInMyBasket + 1;
            }
        }
    }
}
```

The Customer can checkout as:

```
void checkOut(GroceryStore  store) {
     Cashier cashier = store.getCashier();
     double total = cashier.getBill(myShoppingBasket);
     myMoney = myMoney - total;
     cashier.pay(total);
}
```

The final code for our discussion is:

```
public class GroceryStore {
    Cashier myCashier;
    public int KINDS_OF_ITEMS = 4;
    public GroceryItem[ ] item = new GroceryItem[KINDS_OF_ITEMS];
    public int[ ] itemCount = new int[KINDS_OF_ITEMS];
    double money = 1000.00;

    GroceryStore() {
        item[0] = new GroceryItem("milk", 2.12);
        item[1] = new GroceryItem("butter", 2.50);
        item[2] = new GroceryItem("eggs", 0.89);
        item[3] = new GroceryItem("bread", 1.59);
        for (int i = 0; i < KINDS_OF_ITEMS; i++) {
            itemCount[i] = 50;  // the store has lots of everything
        }
    }
    public static void main(String args[]) {
        GroceryStore store = new GroceryStore();
        Cashier cashier = new Cashier();
        store.hire(cashier);
        Customer customer = new Customer();
        customer.shop(store);
    }
    public void hire(Cashier cashier) {
        myCashier = cashier;
        cashier.takePosition(this);          // "this" = this store
    }
    public Cashier getCashier() {
        return myCashier;
    }
}
public class Customer {
    GroceryItem[ ] myShoppingBasket = new GroceryItem[20];
    Random random = new Random();
    double myMoney = 100.00;
    public void shop(GroceryStore store) {
        selectGroceries(store); // because the store holds the groceries
        checkOut(store);
    }
    public void selectGroceries(GroceryStore store) {
        int itemsInMyBasket = 0;
        for (int i = 0; i < store.KINDS_OF_ITEMS; i++) { // for each kind of item
            for (int j = 0; j < 3; j++) {               // choose up to 3 of it
                if (random.nextInt(2) == 1) {
                    myShoppingBasket[itemsInMyBasket] = store.item[i];
                    store.itemCount[i] = store.itemCount[i] - 1;
                    itemsInMyBasket = itemsInMyBasket + 1;
                }
            }
        }
    }
    void checkOut(GroceryStore  store) {
        Cashier cashier = store.getCashier();
        double total = cashier.getBill(myShoppingBasket);
        myMoney = myMoney - total;
        cashier.pay(total);
    }
}
public class Cashier {
        GroceryStore myStore;
        public void takePosition(GroceryStore store) {
            myStore = store;
        }
        public double getBill(GroceryItem[] item) {
            double total = 0;
            int itemNumber = 0;
            while (item[itemNumber] != null) {
                total = total + item[itemNumber].price;
                System.out.println(item[itemNumber].name + " " +  item[itemNumber].price);
                itemNumber = itemNumber + 1;
            }
            System.out.println("TOTAL      " + total);
            return total;
        }
        public void pay(double amount) {
            myStore.money = myStore.money + amount;
        }
}
public class GroceryItem {
    public String name;
    public double price;

    GroceryItem(String name, double price) {
        this.name = name;
        this.price = price;
    }
}
```

In addition to above discussion, it is worth mentioning the other points for extensions:

- Providing interface for changing the prices of items.
- Providing discount coupons while checkouts.
- Categorizing the store employees and assigning roles and many more.

Problem-4 Consider a company which wants to process salary hikes of its employees during recession period. As a precautionary measure, instead of hiking all employee salaries it decided to hike only for the employees who met at least any two of the criteria. Can you design the classes and functions to help the company for processing the hike letters?

- Published at least two research papers.
- Got star of the year award.
- Completed at least 5 years of experience.

Solution: As the problem states, the two basic objects of the problem are: *Employee* and *Company*. For each employee, we need to maintain additional information such as number of papers published by employee, whether the employee got the award or not, and also the number of years he spend in current company. To maintain that information, we can define the interface of *Employee* as:

```
public interface Employee{
    public int getName();               // returns name of employee
    public int getAge();                // returns age of employee
    public int getYearsOnJob();         // number of years on job
    public double getSalary();          // salary in dollars
    public int getID();                 // unique employee ID number
    public beeloan gotAward();          // whether the employee got award or not
    public int getCountPublished(); //number of papers published by employee
}
```

One possible implementation for the employee would be:

