IA2 – Advanced Java

1 Explain the changing the case of characters within a string with example

The method toLowerCase( ) converts all the characters in a string from uppercase to lower case.

The toUpperCase( ) method converts all the characters in a string from lowercase to uppercase.

String toLowerCase( )

String toUpperCase( )

Both methods return a String object that contains the uppercase or lowercase equivalent of the invoking String.

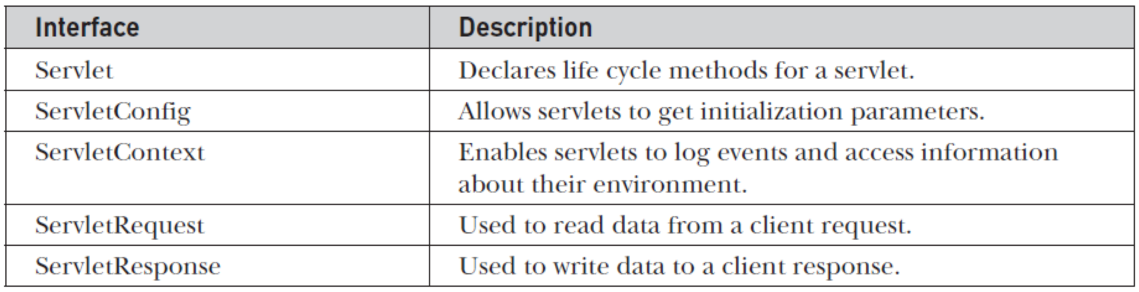
```java
public class CO {

    public static void main(String[] args) {
        String a = "hello";
        String b = "HELLO";

        // Convert the string 'a' to uppercase and print it
        System.out.println("toUpperCase()=" + a.toUpperCase());
        // Output: toUpperCase()=HELLO

        // Convert the string 'b' to lowercase and print it
        System.out.println("toLowerCase()=" + b.toLowerCase());
        // Output: toLowerCase()=hello
    }
}
```

2 Identify the interfaces & classes provided by javax.servlet package.

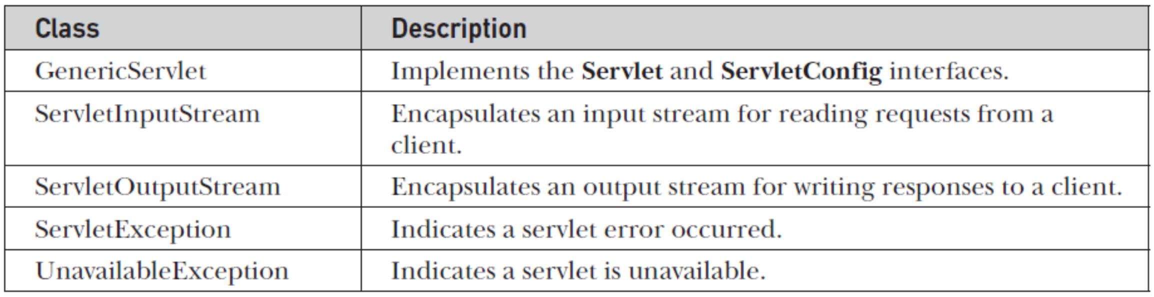
The `javax.servlet` package is a fundamental part of Java EE (Enterprise Edition) used for web development. Here are three key points about it:

1. **Core Interfaces for Servlets:**
    - The package defines essential interfaces such as `Servlet`, `ServletRequest`, `ServletResponse`, `ServletConfig`, and `ServletContext`. These interfaces provide the necessary methods for handling client requests, generating responses, configuring servlets, and interacting with the servlet container.
2. **Request and Response Handling:**
    - `ServletRequest` and `ServletResponse` are central to the package, enabling servlets to read client data (such as form inputs, query parameters) and send back responses (such as HTML content, binary data). These interfaces allow for processing and generating dynamic web content.

3. **Servlet Lifecycle Management:**
   - The `Servlet` interface defines the lifecycle methods of a servlet, including `init()`, `service()`, and `destroy()`. These methods allow for initialization, request processing, and cleanup tasks, respectively, ensuring that servlets can manage resources efficiently and handle client interactions properly.



Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletContext	Enables servlets to log events and access information about their environment.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.



Class	Description
GenericServlet	Implements the **Servlet** and **ServletConfig** interfaces.
ServletInputStream	Encapsulates an input stream for reading requests from a client.
ServletOutputStream	Encapsulates an output stream for writing responses to a client.
ServletException	Indicates a servlet error occurred.
UnavailableException	Indicates a servlet is unavailable.

