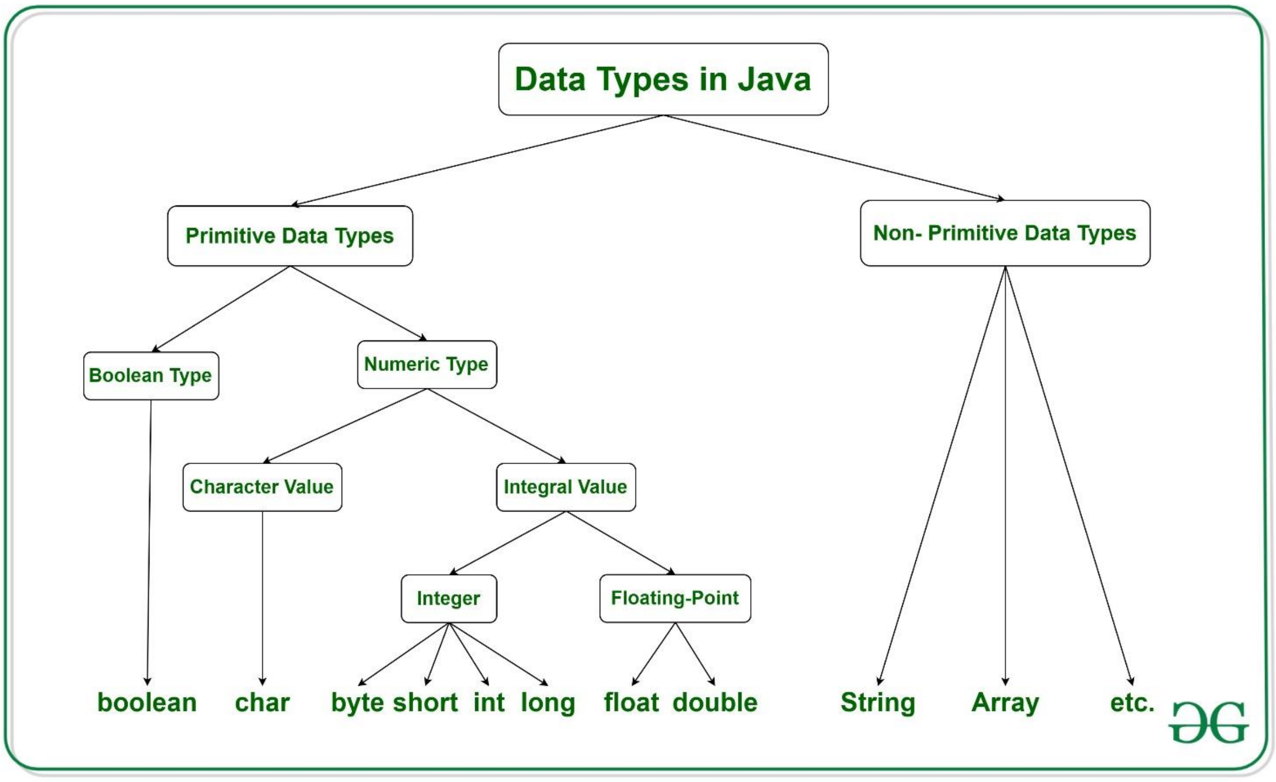
# 1. Data Types



Java Primitive Types

| Type | Size | Range | Default |
|---|---|---|---|
| boolean | 1 bit | true or false | false |
| byte | 8 bits | [-128, 127] | 0 |
| short | 16 bits | [-32,768, 32,767] | 0 |
| char | 16 bits | ['\u0000', '\uffff'] or [0, 65535] | '\u0000' |
| int | 32 bits | [-2,147,483,648 to 2,147,483,647] | 0 |
| long | 64 bits | [$-2^{63}$, $2^{63}-1$] | 0 |
| float | 32 bits | 32-bit IEEE 754 floating-point | 0.0 |
| double | 64 bits | 64-bit IEEE 754 floating-point | 0.0 |

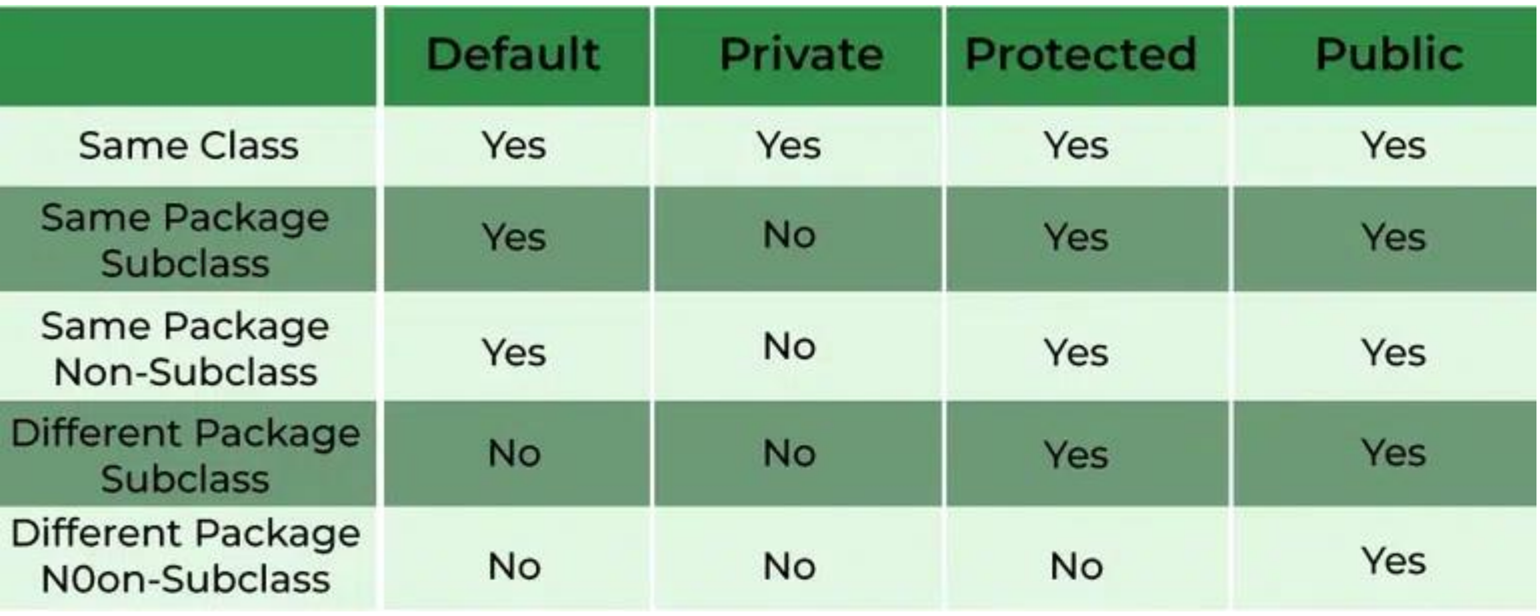
## 2. Access Control

- Access Control in Java refers to the mechanism used to restrict or allow access to certain parts of a Java program, such as classes, methods, and variables.
- Access control determines which classes and objects can access specific codes or data within a program.
- By controlling access to different parts of the program, Java's access control mechanism promotes code encapsulation, and information hiding, and reduces the likelihood of errors and security vulnerabilities in the program.
- Access control in Java can be implemented by using access control modifiers, which are keywords placed before the declaration of the class member.

