String handling in java

Strings

- In Java a string is a sequence of characters.
- Other languages that implement strings as character arrays, Java implements strings as objects of type **String**.
- Strings are Immutable. String object that is created cannot be changed.
- However, a variable declared as a String reference can be changed to point at some other String object at any time.
- Use the class called StringBuffer to perform changes in original strings.
- String, StringBuilder and StringBuffer classes are declared final and there cannot be subclasses of these classes.
- The String, StringBuffer, and StringBuilder classes are defined in java.lang.

Creating Strings

- The default constructor creates an empty string.
  - String s = new String();
- To create strings that have initial values.
  - String(char chars[ ])
- **Examples:**

```
String str = "abc";        is equivalent to:
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

- Construct a string object by passing another string object.
  - String(String strObj)
  - Example: String str2 = new String(str);

Creating Strings

- To 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.
- **Examples:**

```
char data[] = {'a', 'b', 'c',’d’,’e’};
String str = new String(2, 1,data);
```

Creating Strings

- String class provides constructors that initialize a string when given a byte array.
  - String(byte chrs[ ])
  - String(byte chrs[ ], int startIndex, int numChars)
    - Here, chrs specifies the array of bytes. The second form allows you to specify a subrange.
  - **Examples:**

```
// 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
```

String METHODS

- int length( )
  - The length() method returns the length of the string.

    Eg: System.out.println("Hello".length()); // prints 5

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

- The + operator is used to concatenate two or more strings.

  Eg: String name = "Harry"

  String str = "Name : " + name+ ".";
- Java compiler converts an operand to a String whenever the other operand of the + is a String object.

String Concatenation with Other Data Types

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

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

This fragment displays

four: 22

rather than the

four: 4

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

Now s contains the string "four: 4".

String Conversion and toString( )

- To determine the string representation for objects of classes that is created.
- Classes that is created has to override toString( ) and provide your own string representations.
- The toString( ) method has this general form:
  - **String toString( )**
  - can be used in print( ) and println( ) statements and in concatenation expressions.

```
// 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
```

Character Extraction

Character Extraction

- The String class provides a number of ways in which characters can be extracted from a String object.
- The characters that comprise a string within a String object cannot be indexed as if they were a character array.
- Many of the String methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

Character Extraction

- public char **charAt**(int INDEX)
  - Returns the character at the specified index.
  - INDEX is the index of the character that is to be to obtained.
  - An index ranges from 0 to length() - 1.

```
char ch;
ch = "XYZ".charAt(1); // ch = "Y"
```

- Method `getChars`

  Get entire set of characters in `String`

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

  s1.getChars(start, first after, charArray, start);

  compareTo( )

  int compareTo(String str)

  Here, str is the String being compared with the invoking String.

Character Extraction

```
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
```

Character Extraction

```
// 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
```

The word "Now" came out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.

To ignore case differences when comparing two strings, use compareToIgnoreCase( ),
int compareToIgnoreCase(String str)

Character Extraction

- 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( )**
- getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters.

Character Extraction

- toCharArray( )
- 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.

String Comparison

- **equals()** - Compares the invoking string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as the invoking object.

  public boolean **equals**(Object anObject)

- **equalsIgnoreCase()-** Compares this String to another String, ignoring case considerations.
  - When it compares two strings, it considers A-Z to be the same as a-z.
  - Two strings are considered equal ignoring case if they are of the same length, and corresponding characters in the two strings are equal ignoring case.

  public boolean **equalsIgnoreCase**(String anotherString)

```
// 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
```

String Comparison :equals() vs ==

- 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.

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

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

String Comparison

- **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.

String Comparison

- **startsWith()** – Tests if this string starts with the specified prefix.

  public boolean **startsWith**(String prefix)

  “Figure”.startsWith(“Fig”); // true

- **endsWith()** - Tests if this string ends with the specified suffix.

  public boolean **endsWith**(String suffix)

  “Figure”.endsWith(“re”); // true

  - boolean startsWith(String str, int startIndex)
    - Example : "Foobar".startsWith("bar", 3) => returns true.

String Comparison

- **compareTo()** - Compares two strings.
  - 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 result is a negative integer if this String object lexicographically precedes the argument string.
  - The result is a positive integer if this String object lexicographically follows the argument string.
  - The result is zero if the strings are equal.
  - compareTo returns 0 exactly when the equals(Object) method would retu true.

public int **compareTo**(String anotherString) public int

compareToIgnoreCase(String str)

Searching Strings

- **indexOf** – Searches for the first occurrence of a character or substring. Returns -1 if the character does not occur.

- public int **indexOf**(String str) - Returns the index within this string of the first occurrence of the specified substring.