3 Demonstrate briefly the Life Cycle of servlet.

The life cycle of a servlet begins when a user sends an HTTP request to the web server, prompting the server to load the servlet into memory and call its `init()` method for one-time initialization. Each incoming request triggers the `service()` method, where the servlet processes the request and generates a response. This method is invoked multiple times, once for each request. Finally, when the server decides to unload the servlet, it calls the `destroy()` method to release resources and perform cleanup before the servlet is removed from memory. This life cycle ensures efficient request handling and resource management.

- **User Request**
  - A user enters a URL in their web browser.
  - The browser creates an HTTP request for this URL.
  - This request is sent to the appropriate web server.
- **Server Receives Request**
  - The web server receives the HTTP request.
  - The server maps the request to a specific servlet.
  - The servlet is dynamically loaded into the server's memory.
- **Initialization (`init()` method)**
  - The server calls the `init()` method of the servlet.
  - This method is called only once, the first time the servlet is loaded.
  - Initialization parameters can be passed to the servlet for configuration.

- **Processing Requests (`service()` method)**
  - The server calls the `service()` method of the servlet.
  - This method handles the HTTP request.
  - The servlet can read data from the request and create a response for the client.
  - The servlet stays in memory and can handle more requests.
  - The `service()` method is called for each new HTTP request.
- **Shutdown (`destroy()` method)**
  - The server may decide to unload the servlet from memory.
  - The specific time and conditions for unloading depend on the server.
  - The server calls the `destroy()` method of the servlet.
  - This method releases any resources (like file handles) used by the servlet.
  - Important data can be saved to a persistent store.
  - The servlet's memory is then available for garbage collection.

4 Demonstrate length( ) and capacity( ), ensureCapacity( ),setLength( ), charAt( ) and setCharAt( ), getChars( ) with example

length() and capacity()

The `length()` method returns the current length of the `StringBuffer`, while `capacity()` returns the total allocated capacity.

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("buffer = " + sb);
        System.out.println("length = " + sb.length());
        System.out.println("capacity = " + sb.capacity());
    }
}

// Output:
// buffer = Hello
// length = 5
// capacity = 21
```

`ensureCapacity()`

The `ensureCapacity()` method preallocates room for a certain number of characters.

```
class EnsureCapacityDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("Initial capacity = " + sb.capacity());
        sb.ensureCapacity(50);
        System.out.println("Capacity after ensureCapacity(50) = " +
sb.capacity());
    }}

// Output:
// Initial capacity = 21
// Capacity after ensureCapacity(50) = 50
```

`setLength()`

The `setLength()` method sets the length of the `StringBuffer`. If the new length is greater, null characters are added. If it is less, characters beyond the new length are lost.

```
class SetLengthDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("buffer before = " + sb);
        sb.setLength(2);
        System.out.println("buffer after = " + sb);
    }
}

// Output:
// buffer before = Hello
// buffer after = He
```

`charAt()` and `setCharAt()`

The `charAt()` method retrieves the character at a specified index, while `setCharAt()` sets the character at a specified index.

```
// Demonstrate charAt() and setCharAt().
class SetCharAtDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("buffer before = " + sb);
        System.out.println("charAt(1) before = " + sb.charAt(1));
        sb.setCharAt(1, 'i');
        sb.setLength(2);
        System.out.println("buffer after = " + sb);
        System.out.println("charAt(1) after = " + sb.charAt(1));
    }
}

// Output:
// buffer before = Hello
// charAt(1) before = e
// buffer after = Hi
// charAt(1) after = i
```

getChars()

The `getChars()` method copies a substring of the `StringBuffer` into a character array.

```
class GetCharsDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello World");
        char[] target = new char[5];
        sb.getChars(0, 5, target, 0);
        System.out.print("Copied characters: ");
        for(char c : target) {
            System.out.print(c);
        }
    }
}

// Output:
// Copied characters: Hello
```

5 Demonstrate the Modifying a String with program example.

Because String objects are immutable, whenever you want to modify a String, you must either copy it into a StringBuffer or StringBuilder, or use one of the following String methods, which will construct a new copy of the string with your modifications complete.

substring()

The `substring()` method can be used to extract a part of a string. It has two forms:

1. **String substring(int startIndex):**
   - Extracts a substring starting from the specified index to the end of the string.
2. **String substring(int startIndex, int endIndex):**
   - Extracts a substring starting from the specified start index and ending at the specified end index (exclusive).

