Presented to the College of Computer Studies

De La Salle University - Manila

Term 2, A.Y. 2024-2025

In partial fulfillment of the course CSARCH2

In SUBJECT SECTION S15

Simulation Project - Writeup Analysis

Group No. 9

Submitted by:

Ang, Mark Kevin

Quiñones, Angelo

Reyes, Alroy Leon

Suba, Kaye Diosa

Tan, Jiliana Amibelle

Submitted to:

Mr. Roger Uy

March 23, 2024

I. Introduction

In this paper, the group presents an implementation of a Unicode converter, with a particular emphasis on UTF (Unicode Transformation Format) and Unicode representation conversion. Our converter bridges the gap between various encoding systems and enables effective text processing and manipulation by converting text characters encoded in UTF to their matching Unicode code points and vice versa. Our Unicode converter prioritizes user experience by integrating Java GUI, which not only makes text character encoding and decoding easier but also guarantees a fluid and intuitive interaction approach. Our approach seeks to improve text processing activities' efficacy and efficiency by fusing strong functionality with an easy-to-use graphical user interface.

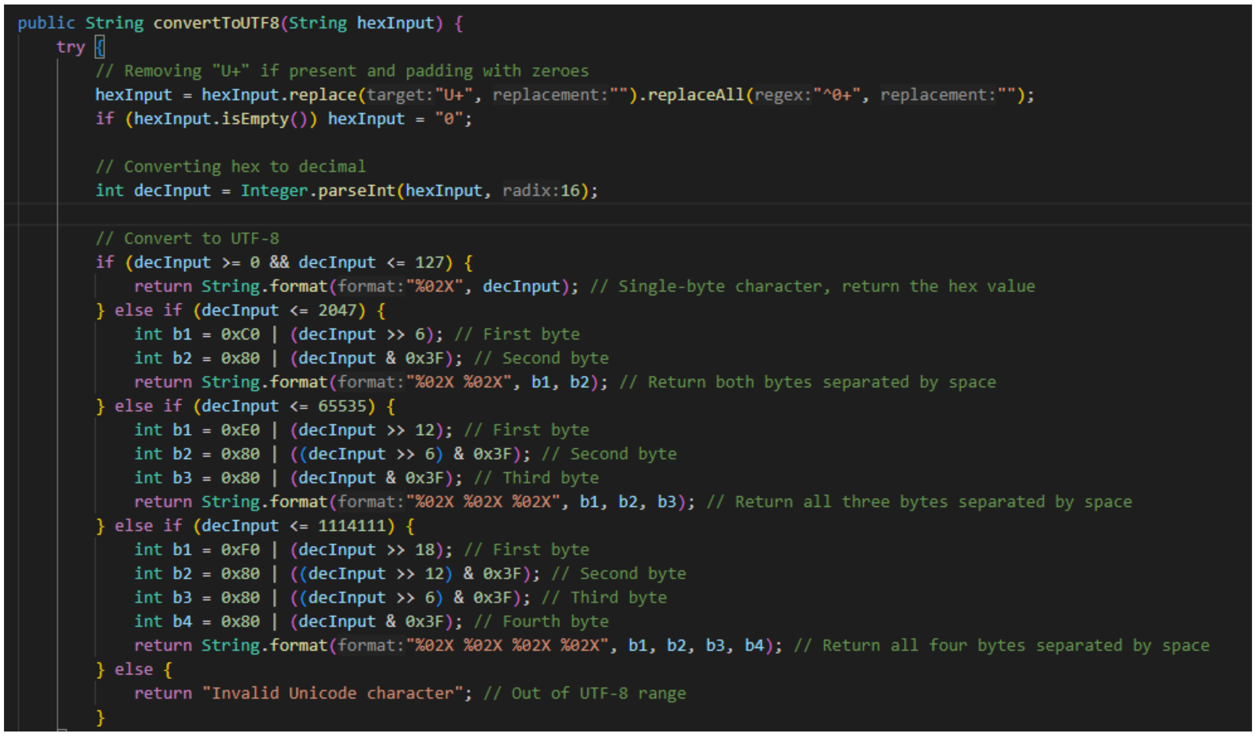
II. Specifications

In order to guarantee the application's usability, functionality, and compatibility with a variety of encoding systems, precise specifications were followed during the development process. The application is made to accept Unicode input for the **converter** feature, and it has strong validation checks in place to make sure that the input contains only legitimate Unicode characters. After conversion, the program outputs data in UTF-8, UTF-16, and UTF-32 formats, using hexadecimal notation to represent each byte. Users can also choose to store the conversion results to a text file for later study or reference. Likewise, with regard to the **translator** feature, the program generates equivalent Unicode code point outputs after accepting input encoded in UTF-8, UTF-16, or UTF-32 formats. You can save this output to a text file as well.