### Access Control Modifiers in Java

Access control modifiers in Java are keywords that can be used to control access to classes, fields, and methods. Access control modifiers determine the level of access that other classes or objects have to a particular class, field, or method.

The four access control levels in Java, from most restrictive to least restrictive, are:



| | Default | Private | Protected | Public |
|---|---|---|---|---|
| Same Class | Yes | Yes | Yes | Yes |
| Same Package Subclass | Yes | No | Yes | Yes |
| Same Package Non-Subclass | Yes | No | Yes | Yes |
| Different Package Subclass | No | No | Yes | Yes |
| Different Package N0on-Subclass | No | No | No | Yes |

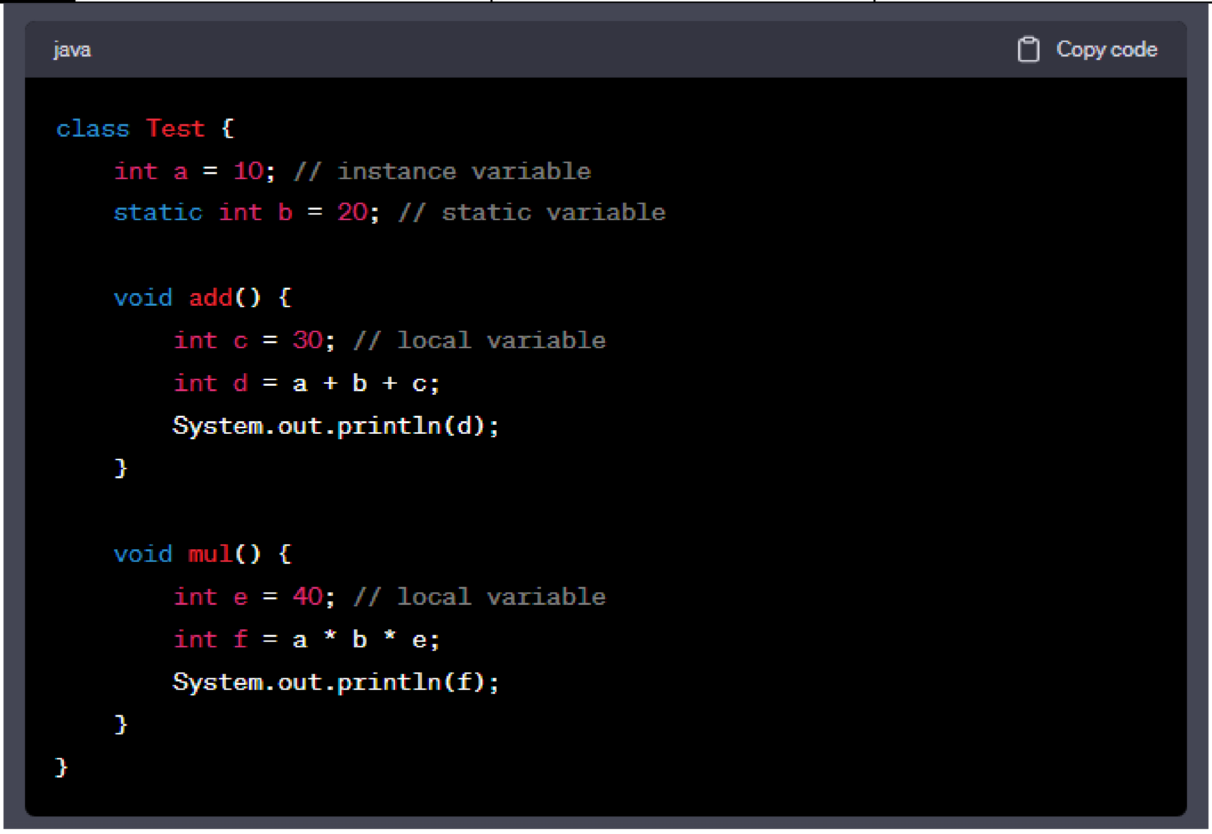
## *3. Scope and Lifetime of Variables*

**Variables:** A variable is the name of a memory location that can contain data. Variables have data types associated with them.

**Literals:** Literals are constant values assigned to variables directly in the code.

**Types Of Variables:** In Java, variables are of three types:

| Aspect | Local Variables | Instance Variables | Static Variables |
|---|---|---|---|
| Declaration | Inside the body of methods, constructors, or blocks | Inside the class but outside methods, constructors, blocks | Declared with the static keyword outside methods, constructors, blocks |
| Scope | Limited to the block in which they are declared | Accessible within all methods, blocks, and constructors within the class | Accessible within all methods, constructors, blocks, including static methods, constructors, blocks |
| Memory Allocation | Allocated when the method, constructor, or block is executed, and destroyed when it exits | Allocated when an object is created and released when the object is destroyed | Allocated when the class is loaded into memory and destroyed when the class is unloaded |
| Stored Memory | Stack memory | Heap memory | Non-heap memory or static memory |
| Default Values | No default values; value must be provided before use | int: 0; boolean: false; object: null, etc. | int: 0; boolean: false; object: null, etc. |
| Access Specifier | Cannot use access specifiers like public, protected, or private | Can use access specifiers like public, protected, or private | Can use access specifiers like public, protected, or private |
| Access | - | Can be accessed directly or using the object's name | Can be accessed directly, using the class name, or using the object reference name |