```
public class CO {
    public static void main(String[] args) {
        String s1 = "hello hru";
        System.out.println("Sub string from index position = " + s1.substring(1));
        System.out.println("Sub string from between index position = " + s1.substring(1, 5));
    }
}

```

```
// Output:
// Sub string from index position = ello hru
// Sub string from between index position = ello

```

concat()

The `concat()` method concatenates two strings, creating a new string that contains the invoking string with the contents of the specified string appended to the end.

```
public class CO {
    public static void main(String[] args) {
        String s1 = "wel come to";
        String s2 = " cec";
        System.out.println("Concatenation = " + s1.concat(s2));
    }
}

// Output:
// Concatenation = wel come to cec
```

The `replace()` method has two forms:

`CharSequence` in Java is like a blueprint for anything that can hold a series of characters you can read. It's more flexible than `String` because it includes `String`, `StringBuilder`, and `StringBuffer` as types of sequences you can work with. While `String` is fixed to one type of sequence, `CharSequence` lets you handle different kinds of character sequences in a similar way.

1. **String replace(char original, char replacement):**
    - Replaces all occurrences of one character with another character.
2. **String replace(CharSequence original, CharSequence replacement):**
    - Replaces one character sequence with another.

```
public class CO {
    public static void main(String[] args) {
        String s1 = "wel come to cec";
        CharSequence s2 = "hello hru";
        System.out.println("Replace string = " + s1.replace('e', 'E'));
        System.out.println("Replace charsequence = " + s1.replace("e", "E"));
    }
}

// Output:
// Replace string = wEl comE to cEc
// Replace charsequence = wEl comE to cEc

```

trim()

The `trim()` method returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

```
public class CO {
    public static void main(String[] args) {
        String s1 = " wel come to cec ";
        System.out.println("trim() = " + s1.trim());
    }
}

// Output:
// trim() = wel come to cec
```

6 Demonstrate data Conversion using Valueof() with example.

- `valueOf()` method converts data into a human-readable `String` format.
- It's static and overloaded in `String` for all Java built-in types and `Object`.
- Overloaded forms:
    - `String valueOf(double num)`
    - `String valueOf(long num)`
    - `String valueOf(Object ob)`
    - `String valueOf(char chars[])`
    - `String valueOf(char chars[], int startIndex, int numChars)`

```
public class CO {
    public static void main(String[] args) {
        // Initialize variables
        int a = 10;
        float b = 10;
        double c = 10.0;
        char d = 'a';
        char e[] = {'a', 'b', 'c'};

        // Output using valueOf() method
        System.out.println("String.valueOf(int) = " +
String.valueOf(a));         // Output: String.valueOf(int)=10
        System.out.println("String.valueOf(float) = " +
String.valueOf(b));       // Output: String.valueOf(float)=10.0
        System.out.println("String.valueOf(double) = " +
String.valueOf(c));      // Output: String.valueOf(double)=10.0
        System.out.println("String.valueOf(char) = " +
String.valueOf(d));        // Output: String.valueOf(char)=a
```

```
        System.out.println("String.valueOf(char, index, index) = " +
String.valueOf(e, 1, 2));  // Output:
String.valueOf(char,index,index)=bc
    }
}
```

- **`String.valueOf(int)`**: Converts integer `a` (value 10) to a string representation.
- **`String.valueOf(float)`**: Converts float `b` (value 10.0) to a string representation.
- **`String.valueOf(double)`**: Converts double `c` (value 10.0) to a string representation.
- **`String.valueOf(char)`**: Converts character `d` (value 'a') to a string representation.

- **`String.valueOf(char, index, index)`**: Converts a portion of character array `e` starting from index 1 and including 2 characters ('b' and 'c') to a string representation.

7 Explain equals ( ) and equalsIgnoreCase ( ),reverse() startsWith( ) & endsWith(), differ with suitable programming example

reverse( ) :
We can reverse the characters within a **StringBuffer** object using **reverse( )**, shown here:

StringBuffer reverse( )

```
public class CO {
    public static void main(String[] args) {
        // Create a StringBuffer object initialized with "Hello"
        StringBuffer b = new StringBuffer("Hello");

        // Print the original string
        System.out.println("Original String: " + b);

