Intermediate Java – Part 3

Yotam Avivi

Oct-Nov 2017

Java Interfaces

Interfaces define the methods exposed to the outside world (i.e. other objects) by the implementing class

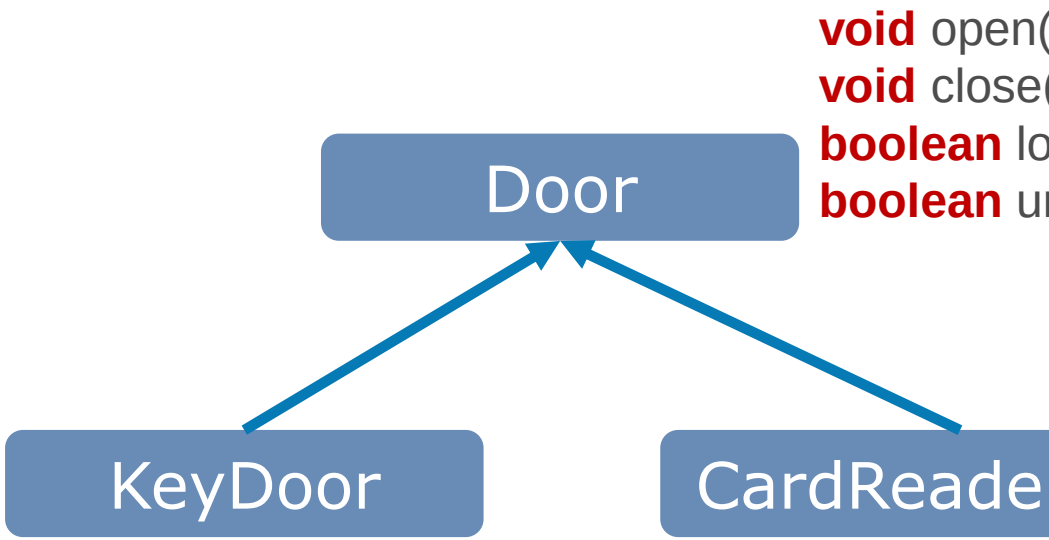
An Interface is like an abstract class whose all methods are abstract

Java Interfaces - Example

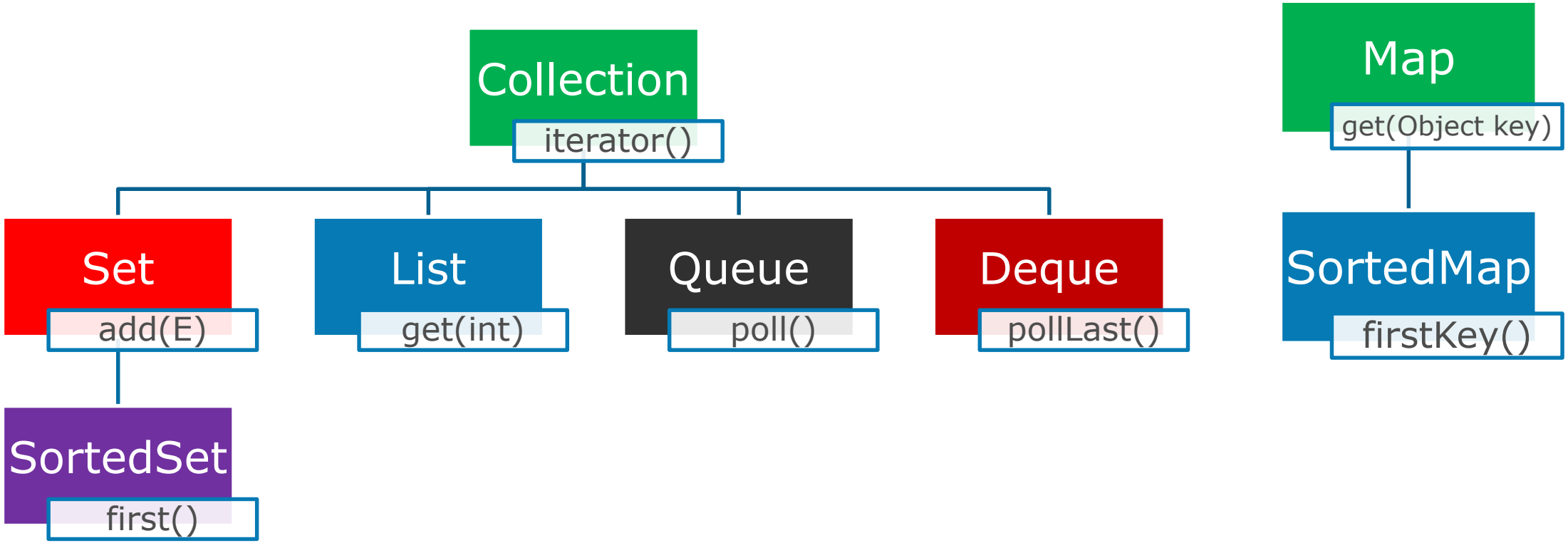
```
public interface Door {
  void open();
  void close();
  boolean lock(Key k);
  boolean unlock(Key k);
}
```

All Methods are Public

Java Interfaces - Example



Java Collections Interfaces



Java Collections (Cont.)

