Codecademy Java Notes

8/12/20

**LESSON 1 NOTES- INTRO**

- class name must match file name "HelloWorld.java" (case sensitive)

- Java is a **COMPILED** language = code is transformed into byte code by a

compiler before it is executed.

- JavaScript is a **INTERPRETED** language = no compiler. but code is read

and executed by some other program

- can compile java in mac terminal with "javac [filename].java"

- this creates a .class file in your current directory, which you can

then run with the below command:

- can run java in mac terminal with "java [filename]"

1) Java files must have at least one class AND one main() method

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

- **public** = allows main to be accessible everywhere (like ruby public methods)

- **static** = means method is a class method (doesn’t need an instance to run)

- **void** = main method does not return any values

- **main** = name of method

- every file/class must have a main method

- entry point for app, all other methods are invoked here

- accepts a single arg. an array of strings

- runs automatically

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello World!");

}

}

**LESSON 2 NOTES- VARIABLES**

1) **PRIMITIVE DATA TYPES**:

- **int** integers. ex. -13, 0, 2

- **double** doubles. ex. 1.7 E+ 308 (17 followed by 307 0's) rational nums

- can store larger/smaller numbers than int

- **boolean** booleans. ex. true, false

- **char** characters ex. 'a', 'A', '!'

- can only have ONE character ('AA' not allowed)

- chars have single quotes ', strings have double quotes "

2) **String** strings ex. "hello"

3) Misc. Notes

- variable names must start with valid letter, or $, or a \_

- Must declare the data type when declaring a variable

Ex:

int year;

2) **OPERATORS**:

- similar to JavaScript

- **integer division** = any remainder is lost (num is floored)

ex.

int uneven = 10 / 4; //=> 2

- can't use **inequality operator** == on strings, must use .equals()

- use + for **string concatenation**

QUESTIONS

- why does String have to be capitalized (var declaration), but int doesn't?

- Bec. String is an object, not a primative data type

public class L2Notes {

public static void main(String[] args) {

String name = "Harry";

int year = 2012;

double bigNum = 3000.00;

boolean lookingForJob = true;

char gender = 'm';

}

}

**LESSON 3 NOTES- Object Oriented Java**- instance variables and instance methods

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

- **public** = allows main to be accessible everywhere (like ruby public methods)

- **static** = ?

- **void** = main method does not return any values

- **main** = name of method

- every file/class must have a main method

- entry point for app, all other methods are invoked here

- accepts a single arg. an array of strings

- runs automatically

2) **CONSTRUCTOR method**: public [class name]()

- like the constructor method in JavaScript

- only gets executed when an instance of the class object is created

- params must be in constructor if passed in when instance created

not like JS where extra args are ignored!!

3) creating an **instance** of an object

Ex.

Store lemonadeStore = new Store();

4) printing an object to console shows objects location in computer memory

Ex.

System.out.println(lemonadeStore); //=> Store@6bc7c054

5) **INSTANCE VARIABLE** (attributes/properties of obj) must be

declared outside constructor method!! inside class (anywhere, usually top)

- assignment of instance variables inside constructor!!

- these vars represent state (properties) and behavior (instance methods)

6) accessing attributes/properties (use dot . operator just like in JS)

7) **Instance methods**

- don't have to be declared outside of constructor like attributes?

8) **SCOPE**

- can't access instance fields (attributes/props) in main function

- can access instance fields in methods

- variables exist in the scope they were declared in

QUESTIONS

- what is "static" in "public static void main()"?

- does public static void main() always take in a string as an arg?

- is there a concept of "this" in Java like in JS?

- does returning void mean returning null or undefined?

public class Store {

// instance fields (analagous to attributes/properties in JavaScript)

String productType;

int inventoryCount;

double inventoryPrice;

// constructor method

public Store(String product, int count, double price) { // can't name args same name as attribute fields (unlike in JS)

productType = product;

inventoryCount = count;

inventoryPrice = price;

}

// main method

public static void main(String[] args) {

// public = allows main to be accessible everywhere (like ruby public methods)

// static = ?

// void = main method does not return any values

Store lemonadeStand = new Store("lemonade", 42, .99);

Store cookieShop = new Store("cookies", 12, 3.75);

System.out.println(

"Our first shop sells " + lemonadeStand.productType +

" at " + lemonadeStand.inventoryPrice + " per unit.");

System.out.println("Our second shop has " + cookieShop.inventoryCount + " units remaining.");

}

}

**LESSON 5.1 NOTES- ARRAYS**

// ARRAY NOTES

// - holds FIXED number of values of ONE type

// - can hold doubles, ints, booleans, objects, ...

// - indexing begins at 0

// 1) Importing Arrays

// - Arrays package has helper methods like .toString() that prints arrays

// - write this at the top of your file (outside class)

// - don't need this to create general arrays

import java.util.Arrays;

public class ArrayNotes {

// Main (entry point)

public static void main(String[] args) { // can pass in arguments when you run file in terminal

// 2) Creating an array (declaration/initialization in one line)

// - create an array of integers, called nums1:

int[] nums1 = { 1, 1, 2, 3, 5 };

// 3) Arrays.toString() method

// - prints array values

// - need to import java.util.Arrays for this to work

int[] nums2 = { 1, 2, 3 };

Arrays.toString(nums2); //=> [1, 1, 2, 3, 5]

// 4) Accessing Values in arrays

// - use square brackets [ index ], just like JS

double[] prices = { 1.0, 1.5, 2.0 };

System.out.println(prices[0]); //=> 1.0

// 5) Creating an empty array, and itializing it with fixed size

// - create an array of Strings called pets, with a size limit of 3

// - once you do this, you cannot change the size!!

