Mobile Store Management

**Grade settings**: Maximum grade: 100  
**Disable external file upload, paste and drop external content**: Yes  
**Run**: Yes **Evaluate**: Yes  
**Automatic grade**: Yes

[**Click here to download the code template**](https://cognizant.tekstac.com/pluginfile.php/54469/mod_vpl/intro/MobileManagement.zip?time=1614834810663)

iMobile Mobile store sells mobiles of different brands.    Due to the increased demand, they have approached Zee Software to automate their various requirements.

You are required to write Junit test case and check the correctness of the application developed.

**Functional Requirements:**

The application has the below classes and methods implemented.

You are provided with a model class Mobile

**Component Specification: Mobile** **(Model Class)**

|  |  |  |
| --- | --- | --- |
| **Type(Class)** | **Attributes** | **Methods** |
| Mobile | String IMEINumber  String mobileType  String brand  String model  double mobilePrice | Necessary getters,setters are provided  A Constructor is also provided |

·         Here, mobileType can take a value either “SmartPhone” or “Basic” [Note: Values are case insensitive]

·         There can be many models for the given brand.

o    For Example: Galaxy S10,  Galaxy F41 are few models of the brand Samsung

**Component Specification:**InvalidMobileException**(This class inherits the Exception Class)**

|  |  |
| --- | --- |
| **Type(Class)** | **Methods** |
| InvalidMobileException | Provided with a single argument constructor – InvalidMobileException(String message) |

The below are the requirements  implemented in the Utility class for which JUnit test cases are to be written and tested.

**Component Specification:**MobileStore **(Utility Class)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Component Name** | **Type (Class)** | **Methods** | **Responsibilities** | **Exception** |
| Validate the mobile  type | MobileStore | public boolean validateMobileType (String mobileType) | Validate the mobileType.  If valid, return true , else this method should throw a user defined exception | Throw a user defined exception “InvalidMobileException”  if the mobileType Is neither “SmartPhone” or “Basic” |
| View Mobile based on IMEINumber | MobileStore | public Mobile viewMobile(List<Mobile> mobileList,String IMEINumber) | This method should return the Mobile object with the IMEINumber passed as parameter from list of  mobiles,which is also passed as parameter.  If the mobileList is empty or if there is no mobile with the given IMEINumber it should throw a user defined exception | Throw a user defined exception “InvalidMobileException” if the  mobileList is empty or if  no mobile exists with the given IMEINumber. |
| View the list of Mobiles for a given brand | MobileStore | public List<Mobile> viewMobilesByBrand (List<Mobile> mobileList ,String brand) | This method takes the mobileList and a brand as an argument. It should return the list of Mobiles for the given brand. If the mobileList is empty it should throw a user defined exception. | Throw a user defined exception “InvalidMobileException” if the  mobileList is empty |
| View all the Mobiles brand wise | MobileStore | public Map<String,List<Mobile> > viewMobilesBrandWise(List<Mobile> mobileList) | This method takes the mobileList as argument. It should return brandwise, all the mobiles in the list. The return type is map, where the brand is key and value is the List of Mobiles  belonging to that brand.  If the mobileList is empty it should throw a user defined exception. | Throw a user defined exception “InvalidMobileException” if the  mobileList is empty. |
| View the number of mobiles for each brand within the given price range | MobileStore | public  Map<String,Integer> countTotalModelsForEachBrand(List<Mobile> mobileList, double priceRange) | This method should return the number of Mobiles for each brand in the mobileList for the given priceRange(count of mobiles having price less than or equal to the priceRange). It takes the mobileList, priceRange  as arguments and returns a Map with key as brand and value as count of mobile within the given price range.                         If the mobileList is empty it should throw a user defined exception. | Throw a user defined exception “InvalidMobileException” if the  mobileList is empty. |

You need to write Junit test for the MobileStore class.

**Testing Scenarios:**

You are provided with a class  “MobileStoreTest”  to do this testing.

**Note:**

To perform testing, the mobileList should contain objects of Mobile.

To do this, in MobileStoreTest   class you are provided with a setup method.  Use this method to populate the static variable mobileList  in MobileStoreTest   class.  That is, create few objects for  Mobile and populate the mobileList given in MobileStoreTest  class with these objects and use that list to test the methods  in MobileStore class that needs a Mobile list to be passed as attribute.

