UNIT-2 PROGRAMMING CONSTRUCTS

Basics:

1. Identifiers
2. Keywords
3. Data types
4. Variables
5. Literals
6. Operators.
7. Control structure
8. Type casting
9. Classes & objects.

* Identifiers:

It is a name given to a variable, array, class, interface, packages, method etc. for identifying purpose.

Rules:

1. It can be the combination of alphabets, digits, underscore (_), dollar ($)
2. Should not start with digit.
3. Keywords cannot be used as identifiers.
4. Case sensitive.
5. No restriction on the length of the identifier.
6. We can use predefined classes and predefined interfaces.

Only digit is not allowed in first place.
$ & _ can be first letter of a word.

Program:

```
class IdentifierDemo
 {
   Public Static void main (string args[])
    {
      int string = 10;
      System.out.Println(String);
      float Runnable = 20.7f; → for floating pt:
      System.out.Println(Runnable);
    }
 }
```

O/P: 10.

O/P: 20.7

Valid / Invalid Identifiers:

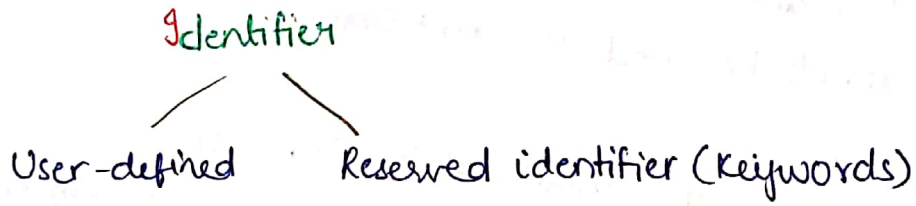
Sum_valid - valid
0 sum - Invalid (Rule 2 fails)
sum # details - Invalid (R 1 fails)
sum details - Invalid (R1 fails)
$ sum - valid
_sum - valid.
sum 123 - valid
float - invalid.

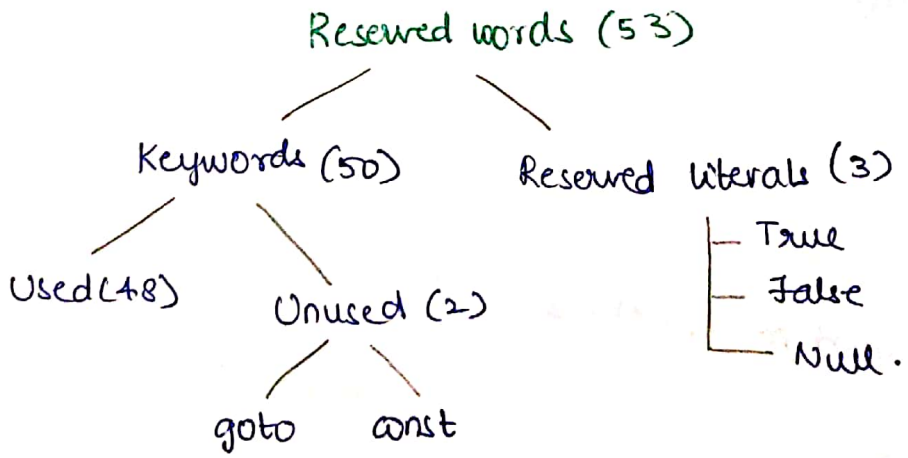
23/12/19

* Reserved words / Keywords:

There are 53 keywords in Java.

→ It is an identifier which is meant or reserved for some special purpose.

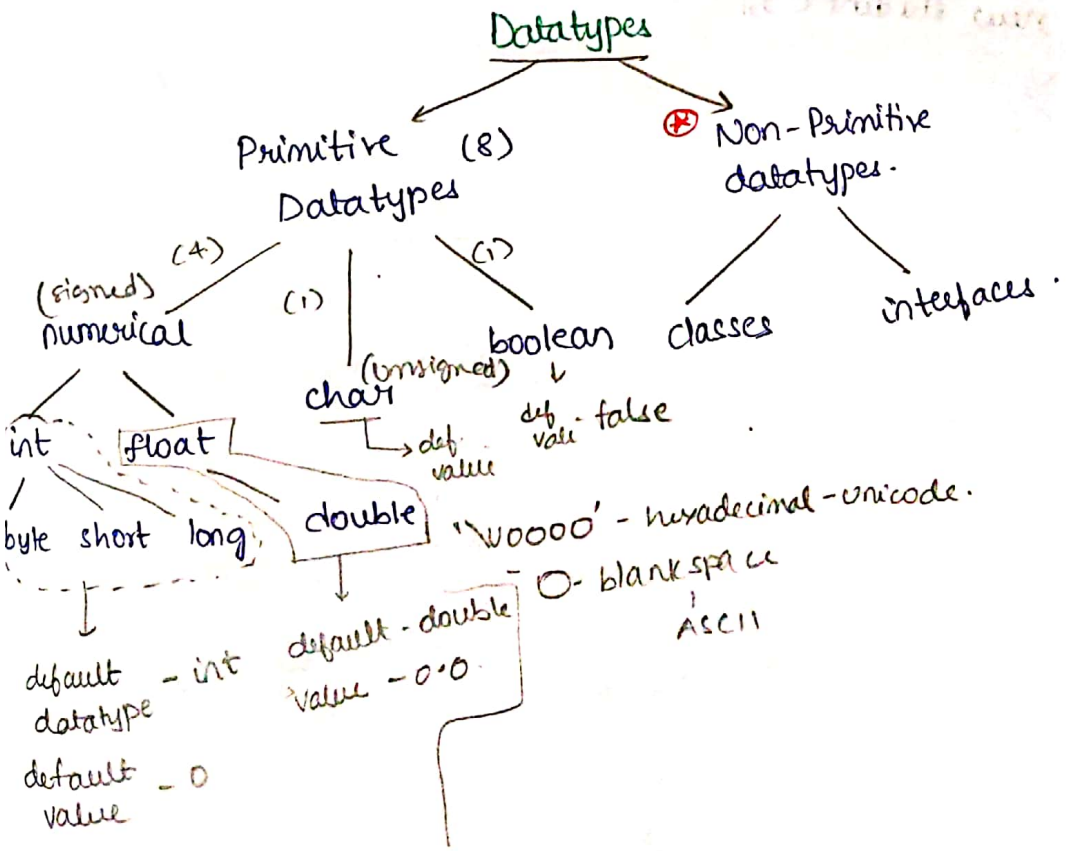




Keywords (48)

Primitive datatype (8)	Flow control (11)	Modifiers (11)	Class (6)	Object (4)	Exception (5)	others (3)
1. byte	1. if	1. private	1. interface	1. new	1. try	1. enum
2. short	2. else	2. protected	2. package	2. instanceof	2. catch	2. void
3. int	3. do	3. public	3. class	3. this	3. finally	3. assert.
4. long	4. while	4. static	4. extends	4. super	4. throw	
5. float	5. for	5. abstract	5. implements		5. throws	
6. double	6. switch	6. final	6. import			
7. char	7. case	7. strictfp				
8. boolean	8. default	8. volatile				
	9. break	9. transient				
	10. continue	10. synchronize				
	11. return.	11. native				

* Data types :



* Classes and Objects:

Class is a template or blueprint of an objects.

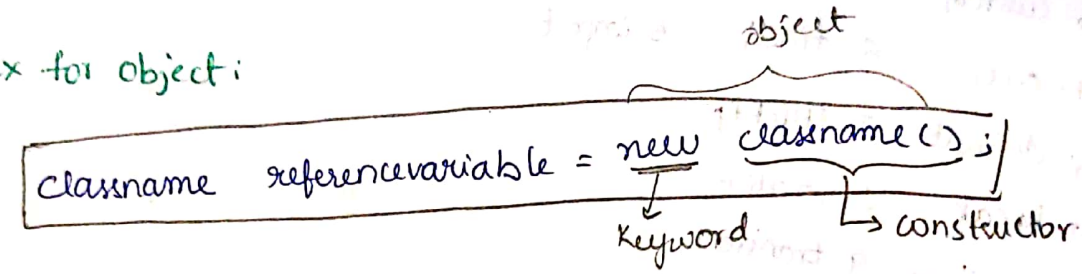
Object is an instance of the class.

Syntax of class:

```
class classname
{ state - properties
  behaviour - methods
}
```

→ Members of the class are accessed by the objects.
→ First class is created then, objects are created.

Syntax for object:



Eg:

```
student s1 = new student();
```

constructor

Eg:

```
s1.rno = 1;
s1.marks = 100;
s1.display();

student s2 = new student();
s2.rno = 2;
s2.marks = 98;
s2.display();
```

→ objects names cannots be same.
→ class is created only once, objects can be created req.

Java program with one class.

```
import java.lang.*;
class student extends object
{
  int rno;
  float marks;
  void display()
  {
    System.out.println("rno is " + rno);
    System.out.println("marks are " + marks);
  }
public static void main(String args[])
{
  student s1 = new student();
    s1.rno = 1;
    s1.marks = 100;
    s1.display();
  student s2 = new student();
    s2.rno = 2;
    s2.marks = 98;
    s2.display();
  }
}
```

Java Program with 2 classes:

```
import java.lang.*; // default
class student extends object
{
 int rno;
 float marks;
 void display()
 {
  system.out.println("rno is" + rno);
  system.out.println("marks are " + marks);
 }
```

— (underline)
↓
optional

```
class StudentDemo extends object.
{
  Public static void main (String [] args)
  {
    Student s1 = new student ();
    s1.rno = 1;
    s1.masks = 100;
    s1.display ();
   Student s2 = new Student ();
    s2.rno = 2;
    s2.masks = 98;
    s2.display ();
  }
}
```

30/12/19

java - filename.java
compile - javac filename.java
execute - java filename.

Data types:

byte - 1
short - 2
int - 4
long - 8

float - 4
double - 8
char - 2
boolean

* Primitive datatypes : (8)

1. byte :

size - 1 byte - 8 bits
range - $(-2^7$ to $2^7 - 1) = -128$ to 127
min value - (-128)
max value - 127

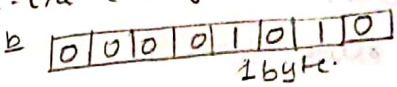
Example :

1) byte b; // 1 byte.

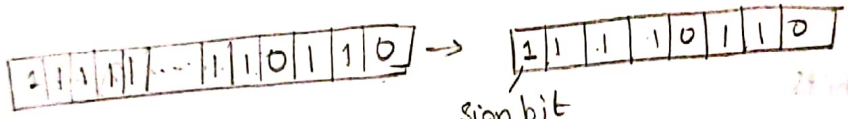
b is a variable that ~~stor~~ is byte datatype.

Size
-2^{n-1} to $2^{n-1}-1$

2) byte b = 10; → def datatype - int (4 bytes)



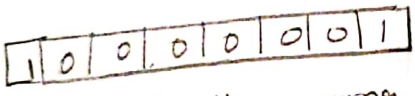
3) byte b = -10; → int 32



10:0
0:0 1
1
1 1 1 0 1 1 0

128 64 32 16 8 4 2 1
1 0 0 0 0 0 0 1

4) byte b = 129.



→ ✗. (byte cannot store)

compile time error.

Numerical, char, Bookan are incompatible datatypes.

5) byte b = 10.5.
byte ↙ double

cannot store → compile time error.

possible loss of precision
found : double
required : byte.

6) byte b = 'A';

compile time error
incompatible datatype.
found: char
required: byte.

2. Short - 2 bytes.

size - 2 bytes - 16 bits

range : -2^{15} to $2^{15}-1$ = -32768 to 32767

min : -32768

max : 32767.

2^{11} - 2048

2^{12} - 4096

2^{13} - 2

4096 | 8
32768 | 8

Eg:

1. short s;
2. short s = 100;
3. short s = -100;
4. short s = -32769 // error - possible loss of precision.
5. short s = 'a' // error - incompatible error
6. short s = 10.5 // error. possible loss of precision.

2 | 100
2 | 50 - 0
2 | 25 - 0
2 | 12 - 1
2 | 6 - 0
2 | 3 - 0
1 - 1

1100100

3. int - 4 bytes

size : 4 bytes - 32 bits

range : -2^{31} to $2^{31}-1$

min : -2^{31}

max : $2^{31}-1$

Eg:

1. int a;
2. int a = 26;
3. int a = -27;
4. int a = 20.5; // possible loss of precision.
5. int a = 'A'; // Incompatible error
6. int a = true; // Incompatible error

4. long:

size - 8 bytes - 64 bits
range - -2^{63} to $2^{63}-1$
min - 2^{-63}
max - $2^{63}-1$

Eg:

```
long l // def value - 0.
long l = 100;
long l = 2147483649
```

(def datatype is int, hence, it cannot store the value.)

```
long l = 2147483649l;
```

→ If the value is within the range of l, then no need to suffix the value with 'l' or 'L'.

→ If the value is outside the range of l, then we need to suffix the value with 'l' or 'L'.

6/1/2020

5 float:

size - 4 bytes
range - ~~31~~ -3.4e3.8 to 3.4e3.8
min - -3.4e3.8
max - 3.4e3.8
single precision.
5 to 6 decimals of precision.
less accuracy

6.233.

Eg:
```
float f1;
System.out.println(f1);
```
O/P: 0.0

```
float f1 = 9.3;   // double
System.out.println(f1);
```
— Error

6. double:

size - 8 bytes
range - -1.7e308 to 1.7e308.
double precision.
14 to 15 decimals of precision.
more accuracy.

Ex:
```
double d1 = 9.3;
double d1 = 9.3d;
```

7. Char : (Unsigned -ve)

Size - 2 bytes

default value '\u0000' or 0.

range - 0 to 65535.

Supports Unicode (Universally accepted code).

It takes hexadecimal notation.

Eg:

- char c1;
  System.out.println(c1);
  O/P: blank space or '\u0000'.
- char c2 = 'A';
- char c3 = true;
  incompatible errors
- char c1 = 'ab'; // compiler error
- char c2 = "abc"; // compiler error
- char c = $A; // compiler errors.

C	Java
1 byte ↓ 0 to 255	2 bytes ↓ 0 to 65535
ASCII	Unicode
→ only Eng alphabets	All the languages.

→ ASCII code is the subset of Unicode.

16 | 65 | 41
64

16) 65 (4
64
1) 4 (4

8. Boolean :

default value - false.

size
range } Not applicable (NA)
↳ virtual machine dependent.

