Module 3

String Handling

String Constructors in Java

The String class in Java provides several constructors to create strings from different types of inputs. Here's an overview of these constructors and their usage:

1. Default Constructor

String s = new String();

- Creates an empty String object with no characters.

2. Character Array Constructor

String(char chars[])

- Initializes the String with the characters from the provided character array.
- Example:

char chars[] = { 'a', 'b', 'c' };

String s = new String(chars); // s = "abc"

3. Subrange of Character Array Constructor

String(char chars[], int startIndex, int numChars)

- Initializes the String with a subrange of the provided character array.
- startIndex specifies where the subrange begins, and numChars specifies the number of characters to use.
- Example:

char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

String s = new String(chars, 2, 3); // s = "cde"

4. String Object Constructor

String(String strObj)

- Initializes the String with the same sequence of characters as another String object.
- Example:

```
class MakeString {
   public static void main(String args[]) {
      char c[] = {'J', 'a', 'v', 'a'};
      String s1 = new String(c);
      String s2 = new String(s1);
```

```
        System.out.println(s1); // Output: Java
        System.out.println(s2); // Output: Java
    }
}
```

5. Byte Array Constructors

String(byte chrs[])

String(byte chrs[], int startIndex, int numChars)

- Initializes the String with characters from the provided byte array.
- The second form allows specifying a subrange.
- The byte-to-character conversion uses the default character encoding of the platform.
- Example:

```
class SubStringCons {
    public static void main(String args[]) {
        byte ascii[] = {65, 66, 67, 68, 69, 70 };
        String s1 = new String(ascii); // s1 = "ABCDEF"
        System.out.println(s1);
        String s2 = new String(ascii, 2, 3); // s2 = "CDE"
        System.out.println(s2);
    }
}
```

6. StringBuffer Constructor

String(StringBuffer strBufObj)

- Initializes the String with the contents of the given StringBuffer object.

7. StringBuilder Constructor

String(StringBuilder strBuildObj)

- Initializes the String with the contents of the given StringBuilder object.

8. Unicode Code Points Constructor

String(int codePoints[], int startIndex, int numChars)

- Initializes the String with characters from an array of Unicode code points.
- startIndex specifies where the subrange begins, and numChars specifies the number of code points to use.

9. Charset Encoding Constructors

- There are constructors that allow specifying a Charset for byte-to-string conversions. This lets you control how bytes are interpreted as characters, especially for different character encodings

String Length in Java

In Java, you can determine the length of a String using the length() method. This method returns the number of characters present in the String.

Syntax

int length()

- **Returns**: The length of the string as an integer.

Example

Here's a simple example demonstrating how to use the length() method:

```
class StringLengthExample {
   public static void main(String args[]) {
      // Create a character array
      char chars[] = { 'a', 'b', 'c' };

      // Create a String from the character array
      String s = new String(chars);

      // Print the length of the String
      System.out.println(s.length()); // Output: 3
   }
}
```

In this example:

- A char array is created with three characters: 'a', 'b', and 'c'.
- A String object s is initialized with this character array.
- The length() method is called on s, which returns 3 because the string "abc" contains three characters.

Special String Operations in Java

Java provides several convenient operations for working with strings, including automatic string creation from literals, concatenation, and conversion of other data types to strings. Here's a closer look at these special operations:

String Literals

In Java, you can create String objects either explicitly using the new operator or implicitly using string literals.

1. **Using String Literals**: When you use a string literal, Java automatically creates a String object. For example:

String s1 = new String("hello"); // Explicit creation

String s2 = "hello"; // Implicit creation using a string literal

Here, s2 is initialized directly with the string literal "hello". This is simpler and more efficient than explicitly creating a String object using the new operator.

2. **String Literal Example**:

char chars[] = { 'a', 'b', 'c' };

String s1 = new String(chars); // Explicit creation

String s2 = "abc"; // Implicit creation using a string literal

// Both s1 and s2 represent the same string

System.out.println(s1); // Output: abc

System.out.println(s2); // Output: abc

In this example, both s1 and s2 contain the string "abc". s2 is created using a string literal, which is more concise.

