Question 1

a) What do each of the following print?

i) `System.out.println(2 + "ru");`

- This concatenates the integer 2 with the string "ru". The output is: `2ru`

ii) `System.out.println(2 + 3 + "ru");`

- This adds the integers 2 and 3 first (because of operator precedence), resulting in 5, and then concatenates the result with the string "ru". The output is: `5ru`

iii) `System.out.println((2+3) + "ru");`

- This adds the integers 2 and 3 first (due to the parentheses), resulting in 5, and then concatenates the result with the string "ru". The output is: `5ru`

iv) `System.out.println("ru" + (2+3));`

- This adds the integers 2 and 3 first (due to the parentheses), resulting in 5, and then concatenates the string "ru" with the result. The output is: `ru5`

v) `System.out.println("ru" + 2 + 3);`

- This concatenates the string "ru" with the integer 2 first, resulting in "ru2", and then concatenates the result with the integer 3. The output is: `ru23`

b) Name the import package required to perform input and output in Java.

- The import package required is `java.util.Scanner`.

c) Using an example, differentiate between a local variable, an instance variable, and a class variable.

- Local Variable: A variable that is declared inside a method and is only accessible within that method.

```
public void myMethod() {
    int localVar = 10; // local variable
}
```

- **Instance Variable**: A variable that is declared inside a class but outside any method and is accessible by all methods in the class.

```
public class MyClass {
    int instanceVar = 20; // instance variable

    public void myMethod() {
        System.out.println(instanceVar);
    }
}
```

- **Class Variable**: A variable that is declared as `static` inside a class, meaning it is shared among all instances of the class.

```
public class MyClass {
    static int classVar = 30; // class variable

    public void myMethod() {
        System.out.println(classVar);
    }
}
```

d) The below Java code will test if a number is positive or negative. Write down the missing code snippets.

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    System.out.print("Input number: ");
    int input = in.nextInt();

    // Missing code snippets
    if (input > 0) {
        System.out.println("The number is positive.");
    } else if (input < 0) {
        System.out.println("The number is negative.");
    } else {
        System.out.println("The number is zero.");
    }
}
```

Question 2

a) Define an array.

An array is a data structure that can hold a fixed number of values of a single type. The elements in an array are indexed, with the first element at index 0.

b) When an array is created, each element of the array is set to the initial value of its type. What is the initial value of each of the following types?

i) **For numeric type**: The initial value is `0`.

ii) **For Boolean type**: The initial value is `false`.

iii) **For reference type**: The initial value is `null`.

c) In this question you will be asked to refer to the below code snippets.

i) Write down the missing code A that declares an array of integers.

```
int[] arr; // Missing code A
```

ii) Write down the missing code B that allocates memory for 5 integers.

```
arr = new int[5]; // Missing code B
```

iii) Write down the missing code C that creates a loop and displays the elements in the array as output in this syntax `Element at index i: value`.

```
for (int i = 0; i < arr.length; i++) {
    System.out.println("Element at index " + i + ": " + arr[i]);
} // Missing code C
```

So, the full code snippet will look like this:

```java
class GFG {
    public static void main(String[] args) {
        // declares an Array of integers.
        int[] arr; // Missing code A

        // allocating memory for 5 integers.
        arr = new int[5]; // Missing code B

        // initialize the first elements of the array
        arr[0] = 10;
        // initialize the second elements of the array
        arr[1] = 20;
        // so on...
        arr[2] = 30;
        arr[3] = 40;
        arr[4] = 50;

        // accessing the elements of the specified array
        for (int i = 0; i < arr.length; i++) {
            System.out.println("Element at index " + i + ": " + arr[i]); // Missing code C
        }
    }
}
```

Question 3

a) Differentiate between a class and an object.

- Class: A class is a blueprint or template for creating objects. It defines a type by bundling data and methods that work on the data into one single unit. A class does not occupy any memory space until an object is created from it.
- Object: An object is an instance of a class. It is a concrete entity based on the class, which occupies memory space. An object can use the methods and properties defined in the class.

b) List the principal characteristics of an object, as defined in the object model, and how are these characteristics represented in Java?

1. Identity: The property that distinguishes each object from other objects. In Java, this is represented by the unique memory address assigned to each object. The identity can be checked using the `==` operator.
2. State: The data or values stored in an object at any point in time. In Java, this is represented by the instance variables of a class.
3. Behavior: The methods or functions that an object can perform. In Java, behavior is represented by methods defined within the class.

c) Consider the following diagram. Describe in your own words the information contained in the diagram about the classes and their relationship.

The diagram shows a hierarchical relationship between the classes `Animal`, `Mammal`, and `Reptile`. `Animal` is the superclass (or base class), and it is the parent of `Mammal` and `Reptile`, which are subclasses (or derived classes) of `Animal`. This indicates that both `Mammal` and `Reptile` inherit properties and behaviors from `Animal`. The inheritance relationship allows `Mammal` and `Reptile` to reuse code from `Animal` and possibly override some of the behaviors defined in `Animal`.

d) Write a Java code example to demonstrate method overriding based on the above diagram.