The below are the test methods to be implemented in MobileStoreTest class.

|  |  |
| --- | --- |
| **Test Method** | **Scenarios / Responsibilities** |
| test11ValidateMobileTypeWhenSmartPhone | This method should test the validateMobileType method when “SmartPhone” is passed as parameter |
| test12ValidateMobileTypeWhenBasic | This method should test the validateMobileType method when “Basic” is passed as parameter. |
| test13ValidateMobileTypeWhenInvalid | This method should test the validateMobileType method when invalid value is passed as parameter  validateMobileType is expected  to throw InvalidMobileException when type is invalid.  Write JUnit to test for the exception thrown  either by using appropriate annotation or by using try catch block. |
| test14ViewMobileForValidIMEINumber | This method should test the correctness of  viewMobile method for  an existing IMEINumber.  Perform testing for the correctness of the value returned. |
| test15ViewMobileForInvalidIMEINumber | This method should test the correctness of  viewMobile method for a non existing IMEINumber.  Perform testing for the correctness of the value returned.  viewMobile method is expected  to throw InvalidMobileException when the given IMEI Number does not exist.  Write JUnit to test for the exception thrown  either by using appropriate annotation or by using try catch block |
| test16ViewMobilesByBrand | This method should test the correctness of  viewMobilesByBrand method.  Perform testing for the correctness of the value returned. |
| test17ViewMobilesByBrandForEmptyList | This method should test the correctness of viewMobilesByBrand method for an empty mobileList.  viewMobilesByBrand method is expected  to throw InvalidMobileException when mobileList is empty.  Write JUnit to test for the exception thrown  either by using appropriate annotation or by using try catch block |
| test18ViewMobilesBrandWise | This method should test the correctness of viewMobilesBrandWise method.  Perform testing for the correctness of the value returned. |
| test19ViewMobilesBrandWiseForEmptyList | This method should test the correctness of viewMobilesBrandWise method for an empty mobileList.  viewMobilesBrandWise method is expected  to throw InvalidMobileException when mobileList is empty.  Write JUnit to test for the exception thrown  either by using appropriate annotation or by using try catch block |
| test20CountTotalModelsForEachBrand | This method should test the correctness of countTotalModelsForEachBrand method.  Perform testing for the correctness of the value returned. |
| test21CountTotalModelsForEachBrandForEmptyList | This method should test the correctness of countTotalModelsForEachBrand method for an empty mobileList.  countTotalModelsForEachBrand method is expected  to throw InvalidMobileException when mobileList is empty.  Write JUnit to test for the exception thrown  either by using appropriate annotation or by using try catch block |

Implement the test methods and provide the needed annotation to all the methods in MobileStoreTest class.

Also this class is provided with  the annotation, so that the test methods are executed in ascending order of the test method names.

You are provided with a Main class with the main method to check the correctness of the test methods written in MobileStoreTest class.

Having completed writing the test methods, uncomment the code in Main class and execute the main method.

### **Automatic evaluation**[**[-]**](javascript:void(0);)

**Proposed grade: 100.0 / 100**  
**Result Description**  
[[-]](javascript:void(0);)**Grading and Feedback**

*Writing JUnit for a Utility class - 60.0 / 60.0(Success)*

*Test Coverage - 40 / 40.0(Success)*

Test Case Passed

#### **MobileManagement/src/com/mobile/exception/InvalidMobileException.java**

1 *package* com.mobile.exception;

2

3 *public* *class* InvalidMobileException *extends* Exception {

4

5 *public* InvalidMobileException(String message) {

6 *super*(message);

7 }

8 }

9

10

#### **MobileManagement/src/com/mobile/model/Mobile.java**

1 *package* com.mobile.model;

2

3

4 *public* *class* Mobile {

5 *private* String IMEINumber;

6 *private* String mobileType;

7 *private* String brand;

8 *private* String model;

9 *private* *double* mobilePrice;

10

11

12 *public* Mobile(){

13

14 }

15

16

17 *public* Mobile(String IMEINumber, String mobileType, String brand, String model, *double* mobilePrice) {

18 *super*();