It takes only 2 values - true, false

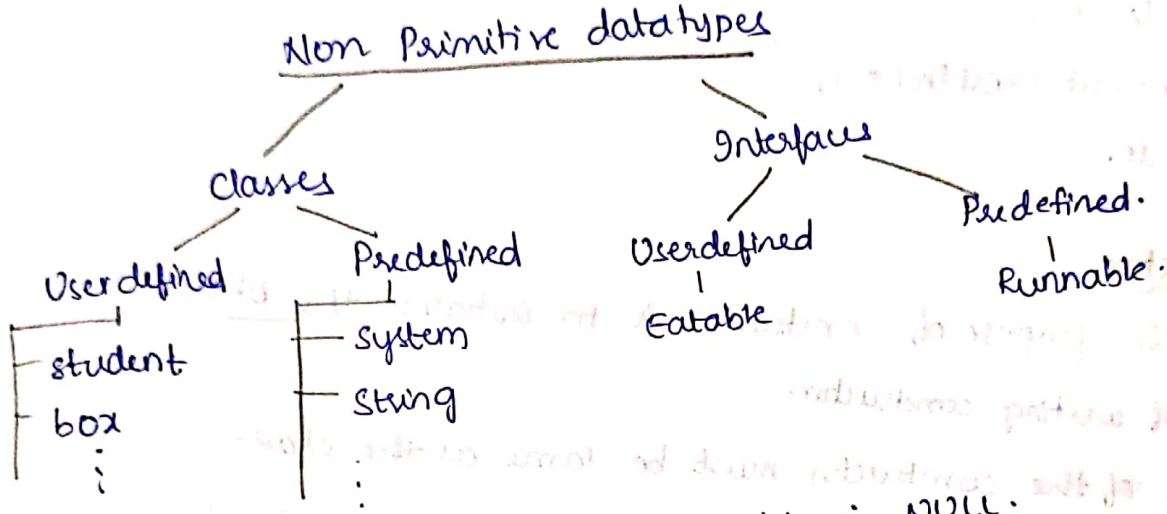
Eg:

- boolean b;
  System.out.println(b); // O/P: false.
- boolean b1 = true;
  System.out.println(b1);
  O/P: true.
- boolean b = True; // error
  Only lower case letters are allowed.

- boolean b1 = 'A'; // Compiles error
  ↓ incompatible error.
- boolean b = 10.9; // compiles error
  ↓ incompatibe error.

* Non-Primitive Datatypes :

Classes — Interfaces.



→ default value for any reference variable is NULL.

Classes:

Ex: Student s1;

s1
NULL

- student s1 = new student(); // creating object.
  - student → datatype
  - s1 → refrence variable → holds the address of the object
  - new → assignes memory to the object
  - student() → variable

Predefined classes:

- String s1;
  O/P: NULL.
- String s1 = new String ("ramu");

s1 | 200 → "ramu"

Interfaces:

For interfaces, we can't create objects, we can only create reference variables.

Eg: Runnable r1;
System.out.println(r1);
O/P : null.

userdefined.

Eg: Eatable e1;
System.out.println(e1);
O/P : null.

* Constructor:

The main purpose of constructor is to initialize the objects.

Rules of writing constructor.

- Name of the constructor must be same as the class.
- No return type.
- syntax: classname ([parameter-list])
{
Initialize the objects.
}

Eg: Student()
{
instance variables { s.no = 1;
marks = 100;
}

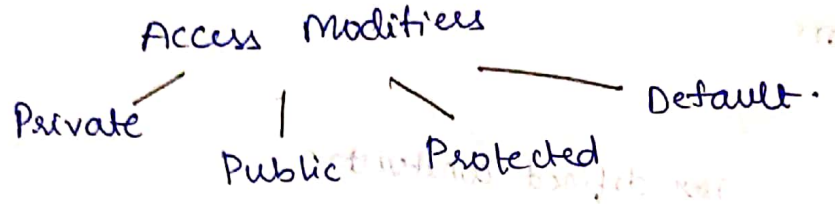
- Difference b/w Method and Constructor:

Constructors donot have return type
Methods must have return type.

- Method name may or may not be same as class name.

Eg: // Method
int Student()
{ =
}

- The only allowed modifiers for the constructor are: Access Modifiers.



Eg:
Student()
{ // no error
}

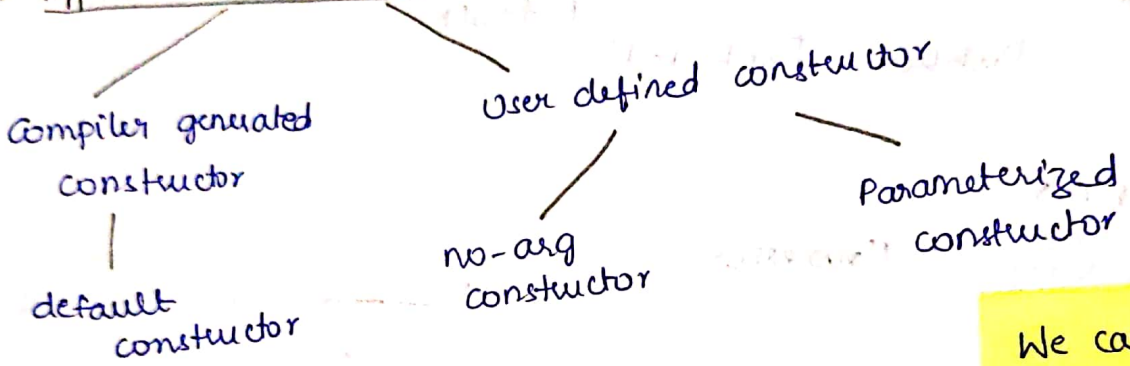
- public Student()
{ // no error.
}

- static Student()
{ // error
}

- final Student()
{ // error.
}

- void Student()
{ // method.
}

⇒ Types of Constructors:



- Default constructor
- User-defined no-arg constructor
- User defined Parameterized constructor.

We cannot create objects for Interfaces.

8/1/2020

Summary of the Primitive datatypes

Data type	Size (bytes)	range (-2^{n-1} to $2^{n-1}-1$)	signed / Unsigned	wrapper class (Predefined)
1. byte	1 (8 bits)	-2^7 to 2^7-1 = -128 to 127	signed	Byte
2. short	2 (16 bits)	-32768 to 32767	signed	Short
3. int	4	-2^{31} to $2^{31}-1$	signed	Integer
4. long	8	-2^{63} to $2^{63}-1$	signed	Long
5. float	4	-3.4e3.8 to 3.4e3.8	signed	Float
6. double	8	-1.7e308 to 1.7e308	signed	Double
7. char	2	0 to 65535	Unsigned	Character
8. boolean	NA	NA	~~signed~~ (NA)	Boolean.

datatype variable [= value] ;

* Variables:

Based on the datatype

Variables
- Based on datatype
- Based on position

Var

Primitive variables

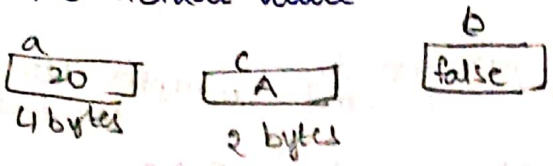
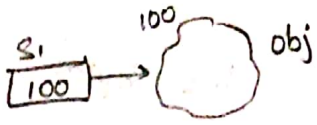
1) A variable which is declared with any one of the 8 primitive data types is called primitive variable.

Eg: int a;
char c;
boolean b;
int a=30;
boolean b= false.

Non-primitive / Reference variables.

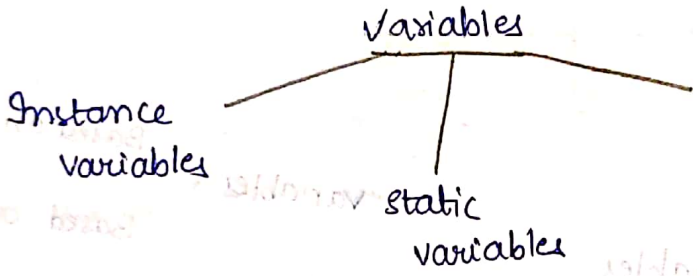
1) A variable which is declared with any one of the two non-primitive datatypes. (class or interface) is reference variable.

Eg:
- Student s1 = new student();
  (Student: datatype; s1: ref variable; new student(): value.)
- String s1 = new String ("ramu");
  (s1: ref → ref variable)
- Runnable r1;
  (r1: ref variable)
- Eatable e1;
  (e1: ref variable)

2. Primitive variables will hold the actual value. 	2. Reference variables will hold the addresses of the object. 
3. The memory is allocated during compilation time hence, it is static.	3. Memory is allocated during the run time using new operator.
4. Static Memory allocation.	4. Dynamic Memory allocation

Memory
- Stack → local
- Heap → Instance
- Method area → static

Based on the Position:



Instance Variables	Static Variables	Local Variables.
1. They are placed inside the class and outside the constructor or method or block. and not declared with static keyword.	1. They are placed inside the class and outside the method or block or constructor and is declared with static keyword.	1. They are placed inside the class and inside the method or block or constructor

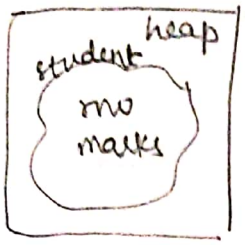
Eg: class student
{
int rno; // primitive var
float marks;
String name; // ref variable
};
	Eg: Class student
	{
	int rno; // primitive & instance
	static int collegecode = 24;
	String name;
	static String clgname = "GRIET";
	};
		Eg: class student
		{
		int rno;
		void display()
		{
		int a; // local & primitive variable
		.
		.
		}
		}

2. Memory is allocated for instance variables inside the objects.

Objects are stored in the heap memory.

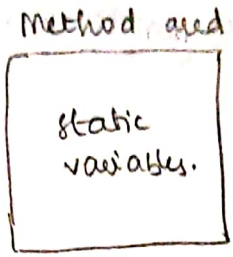
∴ memory is allocated in heap memory.

Eg:

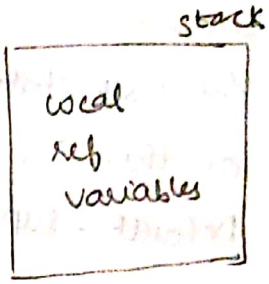


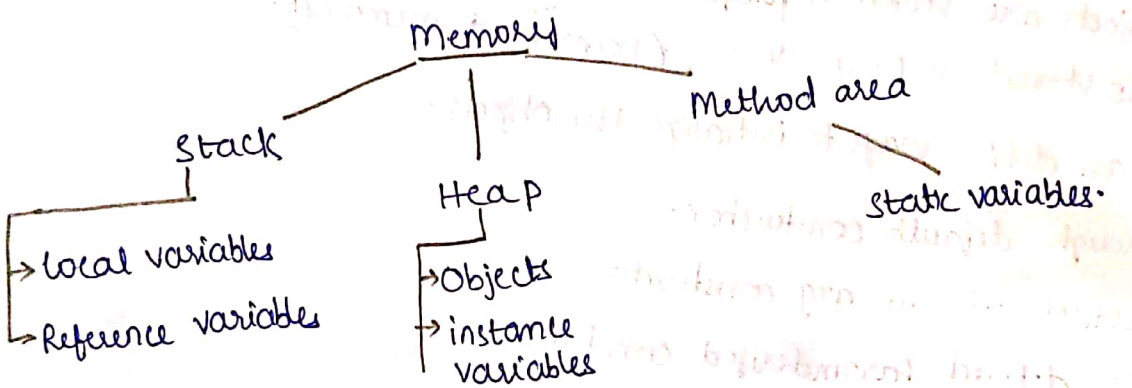
Memory is allocated for instance variable for each object.

2. Static variables are stored in Method area since, memory is allocated for the static variables only once.

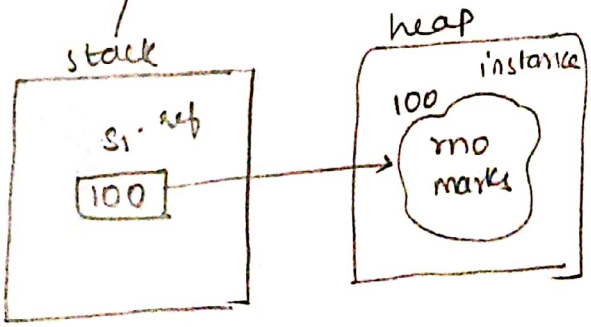


3. Memory for local variables is allocated inside the methods. Memory is allocated in stack area.





Eg: Student s_1 = new student()



9/1/2020 What is constructor & explain diff types of constructors; Q4 (Rules)

* Constructors:

Rule 1: Name of the constructor = name of the class.

Rule 2: Donot have return type.

⋮

Rule : The first statement in each constructor is either super() or this().

Default : super()

* Initialization of an object:

→ Inside each object, instance variables of the class will be stored.

→ Initialization of object is nothing but initialization of instance variables.

→ Objects are stored in heap memory ∴ instance variables are also stored in heap memory (permanent memory).

There are diff. ways to initialize the objects:

- Through default constructor.
- Userdefined no-arg constructor.
- User defined Parameterized constructor
- Direct initialization
- Copying the object
- Using reference variables
- Using no arg method
- Using parameterized method.

Q1

Default Constructor:

If there is no constructor in the class, then the compiler will provide default constructor for the class during the compilation time. (or) if there is atleast one constructor in the class then, compiler will not provide default constructor for the class.

→ Default constructor is no arg constructor.

→ Default constructor will assign default values for the instance variables based on the datatype.

Syntax:

```
Student ()
{
  // default values are assigned to instance variables.
  .
}
```

Program: Default Constructor

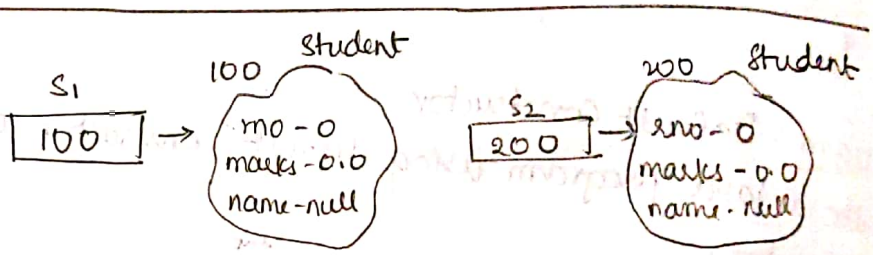
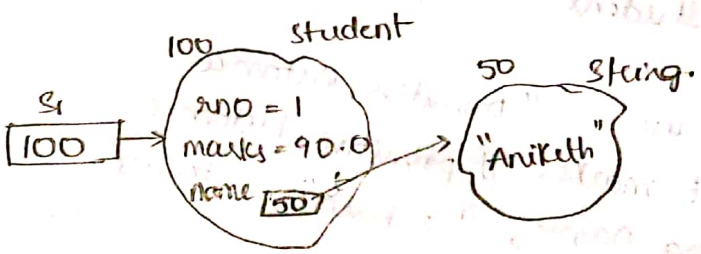
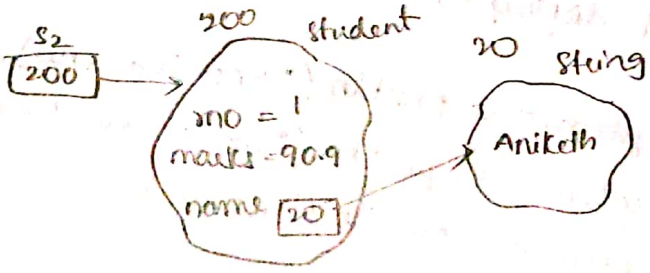
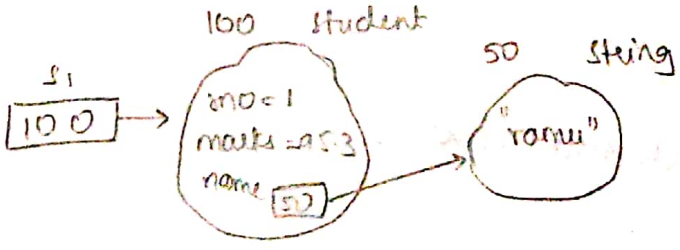
Write a java program using default constructor or compiler generated constructor.

