CLASSES, OBJECTS, [redacted]
JVM Data Areas

Date 20/10/2022

⇒ Today's Topic of discussion :— ① Introduction to OOPS (classes and objects) → Practical
② Types of variables
Division-1 ⇒ a) primitive variable, b) reference variable.
Division-2 ⇒ a) Instance variable, b) local variable, c) static variable

③ JVM Area for execution :- (in RAM)
a) method Area, b) Heap area, c) stack area, d) PC-Register
e) Native method area

1) OOPS :- (Practical)

⇒ It is actually theory concept, which is implemented by many programming language like c++, java, python, ----
⇒ Any real time problem can be solved if we follow oop's principle. In oop's, while solving the problem →
(Practical) ↓ (i) we need to first mark [redacted] the objects. ←
(ii) Every object we mark should have 2 parts →
a) HasPart / fields / Attributes / (store the information as ~~variables~~.) or instance Fields
b) Does-part / behaviours (represent them as method)
(iii) To represent an object, first we need to have a blueprint of an object (class required)
(iv) we use "new" keyword / reserved word to create an object for a ~~B~~lueprint (class).
(v) Every object should always be in constant interaction.
(vi) useless object does'not exists.

Q) What is object ?

⇒ Physical existence of an element we say as object.
eg: book, car, pen, computer, Phone, Dog, etc...
- Realtime example : BookMyShow
Objects :- Person, Ticket, Cinemahall, Chair, Screen, Payment gateway...

⇒ NOTE :- • Software means → collection of many programms.
• Program means → set of instructions.
• TO write instructions we need to have a language.

Q) What is HAS-Part and what Does-Part of an object represents?

⇒ HAS-PART ⇒ indicates what it can hold.
Does-PART ⇒ indicates what it can do.

eg:- Student

⇒ sid, name, age, gender, email, address. (variables / identifiers)

⇒ play, study, drink, sleep, run, score, etc (methods)

Q) What is Blue print of java and how to represent it?

⇒ In java to represent a blue print we have a reserved word called "class".

(*) So, Conventions followed by java developers while writing a class is:—

a) classname should be in "PascalConvention / Case"

eg:- BufferedReader, FileReader, Main, String etc..

b) Variables are represented in "camelCase"

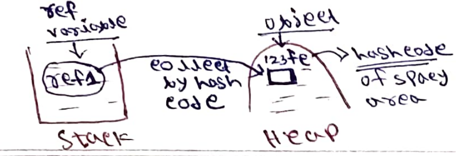
eg:- regNo, firstName, length, javaFullStack, etc---.

c) methods are represented in "camelCase"

e.g:- toUpper(), toLower(), toString(), nextInt()....

Eg#1:-

```
// Blue print of student object
class Student {          // Student → PascalConvention.
    // HAS-Part ---> // camelCase Convention.
    int sid;
    String name;
    int age;
    char gender;
    String address;
    // Does-Part ---->  // camelCase Convention
    void play(){ }
    void study(){ }
    void drink(){ }
    void sleep(){ }
}
```



⊛ ⇒ To creat an object in java we use "new" keyword.

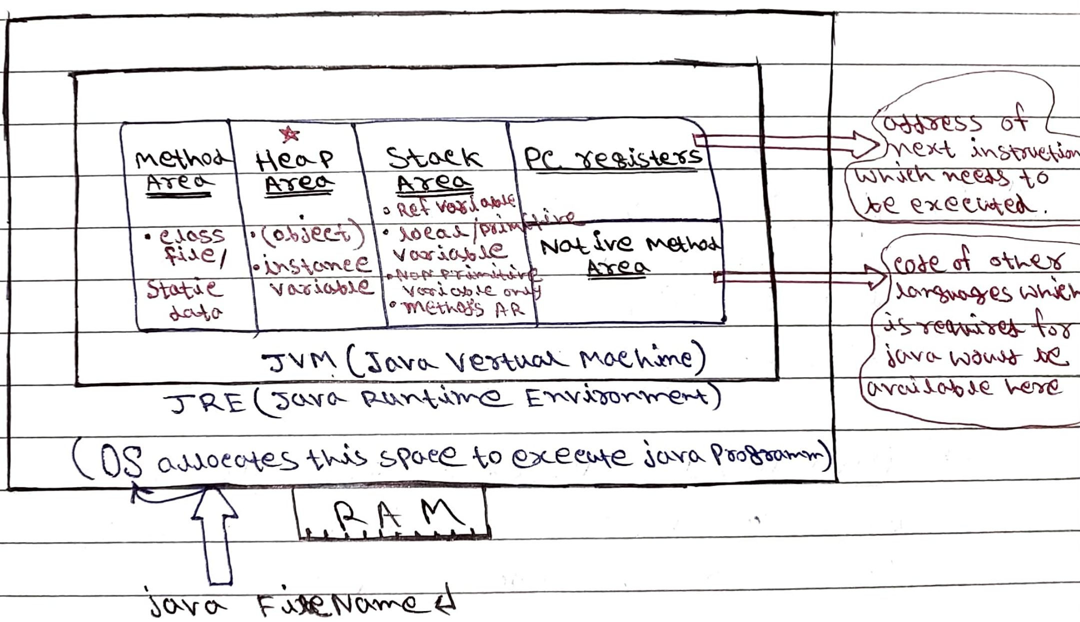
Syntax:- ClassName variable = new ClassName();

