Java

1/27/14

**The World of Java**

Q: Why Java vrs. C++?

A: C++ is a source code language. It needs to be compiled before it can run. But that’s very computer specific. Depends on the hardware of the computer and the binary instructions associated with it. So what runs on a linux based computer may not run on an apple or windows computer. The Executable Binary code is computer specific.

Java on the other hand has a source code that compiles into a universally formatted executable code called “Bytecode”. Then any computer can download a Java Virtual Machine (JVM), which will interpret the “Bytecode” on any computer.

Q: What do you need to run Java?

A: 1) API- Application Program Interface= a huge collection of stored libraries

2) JRE- Java Runtime Environment= you need the Java Virtual Machine + the API to successfully run your java program

3) JDK- Java Development Kit = JRE plus compiler so you can edit and debug, javadoc…

Ex. Ada Source Code is a military program code. It’s not written in Java but in can be compiled into “Bytecode” which can then run on any machine containing the JVM

Java ensures that the program will run the same way on any computer regardless of storage capacity. Java has its own max size (positive integer 2,147,483,647). Therefore every person has this same max integer (even if the computer can’t actually handle that number).

So with Java, computer system, hardware, size, etc. no longer matters anymore because of the universally executable “bytecode” and the JVM.

1/29/14

**Storage in Computers**

Numbers are called different things depending on their size. A byte holds an 8 bit signed two’s complement integer. In general, the number of bits and the range of values of two’s complement integers is 2n-1.If you have a 4 bit number (0000, 0001, 0010 …) it runs from -8 to 7. So a 4 bit number can represent 8 numbers in positive and negative (a total of 16 numbers). The positive numbers are 0-7 and the negative numbers are -1 to -8. There is no negative zero in two’s compliment. So the total range of integers per number of bits is the formula -(2n-1) … (2n-1)-1 .

An integer is a 32 bit two’s complement number. So the range of numbers is -(2n-1)…(2n-1)-1 🡪

-(232-1)…(232-1)-1 🡪 -(231)…(231)-1 = -2,147,483,648….+2,147,483,647 (so the highest number available is roughly 2 billion which is a pretty big number and gives us a pretty big range of numbers to work with)

ASCII is an 8-bit code which stores characters. That gives you 28 available characters (different bytecode combos) which is 256 choices. That was good for Americans but when the internet came about and we had multiple languages that needed characters too, Unicode came out which is a 16-bit code. That gives us 64000 characters to choose from (different ways to make unique binary combos using 16 bits). Since Unicode is 16 bits it can be represented in hexadecimal.

**Classes and Objects**

Class = a template that describes the properties and behaviors of an object. (cookie cutter)

Object = an instance of a class which has specific properties, the class defined. (cookie)

Instantiation = making an *instance* of a class.

Ex.

**Cat pet1, pet2, pet3** //creates 3 objects of the class cat

//define the details of the objects-instantiate them

pet1 = new Cat("tabby","male","gray",4,true);

pet2 = new Cat("calico","female","brown",4,true);

Static vrs. Instance  
 *Static* variables belongs to the class. There is only one version of the variable for every instance of the class. A detail about the class which doesn’t change per instantiation.

*Instance* variables belong to the object. This detail about the object changes per instantiation.

Ex.

**Class Cat {**

**Static int NUMBER\_OF\_LEGS = 4;** //static variables get all capitalized

**String gender;** //classes begin with an uppercase letter, string is a class

**String breed;**

**String color;**

**Boolean neutered;**

**}**

int leg = Cat.NUMBER\_OF\_LEGS; //creates and integer which is equal to the static int NUMBER\_OF\_LEGS

//which is located inside the class Cat

String myCatBreed = pet1.breed //breed is an instance variable so its value comes from a particular

//instance of Cat

You can change static variables but it will cause to changes to the static variable throughout the program. The **final modifier** makes variables constants.

Methods (*functions in C++)* =define the behavior of an object.

Ex. A Cat object could tell you its breed:

public String getBreed() {

return breed;

}

* String b = pet1.getBreed();

Methods can also be static.

Java classes have inheritance.

