Chapter 15
String Handling

Introduction:

- A *string* is a sequence of characters.
- But, unlike many other languages that implement strings as character arrays, Java **implements strings as objects of type *String***.
- Implementing strings as built-in objects allows Java to provide a **full complement of features** that make string handling convenient.
  - For example, Java has ***methods*** to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string.
  - Also, String objects can be constructed a number of ways, making it easy to obtain a string when needed.
- Somewhat unexpectedly, when you create a **String** object, you are creating a string that *cannot be changed*.
- The difference is that each time you need an altered version of an existing string, a ***new String*** object is created that contains the modifications. The original string is left unchanged.
- For those cases in which a modifiable string is desired, Java provides two options: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.

Introduction:

- The **String**, **StringBuffer**, and **StringBuilder** classes are defined in **java.lang**.
- All are declared **final**, which means that none of these classes may be subclassed.
- All three implement the **CharSequence** interface.
- However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

The String Constructors:

- The **String** class supports several constructors.
- 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.

- To create a String initialized by an array of characters, use the constructor shown here:

  String(char *chars*[ ])

- Here is an example:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

This constructor initializes **s** with the string “abc”.

The String Constructors:

- You can specify a subrange of a character array as an initializer using the following constructor:

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

  Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use.
- Here is an example:

  ```
  char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
  String s = new String(chars, 2, 3);
  ```

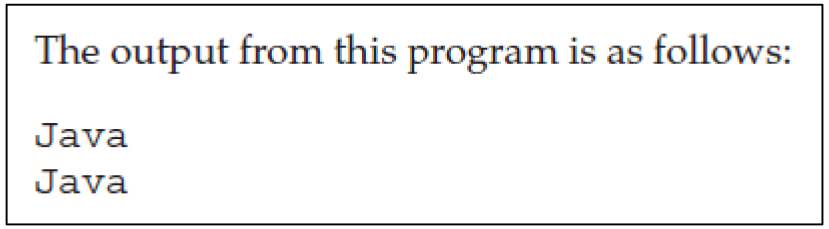
  This initializes s with the characters cde.

- You can construct a String object that contains the same character sequence as another String object using this constructor:

  ```
  String(String strObj)
  ```

The String Constructors:

```
// Construct one String from another.
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);
    System.out.println(s2);
  }
}
```



The output from this program is as follows:

```
Java
Java
```

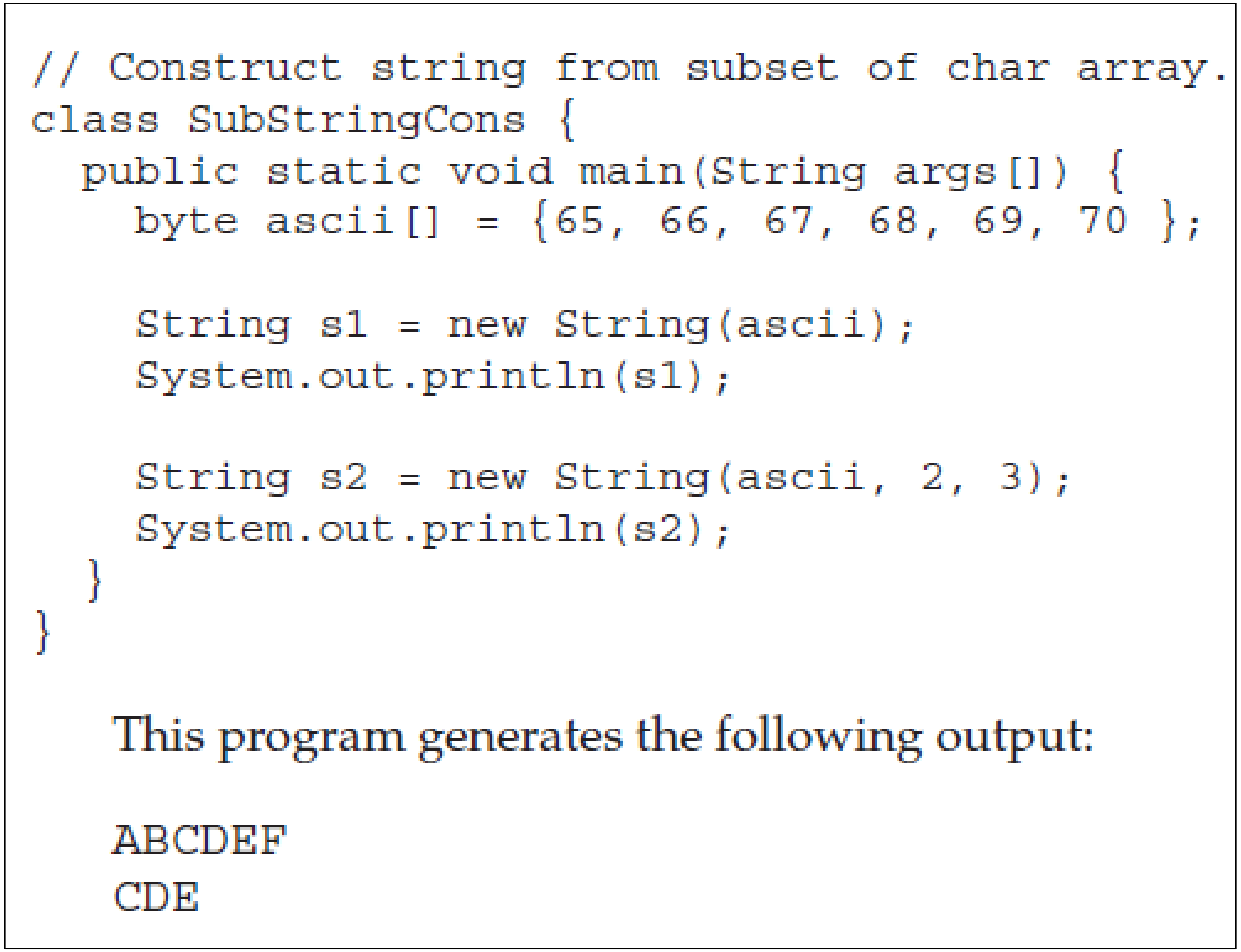
The String Constructors:

- Even though Java’s **char** type uses **16 bits** to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of **8-bit bytes** constructed from the ASCII character set.
- Because **8-bit ASCII strings** are common, the String class provides constructors that initialize a string when given a byte array.
- Their forms are shown here:

```
String(byte asciiChars[ ])
String(byte asciiChars[ ], int startIndex, int numChars)
```

- Here, asciiChars specifies the array of bytes.
- The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform.
- The program in next slide illustrates these constructors.

The String Constructors:



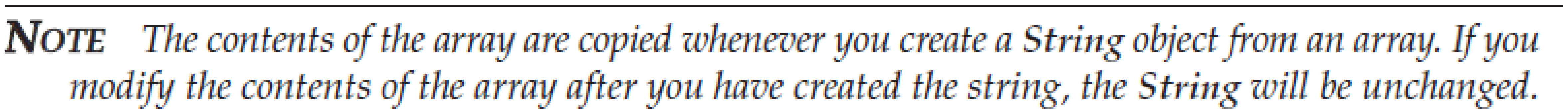
```
// Construct string from subset of char array.
class SubStringCons {
  public static void main(String args[]) {
    byte ascii[] = {65, 66, 67, 68, 69, 70 };

    String s1 = new String(ascii);
    System.out.println(s1);

    String s2 = new String(ascii, 2, 3);
    System.out.println(s2);
  }
}
```

This program generates the following output:

```
ABCDEF
CDE
```

The String Constructors:

- Extended versions of the byte-to-string constructors are also defined in which you can specify the character encoding that determines how bytes are converted to characters. However, most of the time, you will want to use the default encoding provided by the platform.