- Collection - The root of the collection hierarchy. A group of objects. May contain duplicates
- List - An ordered collection
- Set - A collection that cannot contain duplicate elements
- Queue - Typically, order elements in a FIFO (first-in, first-out) manner
- Map - Maps keys to values. Cannot contain duplicate keys

The Collection Interface

A Collection Represents a group of elements.

```
Collection<Person> persons = …;
persons.add(person)              //Adds a person to the collection
persons.contains(p)              //Look for a person that equals p
persons.isEmpty()                //Is the collection empty?
persons.addAll(persons2)         //Add all elements from persons2
persons.size()                   //How much elements in persons?
persons.clear()                  //Clear all elements from collection
persons.remove(element)          //Remove element from the collection
```

Iterating Over The Collection Interface (For Each Loop)

```
Collection<Person> persons = …;

for (Person p : persons) {
   logger.trace(“Saving person “ + p.getName());
   database.save(p);
}
```

The List Interface

List is an ordered sequence of elements. Here are some of its methods:

```
List<Person> persons = new ArrayList<Person>();
persons.remove(5);     //Removes the 5th person
persons.add(4, p);     //Inserts p at the 4th index. Shifts others forward
persons.indexOf(p)     //Returns the index of p in the list
persons.remove(7)      //removes the 7th element in the list. Shifts others
persons.set(3, p)      //Replaces the 3rd element with 'p'
```

A popular List implementation. Based on Array.
Other Impl: **LinkedList**

The Map Interface

A Map is an Object that maps keys to values. Here's an example:

```
Map<String,Person> personsById = new HashMap<String,Person>();

personsById.put("123456789", p);                    //Adds person with key=ID
Person p1 = personsById.get("123456789"); //Looks up a person whose
                                                    //ID="123456789"
```

The Queue Interface

Adds a new Job to Queue:

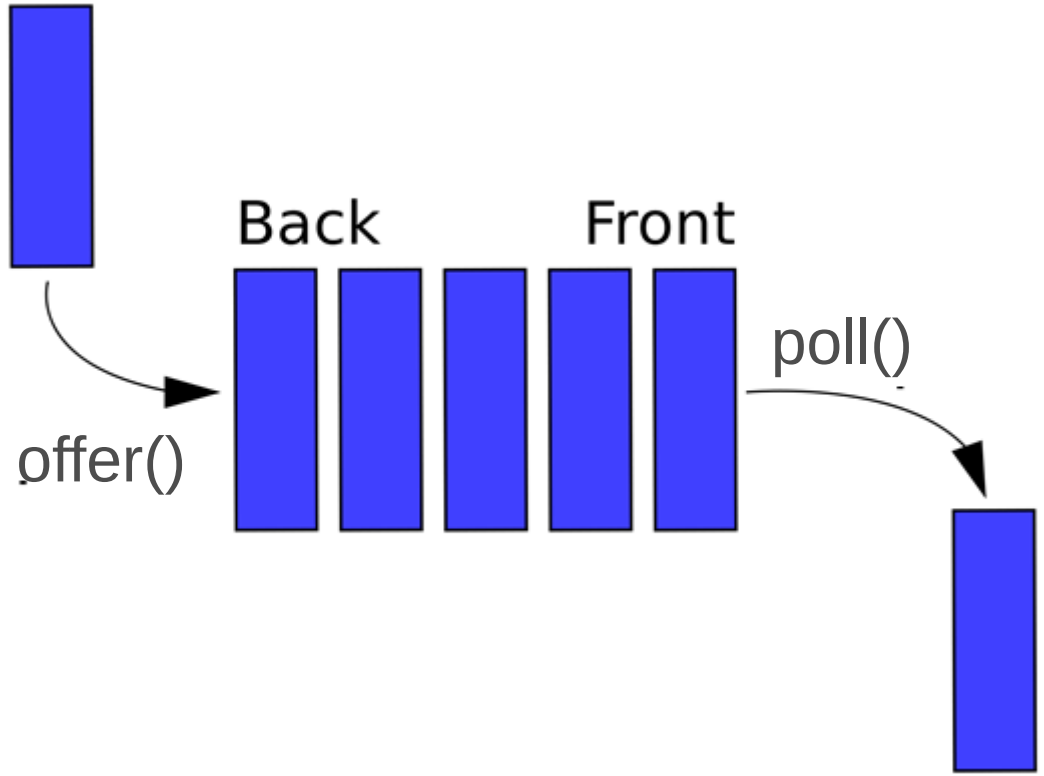
```
offer(job);
```

Peeks job at the head of queue:

```
peek(job);
```

Pulls out job at the head of queue:

```
Job job = poll();
```



Collections / Interfaces Exercise

```
public interface StudentsStore {

    boolean addStudent
    Student updateStudent
    Student deleteStudent
    Student getStudent
    List<Student> getAllStudents


}
```

Exercise 6 – Add the Order Class

Add the (Ticket) Order class to the Models package.