        // Reverse the StringBuffer and print the reversed string
        System.out.println("Reverse String: " + b.reverse());
    }
}
```

`equals()`

- **Usage**: Compares two strings for equality based on character sequence and case sensitivity.
- **General Form**: `boolean equals(Object str)`
- **Description**: Returns `true` if the invoking string object and the specified string `str` have the exact same characters in the same order; otherwise, returns `false`.

`equalsIgnoreCase()`

- **Usage**: Compares two strings for equality while ignoring differences in case sensitivity.
- **General Form**: `boolean equalsIgnoreCase(String str)`

- **Description**: Returns `true` if the invoking string object and the specified string `str` have the same characters in the same order, treating uppercase letters (`A-Z`) as equivalent to their corresponding lowercase letters (`a-z`); otherwise, returns `false`.

```java
public class CO {
    public static void main(String[] args) {
        String a = "hello";
        String b = "Hello";

        // Using equals() method
        System.out.println("equals() = " + a.equals(b));  // Output:
equals() = false

        // Using equalsIgnoreCase() method
        System.out.println("equalsIgnoreCase() = " +
a.equalsIgnoreCase(b));  // Output: equalsIgnoreCase() = true
    }
}
```

`startsWith()`

- **Usage**: Checks if a string begins with a specified prefix.
- **General Forms**:
  - `boolean startsWith(String str)`: Returns `true` if the invoking string starts with the specified string `str`; otherwise, returns `false`.
  - `boolean startsWith(String str, int startIndex)`: Returns `true` if the substring of the invoking string beginning at the specified `startIndex` starts with the string `str`; otherwise, returns `false`.

`endsWith()`

- **Usage**: Checks if a string ends with a specified suffix.
- **General Form**: `boolean endsWith(String str)`
- **Description**: Returns `true` if the invoking string ends with the specified string `str`; otherwise, returns `false`.

```java
public class CO {
    public static void main(String[] args) {
        String a = "hello";

        // Using startsWith() method
        System.out.println("Start with = " + a.startsWith("h"));    //
Output: Start with = true

        // Using endsWith() method
        System.out.println("End with = " + a.endsWith("llo"));      //
Output: End with = true

        // Using startsWith() method with start index
        System.out.println("Start with index = " + a.startsWith("llo",
2));  // Output: Start with index = true
    }
}
```

8 Demonstrate briefly string literals with suitable examples

A **string literal** in Java is just a piece of text enclosed within double quotes, like `"hello"` or `"123"`. When you write a string literal in your Java code, it's automatically treated as a **String object**. This means you can use it like any other object in Java, with methods and operations that are available for strings.

For example, `"hello".length()` will give you the length of the string `"hello"`.

Additionally, if you have an **array of characters** (like `char[] chars = {'a', 'b', 'c'}`), you can create a String object from this array using the `new` keyword:

String str = new String(chars);

Here, `str` will be a String object containing the characters `'a'`, `'b'`, and `'c'`.

Java automatically constructs a **String** object. Thus, you can use a string literal to initialize a **String** object.

String b=”hello”;
System.out.println(“Length of string=”+“hello”.length());

9 Demonstrate getchar ( ) method with suitable programming example. Ans- 4th in pdf

10 Any 2 string classes with examples

```java
public class StrCon {
    public static void main(String[] args) {
        // Create an empty String
        String a = new String();
        System.out.println("Empty String: " + a);

        // Create a String from a char array
        char ch[] = {'a', 'b', 'c', 'd'};
        String b = new String(ch);
        System.out.println("String with one argument as Char: " + b);

        // Output from here onwards as per the provided output
        String c = new String(ch, 1, 3);
        System.out.println("String with Three argument as Char: " + c);

        // Create a String from another String object
        String d = new String(b);
        System.out.println("String with String object: " + d);

        // Create a String from a byte array
        byte e[] = {65, 66, 67, 68, 69};
        String f = new String(e);
        System.out.println("byte to String: " + f);

        // Create a substring from a byte array
        String g = new String(e, 1, 3);
        System.out.println("byte to string for subbyte: " + g);

        // Create a String from a StringBuffer
        StringBuffer h = new StringBuffer("hello");
        String i = new String(h);
        System.out.println("StringBuffer to String: " + i);

        // Create a String from a StringBuilder
        StringBuilder j = new StringBuilder("welcome");
        String k = new String(j);
        System.out.println("StringBuilder to String: " + k);

