Chapter 6 – Methods - Worksheet

1. **What are the three benefits of using a method?**
   Reusability: Methods allow code to be written once and reused multiple times throughout a program.
   Modularity: Methods help break down complex tasks into smaller, manageable units, making the code more organized, easier to understand, and maintain.
   Abstraction: Methods abstract away the implementation details of a particular functionality, allowing other parts of the code to interact with the method.

   a. **What does modularization do for code?**
      Modularization in code is the practice of breaking down a program into separate, self-contained modules or components. It helps in organizing code, making it more manageable, understandable, and maintainable. By dividing code into modules, each responsible for a specific task or functionality, it becomes easier to work on individual parts of the codebase without affecting others.
      i. **Is this why we want to use methods?**
         Yes, modularization is one of the reasons for using methods. By encapsulating specific functionality within methods, we can achieve code modularity and organization. Methods allow us to group related code together, which makes it easier to read, understand, and maintain. They promote reusability and abstraction, allowing us to focus on high-level functionality rather than implementation details.

2. **What is the return type of a `main` method?**
   The return type of the main method in Java is void. The main method serves as the entry point of a Java program and does not return any value explicitly.

3. **How many variables/values can be returned from a method?**
   In Java, a method can only return one variable/value. However, it is possible to return complex objects or collections that encapsulate multiple values.

4. **How many variables/values can be passed to a method?**
   A method in Java can accept any number of variables/values as parameters. It can have zero or more parameters, this allows for flexibility in passing information for the method to operate on.

5. **Identify the following terms on the following Java code:**
   a. Method name: whatToDo
   b. Method definition: public static String whatToDo(int n1, int n2, int n3) { ... }
   c. Invoking statement: System.out.println(whatToDo(2, 3, 6));
   d. Method header: public static String whatToDo(int n1, int n2, int n3)
   e. Method body: The code enclosed within the curly braces { ... } in the method definition.
   f. Modifiers: public, static
   g. Return type: String
   h. Method signature: whatToDo(int n1, int n2, int n3)
   i. Formal parameters: int n1, int n2, int n3
   j. Actual parameters: 2, 3, 6

```
public class Identify{
     public static void main(String [] args){
          System.out.println(whatToDo(2, 3, 6));
```

```
        }
        public static String whatToDo(int n1, int n2, int n3){
             return "nothing";
        }
    }
```

6. **In the method signature, does each parameter have to have its own data type listed?**
   Yes, in the method signature, each parameter should have its own data type listed. This helps to specify the type of data that the method expects to receive when it is called.

7. **In the invoking statement, does each parameter have to have its own data type listed?**
   No, in the invoking statement, you don't need to explicitly specify the data type for each parameter. The data types are already defined in the method signature. You just need to provide the values that match the parameter types in the correct order.

8. **If a value is returned from a method, what does the invoking statement have or be able to do?**
   When a value is returned from a method, the invoking statement needs to be able to receive and store that returned value. It can then use the returned value for further processing or assign it to a variable.

9. **When is the `return` keyword used?**
   The return keyword is used within a method to explicitly exit the method and return a value (if the method has a return type). It can be used at any point in the method where you want to stop the execution and return a specific value.

10. **Where should the `return` keyword be placed in the method?**
    The return keyword should be placed at the point in the method where you want to exit the method and return a value. Typically, it is placed after the necessary computations or operations have been performed within the method.

11. **Do all methods have to use the `return` keyword?**
    No, not all methods have to use the return keyword. It depends on whether the method is designed to return a value or not. If a method doesn't have a return type specified in its signature (void), it doesn't need to use the return keyword.

12. **What does the invoking/calling statement do?**
    The invoking or calling statement is used to execute a method. It passes the required arguments (values) to the method's parameters and triggers the execution of the method's code. The invoking statement is responsible for starting the method's execution and can receive any returned value if applicable.

13. **If there is not an invoking/calling statement to a method, is that method executed?**
    No, if there is no invoking or calling statement for a method, the method will not be executed. In Java, methods are executed when they are explicitly called or invoked. The program flow will only reach and execute the code within a method if there is a calling statement that triggers its execution.