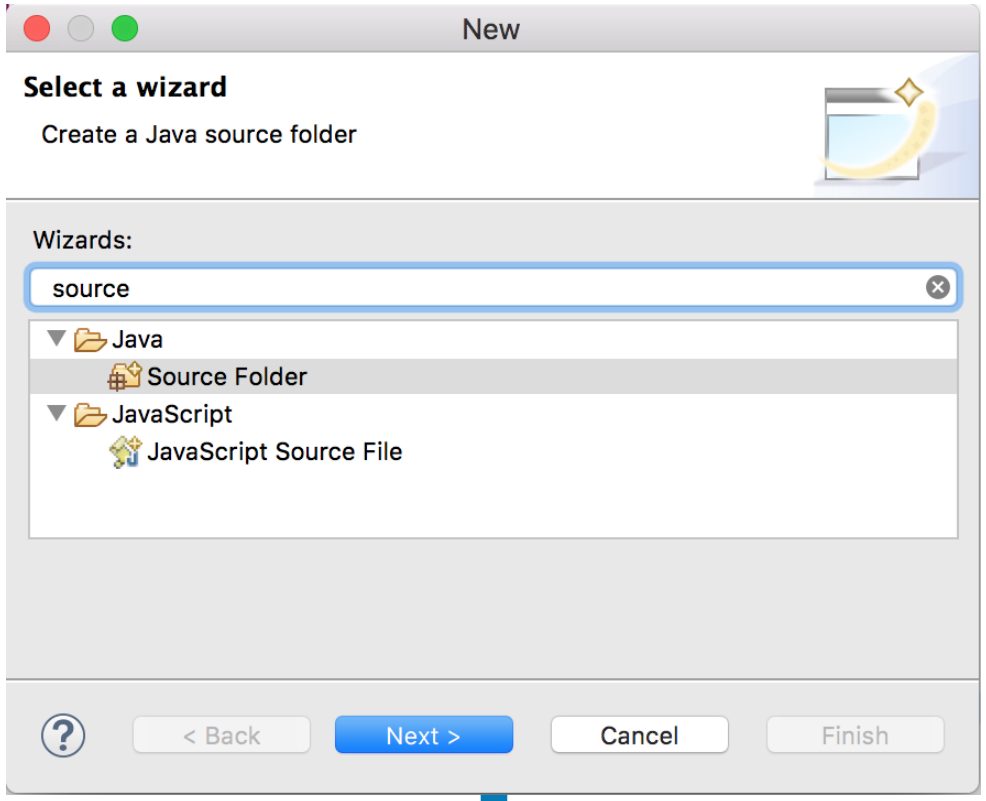
The Order class shall have:

1. An ID
2. A Reference to a MovieShow (to which movie show are the seats being ordered)
3. A list of seats that the user wishes to order (choose an appropriate collection).
4. Two Extra methods (**LEAVE EMPTY, DON'T IMPLEMENT YET**):
    1. boolean addSeat(Seat) → for adding a seat to the order*
    2. double getTotalCost() → for calculating the sub-total order cost**

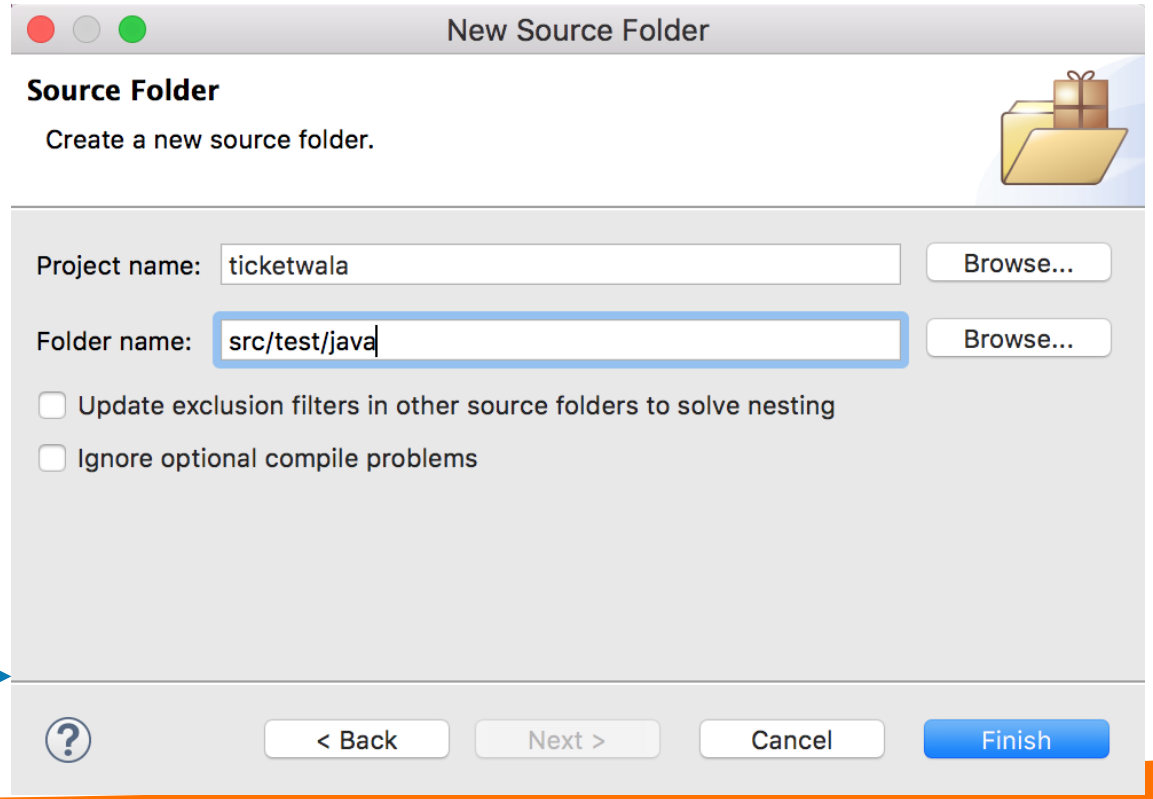
* Leave the method unimplemented by returning false all the time