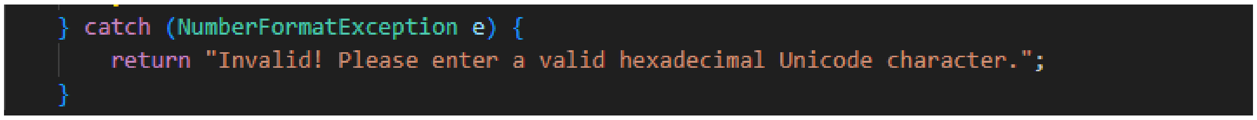
III. Methodology

Converter

The converter class contains a set of functions that are used to implement the Unicode converter application. Every function has a responsibility for translating Unicode characters into the appropriate UTF (Unicode Transformation Format) representations, making sure that everything is done correctly and consistently.



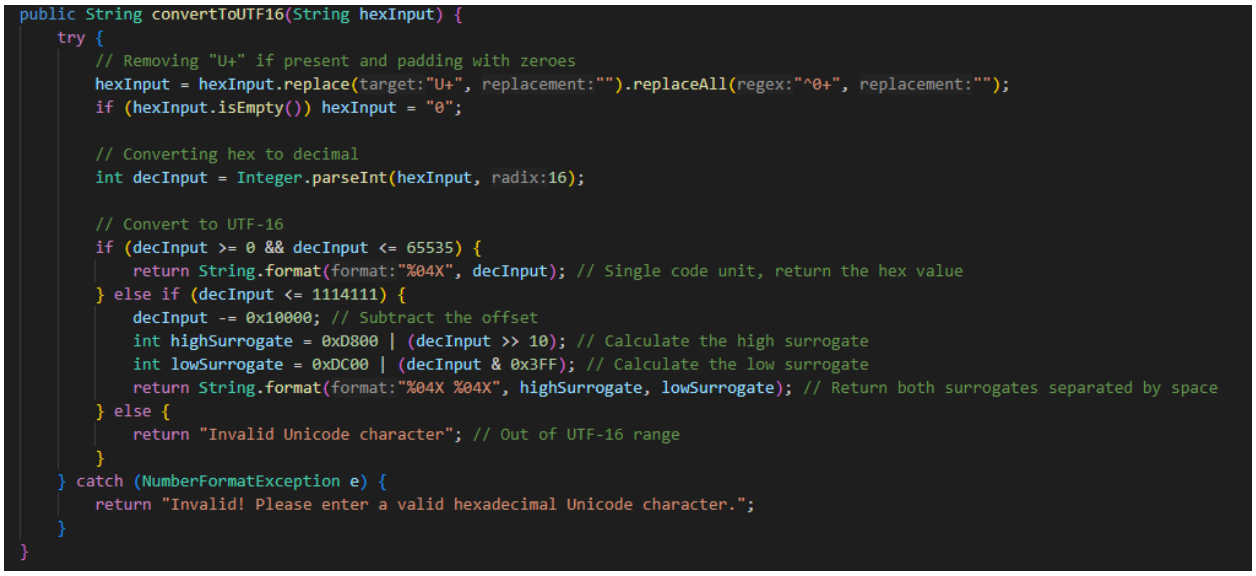
```java
public String convertToUTF8(String hexInput) {
    try {
        // Removing "U+" if present and padding with zeroes
        hexInput = hexInput.replace(target:"U+", replacement:"").replaceAll(regex:"^0+", replacement:"");
        if (hexInput.isEmpty()) hexInput = "0";

        // Converting hex to decimal
        int decInput = Integer.parseInt(hexInput, radix:16);

        // Convert to UTF-8
        if (decInput >= 0 && decInput <= 127) {
            return String.format(format:"%02X", decInput); // Single-byte character, return the hex value
        } else if (decInput <= 2047) {
            int b1 = 0xC0 | (decInput >> 6); // First byte
            int b2 = 0x80 | (decInput & 0x3F); // Second byte
            return String.format(format:"%02X %02X", b1, b2); // Return both bytes separated by space
        } else if (decInput <= 65535) {
            int b1 = 0xE0 | (decInput >> 12); // First byte
            int b2 = 0x80 | ((decInput >> 6) & 0x3F); // Second byte
            int b3 = 0x80 | (decInput & 0x3F); // Third byte
            return String.format(format:"%02X %02X %02X", b1, b2, b3); // Return all three bytes separated by space
        } else if (decInput <= 1114111) {
            int b1 = 0xF0 | (decInput >> 18); // First byte
            int b2 = 0x80 | ((decInput >> 12) & 0x3F); // Second byte
            int b3 = 0x80 | ((decInput >> 6) & 0x3F); // Third byte
            int b4 = 0x80 | (decInput & 0x3F); // Fourth byte
            return String.format(format:"%02X %02X %02X %02X", b1, b2, b3, b4); // Return all four bytes separated by space
        } else {
            return "Invalid Unicode character"; // Out of UTF-8 range
        }
```



```java
    } catch (NumberFormatException e) {
        return "Invalid! Please enter a valid hexadecimal Unicode character.";
    }
```