2/3/14

Java Lingo:

* System.out.print (“This is a sentence”); // prints on the same line only
* System.out.println (“This is a sentence.”); // built in new-line within the ‘println’
* System.out.println (“x is “ + x); // + operator allows concatenation during print to
* float i = 0.0f; //float values must have an f symbol at the end
* **To Declare and array**: int[] myarray; **To Initialize it*:*** myArray = new int[10];

NOTE: array\_name.length is built-in method that returns the length of the array.

* Java main method:

**public class MyClass {**

**public static void main (String[] args) { ….**

**} // main**

**} //MyClass**

NOTE: Java file name MUST match with the public

class name! Otherwise, it will not compile!

* Comparison operators (==, <= …) can only be used with Java’s 8 primitive data type: int, float, double, char, byte, … etc.
* String is a java class, not a primitive data type, so it can’t be compared using comparison operators. Instead use: string1.equals(string2) Note: Case sensitive
* Arrays must be declared and instantiated before it can be used (like all variables). Once an array is instantiated, it’s length cannot be changed

Ex. String[] wordArray = new String[10]; //wordArray- declaring, String[10]- instantiating

* **command-line arguments** are entered when running the program

Ex. java Lab5program1 Lab5input.txt //stores Lab5input.txt in the args array

* **Parsing Arguments** = when you convert data types

Ex. Integer.parseInt(String s1) // Converts s1 into an integer

Note: String s1 MUST be numbers otherwise an exception is thrown.

* To use Java’s built-in GUI (Graphical User Interface), you must declare **import javax.swing.\*** in the beginning of the file.

1. Use **JOptionPane.showMessageDialog(null, “This is a message pop-up window.”);** to show your message.
2. Use **int n = JOptionPane.ConfirmDialog(null, “Click YES or NO.”, “Confirmation”, JOptionPane.YES\_NO\_OPTION);** for confirmation options.
3. Use **String name = JOptionPane.showInputDialog(null, “Please input your name.”);** to prompt user input

* TextFileInput.java and input file (to be read) must be saved in the same directory. **c:>java Lab5Program1 lab5input.txt** is how you call the txt file with your program. Its immediately saved into the args[0] string array.
* Strings are objects but we don’t have to dynamically allocate them using new each time. Since they’re so common, Java allows us to just declare a String and the rest of the dynamic allocation of the object is done behind the scenes (syntactic sugar)
* StringTokenizer- Each line of text is broken down into multiple parts known as “tokens”. A specific “delimiter” is used to break apart the text. Must import java.util.\* library to utilize this class. In legacy class, recommended to use String.split method instead.

1. hasMoreTokens ()- Returns boolean value to represent if there’s any more token left in the line.
2. countTokens()- Returns an integer representing the number of tokens within the current string. (does not advance position)
3. nextToken()- Returns the next piece of data in the String. Position of string is shifted to the beginning of next token after this method is called.

Goal: open a file containing integers on different lines -> place integers into an array -> sort the array:

We need a subArrayLength, and not just array.length, because the array that will hold the numbers from the txt is very big so that it has room for a lot of txt. We only care about how much of the array actually got filled with content though. Not the left over spacious part that’s filled with null. So subArrayLength will represent the length of how much of the array the numbers from the txt filled up.

**subArrayLength = inputFromFile(inputFileName, numbersArray);** /\*subArrayLength - how much of the array is filled ex. If up to numbers[4] contains integers, then the subArrayLength is 5 – is equal to the integer output of function inputFromFile \*/

**private static int inputFromFile(String filename, short[] numbers){**  /\*a private static function

that takes in a string called filename and an array of shorts called numbers, and returns int\*/  
 **TextFileInput in = new TextFileInput(filename);** /\* creates object TextFileInput

called in and instantiates it to whatever’s in the file ‘filename’: opens the file\*/  
 **int lengthFilled = 0;**   
 **String line = in.readLine();** /\*line is equal to what the function readLine reads from

the TextFileInput in- it reads a line of an already open file\*/  
 **while ( lengthFilled < numbers.length && line != null ) {**  
 **numbers[lengthFilled++] = Short.parseShort(line);** /\*the array at that index is filled and then incremented and what’s inside of numbers at that index is set equal to what was

in the string line, but parsed into a short\*/  
 **line = in.readLine();** /\* line is equal to what the function readLine reads from in \*/  
 **} // while**

