Assignment 2
22610018
S2 Batch
1.Explain difference between method overloading and method overriding.
Method overloading and overriding are both fundamental concepts in object-oriented programming (OOP) that deal with methods in classes, but they serve distinct purposes:

Method Overloading

- **Definition:** Involves defining multiple methods within the same class that share the same name but differ in their parameters. This allows you to perform similar actions but with different inputs.
- **Key Points:**
  - Methods must have the same name.
  - Parameters must differ in number, order, or data type.
  - Return type can be the same or different.
  - Occurs within a single class (no inheritance required).
  - Example: `calculateArea(int length)` and `calculateArea(int length, int breadth)` are overloaded methods for calculating area of different shapes (square vs rectangle).
- **Benefits:** Improves code readability and reusability by providing specific methods for different input scenarios.

Method Overriding

- **Definition:** Occurs in inheritance hierarchies, where a subclass redefines a method inherited from its parent class. The subclass provides its own implementation specific to its needs while maintaining the same method name and signature (parameters and return type) as the parent class method.
- **Key Points:**
  - Methods must have the same name and signature (parameters and return type) in both parent and child class.
  - Return type must be the same or a subtype (covariant) in the child class.
  - Requires inheritance between classes.
  - Example: An animal class might have a `makeSound()` method, and subclasses like Dog and Cat can override it with their specific sounds (bark and meow).
- **Benefits:** Enables polymorphism, allowing you to treat objects of different subclasses uniformly through the parent class reference. The specific implementation is determined at runtime based on the actual object type.

In essence, method overloading provides flexibility within a class for handling different function calls, while method overriding allows for specialization in subclasses based on inheritance.

2. Implement all string functions in java.

While implementing all string functionalities directly wouldn't be practical due to the extensive nature of the String class and potential for code duplication, Java provides a rich set of built-in methods that cover various string operations. Here's an overview of some common string methods:

Basic Operations:

- **`length()`:** Returns the number of characters in the string.
- **`charAt(index)`:** Returns the character at the specified index (zero-based).
- **`substring(startIndex, endIndex)`:** Extracts a portion of the string from the `startIndex` (inclusive) to `endIndex` (exclusive).
- **`concat(str)`:** Appends another string (`str`) to the end of the current string.

Case Conversion:

- **`toLowerCase()`:** Converts all characters to lowercase.
- **`toUpperCase()`:** Converts all characters to uppercase.

Searching and Replacing:

- **`indexOf(str)`:** Finds the first occurrence of the specified string (`str`) and returns its index, or -1 if not found.
- **`lastIndexOf(str)`:** Finds the last occurrence of the specified string (`str`) and returns its index, or -1 if not found.
- **`replace(oldChar, newChar)`:** Replaces all occurrences of `oldChar` with `newChar` in the string.

Splitting and Joining:

- **`split(delimiter)`:** Splits the string into an array of substrings based on the specified delimiter.
- **`join(delimiter, strings)`:** Joins an array of strings (`strings`) into a single string with the specified delimiter inserted between each element.

Checking and Trimming:

- **`startsWith(str)`:** Checks if the string starts with the specified string (`str`).
- **`endsWith(str)`:** Checks if the string ends with the specified string (`str`).
- **`trim()`:** Removes leading and trailing whitespace characters.

Additional Methods:

- **`equals(str)`:** Compares the string with another string for equality.
- **`equalsIgnoreCase(str)`:** Compares the string with another string for equality, ignoring case sensitivity.

- **`isEmpty()`:** Checks if the string is empty (zero characters).

3. Implement all stringbuffer functions in java.

Appending and Inserting:

- **`append(String str)`:** Appends the given string (`str`) to the end of the `StringBuffer`.
- **`append(int value)`:** Appends the string representation of the integer (`value`) to the end.
- **`append(char ch)`:** Appends the given character (`ch`) to the end.
- **`insert(int index, String str)`:** Inserts the given string (`str`) at the specified `index`.