Figure 1: UTF8 Conversion

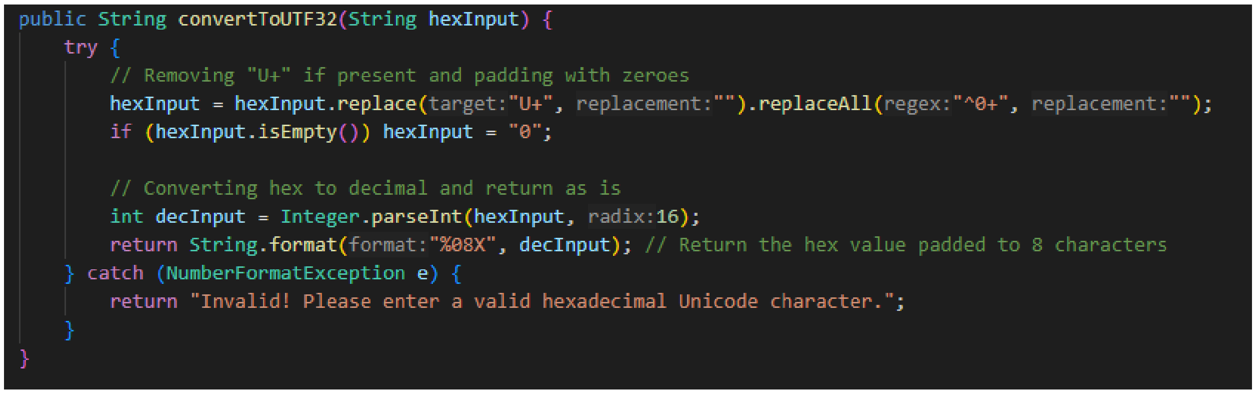
This function takes in a Unicode hexadecimal input and outputs the UTF-8 representation of it. It starts by eliminating any "U+" prefix from the input and, if needed, padding the hexadecimal text with zeros. After parsing the input hexadecimal into its decimal counterpart, the function converts it to UTF-8. The function chooses the right byte sequence for UTF-8 encoding based on the Unicode range, guaranteeing interoperability with various character sets. When an incorrect input or characters outside of the allowed range are entered, the function gently handles exceptions and notifies the user of the error.



```java
public String convertToUTF16(String hexInput) {
    try {
        // Removing "U+" if present and padding with zeroes
        hexInput = hexInput.replace(target:"U+", replacement:"").replaceAll(regex:"^0+", replacement:"");
        if (hexInput.isEmpty()) hexInput = "0";

        // Converting hex to decimal
        int decInput = Integer.parseInt(hexInput, radix:16);

        // Convert to UTF-16
        if (decInput >= 0 && decInput <= 65535) {
            return String.format(format:"%04X", decInput); // Single code unit, return the hex value
        } else if (decInput <= 1114111) {
            decInput -= 0x10000; // Subtract the offset
            int highSurrogate = 0xD800 | (decInput >> 10); // Calculate the high surrogate
            int lowSurrogate = 0xDC00 | (decInput & 0x3FF); // Calculate the low surrogate
            return String.format(format:"%04X %04X", highSurrogate, lowSurrogate); // Return both surrogates separated by space
        } else {
            return "Invalid Unicode character"; // Out of UTF-16 range
        }
    } catch (NumberFormatException e) {
        return "Invalid! Please enter a valid hexadecimal Unicode character.";
    }
}
```

Figure 2: UTF16 Conversion

This function transforms an input of Unicode hexadecimal to UTF-16 representation. It works in a manner similar to that of the convertToUTF8 function, eliminating any "U+" prefix beforehand and then padding the hex string. The function uses the Unicode range to determine the UTF-16 encoding after processing the input into decimal form. A single code unit is used for characters that are part of the Basic Multilingual Plane (BMP), and surrogate pairs are used to represent characters that are not part of the BMP. When necessary, the function generates pertinent error messages and makes sure that erroneous input is handled appropriately.



