Assignment 1 consists of a project called “Assignment1” that has two classes. We will build on this in future assignments.

**Key Rules:**

* The project must compile and run to be marked otherwise it will receive a grade of 0
* The Assignment is **due 11:59 PM the night before session #6**. Late submissions are not permitted
* Class ParkedCarTest.java and parkingMeter.java will be used to mark the majority of your assignment
* This assignment is meant to be done individually, group work will be considered plagiarism.
* All classes must have full Java documentation comments for all public elements( public methods and constructors) using @author , @version, @param, @return tags when applicable and a brief description of the class**. 1 mark will be deducted for each incomplete of missing Java** **documentation tag**
* All classes **must not have magic numbers**, **1 mark will be deducted for each use of a magic number.** Any number other than 0 is considered a magic number

**Requirements:**

Create a class called **ParkedCar**. The class has the following:

|  |
| --- |
| Attributes:  owner name e.g. “Bob Smith”  car make e.g. “Porsche cayenne”  model year e.g. 2015  car license plate number e.g. “1A2B3C”  number of minutes parked e.g. 123 |
| Provide a constructor, the constructor will take parameters in the following order (owner name ,make, model year, license plate number, minutes parked)  The following restrictions should be implemented in the constructor:   * Owner name, car make, and license plate number should not be null, if the provided value was null the value will be rejected and an error message such as “car make cannot be null” or “car license plate number cannot be null” will be produced. * Model year cannot be earlier than 1900 or later than 2019 inclusive otherwise the value will be rejected and an error message “year model cannot be earlier than 1900” or “year model cannot be later than 2019” will be produced.   **HINT**: the current year can be obtained from the system by implementing the following :   * + add the statement **import** java.util.Calendar; before the class header   + assign the following statement to your current year variable   ***CURRENT\_YEAR*** = Calendar.*getInstance*().get(Calendar.***YEAR***);  Alternatively you can assign a number such as 2019 to your current year variable   * number of minutes parked will not be negative value or 0 otherwise the value will be rejected and an error message “minutes parked should be greater than 0” will be produced |
| Provide appropriately named accessor for the fields:  Owner name  Car make  Model year  Car license plate number  Number of minutes parked |
| Provide appropriately named mutators for the fields:  Owner name  Car make  Model year  Car license plate number  Number of minutes parked  The mutators will apply the same validation mentioned in the constructor section above |
| Provide a method displayDetails(), the method will display the details of the ParkedCar object in the following format:  Owner name: Adam White  Car make: Mazda  Car model year: 2010  Car license Plate number: 1A2B3C  Number of minutes parked: 95 |

Create a class called **ParkingMeter**. The class has the following:

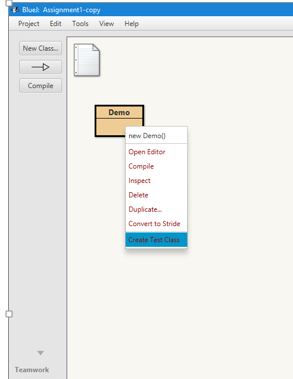
|  |
| --- |
| Attributes:  Meter location e.g. “Burnaby”  has a camera e.g. true  price of one minute in CAD e.g. 10.0  number of purchased minutes e.g. 20 |
| Provide a constructor, the constructor will take parameters in the following order ( meter location, has a camera, price of one minute, number of purchased minutes)  The following restrictions should be implemented in the constructor:   * Meter location should not be null, if the provided value was null the value will be rejected and an error message such as “meter location cannot be null” will be produced * Price of one minute should be greater than 0, if the provided value was 0 or less the value will be rejected and an error message “price of one minute should be greater than 0” will be produced * Number of minutes purchased should be greater than 0 and less or equal to 180. If the provided value was greater than 180 the message “number of purchased minutes cannot be greater than 180, the value will set to 180” and the instance variable will be set to 180. If the provided parameter was less or equal to 180 the value will be rejected and the message “ number of purchased minutes cannot be 0 or negative” will be produced |
| Provide appropriately named accessor for the fields  Meter location  Has a camera  Price of one minute in CAD  Number of purchased minutes |
| Provide appropriately named mutators for the fields  Meter location  Has a camera  Price of one minute in CAD  Number of purchased minutes  The mutators will apply the same validation mentioned in the constructor section above |
| Provide a method displayDetails(), the method will display the details of the ParkingMeter object in the following format:  metre location: Burnaby  meter have a camera: true  price of one minute in CAD: 10.5  number of purchased minutes: 120 |

**Instance variable, constructor and method names must match the following names for the test classes to work properly. Here is a list of the used names**