4-member.

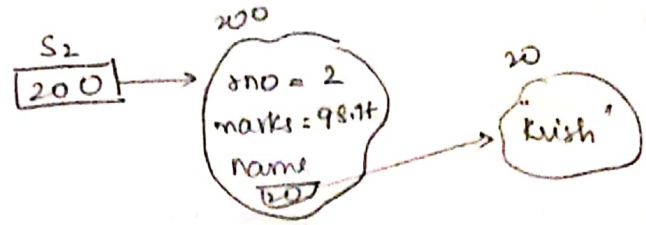
```
Class Student
{
  int rno;    // primitive, instance
  float marks; // primitive, instance
  string name; // reference, instance.

  void display ()
  {
    System.out.println ("rno is" + rno);
    System.out.println ("marks are " + marks);
    System.out.println ("name is " + name);
  }
}

// defaut constructor
Student ()
{ rno = 0;
  marks = 0.0;
  name = null;
}

class StudentDemo
{
```

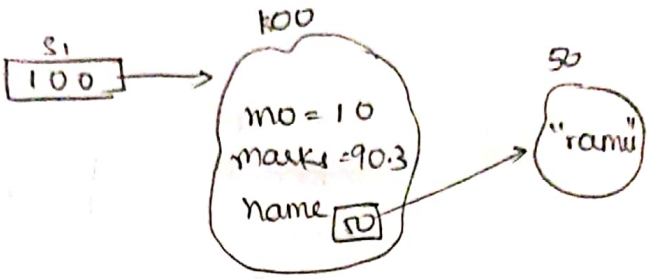
```
public static void main (String args[])
{
    Student s1 = new student();
    Student s2 = new student();

    s1.display();
    s2.display();
  }
}
```

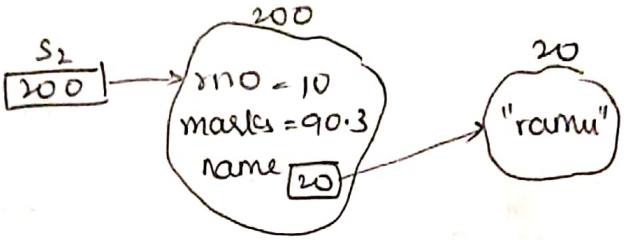
Save the file with StudentDemo.java

Program	Initialization of Object / Instance variables.
Default Constructor.	
User defined No arg constructor	 
Userdefined Parameterized constructor	

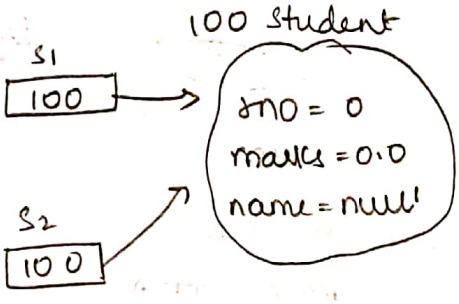
Direct initialization



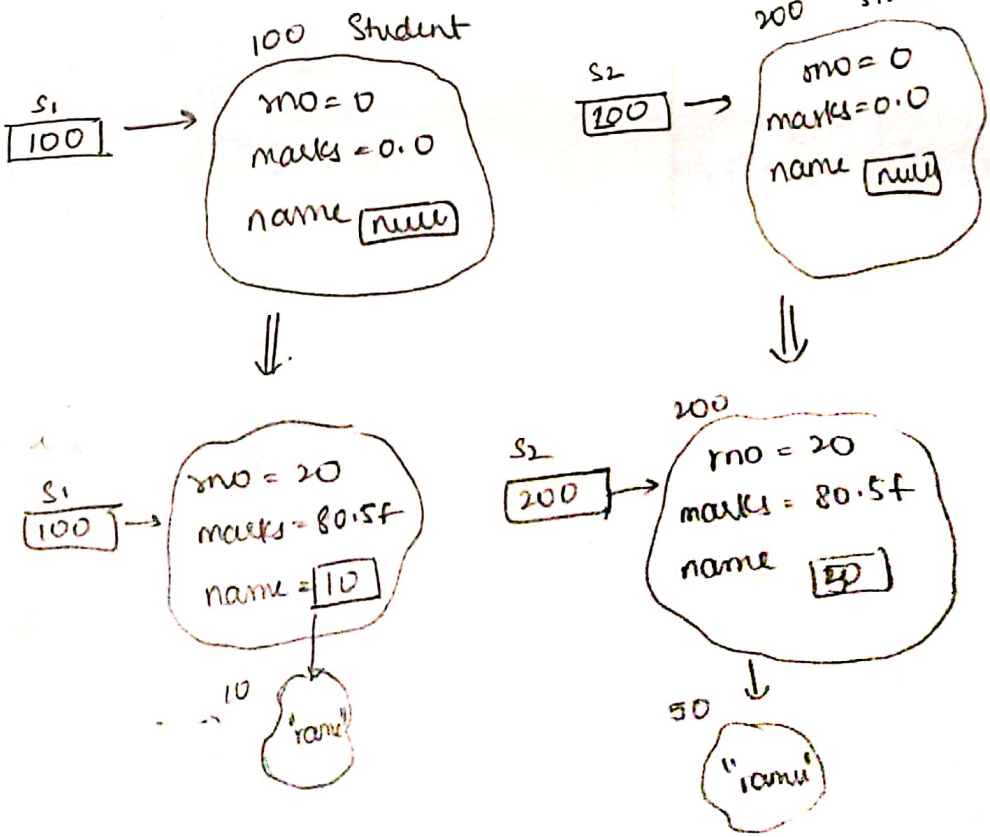




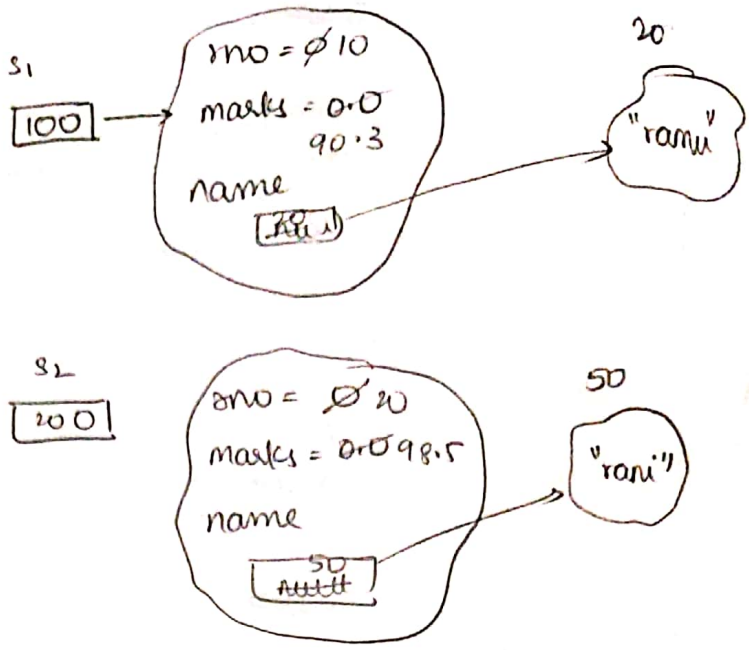
Copying the object.



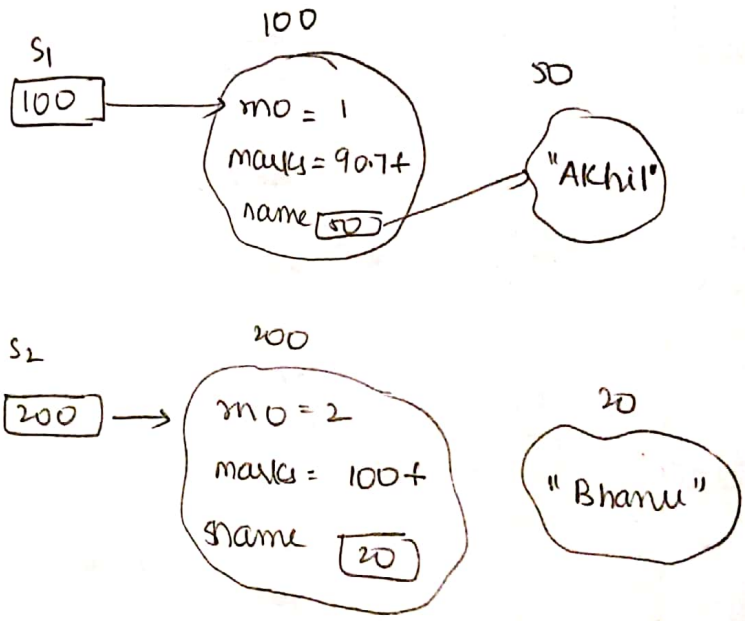
No-args method.



Parameterized method.



Reference variables.



82

Program: User-defined no-arg constructor.

```
Class Student
{
  int rno;
  float marks;
  string name;
  Student () // User-defined no-arg constructor.
    {
      rno = 1;
      marks = 90;
      name = "Aniketh";
    }
  void display
   {
    System.out.println ("rno is " + rno);
    System.out.println ("marks is " + marks);
    System.out.println ("name is " + name);
   }
}
Class StudentDemo
{
 Public Static void main (String args[])
 {
   Student S1 = new Student ();
   Student S2 = new Student ();
   S1.display ();
   S2.display ();
 }
}
```

O/P: rno is 1
marks is 90.0
name is Aniketh
rno is 1
marks is 90.0
name is Aniketh.

- Drawback of Userdefined No arg constructor:

All the objects are given same value.

Q3

Program: User-defined Parameterized Constructor.

```
Class Student
{
  int sno;
 float marks;
 string name;
 Student (int sno1, float marks1, string name1) // Userdefined
                                                  Parameterized constructor.
 {
   sno = sno1;
   marks = marks1;
   name = name1;
 }

 void display
 {
  System.out.println ("sno is " + sno);
  System.out.println ("marks is " + marks);
  System.out.println ("name is " + name);
 }
}

Class StudentDemo
{
public static void main (string args[])
{
 Student s1 = new student (1, 95.3f, "ramu");
 Student s2 = new student (2, 98.9f, "krish");
 s1.display();
 s2.display();
}
```

Program: Direct Initialization.

```
class Student
{
   int rno=10; // direct initialization.
   float marks = 90.3f;
   string name = "ramu";
   void display()
   {
    System.out.println("rno is" +rno);
    System.out.println("marks is" + marks);
    System.out.println("name is" + name);
   }
}
Class StudentDemo
{
  public static void main(String args[])
  { Student s1 = new student();
    Student s2 = new student();
    s1.display();
    s2.display();
  }
}
```

→

```
student()  // default constructor.
{ rno = 10;
  marks = 90.3f;
  name = "ramu";
}
```

Program: Copying the Object.

```
Class student
{
  int rno;
  float marks;
  string name;
  void display()
  {
    System.out.println("rno is "+rno);
```

```
        System.out.println ("marks is " + marks);
        System.out.println ("name is " + name);
    }
}

Class StudentDemo
{
  Public static void main (String args[])
  {
     Student S1 = new student ()
     Student S2 = S1;
     S1.display ();
     S2.display ();
  }
}
```

Method
- No-arg
- parameterized

Program: No-arg method.

```
Class Student
{
  int rno;
  float marks;
  String name;
  void setDetails ()  // No-arg method.
  {
    rno = 20;
    marks = 80.5f;
    name = "ramu";
  }
void display ()
{
  System.out.println ("rno is " + rno);
  System.out.println ("marks is " + marks);
  System.out.println ("name is " + name);
}}}
```

Student () // default constructor
```
{
  rno = 0;
  marks = 0.0;
  name = null;
}
```

```
class StudentDemo
{
public static void main (String args[])
{
  Student s1 = new Student();
  Student s2 = new Student();
  s1.setDetails();
  s2.setDetails();
  s1.displayDetails();
  s2.displayDetails();
}
}
```

. First the constructor is executed and then the method is executed.

Program: Parameterized method.

```
class Student
{
  int rno;
  float marks;
  String name;
  void setDetails(int rno, float marks, String name)
  {
    this.rno = rno;
    this.marks = marks;
    this.name = name;
  }
  void displayDetails()
  {
    System.out.println("rno is " + rno);
    System.out.println("marks is " + marks);
    System.out.println("name is " + name);
  }
}
```

s1 or s2

```
Class StudentDemo
{
 Public Static void main (string args[])
 {
  Student s1 = new student ();
  Student s2 = new student ();
  s1.setDetails (10, 90.3f, "ramu");
  s2.setDetails (20, 98.5f; "rani");
  s1.displayDetails ();
  s2.displayDetails ();
 }
}
```

Program: Reference Variables.

```
Class student
{
  int sno;
  float marks;
  String name;
}
Class StudentDemo
{
 public static void main (string args[])
 {
   student s1 = new student();
   student s2 = new student();
   s1.sno = 1;
   s1.marks = 90.7f;
   s1.name = "Akhil";
   s2.rno = 2;
  s2.marks = 100;
 s2.name = "Bhanu";
```

```
System.out.println(s1.rno+"   "+s1.marks+"   "+s1.name);
System.out.println(s2.rno+"   "+s2.marks+"   "+s2.name);
```

20/1/2020

* Literals | Constants:

Literal: A value which is assigned to variable is called literal.

primitive variable.

Eg: `int a = 10;` → literal

primitive data type

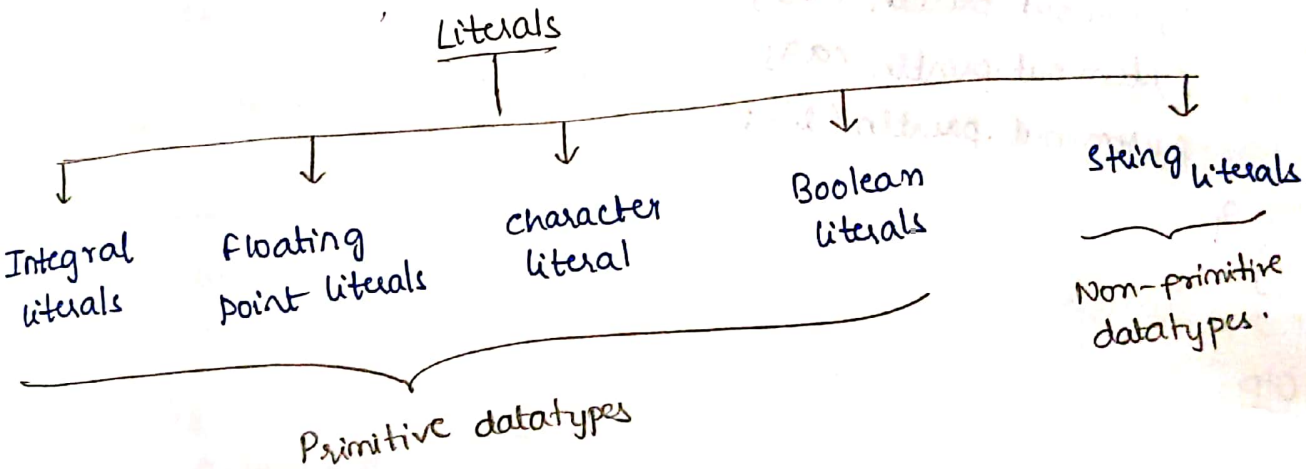
`Student s1 = new Student();` → not a literal.

