Devindu Samarasinghe

OOP IT2030 Lecture 3

Diagram

Description automatically generatedStack and Heap

Stack is where you store the reference and heap is where the actual object and methods are stored

**Two Rules when interchanging stack and heap when creating objects through inheritance:**

Consider: Persons persons = new Employees(); // In this situation, stack can have parent but able to call sub class heap

\*\* This

Employees employee = new Person(); //This is impossible since parent reference cannot call sub class objects.

Person person = **new** Employee(); //When you create inheritance objects, compiler thinks heap object

// is same as the stack reference in compile time.

// BUT During runtime it assigns the sub class object to the super class reference

//This method of creating sub class objects can ensure the inheritance works.

**DATA TYPES**

**In byte**, we can max use 127 or -128 because first bit is used as a sign bit.

1 in first bit considers minus and 0 stands for positive

to describe 128 you need to use all 1 bits in a byte = 111111111, in this situation if we use all bits to show the number, we cannot clarify if the number is positive or negative, therefore to we keep 2^1 bit zero because that defines the sign of the number when its positive. Meaning 01111111 is the maximum a positive byte can go to

THIS IS NOT THE SAME CASE IN -128, in negative numbers sign bit which is 2^1 is always 1, therefore the byte can be fully utilized to

**CONVERSION AND CASTING**

**double** value = 124.654;

**int** no = (**int**)value; //This is called casting

System.***out***.println("Double : " + value);

System.***out***.println("Int of the same after casting: " + (**int**)value);

Multidimensional Arrays

**int** multidimension [][] = **new** **int**[5][6];

**int** multidimension2[][] = **new** **int**[][]{{5,4,6} ,{9,6,8}}; // Remember to not include values

**int** multidimension3[][] = **new** **int**[1][1];

System.***out***.println("Multidimension2: " + multidimension2[1][1]);

Static

**public** **class** StaticEx {

**static** **int** *no* = 10; //STRESS ON THIS PART

**static** {

System.***out***.println("Static Block");

*StaticMethod*();

System.***out***.println("Int No: " + *no*); // If the int isnt static we cannot use this int

} //Static block is always the first to work

//it will only initialize one time

{

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

}

**public** StaticEx() {

System.***out***.println("StaticEx() constructor");

}

**static** **void** StaticMethod() {

System.***out***.println("Static Method");

}

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

StaticEx staticexe = **new** StaticEx();

StaticEx static2 = **new** StaticEx();

}

}

\*\*\*\* Static members can be accessed without a creation of the object, can call it through the class

\*\*\*\* Consider a situation where you create a private static member or an attribute in the class, in this case you need getters and setters to access the static member or the attribute from the class to utilize in the main program.

When creating setters and getters, return of this.attribute or assigning because “this” is often used for reference of object instances of the variables, since it’s used for objects we do not use this for static members or attribute assigning.

**public** **class** person {

**private** String name;

**private** **int** age;

**private** **double** Salary;

**private** **static** String *batchId*;

**static** **public** String getID() {

**return** *batchId*;

}

**static** **public** **void** setID(String batchinfo){

*batchId* = batchinfo;

}

}

**public** **class** MyTestMain {

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

person.*setID*("45IT");

String x = person.*getID*();

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

person p1 = **new** person();

String y = p1.*getID*();

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

}

}

\*\*\*\* IF YOU ARE INSIDE A STATIC METHOD AND TRYING TO UTILIZE A NON STATIC ATTRIBUTE. YOU CANNOT DO THAT, INSTEAD YOU CAN CREATE AN OBJECT WITHIN THE STATIC METHOD AND THEN CALL THE ATTRIBUTE OF THE OBJECT INSIDE THE STATIC METHOD.

**static** **void** show() {

// name = YOU CANNOT DO THIS BECAUSE THIS ATTRIBUTE IS A NON STATIC ATTRIBUTE

// TO ACCESS THE NAME YOU NEED TO CREATE AN OBJECT INSIDE THE STATIC METHOD

//therefore

person person1 = **new** person();

person1.name = "Devindu";

}

\*\*\*\*\*\*\*final is used as CONST, YOU NEED TO INITIALIZE IT RIGHT ON THE SPOT OF DECLARING THE VARIABLE

**class** person{

**private** String name;

**private** **int** age;

**private** **double** sal;

**public** person() {

name = **null**;

age = 0;

sal = 0;

}

**public** person(String name,**int** age,**double** sal) {

**this**.name = name;

**this**.age = age;

**this**.sal = sal;

}

**public** **void** show() {

System.***out***.println("Name: " + name + "\nAge: " + age + "\nSalary: " + sal);

} //This is casual

**public** **void** show1(person x) {

System.***out***.println("Name: " + x.name + "\nAge: " + x.age + "\nSalary: " + x.sal);

} //This is a method to show the pass by reference usage

//In this method, we can pass an object as a reference as the argument. WHEN AN OBJECT IS CREATED, its information is stored in

//memory and it has a memory address. when you pass the object as an argument. The memory address will be called, but for the

//variables to be called out. method should call the object's properties inside the method. Eg: "x.name"

}

**public** **class** MainFile {

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

person p1 = **new** person("Devindu",45,45000.0);

//p1.show();

person p2 = **new** person();

p2.show1(p1);

}

}

Overloading and Overriding

Overloading

Overloading occurs when there are methods with different signatures

* Same method name
* Different parameters

\*\* It should be noted that different data return types while having the same method name and different parameters does not mean overloading.

Eg: A = public int show(int x, int y){}

B = public int show(int z, int a){}

In this situation A and B are not overloaded, because even though the parameter names are different, it still takes the same data type. Which means that the arguments are not really different

Eg: A = public int show(int x, int y){}

B = public int show(int x, int y, int z){}

This is an overloaded method

Eg: A = public void show(int x, double y){}

B = public void show(double x, int y) {}

This is overloaded since the parameter order is different