MET CS 520 – Information Structures with Java

Strings

The Class String

- There is no primitive type for strings in Java
- The class String is a predefined class in Java that is used to store and process strings
- Objects of type String are made up of strings of characters that are written within double quotes
  - Any quoted string is a constant of type String
    - "Live long and prosper."
- A variable of type String can be given the value of a String object

  - String blessing = "Live long and prosper.";

String Declarations

```
String s;
String input, result;

s = “hello”;
input = “123”;
result = “Java strings are fun!”:
```

String Declarations

```
String schoolName = "Boston University";

String test = new String(“Java“);
```

Concatenation of Strings

- Concatenation: Using the + operator on two strings in order to connect them to form one longer string
  - If greeting is equal to "Hello ", and javaClass is equal to "class", then greeting + javaClass is equal to "Hello class"
- Any number of strings can be concatenated together
- When a string is combined with almost any other type of item, the result is a string

  "The answer is " + 42 evaluates to

  "The answer is 42"

Classes, Objects, and Methods

- A class is the name for a type whose values are objects
- Objects are entities that store data and take actions
  - Objects of the String class store data consisting of strings of characters
- The actions that an object can take are called methods
  - Methods can return a value of a single type and/or perform an action
  - All objects within a class have the same methods, but each can have different data values

Classes, Objects, and Methods

- Invoking or calling a method: a method is called into action by writing the name of the calling object, followed by a dot, followed by the method name, followed by parentheses
  - This is sometimes referred to as sending a message to the object
  - The parentheses contain the information (if any) needed by the method
  - This information is called an argument (or arguments)

String Methods

- The String class contains many useful methods for string-processing applications
  - A String method is called by writing a String object, a dot, the name of the method, and a pair of parentheses to enclose any arguments
  - If a String method returns a value, then it can be placed anywhere that a value of its type can be used

```
String greeting = "Hello";
int count = greeting.length();
System.out.println("Length is " + greeting.length());
```

  - Always count from zero when referring to the position or index of a character in a string

Some Methods in the Class String (Part 1 of 8)

Display 1.4 Some Methods in the Class `String`

`int length()`

Returns the length of the calling object (which is a string) as a value of type `int`.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.length()` returns 6.

`boolean equals(Other_String)`

Returns `true` if the calling object string and the *Other_String* are equal. Otherwise, returns `false`.

EXAMPLE

After program executes `String greeting = "Hello";`

```
greeting.equals("Hello") returns true
greeting.equals("Good-Bye") returns false
greeting.equals("hello") returns false
```

Note that case matters. `"Hello"` and `"hello"` are not equal because one starts with an uppercase letter and the other starts with a lowercase letter.

(continued)

Some Methods in the Class String (Part 2 of 8)

Display 1.4 Some Methods in the Class `String`

`boolean equalsIgnoreCase(`*Other_String*`)`

Returns `true` if the calling object string and the *Other_String* are equal, considering uppercase and lowercase versions of a letter to be the same. Otherwise, returns `false`.

EXAMPLE

After program executes `String name = "mary!";`
`greeting.equalsIgnoreCase("Mary!")` returns `true`

`String toLowerCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to lowercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toLowerCase()` returns `"hi mary!"`.

(continued)

Some Methods in the Class String (Part 3 of 8)

Display 1.4 Some Methods in the Class `String`

`String toUpperCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to uppercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toUpperCase()` returns `"HI MARY!"`.

`String trim()`

Returns a string with the same characters as the calling object string, but with leading and trailing white space removed. Whitespace characters are the characters that print as white space on paper, such as the blank (space) character, the tab character, and the new-line character `'\n'`.

EXAMPLE

After program executes `String pause = "  Hmm  ";`
`pause.trim()` returns `"Hmm"`.

(continued)

Some Methods in the Class String (Part 4 of 8)

Display 1.4 Some Methods in the Class `String`

`char charAt(`*Position*`)`

Returns the character in the calling object string at the *Position*. Positions are counted 0, 1, 2, etc.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.charAt(0)` returns `'H'`, and
`greeting.charAt(1)` returns `'e'`.

`String substring(`*Start*`)`

Returns the substring of the calling object string starting from *Start* through to the end of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2)` returns `"cdefG"`.

(continued)

Some Methods in the Class String (Part 5 of 8)

Display 1.4 Some Methods in the Class `String`

`String substring(`*Start*`,` *End*`)`

Returns the substring of the calling object string starting from position *Start* through, but not including, position *End* of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned, but the character at position *End* is not included.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2, 5)` returns `"cde"`.