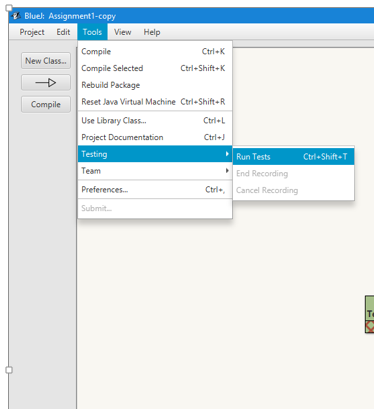
|  |  |
| --- | --- |
| Class ParkedCar | |
| Fields | ownerName  carMake  licensePlateNumber  modelYear  numberOfMinutesParked |
| Constructor | ParkedCar(owner name ,make, model year, license plate number, minutes parked) |
| Methods | getOwnerName  setOwnerName  getCarMake  setCarMake  getLicensePlateNumber  setLicensePlateNumber  getModelYear  setModelYear  getNumberOfMinutesParked  setNumberOfMinutesParked  displayDetails |
| Class PakinMeter | |
| Fields | meterLocation  hasACamera  priceOfOneMinuteInCAD  numberOfPurchasedMinutes |
| Constructor | ParkingMeter(location, camera, priceOfMinute, purchasedMinutes) |
| Methods | getMeterLocation  setMeterLocation  getHasACamera  setHasACamera  getPriceOfOneMinuteInCAD  setPriceOfOneMinuteInCAD  getNumberOfPurchasedMinutes  setNumberOfPurchasedMinutes  displayDetails |

**To use the test classes follow the steps below:**

1. Create a project with the name Assignment#1
2. Create class ParkedCar, implement all the specified requirements
3. Once the parkedCar class compiles create a Test class with the name TestParkedCar. This can be done by right clicking on class ParkedCar and choose create test class



1. Replace the content of class TestParkecCar with the following
2. To run the test go to tools and tools => testing => run tests



1. Repeat the steps above to use class TestParkingMeter. The class is provided below.
2. Both Test classes are provided in the Assignment zipped folder. You can use notepad++ for windows or Komodo edit for Mac. (Download them for free).

