// CS 0401 Spring 2019 Java Example 8

// This handout demonstrates some issues relating to reference variables and

// using them correctly to access objects.

**public** **class** Lab05

{

**public** **static** **void** main (String [] args)

{

// The StringBuilder class is related to the String class. See

// Sections 2.9 and 9.3 in the text for details on the String class.

// (StringBuilder itself is discussed in Section 9.4. However, we

// are just looking at it superficially here.) StringBuilder objects

// can also store and access strings of text. However, unlike String

// class objects, StringBuilder class objects can be modified after

// they are created. We will look more at the benefits of each later

// on.

StringBuilder S1, S2;

S1 = **new** StringBuilder("Hello");

System.***out***.println("S1 is " + S1.toString());

//System.out.println("S2 is " + S2.toString()); // This will give a compilation

// error -- variable might not have been initialized

S2 = **null**;

//System.out.println("S2 is " + S2.toString()); // This will give a

// NullPointerException

S2 = S1;

System.***out***.println("S2 is " + S2.toString()); // This will work but note that

// S2 and S1 are references to a single object.

//System.out.println(S2.value[0]);

System.***out***.println(S2.charAt(0));

//System.out.println(S2.value.length);

// The statements above are accessing data that exists within the

// StringBuilder but that is not accessible to the user. Remove the

// comments and see what happens when you compile the program. We will

// discuss this more soon.

// updating object via S1. Note that S2 also reflects the change

S1.append(" there Java maestros!");

System.***out***.println("S1 is " + S1); // If an object is in a println statement

// with no method call, then implicitly the "toString()" method

// for that object will be called.

System.***out***.println("S2 is " + S2.toString());

StringBuilder S3 = **new** StringBuilder("Hello there Java maestros!");

**if** (S1 == S2) // comparing references

System.***out***.println(S1.toString() + " == " + S2);

**if** (S1 != S3)

System.***out***.println(S1 + " != " + S3);

**if** (S1.toString().equals(S3.toString())) // comparing actual strings

System.***out***.println(S1 + " equals " + S3);

S1 = **null**; // Now S1 cannot be accessed using the "dot" notation but

// S2 still refers to the same object

//S1.append(" This is not allowed");

S2.append(" This is ok");

System.***out***.println("S1 is " + S1); // Even though S1 is null this is allowed

// because of a special case that is handled in the println

// statement. Note the output.

System.***out***.println("S2 is " + S2);

// Additional interesting note about the String class:

// The first initialization below is assigning a String without

// using the new operator, while the second uses new. Java

// Strings cannot be changed (i.e. they are constant), and String

// literals implicitly create objects that are "interned" -- or

// shared among all literals that are the same. Thus, St1 and

// St3 below are == whereas St1 and St2 are not (since a new

// object is made for St2).

String St1;

St1 = "Herb";

String St2 = **new** String("Herb");

String St3 = "Herb";

**if** (St1 == St2)

System.***out***.println("St1 and St2 Equal");

**if** (St1 == St3)

System.***out***.println("St1 and St3 Equal");

**if** (St1.equals(St2))

System.***out***.println("St1 and St2 have same data");

}

}

S1 is Hello

S2 is Hello

H

S1 is Hello there Java maestros!

S2 is Hello there Java maestros!

Hello there Java maestros! == Hello there Java maestros!

Hello there Java maestros! != Hello there Java maestros!

Hello there Java maestros! equals Hello there Java maestros!

S1 is null

S2 is Hello there Java maestros! This is ok

St1 and St3 Equal

St1 and St2 have same data

/\* CS 0401 Spring 2019 Java Example 9

This example demonstrates the idea of encapsulation of data and operations

in a single Object, and of data abstraction. The class used is the

StringBuilder class. Note the comments in various places below. Note in

particular comments about CONSTRUCTORS, ACCESSORS and MUTATORS.

For a LOT more information on the StringBuilder class, including the full

list of available methods see the documentation in the Java API Online:

https://docs.oracle.com/javase/8/docs/api/

\*/

**public** **class** Lab05

{

**public** **static** **void** showInfo(StringBuilder S)

{

// All of the methods used here are ACCESSORS -- they are used

// to access properties of the objects, without changing them.

System.***out***.printf("Data: %-12s Length: %-3d Capacity: %-3d \n",

S.toString(), S.length(), S.capacity());

// Mutate object that S references. This is clearly not a good

// idea in a method called "showInfo" -- it is just for demo purposes.

// Uncomment the line to see its (side) effect. Note that this statement

// is not changing variable S at all. Rather, it is changing the object

// that S refers to.

//S.append("BOGUS");

// Assign S to a new object -- value parames keep this from affecting

// the argument, since it is changing the local variable S (the parameter)

// rather than the argument.