`int indexOf(`*A_String*`)`

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns −1 if *A_String* is not found.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.indexOf("Mary")` returns 3, and
`greeting.indexOf("Sally")` returns −1.

(continued)

Some Methods in the Class String (Part 6 of 8)

Display 1.4 Some Methods in the Class `String`

`int indexOf(A_String, Start)`

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string that occurs at or after position *Start*. Positions are counted 0, 1, 2, etc. Returns −1 if *A_String* is not found.

EXAMPLE

After program executes `String name = "Mary, Mary quite contrary";`
`name.indexOf("Mary", 1)` returns `6`.
The same value is returned if `1` is replaced by any number up to and including 6.
`name.indexOf("Mary", 0)` returns `0`.
`name.indexOf("Mary", 8)` returns `−1`.

`int lastIndexOf(A_String)`

Returns the index (position) of the last occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns −1, if *A_String* is not found.

EXAMPLE

After program executes `String name = "Mary, Mary, Mary quite so";`
`greeting.indexOf("Mary")` returns `0`, and
`name.lastIndexOf("Mary")` returns `12`.

(continued)

Some Methods in the Class String (Part 7 of 8)

Display 1.4 Some Methods in the Class `String`

`int compareTo(A_String)`

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering. Lexicographic order is the same as alphabetical order but with the characters ordered as in Appendix 3. Note that in Appendix 3 all the uppercase letters are in regular alphabetical order and all the lowercase letters are in alphabetical order, but all the uppercase letters precede all the lowercase letters. So, lexicographic ordering is the same as alphabetical ordering provided both strings are either all uppercase letters or both strings are all lowercase letters. If the calling string is first, it returns a negative value. If the two strings are equal, it returns zero. If the argument is first, it returns a positive number.

EXAMPLE

After program executes `String entry = "adventure";`
`entry.compareTo("zoo")` returns a negative number,
`entry.compareTo("adventure")` returns 0, and
`entry.compareTo("above")` returns a positive number.

(continued)

Some Methods in the Class String (Part 8 of 8)

Display 1.4 Some Methods in the Class `String`

`int compareToIgnoreCase(A_String)`

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering, treating uppercase and lowercase letters as being the same. (To be precise, all uppercase letters are treated as if they were their lowercase versions in doing the comparison.) Thus, if both strings consist entirely of letters, the comparison is for ordinary alphabetical order. If the calling string is first, it returns a negative value. If the two strings are equal ignoring case, it returns zero. If the argument is first, it returns a positive number.

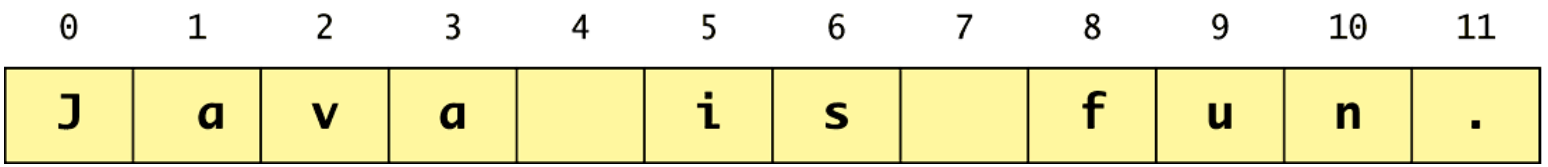
EXAMPLE

After program executes `String entry = "adventure";`
`entry.compareToIgnoreCase("Zoo")` returns a negative number,
`entry.compareToIgnoreCase("Adventure")` returns `0`, and
`"Zoo".compareToIgnoreCase(entry)` returns a positive number.

String Indexes

Display 1.5 String Indexes

The 12 characters in the string `"Java is fun."` have indexes 0 through 11.



Notice that the blanks and the period count as characters in the string.