⇒ "new" → it is a signal to jvm to create some space for the object in the heap area.

explanation ⇒ Whenever the className by java ClassName (common), we inform the classname, JVM create the object in heap, and [redacted] sends the "hashcode" to the user. User should collect the hashcode through "reference variable". and reference variable will store in stack area.

⇓

⊛ JVM DATA AREA :- (jvm Architecture)



⇒ now, During execution at the runtime to execute a java file a space is given in RAM for the JRE. OS gives the space to RAM. next inside that JRE there a space allocated by JRE that is JVM. Inside JVM

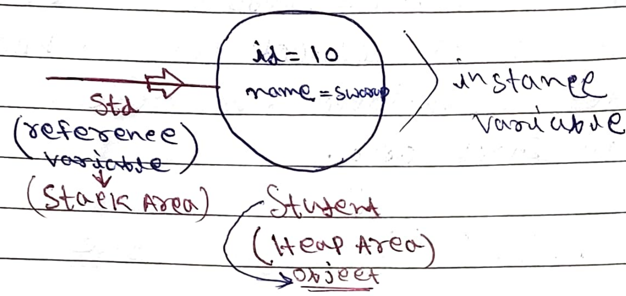
(Datatype of Student class)

⇒ Student std = new Student();

⇓ behind the scenes

(hashcode of the object)

123457ACEF



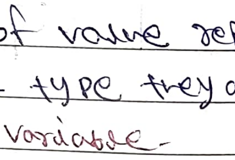
(★) (2) Types of variables :-

int a = (10); → primitive value holds
(a → primitive variable)

Student std = new Student();
(std → ref. variable)

(i) **Division 1 :-** Based on the type of value represented by a variables are divided into 2 type they are:- 1. primitive variables, 2. Reference variable.

⇒ 1. **Primitive variables :-** (Directly Stored in Stack Area)
Primitive variables can be used to represent primitive values. primitive means it can't be break into other type on value (NON-Breakable).
ex: int x = (10);



⇒ 2. **Reference variables :-**
Reference variables can be used to refer objects.

ex:- Student (std) = new Student();

(ii) **Division 2 :-** Based on the name of memory location they are three type :- (1) Instance, (2) local (3) Static variable or (declared location)

=> (1) Instance variable :- (Stored in Heap area)

Basic → If the variable is declared inside the class, but outside the methods such variables are called as instance variables. (we can use it in any method)

or

Advanced → If the value of the variables changes from object to object then such variables are called as "Instance variables"

Ex:- Student std1 = new Student(); // id = 10, name = swarup.
Student std2 = new Student(); // id = 7, name = mantal
std1.id = 10; std1.name = "swarup";

I.Q) When will the memory for instance variable be given?

Ans:- only when the object is created JVM will creat a memory and by default jvm will also assign the default values, based on the datatype of the variable.

eg:- int → 0, float → 0.0f, boolean → false, char → , String → null, ... (default value)

primitive variable can't be null

→ Note:- scope of instance variable would be available only when we have reference pointing to the object. If the object reference becoms null, then we can't access "instance variables".

(*) Key Points about instance variable:-

(i) If the value of a variable is varied from object to object, such type of variable called instance variable.