3. **Calling Methods on String Literals**: You can call methods directly on string literals:

System.out.println("abc".length()); // Output: 3

In this case, the string literal "abc" is used directly with the length() method, which returns the number of characters in the string.

String Concatenation

Java allows you to concatenate strings using the + operator. This operator combines multiple String objects or literals into a single string:

String firstName = "John";

String lastName = "Doe";

String fullName = firstName + " " + lastName;

System.out.println(fullName); // Output: John Doe

In this example, the + operator is used to concatenate firstName, a space " ", and lastName into the fullName string.

Conversion to String

Java provides several methods to convert other data types into strings. For example:

1. **Using String.valueOf()**:

int number = 100;

String numberStr = String.valueOf(number);

System.out.println(numberStr); // Output: 100

String.valueOf() converts the integer 100 to its string representation "100".

2. **Automatic Conversion with + Operator**:

int age = 25;

String ageStr = "Age: " + age;

System.out.println(ageStr); // Output: Age: 25

Here, the + operator automatically converts the integer age to its string representation and concatenates it with "Age: ".

String Concatenation with Other Data Types

In Java, string concatenation with other data types is straightforward and automatically converts non-string operands into their string representations. Here's how it works:

Concatenating Strings with Other Data Types

When you concatenate a string with other data types, Java automatically converts the non-string values into their string representations. For example:

int age = 9;

String s = "He is " + age + " years old.";

System.out.println(s); // Output: He is 9 years old.

In this example, the integer age is converted to its string representation "9", and the resulting string "He is 9 years old." is created.

Operator Precedence in Concatenation

Java follows operator precedence rules which can affect the result of string concatenation, especially when mixed with other operations. Consider the following example:

String s = "four: " + 2 + 2;

System.out.println(s); // Output: four: 22

In this case, the concatenation is performed as follows:

1. "four: " is concatenated with the string representation of the first 2, resulting in "four: 2".
2. "four: 2" is then concatenated with the string representation of the second 2, resulting in "four: 22".

To ensure arithmetic operations are performed before concatenation, use parentheses:

String s = "four: " + (2 + 2);

System.out.println(s); // Output: four: 4

Here, 2 + 2 is evaluated first, giving 4, and then concatenated with "four: ".

String Conversion with toString()

When concatenating objects with strings, Java uses the toString() method to obtain their string representations. Every class in Java inherits the toString() method from the Object class, which you can override to provide meaningful descriptions of your objects.

1. **Default toString() Method**:

Object obj = new Object();

System.out.println(obj.toString()); // Output: java.lang.Object@<hashcode>

The default implementation of toString() provides the class name followed by the object's hash code.

2. **Overriding toString()**: To provide a custom string representation, override the toString() method in your class. For example:

```
class Box {
   double width;
   double height;
   double depth;
   Box(double w, double h, double d) {
      width = w;
      height = h;
      depth = d;
   }

   @Override
   public String toString() {
      return "Dimensions are " + width + " by " + depth + " by " + height + ".";
```

```
    }
}
class toStringDemo {
    public static void main(String args[]) {
        Box b = new Box(10, 12, 14);
        System.out.println(b); // Output: Dimensions are 10.0 by 14.0 by 12.0.
    }
}
```

Character Extraction in Java

Java provides several methods for extracting characters from a String object. Here's a detailed look at the available methods:

1. charAt(int index)

The charAt() method is used to retrieve a single character from a specific index in the string. The index is zero-based, meaning the first character is at index 0.

Syntax:

char charAt(int index)

- **index**: The position of the character to retrieve.

Example:

```
String s = "abc";
char ch = s.charAt(1); // ch will be 'b'
System.out.println(ch); // Output: b
```

2. getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

The getChars() method copies characters from a substring of the string into a specified character array.

Syntax:

void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

- **srcBegin**: The starting index of the substring to copy.
- **srcEnd**: The ending index (one past the end) of the substring.
- **dst**: The destination character array.
- **dstBegin**: The starting index in the destination array.