        // Create a String from a code point array
        int l[] = {66, 67, 68, 69, 70};
        String m = new String(l, 1, 3);
        System.out.println("codepoint to String: " + m);
    }
}
```

```
Output:

Empty String:
String with one argument as Char: abcd
String with Three argument as Char: bcd
String with String object: abcd
byte to String: ABCDE
byte to string for subbyte: BC
StringBuffer to String: hello
StringBuilder to String: welcome
codepoint to String: CDE
```

1	**String()** To create an empty **String**, you call the default constructor. For example, String s = new String(); will create an instance of **String** with no characters in it.
2	**String(byte[] bytes)** **The String** class provides constructors that initialize a string when given a **byte** array. Their forms are shown here: String(byte *asciiChars*[ ]) Here, *asciiChars* specifies the array of bytes
3	**String(byte[] bytes, int startindex, int numchar)**
	This constructs a new String by decoding the specified subarray of bytes using the platform's default charset

11 Character Extraction

5.1 charAt()

To extract a single character from a String, you can use the `charAt()` method. It has this general form:

```
char charAt(int where)
```

Here, `where` is the index of the character that you want to obtain. The value of `where` must be nonnegative and within the bounds of the string. `charAt()` returns the character at the specified location. For example:

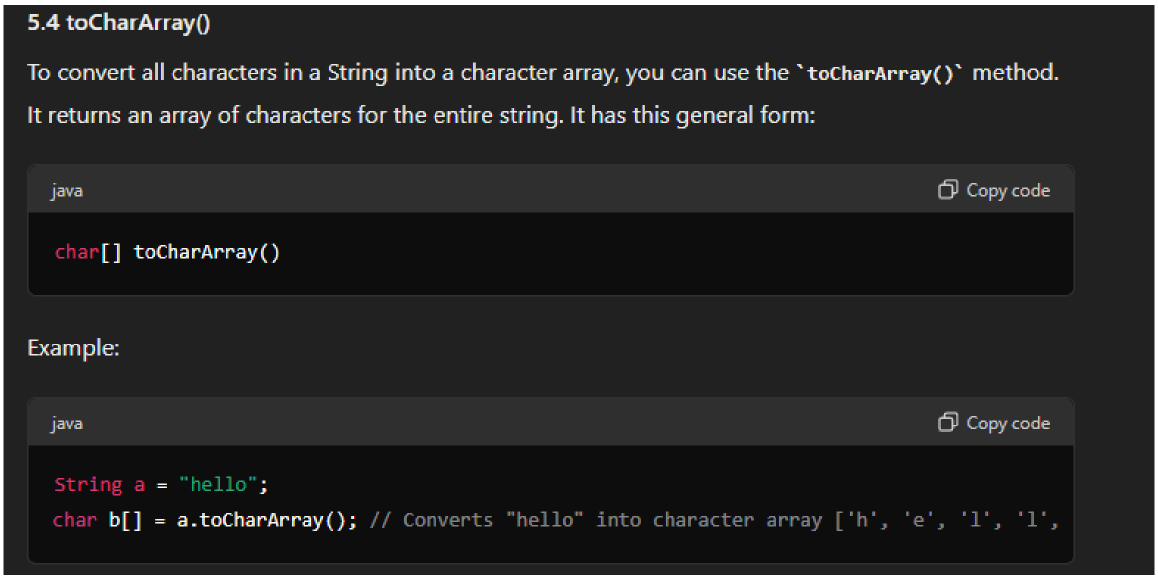
```
char ch;
ch = "abc".charAt(1); // assigns the value 'b' to ch
```

5.2 getChars()

If you need to extract more than one character at a time, you can use the `getChars()` method. It has this general form:

```
void getChars(int sourceStart, int sourceEnd, char target[], int
targetStart)
```

- `sourceStart` specifies the index of the beginning of the substring.
- `sourceEnd` specifies an index that is one past the end of the desired substring.
- The substring extracted contains characters from `sourceStart` through `sourceEnd-1`.
- `target` is the array that will receive the characters.
- `targetStart` is the index within `target` at which the substring will be copied.



5.4 toCharArray()

To convert all characters in a String into a character array, you can use the `` `toCharArray()` `` method. It returns an array of characters for the entire string. It has this general form:

```java
char[] toCharArray()
```

Example:

```java
String a = "hello";
char b[] = a.toCharArray(); // Converts "hello" into character array ['h', 'e', 'l', 'l',
public class CO {
    public static void main(String[] args) {
        String a = "hello";

        // Using charAt()
        char c = a.charAt(1);
        System.out.println("charAt=" + c); // Output: charAt=e

        // Using getChars()
        char ch[] = new char[2];
        a.getChars(1, 3, ch, 0);
        System.out.println(ch); // Output: el

        // Using getBytes()
        byte b[] = a.getBytes();
        System.out.println(b); // Output: [B@19821f (byte array
representation)

        // Using toCharArray()
        char ch1[] = a.toCharArray();
        System.out.println(ch1); // Output: hello
    }
}
```

12 Demonstrate Lambda Expressions Exceptions and Variable Capture.

- **Access to Enclosing Scope**: Lambda expressions in Java can access instance variables and static variables defined in their enclosing class. They can also use `this`, which refers to the instance of the enclosing class where the lambda is defined. This allows lambda expressions to read and modify instance variables or call methods of the enclosing class.

- **Local Variable Capture**: When a lambda expression uses a local variable from its enclosing scope, it must adhere to the rule of effectively final variables. An effectively final variable is one that is assigned a value only once and that value does not change afterward. It doesn't need to be explicitly declared as `final`, but if it were, it would still be valid.

- **Effectively Final Variables**: Lambda expressions cannot modify local variables from their enclosing scope. Modifying such a variable would remove its effectively final status and make it illegal for the lambda expression to capture it. This rule ensures that the lambda expression does not cause unintended side effects or conflicts with other parts of the program.

- **Example Illustration**: In the example provided, `num` is effectively final because it is assigned a value (`int num = 10;`) and not modified afterward. The lambda expression `myLambda` can use `num` in its computation (`int v = num + n;`), but any attempt to modify `num` within the lambda expression (`num++;`) or outside it (`num = 9;`) would result in a compilation error.

- **Instance Variables**: Unlike local variables, lambda expressions can freely access and modify instance variables of their enclosing class without restrictions.

```java
// An example of capturing a local variable from the enclosing scope.
interface MyFunc {
    int func(int n);
}

public class VarCapture {
    public static void main(String args[]) {
        // A local variable that can be captured.
        int num = 10;

        MyFunc myLambda = (n) -> {
            // This use of num is OK. It does not modify num.
            int v = num + n;
            // However, the following is illegal because it attempts
            // to modify the value of num.
            // num++;
            return v;
        };

        // The following line would also cause an error, because
        // it would remove the effectively final status from num.
        // num = 9;
        // Example usage:
        int result = myLambda.func(5);
        System.out.println("Result: " + result); // Output: Result: 15
    }
}
```

13 Demonstrate difference between string buffer and string builder with suitable example.

StringBuffer

1. **Thread-Safe**: StringBuffer is synchronized, meaning it is thread-safe. Multiple threads can safely manipulate a StringBuffer object without external synchronization.
2. **Performance**: Due to synchronization, StringBuffer operations are slower compared to StringBuilder.
3. **Usage**: StringBuffer is used in scenarios where thread safety is required, such as in multi-threaded applications where multiple threads may access and modify the same StringBuffer object concurrently.

Example using StringBuffer:

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

        sb.append(" ");
        sb.append("World");

        System.out.println(sb.toString());  // Output: Hello World
    }
}
```

StringBuilder

1. **Not Thread-Safe**: StringBuilder is not synchronized, making it faster than StringBuffer.
2. **Performance**: StringBuilder operations are faster due to lack of synchronization overhead.
3. **Usage**: StringBuilder is preferred in single-threaded environments or situations where thread safety is not a concern, such as in performance-critical applications.

Example using StringBuilder:

```java
public class StringBuilderExample {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Hello");

        sb.append(" ");
        sb.append("World");

        System.out.println(sb.toString());  // Output: Hello World
    }
}
```