** Leave the method unimplemented by returning -1 all the time

Testing the Order Object with JUNIT (1)

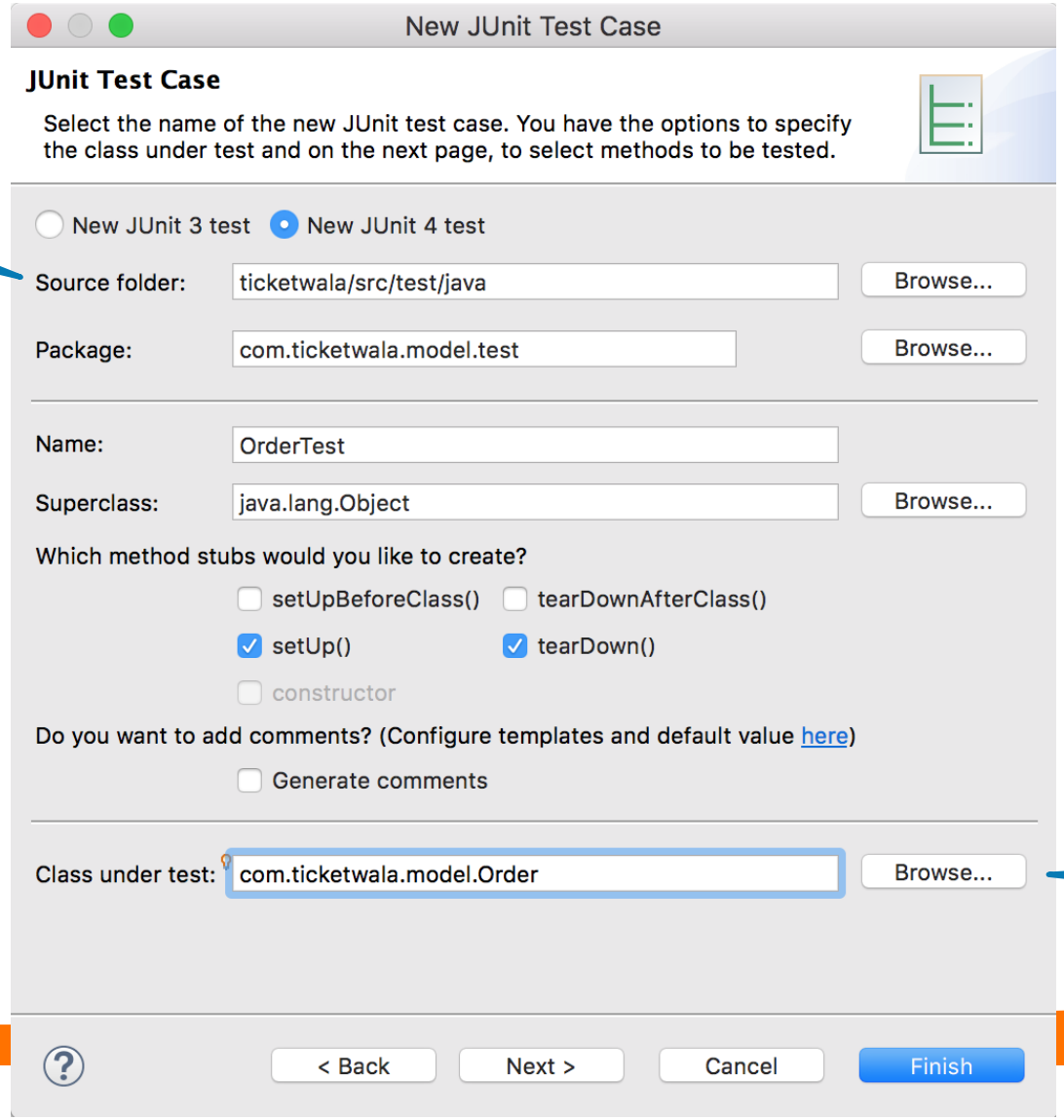


First, we should create a new source folder for our tests



Testing the Order Object with JUNIT (2)