Deleting and Replacing:

- **`delete(int startIndex, int endIndex)`:** Removes characters from the `startIndex` (inclusive) to `endIndex` (exclusive).
- **`replace(int startIndex, int endIndex, String str)`:** Replaces characters from the `startIndex` (inclusive) to `endIndex` (exclusive) with the given string (`str`).

Capacity and Length:

- **`capacity()`:** Returns the current capacity of the `StringBuffer` (the allocated space).
- **`length()`:** Returns the length of the character sequence in the `StringBuffer`.

Reversing and Other:

- **`reverse()`:** Reverses the characters in the `StringBuffer`.
- **`toString()`:** Converts the `StringBuffer` object to a String.

It's important to note that `StringBuffer` methods are **mutable**, meaning they modify the original object. While it offers thread-safety compared to the String class, using `StringBuilder` is generally recommended for performance-critical string manipulations as it's not thread-safe but avoids unnecessary object creation.

4. Explain with example declaration of string using string

literal and new keyword.

1. String Literal:

When you declare a string using a string literal, you directly assign the string

value between double quotes. Java automatically creates a String object to

represent the literal.

```
public class StringDeclarationExample {

public static void main(String[] args) {

// Declaration using string literal

String strLiteral = "Hello, using string literal!";

System.out.println("String Literal: " + strLiteral);

}

}
```

In this example, the strLiteral variable is assigned the value "Hello, using string literal!" directly as a string literal.

2. new Keyword:

You can also use the new keyword to create a String object explicitly. This approach is less common for creating simple string literals but is useful when you want to create strings dynamically or when dealing with mutable string classes like StringBuilder or StringBuffer.

```
public class StringDeclarationExample {

public static void main(String[] args) {

// Declaration using new keyword

String strNew = new String("Hello, using new keyword!");

System.out.println("String with new keyword: " + strNew);

}
}
```

String Literal vs. new Keyword:

When you use a string literal, Java may reuse existing string objects from the string pool for efficiency, so literal declarations are more common and preferred in most cases.

The new keyword always creates a new String object, even if an identical string already exists in the pool.

The string pool is a special area in the Java heap memory where literal strings are stored to optimize memory usage and improve performance.’

5
.C reate a class named Shape with a method to print .This is This is Shape” Then create two other classes named Rectangle Circle inheriting the Shape class, both having a method to print.This is rectangular shape" and "This is circular shape" respectively. Create asubclass Square of Rectangle' having a method to print Square is are ctangle Now c

```
public class Shape {
	public void printShape() {
	System.out.println("This is Shape");
	}
}

class Rectangle extends Shape {
	public void printRectangle() {
	System.out.println("This is rectangular shape");
	}
}

class Circle extends Shape {
	public void printCircle() {
	System.out.println("This is circular shape");
	}
}

class Square extends Rectangle {
	public void printSquare() {
	System.out.println("Square is a rectangle");
	}
}

public class Main {
	public static void main(String[] args) {
	Shape shape = new Shape();
	Rectangle rectangle = new Rectangle();
	Circle circle = new Circle();
	Square square = new Square();

	shape.printShape();
	rectangle.printShape(); // Calls inherited method from Shape
	rectangle.printRectangle();
	circle.printShape(); // Calls inherited method from Shape
	circle.printCircle();
	square.printShape(); // Calls inherited method from Shape and Rectangle
	square.printRectangle(); // Calls inherited method from Rectangle
	square.printSquare();
	}
}
```