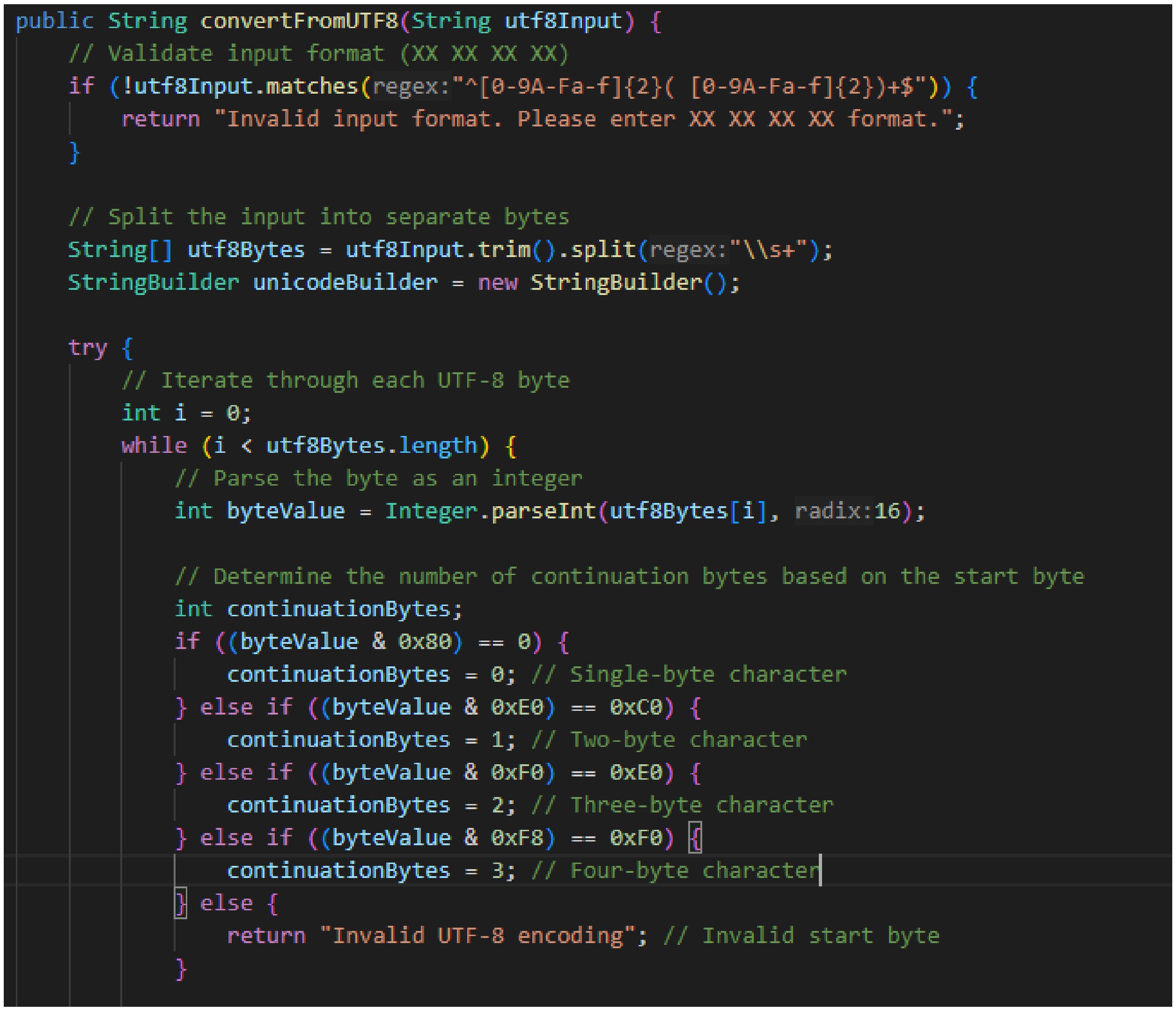
```java
public String convertToUTF32(String hexInput) {
    try {
        // Removing "U+" if present and padding with zeroes
        hexInput = hexInput.replace(target:"U+", replacement:"").replaceAll(regex:"^0+", replacement:"");
        if (hexInput.isEmpty()) hexInput = "0";

        // Converting hex to decimal and return as is
        int decInput = Integer.parseInt(hexInput, radix:16);
        return String.format(format:"%08X", decInput); // Return the hex value padded to 8 characters
    } catch (NumberFormatException e) {
        return "Invalid! Please enter a valid hexadecimal Unicode character.";
    }
}
```