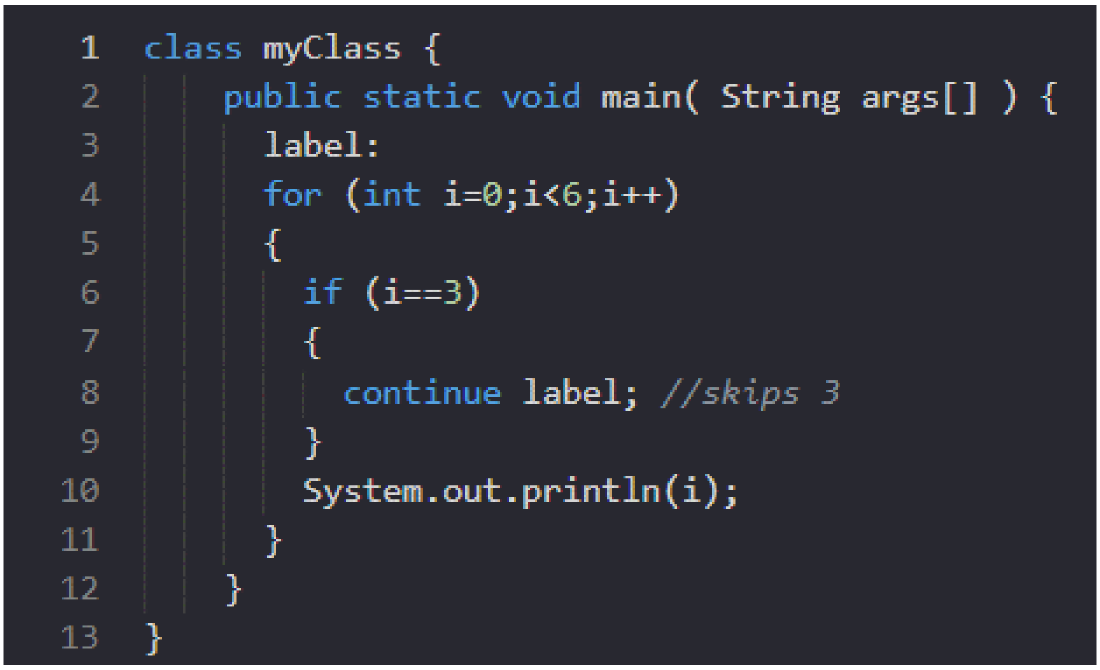


```java
class Test {
    int a = 10; // instance variable
    static int b = 20; // static variable

    void add() {
        int c = 30; // local variable
        int d = a + b + c;
        System.out.println(d);
    }

    void mul() {
        int e = 40; // local variable
        int f = a * b * e;
        System.out.println(f);
    }
}
```

## 4. Control flow statements

In Java, **break**, **continue**, **return**, and labels are control flow statements used to alter the flow of execution within loops and conditional blocks.

1. **Break:**
   - The **break** statement is used to prematurely exit a loop.
   - When encountered, the loop terminates, and the program continues with the statement following the loop.
2. **Continue:**
   - The **continue** statement is used to skip the current iteration of a loop.
   - When encountered, the remaining statements within the loop are skipped, and the loop proceeds to the next iteration.
3. **Return:**
   - The **return** statement is used to exit a method and optionally return a value to the caller.
   - When encountered, the method ends, and control is passed back to the caller.
4. **Labels:**
   - Labels are used to mark specific locations in the code.
   - They are primarily used with **break** and **continue** statements to specify which loop should be exited or continued.



```
class myClass {
    public static void main( String args[] ) {
      label:
      for (int i=0;i<6;i++)
      {
        if (i==3)
        {
          continue label; //skips 3
        }
        System.out.println(i);
      }
    }
}
```

## 5. Type Conversion in Java:

1. Type conversion is the process of converting a value from one data type to another in Java.
2. Java performs automatic or implicit type conversion if the data types are compatible.
3. Automatic type conversion is also known as automatic type promotion.
4. Automatic type conversion is done to convert a lower data type into a higher data type.
5. Java automatically converts smaller data types to larger data types to avoid loss of data or precision.

## Types of Casting in Java:

### 1. Implicit Type Casting (Automatic Type Conversion/Widening conversion ):

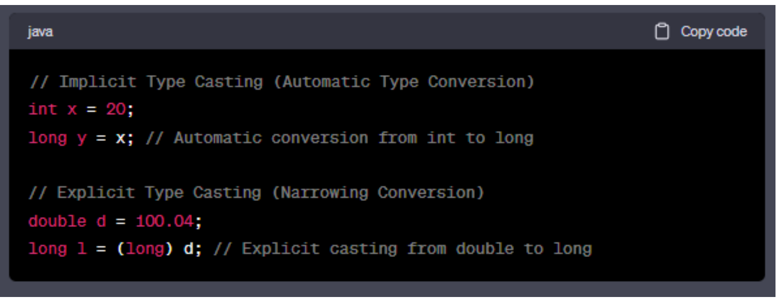
a) Java compiler performs this internally when two data types are compatible and the destination type is larger than the source type.
b) Automatic type conversion is done for primitive data types like int to long, float to double, etc.
c) It is safe as there is no loss of data or precision.

### 2. Explicit Type Casting (Narrowing Conversion):

a) Programmer performs this type of conversion using the cast operator **(type_name)** when two data types are not compatible, and the destination type is smaller than the source type.
b) Explicit type casting is used to convert a higher data type into a lower data type.
c) It is not safe as it may result in data loss or loss of precision.
d) The programmer must handle explicit type casting.

## Example:

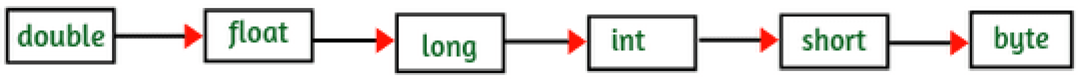
javaCopy code



```java
// Implicit Type Casting (Automatic Type Conversion)
int x = 20;
long y = x; // Automatic conversion from int to long

// Explicit Type Casting (Narrowing Conversion)
double d = 100.04;
long l = (long) d; // Explicit casting from double to long
```

## Note:

- Widening conversion (automatic) takes place when going from smaller data types to larger ones (e.g., byte to int).
- Narrowing conversion (explicit) is required when going from larger data types to smaller ones (e.g., double to int).
- Explicit type casting may lead to data loss, so it should be used with caution.



Narrowing or Explicit type conversion

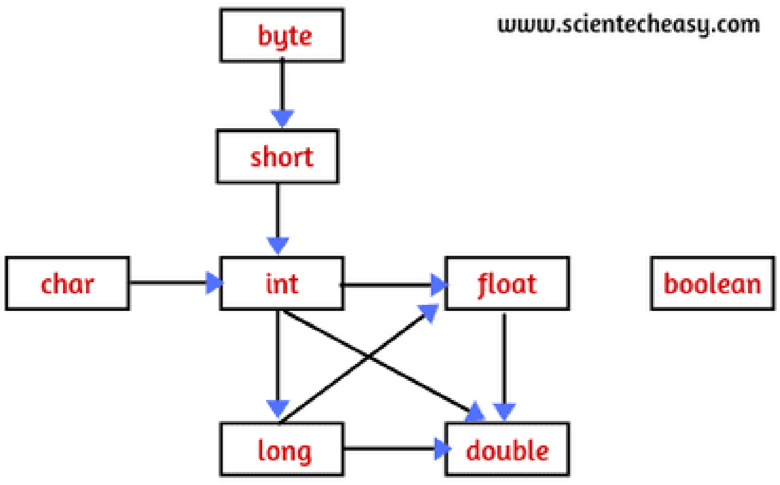


Fig: Automatic type conversion that Java allows.