NOTE *The contents of the array are copied whenever you create a* **String** *object from an array. If you modify the contents of the array after you have created the string, the* **String** *will be unchanged.*

- You can construct a **String** from a **StringBuffer** by using the constructor shown here:

  String(StringBuffer strBufObj)

String Constructors Added by J2SE 5:

- Two constructors added.
- The first supports the **extended Unicode character set** and is shown here:
  String(int codePoints[ ], int startIndex, int numChars)
- Here, codePoints is an array that contains Unicode code points. The resulting string is constructed from the range that begins at startIndex and runs for numChars.

- The second new constructor supports the new **StringBuilder** class.
- It is shown here:
  String(StringBuilder strBuildObj)
- This constructs a **String** from the **StringBuilder** passed in strBuildObj.

String Length:

- The length of a string is the number of characters that it contains.
- To obtain this value, call the **length( )** method, shown here:

```
int length( )
```

- The following fragment prints “3”, since there are three characters in the string s:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Special String Operations

String Literals:

- For each string literal in your program, Java automatically constructs a **String** object.
- Thus, you can use a string literal to initialize a **String** object.

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);

String s2 = "abc"; // use string literal
```

- Because a **String** object is created for every string literal, you can use a string literal any place you can use a **String** object.

```
System.out.println("abc".length());
```

String Concatenation:

- In general, Java does not allow operators to be applied to **String** objects. The one exception to this rule is the **+** operator, which concatenates two strings, producing a **String** object as the result. This allows you to chain together a series of **+** operations. For example, the following fragment concatenates three strings:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

- This displays the string “He is 9 years old.”

String Concatenation:

```
// Using concatenation to prevent long lines.
class ConCat {
  public static void main(String args[]) {
    String longStr = "This could have been " +
      "a very long line that would have " +
      "wrapped around.  But string concatenation " +
      "prevents this.";

    System.out.println(longStr);
  }
}
```

String Concatenation with Other Data Types:

- You can concatenate strings with other types of data. For example, consider this slightly different version of the earlier example:

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

- The compiler will convert an operand to its string equivalent whenever the other operand of the **+** is an instance of **String**.

```
String s = "four: " + 2 + 2;
System.out.println(s);
```

    This fragment displays

```
four: 22
```

    rather than the

```
four: 4
```

- Operator precedence causes the concatenation of “four” with the string equivalent of 2 to take place first.
- `String s = "four: " + (2 + 2);` → Now **s** contains the string “four: 4”.

String Conversion and toString( ):

- When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by **String**. **valueOf( )** is overloaded for all the simple types and for type **Object**.
- For the simple types, **valueOf( )** returns a string that contains the human-readable equivalent of the value with which it is called.
- For objects, **valueOf( )** calls the **toString( )** method on the object.
- The **toString( )** method, because it is the means by which you can determine the string representation for objects of classes that you create.
- The **toString( )** method has this general form:

```
String toString( )
```

String Conversion and toString( ):

```
// Override toString() for Box class.
class Box {
  double width;
  double height;
  double depth;

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

  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);
    String s = "Box b: " + b; // concatenate Box object

    System.out.println(b); // convert Box to string
    System.out.println(s);
  }
}
```

The output of this program is shown here:

```
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
```

<u>Character Extraction</u>

charAt( ):

- To extract a single character from a **String**, you can refer directly to an individual character via 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 specify a location within the string. charAt( ) returns the character at the specified location.
- For example,

  ```
  char ch;
  ch = "abc".charAt(1);
  ```

  assigns the value “b” to ch.

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)
- Here, sourceStart specifies the index of the beginning of the substring, and sourceEndspecifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from sourceStart through sourceEnd–1. The array that will receive the characters is specified by *target*.
- The index within *target* at which the substring will be copied is passed in *targetStart.* Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

getChars( ):

```
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);
  }
}
```

Here is the output of this program:

```
demo
```

getBytes( ):

- There is an alternative to **getChars( )** that stores the characters in an array of bytes. This method is called **getBytes( )**, and it uses the default character-to-byte conversions provided by the platform.
- Here is its simplest form:

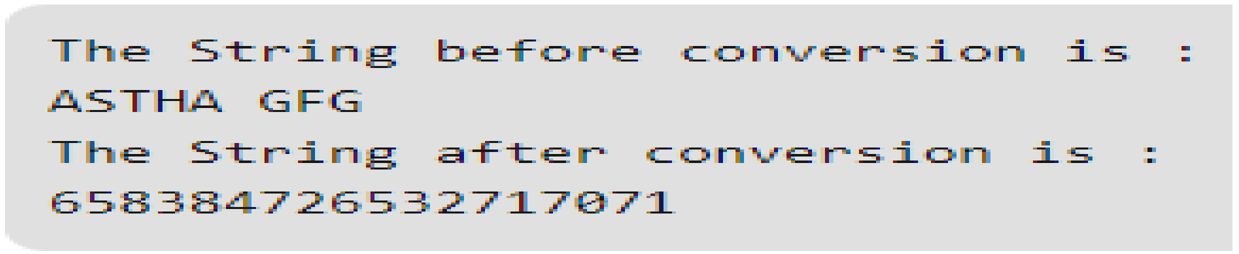
```
byte[ ] getBytes( )
```

- Other forms of **getBytes( )** are also available. **getBytes( )** is most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

<u>getBytes( ):</u>

```
// Java code to demonstrate the working of  getByte()
public class GetByte {
  public static void main(String args[])
    {        // Initializing String
        String gfg = "ASTHA GFG";
       // Displaying string values before  conversion
        System.out.println("The String before conversion is : ");
        System.out.println(gfg);
          // converting the string into byte using getBytes ( converts into ASCII values )
        byte[] b = gfg.getBytes();
          // Displaying converted string after conversion
        System.out.println("The String after conversion is : ");
        for (int i = 0; i < b.length; i++) {
            System.out.print(b[i]);
        }
  } }
```

Output:



```
The String before conversion is :
ASTHA GFG
The String after conversion is :
658384726532717071
```

toCharArray( ):

- If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( )**. It returns an array of characters for the entire string. It has this general form:

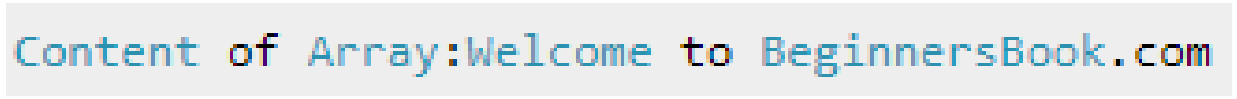
```
        char[ ] toCharArray( )
```

- This function is provided as a convenience, since it is possible to use **getChars( )** to achieve the same result.

toCharArray( ):

```
public class CharArrayExample{
   public static void main(String args[]){
       String str = new String("Welcome to BeginnersBook.com");
       char[] array= str.toCharArray();
       System.out.print("Content of Array:");
       for(char c: array){
           System.out.print(c);
       }
   }
}
```

Output:



```
Content of Array:Welcome to BeginnersBook.com
```

<u>String Comparison</u>

equals( ) and equalsIgnoreCase( ):

- To compare two strings for equality, use **equals( )**. It has this general form:
  boolean equals(Object *str*)
- Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case-sensitive.

- To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**. When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:
  boolean equalsIgnoreCase(String *str*)
- Here, *str* is the **String** object being compared with the invoking **String** object. It, too, returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

equals( ) and equalsIgnoreCase( ):

```
// Demonstrate equals() and equalsIgnoreCase().
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));
  }
}
```

The output from the program is shown here:

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

regionMatches( ):

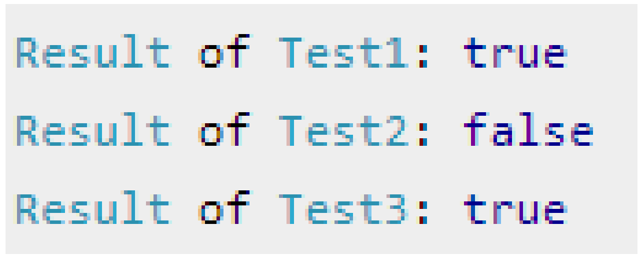
- The **regionMatches( )** method compares a specific region inside a string with another specific region in another string.
- There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

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

- For both versions, *startIndex* specifies the index at which the region begins within the invoking **String** object. The **String** being compared is specified by *str2.* The index at which the comparison will start within *str2* is specified by *str2StartIndex*. The length of the substring being compared is passed in *numChars.* In the second version, if *ignoreCase* is **true**, the case of the characters is ignored. Otherwise, case is significant.

<u>regionMatches( ):</u>

```
public class RegionMatchesExample{
  public static void main(String args[]){
     String str1 = new String("Hello, How are you");
     String str2 = new String("How");
     String str3 = new String("HOW");

     System.out.print("Result of Test1: " );
     System.out.println(str1.regionMatches(7, str2, 0, 3));
     System.out.print("Result of Test2: " );
     System.out.println(str1.regionMatches(7, str3, 0, 3));
     System.out.print("Result of Test3: " );
     System.out.println(str1.regionMatches(true, 7, str3, 0, 3));
  }
}
```



```
Result of Test1: true
Result of Test2: false
Result of Test3: true
```

<u>**startsWith( ) and endsWith( ):**</u>

- The **startsWith( )** method determines whether a given **String** begins with a specified string.
- Conversely, **endsWith( )** determines whether the **String** in question ends with a specified string.
- They have the following general forms:

  boolean startsWith(String *str*)
  boolean endsWith(String *str*)
- Here, *str* is the **String** being tested. If the string matches, **true** is returned. Otherwise, **false** is returned.
- For example,

  ```
  "Foobar".endsWith("bar")
  ```
  and
  ```
  "Foobar".startsWith("Foo")
  ```

  are both **true**.

startsWith( ) and endsWith( ):

- A second form of **startsWith( )**, shown here, lets you specify a starting point:

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

- Here, *startIndex* specifies the index into the invoking string at which point the search will begin.
- For example,

```
"Foobar".startsWith("bar", 3)
```

returns **true**.

equals( ) Versus ==

- It is important to understand that the **equals( )** method and the **==** operator perform two different operations.
- As just explained, the **equals( )** method compares the characters inside a **String** object.
- The **==** operator compares two object references to see whether they refer to the same instance.
- The following program shows how two different **String** objects can contain the same characters, but references to these objects will not compare as equal:

equals( ) Versus ==

```
// equals() vs ==
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:

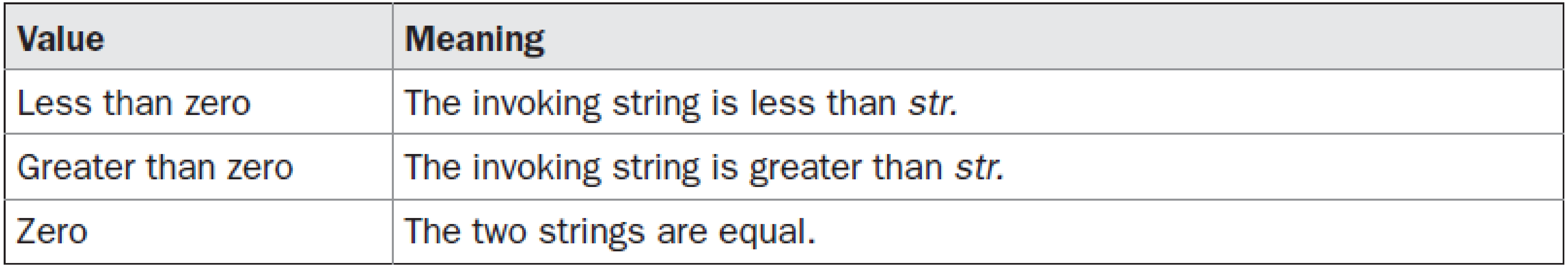
```
Hello equals Hello -> true
Hello == Hello -> false
```

compareTo( ):

- Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is *less than, equal to,* or *greater than* the next. A string is less than another if it comes before the other in dictionary order.
- A string is greater than another if it comes after the other in dictionary order.
- The **String** method **compareTo( )** serves this purpose. It has this general form:

  int compareTo(String *str*)

  Here, *str* is the **String** being compared with the invoking **String**. The result of the comparison is returned and is interpreted, as shown here:



Value	Meaning
Less than zero	The invoking string is less than *str.*
Greater than zero	The invoking string is greater than *str.*
Zero	The two strings are equal.

compareTo( ):

```
// A bubble sort for Strings.
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]);
    }
  }
}
```

The output of this program is the list of words:

```
Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to
```

<u>**compareTo( ):**</u>

- If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase( )**, as shown here:

  int compareToIgnoreCase(String *str*)
- This method returns the same results as **compareTo( )**, except that case differences are ignored.
- You might want to try substituting it into the previous program. After doing so, “Now” will no longer be first.

Searching Strings

Searching Strings:

- The **String** class provides two methods that allow you to search a string for a specified character or substring:
  - **indexOf( )** Searches for the first occurrence of a character or substring.
  - **lastIndexOf( )** Searches for the last occurrence of a character or substring.
- These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or –1 on failure.

Searching Strings:

To search for the first occurrence of a character, use

int indexOf(int *ch*)

To search for the last occurrence of a character, use

int lastIndexOf(int *ch*)

Here, *ch* is the character being sought.

To search for the first or last occurrence of a substring, use

int indexOf(String *str*)
int lastIndexOf(String *str*)

Here, *str* specifies the substring.

You can specify a starting point for the search using these forms:

int indexOf(int *ch*, int *startIndex*)
int lastIndexOf(int *ch*, int *startIndex*)

int indexOf(String *str*, int *startIndex*)
int lastIndexOf(String *str*, int *startIndex*)

Here, *startIndex* specifies the index at which point the search begins. For **indexOf( )**, the search runs from *startIndex* to the end of the string. For **lastIndexOf( )**, the search runs from *startIndex* to zero.