```
public class EmployeeImpl implements Employee{
   private int myName;
   private int myAge;
   private int myYearsExp;
   private double mySalary;
   private int myID;
   private boolean gotAward;
   private int papersPublished;
   public EmployeeImpl(String name, int age, int yearsExp, double salary,
                              int id, boolean award, int papersCount){
            myName = name;
           myAge = age;
           myYearsExp = yearsExp;
           mySalary = salary;
           myID = id;
           gotAward = award;
           papersPublished = papersCount;
   }
   public String getName(){
            return myName;
   }
   public int getAge() {
           return myAge;
   }
   public int getYearsOnJob(){
           return myYearsExp;
   }
   public double getSalary(){
           return mySalary;
   }
   public int getID(){
           return myID;
   }
   public boolean gotAward(){
            return gotAward;
   }
   public int getCountPublished(){
            return papersPublished;
   }
}
```

We are done with *Employee* class and let us start defining the *Company* class. For simplicity we can discard the unrelated functionality (for example, adding new employees, changing name of employee etc..). We can assume that the Company class is the main class for our problem (which means, Company class takes care of adding employees with the values defined). Once, it generates the employee list we can use that and process for salary hikes.

```
public class Company{
   private final static int MIN_PUBLISH_COUNT   = 2;
   private final static int MIN_EXP = 5;
```

```
    private ArrayList myEmployees;
    private Employee[] empList = {
            new EmployeeImpl("James Bond", 25,3,12000,1, true, 3),
            new EmployeeImpl("Steve Jobs",35,6,13000,2,false, 4),
            new EmployeeImpl("Bill Gates",30,2,14000,true, 1),
            new EmployeeImpl("Jeff",23,1,9999,4, true, 5),
            new EmployeeImpl("Steve Gates",57,15,20000,5, true, 10)
    };
    private double myTotalSalary;
    //set myTotalSalary as total budget = sum of all salaries
    private void calcSalaries(){
            myTotalSalary = 0;
            Iterator it = myEmployees.iterator();
            while (it.hasNext()){
               Employee e = (Employee) it.next();
               myTotalSalary += e.getSalary();
            }
    }
    Company(){
            myEmployees = new ArrayList();
            myEmployees.addAll(Arrays.asList(empList));
            calcSalaries();
    }
    public double getBudget(){
            return myTotalSalary;    //returns total of all employee salaries
    }
    public void printAll(){
            System.out.println("Number of employees = " + myEmployees.size());
            for(int k=0; k < myEmployees.size(); k++){
               Employee e = (Employee) myEmployees.get(k);
               System.out.println(k + ".\t id = " + e.getID()+ "\t$"+e.getSalary());
            }
            System.out.println("total budget = "+getBudget());
    }
    private boolean employeeIsEligible(Employee emp){
       return (emp.getCountPublished() >= MIN_PUBLISH_COUNT
                && emp.gotAward()) ||
               (emp.getCountPublished() >= MIN_PUBLISH_COUNT
                                   && emp.getYearsOnJob() >= MIN_EXP) ||
               (emp.getYearsOnJob() >= MIN_EXP &&  emp.gotAward());
    }
    public void processRetirements(){
            Iterator it = myEmployees.iterator();
            while (it.hasNext()){
               Employee e = (Employee) it.next();
               if (employeeIsEligible(e)){
                       System.out.println(e.getID() + " is eligible for salary hike\n");
               }
            }
    }
}
```

Problem-5 Design the library management system.

Solution: We all know how a library system works. The basic components of the library are: library items (like books, CDs, journals etc..), and a mechanism for giving books to users. The interface for LibraryItem is straightforward except for the decision to return a boolean for the method checkOut. To simplify our discussion, we can confine ourselves to only books. The book class can be defined as:

```
public class Book{
    private String theAuthor, theTitle, pageCount, year, edition;
    public Book(String author, String title, String pages,
                                                String yearPublished, String bookEdition){
      theAuthor = author;
      theTitle = title;
      pageCount = pages;
      year = yearPublished;
      edition = bookEdition;
    }
    public String getAuthor(){
      return theAuthor;
    }
    public String getTitle(){
      return theTitle;
    }
    public String getPageCount(){
      return pageCount;
    }
    public String getYear(){
      return year;
    }
    public String getEdition(){
      return edition;
    }
}
public interface LibraryItem{
    public String getID(); // return id of this item
     //return true if checking out (no holder) possible
    //and assign a new holder otherwise (existing holder) return false
    public boolean checkOut(String holder);
    public String getHolder();//return current holder
    // We can add more functions based on requirement
}
```

Now, to create a library book we can simply implement the *LibraryItem* interface and extend the book class. It shouldn't duplicate state from the *Book* class (so no author/title, those are accessible via super.getXX methods), but a *LibraryBook* needs an ID and a holder, so those instance variables are required.