**Class TestParkedCar**

|  |
| --- |
| import java.lang.reflect.Method;  import java.io.ByteArrayOutputStream;  import java.io.PrintStream;  import java.lang.reflect.Field;  import java.lang.reflect.Modifier;  import org.junit.AfterClass;  import org.junit.BeforeClass;  import org.junit.Test;  import static org.junit.Assert.\*;  import org.junit.After;  import org.junit.Before;  import org.junit.Test;  public class TestParkedCar {  private static Grader grader = new Grader(27);  private ParkedCar oneCar;  @AfterClass  public static void tearDownAfterClass() throws Exception {  System.out.println(" ParkedCar Test class Score: " + grader.getMarks() + " / " + grader.getMax());  System.out.println("ParkedCar Test Class ends");  grader = new Grader(27);  }  @BeforeClass  public static void setUpBeforeClass()throws Exception {  System.out.println("ParkedCar Test Class Start");  }  // Instance variables getters and setters tests  @Test  public void testOwnerName() {  testField(ParkedCar.class, "ownerName", String.class, new String[] {"private",},  new String[]{"static","final",});  testMethod(ParkedCar.class,"getOwnerName",String.class,new String[] {"public"}, new String[] {"static", "final"});  testMethod(ParkedCar.class,"setOwnerName",void.class, new String[] {"public"},new String[] {"static","final"},String.class);  grader.addMark(1);  }  @Test  public void testcarMake() {  testField(ParkedCar.class, "carMake", String.class, new String[] {"private",},  new String[]{"static","final",});  testMethod(ParkedCar.class,"getCarMake",String.class,new String[] {"public"}, new String[] {"static", "final"});  testMethod(ParkedCar.class,"setCarMake",void.class, new String[] {"public"},new String[] {"static","final"},String.class);  grader.addMark(1);  }  @Test  public void testLicensePlateNumber() {  testField(ParkedCar.class, "licensePlateNumber", String.class, new String[] {"private",},  new String[]{"static","final",});  testMethod(ParkedCar.class,"getLicensePlateNumber",String.class,new String[] {"public"}, new String[] {"static", "final"});  testMethod(ParkedCar.class,"setLicensePlateNumber",void.class, new String[] {"public"},new String[] {"static","final"},String.class);  grader.addMark(1);  }  @Test  public void testModelYear() {  testField(ParkedCar.class, "modelYear", int.class, new String[] {"private",},  new String[]{"static","final",});  testMethod(ParkedCar.class,"getModelYear",int.class,new String[] {"public"}, new String[] {"static", "final"});  testMethod(ParkedCar.class,"setModelYear",void.class, new String[] {"public"},new String[] {"static","final"},int.class);  grader.addMark(1);  }  @Test  public void testNumberOfMinutesParked() {  testField(ParkedCar.class, "numberOfMinutesParked", int.class, new String[] {"private",},  new String[]{"static","final",});  testMethod(ParkedCar.class,"getNumberOfMinutesParked",int.class,new String[] {"public"}, new String[] {"static", "final"});  testMethod(ParkedCar.class,"setNumberOfMinutesParked",void.class, new String[] {"public"},new String[] {"static","final"},int.class);  grader.addMark(1);  }  // constructor validation tests  @Test  public void constuctorOwnerNameValidationTestNegative() {  try {  ParkedCar car = new ParkedCar(null,"Porsche",2015,"1A2B3C",60);  fail("null owner name must throw an IllegalArgumentException");  }  catch(IllegalArgumentException ex) {  assertEquals("owner name cannot be null", ex.getMessage());  grader.addMark(1);  }  }  @Test  public void constructorOwnerNameValidationTestPositive() {  ParkedCar car = new ParkedCar("Bob Smith", "Porsche", 2015, "1A2B3C", 15);  assertEquals("Bob Smith",car.getOwnerName());  grader.addMark(1);  }  @Test  public void construdtorCarMakeValdationTestNegative() {  try {  ParkedCar car = new ParkedCar("Bob Smith", null, 2015,"1A2B3C",60);  fail(" null car make must throw an IllegalArgumentException");  }  catch(IllegalArgumentException ex) {  assertEquals("car make cannot be null", ex.getMessage());  grader.addMark(1);  }  }  @Test  public void constructorCarMakeValidationTestPositive() {  ParkedCar car = new ParkedCar("Bob Smith", "Porsche", 2015, "1A2B3C", 15);  assertEquals("Porsche",car.getCarMake());  grader.addMark(1);  }  @Test  public void constructorModelYearValidationTestNegative() {  try {  ParkedCar car = new ParkedCar("Bob Smith", "Porsche", (ParkedCar.CAR\_MODEL\_YEAR\_LOWER\_LIMIT-1),"1A2B3C",60);  fail("model year cannot be earlier than " + ParkedCar.CAR\_MODEL\_YEAR\_LOWER\_LIMIT);  }  catch(IllegalArgumentException ex) {  assertEquals("model year cannot be earlier than " + ParkedCar.CAR\_MODEL\_YEAR\_LOWER\_LIMIT, ex.getMessage());  grader.addMark(1);  }  try {  ParkedCar car = new ParkedCar("Bob Smith", "Porsche", (ParkedCar.CURRENT\_YEAR + 1),"1A2B3C",60);  fail("model year must not be later than "+ ParkedCar.CURRENT\_YEAR);  }  catch(IllegalArgumentException ex) {  assertEquals("model year cannot be later than " + ParkedCar.CURRENT\_YEAR, ex.getMessage());  }  }  @Test  public void constructorModelYearValidtionTestPositive() {  ParkedCar car = new ParkedCar("Bob Smith", "Porsche", 2015, "1A2B3C", 15);  assertEquals(2015, car.getModelYear());  grader.addMark(1);  }  @Test  public void constructorLicensePlateNumberValidationNegative() {  try {  ParkedCar car = new ParkedCar("Bob Smith", "Porsche", (ParkedCar.CURRENT\_YEAR + 1),null,60);  fail("license plate number must not be null");  }  catch(IllegalArgumentException ex) {  assertEquals("car license number connot be null", ex.getMessage());  grader.addMark(1);  }  }  @Test  public void constructorLicenseNumberValidationPositive() {  ParkedCar car = new ParkedCar("Bob Smith", "Porsche", 2015, "1A2B3C", 15);  assertEquals("1A2B3C", car.getLicensePlateNumber());  grader.addMark(1);  }  @Test  public void constructorMinutesParkedValidationNegative() {  try {  ParkedCar car = new ParkedCar("Bob Smith", "Porsche", 2015, "1A2B3C", -1);  fail("minutes parked must not be negative or 0");  ParkedCar car1 = new ParkedCar("Bob Smith", "Porsche", 2015, "1A2B3C", 0);  fail("minutes parked must not be negative or 0");  }  catch(IllegalArgumentException ex){  assertEquals("minutes parked should be greater than 0", ex.getMessage());  grader.addMark(1);  }  }  // mutators validation  @Test  public void setOwnerNameTestPositive() {  oneCar = new ParkedCar("adam white","Mazda",2010,"1A2B3C",95);  oneCar.setOwnerName("Adam Smith");  assertEquals("Adam Smith", oneCar.getOwnerName());  grader.addMark(1);  }  @Test  public void setOwnerNameTestPositiveEmptyString() {  oneCar = new ParkedCar("adam white","Mazda",2010,"1A2B3C",95);  oneCar.setOwnerName("");  assertEquals("", oneCar.getOwnerName());  grader.addMark(1);  }  @Test  public void setOwnerNameTestPositiveOneCharName() {  oneCar = new ParkedCar("adam white","Mazda",2010,"1A2B3C",95);  oneCar.setOwnerName("A");  assertEquals("A", oneCar.getOwnerName());  grader.addMark(1);  }  @Test  public void setOwnerNameTestNegative() {  try {  oneCar = new ParkedCar("adam white","Mazda",2010,"1A2B3C",95);  oneCar.setOwnerName(null);  fail("owner name must not be null");  }  catch(IllegalArgumentException ex) {  assertEquals("owner name cannot be null", ex.getMessage());  grader.addMark(1);  }  }  @Test  public void setLicensePlateNumberTestPositive() {  oneCar = new ParkedCar("adam white","Mazda",2010,"1A2B3C",95);  oneCar.setLicensePlateNumber("C4D5E6");  assertEquals("C4D5E6", oneCar.getLicensePlateNumber());  grader.addMark(1);  }  @Test  public void setLicensePlateNumberTestNegative() {  try {  oneCar = new ParkedCar("adam white","Mazda",2010,"1A2B3C",95);  oneCar.setLicensePlateNumber(null);  fail("license plate number must not be null");  }  catch(IllegalArgumentException ex) {  assertEquals("car license number connot be null", ex.getMessage());  grader.addMark(1);  }  }  @Test  public void setModelYearTestNegative() {  try {  oneCar = new ParkedCar("adam white","Mazda",2010,"1A2B3C",95);  oneCar.setModelYear(ParkedCar.CAR\_MODEL\_YEAR\_LOWER\_LIMIT - 1);  fail(" mode year must not be earlier than " +ParkedCar.CAR\_MODEL\_YEAR\_LOWER\_LIMIT);  }  catch (IllegalArgumentException ex) {  assertEquals("model year cannot be earlier than " + ParkedCar.CAR\_MODEL\_YEAR\_LOWER\_LIMIT, ex.getMessage());  grader.addMark(1);  }  try {  oneCar.setModelYear(ParkedCar.CURRENT\_YEAR + 1);  fail(" mode year must not be later than " +ParkedCar.CAR\_MODEL\_YEAR\_LOWER\_LIMIT);  }  catch (IllegalArgumentException ex) {  assertEquals("model year cannot be later than " + ParkedCar.CURRENT\_YEAR, ex.getMessage());  grader.addMark(1);  }  }  @Test  public void setModelYearTestPositive() {  try {  oneCar = new ParkedCar("adam white","Mazda",2010,"1A2B3C",95);  oneCar.setModelYear(ParkedCar.CAR\_MODEL\_YEAR\_LOWER\_LIMIT);  assertEquals(ParkedCar.CAR\_MODEL\_YEAR\_LOWER\_LIMIT, oneCar.getModelYear());  }  catch(IllegalArgumentException ex) {  fail("model year can be " +ParkedCar.CAR\_MODEL\_YEAR\_LOWER\_LIMIT);  }  try {  oneCar.setModelYear(ParkedCar.CURRENT\_YEAR);  assertEquals(ParkedCar.CURRENT\_YEAR, oneCar.getModelYear());  }  catch(IllegalArgumentException ex) {  fail("model year can be " + ParkedCar.CURRENT\_YEAR);  }  oneCar.setModelYear(1901);  assertEquals(1901,oneCar.getModelYear());  grader.addMark(1);  }  @Test  public void setNumberOfMinutesParkedTestNegativeZero() {  try {  oneCar = new ParkedCar("adam white","Mazda",2010,"1A2B3C",95);  oneCar.setNumberOfMinutesParked(0);  fail("number of minutes parked cannot be 0");  }  catch(IllegalArgumentException ex) {  assertEquals("minutes parked should be greater than 0", ex.getMessage());  grader.addMark(1);  }  }  @Test  public void setNumberOfMinutesParkedTestNegative() {  try {  oneCar = new ParkedCar("adam white","Mazda",2010,"1A2B3C",95);  oneCar.setNumberOfMinutesParked(-1);  fail("number of minutes parked cannot be negative");  }  catch(IllegalArgumentException ex) {  assertEquals("minutes parked should be greater than 0", ex.getMessage());  grader.addMark(1);  }  }  @Test  public void setNumberOfMinutesParkedTestPositive() {  oneCar = new ParkedCar("adam white","Mazda",2010,"1A2B3C",95);  