1. (5 Points) What are syntax errors, run time errors and logic errors? Explain and demo

|  |
| --- |
| **Syntax Errors**   * The error in the source code of a program is called syntax error. * The syntax in any program should be correct so that we won't have any compilation errors. * Syntax errors are nothing but that grammatical mistakes, may be limited to a single character such as semicolon, bracket etc. * **Example:** From the below example we observe that there are semicolons and brackets missing in the code, this type of error is called syntax error.     **Logic Errors**   * The bug or error in the source code is called logic error. * This may result in the incorrect or unexpected behavior. * It is a type of runtime error which may give a wrong output. * **Example:** Assigning a value to the wrong variable may cause a series of unexpected program errors. Multiplying two numbers instead of adding them together may also produce unwanted results.   Here, in the below demo, we are expecting the result after addition of two numbers but as we have used a wrong operator it shows wrong output.    **Runtime Error**   * The error that occurs while the program is running. * Program crash is one of the most noticeable type in runtime error. * Examples include dividing by zero, referencing missing files, calling invalid functions, or not handling certain input correctly.   Here, we observe a runtime error – we have divided a number with zero. This shows a exception at runtime |

1. (5 Points) Design a class named Fan to represent a fan. The class contains:
2. Three constants named **SLOW, MEDIUM, a**nd **FAST** with the values **1, 2,** and **3** to denote the fan speed.
3. A private **int** data field named **speed** that specifies the speed of the fan
4. A private **boolean** data field named **on** that specifies whether the fan is on
5. A private **double** data field named **radius** that specifies the radius of the fan
6. A **string** data field named **color** that specifies the color of the fan
7. The accessor and mutator methods for all four data fields.
8. A no-arg constructor that creates a default fan.
9. A method named **toString()** that returns a string description for the fan. If the fan is on, the method returns the fan speed, color, and radius in one combined string. If the fan is not on, the method returns the fan color and radius along with the string “fan is off” in one combined string.
10. Write a Driver class to test all the methods of Fan class and provide sample input and output

|  |
| --- |
| **Fan Class**  package fan;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class Fan {  private static final int slow = 1;  private static final int medium = 2;  private static final int fast = 3;  private int speed;  private boolean on;  private double radius;  private String color;  /\*\*  \*  \* @return speed  \*/  public int getSpeed() {  return speed;  }  /\*\*  \*  \* @param speed  \*/  public void setSpeed(int speed) {  this.speed = speed;  }  /\*\*  \*  \* @return  \* @returns a boolean value  \*/  public boolean isOn() {  return on;  }  /\*\*  \*  \* @param on  \*/  public void setOn(boolean on) {  this.on = on;  }  /\*\*  \*  \* @return radius  \*/  public double getRadius() {  return radius;  }  /\*\*  \*  \* @param radius  \*/  public void setRadius(double radius) {  this.radius = radius;  }  /\*\*  \*  \* @return color  \*/  public String getColor() {  return color;  }  /\*\*  \*  \* @param color  \*/  public void setColor(String color) {  this.color = color;  }  /\*\*  \* A constructor with no parameters  \*/  public Fan() {  }  @Override  public String toString() {  if (isOn()) {    return "Fan is On " + "speed " +getSpeed() + ", radius " +getRadius() + ", color " + getColor();  }  return "Fan is Off" + ", radius " +getRadius() + ", color " + getColor();  }  }**Fan Driver Class**  package fan;  import java.util.\*;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class FanDriver {  /\*\*  \* @param args the command line arguments  \*/  public static void main(String[] args) {  // TODO code application logic here  /\*\*  \* Created a object  \*/  Fan fan = new Fan();  /\*\*  \* set values for Speed Radius Color On  \*/  fan.setSpeed(3);  fan.setRadius(10);  fan.setColor("peach");  fan.setOn(true);  /\*\*  \* Here we refer to the toString() from Fan Class  \*/  System.out.println(fan.toString());  /\*\*  \* set values for Speed Radius Color  \*/  fan.setSpeed(2);  fan.setRadius(10);  fan.setColor("lavender");  fan.setOn(false);  /\*\*  \* Here we refer to the toString() from Fan Class On  \*/  System.out.println(fan.toString());  }  }**Output** |

1. (5 Points) Write the output for below code and explain the concept of equality comparison? Use figures to explain the references.