Searching Strings:

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
  public static void main(String args[]) {
    String s = "Now is the time for all good men " +
               "to come to the aid of their country.";

    System.out.println(s);
    System.out.println("indexOf(t) = " +
                       s.indexOf('t'));
    System.out.println("lastIndexOf(t) = " +
                       s.lastIndexOf('t'));
    System.out.println("indexOf(the) = " +
                       s.indexOf("the"));
    System.out.println("lastIndexOf(the) = " +
                       s.lastIndexOf("the"));
    System.out.println("indexOf(t, 10) = " +
                       s.indexOf('t', 10));
    System.out.println("lastIndexOf(t, 60) = " +
                       s.lastIndexOf('t', 60));
    System.out.println("indexOf(the, 10) = " +
                       s.indexOf("the", 10));
    System.out.println("lastIndexOf(the, 60) = " +
                       s.lastIndexOf("the", 60));
  }
}
```

Here is the output of this program:

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

Modifying a String

- 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( ):

- You can extract a substring using **substring( ).**
- It has two forms. The first is

  String substring(int *startIndex)*

  Here, *startIndex specifies the index at which the substring will begin.*
- *This form returns a copy* of the substring that begins at *startIndex and runs to the end of the invoking string.*

- The second form of **substring( )** allows you to specify both the beginning and ending index of the substring:

  String substring(int *startIndex, int endIndex)*

  Here, *startIndex specifies the beginning index, and endIndex specifies the stopping point.*
- The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

substring( ):

```
// Substring replacement.
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 { // replace all matching substrings
      System.out.println(org);
      i = org.indexOf(search);
      if(i != -1) {
        result = org.substring(0, i);

        result = result + sub;
        result = result + org.substring(i + search.length());
        org = result;
      }
    } while(i != -1);

  }
}
```

The output from this program is shown here:

```
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.
```

concat( ):

- You can concatenate two strings using **concat( ),** shown here:

```
String concat(String str)
```

- This method creates a new object that contains the invoking string with the contents of *str appended to the end.* ***concat( )*** *performs the same function as +.*
- *For example,*

```
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string "onetwo" into **s2.** It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

replace( ):

- The **replace( )** method has two forms.
- The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

  String replace(char *original, char replacement)*

  Here, *original specifies the character to be replaced by the character specified by replacement.*
- The resulting string is returned. For example,

  String s = "Hello".replace('l', 'w');

  puts the string "Hewwo" into **s.**

- The second form of **replace( )** replaces one character sequence with another. It has this general form:

  String replace(CharSequence *original, CharSequence replacement)*
- This form was added by J2SE 5.

Java String replace(CharSequence target, CharSequence replacement) method example

```java
public class ReplaceExample2{

public static void main(String args[]){

String s1="my name is khan my name is java";

String replaceString=s1.replace("is","was");//replaces all occurrences of "is" to "was"

System.out.println(replaceString);

}}
```

Test it Now

```
my name was khan my name was java
```

trim( ):

- The **trim( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed.
- It has this general form:

```
String trim( )
```

- Here is an example:

```
String s = " Hello World ".trim();
```

  This puts the string "Hello World" into **s.**

- The **trim( )** method is quite useful when you process user commands.
- For example, the following program prompts the user for the name of a state and then displays that state's capital.
- It uses **trim( )** to remove any leading or trailing whitespace that may have inadvertently been entered by the user.

trim( ):

```java
// Using trim() to process commands.
import java.io.*;

class UseTrim {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    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 converts data from its internal format into a human-readable form.
- It is a static method that is overloaded within **String** for all of Java's built-in types so that each type can be converted properly into a string.
- **valueOf( )** is also overloaded for type Object, so an object of any class type you create can also be used as an argument. (Recall that Object is a superclass for all classes.)
- Here are a few of its forms:

```
static String valueOf(double num)
static String valueOf(long num)
static String valueOf(Object ob)
static String valueOf(char chars[ ])
```

- valueOf( ) is called when a string representation of some other type of data is needed—for example, during concatenation operations.

Java String valueOf() method example

```
public class StringValueOfExample{
public static void main(String args[]){
int value=30;
String s1=String.valueOf(value);
System.out.println(s1+10);//concatenating string with 10
}}
```

Output:

```
3010
```

Data Conversion Using valueOf( ):

- You can call this method directly with any data type and get a reasonable String representation.
- All of the simple types are converted to their common String representation.
- Any object that you pass to valueOf( ) will return the result of a call to the object's toString() method.
- In fact, you could just call toString( ) directly and get the same result.
- For most arrays, **valueOf( )** returns a rather cryptic string, which indicates that it is an array of some type.
- For **arrays of char**, however, a **String** object is created that contains the characters in the char array. There is a special version of valueOf( ) that allows you to specify a subset of a char array. It has this general form:

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

  Here, chars is the array that holds the characters, startIndex is the index into the array of characters at which the desired substring begins, and numChars specifies the length of the substring.

```java
import java.io.*;
public class Test {

   public static void main(String args[]) {
      double d = 102939939.939;
      boolean b = true;
      long l = 1232874;
      char[] arr = {'a', 'b', 'c', 'd', 'e', 'f','g' };

      System.out.println("Return Value : " + String.valueOf(d) );
      System.out.println("Return Value : " + String.valueOf(b) );
      System.out.println("Return Value : " + String.valueOf(l) );
      System.out.println("Return Value : " + String.valueOf(arr) );
   }
}
```

This will produce the following result –

Output

```
Return Value : 1.02939939939E8

Return Value : true

Return Value : 1232874

Return Value : abcdefg
```

<u>Changing the Case of Characters Within a String</u>

Changing the Case of Characters Within a String:

- The method **toLowerCase( )** converts all the characters in a string from uppercase to lowercase.
- The **toUpperCase( )** method converts all the characters in a string from lowercase to uppercase.
- Nonalphabetical characters, such as digits, are unaffected.
- Here are the general forms of these methods:

```
String toLowerCase( )
String toUpperCase( )
```

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

Changing the Case of Characters Within a String:

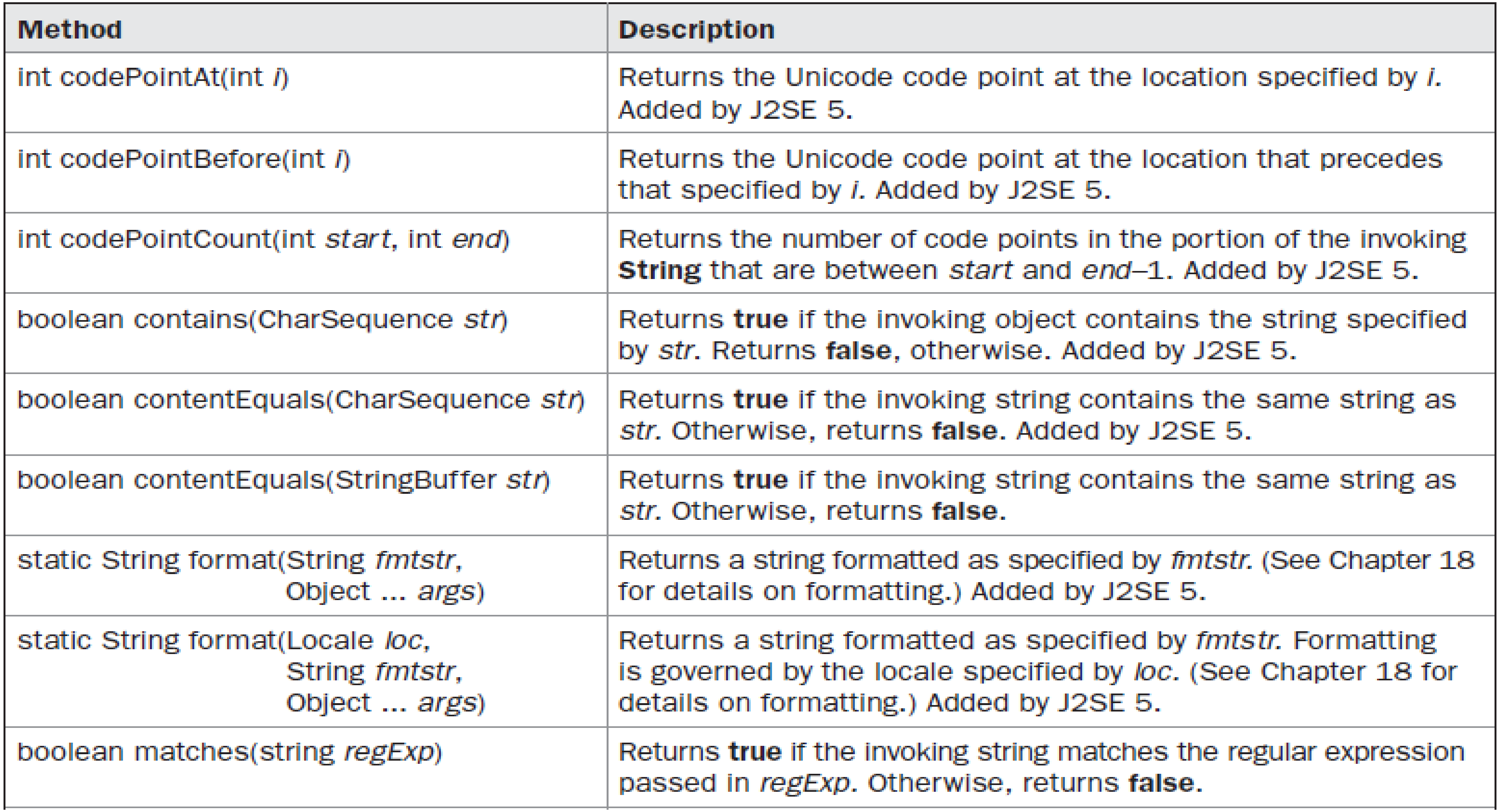
```
// Demonstrate toUpperCase() and toLowerCase().