6.Create game characters using the concept of inheritance.
Suppose, in your game, you want three characters - a maths teacher, a footballer and a businessman. Since, all of the characters are persons, they can walk and talk. However, they also have some special skills. A maths teacher can teach maths, a footballer can play football and a businessman can run abusiness. You can individually create three classes who can walk, talk and perform their special skill as shown in the figure below. In each of the classes, you would be copying the same code for walk and talk for each character. If you want to add a new feature - eat, you need to implement the same code for each character. This can easily become error prone (when copying) and duplicate codes. d be a lot easier if we had a Person class with basic features like talk, walk, eat, sleep, and add special
skills to those features as per our characters. This is done using inheritance Using inheritance, now you don't implement the same code for walk and talk for each class. You just need to inheritthem. So, for Maths teacher(derived class), you inherit all features of a Person(base class) and add anew feature TeachMaths.
Likewise, for a footballer, you inherit all thefeatures of a Person and add a new feature PlayFootball and so on.

```
public class GameCharacter {

  private String name;

  public GameCharacter(String name) {
        this.name = name;
  }

  public void walk() {
        System.out.println(name + " is walking...");
  }

  public void talk() {
        System.out.println(name + " is talking...");
  }

  public void eat() {
        System.out.println(name + " is eating...");
  }

  public void sleep() {
        System.out.println(name + " is sleeping...");
  }
}
```

```
class MathsTeacher extends GameCharacter {

  public MathsTeacher(String name) {
        super(name); // Call parent constructor to set name
  }

  public void teachMaths() {
        System.out.println(getName() + " is teaching Maths..."); // Use getName()
  }
}

class Footballer extends GameCharacter {

  public Footballer(String name) {
        super(name);
  }

  public void playFootball() {
        System.out.println(getName() + " is playing Football...");
  }
}

class Businessman extends GameCharacter {

  public Businessman(String name) {
        super(name);
  }

  public void runBusiness() {
        System.out.println(getName() + " is running a business...");
  }
}



public class Main {

  public static void main(String[] args) {
        MathsTeacher teacher = new MathsTeacher("Mr. Smith");
        Footballer footballer = new Footballer("John Doe");
        Businessman businessman = new Businessman("Jane Doe");

        teacher.walk();
        teacher.talk();
        teacher.eat();
        teacher.sleep();
        teacher.teachMaths();
```

```
        footballer.walk();
        footballer.talk();
        footballer.eat();
        footballer.sleep();
        footballer.playFootball();

        businessman.walk();
        businessman.talk();
        businessman.eat();
        businessman.sleep();
        businessman.runBusiness();
  }
}
```

8. Write a Java Program to demonstrate StringBuilder

```
public class StringBuilderExample {

        public static void main(String[] args) {

        // Create a StringBuilder object
        StringBuilder sb = new StringBuilder();

        // Append strings
        sb.append("Hello");
        sb.append(" World!");

        // Print the current string
        System.out.println(sb); // Output: Hello World!

        // Insert a string at specific index
        sb.insert(5, " Beautiful");

        // Print the string after insertion
        System.out.println(sb); // Output: Hello Beautiful World!

        // Replace a part of the string
        sb.replace(7, 16, "Amazing");

        // Print the string after replacement
        System.out.println(sb); // Output: Hello Amazing World!

        // Delete a part of the string
        sb.delete(7, 12);

        // Print the string after deletion
        System.out.println(sb); // Output: Hello World!
```

```
        // Reverse the string
        sb.reverse();

        // Print the reversed string
        System.out.println(sb); // Output: !dlroW olleH

        // Get the length of the string
        int length = sb.length();

        // Print the length
        System.out.println("Length: " + length); // Output: Length: 12

        // Get a character at specific index
        char character = sb.charAt(3);

        // Print the character
        System.out.println("Character at index 3: " + character); // Output: Character at index
3: l
        }
}
```

9.Write a Java Program to demonstrate Methodoverriding.(create class Result with method result().Override method result() in UGResult and PGResult class).

```
class Result {

  public void result() {
        System.out.println("General Result");
  }
}

class UGResult extends Result {

  @Override
  public void result() {
        System.out.println("Undergraduate Result");
  }
}

class PGResult extends Result {

  @Override
  public void result() {
        System.out.println("Postgraduate Result");
```

```
  }
}

public class Main {

  public static void main(String[] args) {
        Result ug = new UGResult();
        Result pg = new PGResult();

        ug.result(); // Output: Undergraduate Result
        pg.result(); // Output: Postgraduate Result
  }
}
```