# STRINGS

In Java, both StringBuffer and StringBuilder are classes that represent mutable sequences of characters. They are used to efficiently manipulate strings by allowing you to modify them without creating new objects every time, unlike the regular String class, which is immutable.

## StringBuffer Class:

1) StringBuffer was introduced in Java 1.0, and it is part of the original Java API.
2) It provides thread-safe operations, which means it is designed to be used in multi-threaded environments, and its methods are synchronized to ensure thread safety.
3) However, this synchronization comes with a performance overhead, making it less efficient in single-threaded scenarios compared to StringBuilder.
4) Since StringBuffer is synchronized, it is suitable for situations where multiple threads need to access and modify the same string concurrently without causing data corruption or inconsistencies. However, this comes at the cost of some performance trade-offs. If thread safety is not a concern, it is generally recommended to use StringBuilder instead.

## StringBuilder Class:

5) StringBuilder was introduced in Java 1.5 as part of the Java API, and it is essentially the same as StringBuffer in terms of functionality.
6) The key difference is that StringBuilder is not synchronized, which makes it more efficient in single-threaded scenarios compared to StringBuffer.
7) Because StringBuilder is not thread-safe, it is not recommended to use it in multi-threaded environments where multiple threads may access and modify the same string concurrently. However, in single-threaded scenarios, using StringBuilder is preferred when performance is a priority because it avoids the synchronization overhead present in StringBuffer.

## FEATURES

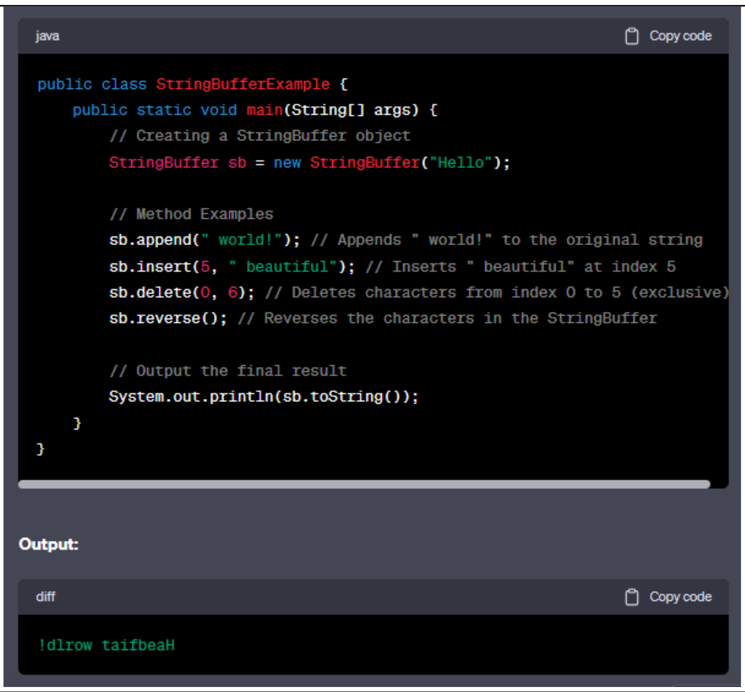
1. **Thread Safety:** StringBuffer is designed to be thread-safe, meaning its methods are synchronized to allow safe access and modification of strings in multi-threaded environments.
2. **Safe for Concurrent Use:** It can be used in scenarios where multiple threads need to access and modify the same string concurrently without causing data corruption or inconsistencies.
3. **Synchronized Methods:** The synchronization comes with a performance overhead, which can make StringBuffer less efficient in single-threaded scenarios compared to StringBuilder.
4. **Ideal for Multi-Threaded Environments:** StringBuffer is preferred when thread safety is a top priority, and you need to ensure that multiple threads can safely manipulate strings.

1. **No Thread Safety:** StringBuilder is not thread-safe, which means it is more efficient in single-threaded scenarios compared to StringBuffer.
2. **Faster Performance:** Because it is not synchronized, StringBuilder avoids the synchronization overhead present in StringBuffer, making it faster in single-threaded contexts.
3. **Not Suitable for Concurrent Use:** It should not be used in multi-threaded environments where multiple threads may access and modify the same string concurrently, as it does not provide inherent thread safety.
4. **Ideal for Single-Threaded Environments:** StringBuilder is the preferred choice when you are working in a single-threaded context or can ensure synchronization through other means.

## 4 METHODS EXAMPLE Implementation

1. **append()** to add " world!" to the end of the string.
2. **insert()** to insert " beautiful" at index 5.
3. **delete()** to remove characters from index 0 to 5 (excluding index 5).
4. **reverse()** to reverse the characters in the StringBuffer.

1. **append()** to add " is" to the end of the string.
2. **replace()** to replace characters from index 0 to 3 with "Python".
3. **deleteCharAt()** to delete the character at index 5 (which was 'i').
4. **insert()** to insert " awesome" at index 5.



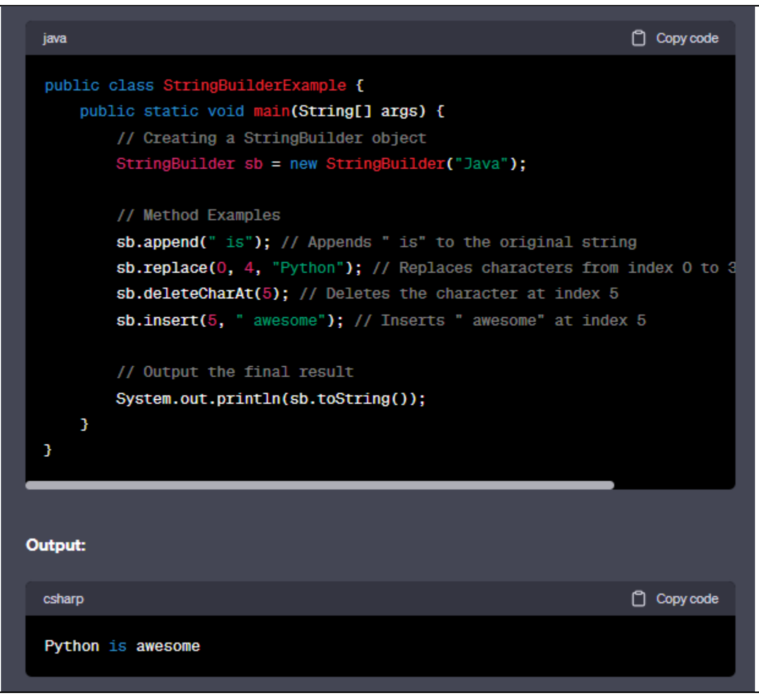
java

```java
public class StringBufferExample {
    public static void main(String[] args) {
        // Creating a StringBuffer object
        StringBuffer sb = new StringBuffer("Hello");

        // Method Examples
        sb.append(" world!"); // Appends " world!" to the original string
        sb.insert(5, " beautiful"); // Inserts " beautiful" at index 5
        sb.delete(0, 6); // Deletes characters from index 0 to 5 (exclusive)
        sb.reverse(); // Reverses the characters in the StringBuffer

        // Output the final result
        System.out.println(sb.toString());
    }
}
```