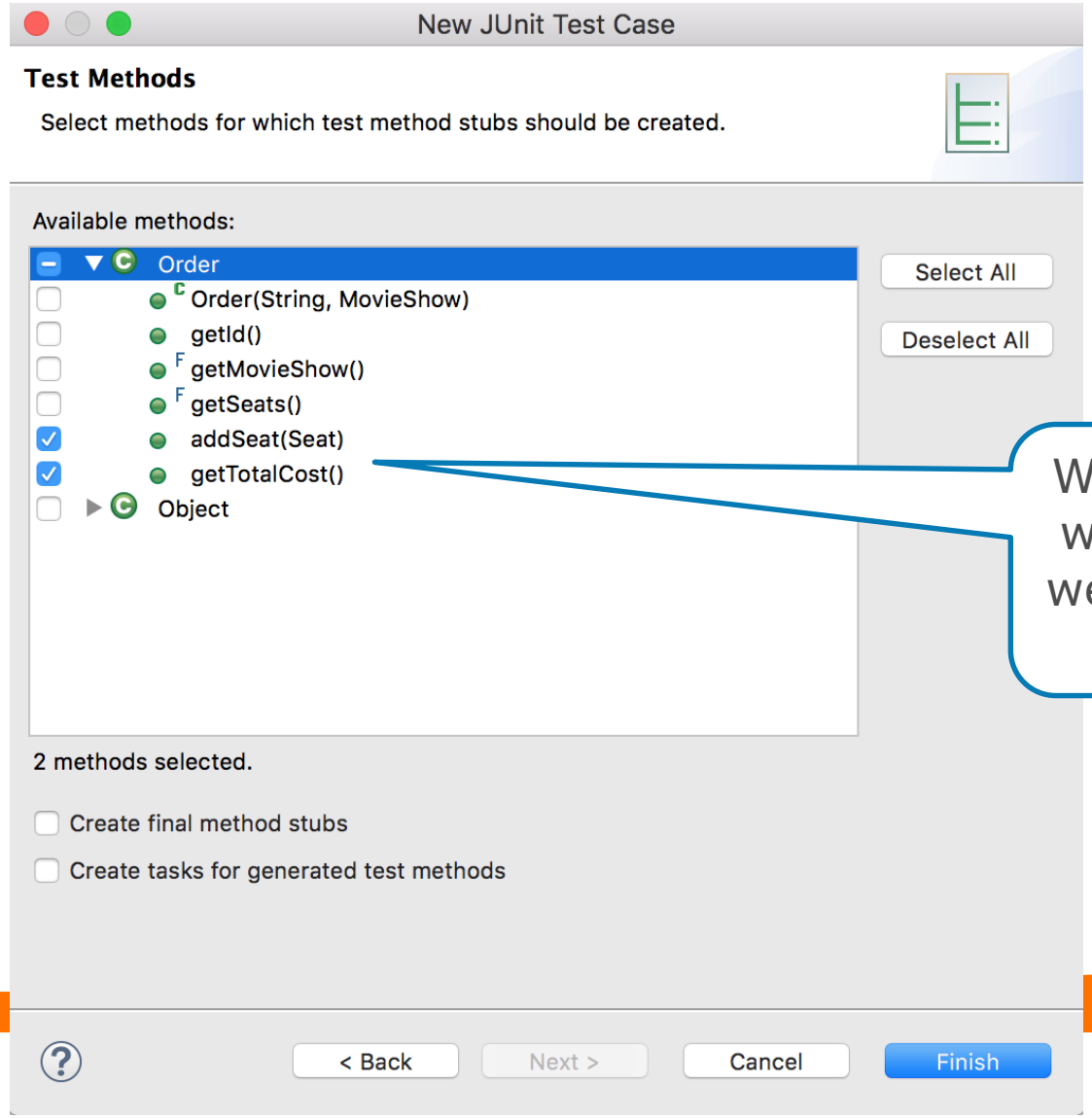
Make sure to choose the test folder



Next, we should create the Test Class

We can indicate which class we would like to test

Testing the Order Object with JUNIT (3)



We can indicate which methods we would like to test

Exercise 7 – Implement the Order.addSeat() Method

Implement the method *Order.addSeat(Seat)* that you created in exercise 6 and test it with JUNIT.

You can use the next slide as an example on how to test it.

Note: The same seat cannot be added twice to an order.
Hint: you can override the Seat.equals() method and use the collection contains() method.

Test Order.addSeat() with JUNIT

```
@Test
public void testAddSeat() {
    Order order = createOrder(); //A utility method for creating an order

    //Add 1 seat to order
    boolean success = order.addSeat(new Seat(1, 1, 30.0));

    //Assert on success and only 1 seat exist
    Assert.assertTrue(success && order.getSeats().size() == 1);

    //Add same seat (notice its a new object but same seat location - 1,1)
    success = order.addSeat(new Seat(1, 1, 30.0));
    Assert.assertTrue(!success && order.getSeats().size() == 1);

    //Add another seat and verify a success
    success = order.addSeat(new Seat(1, 2, 30.0));
    Assert.assertTrue(success && order.getSeats().size() == 2);
}
```

Adding same seat should fail.

Size still == 1

These conditions MUST be true for the test to pass

Adding different seat should succeed.

Size now == 2

Exercise 8 – Implement the Order.getTotalCost() Method in TDD Methodology

Implement the method *Order.getTotalCost()* that you created in exercise 6 and test it with JUNIT.

This time, write the test with TDD methodology:

1. Write a single test first
2. Run the test – make sure it fails
3. Implement the code
4. Run the test **again** – make sure it succeeds
5. Repeat 1-4 until done