Escape Sequences

- A backslash (\) immediately preceding a character (i.e., without any space) denotes an escape sequence or an escape character
  - The character following the backslash does not have its usual meaning
  - Although it is formed using two symbols, it is regarded as a single character

Escape Sequences

Display 1.6 Escape Sequences

```
\" Double quote.
\' Single quote.
\\ Backslash.
\n New line. Go to the beginning of the next line.
\r Carriage return. Go to the beginning of the current line.
\t Tab. White space up to the next tab stop.
```

String Processing

- A String object in Java is considered to be immutable, i.e., the characters it contains cannot be changed
- There is another class in Java called StringBuffer that has methods for editing its string objects
- However, it is possible to change the value of a String variable by using an assignment statement

```
String name = "Soprano";
name = "Anthony " + name;
```

Naming Constants

- Instead of using "anonymous" numbers in a program, always declare them as named constants, and use their name instead

```
public static final int INCHES_PER_FOOT = 12;
public static final double RATE = 0.14;
```

  - This prevents a value from being changed inadvertently
  - It has the added advantage that when a value must be modified, it need only be changed in one place
  - Note the naming convention for constants: Use all uppercase letters, and designate word boundaries with an underscore character

Comments and a Named Constant

Display 1.8 Comments and a Named Constant

```
1  /**
2   Program to show interest on a sample account balance.
3   Author: Jane Q. Programmer.
4   E-mail Address: janeq@somemachine.etc.etc.
5   Last Changed: September 21, 2004.
6  */
7  public class ShowInterest
8  {
9      public static final double INTEREST_RATE = 2.5;

10     public static void main(String[] args)
11     {
12         double balance = 100;
13         double interest; //as a percent

14         interest = balance * (INTEREST_RATE/100.0);
15         System.out.println("On a balance of $" + balance);
16         System.out.println("you will earn interest of $"
17                                                 + interest);
18         System.out.println("All in just one short year.");
19     }
20
21 }
```

Although it would not be as clear, it is legal to place the definition of INTEREST_RATE here instead.

SAMPLE DIALOGUE

```
On a balance of $100.0
you will earn interest of $2.5
All in just one short year.
```

Character Sets

- ASCII: A character set used by many programming languages that contains all the characters normally used on an English-language keyboard, plus a few special characters
  - Each character is represented by a particular number
- Unicode: A character set used by the Java language that includes all the ASCII characters plus many of the characters used in languages with a different alphabet from English

Pitfall: Using == with Strings

- The equality comparison operator (==) can correctly test two values of a primitive type
- However, when applied to two objects such as objects of the String class, == tests to see if they are stored in the same memory location, not whether or not they have the same value
- In order to test two strings to see if they have equal values, use the method equals, or equalsIgnoreCase

```
string1.equals(string2)
string1.equalsIgnoreCase(string2)
```

Lexicographic and Alphabetical Order

- Lexicographic ordering is the same as ASCII ordering, and includes letters, numbers, and other characters
  - All uppercase letters are in alphabetic order, and all lowercase letters are in alphabetic order, but all uppercase letters come before lowercase letters
  - If s1 and s2 are two variables of type String that have been given String values, then s1.compareTo(s2) returns a negative number if s1 comes before s2 in lexicographic ordering, returns zero if the two strings are equal, and returns a positive number if s2 comes before s1
- When performing an alphabetic comparison of strings (rather than a lexicographic comparison) that consist of a mix of lowercase and uppercase letters, use the compareToIgnoreCase method instead

Parsing Strings

- How would you parse this string?

"one two three"

The StringTokenizer Class

- The StringTokenizer class is used to recover the words or tokens in a multi-word String
  - You can use whitespace characters to separate each token, or you can specify the characters you wish to use as separators
  - In order to use the StringTokenizer class, be sure to include the following at the start of the file:

```
import java.util.StringTokenizer;
```

Some Methods in the StringTokenizer Class (Part 1 of 2)

Display 4.17 Some Methods in the Class `StringTokenizer`

The class `StringTokenizer` is in the `java.util` package.

```
public StringTokenizer(String theString)
```

Constructor for a tokenizer that will use whitespace characters as separators when finding tokens in `theString`.

```
public StringTokenizer(String theString, String delimiters)
```

Constructor for a tokenizer that will use the characters in the string `delimiters` as separators when finding tokens in `theString`.

```
public boolean hasMoreTokens()
```

Tests whether there are more tokens available from this tokenizer's string. When used in conjunction with `nextToken`, it returns `true` as long as `nextToken` has not yet returned all the tokens in the string; returns `false` otherwise.