// - values are initialized with NULL!!!

String[] pets = new String[3];

System.out.println(pets[0]); //=> null

// 6) Populating values in an array

// - once array has been declared with size, you can populate values

String[] fruits = new String[3];

fruits[0] = "mango";

// 7) array.length property

// - built in instance field/attribute/state of arrays (like JS)

int[] nums3 = { 1, 1, 1 };

System.out.println(nums3.length); //=> 3

// COMMON ERRORS:

// 1) Can't Delcare an array, then on new line initialize it with { }

// int[] numsError1;

// numsError1 = { 1, 2, 3 }; //=> COMPILE ERROR

// 2) Cant repopulate/initialize

// int[] numsError2 = { 1, 2 };

// numsError2 = { 3, 4 }; //=> COMPILE ERROR

// 3) Can't redeclare variable with same name!

// int[] numsError3 = { 1, 2 };

// int[] numsError3 = { 3, 4 }; //=> COMPILE ERROR

// 4) Can't populate value outside of array size!

// int[] numsError4 = new int[2]

// numsError4[3] = 6; //=> RUNTIME ERROR- out of bounds

}

}

**LESSON 5.2 NOTES- ARRAY LISTS**

// ARRAY LIST NOTES

// - dynamic arrays, can add/remove elements (not fixed size)

// 1) need to import java.util.ArrayList

import java.util.ArrayList;

class ArrayLists {

// Main (entry point)

public static void main(String[] args) {

// 2) DECLARING ArrayLists

// - use angle brackets < >

// - brackets are used for GENERICS = allows us to define classes and objects

// as parameters of an ArrayList

// - can also use: <Double>, <Char>, <String>, ...

ArrayList<Integer> nums1;

// 3) cant use primitives

// ArrayList<int> nums; // COMPILE ERROR

// 4) DECLARING, then INITIALIZING ArrayLists

ArrayList<Integer> nums2;

nums2 = new ArrayList<Integer>();

// System.out.println(nums2); //=> []

// 5) DECLARING, AND INITIALIZING ArrayLists (one line)

ArrayList<Integer> nums3 = new ArrayList<Integer>();

// 6) arrayList.add( val )- ADD METHOD

// - like JS .push method, adds val to end of array

// - only takes in 1 arg

ArrayList<String> nums4 = new ArrayList<String>();

nums4.add("a");

nums4.add("b");

nums4.add("c");

System.out.println(nums4); //=> [ "a", "b", "c" ]

// 7) arrayList.size() = SIZE METHOD

// - .length method doesnt exist for ArrayLists

ArrayList<Integer> nums5 = new ArrayList<Integer>();

nums5.add(3);

nums5.add(4);

System.out.println(nums5.size()); //=> 2

// 8) ACCESSING ArrayList elements- GET METHOD

// - Can't use hard brackets [ ], like for regular arrays

// - use arrayList.get( idx )

ArrayList<Integer> nums6 = new ArrayList<Integer>();

nums6.add(1);

nums6.add(2);

System.out.println(nums6.get(0)); //=> 1

// 9) SETTING/Updating ArrayList elements- SET METHOD

// - Can't use hard brackets [ ] =

ArrayList<Integer> nums7 = new ArrayList<Integer>();

nums7.add(1);

nums7.add(2);

nums7.set(1, 69);

System.out.println(nums7); //=> [ 1, 69 ]

// 10) REMOVING element- .REMOVE METHOD

// - arrayList.remove( idx )

// - returns removed element

ArrayList<String> nums8 = new ArrayList<String>();

nums8.add("a");

nums8.add("b");

nums8.add("c");

System.out.println(nums8.remove(2)); //=> "c"

System.out.println(nums8); //=> [ a, b ]

// 11) REMOVING element- .REMOVE METHOD

// - arrayList.remove( value )

// - returns FIRST instance of the value

ArrayList<String> nums9 = new ArrayList<String>();

nums9.add("a");

nums9.add("b");

nums9.add("a");

System.out.println(nums9.remove("a")); //=> a

System.out.println(nums9); //= [ b, a ]

// 12) FINDING element- .INDEX OF METHOD

// - arrayList.indexOf( value )

// - returns FIRST instance of value

// returns -1 if not found

ArrayList<String> nums10 = new ArrayList<String>();

nums10.add("apple");

nums10.add("pear");

nums10.add("apple");

System.out.println(nums10.indexOf("apple")); //=> 0

System.out.println(nums10.indexOf("mango")); //=> -1

}

}

**LESSON 8 NOTES- INHERITANCE AND POLYMORPHISM**

1) **extends**

- just like JS, used to make a class a child of a parent

[childClassName] extends [parentClassName] { }

- child inherits all of parent instance variables/methods

- also possible to overide parents constructor by rewriting childs

constructor

2) only need one main method, but you must run java app from that file

with the main() method

3) **super( arg )**

- just like JS/Ruby, passes in arg into parent constructor, when new

object is instantiated

4) **private** keyword

- even child cant access parents attributes/methods if private keyword used

- can place before attribute (or method?) name

5) **protected** keyword

- allows child to acces parent attribute/method, but prevents non-child

from accessing parent attribute/method

- can place before attribute (or method?) name

6) **final** keyword

- add infront of method/attribute name, so that method/attribute

can't be over-ridden by any child

7) **POLYMORPHISM**

- allows a child class to share the information and behavior of its parent

- class while also incorporating its own functionality.

- ex. operator + can be used for both doubles and ints

- ex. using a child class as its parent class

instantiate a child object as a member of the parent class

BankAccount moseysAccount = new CheckingAccount(600.00);

8) **Method Overriding**

- use keyword "@Override"

- overrides parent method in child method