You can use the next slide as an example on how to test the method

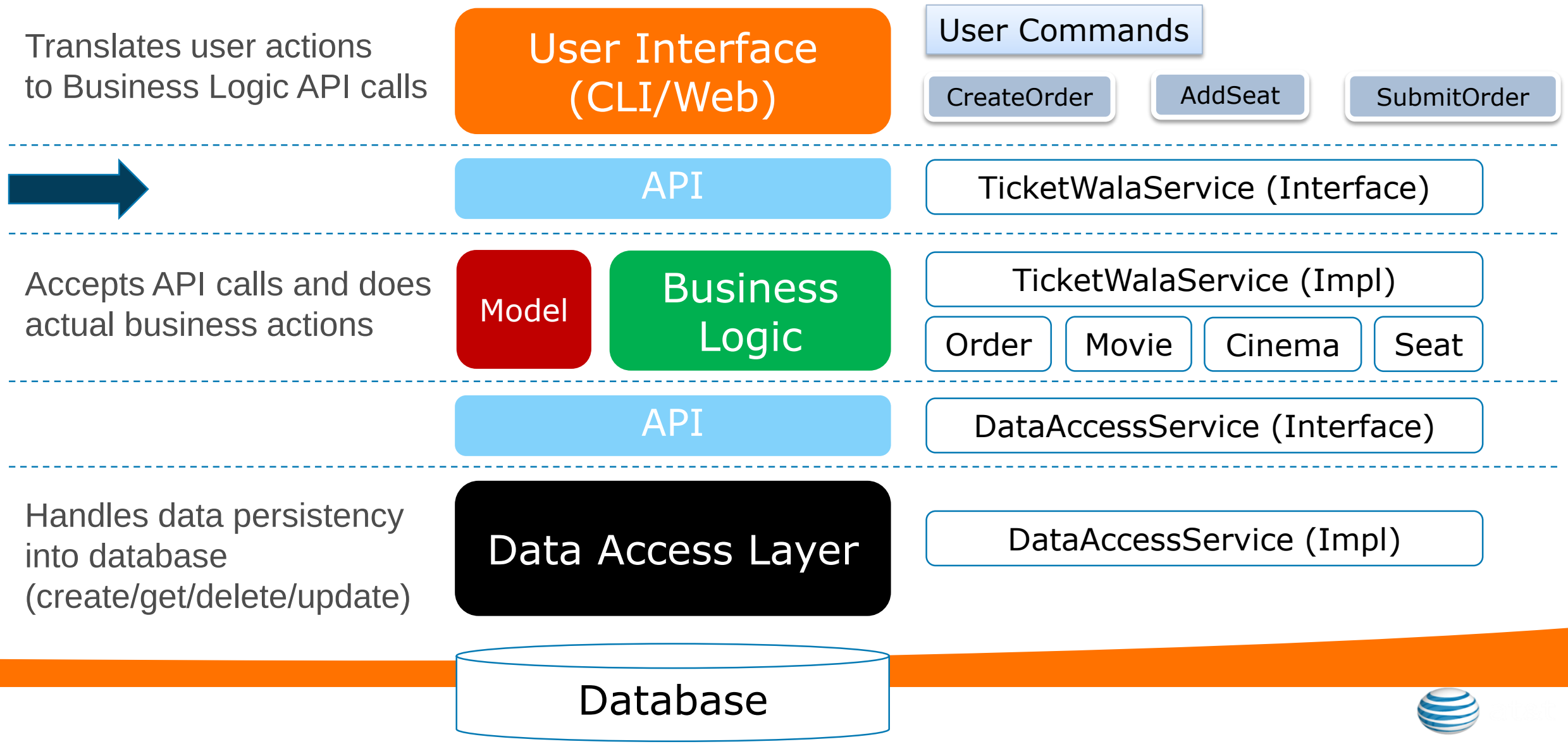
Test Order.getTotalCost with JUNIT

```
@Test
public void testGetTotalCost() {
    Order order = createOrder();

    Assert.assertTrue(order.getTotalCost() == 0);

    order.addSeat(new Seat(1, 1, 30.0));
    order.addSeat(new Seat(1, 2, 30.0));
    order.addSeat(new Seat(1, 3, 30.0));
    Assert.assertTrue(order.getTotalCost() == 90);
}
```