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);
    System.out.println("Lowercase: " + lower);
  }
}
```

The output produced by the program is shown here:

```
Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.
```

Additional String Methods

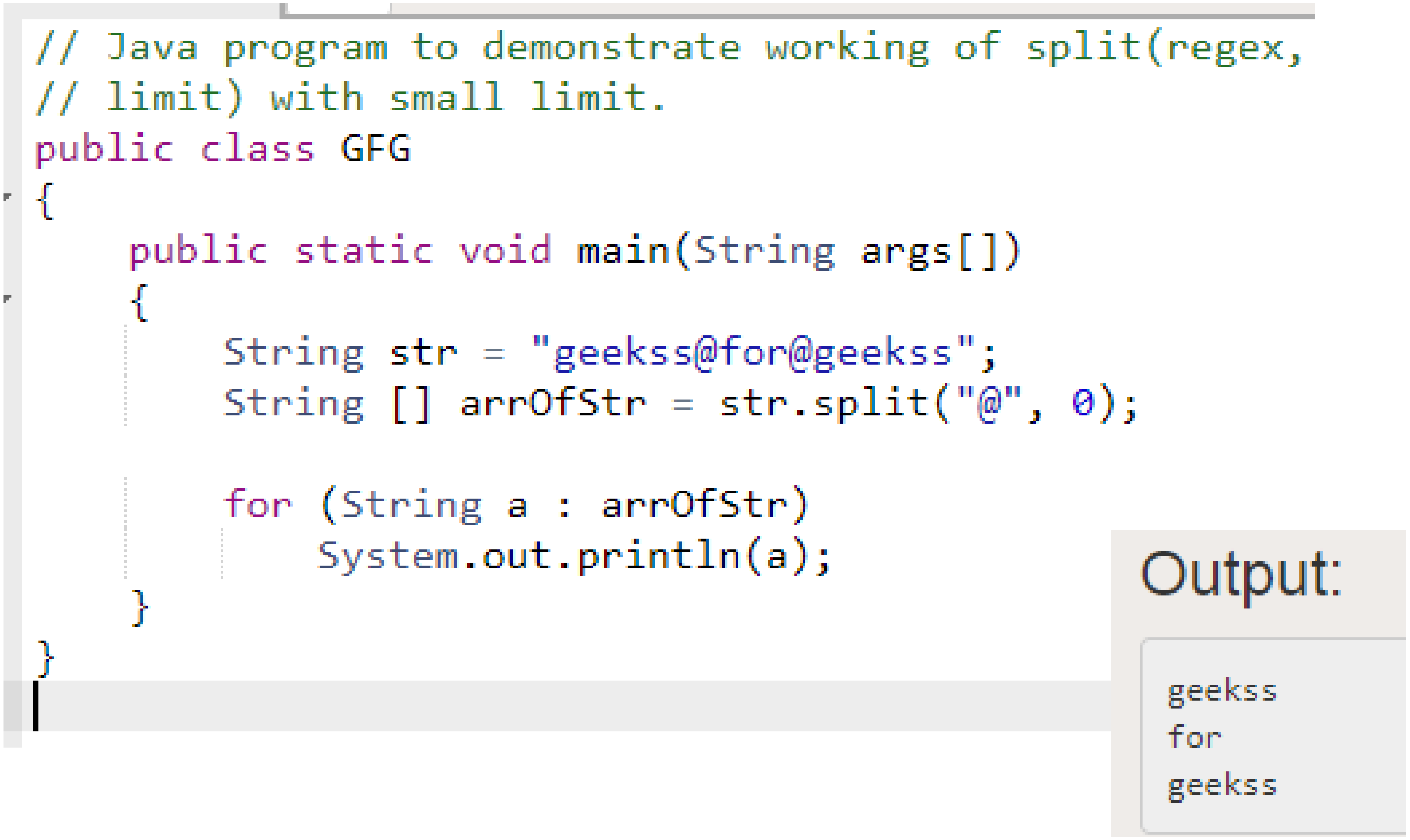
Additional String Methods:



Method	**Description**
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*. Added by J2SE 5.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*. Added by J2SE 5.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1. Added by J2SE 5.
boolean contains(CharSequence *str*)	Returns **true** if the invoking object contains the string specified by *str*. Returns **false**, otherwise. Added by J2SE 5.
boolean contentEquals(CharSequence *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**. Added by J2SE 5.
boolean contentEquals(StringBuffer *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
static String format(String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. (See Chapter 18 for details on formatting.) Added by J2SE 5.
static String format(Locale *loc*, String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. Formatting is governed by the locale specified by *loc*. (See Chapter 18 for details on formatting.) Added by J2SE 5.
boolean matches(string *regExp*)	Returns **true** if the invoking string matches the regular expression passed in *regExp*. Otherwise, returns **false**.

Additional String Methods:

int offsetByCodePoints(int *start*, int *num*)	Returns the index with the invoking string that is *num* code points beyond the starting index specified by *start*. Added by J2SE 5.
String replaceFirst(String *regExp*, String *newStr*)	Returns a string in which the first substring that matches the regular expression specified by *regExp* is replaced by *newStr*.
String replaceAll(String *regExp*, String *newStr*)	Returns a string in which all substrings that match the regular expression specified by *regExp* are replaced by *newStr*.
String[ ] split(String *regExp*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*.
String[ ] split(String *regExp*, int *max*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*. The number of pieces is specified by *max*. If *max* is negative, then the invoking string is fully decomposed. Otherwise, if *max* contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If *max* is zero, the invoking string is fully decomposed.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is now implemented by **String**.