Output:

diff

```
!dlrow taifbeaH
```



java

```java
public class StringBuilderExample {
    public static void main(String[] args) {
        // Creating a StringBuilder object
        StringBuilder sb = new StringBuilder("Java");

        // Method Examples
        sb.append(" is"); // Appends " is" to the original string
        sb.replace(0, 4, "Python"); // Replaces characters from index 0 to 3
        sb.deleteCharAt(5); // Deletes the character at index 5
        sb.insert(5, " awesome"); // Inserts " awesome" at index 5

        // Output the final result
        System.out.println(sb.toString());
    }
}
```

Output:

csharp

```
Python is awesome
```

## 6. VTU QUESTIONS ON STRING

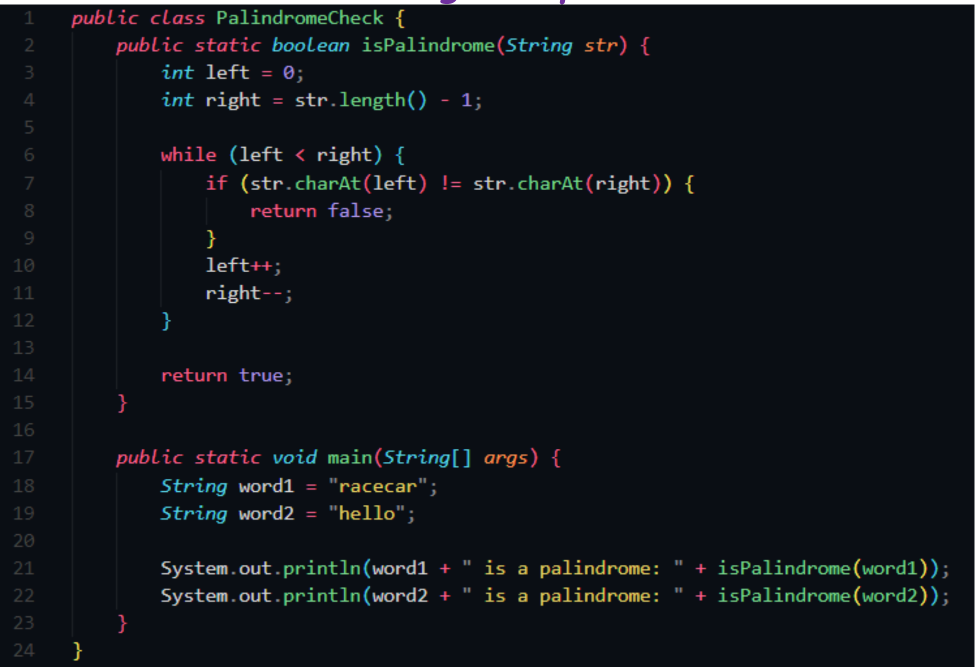
1. What is the significance of the StringBuffer class?
2. Explain the following methods with an example for each:
   i) length() ii) capacity() iii) append() iv) reverse().

3. Explain any four methods present in the StringBuffer class.
4. Give any four methods of the String class with syntax and examples.
5. What is the use of StringBuffer? Explain the StringBuffer class with a suitable example.
6. What is the significance of the StringBuffer class?
   Explain the following methods with an example for each: i) length() ii) capacity() iii) append() iv) reverse().

7. Explain any five methods present in the String class to compare strings.
8. Write a Java program to perform different operations on strings:
   i) length ii) uppercase iii) find substring iv) compare 2 strings v) check if 2 strings are equal vi) change to a character array.
9. Explain the StringBuilder class.
10. What is the String class? Explain any four string comparison methods with an example.