Figure 3: UTF32 Conversion

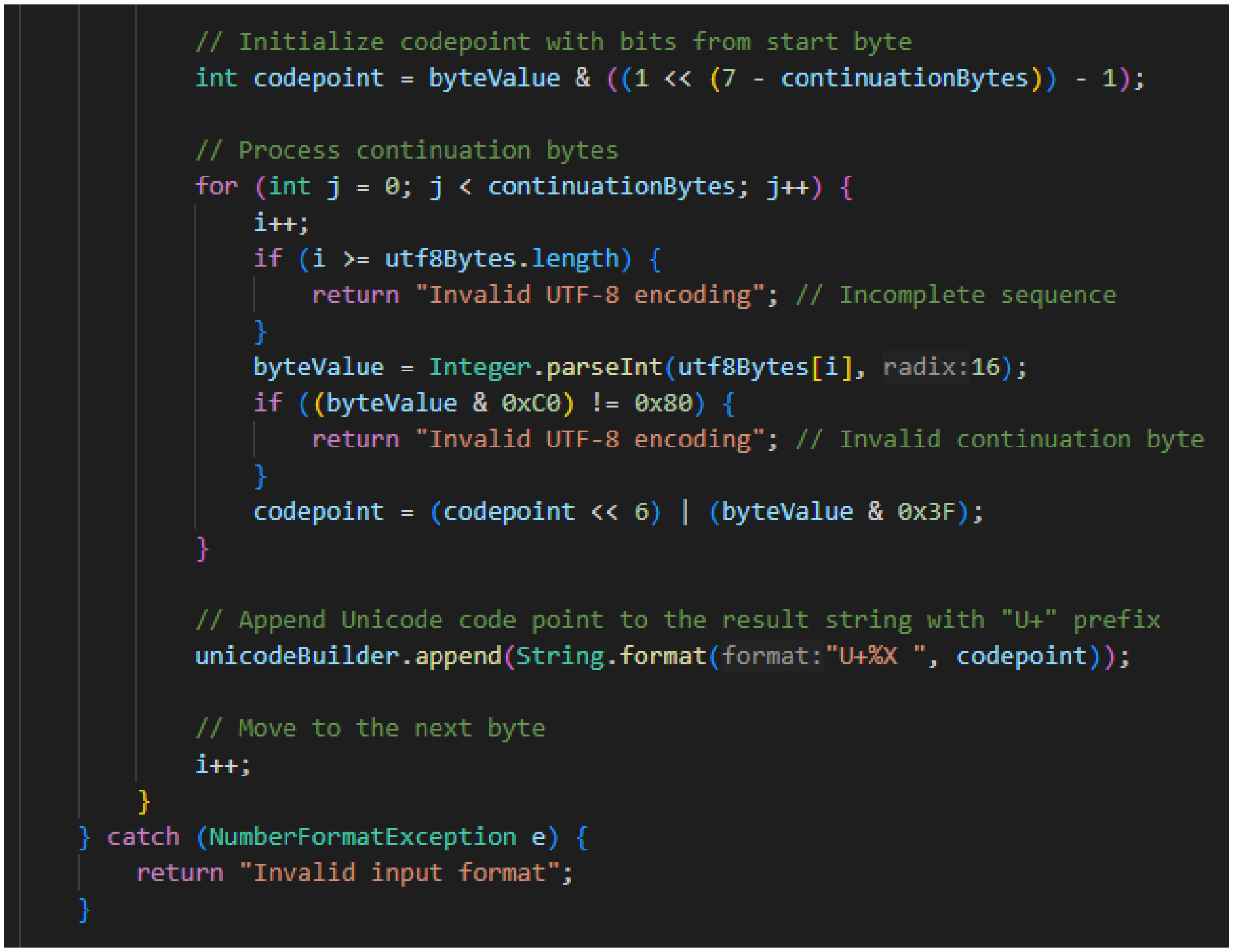
The hexadecimal Unicode input must be converted to its UTF-32 representation by this function. It removes the "U+" prefix and pads the hexadecimal. After the input is processed into decimal form, the decimal value is formatted as a hexadecimal string and padded to eight characters in order to create the UTF-32 encoding. Similar to the other features, input validation and error handling are in place to protect the conversion process' integrity and give the user unambiguous feedback when something goes wrong.

Translator

Text encoded in different UTF (Unicode Transformation Format) formats can be converted back to Unicode code points using functions in the translator class. Every function decodes the input, verifies its format, and produces the matching Unicode code points in a methodical manner.