**if ( line != null ) {**  
 **System.out.println("File contains too many numbers.");**  
 **System.out.println("This program can process only " numbers.length + "numbers.");** System.exit(1**);   
 } // if   
 in.close();**   
 **return lengthFilled;**

**} // method inputFromFile**

1. **// average the numbers**

**sum=0;**

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

**sum += numbers[i];**

**Average = sum/subArrayLength;**

1. **// find the smallest number**

**smallest = numbers[0];**

**for (int i =1; i<subArrayLength; i ++)**

**smallest = Math.min (smallest, numbers[i]);** /\* the new smallest number is equal to the

outcome of the min function on math\*/

1. **// find the *index* of the smallest number**

**indexLowest = 0;**

**for ( int j = 1; j < subArrayLength; j++ )**

**if ( array[j] < array[indexLowest] )**

**indexLowest = j;** /\*if the value in array[j] is less than the value in array[indexLowest] then

the new indexLowest is equal to j \*/

1. Find the *index* of the smallest number. This is the basis of “Selection Sort”. Find the smallest number and swap it with the number at the top of the array:

**private static void selectionSort (short[] array, int length) {**

**for ( int i = 0; i < length - 1; i++ ) {**  /\*length-1 because array index starts at zero\*/

**int indexLowest = i;** /\* we’re gonna start at index 0 andfill it with the smallest

value (using the following nested for loop. When that’s done we move on to the next index and

fill that with the second to smallest number in the array, ect- outer loop serves as an index

pointer and inner loop helps us find the smallest integer\*/

**for ( int j = i + 1; j < length; j++ )** /\* I is the pointer to the smallest index that we

will swap to\*/

**if ( array[j] < array[indexLowest] )**

**indexLowest = j;**

**if ( array[indexLowest] != array[i] ) {** /\*if the value of the current lowest index is not equal to the index that we want to declare the new lowest, then swap\*/

**short temp = array[indexLowest];**

**array[indexLowest] = array[i];**

**array[i] = temp;**

**} // if**

**} // for i**

**} // method selectionSort**

2/17/14

***Primitive type*** parameters are passed by value- just a copy of the value of the parameter is given to the method; the variable in the main program and the variable in the method each have their own memory location = **DOESN’T CAUSE CHANGES TO THE MAIN PROGRAM**

***Object type*** parameters are passed by reference and only a reference to the parameter is given to the method; both the variable in the main method and the other method each have their own memory location for the same reference. If you change the reference in the method memory location, it changes what the method points to. Doesn’t change the reference in the main program.

Example of changing the reference, what’s being pointed to, but not the actual object:

Ex. **public class StringTest {**

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

**String s1="cat", s2="dog"**

**System.out.println("s1 is "+s1+", s2 is "+s2);** /\* “s1 is cat, s2 is dog

**doSomething(s1,s2);** cat dog

**System.out.println("s1 is "+s1+", s2 is "+s2);** “s1 is cat, s2 is dog”\*/

**}**

**public static void doSomething(String x, String y) {**

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

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

**x = "fish";** /\*String x was originally pointing to “cat” (just like String s1 was), but now String x

points to fish instead\*/

**y = "bird";**

**}**

**}**

The only way to make a permanent change to the object is to perform a method on the object.