oneCar.setNumberOfMinutesParked(1);  assertEquals(1, oneCar.getNumberOfMinutesParked());  grader.addMark(1);  }  @Test  public void testdisplayDetails() {  final ByteArrayOutputStream outContent = new ByteArrayOutputStream();  final PrintStream originalOut = System.out;  System.setOut(new PrintStream(outContent));  oneCar = new ParkedCar("Adam White","Mazda",2010,"1A2B3C",95);  oneCar.displayDetails();    System.setOut(originalOut);  assertEquals("Owner name: Adam White"+ System.getProperty("line.separator")  + "Car make: Mazda"+System.getProperty("line.separator")  + "Car model year: 2010" + System.getProperty("line.separator")  + "Car license Plate number: 1A2B3C" + System.getProperty("line.separator")  + "Number of minutes parked: 95" + System.getProperty("line.separator"), outContent.toString());  grader.addMark(1);  }  public void testField(final Class<?> clazz,  final String fieldName,  final Class expectedType,  final String[] expectedModifiers,  final String[] forbiddenModifiers)  {  if(clazz == null){  throw new IllegalArgumentException("clazz cannot be null");  }  if(fieldName == null){  throw new IllegalArgumentException("fieldName cannot be null");  }  try {  final Field field;  final int modifiers;  final Class<?> type;  field = clazz.getDeclaredField(fieldName);  type = field.getType();  if(!(type.equals(expectedType))) {  fail(clazz.getName() + "." + fieldName + " must be declared as \"" + expectedType.getName() + "\"");  }  modifiers = field.getModifiers();  checkRequiredModifiers(clazz, fieldName, expectedModifiers, modifiers);  checkForbiddenModifiers(clazz, fieldName, forbiddenModifiers, modifiers);  }  catch(final NoSuchFieldException ex) {  fail(clazz.getName() + " must have a field named: \"" + fieldName + "\"");  }  }  public void testMethod(final Class<?> clazz,  final String methodName,  final Class expectedReturnType,  final String[] expectedModifiers,  final String[] forbiddenModifiers,  final Class<?>... expectedParameters)  {  if(clazz == null)  {  throw new IllegalArgumentException("clazz cannot be null");  }  if(methodName == null)  {  throw new IllegalArgumentException("methodName cannot be null");  }  try  {  final Method method;  final int modifiers;  final Class<?> returnType;  method = clazz.getDeclaredMethod(methodName, expectedParameters);  returnType = method.getReturnType();  if(!(returnType.equals(expectedReturnType)))  {  fail(clazz.getName() + "." + methodName + " must be return \"" + expectedReturnType.getName() + "\"");  }  modifiers = method.getModifiers();  checkRequiredModifiers(clazz, methodName, expectedModifiers, modifiers);  checkForbiddenModifiers(clazz, methodName, forbiddenModifiers, modifiers);  }  catch(final NoSuchMethodException ex)  {  fail(clazz.getName() + " must have a field named: \"" + methodName + "\"");  }  }  public void checkRequiredModifiers(final Class<?> clazz,  final String name,  final String[] expectedModifiers,  final int actualModifiers)  {  for(final String expected : expectedModifiers)  {  switch(expected)  {  case "public":  {  if(!(Modifier.isPublic(actualModifiers)))  {  fail(clazz.getName() + "." + name + " must be declared \"public\"");  }  break;  }  case "private":  {  if(!(Modifier.isPrivate(actualModifiers)))  {  fail(clazz.getName() + "." + name + " must be declared \"private\"");  }  break;  }  case "protected":  {  if(!(Modifier.isProtected(actualModifiers)))  {  fail(clazz.getName() + "." + name + " must be declared \"protected\"");  }  break;  }  case "final":  {  if(!(Modifier.isFinal(actualModifiers)))  {  fail(clazz.getName() + "." + name + " must be declared \"final\"");  }  break;  }  case "static":  {  if(!(Modifier.isStatic(actualModifiers)))  {  fail(clazz.getName() + "." + name + " must be declared \"static\"");  }  break;  }  }  }  }  public void checkForbiddenModifiers(final Class<?> clazz,  final String name,  final String[] unexpectedModifiers,  final int actualModifiers)  {  for(final String unexpected : unexpectedModifiers)  {  switch(unexpected)  {  case "public":  {  if(Modifier.isPublic(actualModifiers))  {  fail(clazz.getName() + "." + name + " must not be declared \"public\"");  }  break;  }  case "private":  {  if(Modifier.isPrivate(actualModifiers))  {  fail(clazz.getName() + "." + name + " must not be declared \"private\"");  }  break;  }  case "protected":  {  if(Modifier.isProtected(actualModifiers))  {  fail(clazz.getName() + "." + name + " must not be declared \"protected\"");  }  break;  }  case "final":  {  if(Modifier.isFinal(actualModifiers))  {  fail(clazz.getName() + "." + name + " must not be declared \"final\"");  }  break;  }  case "static":  {  if(Modifier.isStatic(actualModifiers))  {  fail(clazz.getName() + "." + name + " must not be declared \"static\"");  }  break;  }  }  }  }  }  class Grader {  private final int max;  private int marks;  public Grader(final int m) {  max = m;  marks = 0;  }  public void addMark(final int mark ) {  marks += mark;  }  public int getMarks() {  return marks;  }  public int getMax() {  return max;  }  } |