```java
// Java program to demonstrate working of split(regex,
// limit) with small limit.
public class GFG
{
    public static void main(String args[])
    {
        String str = "geekss@for@geekss";
        String [] arrOfStr = str.split("@", 0);

        for (String a : arrOfStr)
            System.out.println(a);
    }
}
```

Output:

```
geekss
for
geekss
```

```java
public class SplitExample{
public static void main(String args[]){
String s1="java string split method by javatpoint";
String[] words=s1.split("\\s");//splits the string based on string
//using java foreach loop to print elements of string array
for(String w:words){
System.out.println(w);
}
}}
```

```
java
string
split
method
by
javatpoint
```

StringBuffer

StringBuffer:

- As you know, **String** represents fixed-length, immutable character sequences.
- In contrast, **StringBuffer** represents ***growable and writeable*** character sequences.
- StringBuffer may have characters and substrings inserted in the middle or appended to the end.
- StringBuffer will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.
- Java uses both classes heavily, but many programmers deal only with String and let Java manipulate StringBuffers behind the scenes by using the overloaded + operator.

StringBuffer Constructors:

- **StringBuffer** defines these **four** constructors:

  StringBuffer( )
  StringBuffer(int size)
  StringBuffer(String str)
  StringBuffer(CharSequence chars)

- The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.
- The second version accepts an integer argument that explicitly sets the size of the buffer.
- The third version accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

StringBuffer Constructors:

- StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time.
- Also, frequent reallocations can fragment memory.
- By allocating room for a few extra characters, StringBuffer reduces the number of reallocations that take place.
- The fourth constructor creates an object that contains the character sequence contained in chars.

<u>length( ) and capacity( ):</u>

- The current length of a StringBuffer can be found via the length( ) method, while the total allocated capacity can be found through the capacity( ) method.
- They have the following general forms:

```
int length( )
int capacity( )
```

- Here is an example:

length( ) and 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());
  }
}
```

Here is the output of this program, which shows how **StringBuffer** reserves extra space for additional manipulations:

```
buffer = Hello
length = 5
capacity = 21
```

ensureCapacity( ):

- If you want to **preallocate** room for a certain number of characters after a **StringBuffer** has been constructed, you can use ensureCapacity( ) to set the size of the buffer.
- This is useful if you know in advance that you will be appending a large number of small strings to a StringBuffer.
- ensureCapacity( ) has this general form:

```
void ensureCapacity(int capacity)
```

Here, capacity specifies the size of the buffer.

```
package com.tutorialspoint;

import java.lang.*;

public class StringBufferDemo {

   public static void main(String[] args) {

      StringBuffer buff1 = new StringBuffer("tuts point");
      System.out.println("buffer1 = " + buff1);

      // returns the current capacity of the string buffer 1
      System.out.println("Old Capacity = " + buff1.capacity());

      /* increases the capacity, as needed, to the specified amount in the
         given string buffer object */

      // returns twice the capacity plus 2
      buff1.ensureCapacity(28);
      System.out.println("New Capacity = " + buff1.capacity());

      StringBuffer buff2 = new StringBuffer("compile online");
      System.out.println("buffer2 = " + buff2);

      // returns the current capacity of string buffer 2
      System.out.println("Old Capacity = " + buff2.capacity());

      /* returns the old capacity as the capacity ensured is less than
         the old capacity */
      buff2.ensureCapacity(29);
      System.out.println("New Capacity = " + buff2.capacity());
   }
}
```

result –

```
buffer1 = tuts point
Old Capacity = 26
New Capacity = 54
buffer2 = compile online
Old Capacity = 30
New Capacity = 30
```

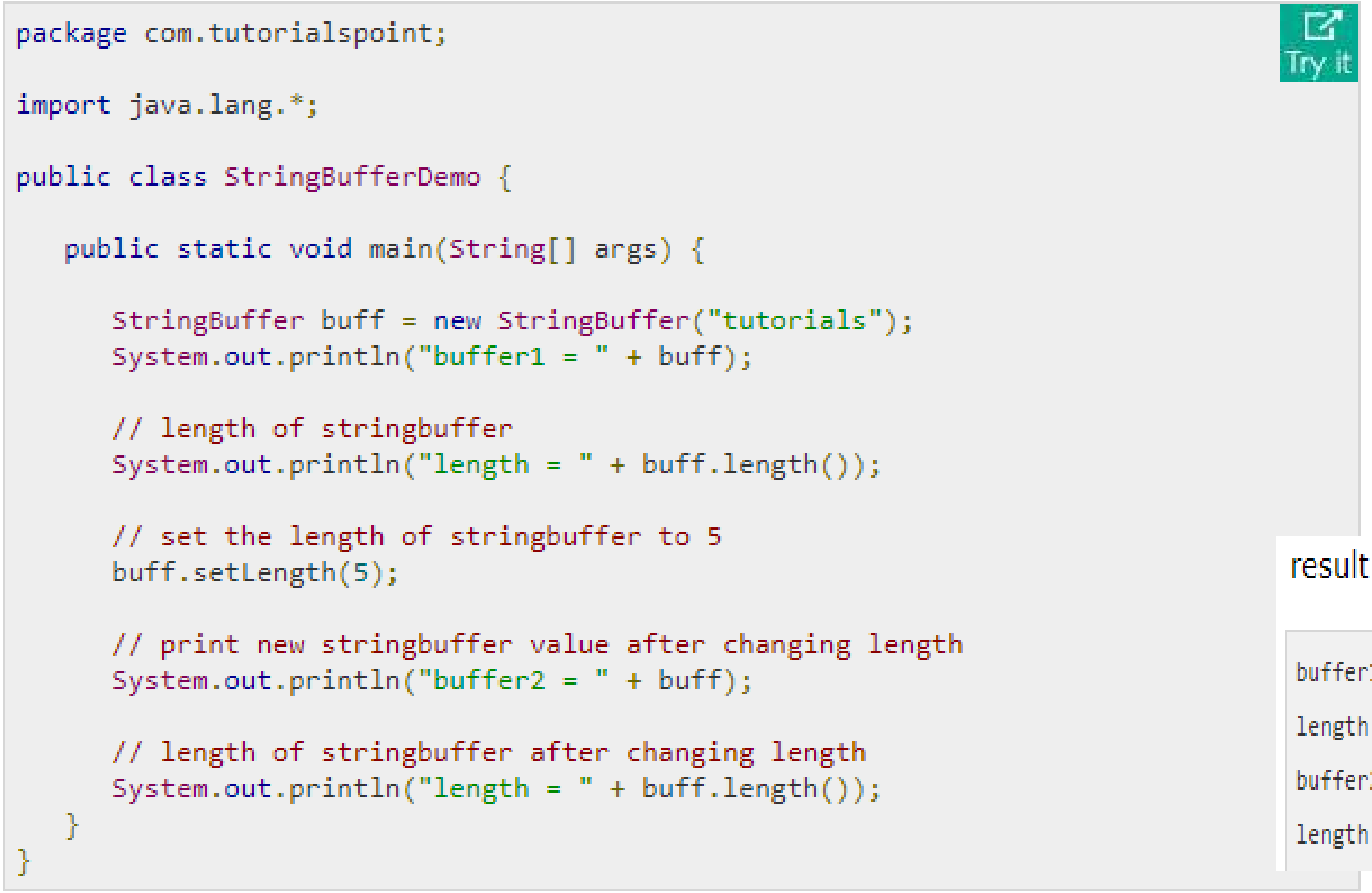
<u>**setLength( ):**</u>

- To set the length of the buffer within a **StringBuffer object, use setLength( ).**
- Its general form is shown here:

  void setLength(int *len)*

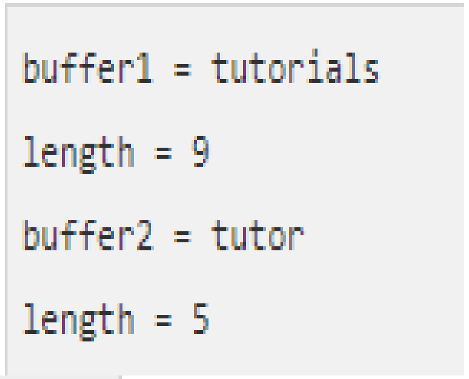
  Here, *len specifies the length of the buffer. This value must be nonnegative.*