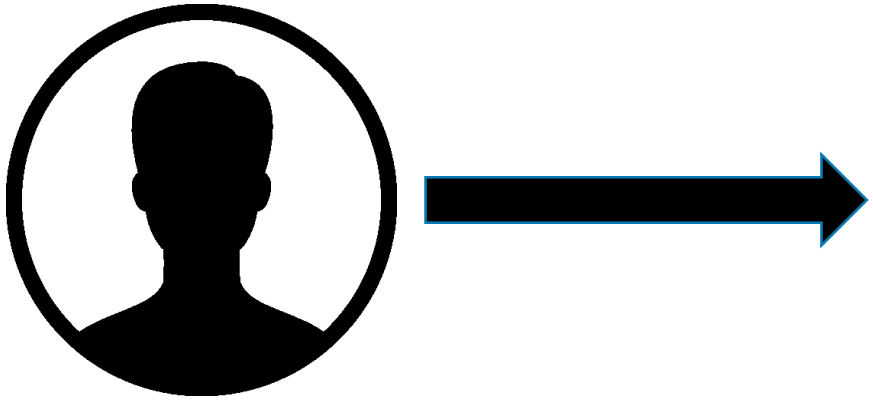
Testing the getTotalCost() method

Test an empty order first and then a non-empty order

Project High Level Design in Layers



The TicketWalaService Interface

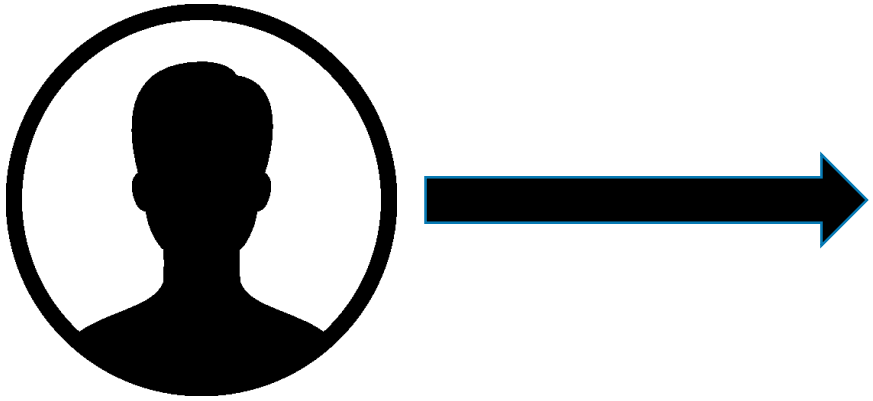


User

{ TicketWala API }

- Get All Movie Shows
- Get (Single) Movie Show
- Create a new Order (for a Movie Show)
- Add a Seat Ticket to an Order
- Submit Order

The TicketWalaService (Admin) Interface



Admin

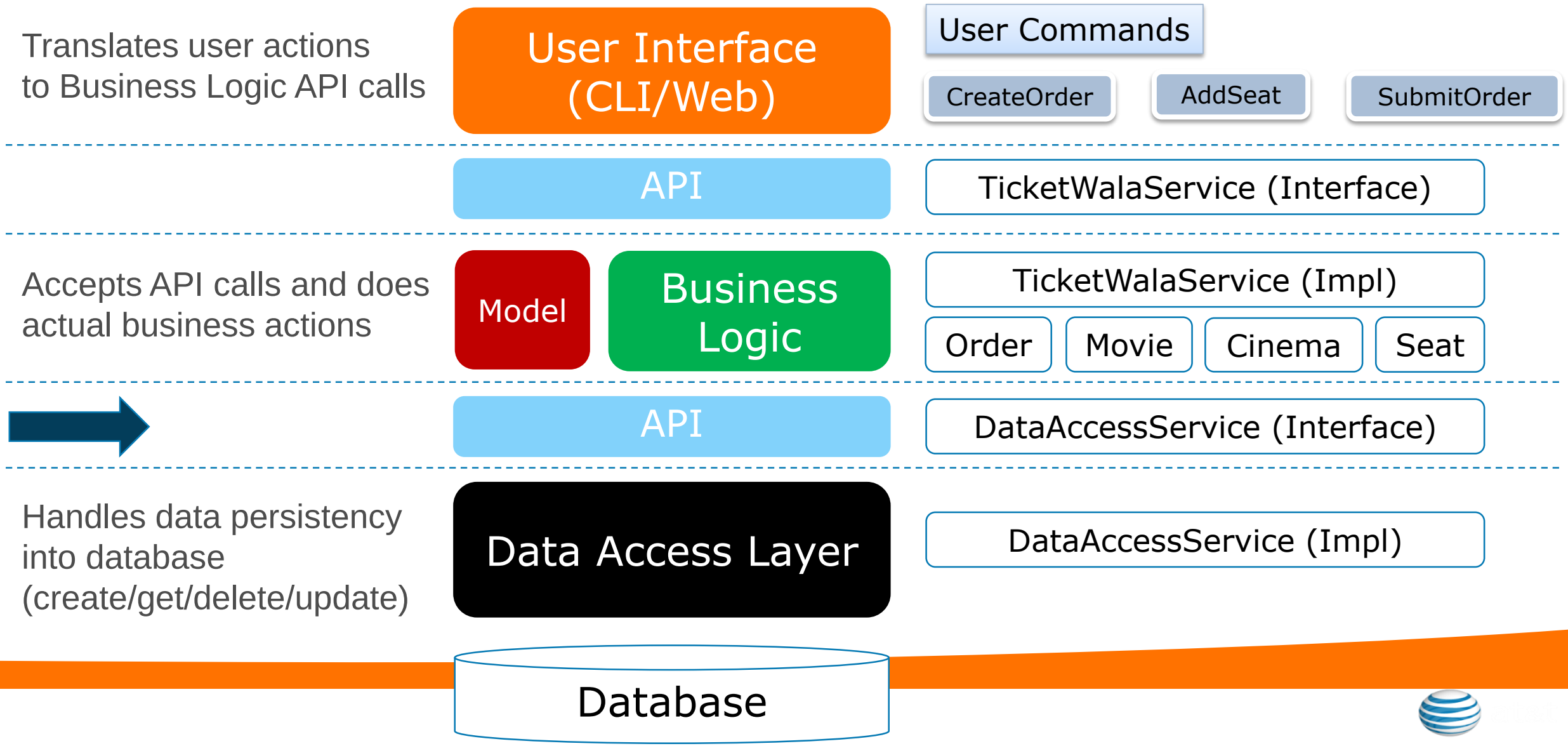
{ TicketWala (Admin) API }

- Create Movie Show
- Delete Movie Show

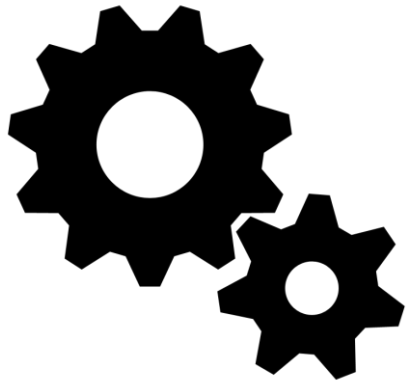
Exercise 8 – Create & Implement the TicketWalaService