**Class TestParkingMeter**

|  |
| --- |
| **import static org.junit.Assert.\*;**  **import org.junit.After;**  **import org.junit.Before;**  **import org.junit.Test;**  **import java.io.ByteArrayOutputStream;**  **import java.io.PrintStream;**  **import java.lang.reflect.Field;**  **import java.lang.reflect.Method;**  **import java.lang.reflect.Modifier;**  **import org.junit.AfterClass;**  **import org.junit.BeforeClass;**  **import org.junit.Ignore;**  **import org.junit.Test;**  **public class TestParkingMeter{ //**  **private static Grader grader = new Grader(21);**  **private ParkingMeter meter;**  **@BeforeClass**  **public static void setUpBeforeClass() {**  **System.out.println("ParkingMeter test class strat");**  **}**  **@AfterClass**  **public static void tearDownAfterClass() throws Exception {**  **System.out.println("ParkingMeter test class Score: " + grader.getMarks() + " / " + grader.getMax());**  **grader = new Grader(21);**  **}**  **// instance variables getters and setters test**  **@Test**  **public void testMeterLocation() {**  **testField(ParkingMeter.class,"meterLocation",String.class, new String[] {"private", },**  **new String [] {"static","final",});**  **testMethod(ParkingMeter.class,"getMeterLocation",String.class, new String[] {"public"},new String[] {"static", "final"});**  **testMethod(ParkingMeter.class,"setMeterLocation",void.class, new String[] {"public"},new String[] {"static", "final"},String.class);**  **grader.addMark(1);**  **//System.out.println("Score: " + grader.getMarks() + " / " + grader.getMax());**  **}**  **@Test**  **public void testHasCamera() {**  **testField(ParkingMeter.class,"hasACamera",boolean.class, new String[] {"private", },**  **new String [] {"static","final",});**  **testMethod(ParkingMeter.class,"getHasACamera",boolean.class, new String[] {"public"},new String[] {"static", "final"});**  **testMethod(ParkingMeter.class,"setHasACamera",void.class, new String[] {"public"},new String[] {"static", "final"},boolean.class);**  **grader.addMark(1);**  **//System.out.println("Score: " + grader.getMarks() + " / " + grader.getMax());**  **}**  **@Test**  **public void testPriceOfOneMinuteInCAD() {**  **testField(ParkingMeter.class,"priceOfOneMinuteInCAD",double.class, new String[] {"private", },**  **new String [] {"static","final",});**  **testMethod(ParkingMeter.class,"getPriceOfOneMinuteInCAD",double.class, new String[] {"public"},new String[] {"static", "final"});**  **testMethod(ParkingMeter.class,"setPriceOfOneMinuteInCAD",void.class, new String[] {"public"},new String[] {"static", "final"},double.class);**  **grader.addMark(1);**  **}**  **@Test**  **public void testNumberOfPurchasedMinutes() {**  **testField(ParkingMeter.class,"numberOfPurchasedMinutes",int.class, new String[] {"private", },**  **new String [] {"static","final",});**  **testMethod(ParkingMeter.class,"getNumberOfPurchasedMinutes",int.class, new String[] {"public"},new String[] {"static", "final"});**  **testMethod(ParkingMeter.class,"setNumberOfPurchasedMinutes",void.class, new String[] {"public"},new String[] {"static", "final"},int.class);**  **grader.addMark(1);**  **}**  **// constructor test**  **@Test**  **public void testContructorMeterLocationNegative() {**  **try {**  **ParkingMeter oneMeter = new ParkingMeter(null,false,10.5,65);**  **fail("null meter location must throw an Illegal Argument Exception");**  **}**  **catch(IllegalArgumentException ex) {**  **assertEquals("meter location cannot be null", ex.getMessage());**  **}**  **grader.addMark(1);**  **}**  **// constructor test**  **@Test**  **public void testConstructorMeterLocationPositive() {**  **ParkingMeter oneMeter = new ParkingMeter("Burnaby",false,10.5,65);**  **assertEquals("Burnaby", oneMeter.getMeterLocation());**  **grader.addMark(1);**  **}**  **@Test**  **public void testConstructorHasACamera() {**  **ParkingMeter oneMeter = new ParkingMeter("Burnaby", true, 10.5,65);**  **assertEquals(true,oneMeter.getHasACamera());**  **grader.addMark(1);**  **}**  **@Test**  **public void testPriceOfMinuteConstructorNegative() {**  **try {**  **ParkingMeter oneMeter = new ParkingMeter("Burnaby",false,0,65);**  **fail("price of one minutes must not be 0");**  **}**  **catch(IllegalArgumentException ex) {**  **assertEquals("price of one minute cannot be 0 or less", ex.getMessage());**  **}**  **try {**  **ParkingMeter oneMeter = new ParkingMeter("Burnaby",false,-1,65);**  **fail("price of one minutes must not be 0");**  **}**  **catch(IllegalArgumentException ex) {**  **assertEquals("price of one minute cannot be 0 or less", ex.getMessage());**  **}**  **grader.addMark(1);**  **}**  **@Test**  **public void testNumberOfPurchasedMinutesConstructorNegativeZero() {**  **try {**  **ParkingMeter oneMeter = new ParkingMeter("Burnaby",false,10.5,0);**  **fail("price of one minutes must not be 0");**  **}**  **catch(IllegalArgumentException ex) {**  **assertEquals("number of purchased minutes cannot be 0 or negative", ex.getMessage());**  **grader.addMark(1);**  **}**  **}**  **@Test**  **public void testNumberOfPurchasedMinutesConstructorNegative() {**  **try {**  **ParkingMeter oneMeter = new ParkingMeter("Burnaby",false,10.5,-1);**  **fail("price of one minutes must not be 0");**  **}**  **catch(IllegalArgumentException ex) {**  **assertEquals("number of purchased minutes cannot be 0 or negative", ex.getMessage());**  **grader.addMark(1);**  **}**  **}**  **@Test**  **public void testNumberOfpurchsedMinutesNegativeUpperLimit() {**  **ParkingMeter oneMeter = new ParkingMeter("Burnaby",false,10.5,181);**  **assertEquals(180, oneMeter.getNumberOfPurchasedMinutes());**  **grader.addMark(1);**  **}**  **@Test**  **public void testNumberOfpurchsedMinutesPositiveUpperLimit() {**  **ParkingMeter oneMeter = new ParkingMeter("Burnaby",false,10.5,180);**  **assertEquals(180, oneMeter.getNumberOfPurchasedMinutes());**  **grader.addMark(1);**  **}**  **@Test**  **public void testNumberOfPurchasedMinutesConstructorPositive() {**  **ParkingMeter oneMeter = new ParkingMeter("Burnaby",false,10.5,1);**  **assertEquals(1,oneMeter.getNumberOfPurchasedMinutes());**  **grader.addMark(1);**  **}**  **// test mutators validation**  **@Test**  **public void testSetMeterLocationNegative() {**  **try {**  **meter = new ParkingMeter("Burnaby",true,10.5,120);**  **meter.setMeterLocation(null);**  **fail("meter location must not be null");**  **} catch(IllegalArgumentException ex) {**  **assertEquals("meter location cannot be null",ex.getMessage());**  **grader.addMark(1);**  **}**  **}**  **@Test**  **public void testSetMeterLocationPositive() {**  **meter = new ParkingMeter("Burnaby",true,10.5,120);**  **meter.setMeterLocation("BCIT");**  **assertEquals("BCIT",meter.getMeterLocation());**  **grader.addMark(1);**  **}**  **@Test**  **public void testSetHasACamera() {**  **//meter.setHasACamera(false);**  **ParkingMeter oneMeter = new ParkingMeter("Burnaby",false,10.5,180);**  **oneMeter.setHasACamera(true);**  **assertEquals(true, oneMeter.getHasACamera());**  **}**  **@Test**  **public void testsetPriceOfMinuteNegative() {**  **try {**  **ParkingMeter oneMeter = new ParkingMeter("Burnaby",false,10.5,180);**  **oneMeter.setPriceOfOneMinuteInCAD(0);**  **fail("price of one minutes must not be 0");**  **}**  **catch(IllegalArgumentException ex) {**  **assertEquals("price of one minute cannot be 0 or less", ex.getMessage());**  **}**  **try {**  **meter = new ParkingMeter("Burnaby",true,10.5,120);**  **meter.setPriceOfOneMinuteInCAD(-1);**  **fail("price of one minutes must not be 0");**  **}**  **catch(IllegalArgumentException ex) {**  **assertEquals("price of one minute cannot be 0 or less", ex.getMessage());**  **}**  **grader.addMark(1);**  **}**  **@Test**  **public void testsetPriceOfMinutepositive() {**  **meter = new ParkingMeter("Burnaby",true,10.5,120);**  **meter.setPriceOfOneMinuteInCAD(1);**  **assertEquals(1,meter.getPriceOfOneMinuteInCAD(),0.01);**  **grader.addMark(1);**  **}**  **@Test**  **public void testSetNumberOfpurchsedMinutesNegativeUpperLimit() {**  **meter = new ParkingMeter("Burnaby",true,10.5,120);**  **meter.setNumberOfPurchasedMinutes(181);**  **assertEquals(180, meter.getNumberOfPurchasedMinutes());**  **grader.addMark(1);**  **}**  **@Test**  **public void testSetNumberOfpurchsedMinutesPositiveUpperLimit() {**  **meter = new ParkingMeter("Burnaby",true,10.5,120);**  **meter.setNumberOfPurchasedMinutes(180);**  **assertEquals(180, meter.getNumberOfPurchasedMinutes());**  **grader.addMark(1);**  **}**  **@Test**  **public void testSetNumberOfPurchasedMinutesPositive() {**  **meter = new ParkingMeter("Burnaby",true,10.5,120);**  **meter.setNumberOfPurchasedMinutes(1);**  **assertEquals(1,meter.getNumberOfPurchasedMinutes());**  **grader.addMark(1);**  **}**  **@Test**  **public void testdisplayDetails() {**  **final ByteArrayOutputStream outContent = new ByteArrayOutputStream();**  **final PrintStream originalOut = System.out;**  **System.setOut(new PrintStream(outContent));**  **ParkingMeter oneMeter = new ParkingMeter("Burnaby",false,10.5,180);**  **oneMeter.displayDetails();**  **System.setOut(new PrintStream(originalOut));**  **assertEquals("metre location: Burnaby" + System.getProperty("line.separator")**  **+ "meter have a camera: false" + System.getProperty("line.separator")**  **+ "price of one minute in CAD: 10.5" + System.getProperty("line.separator")**  **+ "number of purchased minutes: 180"+ System.getProperty("line.separator")**  **, outContent.toString());**  **grader.addMark(1);**  **}**  **public void testField(final Class<?> clazz,**  **final String fieldName,**  **final Class expectedType,**  **final String[] expectedModifiers,**  **final String[] forbiddenModifiers)**  **{**  **if(clazz == null){**  **throw new IllegalArgumentException("clazz cannot be null");**  **}**  **if(fieldName == null){**  **throw new IllegalArgumentException("fieldName cannot be null");**  **}**  **try {**  **final Field field;**  **final int modifiers;**  **final Class<?