|  |
| --- |
| Rectangle box1 = new Rectangle(10, 20, 10, 10);  Rectangle box2 = new Rectangle(10, 20, 10, 10);  Rectangle box3 = box1;  Rectangle box4;  System.out.println(box1 == box2);  System.out.println(box1.equals(box2));  System.out.println(box2 == box3);  System.out.println(box2.equals(box3));  System.out.println(box1 == box3);  box3 = new Rectangle(10, 20, 10, 10);  System.out.println(box1 == box3);  System.out.println(box1.equals(box3));  String s1 = new String("Java");  String s2 = s1;  String s3 = new String(s2);  String s4 = "Java";  System.out.println(s1 == s2);  System.out.println(s1.equals(s2));  System.out.println(s1 == s3);  System.out.println(s1.equals(s3));  System.out.println(s1 == s4);  System.out.println(s1.equals(s4));  System.out.println(s2 == s3);  System.out.println(s2.equals(s3));  System.out.println(s2 == s4);  System.out.println(s2.equals(s4));  System.out.println(s3 == s4);  System.out.println(s3.equals(s4));  Answers :  box2  box1  **False**  **True**    box1    box2  **False**    Box1  box2  **Box3**  **True**    box2    **Box3**  Box1    **True**    **Box3**      box1  **False**    box3  box1  **True**      box1      box3  **True**  S2  S1  **True**    S1      **S2**  **False**  S3  S1  **True**    S1      S3  S2  **False**    S1  S4 = Java    **True**    S1    String S4 = Java  **False**  S3  S2  S1  **True**    S1  **S2**    S3      S1  **S2**  **False**    **S2**  S1  S4 = Java  **True**  **S2**  S1      String S4 = Java  **False**    S3  S4 = Java  **True**  S3      S1  **S2**    String S4 = Java |

1. (5 Points) Write a program that prompts the user to enter a Social Security number in the format DDD-DD-DDDD, where D is a digit. Your program should check whether the input is valid. Provide sample input and output.

|  |
| --- |
| **Class Ssn.java**  package ssn;  import static java.lang.Character.isDigit;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class Ssn {  /\*\*  \*  \*/  public String ssn;  /\*\*  \*  \* @return  \*/  public String getSsn() {  return ssn;  }  /\*\*  \*  \* @param ssn  \*/  public void setSsn(String ssn) {  this.ssn = ssn;  }  /\*\*  \*  \*/  public Ssn() {  }  /\*\*  \*  \* @return  \*/  /\*\*  \* trim the extra spaces enters the loop only if it satisfies the conditions  \* such as length of string and splitting the characters with ‘-‘ Divides  \* the given string after third digits and also after fifth digit.  \* Checks the remaining characters are digits or no  \*/  public String getSsnNumber() {  for (int i = 0; i < ssn.length(); i++) {  if (ssn.trim().length() != 11 || ssn.charAt(3) != ‘-‘ || ssn.charAt(6) != ‘-‘) {  return ssn + “ an invalid social security number”;  }  if (i != 3 && i != 6) {  if (!isDigit(ssn.charAt(i))) {  return ssn + “ an invalid social security number”;  }  }  }  return ssn  + “ an valid social security number”;  }  @Override  public String toString() {  String name = getSsnNumber();  return name;  }  }  **Driver Class – SsnDriver.java**  package ssn;  import static java.lang.Character.isDigit;  import java.util.Scanner;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class SsnDriver {  /\*\*  \* @param args the command line arguments  \*/  public static void main(String[] args) {  // TODO code application logic here  /\*\*  \* User Input is taken  \*/  System.out.print(“Enter a SSN: “);  Scanner scan = new Scanner(System.in);  String ssn = scan.next();  Ssn num= new Ssn();  num.setSsn(ssn);  System.out.println(num.toString());  }  } |

|  |
| --- |
| **Output :**  Valid Output :    Invalid Output : |

1. (5 Points) Write a program that displays all possible combinations for picking two numbers from integers 1 to 7. Display total number of combinations.

|  |
| --- |
| **Combinations Class**  package combinations;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class Combinations {  /\*\*  \* @param args the command line arguments  \*/  public static void main(String[] args) {  // TODO code application logic here  /\*\*  \* Checks for values and increments and as long as it satisfies the  \* condition prints the values  \*/  int count = 0;  for (int i= 1; i < 8; i++) {  for (int j = i + 1; j < 8; j++) {  System.out.println(I + “ “ + j);  count++;  }  }  System.out.println(“The total number of all combinations is “ + count);  }  } |

|  |
| --- |
| **Output :** |

1. (5 Points) The great circle distance is the distance between two points on the surface of a sphere. Let (x1, y1) and (x2, y2) be the geographical latitude and longitude of two points. The great circle distance between the two points can be computed using the following formula:

Write a program that prompts the user to enter the latitude and longitude of two points on the earth in degrees and displays its great circle distance. The average earth radius is 6,371.01 km. Note that you need to convert the degrees into radians using the **Math.toRadians** method since the Java trigonometric methods use radians. The latitude and longitude degrees in the formula are for north and west. Use negative to indicate south and east degrees. After executing the program, what did you notice with Radians and degrees?

|  |
| --- |
| **Latittude.java**  package ll;  import java.util.\*;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class Latitude {  /\*\*  \* @param args the command line arguments  \*/  public static void main(String[] args) {  // TODO code application logic here  String s1;  String s2;  double x1;  double x2;  double y1;  double y2;  /\*\*  \*  \* Reads input from user  \*/  Scanner scan = new Scanner(System.in);  System.out.print(“Enter point 1 (latitude and longitude) in degrees: “);  s1 = scan.nextLine();  /\*\*  \*  \* Converting String input into double  \*/  x1 = Double.parseDouble(s1.substring(0, s1.indexOf(‘,’)));  y1 = Double.parseDouble(s1.substring(s1.indexOf(‘,’) + 1));  /\*\*  \*  \* Reads input from user  \*/  System.out.print(“Enter point 2 (latitude and longitude) in degrees: “);  s2 = scan.nextLine();  x2 = Double.parseDouble(s2.substring(0, s2.indexOf(‘,’)));  y2 = Double.parseDouble(s2.substring(s2.indexOf(‘,’) + 1));  double distance;  /\*\*  \*  \* calculates the distance  \*/  distance = 6371.01 \* Math.acos(Math.sin(Math.toRadians(x1))  \* Math.sin(Math.toRadians(x2))  + (Math.cos(Math.toRadians(x1))  \* Math.cos(Math.toRadians(x2)))  \* Math.cos(Math.toRadians(y1) – Math.toRadians(y2)));  /\*\*  \*  \* Prints the calculated distance  \*/  System.out.println(“The distance between the two points is “ + distance + “ km”);  }  } |
| **Output** :    Here, we observe that toRadians helps toconvert an angle measured in degrees to an approximately equivalent trigonometric value measured in radians. |

1. (8 Points) Java API has the **GregorianCalendar** class in the **java.util** package, which you can use to obtain the year, month, and day of a date. The no-arg constructor constructs an instance for the current date, and the methods **get(GregorianCalendar.YEAR)**, get**(GregorianCalendar.MONTH),**and **get(GregorianCalendar.DAY\_OF\_MONTH)** return the year, month, and day.

Write a program to perform two tasks:

* Display the current year, month, and day.
* The **GregorianCalendar** class has the **setTimeInMillis(long),** which can be used to set a specified elapsed time since January 1, 1970. Set the value to **1234567898765L** and display the year, month, and day.
* Write briefly what did you learn and observe from this program.

|  |
| --- |
| Package calendar;  import java.util.GregorianCalendar;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class Calendar {  public static void main(String[] args) {  /\*\*  \*  \* Object Creation  \*/  GregorianCalendar cal = new GregorianCalendar();  System.out.println(“Current year:” + cal.get(GregorianCalendar.YEAR));  System.out.println(“Current month:” + (cal.get(GregorianCalendar.MONTH) + 1));  System.out.println(“Current day:” + cal.get(GregorianCalendar.DAY\_OF\_MONTH));  cal.setTimeInMillis(1234567898765L);  System.out.println(“Updated year:” + cal.get(GregorianCalendar.YEAR));  System.out.println(“Updated month:” + (cal.get(GregorianCalendar.MONTH) + 1));  System.out.println(“Updated day:” + cal.get(GregorianCalendar.DAY\_OF\_MONTH));  }  }  Output : |
| * The first month of the year in the Gregorian calendars is JANUARY which is 0; the last depends on the number of months in a year i.e. DECEMBER display as 11th month of year. * An object named cal of **GregorianCalendar** Class is initialized with the current date and time in the default locale and timezone. |

1. (10 Points) Explain the concept of method overloading method with examples.