10. Write a java program to create a class called STUDENT with data members PRN, Name and age. Using inheritance, create a classes called UGSTUDENT and PGSTUDENT having fields as semester, fees and stipend. Enter the data for at least 5 students.Find the semester wise average age for all UG and PG students separately.

```
import java.util.HashMap;
import java.util.Scanner;

public class Student {
        protected String PRN;
        protected String name;
        protected int age;

        public Student(String PRN, String name, int age) {
        this.PRN = PRN;
        this.name = name;
        this.age = age;
        }

        public void displayDetails() {
        System.out.println("PRN: " + PRN);
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        }
}

class UGStudent extends Student {
        private int semester;
        private double fees;

        public UGStudent(String PRN, String name, int age, int semester, double fees) {
        super(PRN, name, age);
        this.semester = semester;
```

```
        this.fees = fees;
        }

        @Override
        public void displayDetails() {
        super.displayDetails();
        System.out.println("Semester: " + semester);
        System.out.println("Fees: " + fees);
        }
}

class PGStudent extends Student {
        private int semester;
        private double stipend;

        public PGStudent(String PRN, String name, int age, int semester, double stipend) {
        super(PRN, name, age);
        this.semester = semester;
        this.stipend = stipend;
        }

        @Override
        public void displayDetails() {
        super.displayDetails();
        System.out.println("Semester: " + semester);
        System.out.println("Stipend: " + stipend);
        }
}

public class Main {
        public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        HashMap<Integer, Double> ugSemesterWiseAvgAge = new HashMap<>();
        HashMap<Integer, Double> pgSemesterWiseAvgAge = new HashMap<>();

        // Enter data for 5 students
        for (int i = 0; i < 5; i++) {
        System.out.println("Enter student details (PRN, Name, Age, Semester,
Fees/Stipend):");
        String PRN = scanner.nextLine();
        String name = scanner.nextLine();
        int age = scanner.nextInt();
        int semester = scanner.nextInt();
        scanner.nextLine(); // Consume newline character

        if (i % 2 == 0) { // Even index -> UG student
                double fees = scanner.nextDouble();
                UGStudent ugStudent = new UGStudent(PRN, name, age, semester, fees);
```

```
                ugStudent.displayDetails();

                // Update UG semester-wise average age
                ugSemesterWiseAvgAge.putIfAbsent(semester, 0.0);
                ugSemesterWiseAvgAge.put(semester,
ugSemesterWiseAvgAge.get(semester) + age);
        } else { // Odd index -> PG student
                double stipend = scanner.nextDouble();
                PGStudent pgStudent = new PGStudent(PRN, name, age, semester,
stipend);
                pgStudent.displayDetails();

                // Update PG semester-wise average age
                pgSemesterWiseAvgAge.putIfAbsent(semester, 0.0);
                pgSemesterWiseAvgAge.put(semester,
pgSemesterWiseAvgAge.get(semester) + age);
        }
        scanner.nextLine(); // Consume newline character after numerical input
        }

        // Calculate and print UG semester-wise average age
        System.out.println("\nUG Semester Wise Average Age:");
        for (int semester : ugSemesterWiseAvgAge.keySet()) {
        double avgAge = ugSemesterWiseAvgAge.get(semester) / 2; // Assuming 2 UG
students per iteration
        System.out.println("Semester " + semester + ": " + String.format("%.2f", avgAge));
        }

        // Calculate and print PG semester-wise average age
        System.out.println("\nPG Semester Wise Average Age:");
        for (int semester : pgSemesterWiseAvgAge.keySet()) {
        double avgAge = pgSemesterWiseAvgAge.get(semester) / 2; // Assuming 2 PG
students per iteration
        System.out.println("Semester " + semester + ": " + String.format("%.2f", avgAge));
        }
        }
}
```

