Java Chapter 07 – Characters, Strings, & the String Builder

**STRING DATA PROBLEMS:**

\*\*In Java, String is a class & each created String is an Object 🡪 String variable names are references

\*\*When Comparing 2 Strings, using ‘==’ compares their memory locations NOT their values/contents

* 3 Classes for working w/ text data:
* Character: A Class whose instances can hold a single character value and whose methods manipulate & inspect single-character data
* String: A Class for working with fixed-string data (unchanging data composed of multiple characters)
* StringBuilder & StringBuffer: Classes for storing & manipulating changeable data composed of multiple characters

**CHARACTER CLASS METHODS:**

* char 🡪 primitive 🡪 can compare values using relational operators
* Comparisons are evaluated alphabetically
* Comparisons are made using each character’s Unicode values
* Character Class:
* Inherits from java.lang.Object
* Contains standard methods for testing character values
* Begin w/ ‘is’ 🡪 returns a Boolean value that can be used in comparison statements
* Begin w/ ‘to’ 🡪 returns a Character that has been converted to the stated format

**DECLARING & COMPARING STRING OBJECTS:**

\*\*A **literal string is an unnamed (anonymous) Object** of the String Class

\*\*A **String Variable is a named Object** of the String Class

\*\*\*When declaring a String Object, the String itself (the characters contained in the string) is distinct from the identifier used to refer to it\*\*\*

* Create:
* String aGreeting = new String(“Hello”); 🡪 unnecessary to use ‘new’ & String Constructor
* String aGreeting = “Hello”; 🡪 shortcut provided by Java

\*\*aGreeting stores a reference to a String Object

* \*\*The String variable name refers to a location in memory rather than a particular value
* int x = 10; 🡪 memory address of x holds 10
* String aGreeting = “Hello”; 🡪 memory address of aGreeting DOES NOT hold ‘Hello’, it holds another memory address where the characters are stored
* Assigning New Values:
* aGreeting = “Bonjour”; 🡪 assigns new value – memory address is altered
* \*\* “Bonjour” is an entirely new Object created w/ its own location
* The “Hello” String is still in memory but aGreeting no longer holds its address
* Eventually the garbage collector discards the “Hello” characters so the memory address can be used for something else
* Therefore, Strings are never actually changed; new Strings are created & String

References hold the new addresses

\*\*Comparing 2 String Objects, 1 is assigned directly & the other is entered from the keyboard:

* public class **TryToCompareStrings**

{

public static void main(String[] args)

{

String aName = “Carmen”;

String anotherName;

Scanner input = new Scanner(System.in);

System.out.print(“Enter your name: ”);

anotherName = input.nextLine();

if (aName == anotherName)

System.out.println(aName + “ equals ” + anotherName);

else

System.out.println(aName “ does not equal ” + anotherName);

}

}

* \*\*Output = ‘Carmen does not equal Carmen’ because even though the variables contain

the same series of characters, one is assigned directly while the other is entered from the

keyboard & is therefore stored in a different area of memory

\*\*Compares 2 declared String Objects that are initialized the same value/content. The value is stored once & both Object references hold the same address:

* String firstString = “abc”;

String secondString = “abc”;

if (firstString == secondString)

System.out.println(“Strings are the same”);

* \*\*Output = ‘Strings are the same’ because the memory address held by each are the same,

NOT because the contents are the same

**STRING CLASS METHODS:**

\*\*String Class provides many methods that compare Strings the way you intend:

* equals() :

\*Method Header: public bool equals(String s)

* Evaluates contents of 2 String Objects to determine if they’re equal
* Can take either a Variable String Object or a Literal String as its argument
* returns true if the objects have identical content no matter how it was assigned
* performs lexicographical comparison NOT alphabetical comparison
* public class **CompareStrings**

{

public static void main(String[] args)

{

String aName = “Carmen”;

String anotherName;

Scanner input = new Scanner(System.in);

System.out.print(“Enter your name: ”);

anotherName = input.nextLine();

if (aName == anotherName)

System.out.println(aName + “ equals ” + anotherName);

else

System.out.println(aName “ does not equal ” + anotherName);

}

}

Empty Strings VS Null Strings:

String word1 = “”; 🡪 empty String that references a memory address where no characters are stored