19 *this*.IMEINumber = IMEINumber;

20 *this*.mobileType = mobileType;

21 *this*.brand = brand;

22 *this*.model = model;

23 *this*.mobilePrice = mobilePrice;

24 }

25

26 *public* String getIMEINumber() {

27 *return* IMEINumber;

28 }

29 *public* *void* setIMEINumber(String iMEINumber) {

30 IMEINumber = iMEINumber;

31 }

32

33 *public* String getMobileType() {

34 *return* mobileType;

35 }

36 *public* *void* setMobileType(String mobileType) {

37 *this*.mobileType = mobileType;

38 }

39

40 *public* String getBrand() {

41 *return* brand;

42 }

43 *public* *void* setBrand(String brand) {

44 *this*.brand = brand;

45 }

46

47 *public* String getModel() {

48 *return* model;

49 }

50 *public* *void* setModel(String model) {

51 *this*.model = model;

52 }

53

54 *public* *double* getMobilePrice() {

55 *return* mobilePrice;

56 }

57 *public* *void* setMobilePrice(*double* mobilePrice) {

58 *this*.mobilePrice = mobilePrice;

59 }

60

61 }

62

#### **MobileManagement/src/com/mobile/skeleton/SkeletonValidator.java**

1 *package* com.mobile.skeleton;

2

3 *import* java.lang.reflect.Method;

4 *import* java.util.logging.Level;

5 *import* java.util.logging.Logger;

6

7 /\*\*

8 \* This class is used to verify if the Code Skeleton is intact

9 \* and not modified by participants thereby ensuring smooth auto

10 \* evaluation

11 \*/

12

13 *public* *class* SkeletonValidator {

14 *public* SkeletonValidator() {

15 validateClassName("com.mobile.util.MobileStore");

16 validateClassName("com.mobile.model.Mobile");

17 validateClassName("com.mobile.exception.InvalidMobileException");

18 validateClassName("com.mobile.test.MobileStoreTest");

19

20 validateMethodSignature(

21 "getBrand:java.lang.String,setIMEINumber:void,getModel:java.lang.String,setMobilePrice:void,setModel:void,setBrand:void,getMobileType:java.lang.String,getMobilePrice:double,setMobileType:void,getIMEINumber:java.lang.String",

22 "com.mobile.model.Mobile");

23 validateMethodSignature(

24 "validateMobileType:boolean,countTotalModelsForEachBrand:java.util.Map,viewMobilesBrandWise:java.util.Map,viewMobilesByBrand:java.util.List,viewMobile:com.mobile.model.Mobile",

25 "com.mobile.util.MobileStore");

26 validateMethodSignature(

27 "test11ValidateMobileTypeWhenSmartPhone:void,test12ValidateMobileTypeWhenBasic:void,test13ValidateMobileTypeWhenInvalid:void,test14ViewMobileForValidIMEINumber:void,test15ViewMobileForInvalidIMEINumber:void,test16ViewMobilesByBrand:void,test17ViewMobilesByBrandForEmptyList:void,test18ViewMobilesBrandWise:void,test19ViewMobilesBrandWiseForEmptyList:void,test20CountTotalModelsForEachBrand:void,test21CountTotalModelsForEachBrandForEmptyList:void",

28 "com.mobile.test.MobileStoreTest");

29

30 }

31

32 *private* *static* *final* Logger LOG = Logger.getLogger("SkeletonValidator");

33

34 *protected* *final* *boolean* validateClassName(String className) {

35

36 *boolean* iscorrect = *false*;

37 *try* {

38 Class.forName(className);

39 iscorrect = *true*;

40 LOG.info("Class Name " + className + " is correct");

41

42 } *catch* (ClassNotFoundException e) {

43 LOG.log(Level.SEVERE, "You have changed either the " + "class name/package. Use the correct package "

44 + "and class name as provided in the skeleton - "+className);

45

46 } *catch* (Exception e) {

47 LOG.log(Level.SEVERE,

48 "There is an error in validating the " + "Class Name. Please manually verify that the "

49 + "Class name is same as skeleton before uploading");

50 }

51 *return* iscorrect;

52 }

53

54 *protected* *final* *void* validateMethodSignature(String methodWithExcptn, String className) {