```
class Animal {
    // Method to be overridden
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Mammal extends Animal {
    // Overriding the sound method
    @Override
    void sound() {
        System.out.println("Mammal makes a sound");
    }
}

class Reptile extends Animal {
    // Overriding the sound method
    @Override
    void sound() {
        System.out.println("Reptile makes a sound");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal(); // Animal reference and object
        Animal myMammal = new Mammal(); // Animal reference but Mammal object
        Animal myReptile = new Reptile(); // Animal reference but Reptile object

        myAnimal.sound(); // Calls Animal's sound method
        myMammal.sound(); // Calls Mammal's sound method
        myReptile.sound(); // Calls Reptile's sound method
    }
}
```

Question 4

a) Define an exception in Java and how is it handled?

An exception in Java is an event that disrupts the normal flow of the program. It is an object that the Java Virtual Machine (JVM) or the application code can throw when an abnormal condition arises. Exceptions are used to handle errors and other exceptional events in a controlled manner.

Exceptions in Java are handled using `try-catch` blocks. The code that might throw an exception is placed inside the `try` block, and the code to handle the exception is placed inside the `catch` block. If an exception occurs, the control is transferred to the appropriate `catch` block.

b) What type of exception have occurred in the given code?

The given code can throw two types of exceptions:

1. `FileNotFoundException` if the file specified in the `FileReader` constructor is not found.
2. `IOException` if an input/output operation fails or is interrupted.

c) Write down the missing line of codes for:

i) **Exception 1**

The missing line for `Exception 1` is:

```
catch (FileNotFoundException e) {
    System.err.println("File not found");
}
```

ii) **Exception 2**

The missing line for `Exception 2` is:

```
catch (IOException e) {
    System.err.println("Unable to read the file.");
}
```

d) Instead of using the try-catch block to handle the exception, mention another way of declaring this exception through an example.

Another way to handle exceptions is to declare them using the `throws` keyword in the method signature. This passes the responsibility of handling the exception to the calling method.

Example:

```
import java.io.*;

public class Example {
    public static void main(String[] args) {
        try {
            readFile();
        } catch (FileNotFoundException e) {
            System.err.println("File not found");
        } catch (IOException e) {
            System.err.println("Unable to read the file.");
        }
    }

    public static void readFile() throws FileNotFoundException, IOException {
        BufferedReader br = new BufferedReader(new FileReader("/home/students/test.txt"));
        String strLine;
        while ((strLine = br.readLine()) != null) {
            System.out.println(strLine);
        }
        br.close();
    }
}
```

e) Why is it necessary to add a finally block when handling an exception?

It is necessary to add a `finally` block when handling an exception to ensure that certain code is always executed, regardless of whether an exception is thrown or not. The `finally` block is commonly used to close resources such as files or database connections, ensuring that these resources are released properly to avoid resource leaks.

Example:

```
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("/home/students/test.txt"));
    String strLine;
    while ((strLine = br.readLine()) != null) {
        System.out.println(strLine);
    }
} catch (FileNotFoundException e) {
    System.err.println("File not found");
} catch (IOException e) {
    System.err.println("Unable to read the file.");
} finally {
    if (br != null) {
        try {
            br.close();
        } catch (IOException e) {
            System.err.println("Error closing the file.");
        }
    }
}
```

The `finally` block ensures that the `BufferedReader` is closed even if an exception occurs.

Question 5

a) Refer to the below diagram, write down the label A and B.

- A: Object
- B: Byte stream

b) Differentiate between serialization and deserialization.

- Serialization: The process of converting an object into a byte stream so that it can be easily saved to a file, sent over a network, or stored in a database. It captures the state of an object.
- Deserialization: The reverse process of converting a byte stream back into a copy of the original object. It reconstructs the object from the byte stream.

c) How to make a Java class serializable?

To make a Java class serializable, the class must implement the `Serializable` interface. This is a marker interface and does not contain any methods. Here is an example:

```
import java.io.Serializable;

public class MyClass implements Serializable {
    private static final long serialVersionUID = 1L;
    private int id;
    private String name;

    // Constructors, getters, and setters
}
```

d) While serializing you want some of the members not to serialize. How do you achieve it?

To exclude certain members from being serialized, you can mark them as `transient`. The `transient` keyword in Java is used to indicate that a field should not be serialized.

Example:

```
import java.io.Serializable;

public class MyClass implements Serializable {
    private static final long serialVersionUID = 1L;
    private int id;
    private String name;
    private transient String password; // This field will not be serialized

    // Constructors, getters, and setters
}
```

e) Name four (4) Java GUI components.

1. **JButton**: A push button used to perform an action when clicked.
2. **JLabel**: A display area for a short text string or an image, or both.
3. **JTextField**: A single-line text field that allows the user to enter text.
4. **JPanel**: A generic container that can store a group of components, including other panels.