`String s1 = "ramu";` → string literal.

→ ref variable

non-primitive data type

`String s1 = new String("ramu");` → not a literal.



⇒ **Integral literals:**

- byte
- short
- int
- long

Integral literals can be written in

1. decimal notation (10) - 0 to 9
2. octal notation (8) - 0 to 7 prefix is 0.
3. Hexadecimal (16) - 0 to 9, A to F. prefix is 0x (or) 0X

Prog: // Java prog using integral literals

```
class Test
{
  public static void main (String [ ] args)
  {
    byte b=10; // decimal notation.
    int a=010; // octal notation
    long l1=0x10; // hexadecimal notation
    System.out.println (b);
    system.out.println (a);
    System.out.println (l1);
  }
}
```

O/P:

10
8
16

⇒ Floating point literals:

float
double

Prog: // Java prog using Floating point literals

```
class Test
{
  public static void main (String [] args)
  {
    float f1=2.3f ;
    double d1= 10.3d;
    System.out.println (f1);
    System.out.println (d1);
  }
}
```

⇒ Character literals :

// Java program using Character literals

```
class Test
{
  public static void main (String [] args)
  {
    char c1 = 'a';
    char c2= 97;
    char c3 = '\u0061';
    System.out.println (c1);
    System.out.println (c2); // ASCII
    System.out.println (c3); //Unicode.
  }
}
```

Hexadecimal notation

0 to 65535

'\uxxxx' // unicode

0 to 15

0 to 9 , A to F

0 0 6 1

96+1 = 97

O/P :

a

a

a

⇒ Boolean literals:

(true, false)

```
class Test
{
  public static void main (String [] args)
  {
    boolean b= true ;        → boolean literal
    boolean a = false ;
    System.out.println ( b+"    "+ a);
  }
}
```

⇒ String literals:

↳ non-primitive data type.

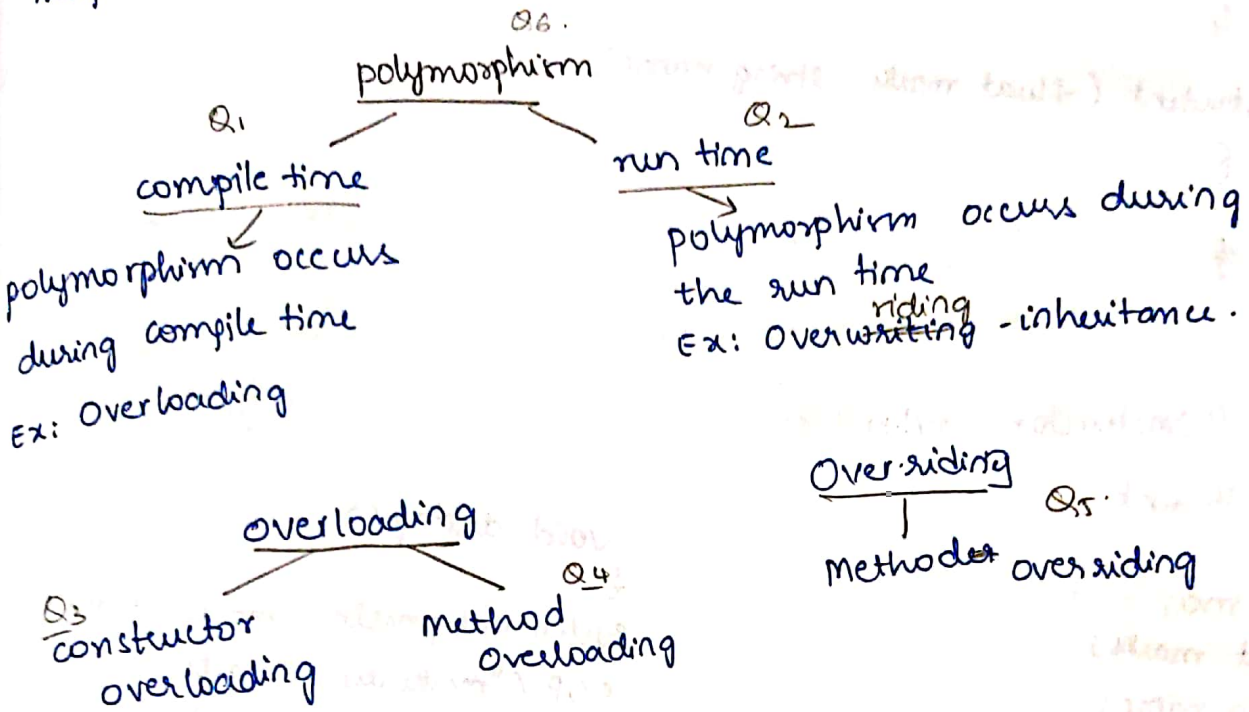
```
class Test
{
  public static void main (String [] args)
  {
    String s1 = "ramu";
    String s2 = "Krishna";
    System.out.println (s1);
    System.out.println (s2);
  }
}
```

* Polymorphism:

(many structures)

poly - many

morphism - structures.



Overloading:

Constructor overloading:

Two or more constructors having the same name and they differ either in the no. of arguments or type of arguments or order of arguments then the constructors are said to be overloading constructors.

Ex:

```
class Student
{
  Student ( )
   {

   }
  Student ( int rno)
   {

   }
}
```

// Student constructor is overloading / overloaded.

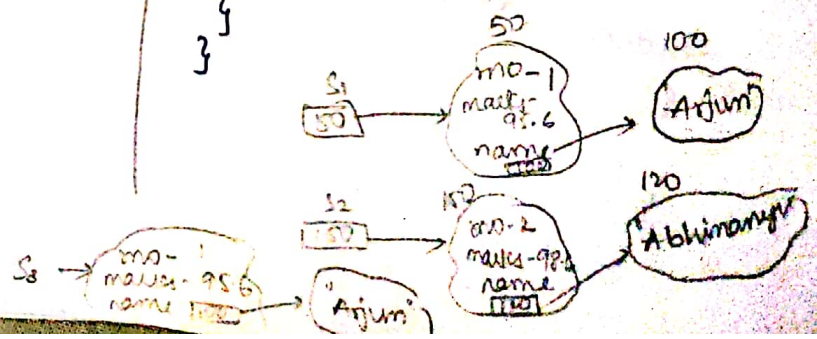
```
class Student
{
  Student (int rno , float marks)
  {

  }
  Student (float marks , string name)
  {

  }
}
```

Prog: // Constructor Overloading.

```
class Student
{
 int rno;
 float marks;
 String name;
 Student ( ) //userdefined no arg
                constructor.
 {
  rno = 1;
  marks = 95.6f;
  name = "Arjun";
 }
Student (int rno, float marks,
     String name) //userdef parameterized
                   constructor
{
  this.rno = rno;
  this.marks = marks;
  this.name = name;
 }
//user def parameterized constructor
 Student ( Student obj)
 { rno = obj.rno ;
  marks = obj.marks;
 } name = obj.name;
 void display ( )
 {
 System.out.println ("rno is " + rno);
  SOP ("marks are " + marks);
 S.O.P ("name is " + name);
 }
}
class Demo
{
 public static void main ( String[] args)
 {
  Student S1 = new student ( );
  Student S2 = new Student ( 2,
              98.6f, "Abhinanyu");
  student S3 = new Student (S1);
  S1.display ( );
  S2.display ( );
  S3.display ( );
 }
}
```

O/P:

```
1
95.6
Arjun
2
98.6
Abhinanyu
1
95.6
Arjun
```



Method overloading:

Two or more methods having the same name which may differ either with the no. of arguments or type of arguments or order of arguments but not on the return type.

Ex:

```
class Test
{
 void sum (int a, float b)
 {
 }
int sum(float a, int b)
 {
 }
}
```

Donot check the return type, check only the arguments.

Ex:

```
class Test //not overloading
{
 void sum (int a)
 {
 }
 int sum (int a)
 {
 }
}
// method sum() is not overloaded.
```

24/1/2020.

Prog: // Method overloading (number of arguments)

```
class overloadDemo
{
 void sum (int a, int b)
 {
   int c = a+b;
   System.out.println(c);
 }
 int sum( int a, int b, int c)
 {
   int d = a+b+c;
   return d;
 }
 int sum (int a, int b, int c, int d)
 {
   return (a+b+c+d);
 }
}
class Test
{
 public static void main ( String [] args)
 {
   overloadDemo O1 = new overloadDemo( );
    O1.sum(10,20);
 int x = O1.sum(10,20,30);
 System.out.println (x);
 int y = O1.sum (10,20,30,40); //
 System.out.println ( O1.sum (10,20,30,40) );
 }
}
```

can be written as
SOP ("sum is " + (a+b))

* Inheritance:

The main advantage of inheritance is Code Reusability.

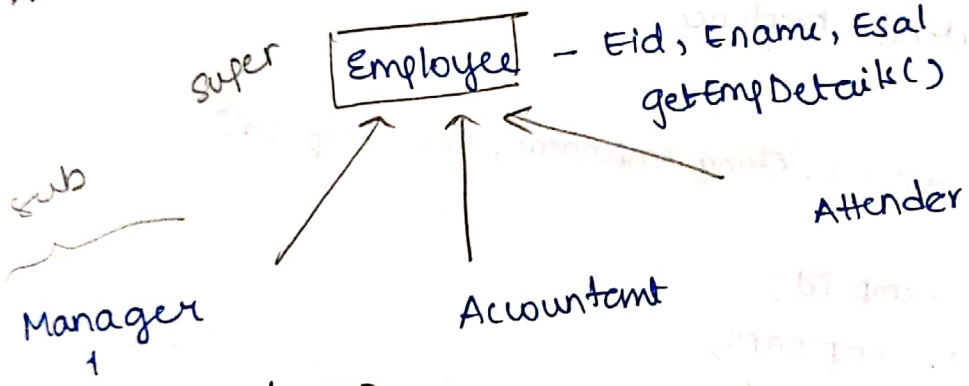
. Prog (without inheritance):

```
class Manager
{
  int empid;
  float empsalary;
  char ename;
String
  void getemployeeDetails()
  {
    System.out.println("emp id is" + empid);
    System.out.println("employee name is " + ename);
    System.out.println("employee salary is "+empsalary);

  }
}
class Accountant
{
  int empid;
  float empsalary;
  string ename;
  void getemployeeDetails()
  {
    System.out.println("emp Id is" + empid);
                "          ("employee name is " + ename);
                "          ("employee salary is "+empsalary);

  }
}
class Attender
{ same details
}
```

```
class Test
{
 public static void main(String[] args)
  {
   Manages m1 = new Manages();
   m1.getemployeeDetails();
   Account a1 = new Account();
   a1.getemployeeDetails();
   Attender at1 = new Attender();
   at1.getEmployeeDetails();
  }
}
```

o/p- 0, null, 0.0

With Inheritance: Structure



1

1. parameterized constructors } own
2. getManagerDetails()
3. 3 instance variables } inherited.
4. getEmpDetails

Total:

- 3 instance variables
- 1 Parameterized constructors
- 2 methods.

Prog: // With Inheritance.

```
class Employee
{
  int emp_id;
  String emp_name;
  float emp_sal;
 void getEmpdetails()
 {System.out.println("emp id is " + emp_id);
  System.out.println("emp name is " + emp_name);
  System.out.println("emp sal is" + emp_sal);
 }
}

class Manager extends Employee
{
  Manager(int emp_id, String empname, float emp_sal)
  {
    this.emp_id = emp_id;
    this.emp_name = emp_name;
    this.emp_sal = emp_sal;
  }
 void getManagerDetails()
 {
  System.out.println("Manager Details");
  System.out.println("---------");
  GetEmpdetails();
 }
}
```

```
class Accountant extends Employee
{
  Accountant (int emp_id, String emp_name, float emp_sal)
  {
    this.emp_id = emp_id;
    this.emp_name = emp_name;
    this.emp_sal = emp_sal;
  }
  void getAccountantDetails()
  {
    System.out.println("Accountant Details");
    System.out.println("----------");
    getEmpdetails();
  }
}
class Test
{
  public static void main(String[] args)
  {
    Manager m1 = new Manager(101, "Arjun", 50000.0f);
    m1.getManagerDetails();
    Accountant a1 = new Accountant(200, "Abhimanyu", 100000.0f);
    a1.getAccountantDetails();
  }
}
```

Test.java

```
O/P : Manager Details
      101
      Arjun
      50000.0
      Accountant Details
      200
      Abhimanyu
      100000.0
```

* What is Inheritance ?

The process of getting variables and methods from one class into another class is called Inheritance.

→ We use extends keyword in inheritance.

Types of Inheritance:

Java supports

1. Single inheritance / single level inheritance.

A - super class
↑
B - subclass

The process of getting variables or methods from one superclass to one or more subclasses is called single level inheritance.

Eg: Animal
↑
cow

Vehicle
car ↑ lorry ↑ bus

2. Java supports Multilevel inheritance.

A
↑
B → own code + A's code
↑
C → own code + B's code + A's code.

Java does not support.

1. Multiple Inheritance.

More than one super class and one sub class.

B C
↖ ↗
A

class A extends B & C.

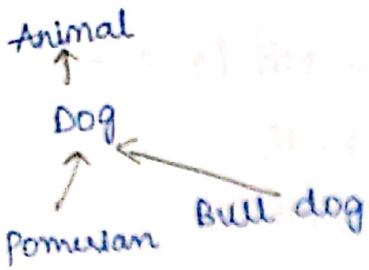
- In Java, one class can extend only one class.

2. Hybrid Inheritance.

It is a combination of multiple inheritance & multi-level inheritance.

Eg:

A
B C
D

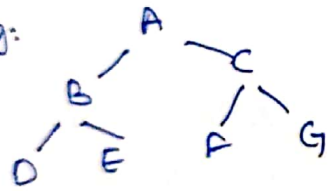


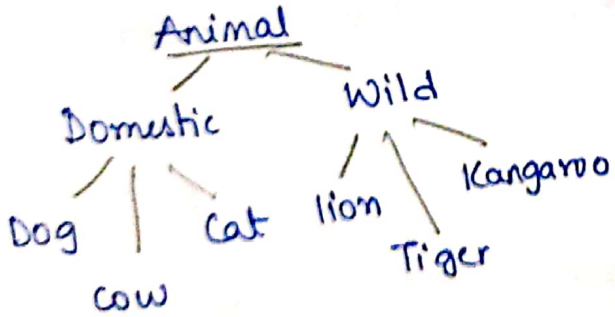
It is a combination of single level inheritances with two more than one level.

3. Hierarchial inheritances

It is a combination of single and multilevel inheritances in heirarchial fashion.