55 Class cls = *null*;

56 String methodName = *null*;

57 *try* {

58

59 String[] actualmethods = methodWithExcptn.split(",");

60 *boolean* errorFlag = *false*;

61 String[] methodSignature;

62

63 String returnType = *null*;

64

65 *for* (String singleMethod : actualmethods) {

66 *boolean* foundMethod = *false*;

67 methodSignature = singleMethod.split(":");

68

69 methodName = methodSignature[0];

70 returnType = methodSignature[1];

71 cls = Class.forName(className);

72 Method[] methods = cls.getMethods();

73 *for* (Method findMethod : methods) {

74 *if* (methodName.equals(findMethod.getName())) {

75 foundMethod = *true*;

76 *if* (!(findMethod.getReturnType().getName().equals(returnType))) {

77 errorFlag = *true*;

78 LOG.log(Level.SEVERE, " You have changed the " + "return type in '" + methodName

79 + "' method. Please stick to the " + "skeleton provided");

80

81 } *else* {

82 LOG.info("Method signature of " + methodName + " is valid");

83 }

84

85 }

86 }

87 *if* (!foundMethod) {

88 errorFlag = *true*;

89 LOG.log(Level.SEVERE, " Unable to find the given public method " + methodName

90 + ". Do not change the " + "given public method name. " + "Verify it with the skeleton");

91 }

92

93 }

94 *if* (!errorFlag) {

95 LOG.info("Method signature is valid");

96 }

97

98 } *catch* (Exception e) {

99 LOG.log(Level.SEVERE,

100 " There is an error in validating the " + "method structure. Please manually verify that the "

101 + "Method signature is same as the skeleton before uploading. Class Name : "+className+", method name : "+methodName);

102 }

103 }

104

105 }

106

#### **MobileManagement/src/com/mobile/test/Main.java**

1 *package* com.mobile.test;

2

3 *import* org.junit.runner.JUnitCore;

4 *import* org.junit.runner.Result;

5

6 *import* com.mobile.skeleton.SkeletonValidator;

7

8 *public* *class* Main {

9

10 *public* *static* *void* main(String[] args) {

11

12 SkeletonValidator validator = *new* SkeletonValidator();

13

14 Result result = JUnitCore.runClasses(MobileStoreTest.*class*);

15 *if* (result.getFailureCount() == 0) {

16 System.out.println("All Test cases Cleared");

17 } *else* {

18 System.out.println("One or more test case(s) failed");

19 System.out.println("===============================");

20 result.getFailures().forEach(x -> System.out.println(x.getMessage()));

21 }

22

23 }

24

25 }

26

#### **MobileManagement/src/com/mobile/test/MobileStoreTest.java**

1 *package* com.mobile.test;

2

3 *import* *static* org.junit.Assert.assertTrue;

4 *import* *static* org.junit.Assert.assertFalse;

5

6 *import* java.util.ArrayList;

7 *import* java.util.List;

8 *import* java.util.Map;

9 *import* java.util.Set;

10

11 *import* org.junit.Assert;

12

13 *import* org.junit.Test;

14

15

16 *import* org.junit.BeforeClass;

17 *import* org.junit.FixMethodOrder;

18 *import* org.junit.runners.MethodSorters;

19

20 *import* com.mobile.model.Mobile;

21 *import* com.mobile.util.MobileStore;

22 *import* com.mobile.exception.InvalidMobileException;

23

24 @FixMethodOrder(MethodSorters.NAME\_ASCENDING)

25 *public* *class* MobileStoreTest {

26

27 *private* *static* List<Mobile> mobileList = *new* ArrayList<Mobile>();

28 *private* *static* MobileStore mobileStoreObj;

29

30

31 @BeforeClass

32 *public* *static* *void* setUp() throws Exception {

33

34 mobileStoreObj = *new* MobileStore();

35

36 //Create few objects for Mobile class and add to mobileList.