> type;**  **field = clazz.getDeclaredField(fieldName);**  **type = field.getType();**  **if(!(type.equals(expectedType))) {**  **fail(clazz.getName() + "." + fieldName + " must be declared as \"" + expectedType.getName() + "\"");**  **}**  **modifiers = field.getModifiers();**  **checkRequiredModifiers(clazz, fieldName, expectedModifiers, modifiers);**  **checkForbiddenModifiers(clazz, fieldName, forbiddenModifiers, modifiers);**  **}**  **catch(final NoSuchFieldException ex) {**  **fail(clazz.getName() + " must have a field named: \"" + fieldName + "\"");**  **}**  **}**  **public void testMethod(final Class<?> clazz,**  **final String methodName,**  **final Class expectedReturnType,**  **final String[] expectedModifiers,**  **final String[] forbiddenModifiers,**  **final Class<?>... expectedParameters)**  **{**  **if(clazz == null)**  **{**  **throw new IllegalArgumentException("clazz cannot be null");**  **}**  **if(methodName == null)**  **{**  **throw new IllegalArgumentException("methodName cannot be null");**  **}**  **try**  **{**  **final Method method;**  **final int modifiers;**  **final Class<?> returnType;**  **method = clazz.getDeclaredMethod(methodName, expectedParameters);**  **returnType = method.getReturnType();**  **if(!(returnType.equals(expectedReturnType)))**  **{**  **fail(clazz.getName() + "." + methodName + " must be return \"" + expectedReturnType.getName() + "\"");**  **}**  **modifiers = method.getModifiers();**  **checkRequiredModifiers(clazz, methodName, expectedModifiers, modifiers);**  **checkForbiddenModifiers(clazz, methodName, forbiddenModifiers, modifiers);**  **}**  **catch(final NoSuchMethodException ex)**  **{**  **fail(clazz.getName() + " must have a method named: \"" + methodName + "\"");**  **}**  **}**  **public void checkRequiredModifiers(final Class<?> clazz,**  **final String name,**  **final String[] expectedModifiers,**  **final int actualModifiers)**  **{**  **for(final String expected : expectedModifiers)**  **{**  **switch(expected)**  **{**  **case "public":**  **{**  **if(!(Modifier.isPublic(actualModifiers)))**  **{**  **fail(clazz.getName() + "." + name + " must be declared \"public\"");**  **}**  **break;**  **}**  **case "private":**  **{**  **if(!(Modifier.isPrivate(actualModifiers)))**  **{**  **fail(clazz.getName() + "." + name + " must be declared \"private\"");**  **}**  **break;**  **}**  **case "protected":**  **{**  **if(!(Modifier.isProtected(actualModifiers)))**  **{**  **fail(clazz.getName() + "." + name + " must be declared \"protected\"");**  **}**  **break;**  **}**  **case "final":**  **{**  **if(!(Modifier.isFinal(actualModifiers)))**  **{**  **fail(clazz.getName() + "." + name + " must be declared \"final\"");**  **}**  **break;**  **}**  **case "static":**  **{**  **if(!(Modifier.isStatic(actualModifiers)))**  **{**  **fail(clazz.getName() + "." + name + " must be declared \"static\"");**  **}**  **break;**  **}**  **}**  **}**  **}**  **public void checkForbiddenModifiers(final Class<?> clazz,**  **final String name,**  **final String[] unexpectedModifiers,**  **final int actualModifiers)**  **{**  **for(final String unexpected : unexpectedModifiers)**  **{**  **switch(unexpected)**  **{**  **case "public":**  **{**  **if(Modifier.isPublic(actualModifiers))**  **{**  **fail(clazz.getName() + "." + name + " must not be declared \"public\"");**  **}**  **break;**  **}**  **case "private":**  **{**  **if(Modifier.isPrivate(actualModifiers))**  **{**  **fail(clazz.getName() + "." + name + " must not be declared \"private\"");**  **}**  **break;**  **}**  **case "protected":**  **{**  **if(Modifier.isProtected(actualModifiers))**  **{**  **fail(clazz.getName() + "." + name + " must not be declared \"protected\"");**  **}**  **break;**  **}**  **case "final":**  **{**  **if(Modifier.isFinal(actualModifiers))**  **{**  **fail(clazz.getName() + "." + name + " must not be declared \"final\"");**  **}**  **break;**  **}**  **case "static":**  **{**  **if(Modifier.isStatic(actualModifiers))**  **{**  **fail(clazz.getName() + "." + name + " must not be declared \"static\"");**  **}**  **break;**  **}**  **}**  **}**  **}**    **}** |

**Grading**

The classes will be marked as follows**:**

|  |  |
| --- | --- |
| Class parkedCar | 27 marks |
| Class ParkingMeter | 21 marks |
| Style | 12 marks |
| Total | 60 marks |

Marks will be given for:

* Comments – appropriate and complete, including Javadoc tags @author, @version, @return and @param.
* Style. – In addition to comments style includes following the Java naming convention of classes, variables and methods. It also includes correct indentation and disuse of magic numbers
* Correctness and completeness – code meets the requirements listed above Classes must run successfully with the provided test classes

Create a .zip file containing your entire BlueJ project (zip the folder, not the individual files. Name your project Assignment#1.Upload the file to D2L before the cut-off time.