14. **Identify and correct the errors in the following program:**

```
public class ProgName{
```

```
   public static method1(int n, m){
        n+= m;
        method2(3.4);
   }

   public static int method2(int n){
        if (n > 0) return 1;
        else if (n == 0) return 0;
        else if(n < 0) return -1;
   }
}
```

Corrected

```
public class ProgName {

    public static void method1(int n, int m) {
        n += m;
        method2(3.4);
    }

    public static int method2(double n) {
        if (n > 0) return 1;
        else if (n == 0) return 0;
        else if (n < 0) return -1;
        return 0; // Added a default return statement since all
conditions are not covered.
    }
}
```

Error in method1:

method1 is missing a return type. It should have a return type specified, or a void return type.

The parameters n and m in the method1 method are missing their data types.

Error in method2:

method2 is declared to return an int value, but there is no default return statement outside of the if-else if conditions.

argument 3.4 in the invoking statement does not match the parameter type of int in method2.

15. Write method headers and the invoking statement for:

a. Compute a sales commission, given the sales amount and the commission rate.

Method Header:
public static double computeSalesCommission(double salesAmount, double commissionRate) { // Method code}

Invoking Statement:
double commission = computeSalesCommission(5000.0, 0.05);

b. Compute a square root of a number.

Method Header:
public static double computeSquareRoot(double number) { // Method code}

Invoking Statement:
double squareRoot = computeSquareRoot(16.0);

c. Display a message a specified number of times.

Method Header:
public static void displayMessageMultipleTimes(String message, int times) { // Method code}

Invoking Statement:
displayMessageMultipleTimes("Hello, World!", 5);

d. Return the area of a square, given the length of the side

Method Header:
public static double calculateSquareArea(double sideLength) { // Method code}

Invoking Statement:
double area = calculateSquareArea(4.5);

e. Return the average of 4 whole numbers

Method Header:
public static double calculateAverage(int num1, int num2, int num3, int num4) { // Method code}

Invoking Statement:
double average = calculateAverage(85, 90, 92, 88);

f. Return the payment amount, given the total price, the interest rate, and how many years the loan will be

Method Header:
public static double calculatePaymentAmount(double totalPrice, double interestRate, int years) { // Method code}

Invoking Statement:
double payment = calculatePaymentAmount(20000.0, 0.08, 5);

g. Printing the date, nothing passed in and nothing returned

Method Header:
public static void printDate() { // Method code}

Invoking Statement:
printDate();

h. Return the percent of a decimal number

Method Header:
public static double convertDecimalToPercent(double decimal) { // Method code}

Invoking Statement:
double percent = convertDecimalToPercent(0.75);

i. Returns a description of how well you did on a test score

Method Header:
public static String getTestScoreDescription(int score) { // Method code}

Invoking Statement:
String description = getTestScoreDescription(85);

16. **Why do we want to be able to overload methods?**

Method overloading allows us to define multiple methods with the same name but different parameters within the same class. This feature provides flexibility and improves code readability.

why might we want to be able to overload methods? Well:
It allows us to provide different ways of using a method with varying parameter types or numbers.
It enhances code reusability by allowing us to define similar operations with different inputs.
It promotes clean and intuitive code as method names can be consistent, regardless of the specific variations in parameters.
It simplifies the calling process for users since they can use the same method name with different arguments based on their needs.

17. How does one overload methods?

To overload a method:
Define multiple methods with the same name within the class.
Ensure that the methods have different parameter lists (number of parameters, parameter types, or both).
Java will determine the appropriate method to execute based on the arguments provided at the method invocation.

18. What is the scope of a variable?

In Java, variables can have different scopes, such as:
Method scope: Variables declared within a method are accessible only within that method.
Block scope: Variables declared within a code block (enclosed within curly braces) are accessible only within that block.
Class scope: Variables declared at the class level (outside any method) are accessible throughout the class.
Instance scope: Instance variables are declared within a class but outside any method and are accessible throughout the class's instances (objects).

19. Using the following code, identify the scope of x.

```
public class Scope{
  public static void main(String [] args){
       int y = 9;
       String s = "word";
       if(y == 0){
            System.out.println("y is 0");
            int x = y;
       }
  }
}
```

the scope of variable x is limited to the if block. It means that x can only be accessed within the if block where it is declared. Outside of the if block, x is not visible or accessible. The variable x is initialized with the value of y within the if block, but it cannot be accessed or used elsewhere in the code.