Ex. s1.equals(“fish”);

Ex. **public class TemperatureTest {**

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

**Temperature t1= new Temperature(32.0f);** /(\*Instantiates) the object

temperature, setting it equal to 32.0f\*/

**System.out.println("t1 is "+t1);**

**doSomething(t1);**

**System.out.println("t1 is "+t1);**

**}**

**public static void doSomething(Temperature x) {**

**x.setTemperature(98.6f);** /\* performs setTemperature method on object

x- permanently setting the temperature=98.6f \*/

**}**

**}**

**Prints:** “t1 is 32.0, t1 is 98.6”

Passing arrays (an object) into methods cause permanent changes since the array is syntactic sugar for functions actually being performed on the array at that index. Swapping just values in a function causes nothing to happen in the main program

Ex. **public static void swap (int[] a, int x, int y) {** /\*takes in an array and two integers\*/

**int temp;** /\*temp only exists inside this method\*/

**temp = a[x];**

**a[x]=a[y];** /\* (a+x).set(a[y])- because the object is having a method performed on it,

the change applies to the main program too\*/

**a[y]=temp;**

**}**

2/19/14 – Program Modularity and Error Handling

**Modularity**- Modularity is the degree to which a system's components may be separated and recombined.

When given a big task, just break it down into smaller parts. Make a skeleton program: logic layout, variables you will need, functions and their parameters and return type, error checks- data validation, both from user input, file input, and input and return type from other methods…

Test your program with all different inputs and see if it produces the expected output. Instead of inputting in data each time you run, input a file with a variety of test cases.

Putting the SSN into a string will help fix the leading zeroes problem. If stored as an integer, the leading zeroes will still be saved and act as place holders, but when asked to print the number, java will omit the zeroes. Therefore we save it as a string to avoid the problem all together.

Make ssnSize a global variable so that changes to it happen throughout the whole program.

Data Validation- making sure that the input received is in accordance with what we are looking for (that the user is actually importing social security numbers). Check input from users, from files, from values received from other methods. Always verify what you are receiving.

Ex. Check that the social security number has 9 digits like all other ssn methods-.length is a method you can use on strings; also make sure that the user only entered numbers and no characters or symbols; make sure your array exists, and that it has storage space left

Run-time error messages can stem from bad code or bad user input. Make sure your methods are doing what they are supposed to do.

**Gui**

The Gui comes with a lot of different variables: size, location, title, default close operator, visibility. Once you’ve initialized a GUI, here’s how to insert data into it:

1. Make a container and attach it to the Jframe:

Ex. **Container myContentPane = jf.getContentPane**

1. Make a text area: **TextArea myTextArea = new TextArea();**
2. Add the text area to the content pane (which is attached to the JFrame)

Ex. **myContentPane.add(myTextArea);**

1. Append words to the text area (which is attached to the content pane which is attached to the JFrame):

Ex. **myTextArea.append(list[i] + “\n”);**

1. Don’t forget to set the JFrame visibility option to true: **jf.setVisible(true);**

In the main program we instantiate the GUI and using our object reference (name) we manipulate it to how we want it. The info about what the GUI object is though (its formatting), is stored in a separate class from the main class. (we define the GUI in its own class and use the GUI in the main class).

Content pane is where the information in the gui is printed. But we can’t put actual strings there. We put a reference to an object that can contain strings inside of it- the text area- into the content pane. The text area can only hold strings, so will need to parse integers into strings in order to have it represented from the text area onto the content pane. The gui just represents what we want it to, all the calculations and manipulations are done inside the main program. A separate class is made for the formatting of the gui.

To utilize ***Inheritance*** add “extends JFrame” to the title of the class: **public class SSNGUI extends JFrame{…}**

The LayoutManager allows us to divide the Content Pane into different areas:

1. BorderLayout*-* **myContentPane.add(myTextArea, BorderLayout.EAST);**
2. GridLayout- **mySSNGUI.setLayout(new GridLayout(1,2));**

**Non-application Classes**= no main method. Classes without main methods are true objects. They can’t exist without being instantiated. A special method called a **constructor** is automatically executed when an object is instantiated can utilize inheritance.

Can two methods with the same name be in the same class? It depends if the signature of the methods are the same (output and input parameters). As long as the signatures are different then it’s ok if their names are the same.