```
public class LibraryBook extends Book implements LibraryItem{
   private String theID;
   private String theHolder;
   public LibraryBook(String author, String title, String id,
                                                    int pageCount, int year, int edition){
      super(author, title, pageCount, year, edition);
      theID = id;
   }
```

```
    public String getHolder(){
        return theHolder;
    }
    public boolean checkOut(String holder){
        if (theHolder == null){
            theHolder = holder;
            return true;
        }
        return false;
    }
    public String getID(){
        return theID;
    }
}
```

Now, let us define the library class which maintains the library items as a collection. The problem statement pretty much requires a mapping of IDs toLibraryItems to facilitate efficient implementation of the *checkOut* and *getHolder* methods. For this, we can use a hash map.

```
public class Library{
    private Map items;
    public Library(){
        items = new HashMap();
    }
    public void add(LibraryItem theItem){
        items.put(theItem.getID(), theItem);
    }
    public void checkout(String id, String holder){
        LibraryItem item = (LibraryItem) items.get(id);
        item.checkOut(holder); // ignore return value here
    }
   public String getHolder(String id){
        LibraryItem item = (LibraryItem) items.get(id);
        return item.getHolder();  // precondition: item in library
   }
}
```

Problem-6 Design a Cinema Management System.

Solution: Let us start our discussion by understanding the components of a typical Cinema Management System (CMS). As we know any cinema (movie that is being played) associated with a number of theaters and each theater contains a set of rows with a number of seats. Also, each seat will have a specification.

- *Seat specification*: Specifies the properties of a seat.
- *Seat*: A Seat keeps track of what type it is and whether or not it is reserved.
- *Row of seats*: Models a row of seats.
- *Seating chart*: Models a seating chart (set of rows of seats) for a theater.
- *Theater*: Models one theater, including its configuration of rows of seats.
- *Movie*: Models a movie. A movie screening may be the event at a Show.
- *CineGoer*: Models a cinema goer, a customer of the Cinema. Human customers will interact with the software system.
- *Show*: Models a show. For now, assume that the event at all shows is the screening of a movie, in one of the theaters of the Cinema, at a given time.

- *Reservation specification*: Specifies the properties of a reservation and mimics the SeatSpecification class.
- *Reservation*: Models a reservation by some movie-goer of some tickets for some show.
- *CinemaManagementSystem*: The Cinema class controls how many theatres it holds, and indexes those theatres.

Now, let us work on each of these components to make CMS complete. We can treat each of these as a separate class.

Seat specification: This class specifies the properties of a seat. For simplicity, we can implement this as a Hashmap <String, Object>. Also, as of this moment we need few methods to check what type a seat object is.

- *addFeature*: adds a specified feature to this seat featureLabel name of the feature e.g. aisle, front row, and featureValue value, typically true as input.
- *checkFeature*: returns the object of the given feature.
- *match*: returns true when this SeatSpecification has all the that givenSpec has, otherwise returns false. Assume that it takes givenSpec as input parameter and indicates the specification that this is being matched against.

```
public class SeatSpecification implements Serializable, Cloneable {
   private HashMap <String, Object> features;
   public SeatSpecification (){
      features = new HashMap <String, Object> ();
   }
   public void addFeature (String featureLabel, Object featureValue)
                                        throws InvalidParameterException {
      if (featureValue == null)
         throw new InvalidParameterException ("feature value cannot be null");
      else  features.put (featureLabel, featureValue);
   }
   public Object checkFeature(String featureLabel) {
      return features.get (featureLabel);
   }
   public boolean match (SeatSpecification givenSpec) {
      String currFeature;
      if (givenSpec == null)
         return false;
      else {
         Iterator it = givenSpec.features.keySet().iterator();
         while (it.hasNext()) {
            currFeature = (String) it.next();
            if ( ! ( features.get(currFeature) .equals (
                     givenSpec.features.get(currFeature) ) ) )
               return false;
         }
         return true;
      }
   }
   public SeatSpecification clone() {
      try {
         SeatSpecification c = (SeatSpecification) super.clone();
         c.features = new HashMap <String, Object> ();
         Iterator it = features.keySet().iterator();
```

