Assignment 2

Student Name:	Grader Name:
Student UIN:	Grader UIN:

Reading Assignment: How to Program Java, 9th edition

- Chapter 4 – Control Statements: Part 1
- Chapter 5 – Control Statements: Part 2
- Chapter 6 – Methods: A Deeper Look
- Chapter 27 – Networking (Sections 27.4, 27.5, 27.6)

True or False:

1. Java is a strongly typed language.
2. Primitive types in Java are not guaranteed to be identical from computer to computer.

Short Questions:

1. Java has only three kinds of control structures: the sequence statement, selection statements (three types) and repetition statements (three types). Name the three types of repetition statements.
2. Explain the syntax of the conditional operator (?:).
3. The compound assignment operators abbreviate assignment expressions. If `c = 7;` is followed by `c += 3;`, what is the new value of `c`?
4. Assume `int a = 4, b = 5, c = 6, d = 15, e = 17`. Complete the following table with the proper integer value.

Expression	Explanation	Assignment
`a += 6`	`a = a + 6`	10 to `a`
`b -= 2`		
`c *= 3`		
`d /= 4`		
`e %= 5`		

5. What is wrong with the following `while` statement?

```
while ( x >= 0 )
  sum += x;
```

6. Suppose that the initialization expression in the `for` header declares the control variable (i.e., the control variable's type is specified before the variable name). What is the scope of this variable and where can it be employed?

7. Write a `for` header with control variable `index` such that it meets the following requirements.

   (a) Variable `index` varies from 1 to 10 in increments of 1.

   (b) Variable `index` varies from 10 to 1 in decrements of 1.

   (c) Variable `index` varies from 1 to 256 in powers of 2.

8. What is a *utility method* (or *helper method*)?

9. Describe the roles of the three expressions in the header of a `for` statement, i.e., `for ( expression1 ; expression 2 ; expression 3 )`.

10. What happens when a local variable or parameter in a method has the same name as a field?

11. What is *method overloading*?

Programming Challenges

This programming challenge explores the use of stream sockets.

Simple Server Using Stream Sockets

Establishing a simple server in Java requires five steps.

1. Create a `ServerSocket` object:

   ```
   ServerSocket server = new ServerSocket(portNumber, queueLength);
   ```

   where `portNumber` is an admissible TCP port number and `queueLength` is the maximum number of clients that can wait to connect to the server.

2. Wait for a connection:

   ```
   Socket connection = server.accept();
   ```

   In this step, the server listen indefinitely for an attempt by a client to connect. The method returns a `Socket` when a connection with a client is established.

3. Manage the I/O streams associated with the socket:

```
connection.getOutputStream();
connection.getInputStream();
```

These objects can subsequently be employed to send or receive bytes with the `OutputStream` method `write` and the `InputStream` method `read`, respectively. One can also use classes such `ObjectInputStream` and `ObjectOutputStream` to enable entire objects to be read from or written to a stream, a technique called wrapping.

4. Perform the processing: In the processing phase, the server and the client communicate via the `OutputStream` and `InputStream` objects.

5. Closing the connection: The server closes the connection by invoking the `close` method on the streams and on the `Socket`.

Simple Client Using Stream Sockets

Establishing a simple client in Java necessitates four steps.

1. Create a `Socket` to connect to the server:

   ```
   Socket connection = new Socket( serverAddress, port);
   ```

   When the connection attempt is successful, this returns a `Socket`.

2. Manage the I/O streams.

3. Perform the processing.

4. Close the connection: The client closes the connection by invoking the `close` method on the streams and on the `Socket`.

Challenge

In this challenge, you will enhance the `Java1` application by performing the computation on a server. That is, you should program a `Java2Client` application that present an interface similar to `Java1`, and a `Java2Server` application that takes two integers and returns the sum. This should be implemented using stream sockets and the paradigm discussed above. Specifically, `Java2Client` should not sum the two integers locally. Rather, it should contact a `Java2Server`, establish a connection, send the two integers, and wait for their sum. In a complementary fashion, `Java2Server` should listen for a connection, read the two integers and return the sum. Commit your server code as a project labeled `Java2Server`, and your client code as a project labeled `Java2Client`.