37 //Use that list to test all the methods in MobileTest class that requires a list of Mobile

38

39 mobileList.add(*new* Mobile("Samsung1234","Basic","Samsung","S102",5000.0));

40 mobileList.add(*new* Mobile("Nokia1234","Basic","Nokia","N12",6500.0));

41 mobileList.add(*new* Mobile("samsung5678","SmartPhone","Samsung","SP123",50000.0));

42 mobileList.add(*new* Mobile("Samsung6789","Basic","Samsung","SB12",4500.0));

43 mobileList.add(*new* Mobile("Nokia2468","Basic","Nokia","N14",3500.0));

44 mobileList.add(*new* Mobile("Nokia1357","SmartPhone","Nokia","N104",20000.0));

45

46 }

47

48 //Test the validateMobileType method when the value is SmartPhone

49 @Test

50 *public* *void* test11ValidateMobileTypeWhenSmartPhone() throws InvalidMobileException{

51

52 //fill code

53 assertTrue(mobileStoreObj.validateMobileType("SmartPhone"));

54

55 }

56

57 //Test the validateMobileType method when the value is Basic

58 @Test

59 *public* *void* test12ValidateMobileTypeWhenBasic() throws InvalidMobileException {

60

61 //fill code

62 assertTrue(mobileStoreObj.validateMobileType("Basic"));

63

64 }

65

66 //Test the validateMobileType method when the value is Invalid

67 @Test(expected=InvalidMobileException.*class*)

68 *public* *void* test13ValidateMobileTypeWhenInvalid() throws InvalidMobileException{

69

70 //fill code

71 mobileStoreObj.validateMobileType("touch");

72 }

73

74 //Test the viewMobile method for a valid IMEINumber

75 @Test

76 *public* *void* test14ViewMobileForValidIMEINumber() throws InvalidMobileException{

77

78 //fill code

79 //assertTrue(mobileStoreObj.viewMobile(mobileList,"Samsung1234"));

80 assertTrue(mobileStoreObj.viewMobile(mobileList,"Samsung1234").getIMEINumber()=="Samsung1234");

81

82 }

83

84 //Test the viewMobile method for invalid IMEINumber

85 @Test(expected=InvalidMobileException.*class*)

86 *public* *void* test15ViewMobileForInvalidIMEINumber() throws InvalidMobileException{

87

88 //fill code

89 mobileStoreObj.viewMobile(mobileList,"12345678");

90 }

91

92 //Test the viewMobilesByBrand method

93 @Test

94 *public* *void* test16ViewMobilesByBrand() throws InvalidMobileException{

95

96 //fill code

97 List<Mobile>m1=mobileStoreObj.viewMobilesByBrand(mobileList,"Nokia");

98 assertTrue(m1.get(0).getBrand()=="Nokia");

99

100 }

101

102 //Test the viewMobilesByBrand method for empty list

103 @Test(expected=InvalidMobileException.*class*)

104 *public* *void* test17ViewMobilesByBrandForEmptyList() throws InvalidMobileException {

105

106 //fill code

107 ArrayList<Mobile>m=*new* ArrayList<>();

108 mobileStoreObj.viewMobilesByBrand(m,"nnn");

109

110 }

111

112 //Test the viewMobilesBrandWise method

113 @Test

114 *public* *void* test18ViewMobilesBrandWise() throws InvalidMobileException {

115

116 //fill code

117 //Map<StringObj.viewMobilesBrandWise(mobileList);

118 //Set<String>Key=m2.keySet();

119 assertFalse(mobileStoreObj.viewMobilesBrandWise(mobileList).isEmpty());

120

121

122 }

123 *private* *void* assertFalse(*boolean* empty){

124 //TODO Auto-generating mathod stub

125 }

126 //Test the viewMobilesBrandWise method for empty list

127 @Test(expected=InvalidMobileException.*class*)

128 *public* *void* test19ViewMobilesBrandWiseForEmptyList() throws InvalidMobileException{

129

130 //fill code

131 ArrayList<Mobile> m=*new* ArrayList<>();

132 mobileStoreObj.viewMobilesBrandWise(m);

133 }

134

135 //Test the countTotalModelsForEachBrand method

136 @Test

137 *public* *void* test20CountTotalModelsForEachBrand() throws InvalidMobileException{

138

139 //fill code

140 Map<String,Integer>m3=mobileStoreObj.countTotalModelsForEachBrand(mobileList,100000.0);

141 //Set<String>key=m3.keySet();