```
          while(it.hasNext()){
             String currFeatureLabel = (String) it.next();
             Object currFeatureValue = features.get(currFeatureLabel);
             c.features.put (currFeatureLabel, currFeatureValue.clone());
          }
          return c;
       }
       catch (CloneNotSupportedException e) {
          return null;
       }
    }
}
```

Seat: A *seat* keeps track of what type it is and whether or not it is reserved. Every seat is associated with a SeatSpecification and a seat ID. Also, assume that we make a seat by combining a row id and seat# e.g. A1. Below are functions that can be added to *Seat* class.

- *setSeatFeature*: adds a specified feature to this seat (calls seat specification method)
- *match*: return true when this seat matches givenSpec, otherwise returns false (calls seat specification method)
- *isReserved*: return true if seat is reserved

```
public class Seat implements Serializable, Cloneable{
    private SeatSpecification spec;
    private String id;
    public Seat(String id) {
       this.id = id;
       spec = new SeatSpecification ();
    }
    public void setSeatFeature (String featureLabel, Object featureValue)
                        throws InvalidParameterException {
       spec.addFeature (featureLabel,  featureValue);
    }
    public boolean match(SeatSpecification givenSpec)  {
      return spec.match(givenSpec);
    }
    public String getID() {
       return id;
    }
    public boolean isReserved(){
       return true;
    }
   public Seat clone() {
       Seat cloneSeat = new Seat(id);
       cloneSeat.clone();
       return cloneSeat;
   }
}
```

Row of seats: Models a row of seats. Assume that a collection of seats are maintained in a hash map (HashMap <String, Seat>). Below are functions that can be added to this class.

- *setSeatFeature*: adds a specified feature to a specified seat. It takes seatNum (desired seat number), featureLabel (name of the feature e.g. aisle, front row) and featureValue (value, typically true) as input.
- *addSeat*: adds a single seat to current row of seats.
- *removeSeat*: removes a specified seat from current row of seats.

```
public class RowOfSeats implements Serializable, Cloneable{
    private HashMap<String, Seat> seats;                      //collection of seats
    private String rowId;
    private int numOfSeats;
    ArrayList<String> keys;
    public RowOfSeats (String rowId, int numOfSeats)
          throws InvalidParameterException {
       seats = new HashMap();
       int i = 0; starts at 1
       Seat s;
       while (i < numOfSeats) {
          s = new Seat (Integer.toString(i));
          seats.put(s.getID(), s);
          i++;
       }
       keys = new ArrayList<String>();
       this.rowId = rowId;
       this.numOfSeats = numOfSeats;
    }
    public void setSeatFeature (String seatID, String featureLabel,
                                    Object featureValue) throws InvalidParameterException {
        if (seats.containsKey(seatID)) {
            Seat s = seats.get(seatID);
            s.setSeatFeature (featureLabel,  featureValue);
        }
        else {
            throw new InvalidParameterException (" no seat " + seatID + " in row " + rowId);
        }
   }
   public String getRowId() {
       return rowId;
   }
   public int getNumOfSeats() {
       return seats.size();
   }
   public void addSeat() {
       Seat s;
       numOfSeats += 1;
       s = new Seat (Integer.toString(numOfSeats));
       seats.put(s.getID(), s);
       keys.add(Integer.toString(numOfSeats));
   }
   public void removeSeat(String seatID) throws InvalidParameterException {
       if (seats.containsKey(seatID)) {
          seats.remove(seatID);
        }
        else
        throw new InvalidParameterException("The specified Seat could not be found!");
    }
    public RowOfSeats clone() {
          RowOfSeats newRowOfSeats = (RowOfSeats) super.clone();
          Iterator it = seats.keySet().iterator();
          for(int i = 0; i <= keys.size(); i++) {
             String seatID = keys.get(i);
             while(it.hasNext()) {
                 newRowOfSeats.put(seatID, seats.get(seatID).clone());
             }
          }
         return newRowOfSeats;
      }
}
```

Seating chart: Models a seating chart for a theater and can be cloned by *Theater* class. Seating chart implements the following:

- Hash Map of Rows -> collection of rows with String keys
- *addRowToBack*: add a row from the chart
- *removeRow*: remove a row from the chart
- *setSeatFeature*: set a seat spec within a row
- *getNumberOfRows*: can return number of rows/ number of seats in a row