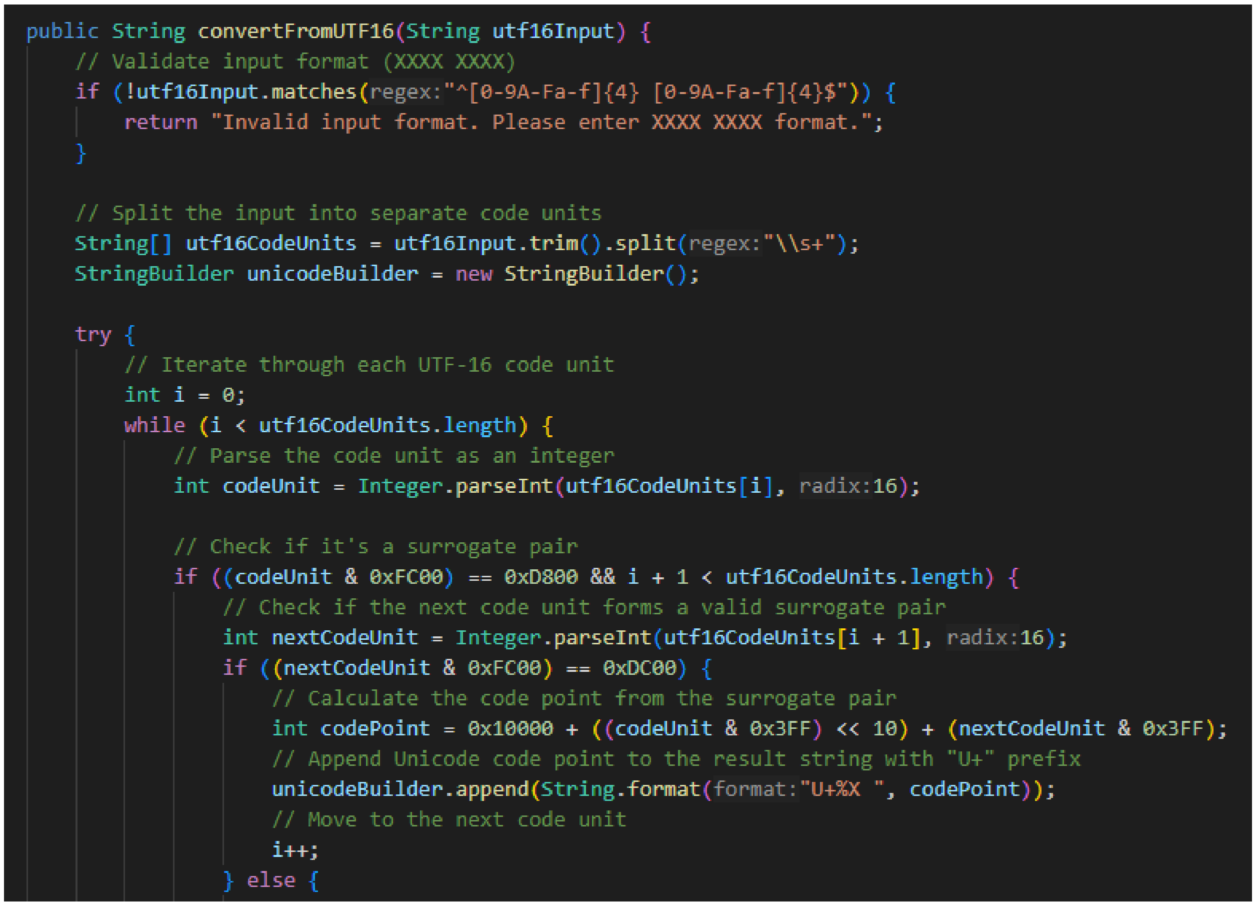
```java
public String convertFromUTF8(String utf8Input) {
    // Validate input format (XX XX XX XX)
    if (!utf8Input.matches(regex:"^[0-9A-Fa-f]{2}( [0-9A-Fa-f]{2})+$")) {
        return "Invalid input format. Please enter XX XX XX XX format.";
    }

    // Split the input into separate bytes
    String[] utf8Bytes = utf8Input.trim().split(regex:"\\s+");
    StringBuilder unicodeBuilder = new StringBuilder();

    try {
        // Iterate through each UTF-8 byte
        int i = 0;
        while (i < utf8Bytes.length) {
            // Parse the byte as an integer
            int byteValue = Integer.parseInt(utf8Bytes[i], radix:16);

            // Determine the number of continuation bytes based on the start byte
            int continuationBytes;
            if ((byteValue & 0x80) == 0) {
                continuationBytes = 0; // Single-byte character
            } else if ((byteValue & 0xE0) == 0xC0) {
                continuationBytes = 1; // Two-byte character
            } else if ((byteValue & 0xF0) == 0xE0) {
                continuationBytes = 2; // Three-byte character
            } else if ((byteValue & 0xF8) == 0xF0) {
                continuationBytes = 3; // Four-byte character
            } else {
                return "Invalid UTF-8 encoding"; // Invalid start byte
            }
```