Eg:





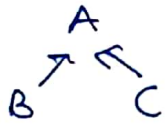
Prog: // single level inheritance.

```
class A
{
 int a=10;
 void m1()
 {
  System.out.println("A's m1()
                 method");
 }
}

class B extends A
{
 int b=20;
 void m2()
 {
  System.out.println("B's m2()
        method");
 }
}

class Test
{
public static void main(String[]
                    args)
{
  // super class object
 A a1 = new A();
System.out.println(a1.a);
   a1.m1();
// sub class object
  B b1 = new B();
System.out.println(b1.a +
    " " + b1.b);
b1.m1(); b1.m2();
}
}
```

O/P:

```
10
A's m1() method
10  20
A's m1() method
B's m2() method.
```

Prog:



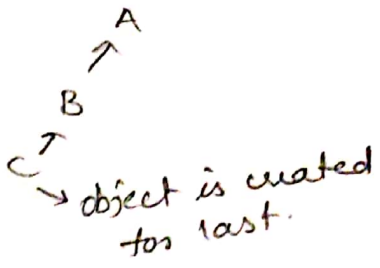

```
class A
{
 int a=10;
 void m1()
 {
  System.out.println(" A's m1() method ");
 }
}
class B extends A
{
 int b=20;
 void m2()
 {
  System.out.println("B's m2() method ");
 }
}
class C extends A
{
  int c=30;
  void m3()
  {
   System.out.println("C's m3() method");
  }
}
class Test
{ public static void main (string [] args)
  {
    A a1 = new A();
    System.out.println(a1.a);
    a1.m1();
    B b1 = new B();
    System.out.println(b1.a + "   " + b1.b);
    b1.m1(); b1.m2();
```

* Constructor chaining in inheritance:

Inside each constructor, the first statement can be either super() or this()

super() default.

// class object
{ object() } null.
{ }



Prog:

```
class A extends object
{
  A()
  { super(); //default
    System.out.println("A's constructor");
  }
}
class B extends A
{
  B()
  { super(); default //
    SOP("B's constructor");
  }
}
class C extends B
{
  C()
  { super(); // default
    SOP("C's constructor");
  }
}
class Test
{
  Public static void main(String[] args)
  {
    C c1 = new C();
  }
}
```

O/P:

A's constructor
B's constructor
C's constructor

27/1/2020.

* Inheritance: (Code reusability)

The process of getting variables and methods from one class into another class is called Inheritance.

Super class (only one) — Sub classes (more than 1) } single Inheritance.

- Constructors are not inherited to its sub classes of super class.

As per the def of inheritance, only variables and methods of super class are inherited to its subclasses

```
Super - A - a, m1() { }
            A() { } - constructor.

sub - B   - b, m2() { }
            B() { } // parameterized constructor with two parameters
            a, m1() { }
            A() { },
```

Prog on single level inheritance:

```
class A
{
  int a;
  void m1()
  {
   System.out.println("A's method is m1()");
  }
  A(int a)
  {
   this.a = a;
  }
}
```

```
class B extends A
{
  int b;
  void m2()
  {
    System.out.println("B's method is m2()");
  }
  B(int a, int b)
  {
    this.a = a;
    this.b = b;
  }
}

class Test
{
  public static void main(String[] args)
  {
      A a1 = new A(10);
      B b1 = new A(20,30);
      a1.m1();
      System.out.println(a1.a);
      System.out.println(b1.a + "  " + b1.b);
      b1.m1();
      b1.m2();
  }
}
```

O/P:

```
A's method is m1()
10
20  30
A's method is m1()
B's method is m2().
```

Prog on multilevel inheritance with user defined constructors.

A - a, m1( ){ }
A( ){ }

B - b, m2 ( ) { }
B(int a, b)

C - c, m3 ( ){ }
C(, ,)

```
class A
{
  int a;
  void m1( )
  {
   System.out.println ("A's method is m1( ) ");
  }
  A(int a)
  {
   this.a = a;
  }
}
Class B extends A
{
  int b;
  void m2 ( )
  {
   System.out.println ("B's method is m2 ( )");
  }
  B(int a, int b).
  {
    this.a = a;
    this.b= b;
  }
}
```

```
class C extends class B
{ int c;
  void m3()
  { System.out.println("C's method is m3()");
  }
  C(int a, int b, int c)
  {
    this.a = a;
    this.b = b;
    this.c = c;
  }
}
class Test
{
  public static void main(String[] args)
  {
    A a1 = new A(10);
    B b1 = new B(20,30);
    C c1 = new C(40,50,60);
    System.out.println(a1.a);
    a1.m1();
    System.out.println(b1.a + "  " + b1.b);
    b1.m1();
    b1.m2();
    System.out.println(c1.a + "  " + c1.b + "  " + c1.c);
    c1.m1();
    c1.m2();
    c1.m3();
  }
}
```

O/P:

```
10
A's method is m1()
20  30
A's method is m1()
B's method is m2()
40  50  60
A's method is m1()
B's method is m2()
C's method is m3()
```

* Method Overwriting (riding):

If the super class and sub class are having the same method with same super type then, sub class method will over write the super class method or super class method is overwritten by subclass method. [Return type must be same].

* Instance variable hiding:

If both super class and sub class are having the same instance variable, then sub class instance variable will hide the super class instance variable.

→ sub class object will give priority will give priority to its own class members.

If they are not there in sub class then only it will inherit from its super class.

i.e., always sub class will give priority to its own members. If they are not there in its own class then only it will inherit from its super class.

Prog: // Instance variable hiding and method overriding.

Super - Animal - a, b - eat(){}, sleep(){}

Sub - dog - a, c - eat(){}, bark(){}

Method overriding Eg:

```
//super
  void m1(int a, int b)
   {
   }
//sub.
  void m1(int c, int d)
   {
   }
```

Ex:

```
int m1(int a, float b)   //super
 {
 }
int m1 (int a, float b)  //sub.
 {
 }
```

Prog:

```
class Animal
{
 int a=10, b=20;
 void eat()
 {
  System.out.println ("Animal's eat() method");
 }
 void sleep()
 {System.out.println ("Animal's sleep() method");
 }
}
class Dog extends Animal
{
 int a = 20, b= 30;
 void eat()
 {
  System.out.println (" Dog's sleep() method ");
 }
 void bark()
 {
  System.out.println(" Dog's bark() method");
 }
}
```

```
class Test
{
 public static void main (String args[])
 {
  Animal a1 = new Animal();
  Dog d1 = new Dog(); // Run time polymorphism.
  System.out.println(a1.a+"    "+a1.b);
   a1.eat();
   a1.sleep();
  System.out.println(d1.a+"    "+d1.c);
   d1.eat();
   d1.bark();
 }
}
```

O/P:

```
   10    20
Animal's eat() method
Animal's sleep() method
   20    30
Dog's eat() method
Dog's bark() method.
```

- In the above prog, animal's eat method is overridden by dog's eat method.
  Dog's eat method is overriding the animal's eat method.
- Dog's eat method is called overriding method
  Animal's eat method is called overridden method.

Method overloading - compile time polymorphism.
Method overriding - Run time polymorphism.

→ We can't access animal's a and animal's eat() method with sub class object.

. Using super.member, we can access
variable method.

→ In case of instance variable hiding and method over riding, to access the super class methods inside the sub class, we should use super.member inside the sub class.

→ To prevent method overriding & instance variable hiding we use super.member inside sub class.

Prog using super.member.

```
class Animal
{ int a=10, b=20;
  void eat()
  {
   System.out.println("Animal's eat() method");
  }
  void sleep()
  { System.out.println("Animal's sleep() method");
  }
}
class Dog extends Animal.
{
  int a=30, c=40;
  void eat()
  { S.O.P("Dog's eat() method");
  }
  void Bark()
  { S.O.P("Dog's bark() method");
  }
  void m1()
  { S.O.P(super.a);
    super.eat();
  }
```

```
class Test
{Public static void main(String() args)
 {
  Animal a1 = new Animal();
  SOP (a1.a + "    " + a1.b);
   a1.eat();
   a1.sleep();
  Dog d1 = new Dog();
  SOP (d1.a + "    " + d1.b + "    " + d1.c);
   d1.eat();
   d1.sleep();
   d1.bark();
   d1.m1();
 }
}
```

30/1/2020

* Super Keyword:

→ We can use super keyword as super.method.

super.variable super.member.

→ super() - Constructor chaining.

→ super(parameters)

prog on super():

Inside each constructor, whether it is user defined or compiler generated constructor, the first statement must be super() by default.

Constructor chaining or super():

prog on super(parameters):

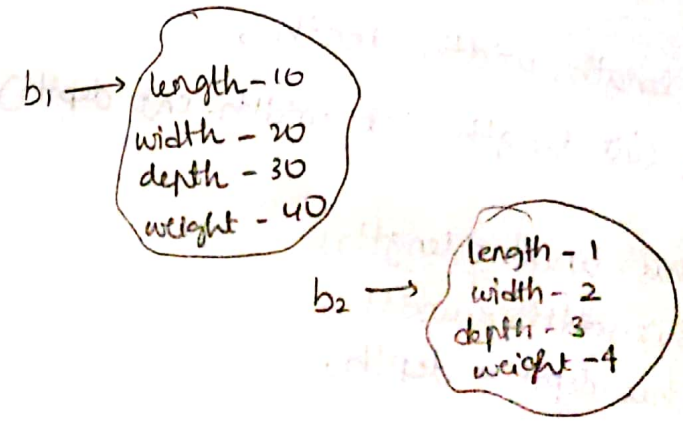
Box - 1. width, depth, length
2. Box (-,-,-)
3. volume

Box Weight - 1. Weight
2. BoxWeight (-,-,-,-)
{ super (-,-,-)
—
}

```
class Box
{
 int length, width, depth ;
 Box (int length, int width, int depth)
 {
  this. length = length;
  this.width = width;
  this.depth = depth;
 }
 void volume ()
  {
   System.out.println ("Volume is " +( length * width * depth));
  }
}
class Boxweight extends Box
 { int weight;
  Boxweight(int l, int w, int d, int wt)
  { super (w, d, wt);
    length = l ;
    width = w ;
    depth = d ;
    weight = wt;
 }}
```

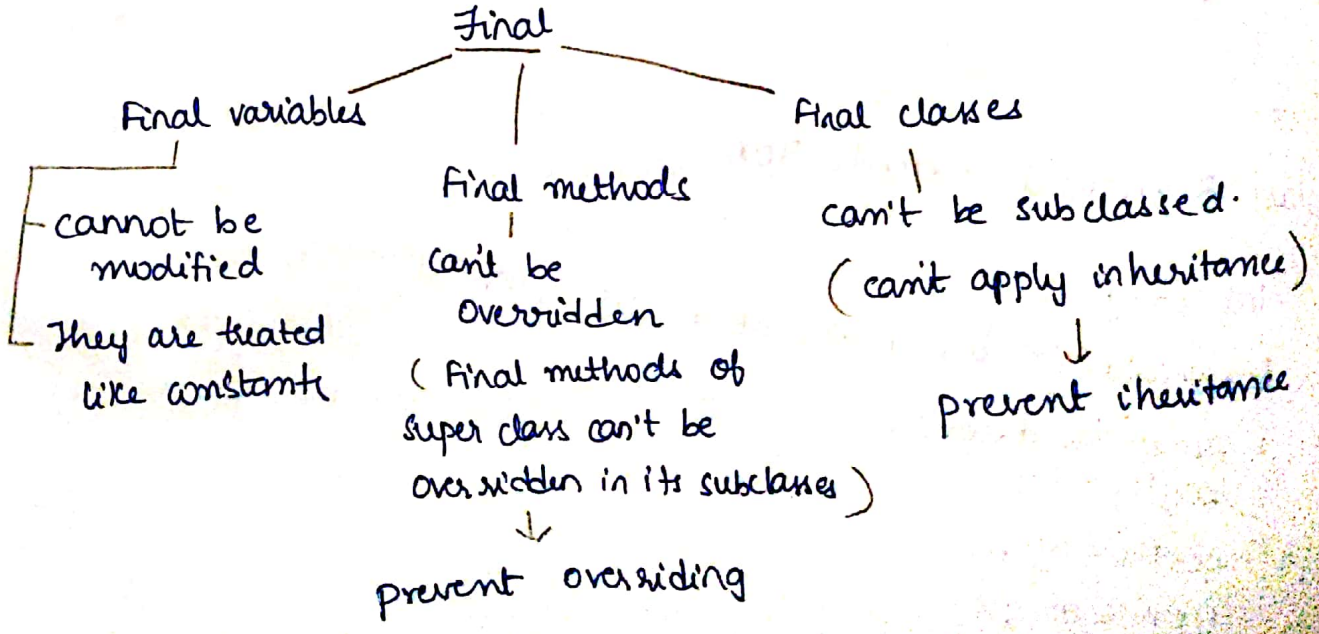
```
class Test
{
 public static void main (String [] args)
 {
   Boxweight b1 = new Boxweight (10, 20, 30, 40);
   Boxweight b2 = new Boxweight (1, 2, 3, 4);

    b1.volume();
    b2.volume();
 System.out.println (b1.weight);
 System.out.println (b2.weight);
 }
}
```

O/P :
volume is 6000
volume is 6
40
4.



* Final Keyword :

There are three ways to use final keyword.



⇒ Final classes:

```
final class Box
{
int width, height, depth;
}
class Boxweight extends Box
 {
  }
```

compiler error.

O/P: Error

⇒ Final methods:

```
class Animal
{
final void eat()
 {
 System.out.println ("Animal's eat method");
 }
}
class Dog extends Animal
 {
  void eat()
  {S.O.P. ("Dog's eat method");
   }
  }
```

compiler error.

O/P: Cannot inherit from final Box

⇒ Final variables:

In inheritance only we use final classes and final methods.
(no inheritance for final variables).

```
class Test
 {
final int a=10; // a is instance variable
final static int b=20; // b is static variable.
      public static void main (String [] args)
      {
final int c=30; c is local variable.
```

```
Test t1 = new Test();
System.out.println(t1.a + "  " + t1.b + "  " + t1.c);
                                                or c.
t1.a = t1.a + 10;
t1.b = t1.b + 20;
t1.c = t1.c + 30;
```

} not possible

a 10
b 20
c 30

```
    }
}
```

O/P : Compiler error.

31/1/2020.

* Type Casting :

Type Casting

- **Primitive type casting (c, java)**
  - (8 primitive datatypes)
  - ↓
  - casting is done with primitive datatype
    - **Implicit typecasting**
      - ↓
      - lower datatype is assigned to higher.
    - **Explicit typecasting**
      - ↓
      - Higher datatype is assigned to lower.
- **Non-primitive type casting. (java)**
  - (classes, interfaces)
  - ↓
  - Casting is done with non-primitive datatypes.

Non-primitive type casting. - Inheritance (available in)

- **Upcasting**
  - ↓
  - If casting is done w.r.t. super class, then it is called upcasting
- **Downcasting**
  - ↓
  - If casting is done w.r.t. sub class then it is called Downcasting.

→ Inheritance — not req in primitive
\ req. in non-primitive.