```
String str = "How was your day today?";
str.indexof('t');
```

- **lastIndexOf()** –Searches for the last occurrence of a character or substring.
- The methods are similar to indexOf().

- 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.

```
// 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

- **substring()** - Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

public String **substring**(int beginIndex)

Eg: "unhappy".substring(2)

returns "happy"

public String **substring**(int beginIndex, int endIndex)

Eg: "smiles".substring(1, 5)

returns "mile“

```
// 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.
```

String METHODS

Method call	Meaning
S2=s1.toLowerCase()	Convert string s1 to lowercase
S2=s1.toUpperCase()	Convert string s1 to uppercase
S2=s1.repalce('x', 'y')	Replace occurrence x with y
S2=s1.trim()	Remove whitespaces at the beginning and end of the string s1
S1.equals(s2)	If s1 equals to s2 return true
S1.equalsIgnoreCase(s2)	If s1==s2 then return true with irrespective of case of charecters
S1.length()	Give length of s1
S1.CharAt(n)	Give nth character of s1 string
S1.compareTo(s2)	If s1<s2 –ve no If s1>s2 +ve no If s1==s2 then 0
S1.concat(s2)	Concatenate s1 and s2
S1.substring(n)	Give substring staring from nth character

String Operations

String Operations

concat() - Concatenates the specified string to the end of this string.

If the length of the argument string is 0, then this String object is returned.

Otherwise, a new String object is created, containing the invoking string with the contents of the str appended to it.

public String concat(String str)

"to".concat("get").concat("her")

returns "together"

String Operations

- replace()- Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

public String **replace**(char oldChar, char newChar)

“iam aq iqdiaq ” . replace(‘q', ‘n')

returns “I am an indian"

String Operations

- **trim()** - Returns a copy of the string, with leading and trailing whitespace omitted.

  public String trim()

  String s = “ Hi Mom! “

  s.trim();

  S = “Hi Mom!”

- **valueOf()** – Returns the string representation of the char array argument.

  public static String **valueOf**(char[] data)

String Operations

- **toLowerCase():** Converts all of the characters in a String to lower case.
- **toUpperCase():** Converts all of the characters in this String to upper case.

public String **toLowerCase**()

public String **toUpperCase**()

Eg: “HELLO THERE”.toLowerCase();

“hello there”.toUpperCase();

Several are summarized in the following table:

Method	Description
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*.
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.
boolean contains(CharSequence *str*)	Returns **true** if the invoking object contains the string specified by *str*. Returns **false** otherwise.
boolean contentEquals(CharSequence *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
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 19 for details on formatting.)
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 19 for details on formatting.)
boolean isEmpty( )	Returns **true** if the invoking string contains no characters and has a length of zero.
Stream<String> lines( )	Decomposes a string into individual lines based on carriage return and line feed characters, and returns a **Stream** containing the lines. (Added by JDK 11.)
boolean matches(string *regExp*)	Returns **true** if the invoking string matches the regular expression passed in *regExp*. Otherwise, returns **false**.
int offsetByCodePoints(int *start*, int *num*)	Returns the index within the invoking string that is *num* code points beyond the starting index specified by *start*.

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, but no trailing empty strings will be included.
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 implemented by **String**.

Wrapper class

- To handle primitive data types java support it by using wrapper class.
- **java** provides the mechanism *to convert primitive into object and object into primitive*.
- **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically.
- The automatic conversion of primitive into object is known and autoboxing and vice-versa unboxing.

Example of wrapper class

```
public class Wrapper{
public static void main(String args[]){
//Converting int into Integer
 int k=20;
Integer i=new Integer(k); //converting int into Integer
Integer j=k;//autoboxing, compiler will write Integer.valueOf(a) internally
System.out.println(k+" "+i+" "+j);
}}
```

Output:

20 20 20