(ii) For every object a separate copy of instance variable will be created.

(iii) Instance variables will be created at the time of object creation and destroyed at the time of object destruction. hence, the scope of instance variables is exactly same as scope of object.

(iv) Instance variable will be stored on the Heap as the part of objects.

(v) I.V. should be declared with in the class directly but, outside of any method or block or constructor.

(vi) Instance variables can be accessed directly from Instance area. But cannot be accessed directly from Static area.

(vii) But using object reference we can access instance variables from static area.

Q.8#1
```
Public class Test {
    boolean b; //instance variable
    Public Static void main(String[] args) {
        Test t = new Test();
        S.o.t.println(t.b); // code will run ans will = false (default value of boolean)
    }
}
```

Q.8#2
```
Public class Test {
    int i = 10; //(instance variable)
    // Sout.Pln(i); //C.E
    P.S.V. main(String[] args) {
        //S.O.t.Pln(i); //C.E ⇒ I.V can't be accessed directly in static context.
        Test t = new Test(); //object created i=10 is stored in heap area.
        S.O.Pln(t.i); // 10
    }
    Public void method2re() {
        // inside I.method I.V can be directly accessed
        S.O.Pln(i); //10, bez it is an I.V
    }
}
```

⇒ ①. Local variable:-

(i) Variable which are created inside a method are called local variable and memory for those variables will be created in the StackArea.

(ii) During the execution of the method the memory for L.V will be given, and after the execution of the method the memory for L.V. will be taken out from stack area.

(iii) L.V. default value will not be given by JVM. programmer should give the default value.

(iv) If the programmer doesn't give default value and if he uses the variable inside the method the program would result in "C.E".

Key points of Local variables:-

⇒ Some times to meet temporary requirments of the programmer we can declare variables inside a method or block or constructors such type of variables are called. Local variables or Automatic variables or Temporary variables or Stack variables.

⇒ It is highly recommended to perform initialization for the L.V at the time of declaration at least with default values.

e.g#1)
```
Public class Test {
    P.S.V.m(S[] args){
        int i = 0;
        for (int j=0; j<3; j++){
            i = i+j; or (i+= j)   // i = 3 and j = 0,1,2 so ②
        }
        S.O.Pln(i); // valid ✓
        S.O.Pln(j); // C.E: 'j' variable not declared ✗
```

e.g#2)
```
class Test {
    P.S.V.m(S[] args){
        try {
            int i = Integer.parseInt("ten");
        }                          // i declares in only try scope
        catch(NullPointerException e){ not in catch so C.E.
            S.O.Pln(i); // CE: 'i' not declared.
    } }
```

e.g#3)
```
class Test {
    P.S.V.m(S[] args){
    int x;
    S.O.Pln("hello"); // hello (valid) bcz x not used anywhere
} }
```

e.g#4)
```
class Test {
    P.S.V.m(S[] args){
    int (x);
    S.O.Pln(x); // C.E: 'x' not initialized.
```

Code snippets :- ⭐ switch (condition) {
case label1; stmt-1;
case label2; stmt-2;
default : stmt-n;
}

⇒ So, label in switch should be "compileTime constants", meaning the value should be known to compiler other wise C.E>

(1) Public class Test {
P.S.V m (S[] a){
int x = 10;
int y = 20;
switch (x) {
⭐ case 10; S.o.Pln ("hello");
break;
case y; S.o.Pln("hiee"); // CE: 'y' value is not compileTime constant.
break;
} } }

Ans ⇒ C.E

(2) P C Test {
P.S.V.m(S[] a){
int x = 10;
⭐ final int y = 20; // final means compiler will get to know the value and compiler treats it as "compileTime Constant"
switch (x){
case 10: S.o.Pln("hello");
break;
case y: S.o.Pln("hiee"); // code will run and give output as hello.
break;
} } }

(3) int x = 10;
switch (x+1) {
case 10;
case 10+20;
case 10+20+30: } // all valid (no output)

(4) byte x = 10;
switch (x){
case 10:
case 100:
case 1000: // C.E: possibly loss of precession from byte to int
}

(5) byte x = 10;
switch (x+1){ // byte + int = int, so, switch (int)
case 10;
case 100:
case 1000: // all valid Ans:- No output
}