Example:

```
class GetCharsDemo {
   public static void main(String[] args) {
      String s = "This is a demo of the getChars method.";
      int start = 10;
      int end = 14;
      char[] buf = new char[end - start];
      s.getChars(start, end, buf, 0);
      System.out.println(buf); // Output: demo
   }
}
```

3. getBytes()

The getBytes() method converts the string into an array of bytes using the platform's default character encoding.

Syntax:

byte[] getBytes()

Example:

String s = "hello";

byte[] bytes = s.getBytes();

System.out.println(Arrays.toString(bytes)); // Output might be [104, 101, 108, 108, 111] for ASCII encoding

For specific character encodings, you can use:

byte[] getBytes(String charsetName)

Example:

byte[] bytes = s.getBytes("UTF-8");

4. toCharArray()

The toCharArray() method converts the entire string into a new character array.

Syntax:

char[] toCharArray()

Example:

String s = "hello";

char[] chars = s.toCharArray();

System.out.println(Arrays.toString(chars)); // Output: [h, e, l, l, o]

String Comparison in Java

Java's String class provides several methods for comparing strings or substrings. Here's a detailed look at the primary methods used for string comparison:

1. equals(Object obj)

The equals() method compares the invoking String object with another String object to determine if they are equal. The comparison is case-sensitive.

Syntax:

boolean equals(Object obj)

- **obj**: The String object to compare with the invoking String object.

Example:

String s1 = "Hello";

String s2 = "Hello";

String s3 = "Good-bye";

System.out.println(s1.equals(s2)); // Output: true

System.out.println(s1.equals(s3)); // Output: false

2. equalsIgnoreCase(String str)

The equalsIgnoreCase() method compares two String objects for equality, ignoring case differences. This means it treats uppercase and lowercase letters as equivalent.

Syntax:

boolean equalsIgnoreCase(String str)

- **str**: The String object to compare with the invoking String object.

Example:

String s1 = "Hello";

String s4 = "HELLO";

System.out.println(s1.equalsIgnoreCase(s4)); // Output: true

Example Program

Here is a sample program that demonstrates the use of equals() and equalsIgnoreCase() methods:

```
class EqualsDemo {
   public static void main(String[] args) {
      String s1 = "Hello";
```

```
        String s2 = "Hello";
        String s3 = "Good-bye";
        String s4 = "HELLO";

        System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
        System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3));
        System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4));
        System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " + s1.equalsIgnoreCase(s4));
    }
}
```

Output:

Hello equals Hello -> true

Hello equals Good-bye -> false

Hello equals HELLO -> false

Hello equalsIgnoreCase HELLO -> true

3. RegionMatches()

The regionMatches() method compares a specific region of one string with a region of another string. There are two overloaded forms of this method:

Syntax:

boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars)

boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)

- **startIndex**: Index within the invoking String object where the region starts.
- **str2**: The String object being compared.
- **str2StartIndex**: Index within str2 where the comparison begins.
- **numChars**: Number of characters to compare.
- **ignoreCase** (in the second version): If true, case is ignored during comparison; otherwise, it is case-sensitive.

Example:

String s1 = "HelloWorld";

String s2 = "World";

```
boolean result1 = s1.regionMatches(5, s2, 0, 5); // true

boolean result2 = s1.regionMatches(true, 5, "WORLD", 0, 5); // true, case ignored

System.out.println(result1);

System.out.println(result2);
```

4. startsWith() and endsWith()

The startsWith() and endsWith() methods are used to check if a string starts or ends with a specified substring.

Syntax:

```
boolean startsWith(String str)

boolean endsWith(String str)

boolean startsWith(String str, int startIndex)
```

- **str**: The String being checked.
- **startIndex**: Index at which to start the search for the startsWith() method.

Example:

```
String s = "Foobar";

boolean starts = s.startsWith("Foo"); // true

boolean ends = s.endsWith("bar"); // true

boolean startsFrom = s.startsWith("bar", 3); // true


System.out.println(starts);

System.out.println(ends);

System.out.println(startsFrom);
```

5. equals() vs ==

- **equals()**: Compares the content of two String objects. It returns true if the strings have the same sequence of characters.
- ==: Compares the references of two String objects to see if they point to the same memory location.

