Java Methods - Lab

Title: The Quest of the Java Sorcerer

Plot:

In a mystical realm, there existed a Java Sorcerer known for his prowess in harnessing the power of methods. This sorcerer sought an apprentice, and the chosen one was a budding Java enthusiast named Alex. As Alex embarked on this journey, the sorcerer unveiled an ancient scroll foretelling a looming crisis. The once-peaceful kingdom was shrouded in darkness due to a malicious code that disrupted the flow of magic through the realm.

Question 1: The Scroll's Riddle

Upon unraveling the scroll, Alex encountered a riddle: "In the realm of strings and selections, find the method to dispel the code's curse. The secret lies in breaking the spell with a code that identifies and replaces specific elements. Write a Java method to replace all occurrences of a certain character within a given string."

Objective: Create a method that replaces all occurrences of a specified character in a given string.

Technical Instructions: - Define a method named `replaceCharacter` that takes in three parameters: a string (`inputStr`), a character to be replaced (`targetChar`), and a replacement character (`replacementChar`). - Implement the method to traverse through the `inputStr`, replacing all instances of `targetChar` with `replacementChar`. - Return the modified string after replacement.

Java Template:

```java
public class StringReplacer {
    // Define the method to replace characters in a string
    public static String replaceCharacter(String inputStr, char targetChar, char replacementChar) {
        // Iterate through inputStr and replace all instances of targetChar with replacementChar

        // Return the modified string
    }

    // Main method for testing
    public static void main(String[] args) {
        // Test your method here
        String input = "sample string"; // Provide the input string
        char target = 's'; // Provide the target character to replace
        char replacement = 'x'; // Provide the replacement character

        String modifiedString = replaceCharacter(input, target, replacement);
        System.out.println("Modified String: " + modifiedString);
    }
}
```

Sample Output:

```
Modified String: xample xtring
```

Plot Progression:

```java
public class StringReplacer {
    // Define the method to replace characters in a string
    public static String replaceCharacter(String inputStr, char targetChar, char replacementChar) {
        String modifiedString = "";

        for (int i = 0; i < inputStr.length(); i++) {
            char currentChar = inputStr.charAt(i);
            if (currentChar == targetChar) {
                modifiedString += replacementChar;
            } else {
                modifiedString += currentChar;
            }
        }

        return modifiedString;
    }

    // Main method for testing
    public static void main(String[] args) {
        // Test your method here
        String input = "sample string"; // Provide the input string
        char target = 's'; // Provide the target character to replace
        char replacement = 'x'; // Provide the replacement character

        String modifiedString = replaceCharacter(input, target, replacement);
        System.out.println("Modified String: " + modifiedString);
    }
}
```

With determination, Alex delved into his Java knowledge. After pondering and experimenting, Alex crafted a method capable of replacing all instances of the malevolent character. The sorcerer nodded in approval, and as the curse began to lift, a faint glimmer of hope rekindled in the kingdom.

Question 2: Artifacts' Deciphering

While traversing the realm, Alex and the Java Sorcerer encountered enchanted artifacts that held the essence of ancient knowledge. To unlock their secrets, the Sorcerer unveiled that a series of interlinked methods needed crafting. These methods would decode the artifacts' enchantments and reveal the wisdom they concealed. Each method relied on the output of the previous one, forming a chain of knowledge crucial for unraveling the artifacts' mysteries.

Objective: Create a series of methods to decode enchanted artifacts by performing specific transformations on the artifacts' essence.

Technical Instructions: - Define three methods: `extractEssence`, `purifyEssence`, and `decodeArtifact`. - `extractEssence` should take a string parameter `artifact` and return a modified string with the essence extracted. - `purifyEssence` should take the extracted essence and return a purified version. - `decodeArtifact` should utilize the output of `purifyEssence` and perform the final transformation, returning the decoded artifact.

Java Template:

```java
public class ArtifactDecoder {
    // Define method to extract essence from the artifact
    public static String extractEssence(String artifact) {
    // Implement logic to extract essence from the artifact
    // Example: extracting essence by removing first and last 9 characters
    // Return modified string with essence extracted
    return ""; // Placeholder return statement
    }

    // Define method to purify the extracted essence
    public static String purifyEssence(String extractedEssence) {
    // Implement logic to purify the extracted essence
    // Example: converting to lowercase
    // Return purified essence
    return ""; // Placeholder return statement
    }

    // Define method to decode the artifact using purified essence
    public static String decodeArtifact(String purifiedEssence) {
    // Implement logic to decode the artifact using purified essence
    // Example: reversing the string
    // Return the decoded artifact
    return ""; // Placeholder return statement
    }

    // Main method for testing
    public static void main(String[] args) {
    // Test your methods by calling them in a sequence
    String artifact = "Enchanted artifact string"; // Provide the enchanted artifact

    String extractedEssence = extractEssence(artifact);
    String purifiedEssence = purifyEssence(extractedEssence);
    String decodedArtifact = decodeArtifact(purifiedEssence);

    System.out.println("Decoded Artifact: " + decodedArtifact);
    }
}
```