|  |
| --- |
| * Method Overloading is basically having more than one method having the same name but different arguments, either the type or the number of arguments. * It is like constructor overloading, where a class can have multiple constructors with different arguments. * The use of method overloading concept reduces complexity. As it is not necessary for us to memorize a greater number of methods. The use method overloading concept increases flexibility. * Java allows method overloading concept i.e. having multiple methods in the same class with the same name but accept different types of parameters. * Method overloading is also called compile time polymorphism. * Example :   **Overload Class**  package overload;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class Overload {  /\*\*  \*  \* Use of same method(load1) with different arguments  \*/  public void load1(int x)  {  System.out.println("int : " + x);  }  public void load1(String s)  {  System.out.println("String : " + s);  }  public void load1(double d, double e)  {  double f = d+e;  System.out.println("double : " + f);  }  }  **Driver Class**  package overload;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class OverloadDriver {  /\*\*  \* @param args the command line arguments  \*/  public static void main(String[] args) {  // TODO code application logic here  //object creation  Overload obj = new Overload();  // Methods are invoked according with different arguments  obj.load1("Hey, This is Nithya Karepe "); //String  obj.load1(1150); // Int  obj.load1(7.5, 9.7); //double  }  }       * Here, we observe that the method load1 is used with different arguments. * We have taken load1 method with arguments type such as string, int, double. * This usage of a method will help reduce complexity of memorizing all methods to be used further, instead we have one method which is used in entire code. |

1. (10 Points) You have just started a sales job in a department store. Your pay consists of a base salary and a commission. The base salary is $5,000. The scheme shown below is used to determine the commission rate.

|  |
| --- |
| **Sales.java**  package sales;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class Sales {  public static void main(String[] args) {  double baseSalary = 5000;  double target = 30000;  double requiredCommission = target - baseSalary;  double commission = 0;  double saleAmount = 0;  /\*\*  \*  \* Calculating commission for minimum sales  \*/  do {  if (saleAmount <= 5000) {  commission = 0.08 \* saleAmount;  } else if ((5000 < saleAmount) && (saleAmount <= 10000)) {  commission = (.08 \* 5000) + (0.1 \* (saleAmount - 5000));  } else if (saleAmount > 10000) {  commission = (.08 \* 5000) + (0.1 \* 5000) + (0.12 \* (saleAmount - 10000));  }  saleAmount = saleAmount + .01;  } while (commission <= requiredCommission);  /\* \*  \*  Prints minimum sales  \*/  System.out.printf("The minimum sales needed to earn 30,000: %.2f", saleAmount);  }  } |

|  |
| --- |
| Output : |

1. (15 Points) Credit card numbers follow certain patterns. A credit card number must have between 13 and 16 digits. It must start with:

* 4 for Visa cards
* 5 for Master cards
* 37 for American Express cards
* 6 for Discover cards

In 1954, Hans Luhn of IBM proposed an algorithm for validating credit card numbers. The algorithm is useful to determine whether a card number is entered correctly or whether a credit card is scanned correctly by a scanner. Credit card numbers are generated following this validity check, commonly known as the Luhn check or the Mod 10 check, which can be described as follows (for illustration, consider the card number 4388576018402626):

1. Double every second digit from right to left. If doubling of a digit results in a two-digit number, add up the two digits to get a single-digit number.

4388576018402626

2 \* 2 = 4

2 \* 2 = 4

4 \* 2 = 8

1 \* 2 = 2

6 \* 2 = 12 (1+2 = 3)

5 \* 2 = 10 (1+0 = 1)

8 \* 2 = 16 (1+6 = 7)

4 \* 2 = 8

1. Now add all single-digit numbers from Step 1.

4 + 4 + 8 + 2 + 3 + 1 + 7 + 8 = 37

1. Add all digits in the odd places from right to left in the card number.

6 + 6 + 0 + 8 + 0 + 7 + 8 + 3 = 38

1. Sum the results from Step 2 and Step 3.
2. + 38 = 75
3. If the result from Step 4 is divisible by 10, the card number is valid; otherwise,

It is invalid. For example, the number 4388576018402626 is invalid, but the

number 4388576018410707 is valid.