Example:

```
class EqualsNotEqualTo {

   public static void main(String[] args) {

      String s1 = "Hello";
```

```
        String s2 = new String(s1);
        System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
        System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
    }
}
```

Output:

Hello equals Hello -> true

Hello == Hello -> false

In this example, s1 refers to the string literal "Hello", while s2 is a new String object initialized with the value of s1. Although the contents of s1 and s2 are the same, they are different objects in memory, which is why s1 == s2 returns false.

compareTo() and compareToIgnoreCase() Methods in Java

The compareTo() method is essential for comparing strings in Java, particularly for sorting. It allows you to determine the order of two strings based on their lexicographic (dictionary) order. Here's how it works and how it differs from compareToIgnoreCase():

1. compareTo()

The compareTo() method is used to compare two strings lexicographically. It returns an integer that indicates the relative order of the strings.

Syntax:

int compareTo(String str)

- **str**: The String object to be compared with the invoking string.

Return Value:

- **Less than zero**: The invoking string is lexicographically less than the str.
- **Greater than zero**: The invoking string is lexicographically greater than the str.
- **Zero**: Both strings are equal.

Example:

```
class CompareToDemo {
    public static void main(String[] args) {
        String s1 = "apple";
        String s2 = "banana";
        String s3 = "apple";
```

```
        System.out.println(s1.compareTo(s2)); // Negative value (s1 < s2)
        System.out.println(s1.compareTo(s3)); // Zero (s1 == s3)
        System.out.println(s2.compareTo(s1)); // Positive value (s2 > s1)
    }
}
```

Output:

-1

0

1

2. compareToIgnoreCase()

The compareToIgnoreCase() method compares two strings lexicographically, ignoring case differences.

Syntax:

int compareToIgnoreCase(String str)

- **str**: The String object to be compared with the invoking string.

Return Value:

- **Less than zero**: The invoking string is less than the str when case differences are ignored.
- **Greater than zero**: The invoking string is greater than the str when case differences are ignored.
- **Zero**: Both strings are equal when case differences are ignored.

Example:

```
class CompareToIgnoreCaseDemo {
    public static void main(String[] args) {
        String s1 = "Apple";
        String s2 = "banana";
        String s3 = "APPLE";
        System.out.println(s1.compareToIgnoreCase(s2)); // Negative value (s1 < s2, case ignored)
        System.out.println(s1.compareToIgnoreCase(s3)); // Zero (s1 == s3, case ignored)
        System.out.println(s2.compareToIgnoreCase(s1)); // Positive value (s2 > s1, case ignored)
```

```
    }
}
```

Output:

```
-1
0
1
```

Sorting Strings with compareTo()

Here's an example that demonstrates sorting an array of strings using compareTo():

Bubble Sort Example:

```
class SortString {
    static String arr[] = {
        "Now", "is", "the", "time", "for", "all", "good", "men",
        "to", "come", "to", "the", "aid", "of", "their", "country"
    };

    public static void main(String args[]) {
        for (int j = 0; j < arr.length; j++) {
            for (int i = j + 1; i < arr.length; i++) {
                if (arr[i].compareTo(arr[j]) < 0) {
                    String t = arr[j];
                    arr[j] = arr[i];
                    arr[i] = t;
                }
            }
            System.out.println(arr[j]);
        }
    }
}
```

Output:

```
Now
```

aid

all

come

country

for

good

is

men

of

the

the

their

time

to

to

Searching Strings in Java

The String class in Java offers several methods for searching for characters or substrings within a string. The primary methods are indexOf(), lastIndexOf(), contains(), and startsWith()/endsWith(). Here's an overview of each method:

1. indexOf()

The indexOf() method returns the index of the first occurrence of a specified character or substring within the invoking string. If the character or substring is not found, it returns -1.

Syntax:

int indexOf(int ch)

int indexOf(int ch, int fromIndex)

int indexOf(String str)

int indexOf(String str, int fromIndex)

- **int ch**: The character to search for.
- **int fromIndex**: The index from which to start the search.
- **String str**: The substring to search for.