- When you increase the size of the buffer, null characters are added to the end of the existing buffer.
- If you call setLength( ) with a value less than the current value returned by length( ), then the characters stored beyond the new length will be lost.
- The setCharAtDemo sample program in the following section uses setLength( ) to shorten a StringBuffer.



```
package com.tutorialspoint;

import java.lang.*;

public class StringBufferDemo {

   public static void main(String[] args) {

      StringBuffer buff = new StringBuffer("tutorials");
      System.out.println("buffer1 = " + buff);

      // length of stringbuffer
      System.out.println("length = " + buff.length());

      // set the length of stringbuffer to 5
      buff.setLength(5);

      // print new stringbuffer value after changing length
      System.out.println("buffer2 = " + buff);

      // length of stringbuffer after changing length
      System.out.println("length = " + buff.length());
   }
}
```

result –



```
buffer1 = tutorials

length = 9

buffer2 = tutor

length = 5
```

charAt( ) and setCharAt( ):

- The value of a single character can be obtained from a StringBuffer via the charAt( ) method.
- You can set the value of a character within a StringBuffer using setCharAt( ).
- Their general forms are shown here:

```
char charAt(int where)
void setCharAt(int where, char ch)
```

- For charAt( ), where specifies the index of the character being obtained.
- For setCharAt( ), where specifies the index of the character being set, and ch specifies the new value of that character.
- For both methods, where must be nonnegative and must not specify a location beyond the end of the buffer.

charAt( ) and setCharAt( ):

```
// 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));
  }
}
```

Here is the output generated by this program:

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

getChars( ):

- To copy a substring of a StringBuffer into an array, use the getChars( ) method.
- It has this general form:

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

  Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from sourceStart through sourceEnd–1. The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart.
- Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

append( ):

- The append( ) method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object. It has several overloaded versions.
- Here are a few of its forms:

```
StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj)
```

- String.valueOf( ) is called for each parameter to obtain its string representation.
- The result is appended to the current StringBuffer object. The buffer itself is returned by each version of append( ).

append( ):

```
// Demonstrate append().
class appendDemo {
  public static void main(String args[]) {
    String s;
    int a = 42;
    StringBuffer sb = new StringBuffer(40);

    s = sb.append("a = ").append(a).append("!").toString();
    System.out.println(s);
  }
}
```

The output of this example is shown here:

```
a = 42!
```

append( ):

- The append( ) method is most often called when the + operator is used on String objects.
- Java automatically changes modifications to a String instance into similar operations on a StringBuffer instance.
- Thus, a concatenation invokes append( ) on a StringBuffer object.
- After the concatenation has been performed, the compiler inserts a call to toString( ) to turn the modifiable StringBuffer back into a constant String.
- All of this may seem unreasonably complicated.
- Why not just have one string class and have it behave more or less like StringBuffer?
    - The answer is performance. There are many optimizations that the Java run time can make knowing that String objects are immutable.
    - Thankfully, Java hides most of the complexity of conversion between Strings and StringBuffers.
- Actually, many programmers will never feel the need to use StringBuffer directly and will be able to express most operations in terms of the + operator on String variables.

insert( ):

- The insert( ) method inserts one string into another. It is overloaded to accept values of all the simple types, plus Strings, Objects, and CharSequences.
- Like append( ), it calls String.valueOf( ) to obtain the string representation of the value it is called with.
- This string is then inserted into the invoking StringBuffer object.
- These are a few of its forms:

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

- Here, index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

insert( ):

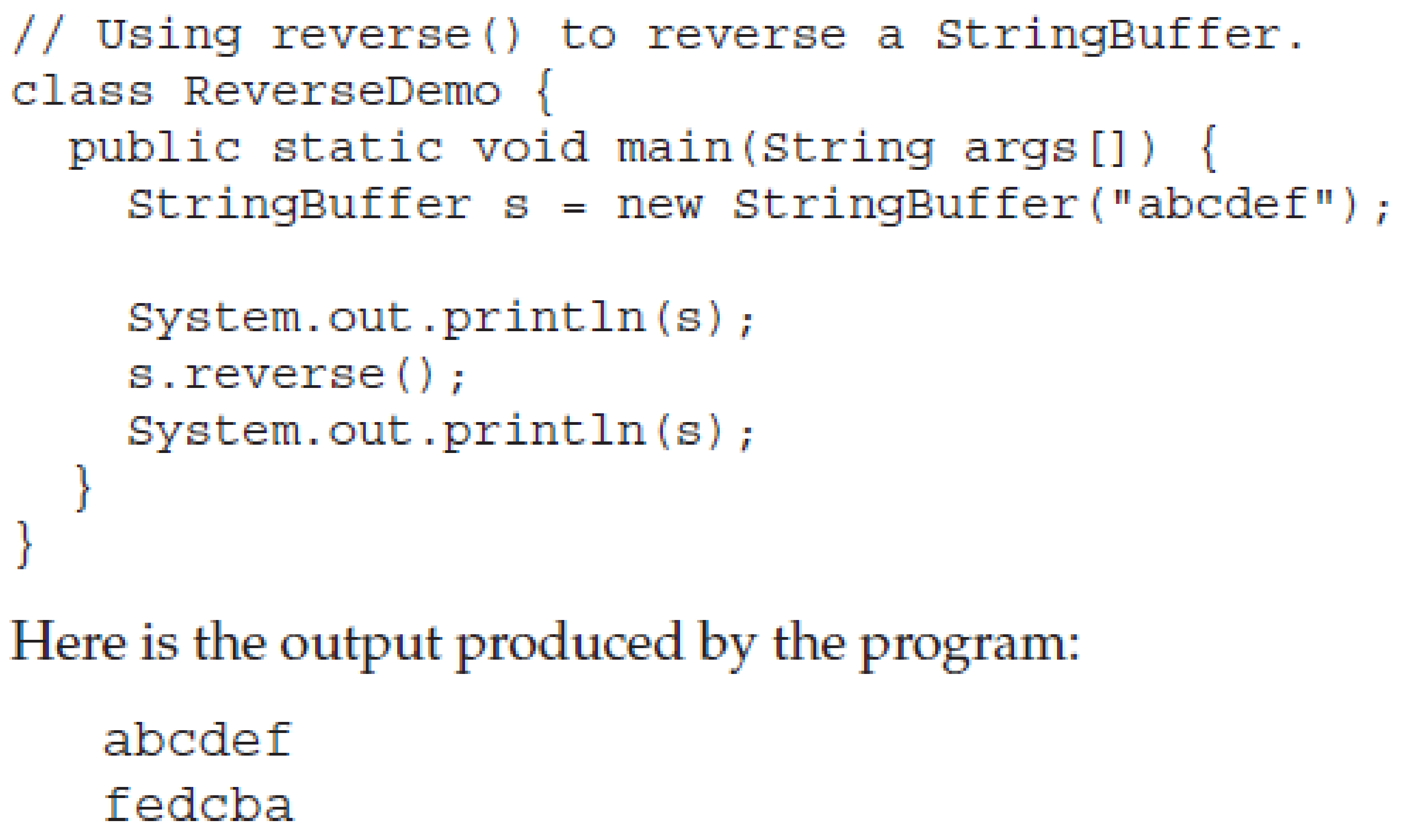
```
// Demonstrate insert().
class insertDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("I Java!");

    sb.insert(2, "like ");
    System.out.println(sb);
  }
}
```