```
public class SeatingChart implements Cloneable, Serializable{
    HashMap <String, RowOfSeats> chart;
    ArrayList<String> keys;
    public SeatingChart() {
        chart = new HashMap <String, RowOfSeats>();
        keys = new ArrayList<String>();
    }
    public void addRowToBack(String rowId, int numOfSeats)
        throws InvalidParameterException {
           Iterator it = chart.keySet().iterator();
           while(it.hasNext()){
                if(chart.containsKey(rowId)){
                    throw new InvalidParameterException ("The row id " + rowId +
                                                                            "already exists.");
                }
            }
           RowOfSeats row = new RowOfSeats (rowId, numOfSeats);
           chart.put(rowId, row);
           keys.add(rowId);
    }
    public void removeRow(String rowId)
        throws InvalidParameterException {
        Iterator it = chart.keySet().iterator();
        while(it.hasNext()){
                if(chart.containsKey(rowId)){
                    chart.remove(rowId);
                    for(int i = 0; i <= keys.size(); i++){
```

```
                    if(rowId.equals(keys.get(i))){
                        keys.remove(i);
                    }
                }
            }
            else {
                throw new InvalidParameterException("that row id does not exist");
            }
        }
    }
    public void setSeatFeature (String rowId, String seatId, String featureLabel,
                                                                                    Object featureValue)
        throws InvalidParameterException {
        Iterator it = chart.keySet().iterator();
        while(it.hasNext()){
                if(chart.containsKey(rowId)){
                }
                else {
                    throw new InvalidParameterException("that row id or
                                                                            seat id does not exist");
                }
        }
    }
    public int getNumberOfRows(){
        return chart.size();
    }
    public int getNumberOfSeatsInRow(String rowId) {
        RowOfSeats currentRow = chart.get(rowId);
        return currentRow.getNumOfSeats();
    }
    public SeatingChart clone() {
        try {
            SeatingChart c = (SeatingChart) super.clone();
            c.keys = new ArrayList<String> ();
            for (String s : keys)
                c.keys.add (s);
            c.chart = new HashMap <String, RowOfSeats> ();
            Iterator it = chart.keySet().iterator();
            while(it.hasNext()){
                String currRowId = (String) it.next();
                RowOfSeats currROS = chart.get(currRowId);
                c.chart.put (currRowId, (RowOfSeats) currROS.clone());
            }
            return c;
        }
        catch (CloneNotSupportedException e) {
            return null;
        }
    }
}
```

Theater: Models one theater, including its configuration of rows of seats. This class creates a new theatre by taking *theaterID* (the unique name), *startTime* (the hour of this theatre is allowed to begin showing events, in [0,23]), *setupTime* (the number of minutes that it takes to clean this theatre after an event), and *seatingChart* (the layout of rows for the given seating chart) as input. This class can also add a row of seats, adds a specified feature to a specified seat, and can set/get the clean-up time.

- *addRow*: adds a row of numOfSeats plain-vanilla seats to the back of this theater
- *setSeatFeature*: adds a specified feature to a specified seat
- *getBlankSeatingChart*: returns a SeatingChart with all seats unreserved
- *getCleanupTime*: returns the cleanup time needed for the theater (in minutes)
- *setCleanupTime*: sets the cleanup time for the theater (in minutes)

```
public class Theater implements Serializable, Cloneable{
     private SeatingChart seatingChart;
     private String theaterID;
     private int cleanupTime;
     public Theater(String theaterID, int cleanupTime, SeatingChart seatingChart) {
        this.theaterID = theaterID;
        this.cleanupTime = cleanupTime;
        this.seatingChart = seatingChart;
     }
     public void addRow (String rowId, int numOfSeats)
        throws InvalidParameterException {
        try{
            seatingChart.addRowToBack(rowId, numOfSeats);
        }
        catch(InvalidParameterException e){
            System.out.println("Invalid parameter passed to addRow");
        }
    }
    public void setSeatFeature (String rowId, String seatId, String
                    featureLabel, Object featureValue) throws InvalidParameterException {
        try {
         seatingChart.setSeatFeature(rowId, seatId, featureLabel, featureValue);
        }
        catch(InvalidParameterException e) {
            System.out.println("Invalid parameter passed to setSeatFeature");
        }
    }
    public SeatingChart getBlankSeatingChart() {
        SeatingChart newSeatingChart = seatingChart.clone();
         return newSeatingChart;
    }
    public String getID() {
        return theaterID;
    }
    public int getCleanupTime() {
        return cleanupTime;
    }
    public void setCleanupTime(int cleanupTime) {
        this.cleanupTime = cleanupTime;
    }
```