142 assertFalse(mobileStoreObj.countTotalModelsForEachBrand(mobileList,100000.0).isEmpty());

143

144

145 }

146

147 //Test the countTotalModelsForEachBrand method for empty list

148 @Test(expected=InvalidMobileException.*class*)

149 *public* *void* test21CountTotalModelsForEachBrandForEmptyList() throws InvalidMobileException {

150

151 //fill code

152 ArrayList<Mobile>m=*new* ArrayList();

153 mobileStoreObj.countTotalModelsForEachBrand(m,50.0);

154

155

156

157 }

158 }

159

#### **MobileManagement/src/com/mobile/util/MobileStore.java**

1 *package* com.mobile.util;

2

3 *import* java.util.ArrayList;

4 *import* java.util.LinkedHashMap;

5 *import* java.util.List;

6 *import* java.util.Map;

7

8 *import* com.mobile.exception.InvalidMobileException;

9 *import* com.mobile.model.Mobile;

10

11 *public* *class* MobileStore {

12

13 *public* *boolean* validateMobileType (String mobileType) throws InvalidMobileException {

14 *if*(mobileType.equalsIgnoreCase("smartphone") || mobileType.equalsIgnoreCase("basic"))

15 *return* *true*;

16 *else*

17 *throw* *new* InvalidMobileException("Mobile type invalid");

18 }

19

20 *public* Mobile viewMobile(List<Mobile> mobileList,String IMEINumber) throws InvalidMobileException {

21 *if*(mobileList.size()==0){

22 *throw* *new* InvalidMobileException("List is empty");

23 }

24 *else* {

25 *for*(Mobile m : mobileList){

26 *if*(m.getIMEINumber().equals(IMEINumber))

27 *return* m;

28 }

29 *throw* *new* InvalidMobileException("IMEI Number is invalid");

30 }

31 }

32

33 *public* List<Mobile> viewMobilesByBrand (List<Mobile> mobileList ,String brand) throws InvalidMobileException {

34 *if*(mobileList.size()==0){

35 *throw* *new* InvalidMobileException("List is empty");

36 }

37 *else* {

38 List<Mobile> result = *new* ArrayList<>();

39 *for*(Mobile m : mobileList){

40 *if*(m.getBrand().equals(brand))

41 result.add(m);

42 }

43 *return* result;

44 }

45 }

46

47 *public* Map<String,List<Mobile>> viewMobilesBrandWise(List<Mobile> mobileList) throws InvalidMobileException {

48 *if*(mobileList.size()==0){

49 *throw* *new* InvalidMobileException("List is empty");

50 }

51 *else* {

52 Map<String,List<Mobile>> result = *new* LinkedHashMap<>();

53

54 *for*(Mobile m : mobileList){

55 *if*(!result.containsKey(m.getBrand())){

56 result.put(m.getBrand(),*new* ArrayList<Mobile>());

57 }

58 List<Mobile> temp=result.get(m.getBrand());

59 temp.add(m);

60 result.put(m.getBrand(), temp);

61 }

62 *return* result;

63 }

64 }

65

66 *public* Map<String,Integer> countTotalModelsForEachBrand(List<Mobile> mobileList, *double* priceRange) throws InvalidMobileException {

67 *if*(mobileList.size()==0){

68 *throw* *new* InvalidMobileException("List is empty");

69 }

70 *else* {

71 Map<String,Integer> result = *new* LinkedHashMap<>();

72

73 *for*(Mobile m : mobileList){

74 *if*(m.getMobilePrice()<=priceRange){

75 *if*(!result.containsKey(m.getBrand())){

76 result.put(m.getBrand(),1);

77 }

78 *else*

79 {

80 *int* temp=result.get(m.getBrand());

81 result.put(m.getBrand(), temp+1);

82 }

83 }

84 }

85 *return* result;

86 }

87 }

88

89 }

## Grade

Reviewed on Wednesday, 2 June 2021, 8:24 PM by Automatic grade  
**Grade** 100 / 100  
**Assessment report**  
[[-]](javascript:void(0);)**Grading and Feedback**

*Writing JUnit for a Utility class - 60.0 / 60.0(Success)*

*Test Coverage - 40 / 40.0(Success)*

Test Case Passed