The output of this example is shown here:

```
I like Java!
```

reverse( ):

- You can reverse the characters within a StringBuffer object using reverse( ), shown here:
  StringBuffer reverse( )
- This method returns the reversed object on which it was called.



```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
  public static void main(String args[]) {
    StringBuffer s = new StringBuffer("abcdef");

    System.out.println(s);
    s.reverse();
    System.out.println(s);
  }
}
```

Here is the output produced by the program:

```
   abcdef
   fedcba
```

delete( ) and deleteCharAt( ):

- You can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ).
- These methods are shown here:

  StringBuffer delete(int startIndex, int endIndex)
  StringBuffer deleteCharAt(int loc)
- The delete( ) method deletes a sequence of characters from the invoking object.
- Here, startIndex specifies the index of the first character to remove, and endIndex specifies an index one past the last character to remove.
- Thus, the substring deleted runs from startIndex to endIndex–1.
- The resulting StringBuffer object is returned.

- The deleteCharAt( ) method deletes the character at the index specified by loc. It returns the resulting StringBuffer object.

delete( ) and deleteCharAt( ):

```
// Demonstrate 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);
  }
}
```

The following output is produced:

```
After delete: This a test.
After deleteCharAt: his a test.
```

replace( ):

- You can replace one set of characters with another set inside a StringBuffer object by calling replace( ).
- Its signature is shown here:

  StringBuffer replace(int startIndex, int endIndex, String str)
- The substring being replaced is specified by the indexes startIndex and endIndex. Thus, the substring at startIndex through endIndex–1 is replaced.
- The replacement string is passed in str.
- The resulting StringBuffer object is returned.

replace( ):

```
// Demonstrate 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);
     }
}
```

Here is the output:

```
After replace: This was a test.
```

substring( ):

- You can obtain a portion of a StringBuffer by calling substring( ).
- It has the following two forms:

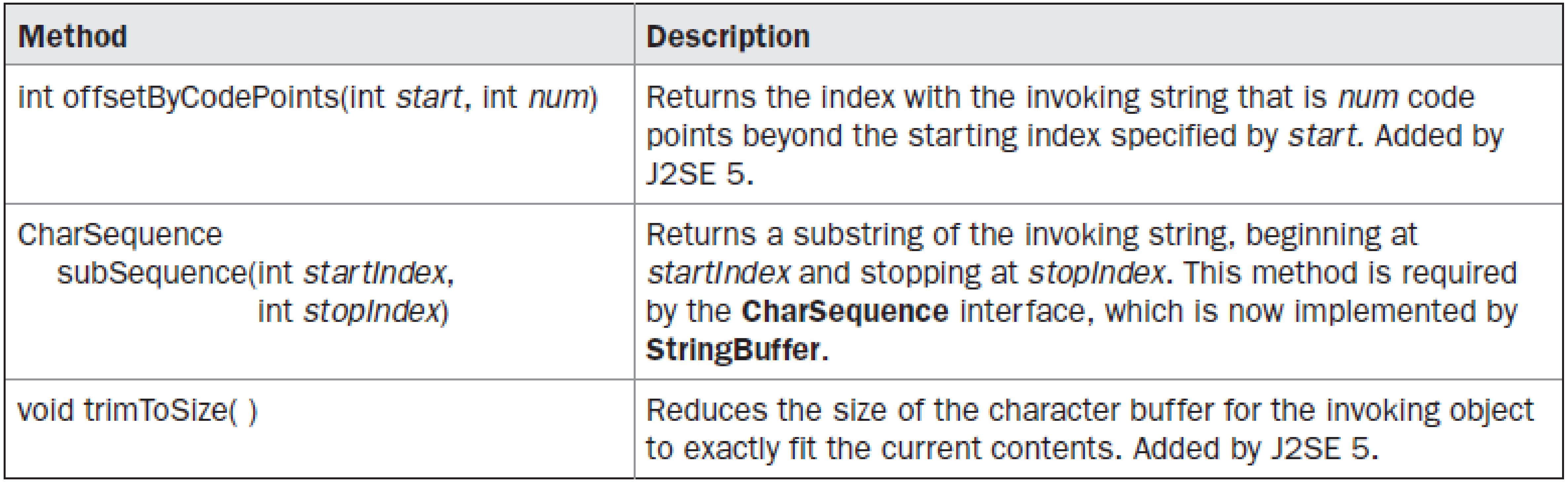
  String substring(int startIndex)

  String substring(int startIndex, int endIndex)
- The first form returns the substring that starts at startIndex and runs to the end of the invoking StringBuffer object.
- The second form returns the substring that starts at startIndex and runs through endIndex–1.
- These methods work just like those defined for String that were described earlier.

Additional StringBuffer Methods

Method	Description
StringBuffer appendCodePoint(int *ch*)	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned. Added by J2SE 5.
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*. Added by J2SE 5.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*. Added by J2SE 5.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1. Added by J2SE 5.
int indexOf(String *str*)	Searches the invoking **StringBuffer** for the first occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int indexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the first occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*)	Searches the invoking **StringBuffer** for the last occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the last occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.

Additional StringBuffer Methods



Method	**Description**
int offsetByCodePoints(int *start*, int *num*)	Returns the index with the invoking string that is *num* code points beyond the starting index specified by *start*. Added by J2SE 5.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is now implemented by **StringBuffer**.
void trimToSize( )	Reduces the size of the character buffer for the invoking object to exactly fit the current contents. Added by J2SE 5.

Additional StringBuffer Methods

```
class IndexOfDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("one two one");
    int i;

    i = sb.indexOf("one");
    System.out.println("First index: " + i);

    i = sb.lastIndexOf("one");
    System.out.println("Last index: " + i);
  }
}
```

The output is shown here:

```
First index: 0
Last index: 8
```

StringBuilder

- J2SE 5 adds a new string class to Java’s already powerful string handling capabilities.
- This new class is called StringBuilder.
- It is ***identical to StringBuffer except for one important difference***: it is not synchronized, which means that it is not thread-safe.
- The advantage of StringBuilder is ***faster performance***.
- However, in cases in which you are using multithreading, you must use StringBuffer rather than StringBuilder.

```
class SSB {     public static void main(String []args) {
          String s = new String("Hello");
          String s1= new String(s);
          StringBuffer b = new StringBuffer("Hello");
          StringBuffer b1 = new StringBuffer(b);
          if(s.equals(s1))
                    System.out.println("Same..");
          else
                    System.out.println("Not same..");
          if(b.equals(b1))
                    System.out.println("Same..");
          else
                    System.out.println("Not same..");
  } }
```

Output ?

Refereces

- https://www.javatpoint.com/
- http://www.geeksforgeeks.org
- https://www.tutorialspoint.com

Thank You