String word2 = null; 🡪 declared null – does not yet hold a memory address

String word3; 🡪 null by default – does not yet hold a memory address

\*\*word1 can be used w/ String Methods, word2 & word3 cannot

**OTHER STRING METHODS:**

**\*\*\*All examples use –** String myName = “Stacy”;

* toUpperCase() & toLowerCase() :
* Convert any string to its uppercase or lowercase equivalent
* Length() :
* Accessor Method that returns the length of a string
* IndexOf() :
* Determines whether a specific character occurs w/in a string
* If the character does in fact occur, it returns the position of the character
* charAt() :
* Requires an integer argument that indicates the position of the character that the method returns

myName.charAt(0); 🡪 returns “S” ; myName.charAt(4); 🡪 returns “y”

* endsWith() & startsWith() :
* Each take a String argument and return true or false if a String object does or does not end or start with the specified argument; \*case sensitive\*
* replace() :
* Allows you to replace all occurrences of some character with a String
* substring() :
* Used to extract part of a String; takes 2 integer arguments – a start & an end position
* regionMatches() :
* Used to test whether 2 regions are the same; 2 versions of this method exist
* Which regionMatches() expression will result in a true value

String oneStr = “Welcome Jim”; String “Welcome Joe”;

oneStr.regionMatches(0, twoStr, 0, 7);

* compareTo():
* Used to compare Strings; provides additional info to user in form of an integer value

Converting String Objects to Numbers:

\*\*If a String contains all numbers (like “123”) you can convert it from a String to a number so you can use it for arithmetic, or like any other number 🡪 Ex. For use of a user’s input value from an input dialog box

\*\*To convert a String to an integer you use the Integer Class :

* **Integer Class: (Wrapper)**
* Contains a simple integer and useful methods to manipulate it
* parseInt() : takes a String argument and returns its integer value;

int anInt = Integer.parseInt(“123”); 🡪 stores 123 in anInt

* **Double Class: (Wrapper)**
* parseDouble() : takes a String argument and returns its double value

double doubleVal = Double.parseDouble(“1.23”); 🡪 stores 1.23 is doubleVal

\*These are **Static Methods – used w/ Class name instead of an Object**

**OTHER METHODS STILL:**

* nextLine()
* toString() :
* Originates in Object Class
* Defined for other Classes to convert their Objects into Strings;

\*Converts any Object to a String

* String theString;

int someInt = 4;

theString = Integer.toString(someInt); 🡪 stores “4” in theString

\*\*You can also use concatenation to convert any primitive type (variable or constant) to a string using ‘+’

**STRING BUILDER/ STRING BUFFER:**

* Alternatives to the String Class
* Used when a String will be modified repeatedly; usually can use anywhere you would use a String
* **StringBuilder:** more efficient
* StringBuffer: thread safe – best used in applications that run multiple threads of execution

STRING BUILDER:

* Create a StringBuilder Object containing a String

Ex.

* Work w/ input as a StringBuilder Object 🡪 convert String using StringBuilder Constructor

Ex.

* Can combine the 2 Statements & avoid declaring the variable
* StringBuilder Constructor that allows assignment of capacity
* Syntax: StringBuilder bigString = new StringBuilder(500);

\*\*When creating a String – sufficient memory is allocated to accommodate the number of Unicode characters in the String, BUT w/ StringBuilder – contains buffer that may/may not contain a String

* The length of the String can be different than the length of the buffer
* The actual length of the buffer 🡪 capacity of the StringBuilder
* The length of the StringBuilder Object = # of characters in the string contained in the SB Object
* setLength() : allows you to change the length of the SB Object’s length
* When increased to longer than the String it holds 🡪 extra characters contain ‘\u0000
* When decreased to shorter than its String 🡪 the string is truncated/ cut off
* capacity() : finds capacity of the SB Object

\*\*When creating a SB Object w/ a String as an argument to the constructor, the SB capacity is the length of the String + 16 🡪 for reasonable modofications

* append() : allows you to add characters to the end of the SB Object
* insert() : allows you to add characters to a specific location w/in a SB Object
* setCharAt() : allows you to change a character at a specified position
* 2 arguments 🡪 integer position, ‘character’
* charAt() : extracts a character from a SB Object to use
* 1 argument 🡪 the character position starting at the beginning of the String, which is 0