Prog: Java prog on Primitive type casting.

```
class Test
{
  public static void main (String args[])
  {
    // implicit type casting
    byte b = 10;
    int b = a;  (or)  int a = (int) b;
    System.out.println (a+b);
    System.out.println (a+" "+b);
    // Explicit type casting.
      int a = 100;
    //  byte b = a;  Compiler error.
      byte b = (byte) a;
    System.out.println (a+" "+b);
  }
}
```

compiler.

~~Prog: Java prog on Non-primitive type casting.//~~

* Non-primitive type casting:

⇒ Upcasting:

Witho inheritance we explain both upcasting and down casting.

→ Casting is done with super class.

A - Super class

↑

B - sub class.

def: Sub class object is assigned to super class reference variable.

- With that reference variable, we can access all the members of super class except overridden method.

Prog: //Upcasting

```
class Animal
{
 int a=10, b=20;
 void eat()
 {
  System.out.println("Animal's eat method");
 }
 void sleep()
 {
  System.out.println("Animal's sleep method");
 }
}
class Dog extends Animal
{
 int a=20, c=30;
 void eat()
 {
  System.out.println("Dog's eat method");
 }
 void bark()
 {
  System.out.println("Dog's bark method");
 }
}
class Test
{
 public static void main(String args[])
 {
  Animal a1 = new Animal(); // super class object.
  Dog    d1 = new Dog();    // sub class obj.
  System.out.println(" a1.a + "  " + a1.b);
  a1.eat(); a1.sleep();
```

Animal - a, b
eat(), sleep
↑
Dog - a, c
eat(), bark().

```
        System.out.println (d1.a+" "+d1.b+"  "+d1.c);
        d1.eat();
        d1.sleep();
        d1.bark();
    }
}
```

O/P:

```
10  20
Animal's eat method
  "    sleep  "
20  20  30
Dog's eat method
Animal's sleep method
Dog's bark method.
```

```
class Test //Upcasting
{
public static void main (String [] args)
Animal a1= new Dog();
```

→ Super class ref variable

(or)

```
Animal a2 = (Animal) new Dog();
System.out.println (a2.a +" "+a2.b);
a2.eat();
a2.sleep();
}
```

O/P: 10 20
Dog's eat method.
Animal's sleep method.

→ Drawback of upcasting:

We cannot access the remaining members of subclass.

Eg: subclass bark method, a, C,

- To execute those members we use down casting.

Down casting:

Down casting must be followed by Upcasting i.e., it cannot be done individually.

(cont...)

```
class Test
{
 public static void main(String [] args)
 {Animal a3 = (Animal) new Dog();
   Dog d2 = (Dog) a3;
S.O.P(d2.a + "  " + d2.b + "  " + d2.c);
    d2.eat();
    d2.sleep();
    d2.bark();
 }
}
```

O/P:

```
20 20 30
Dog's eat method
Animal's sleep method
Dog's bark method.
```

→ In inheritance w.r.t. super class and sub class, we will create the objects as follows.

(i) `Animal a1 = new Animal();` // Super class obj.

(ii) `Dog d1 = new Dog();` // Sub class obj.

(iii) `Animal a1 = new Dog();` // Upcasting

✗ (iv) `Dog d2 = new Animal();` // Error (Compiler error)

(v)
```
Animal a1 = new Dog();
Dog d1 = (Dog) a1;
```
} // Down casting.

✗ (vi)
```
Animal a1 = new Animal();
Dog d1 = (Dog) a1;
```
} Run time error. ↓ class cast exception.

1/2/2020

Methods

Concrete Methods

- should have method body
- not declared with abstract keyword

Ex: void m1()
{
}
↳ concrete method

Abstract methods

- Do not have method body
- Must be declared with abstract keyword
- Ends with semicolon.

Ex: abstract void m1();
abstract int m2();
↳ abstract methods.

1. Interfaces
2. Abstract classes } Inheritance.

Classes

Concrete classes

Must contain only concrete methods and should not be declared with abstract keyword.

Ex: class A
{
void m1()
{
}
void m2()
{
}
}

Abstract classes

may contain only concrete methods or only abstract methods or both concrete and abstract methods.

Concrete | Abstract | both

- The class must be declared with abstract keyword. Then, the class is called abstract class.

Eg: abstract class A //only concrete.
{ void m1()
{
}
}

```
abstract class B // only abstract
{
 abstract void m1();
 abstract int m2();
}
abstract class A
{
 void m1()
 {
 }
 abstract void m2();
 {
 }
```

::Rules:

1) Abstract classes may contain concrete methods, abstract methods or both.

2) Abstract classes may contain concrete methods but a class which contains atleast one abstract method must be declared as abstract. otherwise compiles error.

Eg:

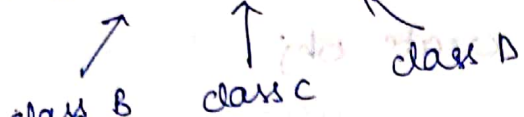
```
class C
{
 abstract void m1();
}
```

O/P: compiler error.

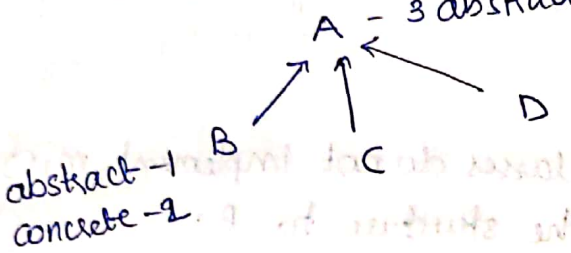
3) Abstract classes must be extended.

4) There must be atleast one sub class (concrete) which extends the abstract class.

Eg: abstract class A { }



5) A class which extends the abstract class must provide implementation to all the abstract methods of the abstract class.



Eg: A - abstract class
3 abstract methods
↑
B - 3 concrete

A - 3 abstract methods
↑
abstract B - 2 concrete
1 abstract
↑
C - 1 concrete.

Eg Eg

6) If the sub class doesn't want to implement all the abstract methods of abstract class, then declare the sub class as abstract and there must be a sub-class which extends the abstract class.

A
↑
B
↑
C
↑
D
↑
E
↑
F - concrete

7) For abstract classes we cannot create objects, we can only create reference variables.

Ex: abstract class A
{ }
A a1; // Reference variables
X A a1 = new A(); // cannot create obj.
}

- We can access the members of abstract class using Upcasting.

8) Abstract method provides more sharability when compared to concrete methods.

A m1(){} → abstract classes do not implement m1() they provide the structure to B.
↑
B m1(){}

. One structure can be used by many classes.

Programs on Abstract classes (Using single level inheritance.)

abstract A – m1(){} – 1 concrete.
↑ m2(); m3(); } 2 abstract
concrete B – m2(){} m3(){} } inherited [m2, m3 are implemented]
m4(){}

```
abstract class A
{
  void m1()
  {
  System.out.println(" A's m1()");
  }
```

```
abstract void m2();
abstract void m3();

}
class B extends A
{
  void m2()
  {
   System.out.println("B's m2()");
  }
  void m3()
  {System.out.println("B's m3()");
   }
  void m4()
   {
    System.out.println("B's m4()");
    }
  }

class Test
{
 public static void main (string args[])
 {  A a1 = new B(); // Upcasting.
    a1.m1();
    a1.m2();
    a1.m3();
 // a1.m4(); Compiler error
    B b1 = new B(); // subclass obj
    b1.m1();
    b1.m2();
    b1.m3();
    b1.m4();
   }
  }
```

Prog:

abstract A - $m_1()$ - concrete
↑ $m_2()$; - abstract
m_3; - abstract.

abstract B - $m_2()\{\}$
↑ $m_3()$;
$m_4()\{\}$

concrete C - $m_3()\{\}$
$m_5()\{\}$

```
abstract class A
{
 void m1()
 {
  System.out.println("A's m1()");
 }
 abstract void m2();
 abstract void m3();
}

abstract class B extends A.
{ void m2()
  {
   System.out.println("B's m2()");
  }
  abstract void m3();
  void m4()
  {
   System.out.println("B's m4()");
  }
}
```

```
class C extends B
{
 void m3()
 {
  System.out.println("C's m3()");
 }
 void m5()
 {
  System.out.println("C's m5()");
 }
}
class Test
{
public static void main (String [] args)
 {
// A a1 = new A(); Compiler error
// B b1 = new B(); compiles error.
    A a1 = new C(); // Upcasting.
    a1.m1();
    a1.m2();
    a1.m3();
    B b1 = new C();
    b1.m1();
    b1.m2();
    b1.m3();
    b1.m4();
    C c1 = new C();
    c1.m1();
    c1.m2();
    c1.m3();
    c1.m4();
  } c1.m5();
}
```

OP:

A's m1()
B's m2()
C's m3()

A's m1()
B's m2()
C's m3()
B's m4()

A's m1()
B's m2()
C's m3()
B's m4()
C's m5().

* Command line Arguments:

Write a java prog to find the sum of two integers using command line arguments.

P1:
```
class Test  // without command line arguments.
{
 public static void main(String [] args)
  {
    int a=10, b=20;
    System.out.println("Sum of a & b is " +(a+b));
  }
}
```

O/P: Sum of a & b is 30.

P2: // with Command line Arguments.

```
class Test
{
 public static void main(String[] args)
  {
    int a=Integer.parseInt(args[0]);   // string is converted into int.
    int b=Integer.parseInt(args[1]);
    System.out.println("Sum of a & b is "+(a+b));
  }
}
```

O/P:
```
Test.java
javac Test.java
java Test 10 20      -> command line -> at runtime.
```

string args -> | 0: "10" | 1: "20" |

Integer is the wrapper class w.r.t. primitive datatype int.

```
class Integer
{
 static int parseInt(" ")
  {
   returns int
  }
}.
```

→ Except for char and ~~double~~ boolean, all the 6 number datatypes (byte, short, int, long, float, double) have wrapper classes.

→ Factorial of a no. using CLA.
→ Sum of integer and float using CLA.

Integer.parseInt(args[0])

- Integer: wrapper class corresponding datatype int
- parseInt: static method of integer class which is taking argument as string & returns integer.

Since, parseInt is a static method of integer class, and we are accessing it in the Test class i.e., outside the class, we must use classname.method name.

Eg: Integer.ParseInt.
Float.ParseFloat
Byte.parseByte etc.

[Integer, Byte, Short, Long, Float, Double] - Wrapper classes

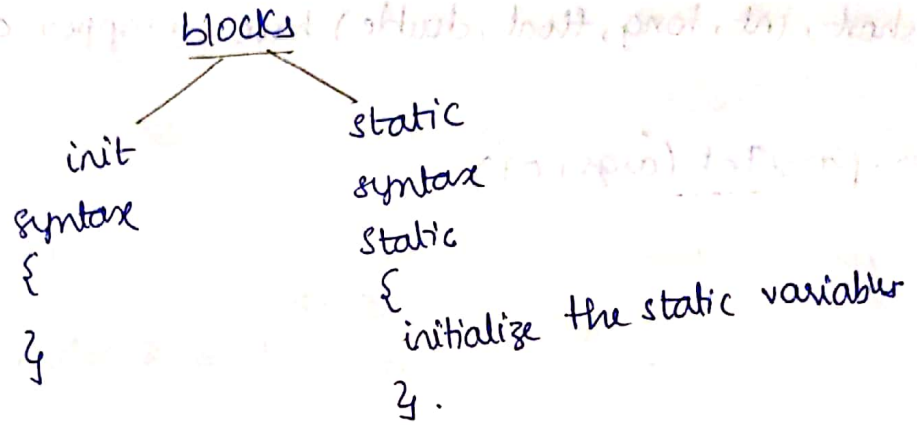
↘ Parse methods are used to convert strings into corresponding datatypes.

3/2/2020

* Static Keyword: Only once memory is created and can be accessed many times.

There are four ways to use static keyword.

1. Static variables.
2. static methods.
3. Static blocks.
4. Static import (packages)



* Static variables:

1. Static variables are defined inside the method | block | constructor and declared with static keyword.
2. Static variables are class variables i.e., the scope of variables is throughout the class.
3. Memory for static variables is allocated during the class loading time and released during the class unloading time.

- static variables must be initialised at the time of declaration.

1. Instance variables are defined inside the method | block | constructor and not declared with static keyword.
2. Instance variables are object level variables. i.e., the scope is within the object.
3. Memory for Instance variables is allocated at the time of object creation and released at the time of object destruction.

- May or may not be initialized at the time of declaration.

accessed
nes.

3. Memory for static variables is allocated only once for the entire class.

6. Static variables are stored in method area.

6. Common properties are declared under static variables.

4. Memory for instance variables is created for each object.

5. Instance variables are stored inside heap area. because instance variables are part of the object which are stored in heap.

6. For different properties, use instance variables.

Prog on static Variables:

P1: // How to use static variables within the same class. (3 ways)

a) With class name
b) With object
c) Direct.

P2: How to use static variables outside the class (2 ways)

a) With class name
b) with object.

→ javac Test.java → Test.class file is created.

java Test.

① Test.class file is loaded into memory

↓

during the class loading time all the static variables and static blocks will be executed from top to bottom.

② main() method of Test.class file.

③ If any class files are there Eg: class A, A.class files are created.

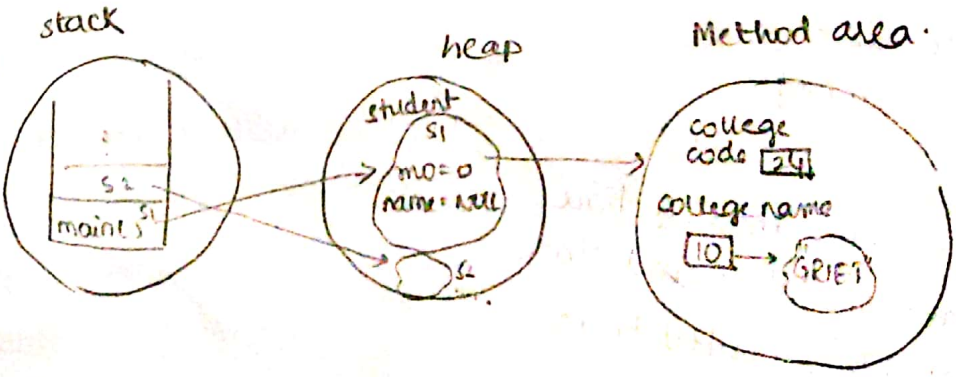
How to access static variables: (3 ways) + (2 ways)

- JVM – loads
  | \ verifies
  executes.

Prob:

```
// within the same class.
class Student
{
  static int collegecode = 24;
  static String collegename = "GRIET";   } before main()
         int rno;
        String name;
  public static void main (String[] args)
  {
    System.out.println (Student.collegecode);   } // with class name
    System.out.println (Student.collegename);
    System.out.println (collegecode + "  " + collegename); // direct.
     Student s1 = new Student ();
    System.out.println (s1.collegecode + "  " + s1.collegename);
    System.out.println (s1.rno + ".  " + s1.name);   // object
  }
}
```

java student
↓
student.class file is loaded into memory.



O/P: 24
GRIET
24 GRIET

24 GRE
0 null

Prog 2: Outside the class.

→ The object in java are destroyed by Garbage collector.