11.11. Implement hybrid inheritance using all access specifiers (public, private,protected).

```
        class Base {
        private int privateField;
        protected int protectedField;

        public Base() {
```

```
        privateField = 10;
        protectedField = 20;
        }

        public void baseMethod() {
        System.out.println("Base method: accessing privateField (not recommended) = " +
privateField);
        System.out.println("Base method: accessing protectedField = " + protectedField);
        }
}

public class A extends Base {
        public int publicField;

        public A() {
        publicField = 30;
        }

        public void methodA() {
        System.out.println("Method A: accessing publicField (inherited) = " + publicField);

        // Cannot access privateField from Base (not inherited)
        // System.out.println("Method A: accessing privateField (not inherited) = " +
privateField);

        // Can access protectedField from Base (inherited)
        System.out.println("Method A: accessing protectedField (inherited) = " +
protectedField);
        }
}

public class B extends Base {
        public int publicField;

        public B() {
        publicField = 40;
        }

        public void methodB() {
        System.out.println("Method B: accessing publicField (inherited) = " + publicField);

        // Cannot access privateField from Base (not inherited)
        // System.out.println("Method B: accessing privateField (not inherited) = " +
privateField);

        // Can access protectedField from Base (inherited)
        System.out.println("Method B: accessing protectedField (inherited) = " +
protectedField);
```

```
        }
}

public class C extends B {
        public int publicField;

        public C() {
        publicField = 50;
        }

        public void methodC() {
        System.out.println("Method C: accessing publicField (inherited) = " + publicField);

        // Cannot access privateField from Base (not inherited)
        // System.out.println("Method C: accessing privateField (not inherited) = " +
privateField);

        // Can access protectedField from Base (inherited through B)
        System.out.println("Method C: accessing protectedField (inherited) = " +
protectedField);
        }
}

public class Main {
        public static void main(String[] args) {
        A objA = new A();
        B objB = new B();
        C objC = new C();

        objA.methodA();
        objB.methodB();
        objC.methodC();
        }
}
```

12. Write a program to implement a class Teacher contains two fields Name and Qualification. Extend the class to Department, it contains Dept. No and Dept. Name. An Interface named as College it contains one field Name of the College. Using the above classes and Interface get the appropriate information and display it.

```
        public interface College {
        String getCollegeName();
}

public class Teacher {
        private String name;
```

```
	private String qualification;

	public Teacher(String name, String qualification) {
	this.name = name;
	this.qualification = qualification;
	}

	public String getName() {
	return name;
	}

	public String getQualification() {
	return qualification;
	}
}

public class Department extends Teacher {
	private int deptNo;
	private String deptName;

	public Department(String name, String qualification, int deptNo, String deptName) {
	super(name, qualification); // Call parent constructor to initialize inherited fields
	this.deptNo = deptNo;
	this.deptName = deptName;
	}

	public int getDeptNo() {
	return deptNo;
	}

	public String getDeptName() {
	return deptName;
	}
}

public class Main {
	public static void main(String[] args) {
	College college = new College() { // Anonymous inner class implementation
	@Override
	public String getCollegeName() {
		return "ABC College";
	}
	};

	Department department = new Department("John Doe", "M.Sc.", 101, "Computer
Science");

	System.out.println("College Name: " + college.getCollegeName());
```

```
        System.out.println("Department Details:");
        System.out.println("\tDepartment No: " + department.getDeptNo());
        System.out.println("\tDepartment Name: " + department.getDeptName());
        System.out.println("Teacher Details:");
        System.out.println("\tTeacher Name: " + department.getName()); // Inherited from
Teacher
        System.out.println("\tTeacher Qualification: " + department.getQualification()); //
Inherited from Teacher
        }
}
```