Example:

```
class IndexOfDemo {
   public static void main(String[] args) {
      String str = "Hello, world!";
      System.out.println(str.indexOf('o')); // Output: 4
      System.out.println(str.indexOf('o', 5)); // Output: 8
      System.out.println(str.indexOf("world")); // Output: 7
      System.out.println(str.indexOf("Java")); // Output: -1
   }
}
```

2. lastIndexOf()

The lastIndexOf() method returns the index of the last occurrence of a specified character or substring within the invoking string. If the character or substring is not found, it returns -1.

Syntax:

```
int lastIndexOf(int ch)
int lastIndexOf(int ch, int fromIndex)
int lastIndexOf(String str)
int lastIndexOf(String str, int fromIndex)
```

- **int ch**: The character to search for.
- **int fromIndex**: The index from which to start the search, searching backwards.
- **String str**: The substring to search for.

Example:

```
class LastIndexOfDemo {
   public static void main(String[] args) {
      String str = "Hello, world! Hello again!";
      System.out.println(str.lastIndexOf('o')); // Output: 20
      System.out.println(str.lastIndexOf('o', 10)); // Output: 4
      System.out.println(str.lastIndexOf("Hello")); // Output: 13
      System.out.println(str.lastIndexOf("Java")); // Output: -1
   }
}
```

3. contains()

The contains() method checks if the invoking string contains a specified substring. It returns true if the substring is found, and false otherwise.

Syntax:

boolean contains(CharSequence sequence)

- **CharSequence sequence**: The sequence of characters to search for.

Example:

```
class ContainsDemo {
   public static void main(String[] args) {
      String str = "Hello, world!";
      System.out.println(str.contains("world")); // Output: true
      System.out.println(str.contains("Java")); // Output: false
   }
}
```

4. startsWith() and endsWith()

- **startsWith(String prefix)**: Checks if the invoking string starts with the specified prefix. Returns true if it does, otherwise false.
- **endsWith(String suffix)**: Checks if the invoking string ends with the specified suffix. Returns true if it does, otherwise false.

Syntax:

boolean startsWith(String prefix)

boolean startsWith(String prefix, int toffset)

boolean endsWith(String suffix)

- **String prefix**: The prefix to check for.
- **String suffix**: The suffix to check for.
- **int toffset**: The index from which to start the search for the prefix.

Example:

```
class StartsEndsWithDemo {
   public static void main(String[] args) {
      String str = "Hello, world!";
      System.out.println(str.startsWith("Hello")); // Output: true
```

```
        System.out.println(str.startsWith("world", 7)); // Output: true
        System.out.println(str.endsWith("world!")); // Output: true
        System.out.println(str.endsWith("world")); // Output: false
    }
}
```

Modifying a String in Java

Because String objects in Java are immutable, modifying a string involves creating a new string with the desired changes. Here are some common methods used to achieve this:

1. substring()

The substring() method extracts a part of the string and returns it as a new string.

Syntax:

java

Copy code

String substring(int startIndex)

String substring(int startIndex, int endIndex)

- **startIndex**: The beginning index (inclusive).
- **endIndex**: The ending index (exclusive).

Example:

```
class SubstringDemo {
    public static void main(String[] args) {
        String str = "Hello, world!";
        System.out.println(str.substring(7)); // Output: world!
        System.out.println(str.substring(0, 5)); // Output: Hello
    }
}
```

Replacing Substrings

The following program replaces all instances of one substring with another within a string:

```
class StringReplace {
    public static void main(String[] args) {
        String org = "This is a test. This is, too.";
        String search = "is";
```

```
        String sub = "was";
        String result = "";
        int i;
        do {
            System.out.println(org);
            i = org.indexOf(search);
            if (i != -1) {
                result = org.substring(0, i);
                result += sub;
                result += org.substring(i + search.length());
                org = result;
            }
        } while (i != -1);
    }
}
```

Output:

This is a test. This is, too.

Thwas is a test. This is, too.