Useless objects will be destroyed by the Garbage collector.

↳ no reference variable for the object.

```
class student
{
 int sno;
 string name;
 static ~~int~~ string collegename = "GRIET";
 static int collegecode = 24;
}
class Test.
{
 public static void main (string [] args)
 {
  System.out.println (student.collegecode+" "+ student.collegename);
  student s1 = new student();
  System.out.println(s1.collegecode+" "+s1.collegename);
  System.out.println(s1.sno +" "+s1.name);
 }
}
```

O/P:

```
24   GRIET
24   GRIET
 0   null.
```

Note:

- Why should we declare common properties as static?
  If we declare common properties as static, the memory is allocated for the static variables only once and all the objects of that class can use the static variable.
- If we won't declare, then memory will be allocated for each object since they are treated as instance variables. Hence, memory is wasted.

Prog:

→ If we change a static variable with any one of the objects, then the change is reflected to all the objects.

——→ include class student (prev prog)

```
class Test
{
 public static void main (String [] args)
 {
  System.out.println (Student.collegecode + "    " + Student.collegename);
  Student s1 = new Student ();
  System.out.println (s1.collegecode + "  " + s1.collegename);
  Student s2 = new Student ();
  System.out.println (s2.collegecode + "  " + s2.collegename);
  s1.collegecode = 40;
  System.out.println (s2.collegecode );
  s2.collegename = "VNR";
  System.out.println (s1.collegename);
 }
}
```

O/P: 24 GRIET
24 GRIET
24 GRIET
40
VNR.

⇒ In inheritance, static variables are inherited.

* Interfaces:

Interfaces contain 100% abstract methods.

	(concrete) classes	abstract classes	interfaces.
Methods	Consists of only concrete methods	Consists of both concrete & abstract methods.	consists of only abstract methods.
variables	Consists of both final & non-final variables	Both final & non-final variables	Only final variables.

- Interfaces contain final variables and abstract methods.

Syntax: (Keyword)

```
interface interfacename
{
  final variables
  abstract methods.
}
```

Ex:

```
interface I1  // access modifier here is <default>
{
// final int a=10, b=20;
public static final int a=10, b=20;
public abstract void m1();
```

(public static final → default; public abstract → default.)

- By default, all the variables of the interface are public, static and final.
  By default, all the methods are public & abstract.
  The access modifier of the interface is default.
- If we want to write the access modifier for interface, we can use only public (explicitly)
  i.e. Only two access modifiers can be used i.e. default & public.

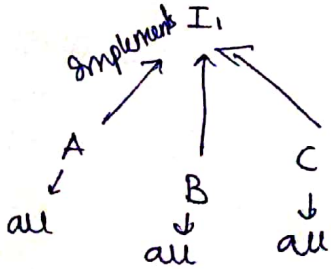
Ex:

```
public interface I1
{
  int a=10, b=20;
  void m1();
}
```

- After defining the interface, there must be atleast one class which implements the interface. and that class has to provide implementations to all the abstract methods of that interface. That class can also be called implementing class.

Ex:

```
interface I1 { }
class A implements I1{ }
:
class B implements I1{ }.
```



• For interfaces, we can't create ~~reference variables~~ objects like abstract classes.
but, we can create reference variables.

• Use upcasting to access the interface methods / members.
↳ implementing class object is assigned to interface reference variable.
With this reference variable, we can access all the members of interface.

⇒ Programs on interfaces:

P_1: interface I_1, class A implements I_1.

P_2: interface I_1, class A implements I_1, class B implements I_1.

P_3: interface I_1, interface I_2, class A implements I_1, I_2. (Multiple inheritance)

P_4: one class can extend one class and implements any number of interfaces.
interface I_1, interface I_2, class A, class B extends A implements I_1, I_2.

• Interfaces can also be extended like classes.

P_5: interface I_1, interface I_2 extends I_1.

• One interface can extend any no. of interfaces.

P_6: interface I_1, interface I_2, interface I_3 extends I_1, I_2
(Multiple inheritance).

• Interface using abstract classes.

P_7: interface I_1, abstract class A implements I_1., class B extends A.

Between classes, interfaces } extends

interface - class → implements.

prog1:

```
interface I1
{
 public static final int a = 10;
 public abstract void m1();
 public abstract void m2();
}

class A implements I1
{
 public void m1()
 {
System.out.println("A's m1()");
 }

 public void m2()
 {
  System.out.println("A's m2()");
 }
}

// own method.
void m3()
{
 System.out.println("A's m3()");
}

class Test
{
 public static void main(String[] args)
 {
  // upcasting.
   I1 i1 = new A();
// With upcasting, only a, m1, m2 can be accessed.
   i1.m1();
   i1.m2();
// i1.m3(); compiler error.
 System.out.println(I1.a + "  " + i1.a);
```

private < default < protected < public.

Superclass overridden $\leq$ = subclass overriding.

→ The access modifier of the sub-class overriding method must be greater than or equal to superclass overridden method.

Eg:

protected — protected public

```
A a1 = new A();
System.out.println( A.a + "  " + a1.a);
a1.m1();
a1.m2();
a1.m3();
}
}
```

O/P:

```
A's m1()
A's m2()
10   10
10   10
A's m1()
A's m2()
A's m3().
```

→ I1 i1 = new A();

All the variables are executed (LHS) during compile time.

and overridden methods of super class all the methods that are over riding i.e., only over riding methods of A will be executed.

↓ by

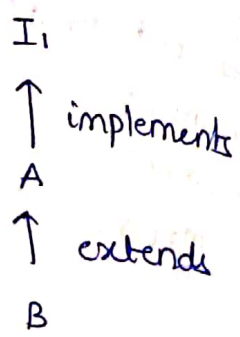
JVM

↓ during

Runtime

P7:

```
interface I1
{
 public static void main
 Public static final int a=10;
 public abstract void m1();
 public abstract void m2();
}

abstract class A implements I1
{
  int b=20;
  public void m1()
   {
    System.out.println("A's m1()");
   }
  abstract void m2();
  void m3()
   {
    System.out.println("A's m3()");
   }
}

class B extends A
{
 void m2()
 {
  System.out.println("B's m2()");
 }
}

class Test
{
 public static void main(sting[] args)
 {
```



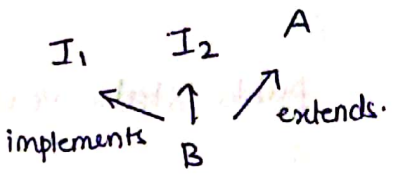
```
// upcasting
I1 i1 = new B();
System.out.println(i1.a+" "+I1.a);
  i1.m1();
  i1.m2();
 //I1 i1
}
}
```

O/P:

```
10  10
A's m1()
B's m2().
```

P4:

```
interface I1
{
 [public static final] int a=10;
 [public abstract] void m1();
}
interface I2
{
 [public static final] int b=20;
 [public abstract] void m2();
}
class A
{
 void m3()
 {
  System.out.println("A's m3()");
 }
}
```



```
class B extends A. implements I1, I2.
{
  public void m1()
  {
    System.out.println("B's m1()");
  }
  public void m2()
  {
    System.out.println("B's m2()");
  }
  void m4()
  {
    System.out.println("B's m4()");
  }
}

Class Test
{
  public static void main (string[] args)
  {
    I1 i1 = new B();
    System.out.println(i1.a + " " + I1.a);
    i1.m1();
    i1.m2();
    I2 i2 = new B();
    System.out.println(i2.b + " " + I2.b);
    i2.m2();
    A a1 = new A();
    B b1 = new B();
    a1.m3();
    b1.m4();
  }
}
```

O/P:

```
10 10
B's m1()
20 20
B's m2()
A's m3()
B's m4()
```

P5:

```
interface I1
{ int a=10;
  void m1();
}
interface I2
{ int b=20;
  void m2();
}
interface I3 extends I1, I2
{
  int c=30;
  void m3();
}
class A implements I3
{
  public void m1()
  {
    System.out.println("A's m1()");
  }
  public void m2()
  {
    S.O.P("A's m2()");
  }
  public void m3()
  {
    S.O.P("A's m3()");
  }
  void m4()
  {
    S.O.P("A's m4()");
  }
}
```

I_3 has 3 abstract methods.
3 final variables.

```
class Test
{
 public static void main (String[] args)
  {
    I3 i3 = new A();
   System.out.println(I1.a + " " + I2.b + "  " + I3.c);
   i3.m1();
   i3.m2();
   i3.m3();
 // i3.m4(); Compiler error.
   A a1 = new A();
   a1.m4();
  }
}
```

O/P:

```
10   20   30
A's m1()
A's m2()
A's m3()
A's m4().
```

10/2/2020

ARRAYS

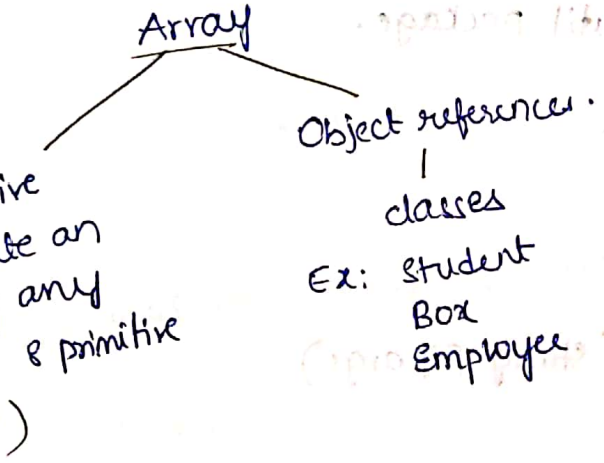
Arrays are objects in Java.

→ length() is the instance variable which is available in array class which gives the size of the variable.

→ Array is a predefined class in Java.

```
class Array
{
  int length;
  :
}
```

Based on datatype:



→ One Dimensional arrays:

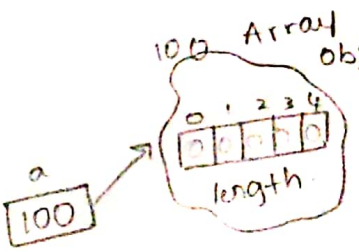
Arrays of primitives:

1. Declare - int [] a; or int a[]; byte[] b; boolean [] c;
2. Create - int [] a = new int[5]; → reference variable.
3. initialize.

Size must be given on RHS

→ a is a variable which is of integer type array.

→ int [] a;
a = new int[5];



a.length = 5.

a[0] = 0
a[1] = 0
a[2] = 0
a[3] = 0
a[4] = 0

- Inside each array object 'length' instance variable is stored.

default values are stored in the array based on the datatype.

initialization:

1. Direct initialization.
2. for loop
3. Command Line Arguments
4. Buffered Reader class
5. Scanner class

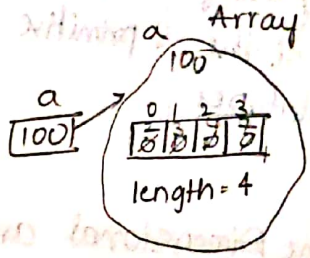
→ Through keyboard.
(like scanf in C)

- Q. How to take the input through keyboard in Java language.

A. 1. To: BufferedReader class - java.io package.
2. Scanner class - java.util package.

Prog: // Direct Initialization.

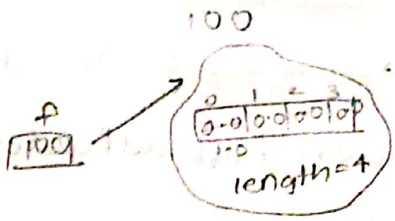
```
class Test
{
  public static void main (String [] args)
  {
    int [] a = new int [4] ;
    //initialize the array - directly
    a[0] = 2; a[1] = 3; a[2] = 4; a[3] = 7;
    System.out.println (" Array elements are");
    System.out.println ("a[0] + " " + a[1] + " " + a[2] + " " +
                                        a[3]);
  }
}
```



O/P: 2 3 4 7.

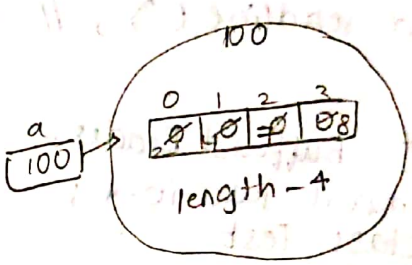
prog 2: // using for loop.

```
class Test
{ public static void main (string [] args)
  { float [] f = new float double [4];
reading ← for (int i = 0; i < f.length ; i++)
    { f [i] = i + 1;
    }
writing ← for (i = 0; i < f.length ; i++)
    { System.out.println (f [i])
    }
  }
}
```



prog 3: // Command Line Arguments.

```
class Test
{ public static void main (string [] args)
  { int [] a = new int [4];
    for (int i = 0; i < a.length ; i++)
    { a [i] = Integer.ParseInt (args [i])
    }
    for (int i = 0; i < a.length ; i++)
    { System.out.println (args a [i]);
    }
  }
}
```



java Test 2 4 7 8

Buffered Reader class:

It is a pre-defined class. It is available in java.io package.

```
class BufferedReader
{
  String readline() // non-static method
```

→ // readline() can be accessed only through one way i.e., using objects.

→ // Static methods can be accessed in two ways { with classname / with object.

→ // Non-static methods can be accessed with object.

→ Object initialisation for BufferedReader.

```
BufferedReader br = new BufferedReader(new InputStreamReader
                                       (System.in));
br.readline(); // calling the method.
```

Prog 1: Buffered Reader.

```
import java.io.*;
```
(or) io.BufferedReader, InputStreamReader;

```
class Test
{
public static void main (String[] args)
 {
  int[] a = new int[4];
  BufferReader br = new BufferedReader (new InputStreamReader
                                        (System.in));

  for(int i=0; i<a.length; i++)
  { a[i] = Integer.parseInt (br.readLine());
  }
  for(int i=0; i<a.length; i++)
  { System.out.println(a[i]);
  }
} }
```

→ In BufferedReader class, only one method is available i.e., readline() method.
It always returns string value which need to be converted again.

Scanner class:

Scanner is a pre-defined class. It is available in java.util package.

Methods in scanner class:

- nextInt()
- nextByte()
- nextShort()
- double nextDouble()
- string nextLine()

} non-static

• for Boolean, this cannot be applied. Only directly initialization is possible.

↳

```
class Test
{
  main()
  {
    Boolean [] b = new
      Boolean[3];
    b[0] = true;
    b[1] = false;
    b[2] = true;
    for ( . . . )
  }
```

→ Create objects:

```
Scanner s = new Scanner(system.in);
```

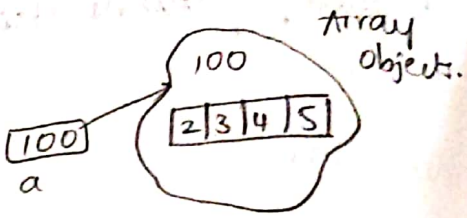
prog 5: Scanner

```
import java.lang.*;
import java.util.*;
class Test
{
  public static void main(string [] args)
  {
    int [] a = new int[5];
    Scanner s = new Scanner (System.in);
    for (int i=0; i<a.length; i++)
    {
      System.out.println("Enter the value");
      a[i] = s.nextInt();
    }
    for (int i = 0; i<a.length; i++)
    {
      System.out.println(a[i]);
    }
  }
}
```

12/2/2020.