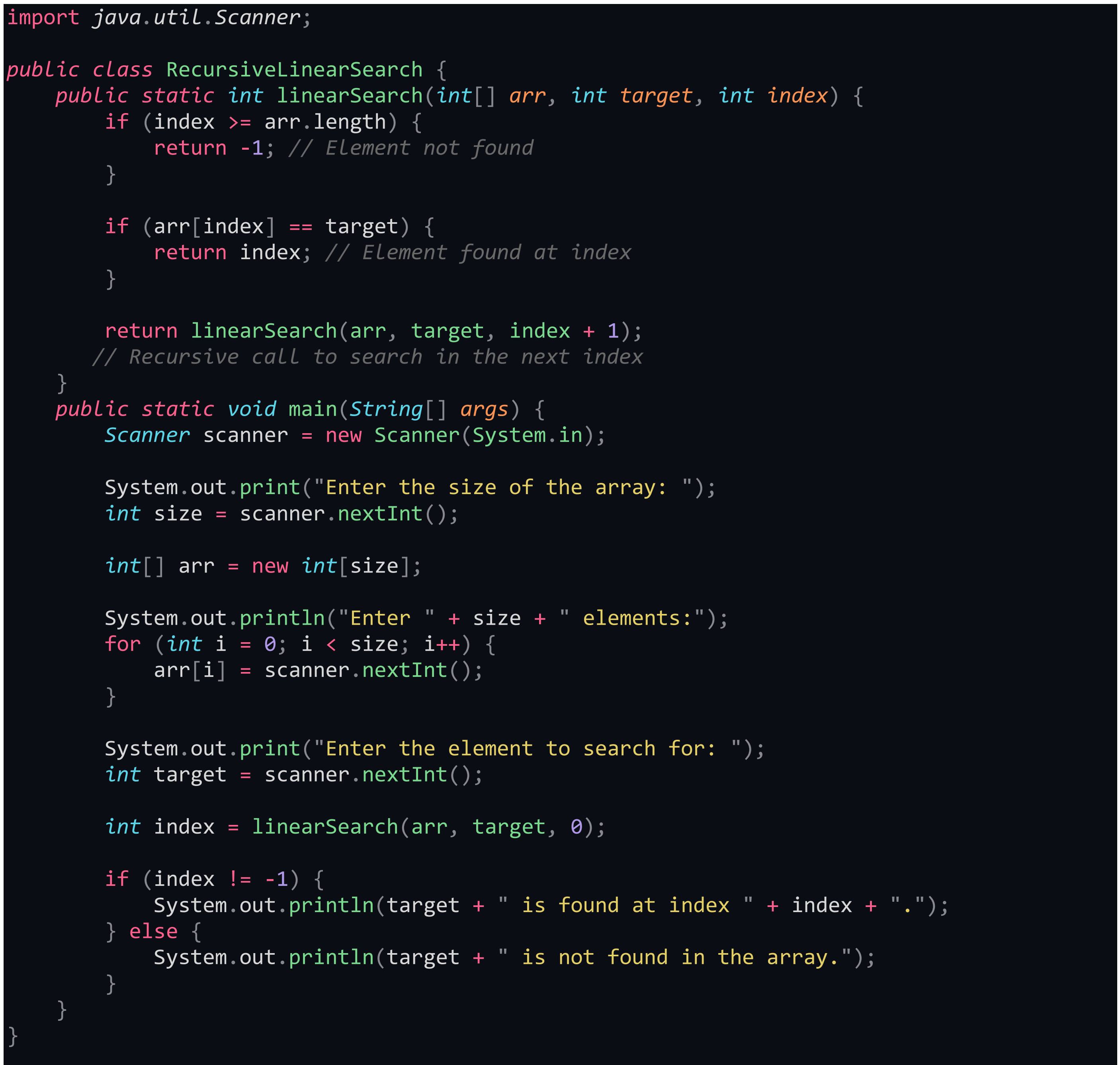
# 1. PRACTICAL CODES

## 1. To check if a string is a palindrome without reversing



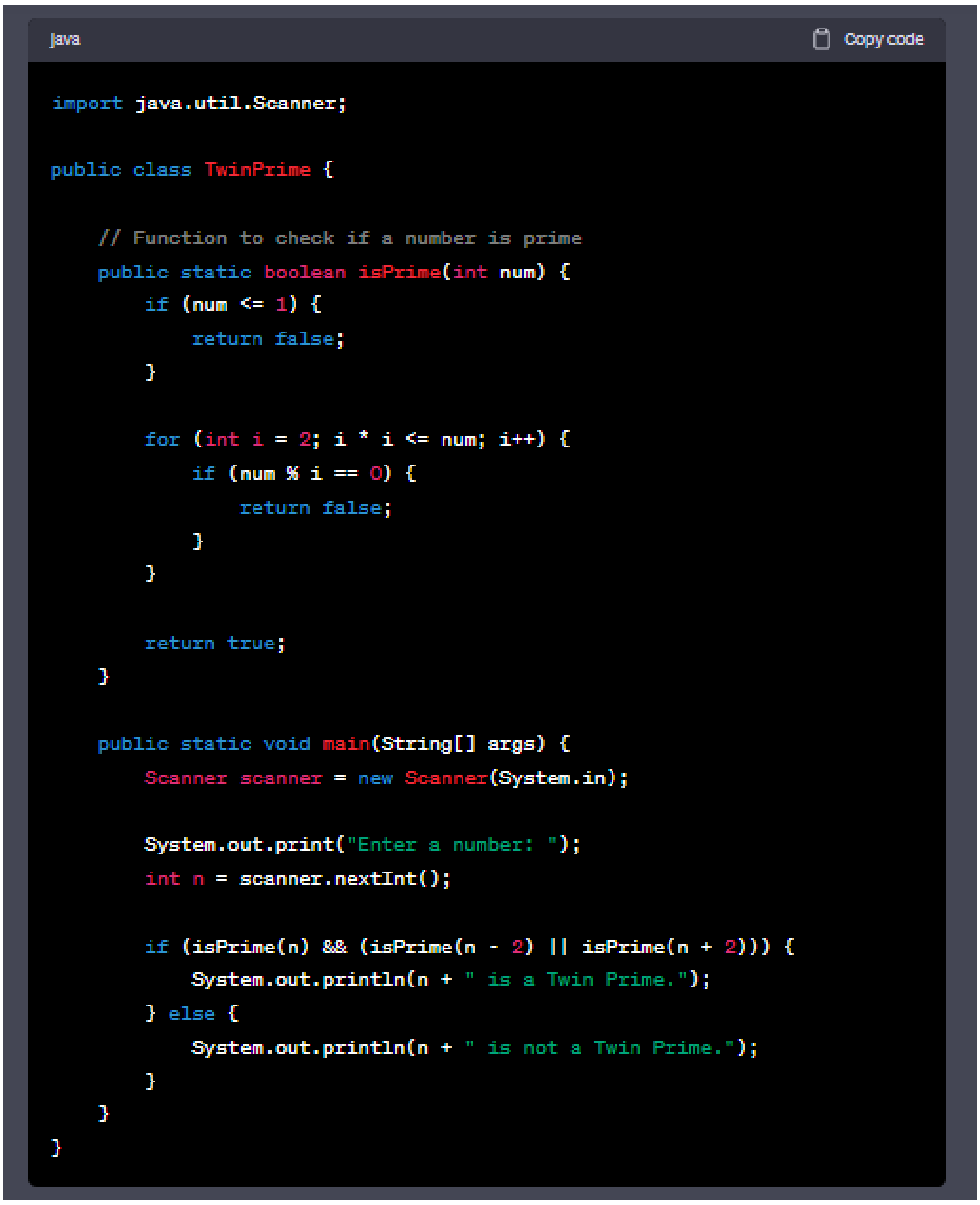
```java
public class PalindromeCheck {
    public static boolean isPalindrome(String str) {
        int left = 0;
        int right = str.length() - 1;

        while (left < right) {
            if (str.charAt(left) != str.charAt(right)) {
                return false;
            }
            left++;
            right--;
        }

        return true;
    }

    public static void main(String[] args) {
        String word1 = "racecar";
        String word2 = "hello";

        System.out.println(word1 + " is a palindrome: " + isPalindrome(word1));
        System.out.println(word2 + " is a palindrome: " + isPalindrome(word2));
    }
}
```

