Q1:

public class Q1 {

public static void main(String[] args) {

Operations op = (a, b) -> a + b;

System.out.println(op.add(5, 3));

System.out.println(op.subtract(5, 3));

}

}

interface Operations {

int add(int a, int b);

int subtract(int a, int b);

}

**Question**: Review the program provided above and identify any errors within it. If you find no errors, state the expected output of the program. Explain your answer.

**Explanation**: Lambda expressions can only be used with functional interfaces, which have a single abstract method. In this case, the Operations interface has two abstract methods, add and subtract. As a result, a lambda expression cannot be used here because it cannot provide an implementation for both methods simultaneously.

**Mark:** 1 mark for error, 1 mark for correct explanation

Q2:

**public class Q2 {**

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

**Shape rectangle = new Shape() {**

**public double area() {**

**return 10.0;**

**}**

**};**

**System.out.println("Area: " + rectangle.area());**

**}**

**}**

**abstract class Shape {**

**abstract double area();**

**abstract double perimeter();**

**}**

**Question**: Review the program provided above and identify any errors within it. If you find no errors, state the expected output of the program. Explain your answer.

**Explanation**: The error here is that the Shape class is abstract and contains two abstract methods: area() and perimeter(). In the anonymous class, only the area() method is implemented, but perimeter() remains unimplemented.

**Mark:** 1 mark for error, 1 mark for correct explanation

Q3:

public class Q3{

public static void main(String[] args) {

MathOperations operations = (a, b) -> a \* b;

System.out.println(operations.multiply(5, 2));

System.out.println(operations.divide(10, 2));

}

}

interface MathOperations {

int multiply(int a, int b);

int divide(int a, int b);

}

**Question**: Review the program provided above and identify any errors within it. If you find no errors, state the expected output of the program. Explain your answer.

**Explanation**: Lambda expressions can only be used with functional interfaces, which have a single abstract method. In this case, the Operations interface has two abstract methods, add and subtract. As a result, a lambda expression cannot be used here because it cannot provide an implementation for both methods simultaneously.

**Mark:** 1 mark for error, 1 mark for correct explanation

Q4:

public class Q4 {

public static void main(String[] args) {

Vehicle car = new Vehicle() {

public String fuelType() {

return "Petrol";

}

};

System.out.println("Fuel Type: " + car.fuelType());

}

}

abstract class Vehicle {

abstract String fuelType();

abstract int numberOfWheels();

}

**Question**: Review the program provided above and identify any errors within it. If you find no errors, state the expected output of the program. Explain your answer.

**Explanation** The Vehicle abstract class has two abstract methods: fuelType() and numberOfWheels(). In the anonymous class, only the fuelType() method is implemented, but the numberOfWheels() method is not. Since all abstract methods in an abstract class must be implemented in an anonymous class, this leads to an error.

**Mark:** 1 mark for error, 1 mark for correct explanation

Q5:

public class Q5 {

public static void main(String[] args) {

Calculator calc = (a, b) -> {

return a + b;

};

System.out.println(calc.add(5, 3));

}

}

interface Calculator {

int add(int a, double b);

}

**Question**: Review the program provided above and identify any errors within it. If you find no errors, state the expected output of the program. Explain your answer.

**Explanation** The lambda expression here expects two int parameters, but the add method in the Calculator interface requires the second parameter to be a double. Since the lambda expression does not match the method signature (int, double), this will lead to a compile-time error.

**Mark:** 1 mark for error, 1 mark for correct explanation

Q6:

public class Q6 {

public static void main(String[] args) {

MathOperation operation = new MathOperation() {

@Override

public double compute(int x, int y) {

return x + y;

}

};

System.out.println("Result: " + operation.compute(5, 3));

}

}

interface MathOperation {

int compute(int a, int b);

}

**Question**: Review the program provided above and identify any errors within it. If you find no errors, state the expected output of the program. Explain your answer.

**Explanation** In this example, the Math Operation interface declares the compute method to return an int, but the anonymous class implementation returns a double. Since the return type of the method signature in the anonymous class (double) does not match the interface (int), this will result in a compile-time error.

**Mark:** 1 mark for error, 1 mark for correct explanation

Q7:

**public class Q7 {**

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

**StringProcessor processor = (str) -> str.length();**

**if (processor.process("Lambda Test")) {**

**System.out.println("String processed successfully.");**

**} else {**

**System.out.println("String processing failed.");**

**}**

**}**

**}**

**interface StringProcessor {**

**boolean process(String s);**

**}**

**Question**: Review the program provided above and identify any errors within it. If you find no errors, state the expected output of the program. Explain your answer.

**Explanation** In this code, the StringProcessor interface has a method process that returns a boolean, but the lambda expression in main returns an int (the length of the string). The mismatch between the return type of the lambda (int) and the expected return type of the interface method (boolean) leads to a compile-time error.

**Mark:** 1 mark for error, 1 mark for correct explanation

**Q8:**

public class Q8 {

public static void main(String[] args) {

Animal animal = new Animal() {

@Override

public void sound() {

System.out.println("Woof");

}

};

animal.sound();

animal.eat();

}

}

abstract class Animal {

abstract void sound();

void eat() {

System.out.println("Eating...");

}

}

**Question**: Review the program provided above and identify any errors within it. If you find no errors, state the expected output of the program. Explain your answer.

**Explanation** In this example, the Animal class has one abstract method sound() and one concrete method eat(). The anonymous class correctly implements the sound() method while also utilizing the inherited eat() method from the Animal class. The usage of both methods (one abstract, one concrete) is correct.

**Mark:** 1 mark for no error, 1 mark for correct explanation

**Q9:**

**public class Q9{**

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

**StringManipulator manipulator = new StringManipulator() {**

**@Override**

**public void manipulate(String str, int times) {**

**for (int i = 0; i < times; i++) {**

**System.out.println(str.toUpperCase());**

**}**

**}**

**};**

**manipulator.manipulate("Hello", 3);**

**}**

**}**

**interface StringManipulator {**

**String manipulate(String str);**

**}**

**Question**: Review the program provided above and identify any errors within it. If you find no errors, state the expected output of the program. Explain your answer.

**Explanation** In this code, the StringManipulator interface's manipulate method is expected to return a String and accept only one parameter (String str). However, the anonymous class method signature uses two parameters (String str, int times) and does not return any value (void). This mismatch between the method signature in the anonymous class and the interface leads to a compile-time error.

**Mark:** 1 mark for error, 1 mark for correct explanation

**Q10：**

**public class Q10{**

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

**StringProcessor processor = (str) -> str.toUpperCase();**

**String result = processor.process("lambda example");**

**System.out.println("Processed String: " + result);**

**System.out.println("Default Processed String: " + processor.addSuffix("lambda example"));**

**}**

**}**

**interface StringProcessor {**

**String process(String s);**

**// Default method in the interface**

**default String addSuffix(String s) {**

**return s + "\_DEFAULT\_SUFFIX";**

**}**

**}**

**Question**: Review the program provided above and identify any errors within it. If you find no errors, state the expected output of the program. Explain your answer.

**Explanation** In this example, the StringProcessor interface includes both an abstract method process() and a default method addSuffix(). The lambda expression correctly implements the process() method, which converts the input string to uppercase. The default method addSuffix() is also used to append a suffix to the string. Since default methods are allowed in interfaces, this usage is valid.

**Mark:** 1 mark for no error, 1 mark for correct explanation