String Builder

```java
public class StrCon {
    public static void main(String[] args) {
        // Creating an empty String
        String a = new String();
        System.out.println("Empty String: " + a);

        // Creating a String from a char array
        char ch[] = {'a', 'b', 'c', 'd'};
        String b = new String(ch);
        System.out.println("String with one argument as Char: " + b);

        // Creating a substring from a char array
        String c = new String(ch, 1, 3);
        System.out.println("String with Three arguments as Char: " +
c);

        // Creating a String from another String object
        String d = new String(b);
        System.out.println("String with String object: " + d);

        // Creating a String from a byte array
        byte e[] = {65, 66, 67, 68, 69};
        String f = new String(e);
        System.out.println("byte to String: " + f);

        // Creating a substring from a byte array
        String g = new String(e, 1, 3);
        System.out.println("byte to String for subbyte: " + g);
        // Creating a String from a StringBuffer
        StringBuffer h = new StringBuffer("hello");
        String i = new String(h);
        System.out.println("StringBuffer to String: " + i);

        // Creating a String from a StringBuilder
        StringBuilder j = new StringBuilder("welcome");
        String k = new String(j);
        System.out.println("StringBuilder to String: " + k);

        // Creating a String from an int array (code points)
        int l[] = {66, 67, 68, 69, 70};
        String m = new String(l, 1, 3);
        System.out.println("code point to String: " + m);
    }
}
```

14 Demonstrate Bounded types with programming example.

In Java generics, bounded types refer to restrictions that can be placed on the types that can be used as type arguments when instantiating a generic class or invoking a generic method. These restrictions enforce specific relationships or capabilities that the type parameter must have.

There are two main types of bounded types in Java generics:

1. **Upper Bounded Wildcards (`<? extends Type>`):**
   - Specifies that the type argument must be a subtype of a particular class or implement a particular interface.
   - Example: `<? extends Number>` means the type can be any subclass of `Number`, such as `Integer`, `Double`, etc.
   - This allows the generic type to accept instances of the specified type or any of its subclasses.
2. **Lower Bounded Wildcards (`<? super Type>`):**
   - Specifies that the type argument must be a superclass of a particular class.
   - Example: `<? super Integer>` means the type can be `Integer` or any superclass of `Integer`, such as `Number` or `Object`.
   - This allows the generic type to accept instances of the specified type or any superclass.

- **Problem Statement:** You want to create a generic class `Stats` that computes the average of an array of numbers of any type (`Integer`, `Float`, `Double`, etc.).

- **Initial Attempt:** You define the class `Stats<T>` where `T` is a generic type parameter. The class has an array `nums` of type `T` to hold numbers passed to it.

- **Issue:** In the `average()` method of `Stats`, you try to compute the average by summing up the values in `nums` array. To do this, you attempt to call `doubleValue()` on each element of `nums` to convert it to `double`. However, the compiler doesn't know that `T` will always be a numeric type that supports `doubleValue()`.

- **Compiler Error:** Since `T` could be any type (not necessarily numeric), the compiler reports an error because it can't verify that `doubleValue()` exists for all possible types `T`.

- **Solution - Bounded Type Parameter:** To solve this, you can use a bounded type parameter that restricts `T` to be only numeric types (subclasses of `Number`). This ensures:

    - The compiler knows that `T` will have a `doubleValue()` method.
    - Only numeric types can be passed to `T`, preventing accidental use with non-numeric types.

- **Revised Approach:** Modify `Stats` class to use a bounded type parameter `T extends Number`. This tells the compiler that `T` can only be a subclass of `Number`, ensuring `doubleValue()` is available for all instances of `T`.

```
// Corrected Stats class using bounded type parameter
class Stats<T extends Number> {
    T[] nums; // nums is an array of type T

    // Constructor accepting an array of type T
    Stats(T[] o) {
        nums = o;
    }

    // Method to compute the average
    double average() {
        double sum = 0.0;
        for(int i = 0; i < nums.length; i++) {
            sum += nums[i].doubleValue(); // Compiler now knows this is
valid
        }
        return sum / nums.length;
    }
}
```

- **Bounded Types in Java:**

  - Java provides bounded types to constrain the types that can be used as type arguments in generic classes or methods.
  - An upper bound is specified using the `extends` keyword followed by a superclass or interface.
  - This ensures that the type parameter (`T`) can only be replaced by types that are either the specified superclass or subclasses thereof.