## 2. Recursive linear search



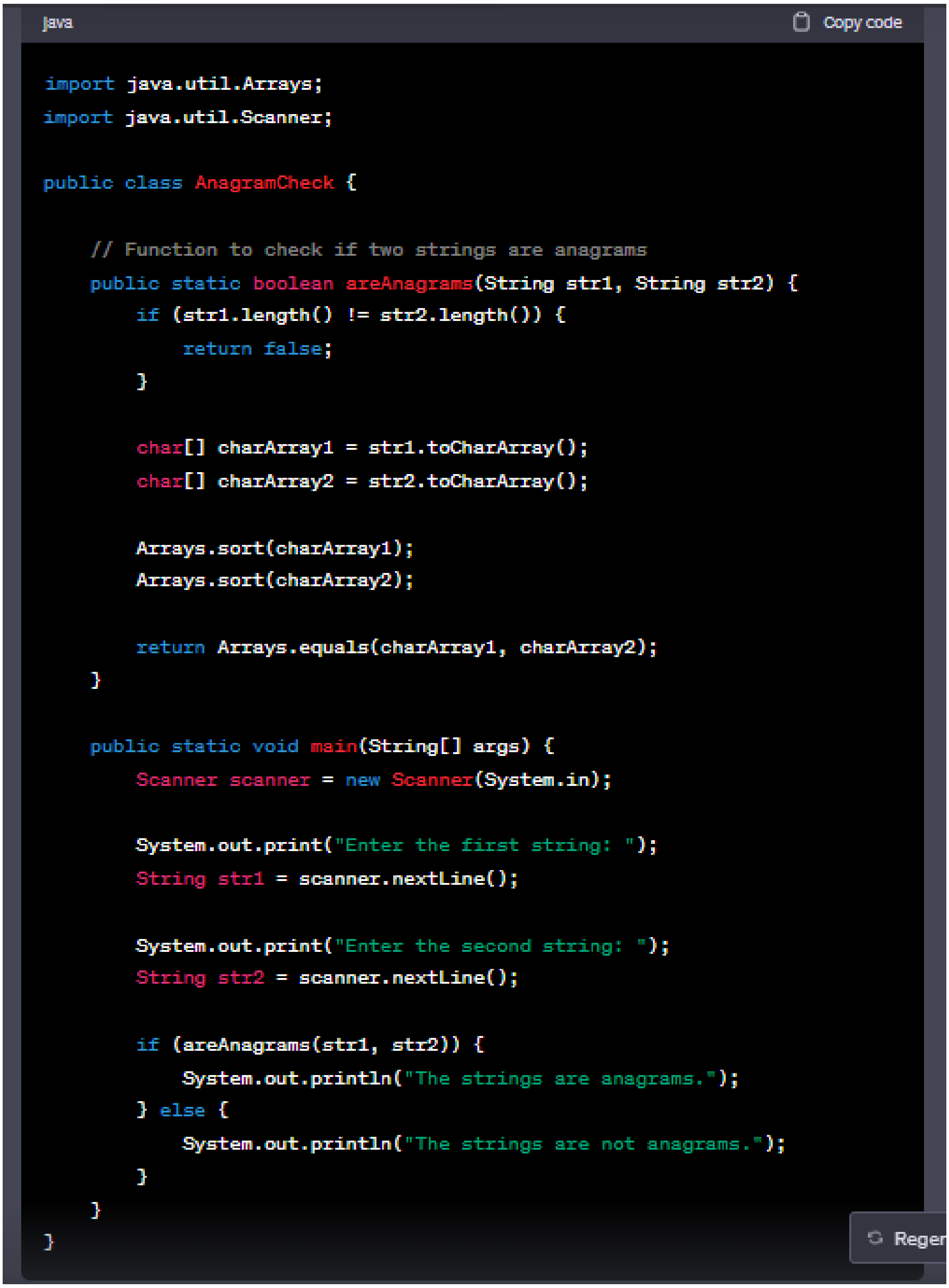
```java
import java.util.Scanner;

public class RecursiveLinearSearch {
    public static int linearSearch(int[] arr, int target, int index) {
        if (index >= arr.length) {
            return -1; // Element not found
        }

        if (arr[index] == target) {
            return index; // Element found at index
        }

        return linearSearch(arr, target, index + 1);
       // Recursive call to search in the next index
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the size of the array: ");
        int size = scanner.nextInt();

        int[] arr = new int[size];

        System.out.println("Enter " + size + " elements:");
        for (int i = 0; i < size; i++) {
            arr[i] = scanner.nextInt();
        }

        System.out.print("Enter the element to search for: ");
        int target = scanner.nextInt();

        int index = linearSearch(arr, target, 0);

        if (index != -1) {
            System.out.println(target + " is found at index " + index + ".");
        } else {
            System.out.println(target + " is not found in the array.");
        }
    }
}
```

## 3. TWIN PRIME



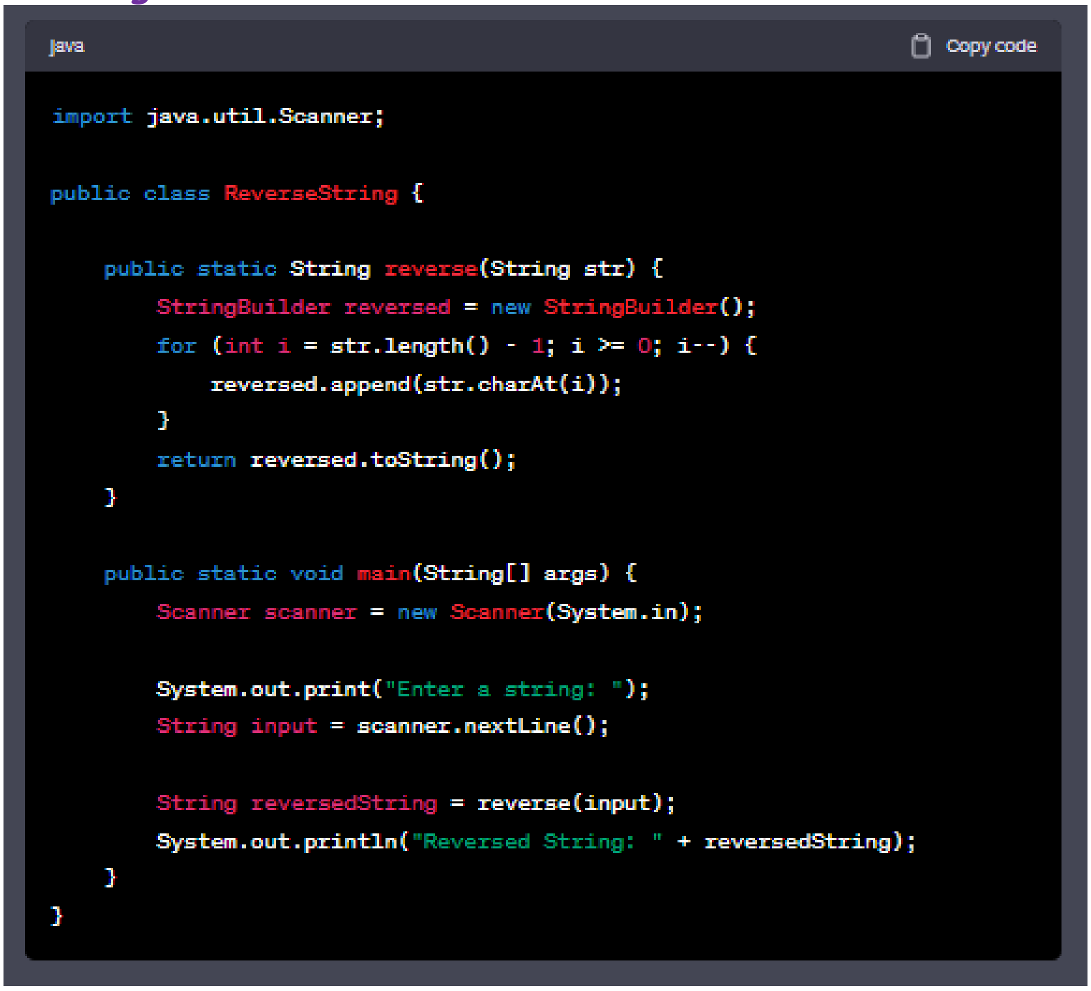
```java
import java.util.Scanner;

public class TwinPrime {

    // Function to check if a number is prime
    public static boolean isPrime(int num) {
        if (num <= 1) {
            return false;
        }

        for (int i = 2; i * i <= num; i++) {
            if (num % i == 0) {
                return false;
            }
        }

        return true;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a number: ");
        int n = scanner.nextInt();

        if (isPrime(n) && (isPrime(n - 2) || isPrime(n + 2))) {
            System.out.println(n + " is a Twin Prime.");
        } else {
            System.out.println(n + " is not a Twin Prime.");
        }
    }
}
```