//S = new StringBuilder("ZANY");

//System.out.println(S);

}

**public** **static** **void** main (String [] args)

{

StringBuilder S1, S2, S3;

S1 = **new** StringBuilder(); // Create an empty StringBuilder using a

// CONSTRUCTOR

S2 = **new** StringBuilder("abstraction");

// Create a StringBuilder that has some initial data in it, using a different CONSTRUCTOR.

*showInfo*(S1); Lab05.*showInfo*(S2);

System.***out***.println();

// The append() method is a MUTATOR. Mutators alter the object in some way.

// In this case, they are adding additional characters to the StringBuilder.

S1.append("wacky");

S2.append(" contraption");

*showInfo*(S1); *showInfo*(S2);

System.***out***.println();

// insert() is another MUTATOR. Note in the second line below we are

// also using an ACCESSOR. The idea is to insert "distraction " after the

// first space in the previous string.

S1.insert(0, "very ");

S2.insert(S2.indexOf(" ")+1, "distraction ");

*showInfo*(S1); *showInfo*(S2);

System.***out***.println();

Recall from our previous discussions about reference types and the

implications of using them. Here S3 and S1 are now different

variables referencing the SAME StringBuilder object. Thus when we

MUTATE S3 using the reverse() method, and then print S1, we see that

// that object has been reversed.

S3 = S1;

S3.reverse();

*showInfo*(S1);

}

}

Data: Length: 0 Capacity: 16

Data: abstraction Length: 11 Capacity: 27

Data: wacky Length: 5 Capacity: 16

Data: abstraction contraption Length: 23 Capacity: 27

Data: very wacky Length: 10 Capacity: 16

Data: abstraction distraction contraption Length: 35 Capacity: 56

Data: ykcaw yrev Length: 10 Capacity: 16

// CS 0401 Sprint 2019

// Simple example of using arrays and text files

**import** java.util.\*;

**import** java.io.\*; // import needed for Java file classes

**public** **class** ex11

{

// Note again the "throws IOException" tags on various methods. This is

// necessary for creating, reading or writing to Java files.

**public** **static** **void** main(String [] args) **throws** IOException

{

// A Scanner is used for both the keyboard input and the file

// input. The only difference is in the argument to the constructor.

Scanner keyIn = **new** Scanner(System.***in***);

System.***out***.print("Input file name: ");

String fName = keyIn.nextLine();

File inFile = **new** File(fName);

Scanner fileIn = **new** Scanner(inFile);

**int** num = fileIn.nextInt(); // Read an int from the file that indicates

// how many values are in the rest of the file.

// Note how the array size is determined dynamically. This way,

// the program will work for any properly formatted file.

**int** [] A = **new** **int**[num];

**for** (**int** i = 0; i < num; i++)

A[i] = fileIn.nextInt();

fileIn.close();

//A.length++;

*show*(A);

*modify*(A, keyIn);

*save*(A, fName);

// Uncomment the line below to see an example of appending to the

// end of a file. Note that this item will not be read in again when

// the program is re-run, since the int indicating how many items are

// to be read will not take this value into account.

//addToEnd(99, fName);

}

// Demonstration of the "foreach" loop in Java. This is useful if you

// are only concerned with the data and not the indices.

**public** **static** **void** show(**int** [] Ar)

{

**for** (**int** value : Ar)

System.***out***.print(value + " ");

System.***out***.println();

}

// Even though Java parameters are value, since we are passing a

// reference to the method, this allows us to update the contents

// of the array (i.e. mutate it) from within the method.

**public** **static** **void** modify(**int** [] Ar, Scanner keyBoard)

{

**int** index = 0;

**do**

{

System.***out***.print("Enter index you wish to update: ");

index = keyBoard.nextInt();

} **while** (index < 0 || index >= Ar.length);

System.***out***.print("Enter new value: ");

Ar[index] = keyBoard.nextInt();

}

// Note how the PrintWriter is used to output the array data back to

// the text file. Don't forget to close the file at the end --

// otherwise some data may be lost.

**public** **static** **void** save(**int** [] Ar, String fName) **throws** IOException

{

PrintWriter fileOut = **new** PrintWriter(fName);

fileOut.println(Ar.length);

**for** (**int** data : Ar)

fileOut.println(data);

fileOut.close();

}

// This method uses a FileOutputStream to enable an append to the end

// of a file. When a file is opened in this way, writing via printlns

// begins at the end of the file, rather than at the beginning, thus

// preserving the data that was previously in the file

**public** **static** **void** addToEnd(**int** x, String fName) **throws** IOException

{

PrintWriter fileOut = **new** PrintWriter(**new** FileOutputStream(fName,**true**));

// FileOutputStream allows file

// to be opened in "append" mode

fileOut.println(""+x);

fileOut.close();

}

}