Sample Output:

```
Decoded Artifact: tcafitra
```

Plot Progression:

```java
public class ArtifactDecoder {
    // Define method to extract essence from the artifact
    public static String extractEssence(String artifact) {
        // Implement logic to extract essence from the artifact
        return artifact.substring(9, artifact.length() - 7); // Example: extracting essence by removing first ar
    }

    // Define method to purify the extracted essence
    public static String purifyEssence(String extractedEssence) {
        // Implement logic to purify the extracted essence (example: converting to lowercase)
        return extractedEssence.toLowerCase();
    }

    // Define method to decode the artifact using purified essence
    public static String decodeArtifact(String purifiedEssence) {
        // Implement logic to decode the artifact using purified essence (example: reversing the string)
        return new StringBuilder(purifiedEssence).reverse().toString();
    }

    // Main method for testing
    public static void main(String[] args) {
        // Test your methods by calling them in a sequence
        String artifact = "Enchanted artifact string"; // Provide the enchanted artifact

        String extractedEssence = extractEssence(artifact);
        String purifiedEssence = purifyEssence(extractedEssence);
        String decodedArtifact = decodeArtifact(purifiedEssence);

        System.out.println("Decoded Artifact: " + decodedArtifact);
    }
}
```

As Alex delved into the crafting of the methods, they realized the intricate connection between `extractEssence`, `purifyEssence`, and `decodeArtifact`. The `extractEssence` method isolated the artifact's essence, feeding it into `purifyEssence`, which meticulously refined it. Finally, `decodeArtifact` utilized the purified essence, unraveling the encrypted wisdom within the artifact. Each method's output seamlessly flowed into the next, revealing the long-hidden knowledge contained within the enchanted artifacts.

Question 3: The Creatures' Enchantment

As Alex continued the journey with the Java Sorcerer, they encountered mythical creatures affected by a strange anomaly. These creatures, once peaceful, were now aggressive due to the disruption in their spellcasting. The Sorcerer revealed that only a specific method could counter these spells and restore the creatures' serenity. As they proceeded, they encountered various creatures, each with different spell types and vulnerabilities.

Objective: Create a method to calculate damage based on the spell type and its multiplier.

Technical Instructions: - Define a method named `calculateDamage` that takes two parameters: `spellType` (a string) and `multiplier` (a double). - Use conditional statements to evaluate different `spellType` cases and calculate the damage based on the provided `multiplier`. - Return the calculated damage as an integer.

Java Template:

```java
public class SpellDamageCalculator {
    // Define the method to calculate damage based on spell type and multiplier
    public static int calculateDamage(String spellType, double multiplier) {
        // Use conditional statements to calculate damage based on spellType and multiplier

        // Return the calculated damage as an integer
        return 0; // Placeholder return statement
    }

    // Main method for testing
    public static void main(String[] args) {
        // Test your method here by calling it multiple times with different parameters
        int damage1 = calculateDamage("Fire", 1.5); // Calculate damage for Fire spell with a 1.5 multiplier
        System.out.println("Damage 1: " + damage1);

        int damage2 = calculateDamage("Ice", 1.2); // Calculate damage for Ice spell with a 1.2 multiplier
        System.out.println("Damage 2: " + damage2);

        // Calculate damage for other spell types and multipliers
        // ...
    }
}
```

Sample Output:

```
Damage 1: 15
Damage 2: 12
```

Plot Progression:

```java
public class SpellDamageCalculator {
    // Define the method to calculate damage based on spell type and multiplier
    public static int calculateDamage(String spellType, double multiplier) {
        int baseDamage = 10; // A base damage value for all spells

        // Use conditional statements to calculate damage based on spellType and multiplier
        if (spellType.equals("Fire")) {
            return (int) (baseDamage * multiplier); // Fire spell damage calculation
        } else if (spellType.equals("Ice")) {
            return (int) (baseDamage * multiplier); // Ice spell damage calculation
        } else {
            // Handle other spell types or provide a default damage value
            return 0; // Placeholder for handling other spells
        }
    }

    // Main method for testing
    public static void main(String[] args) {
        // Test your method here by calling it multiple times with different parameters
        int damage1 = calculateDamage("Fire", 1.5); // Calculate damage for Fire spell with a 1.5 multiplier
        System.out.println("Damage 1: " + damage1);

        int damage2 = calculateDamage("Ice", 1.2); // Calculate damage for Ice spell with a 1.2 multiplier
        System.out.println("Damage 2: " + damage2);

        // Calculate damage for other spell types and multipliers
        // ...
    }
}
```

As Alex implemented the `calculateDamage` method, they encountered creatures casting various spells. Using the method, Alex successfully calculated the damage inflicted by the Fire spell with a 1.5 multiplier, dealing devastating blows to fire-vulnerable creatures. Furthermore, by utilizing the method for an Ice spell with a 1.2 multiplier, Alex was able to decipher its effect on ice-attuned beings. Each successful use of the method unveiled the power of defining and reusing methods across diverse situations, showcasing the versatility and efficiency of Java's method implementation.

As Alex's proficiency in Java methods grew, so did the brightness of hope in the realm. The sorcerer beamed with pride, acknowledging the apprentice's remarkable progress. With each challenge overcome, the darkness faded, paving the way for the restoration of peace and magic within the kingdom.