## 4. ANAGRAM



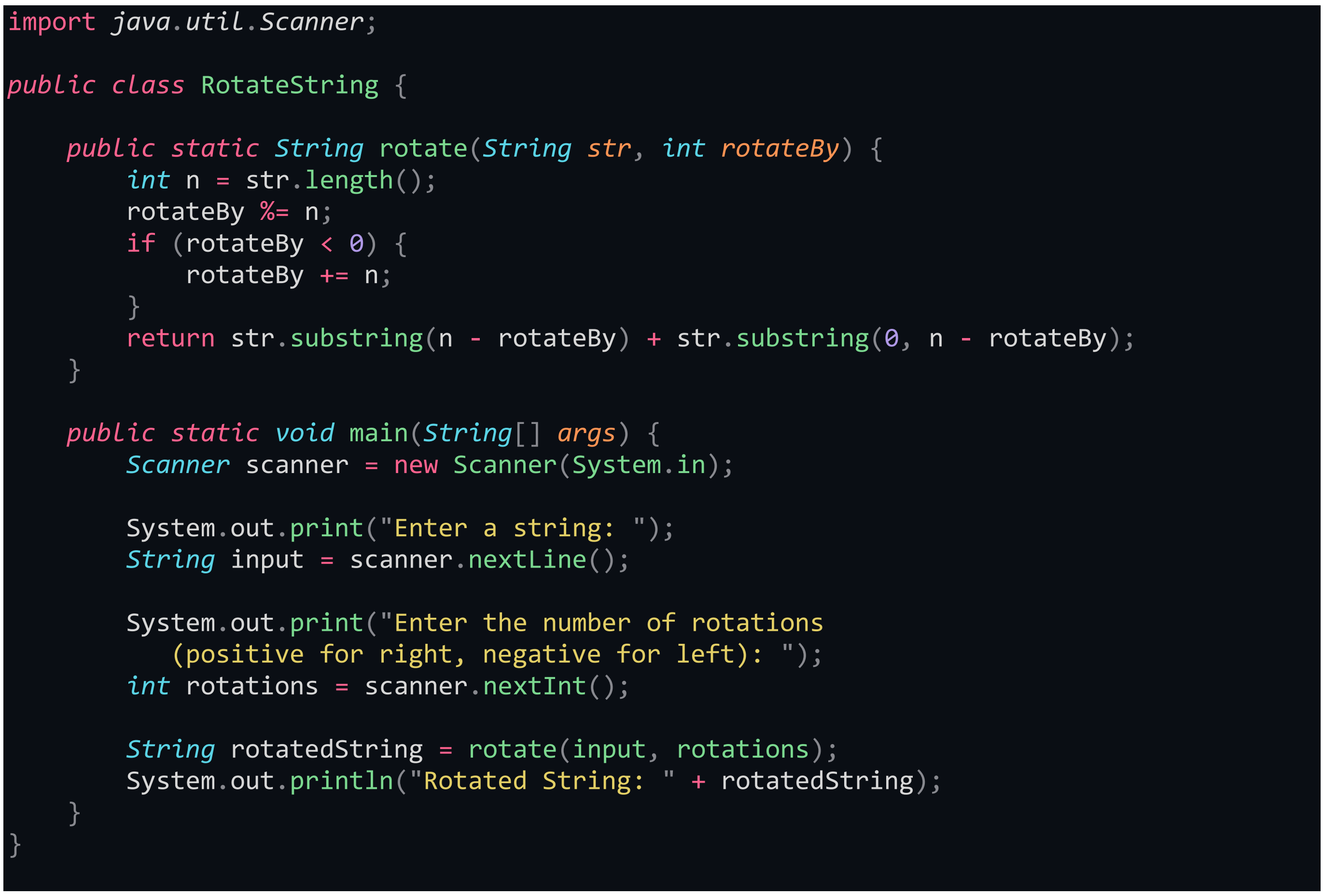
```java
import java.util.Arrays;
import java.util.Scanner;

public class AnagramCheck {

    // Function to check if two strings are anagrams
    public static boolean areAnagrams(String str1, String str2) {
        if (str1.length() != str2.length()) {
            return false;
        }

        char[] charArray1 = str1.toCharArray();
        char[] charArray2 = str2.toCharArray();

        Arrays.sort(charArray1);
        Arrays.sort(charArray2);

        return Arrays.equals(charArray1, charArray2);
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the first string: ");
        String str1 = scanner.nextLine();

        System.out.print("Enter the second string: ");
        String str2 = scanner.nextLine();

        if (areAnagrams(str1, str2)) {
            System.out.println("The strings are anagrams.");
        } else {
            System.out.println("The strings are not anagrams.");
        }
    }
}
```

## 5. Reverse A string



```java
import java.util.Scanner;

public class ReverseString {

    public static String reverse(String str) {
        StringBuilder reversed = new StringBuilder();
        for (int i = str.length() - 1; i >= 0; i--) {
            reversed.append(str.charAt(i));
        }
        return reversed.toString();
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a string: ");
        String input = scanner.nextLine();

        String reversedString = reverse(input);
        System.out.println("Reversed String: " + reversedString);
    }
}
```

## 6. Rotate a string



```java
import java.util.Scanner;

public class RotateString {

    public static String rotate(String str, int rotateBy) {
        int n = str.length();
        rotateBy %= n;
        if (rotateBy < 0) {
            rotateBy += n;
        }
        return str.substring(n - rotateBy) + str.substring(0, n - rotateBy);
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a string: ");
        String input = scanner.nextLine();

        System.out.print("Enter the number of rotations
           (positive for right, negative for left): ");
        int rotations = scanner.nextInt();

        String rotatedString = rotate(input, rotations);
        System.out.println("Rotated String: " + rotatedString);
    }
}
```