- **Applying Upper Bound in Generic Classes:**

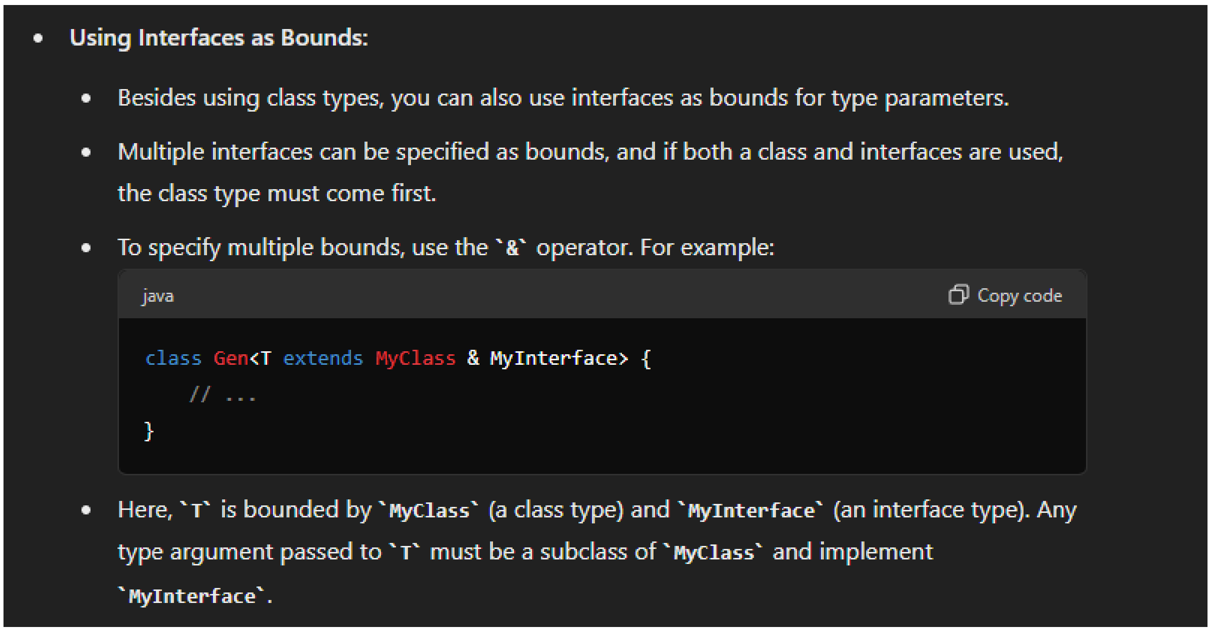
  - To ensure that a generic class handles specific types only (e.g., numeric types), you can use an upper bound.
  - In the example, the `Stats<T extends Number>` specifies that `T` can only be `Number` or its subclasses.

```
// In this version of Stats, the type argument for
// T must be either Number, or a class derived
// from Number.
class Stats<T extends Number> {
    T[] nums; // array of Number or subclass

    // Constructor accepting an array of type T
    Stats(T[] o) {
        nums = o;
    }

    // Method to compute the average
    double average() {
        double sum = 0.0;
        for(int i = 0; i < nums.length; i++) {
            sum += nums[i].doubleValue(); // Accessing doubleValue()
from Number
        }
        return sum / nums.length;
    }
}

// Demonstrate Stats.
class BoundsDemo {
    public static void main(String args[]) {
        // Integer array
        Integer inums[] = { 1, 2, 3, 4, 5 };
        Stats<Integer> iob = new Stats<Integer>(inums);
        double v = iob.average();
```

```java
        System.out.println("iob average is " + v);

        // Double array
        Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Double> dob = new Stats<Double>(dnums);
        double w = dob.average();
        System.out.println("dob average is " + w);

        // This won't compile because String is not a subclass of
Number.
        /*
        String strs[] = { "1", "2", "3", "4", "5" };
        Stats<String> strob = new Stats<String>(strs);
        double x = strob.average();
        System.out.println("strob average is " + v);
        */
    }
}
```

Advantages of Bounded Types (`Stats<T extends Number>`):

- By bounding type `T` with `Number`, Java ensures that all objects of type `T` can call `doubleValue()` because it's a method declared by `Number`.
- This restriction prevents the creation of non-numeric `Stats` objects. For instance, trying to instantiate `Stats<String>` will result in compile-time errors because `String` is not a subclass of `Number`.



- **Using Interfaces as Bounds:**
  - Besides using class types, you can also use interfaces as bounds for type parameters.
  - Multiple interfaces can be specified as bounds, and if both a class and interfaces are used, the class type must come first.
  - To specify multiple bounds, use the `&` operator. For example:

```java
class Gen<T extends MyClass & MyInterface> {
    // ...
}
```

  - Here, `T` is bounded by `MyClass` (a class type) and `MyInterface` (an interface type). Any type argument passed to `T` must be a subclass of `MyClass` and implement `MyInterface`.