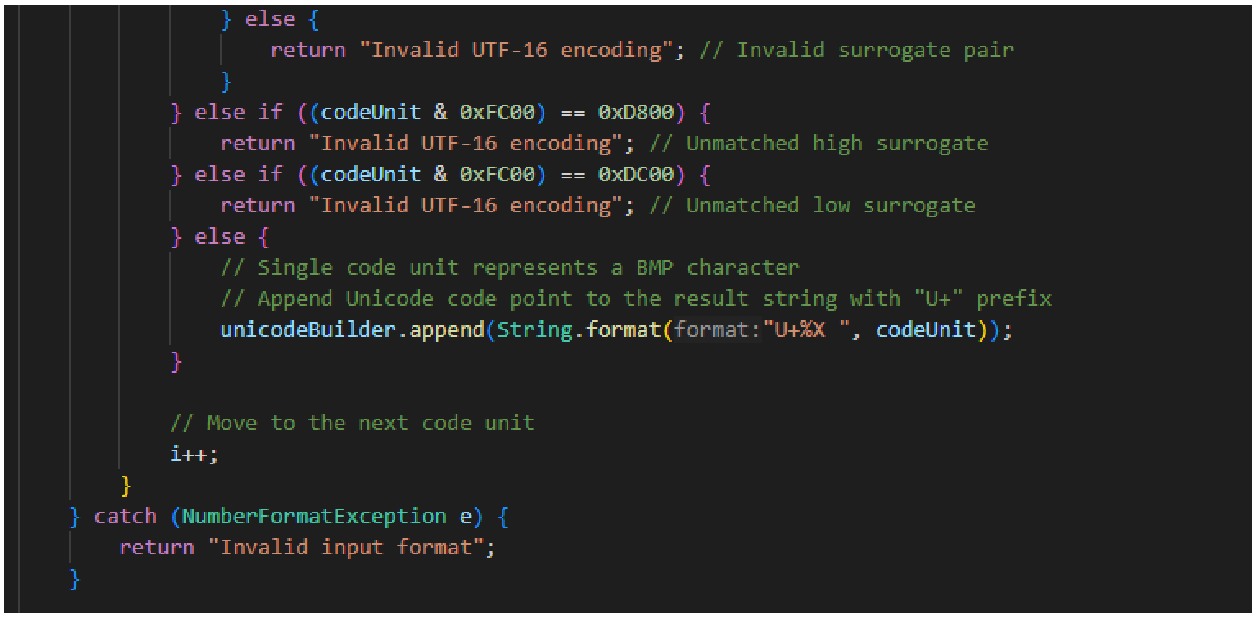
```
                // Initialize codepoint with bits from start byte
                int codepoint = byteValue & ((1 << (7 - continuationBytes)) - 1);

                // Process continuation bytes
                for (int j = 0; j < continuationBytes; j++) {
                    i++;
                    if (i >= utf8Bytes.length) {
                        return "Invalid UTF-8 encoding"; // Incomplete sequence
                    }
                    byteValue = Integer.parseInt(utf8Bytes[i], radix:16);
                    if ((byteValue & 0xC0) != 0x80) {
                        return "Invalid UTF-8 encoding"; // Invalid continuation byte
                    }
                    codepoint = (codepoint << 6) | (byteValue & 0x3F);
                }

                // Append Unicode code point to the result string with "U+" prefix
                unicodeBuilder.append(String.format(format:"U+%X ", codepoint));

                // Move to the next byte
                i++;
            }
        } catch (NumberFormatException e) {
            return "Invalid input format";
        }
```

Figure 4: Conversion from UTF8

Text encoded in UTF-8 is decoded by this function and transformed into Unicode code points. The input format is first validated to make sure it follows the expected pattern (XX XX XX XX). The input is subsequently divided into individual bytes, and the function iterates through each UTF-8 byte, decoding it in accordance with the UTF-8 encoding requirements. In order to reconstitute the Unicode code points and ensure correct handling of surrogate pairs and erroneous encodings, continuation bytes are handled. Lastly, the function returns the result after building the Unicode code points using the "U+" prefix.



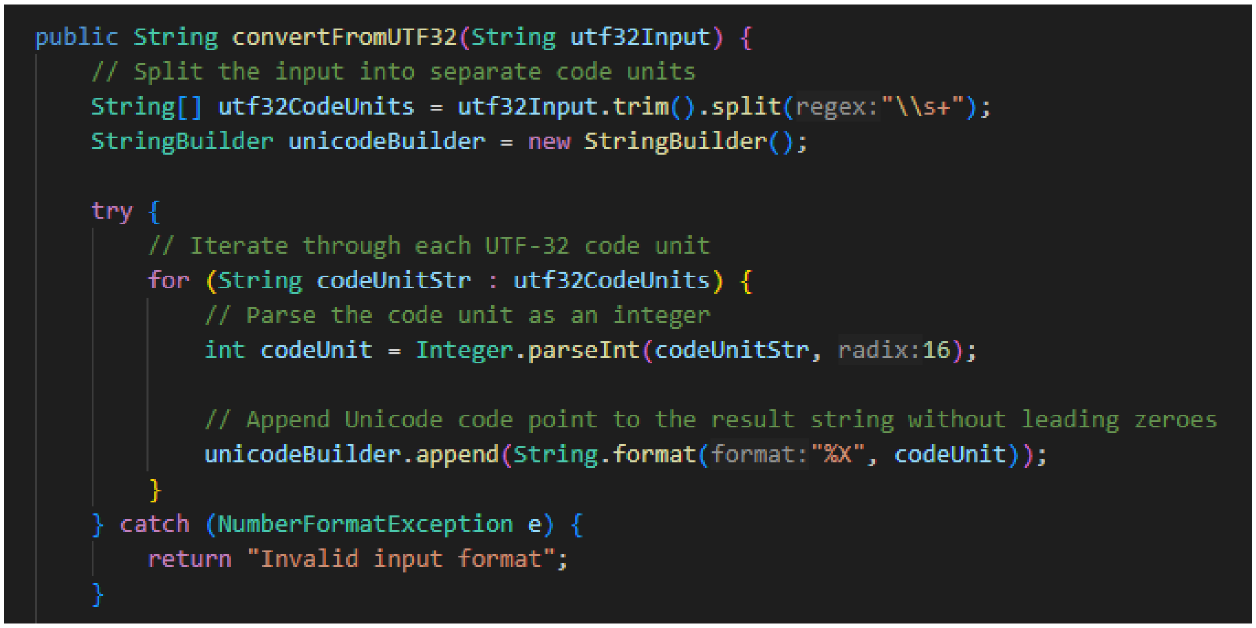
```java
public String convertFromUTF16(String utf16Input) {
    // Validate input format (XXXX XXXX)
    if (!utf16Input.matches(regex:"^[0-9A-Fa-f]{4} [0-9A-Fa-f]{4}$")) {
        return "Invalid input format. Please enter XXXX XXXX format.";
    }

    // Split the input into separate code units
    String[] utf16CodeUnits = utf16Input.trim().split(regex:"\\s+");
    StringBuilder unicodeBuilder = new StringBuilder();

    try {
        // Iterate through each UTF-16 code unit
        int i = 0;
        while (i < utf16CodeUnits.length) {
            // Parse the code unit as an integer
            int codeUnit = Integer.parseInt(utf16CodeUnits[i], radix:16);

            // Check if it's a surrogate pair
            if ((codeUnit & 0xFC00) == 0xD800 && i + 1 < utf16CodeUnits.length) {
                // Check if the next code unit forms a valid surrogate pair
                int nextCodeUnit = Integer.parseInt(utf16CodeUnits[i + 1], radix:16);
                if ((nextCodeUnit & 0xFC00) == 0xDC00) {
                    // Calculate the code point from the surrogate pair
                    int codePoint = 0x10000 + ((codeUnit & 0x3FF) << 10) + (nextCodeUnit & 0x3FF);
                    // Append Unicode code point to the result string with "U+" prefix
                    unicodeBuilder.append(String.format(format:"U+%X ", codePoint));
                    // Move to the next code unit
                    i++;
                } else {
```



```java
                } else {
                    return "Invalid UTF-16 encoding"; // Invalid surrogate pair
                }
            } else if ((codeUnit & 0xFC00) == 0xD800) {
                return "Invalid UTF-16 encoding"; // Unmatched high surrogate
            } else if ((codeUnit & 0xFC00) == 0xDC00) {
                return "Invalid UTF-16 encoding"; // Unmatched low surrogate
            } else {
                // Single code unit represents a BMP character
                // Append Unicode code point to the result string with "U+" prefix
                unicodeBuilder.append(String.format(format:"U+%X ", codeUnit));
            }

            // Move to the next code unit
            i++;
        }
    } catch (NumberFormatException e) {
        return "Invalid input format";
    }
```