Thwas was a test. This is, too.

Thwas was a test. Thwas is, too.

Thwas was a test. Thwas was, too.

2. concat()

The concat() method concatenates two strings.

Syntax:

String concat(String str)

Example:

```
class ConcatDemo {
    public static void main(String[] args) {
        String s1 = "one";
        String s2 = s1.concat("two");
```

```
        System.out.println(s2); // Output: onetwo


        String s3 = s1 + "two";
        System.out.println(s3); // Output: onetwo
    }
}
```

3. replace()

The replace() method replaces occurrences of characters or character sequences.

Syntax:

String replace(char original, char replacement)

String replace(CharSequence target, CharSequence replacement)

Example:

```
class ReplaceDemo {
    public static void main(String[] args) {
        String str = "Hello";
        String result = str.replace('l', 'w');
        System.out.println(result); // Output: Hewwo


        String str2 = "Hello World";
        String result2 = str2.replace("World", "Java");
        System.out.println(result2); // Output: Hello Java
    }
}
```

4. trim()

The trim() method removes leading and trailing whitespace.

Syntax:

String trim()

Example:

```
class TrimDemo {
    public static void main(String[] args) {
```

```
        String str = "   Hello World   ";
        String result = str.trim();
        System.out.println(result); // Output: Hello World
    }
}
```

Using trim() in a practical example to process user input:

```
import java.io.*;
class UseTrim {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str;
        System.out.println("Enter 'stop' to quit.");
        System.out.println("Enter State: ");
        do {
            str = br.readLine();
            str = str.trim(); // remove whitespace
            if (str.equals("Illinois")) {
                System.out.println("Capital is Springfield.");
            } else if (str.equals("Missouri")) {
                System.out.println("Capital is Jefferson City.");
            } else if (str.equals("California")) {
                System.out.println("Capital is Sacramento.");
            } else if (str.equals("Washington")) {
                System.out.println("Capital is Olympia.");
            }
        } while (!str.equals("stop"));
    }
}
```

Data Conversion Using valueOf()

The valueOf() method in the String class converts various data types into their string representation. It is a static method overloaded to handle different data types.

Method Signatures

Here are some common forms:

static String valueOf(double num)

static String valueOf(long num)

static String valueOf(Object obj)

static String valueOf(char[] chars)

For a subset of a char array:

static String valueOf(char[] chars, int startIndex, int numChars)

Example:

```
class ValueOfDemo {
   public static void main(String[] args) {
      double num = 123.45;
      long lng = 9876543210L;
      char[] chars = {'J', 'a', 'v', 'a'};

      String strNum = String.valueOf(num);
      String strLng = String.valueOf(lng);
      String strChars = String.valueOf(chars);
      String strSubset = String.valueOf(chars, 1, 2);

      System.out.println("Double to String: " + strNum); // Output: 123.45
      System.out.println("Long to String: " + strLng); // Output: 9876543210
      System.out.println("Char array to String: " + strChars); // Output: Java
      System.out.println("Subset of Char array: " + strSubset); // Output: av
   }
}
```

Changing the Case of Characters Within a String

The toLowerCase() and toUpperCase() methods convert all characters in a string to lowercase and uppercase, respectively. They are locale-sensitive.

Method Signatures

String toLowerCase()

String toUpperCase()

Example:

```
class ChangeCase {
   public static void main(String[] args) {
      String s = "This is a test.";
      System.out.println("Original: " + s);
      String upper = s.toUpperCase();
      String lower = s.toLowerCase();

      System.out.println("Uppercase: " + upper); // Output: THIS IS A TEST.
      System.out.println("Lowercase: " + lower); // Output: this is a test.
   }
}
```

Joining Strings

The join() method concatenates two or more strings, separated by a delimiter.

Method Signature

static String join(CharSequence delim, CharSequence... strs)

Example:

```
class StringJoinDemo {
   public static void main(String[] args) {
      String result = String.join(" ", "Alpha", "Beta", "Gamma");
      System.out.println(result); // Output: Alpha Beta Gamma

      result = String.join(", ", "John", "ID#: 569", "E-mail: John@HerbSchildt.com");
      System.out.println(result); // Output: John, ID#: 569, E-mail: John@HerbSchildt.com
   }
}
```

In the first call to join(), a space is used as the delimiter. In the second call, a comma followed by a space is used as the delimiter.