Write a program that prompts the user to enter a credit card number as a long integer. Display whether the number is valid or invalid. Design your program to use the following methods:

/\*\* Return true if the card number is valid \*/

**public static boolean** isValid(**long** number)

/\*\* Get the result from Step 2 \*/

**public static int** sumOfDoubleEvenPlace(**long** number)

/\*\* Return this number if it is a single digit, otherwise,

\* return the sum of the two digits \*/

**public static int** getDigit(**int** number)

/\*\* Return sum of odd-place digits in number \*/

**public static int** sumOfOddPlace(**long** number)

/\*\* Return true if the digit d is a prefix for number \*/

**public static boolean** prefixMatched(**long** number, **int** d)

/\*\* Return the number of digits in d \*/

**public static int** getSize(**long** d)

/\*\* Return the first k number of digits from number. If the

\* number of digits in number is less than k, return number. \*/

**public static long** getPrefix(**long** number, **int** k)

Here are sample runs of the program: (You may also implement this program by reading the input as a string and processing the string to validate the credit card.)

|  |
| --- |
| Enter a credit card number as a long integer: 4388576018410707  4388576018410707 is valid |

|  |
| --- |
| Enter a credit card number as a long integer: 4388576018402626  4388576018402626 is invalid |

|  |
| --- |
| CreditCard.java :  package creditCard;  import java.util.Scanner;  /\*\*  \*  \* @author Nithya Karepe  \*/  public class CreditCard {  /\*\*  \* @param args the command line arguments  \*/  public static void main(String[] args) {  // TODO code application logic here  Scanner scan = new Scanner(System.in);  System.out.println("Enter a credit card number as a long integer: ");  long cardNumber = scan.nextLong();  if (isValid(cardNumber)) {  System.out.println(cardNumber + " is valid");  } else {  System.out.println(cardNumber + " is invalid");  }  }  /\*\*  \* Return true if the card number is valid  \*Checks for the length of the card and also checks  \* with what digit the credit card number starts with.  \* @param number  \* @return  \*/  public static boolean isValid(long number) {  return (getSize(number) >= 13 && getSize(number) <= 16)  && (prefixMatched(number, 4) || prefixMatched(number, 5)  || prefixMatched(number, 37) || prefixMatched(number, 6))  && ((sumOfDoubleEvenPlace(number) + sumOfOddPlace(number)) % 10 == 0);  }  /\*\*  \* Get the result from Step 2  \*  \* @param number  \* @return  \*/    public static int sumOfDoubleEvenPlace(long number) {  String no = String.valueOf(number);  int sum = 0;  for (int i = getSize(number) - 1; i >= 0; i--) {  int val = Integer.parseInt(String.valueOf(no.charAt(i)));  if (i % 2 == 0 && getSize(number) % 2 == 0) {  sum += getDigit(val \* 2);  } else if (i % 2 != 0 && getSize(number) % 2 != 0) {  sum += getDigit(val \* 2);  }  }  return sum;  }  /\*\*  \* Return this number if it is a single digit, otherwise, return the sum of  \* the two digits  \*  \* @param number  \* @return  \*/  public static int getDigit(int number) {  if (number > 9) {  number = getDigit(number / 10 + number % 10);  }  return number;  }  /\*\*  \* Return sum of odd-place digits in number  \*  \* @param number  \* @return  \*/  public static int sumOfOddPlace(long number) {  String no = String.valueOf(number);  int sum = 0;  for (int i = getSize(number) - 1; i >= 0; i--) {  int val = Integer.parseInt(String.valueOf(no.charAt(i)));  if (i % 2 != 0 && getSize(number) % 2 == 0) {  sum += getDigit(val);  } else if (i % 2 == 0 && getSize(number) % 2 != 0) {  sum += getDigit(val);  }  }  return sum;  }  /\*\*  \* Return true if the digit d is a prefix for number  \*  \* @param number  \* @param d  \* @return  \*/  public static boolean prefixMatched(long number, int d) {  return String.valueOf(number).startsWith(String.valueOf(d));  }  /\*\*  \* Return the number of digits in d  \*  \* @param d  \* @return  \*/  public static int getSize(long d) {  return String.valueOf(d).length();  }  /\*\*  \* Return the first k number of digits from number.If the number of digits  \* in number is less than k, return number.  \*  \* @param number  \* @param k  \* @return  \*/  public static long getPrefix(long number, int k) {  if (getSize(number) < k) {  return number;  }  return Long.getLong(String.valueOf(number).substring(0, k));  }  } |

|  |
| --- |
| Output :  Valid    Invalid      GitHub : <https://github.com/KarepeN/ResubmitFall2020Exam01/tree/main/Karepe_Exam01> |