(continued)

Some Methods in the StringTokenizer Class (Part 2 of 2)

Display 4.17 Some Methods in the Class `StringTokenizer`

`public String nextToken()`

Returns the next token from this tokenizer's string. (Throws `NoSuchElementException` if there are no more tokens to return.)[5]

`public String nextToken(String delimiters)`

First changes the delimiter characters to those in the string `delimiters`. Then returns the next token from this tokenizer's string. After the invocation is completed, the delimiter characters are those in the string `delimiters`.
(Throws `NoSuchElementException` if there are no more tokens to return. Throws `NullPointerException` if `delimiters` is `null`.)[5]

`public int countTokens()`

Returns the number of tokens remaining to be returned by `nextToken`.

Tokenize a string

```
String input = JOptionPane.showInputDialog("Type a sentence:");

// Tokenize using the default delimiters
StringTokenizer st = new StringTokenizer(input);
```

How many tokens?

```
System.out.printf("Total Tokens = %d\n", st.countTokens());
```

Iterating the tokens

```
// Iterate over the tokens
while (st.hasMoreTokens())
{
   String token = st.nextToken();
   System.out.printf("Token:%s\n",token);
}
```

Delimiters

- **`StringTokenizer(String input)`**—the *StringTokenizer* object is created for the specified input string. The default delimiters (space, tab, newline …) are used for tokenizing the input.
- **`StringTokenizer(String input, String delimiter)`**—the *StringTokenizer* object is created for the specified input string using the specified delimiter. Since the second argument is a string, any character in the string will act as a delimiter. If the values are separated by the *comma*, the delimiter will be the string `","`. If some values are separated by the *comma* and some others are separated by a *semicolon* in the specified string, the delimiter in this case will be the string `",;"`

StringBuffer Class

String objects are immutable.

```
String s = “chocolate”;
s = s + “is good”;
```

- The final value for s is a new String object with the value “chocolate is good”.
- The original value “chocolate” is no longer accessible and marked for garbage collection.
- Lots of String operations generate lots of String objects.

StringBuffer Class

- The StringBuffer class can be used to create a mutable string.

```
StringBuffer sb1 = new StringBuffer();
StringBuffer sb2 = new StringBuffer("orange");
```

- Creates an object that has an internal buffer to hold the value.
- The StringBuffer class has methods to modify the buffer to contain the new value.

StringBuffer Class

- StringBuffer **append**(data)—the append operation takes any of the primitive data type values and adds it to the end of the current string representation. If an object is specified as the argument, the toString representation of the object is used.The operation returns the same reference of the string buffer.
- char **charAt**(int index)—returns the character of the underlying string which is at the specified index.

StringBuffer Class

- StringBuffer **delete**(int startIndex, int endIndex)—the operation removes the characters from the specified start index (inclusive) to the specified end index (exclusive). The operation returns the same reference of the string buffer.
- StringBuffer **deleteCharAt**(int index)—the operation removes the character which is at the specified index. The operation returns the same reference of the string buffer.

StringBuffer Class

- int **indexOf**(String input)—returns the starting index of the first occurrence of the specified input in the underlying string. If there is no match, a value of -1 is returned.
- int **indexOf**(String input, int beginIndex)—returns the starting index of the first occurrence of the specified input in the underlying string starting from the specified index. If there is no match, a value of -1 is returned.

StringBuffer Class

- StringBuffer **insert**(int offset, data)—the insert operation takes any of the primitive data type values and inserts it at the specified offset of the current string representation. If an object is specified as the argument, the toString representation of the object is used.The operation returns the same reference of the string buffer.

StringBuffer Class

- int **lastIndexOf**(String input)—returns the starting index of the last occurrence of the specified input in the underlying string. If there is no match, a value of -1 is returned.
- int **lastIndexOf**(String input, int beginIndex)—returns the starting index of the last occurrence of the specified input in the underlying string starting from the specified index. If there is no match, a value of -1 is returned.
- int **length**()—returns the length of the string representation.

StringBuffer Class

- StringBuffer **replace**(int startIndex,int endIndex, String input)—the operation replaces the characters from the specified start index (inclusive) to the specified end index (exclusive) with the specified input string. The operation returns the same reference of the string buffer.
- StringBuffer **reverse**()—reverses the underlying string. The operation returns the same reference of the string buffer.

StringBuffer Class

- void **setCharAt**(int index, char input)—the specified input character replaces the existing character at the specified index.
- String **substring**(int startIndex)—the operation returns a new string using the characters in the underlying string from the specified start index.
- String **substring**(int startIndex, int endIndex)—the operation returns a new string using the characters in the underlying string from the specified start index (inclusive) to the specified end index (exclusive).

StringBuffer Class

- String **toString**()—returns a new string representing the underlying string.

StringBuffer Demo