**import javax.swing.\*;** /\*imports jframe material\*/

**public class SSNGUI extends JFrame{** /\*gui object for SSNGUi, child of JFrame class\*/

**public SSNGUI(String title, int height, int width){**  this is the consturctor which initializes the

object\*/

**setTitle(title);**

**setSize(height, width);**

**}//constructor**

**public static void initialize(){** /\*initializing method that the SSNGUI class can use\*/

**mySSNGUI.setLocation(100,100);**

**mySSNGUI.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);**

**mySSNGUI.setVisible(true);**

**}//initialize**

**public static void printSSNtoJFrame(SSNGUI ssnGUI, String[] list, int size){** /\*putting data to

JFrame method the SSNGUI class can use\*/

**ssnGUI.initialize();**

**Container myContentPane = ssnGui.getContentPane();**

**TextArea myTextArea = new TextArea();**

**TextArea mySubscripts = new TextArea();**

**myContentPane.add(myTextArea, BorderLayout.EAST);**

**myContentPane.add(mySubscripts, BorderLayout.WEST);**

**for(int i=0; i<size; i++){**

**mySubscripts.append(Integer.toString(i)+"\n");**

**if(!isValidSSN(list[i]))**

**myTextArea.append("Invalid SSN: "+list[i]+"\n");**

**else**

**myTextArea.append(list[i]+"\n");**

**}//for**

**}//printSSNtoJFrame method**

**}//SSNGUI class**

**//main class**

**public class SSN{**

**SSNGUI mySSNGUI = new SSNGUI("Social Security Number", 400, 200);** //creates an SSNGUI

**String[] list = {"123456789", "987654321"};**

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

**printSSNtoJFrame(mySSNGUI, list, list.length);**

**}//main**

**}//SSN**

Classes

An object is defined by its attributes- data values, and behavior- methods

Every Class has a constructer.

The keyword **private** is used to ensure that the object retains correct data values. Public data values can be changed by the user. A private method is not called from the outside. It can only be accessed by members of the same class.

**Static** methods are methods which are not the behavior of the object. They can be called on any object anywhere in the program.

Check that methods that assign values to the data return a valid value. If not then throw an exception.

The **String toString()** method is used to get a String object representing the value of the Number Object. If the method takes a primitive data type as an argument, then the String object representation of the primitive data type value is returned. Ex.

public class Test{

public static void main(String args[]){

Integer x = 5;

System.out.println(x.toString());// “5”

}

}

System.out.println(Integer.toString(12));// “12”

The keyword **this** is a reference to the current object- the object who’s method or constructer is being called. Can only refer to the current object as this during it’s instance method or constructer method. The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter. Ex. **public class Point {**

**public int x = 0;**

**public int y = 0;**

**//constructor**

**public Point(int x, int y) {**

**this.x = x;**

**this.y = y;**

**}**

**}**

**public class SSN{**

**private String SSNumber;//we don't want people changing it so it's private**

**public SSN (String SSNumber){//constructer- when you instantiate: SSN name = new SSN("123456789");**

**if(!isValidSSN(a))//make sure it's valid since this method is assigning value**

**throw new IllegalArgumentException("Invalid SSN.");**

**else**

**this.SSNumber = SSNumber**; /\*the thing (the SSNumber object) I'm consturcting is

equal to the input of the constructor (a string called SSNumber)\*/

**}**

**public void setSSN(String s){//name.setSSN("987654321")**

**if(!isValidSSN(a))**

**throw new IllegalArgumentExeption("Invalid SSN)";**

**else**

**SSNumber = s;**

**}**

**public String getSSN(){//String get = name.getSSN;**

**return SSNumber;**

**}**

**private static boolean isValidSSN(String s){//only the other methods from this class can call this function**

**if(s.length() != 9)**

**return (false);**

**for(int i=0; i<9; i++)**

**if(! Character.isDigit(s.charAt(i)))**

**return (false);**

**return (true);**

**}**

**public boolean equals(Object other){**

**return ( other != null && getClass() == other.getClass() && SSNumber.equals((SSN) other).SSNumber));**

**}**

**public int compareTo(SSN other){**

**return SSNumber.compareTo(other.toString());//compare your SSNumber to the other SSNumber (needs to be converted to a string to compare)**

**}**

**public String toString(){ //overriding functions that are inherited from the parent Object class**

**return SSNumber;**

**}**