StringBuffer in Java

Overview

The StringBuffer class in Java provides a modifiable sequence of characters. Unlike the String class, which represents immutable character sequences, StringBuffer allows characters and substrings to be inserted or appended dynamically. This class is designed for creating and manipulating strings that will change over time, providing efficient memory usage by pre-allocating space for growth.

Constructors

StringBuffer has several constructors to initialize the buffer in different ways:

1. **Default Constructor:**

StringBuffer()

   - Reserves room for 16 characters without reallocation.

2. **Size Constructor:**

StringBuffer(int size)

   - Accepts an integer to set the initial size of the buffer.

3. **String Constructor:**

StringBuffer(String str)

   - Initializes the buffer with a given string and reserves room for 16 additional characters.

4. **CharSequence Constructor:**

StringBuffer(CharSequence chars)

   - Initializes the buffer with the character sequence contained in chars and reserves room for 16 more characters.

Methods

length() and capacity()

- **length()**: Returns the current length of the character sequence.

int length()

- **capacity()**: Returns the total allocated capacity of the buffer.

int capacity()

ensureCapacity()

Ensures that the buffer has a minimum capacity.

void ensureCapacity(int minCapacity)

setLength()

Sets the length of the character sequence, truncating or extending as necessary.

void setLength(int len)

charAt() and setCharAt()

- **charAt()**: Returns the character at the specified index.

char charAt(int index)

- **setCharAt()**: Sets the character at the specified index.

void setCharAt(int index, char ch)

getChars()

Copies a substring into a character array.

void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

append()

Appends various types to the end of the buffer.

StringBuffer append(String str)

StringBuffer append(int num)

StringBuffer append(Object obj)

insert()

Inserts a string or value at a specified index.

StringBuffer insert(int offset, String str)

StringBuffer insert(int offset, char ch)

StringBuffer insert(int offset, Object obj)

reverse()

Reverses the character sequence.

StringBuffer reverse()

delete() and deleteCharAt()

- **delete()**: Removes characters from the buffer.

StringBuffer delete(int start, int end)

- **deleteCharAt()**: Removes the character at the specified index.

StringBuffer deleteCharAt(int index)

replace()

Replaces a substring with another string.

StringBuffer replace(int start, int end, String str)

substring()

Extracts a substring from the buffer.

String substring(int start)

String substring(int start, int end)

Examples

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

Using append()

```
class AppendDemo {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("Hello");
    sb.append(" World");
    System.out.println(sb);
  }
}
```

Hello World

Using insert()

```
class InsertDemo {
  public static void main(String[] args) {
```

```
    StringBuffer sb = new StringBuffer("I Java!");
    sb.insert(2, "like ");
    System.out.println(sb);
  }
}
```

Output:

I like Java!

Using reverse()

```
class ReverseDemo {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("abcdef");
    sb.reverse();
    System.out.println(sb);
  }
}
```

Output:

fedcba

Using delete() and deleteCharAt()

```
class DeleteDemo {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("This is a test.");
    sb.delete(4, 7);
    System.out.println("After delete: " + sb);
    sb.deleteCharAt(0);
    System.out.println("After deleteCharAt: " + sb);
  }
}
```

Output:

After delete: This a test.

After deleteCharAt: his a test.

Using replace()

```
class ReplaceDemo {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("This is a test.");
    sb.replace(5, 7, "was");
    System.out.println("After replace: " + sb);
  }
}
```

Output:

After replace: This was a test.

Using substring()

```
class SubstringDemo {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("This is a test.");
    System.out.println(sb.substring(5));
    System.out.println(sb.substring(5, 7));
  }
}
```

Output:

is a test.

is

StringBuffer provides a versatile way to handle mutable strings in Java, offering methods to modify, manipulate, and inspect character sequences efficiently.