Figure 5: Convert from UTF16

The conversion of text encoded in UTF-16 to Unicode code points is handled by this method. It divides the input into distinct UTF-16 code units and verifies that the input format matches the anticipated pattern (XXXX XXXX). The function makes a distinction between surrogate pairs and single code units as iteratively goes over each code unit. Single code units directly represent BMP (Basic Multilingual Plane) characters, while surrogate pairs are merged to generate the matching code point. Using the "U+" prefix, the function creates Unicode code points and then returns the outcome.



```java
public String convertFromUTF32(String utf32Input) {
    // Split the input into separate code units
    String[] utf32CodeUnits = utf32Input.trim().split(regex:"\\s+");
    StringBuilder unicodeBuilder = new StringBuilder();

    try {
        // Iterate through each UTF-32 code unit
        for (String codeUnitStr : utf32CodeUnits) {
            // Parse the code unit as an integer
            int codeUnit = Integer.parseInt(codeUnitStr, radix:16);

            // Append Unicode code point to the result string without leading zeroes
            unicodeBuilder.append(String.format(format:"%X", codeUnit));
        }
    } catch (NumberFormatException e) {
        return "Invalid input format";
    }
```

Figure 6: Convert from UTF32

This method converts text encoded in UTF-32 to Unicode code points by decoding the input. The input is divided into distinct UTF-32 code units, and then iteratively parses each code unit as an integer. The function adds the "U+" prefix to the returned string after creating the Unicode code points from the code units. Lastly, the output is the concatenated Unicode code points.

IV. Conclusion

Our research describes a new Unicode converter designed to bridge the gap between different encoding systems, with a particular emphasis on UTF and Unicode representation conversion. Our converter is user-friendly, with a Java-based graphical interface that makes it simple and intuitive to use. We verified its dependability and compatibility by painstakingly following specifications, conducting rigorous validation checks, and supporting multiple UTF formats.

The methodology section describes our extensive approach to designing the converter and translator capabilities. Each component of the converter is meticulously designed to correctly handle Unicode characters and convert them into the appropriate UTF representations. Similarly, the translator function effectively decodes text encoded in a variety of UTF formats, assuring precise and consistent conversion to Unicode code points.

Overall, our Unicode converter provides an adaptable tool for text processing and manipulation, making it easier to work with a variety of character sets. Our goal is to give users with a tool that simplifies text encoding and decoding operations by combining robust functionality and a user-friendly design. As technology evolves and worldwide communication expands, our converter seeks to enable smooth text processing across multiple platforms.