```
        public void setTheaterID(String theaterID){
            this.theaterID = theaterID;
        }
         public void setSeatingChart(SeatingChart seatingChart){
            this.seatingChart = seatingChart;
        }
}
```

Movie: Models a movie. A movie screening may be the event at a *Show*. For now, the movie stores its title, director, actor(s), running time (in minutes), rating. It may need to store other attributes if required. This also provides a method for removing an actor from an ArrayList of actors. If the actor is in the ArrayList and is successfully removed then it returns true. If the actor is *not* in the ArrayList and is *not* removed it returns false. The GUI should display a message confirming that the actor was removed or tell them that the actor doesn't exist in this movie.

```
public class Movie implements Serializable{
    private String              title;
    private ArrayList<String>   actors;
    private String              director;
    private int                runtime;
    private String              rating;
    private String              screenFormat;
    public Movie(String title, String director, int runtime, String rating,
                                                                String screenFormat){
        this.title = title;
        this.director = director;
        this.runtime = runtime;
        this.rating = rating;
        this.screenFormat = screenFormat;
        actors = new ArrayList<String>();
    }
    public String getTitle() {
        return title;
    }
    public String getDirector() {
        return director;
    }
    public String getRating() {
        return rating;
    }
    public int getRuntime() {
        return runtime;
    }
    public String getScreenFormat() {
        return screenFormat;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public void setDirector(String director) {
        this.director = director;
    }
    public void setRating(String rating) {
        this.rating = rating;
    }
    public void setRuntime(int runtime) {
        this.runtime = runtime;
    }
    public void setScreenFormat(String screenFormat) {
        this.screenFormat = screenFormat;
    }
    public void addActor(String actor) {
        actors.add(actor);
    }
    public boolean removeActor(String actor) {
        int i=0;
        boolean removed = false;
        for(String actor1 : actors){
            if(actor.equals(actors.get(i))){
                actors.remove(i);
                removed = true;
                break;
            }
        }
       return removed;
    }
    public ArrayList<String> getActors() {
        return actors;
    }
}
```

CineGoer: Models a cinema goer, a customer of the Cinema. Human customers will interact with the software system. This class manages the user profiles and uses phone number as a default password for the user. Also, it provides a method for changing the password of the user.

```
public class CineGoer{
    private String firstName;
    private String lastName;
    private String phoneNumber;            // default password to log into their account
    private int seatNumber;                 // The Number of the seat that they reserved
    private String movieGoingToSee;     // The Movie they are going to see
    private String password;                // The password used to access the account
    private String username;
    private String address;
    private String town;
    private String state;
    private String zipcode;
    public CineGoer(String firstName, String lastName, String phoneNumber,
                                        String address, String town, String state, String zipcode){
        this.firstName = firstName;
        this.lastName = lastName;
        this.phoneNumber = phoneNumber;
        this.address = address;
        this.town = town;
        this.state = state;
        if(zipcode.length() == 5){
            this.zipcode = zipcode;
        }
        else
            this.zipcode = "00000";
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public String getPhoneNumber() {
        return phoneNumber;
    }
    public String getAddress(){
        return address;
    }
    public String getTown() {
        return town;
    }
    public String getState() {
        return state;
    }
    public String getZipcode() {
        return zipcode;
    }
    public int getSeatNumber(){
        return seatNumber;
    }
    public String getMovieGoingToSee() {
        return movieGoingToSee;
    }
     public void setMovieGoingToSee(String newMovie) {
        movieGoingToSee = newMovie;
    }
    public void getSeatNumber(int newSeatNumber) {
        seatNumber = newSeatNumber;
    }
    public boolean setPassword(String newPassword) {
        boolean accepted = false;
        if(newPassword.length() >= 6 && newPassword.length() <= 16) {
            password = newPassword;
            accepted = true;
        }
        return accepted;
     }
    public String getPassword() {
        return password;
    }
}
```

Show: Models a show. For now, we can assume that the event at all shows is the screening of a movie, in one of the theaters of the Cinema, at a given time.