1. Write the TicketWalaService Interface (for both User and Admin)
   Use a dedicated API package for the service interface
   For example:
   com.ticketwala.service.api
2. Create an **empty** Implementation for the Service
   Use a dedicated Impl package for the service.
   For example:
   com.ticketwala.service.impl

Project High Level Design in Layers



The DataAccessService Interface



{ Application Business Logic }

{ Data Access Service API }

- Create Movie Show
- Find Movie Show
- Delete Movie Show
- Update Movie Show
- Get All Movie Shows
- Delete All Movie Shows
- Commit Ticket Order

Exercise 9 – Create & Implement the DataAccessService

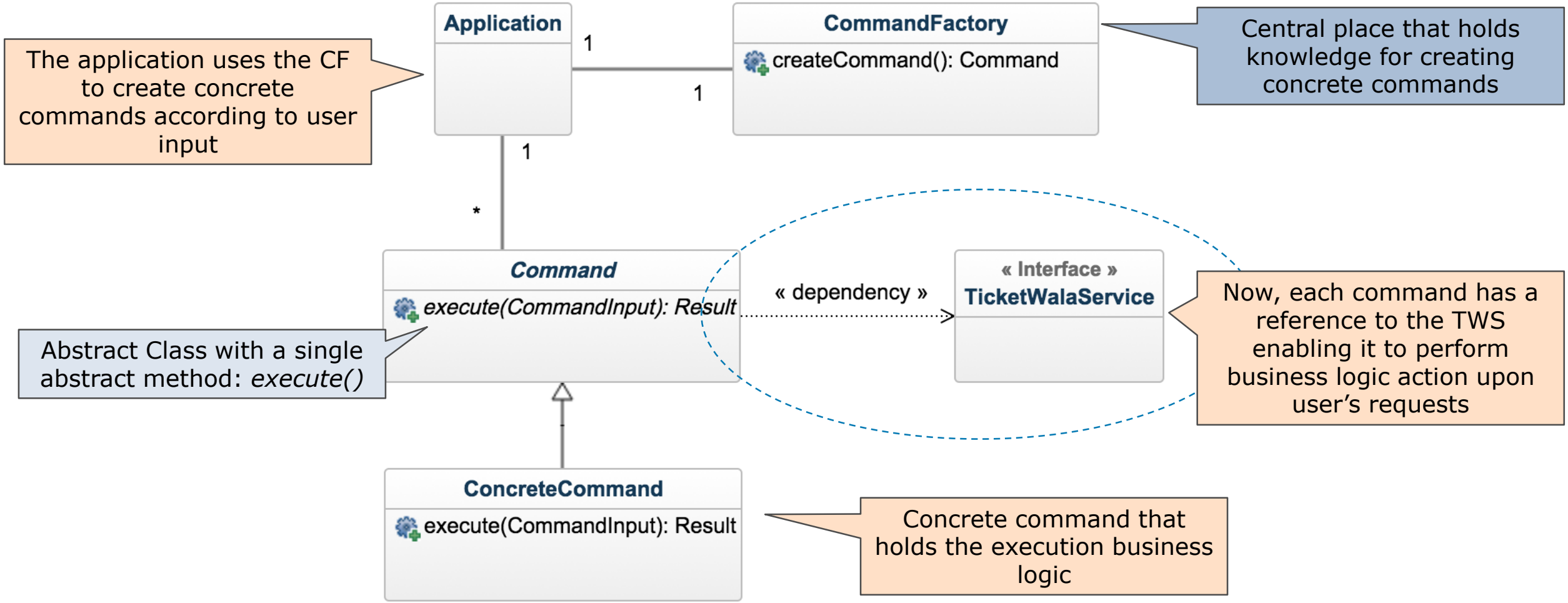
1. Write the DataAccessService
   Use a dedicated API package for the service interface
   For example:
   com.ticketwala.dao.api (dao=data-access-object)
2. Create an **empty** Implementation for the Service
   Use a dedicated Impl package for the service.
   For example:
   com.ticketwala.dao.impl

Exercise 10 - Implement TicketWalaServiceImpl

- Implement all user facing methods:

1. getMovieShows
2. getMovieShow
3. createOrder
4. addSeatTicket
5. submitOrder

Integrate CLI Commands with out TWS Service



Integrate CLI Commands with TWS Service (Cont.)

```
public abstract class Command {

    protected String name;
    protected Object commandInput;
    protected TicketWalaService ticketWalaService;

    public Command(Object commandInput, TicketWalaService tws) {
        this.commandInput = commandInput;
        this.ticketWalaService = tws;
    }

    public abstract Result execute();
}
```

One way to integrate our commands with TWS is to pass it via the constructor and hold it as a data-member

An example on how to create a new Concrete Command

```
orderId = commandArray.get(1);
seatRow = Integer.parseInt(commandArray.get(2));
seatNumber = Integer.parseInt(commandArray.get(3));
AddSeatCommandInput asi = new AddSeatCommandInput(orderId, seatRow, seatNumber);
result = new AddSeatCommand(asi, tws);
```

Exercise 11 - Implement all user facing commands:

Implement all user facing Commands:

1. create-order [movieShowId]
2. add-seat [orderId] [row] [seatNumber]
3. submit-order [orderId]