* How do we declare, create and initialize in one step.

```
int [ ] a = { 2, 3, 4, 5 }
```



* Ananymous arrays:

No name for the array.

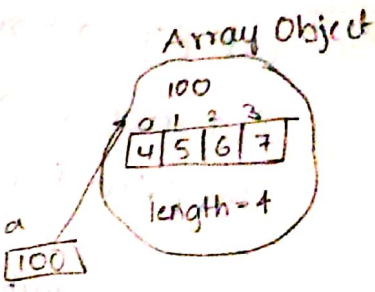
→ We use ananymous arrays as method arguments.

Syntax:

```
new int [ ] = { arguments }
```

prog:

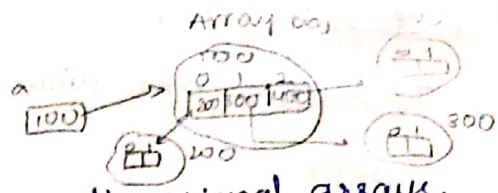
```
class TestArray
{
  void m1(int [ ] a)
  {
    for (int i = 0; i < a.length ; i++)
    {
      System.out.println (a[i]);
    }
  }
public static void main(string [ ] args)
{
  TestArray t1 = new TestArray ( );
    t1.m1( new int [ ] {4,5,6,7} );
}
}
```



* Two-Dimensional arrays:

1. Declare and create:

```
int [][] a = new int [3][2] ;
```



- Two-Dimensional array is an array of one-dimensional arrays.

In the above case,
there are 3 one-dimensional arrays each of one-dimensional array is 2. size of

- Initially, default values are stored in the last level.

2. Initialization:

- Direct initialization
- For loops
- Command line argument
- BufferedReader.
- Scanner

prog:

```
import java.util.*;
class TestArray
{
  public static void main (String [] args)
  { int[][] a = new int [3][2];
    Scanner sc = new Scanner (System.in);
    System.out.println ("Initializing the array");
    for (int i=0 ; i<a.length ; i++)
      for (int j=0; j< a[0].length ; j++)
        a[i][j] = sc.nextInt();
    for (int i=0 ; i<a.length ; i++)
      for (int j=0 ; j<a[0].length ; j++)
        System.out.println (a[i][j]);
  }
}
```

7 8 3 5 6 1

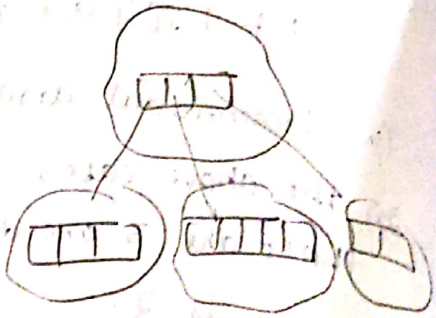
* Jagged arrays:

To declare one-dimensional array of different sizes.

Syntax:

```
int [][] a = new int[3][];
a[0] = new int[3];
a[1] = new int[5];
a[2] = new int[2];
```

} for each 1-d.



* Three-dimensional arrays:

```
int [][][] a = new int[3][2][2];
```

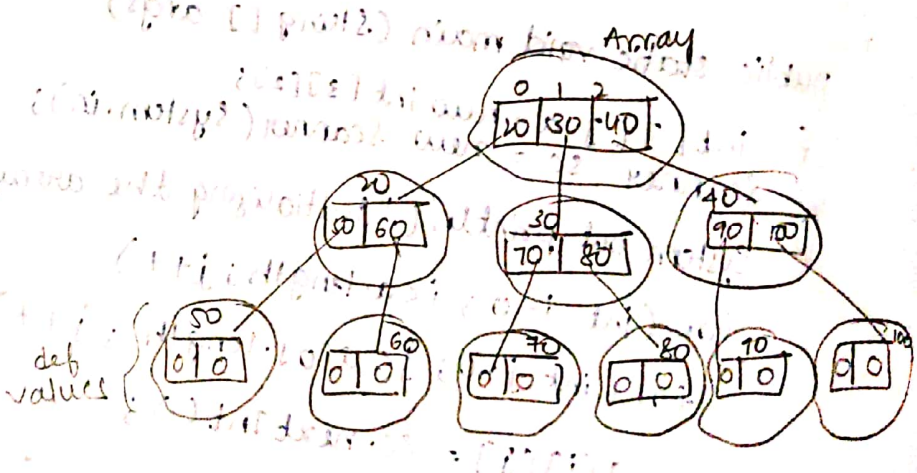
Three-dimensional array is an array of two dimensional array.

In the above case;

int[3][2][2] → size of 1-d array.

└ 2 one-dimensional

3 two dimensional



prog:

```
import java.util.*;
class TestArray
{
  public static void main(String [] args)
  {
    int [][][] a = new int [3][4][5];
    Scanner sc = new Scanner (System.in);
    System.out.println ("Initializing the array ");
    for (int i=0; i<a.length ; i++)
      for (int j=0; j<a[0].length ;j++)
        for (int k=0; k< a[0][0].length ; k++)
          a[i][j][k] = sc.nextInt();

    for (int i=0; i<a.length ; i++)
      for (int j=0; j<a[0].length ; j++)
        for (int k=0; k<a[0][0].length ; k++)
          System.out.println (a[i][j][k]);

  }
}
```

Q. How do we declare, create and initialize 3-dimensional & 2-dimensional on one step.

```
int[][] a =
  int [][] a = {{1,2,3},{4,5,6},{7,8}};
  int [][][] b = {{{1,2},{3,4},}{{7,8,9}{10}}};
```

* Arrays of Object References:

Syntax:

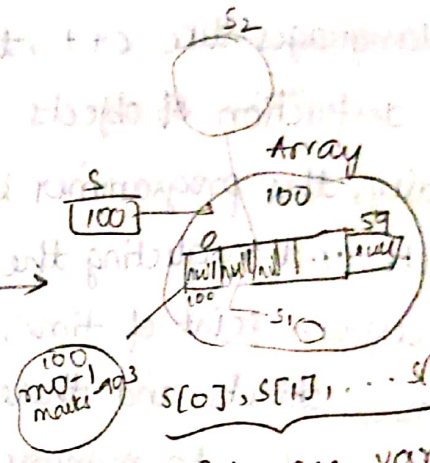
```
classname [] var = new classname [size];
```

prog: // without using arrays.

```
class Student
{
  int rno;
 float marks;
 public static v
 Student ()
 {
   rno = 1;
   marks = 90.3f;
 }
 Student (int rno, float marks)
 {
   this.rno = rno;
   this.marks = marks;
 }
 void display ()
 {
 System.out.println (rno + " " + rno);
 System.out.println (" marks are " + marks);
 }
public static void main (String [] args)
{
    Student s1 = new Student ();
    Student s2 = new Student (2, 90.0);
    Student s3 = new Student (3, 91.0);
      :
    Student s60 = new Student (60, 93.5);
  s1.display ();
  s2.display ();
    :
  s60.display ();
} }
```

prog : //Using arrays.

```
class student
{
  int rno;
  float marks;
  student()
  {
     same as prev prog
  }
public static void main (steing [] args)
{ //declaration
  Student [] s = new student[60];
  //initialization.
  s[0] = new student();
  s[1] = new student(2, 93f);
  s[2] = new student(3, 98.5f);
    :
  s[59] = new student(60, 93f);
  s[0].display();
  s[1].display();
  s[2].display();
    :
s[59].display();
}
}
```




```
for(int i=0; i<s.display; i++)
{ System.out.println (s[i].rno);
                      (s[i].marks);
  :  :  :
}
```

17/2/2020.

* Garbage Collection:

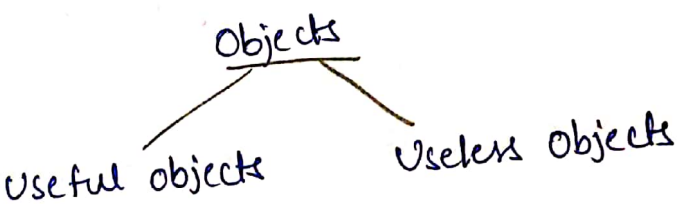
1. Introduction
2. Making objects available eligible for G.C.
3. Methods to run request JVM to run G.C.
4. finalization - finalize()

1) Introduction:

In languages like C++, the program is responsible for both creation and destruction of objects

Usually, the programmer is taking very much care while creating the objects and neglecting the destruction of objects. Due to this negligence at certain point of time, the memory may not be available for creation of new objects and there may be a maximum chance of failing the application due to memory problem.

- In Java, the programmer is only responsible for creation of objects.
- (Some) Sun people provided an assistant called Garbage collector which is always running in the background.
- Due to this Garbage collector, there may be a less chance of fail in java application and hence, java is considered as Robust language.



Useful object:

When there is a reference variable to an object, then that object is called useful object

Eg:

```
Student s1 = new Student();
```

Useless object:
When there is no reference variable to an object, then that object is called useless object.

Eg: sno = 1;

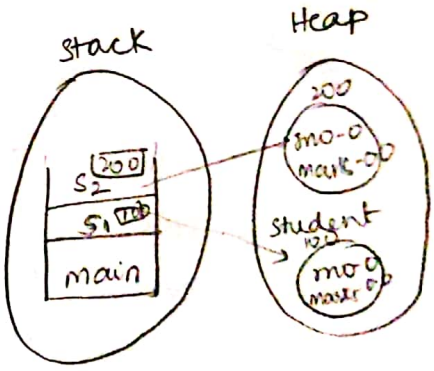
→ Garbage collector will destroy only useless objects.
- JVM will invoke Garbage collector then, Garbage collector will enter into the heap memory to destroy the useless objects. But, we don't know how many objects are destroyed by Garbage collector when it enter into the heap memory.
- The programmer cannot invoke Garbage collector but, the programmer can request the ~~Garbage collector~~ JVM to run Garbage collector.

programmer → JVM → Garbage collector

2) Making objects eligible for Garbage collection:
Even though, the programmer is not responsible for destruction of useless objects, but, it is a good programming practice to make an object ~~useful~~ eligible for Garbage collection when it is no longer required.

Eg:

```
class Student
{
 int sno;
 float marks;
 public static void main (string [] args)
 {
  Student s1 = new Student ();
  Student s2 = new Student ();
 }
}
```



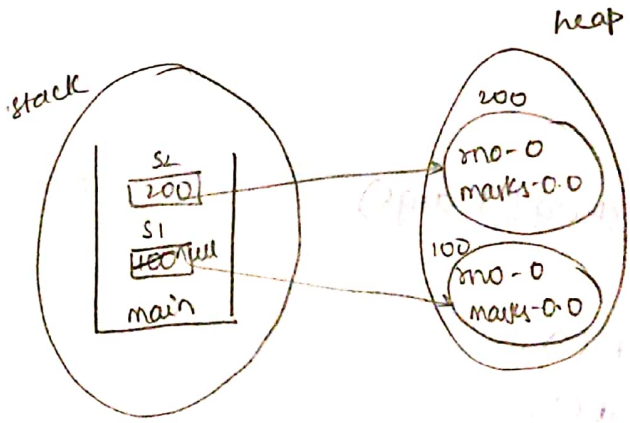
→ In stack, reference variables are stored & it is temporary.
Heap, objects are stored & it is permanent memory.

→ After the completion, the stack becomes empty and the variables in heap becomes useless objects since, s1 & s2 are deleted / popped.

→ We can make the object eligible for G.C by assigning NULL to the reference variable, if it is no longer required.

Prog:

```
class Student
{
  int rno;
  float marks;
  public static void main (String [] args)
  {
    Student s1 = new Student();
    Student s2 = new Student();
    ==========
    s1=null; - How many objects are eligible for garbage collection
    ==========
    s2=null; - How many objs are eligible for GC.
    ==========
  }
```



→ To release the memory, we are making the objects eligible for GC if they are no longer required.

3) Methods to request JVM to run Garbage collector:

The programmer cannot invoke GC directly, but the programer can send a request to JVM to run garbage collector programatically.

- The following are the methods to request JVM.

. In the predefined class system, there is one static method gcc().

```
class system
{
 static gc()
   {
   }
 System.gc().
}
```

System.gc() is used since gc() is static class.

prog:

```
class Student
{
 int no;
 float marks;
 public static void main(String [] args)
 {
  Student s1 = new Student();
  Student s2 = new Student();
  ======
  ======
  s1 = null;
  ======
  ======
  s2 = null;
  System.gcc(); // we are sending a request to JVM to run
                   Garbage collector.
 }
}
```

Note:

- Using System.gc(), we are sending a request to JVM to run garbage collector.

  But, we don't know whether our request is accepted by JVM or not.

- We also don't know whether GC destroys all the useless objects.

Using Runtime class:

Runtime is a predefined class & is a singleton class. For singleton classes, we cannot create objects with new operator.

```
Runtime r1 = new Runtime(); // error.
```

We cannot create.

→ To create objects for singleton classes, we use factory methods.

getRuntime() is a factory method and which is a static method of Runtime() class and returns runtime class object. and which is a static method.

```
Runtime r = Runtime.getRuntime();
            // getRuntime is a static method of Runtime class.
```

Prog:

```
class Runtime
{
static Runtime getRuntime()
  {
    return obj
  }
  gc() //non-static method
  {
  }
int totalMemory()
  {
  }
int freeMemory()
  {
}}
```

→ totalMemory() - non-static

Used to find the size of the heap.

freeMemory() - non static -

Used to find the free memory available in heap.

Prog: // using Runtime class.

```
class RuntimeDemo
{
public static void main (string [] args)
{
Runtime r = Runtime.getRuntime();
System.out.println(r.totalMemory()); //529 bytes } 529-420
System.out.println(r.freeMemory()); // 420 bytes.
for(int i=0; i<10000; i++)
{
  Date d = new Date();
  d= null;
}
System.out.println(r.freeMemory()); // 390.
r.gc();
System.out.println(r.totalMemory()); // 470
}
}
```

memory occupied by objects.

19/2/2020.

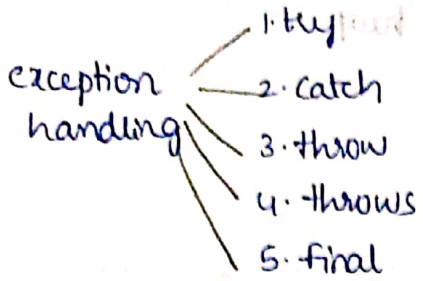
4) Finilization - Finalize()

Just before destroying any object, the Garbage collector invokes finalize() method on that object to perform clean up activities like closing database connections, closing network connections, closing file streams.

Q. Explain the difference between finalize() method & finally block.

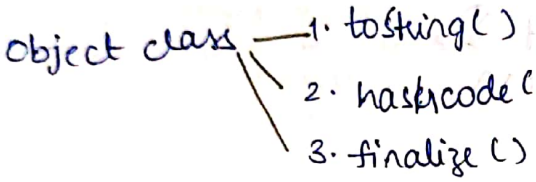
(or)

Explain finalization in Garbage collection.



→ Finalize method is available in object class.

predefined method.



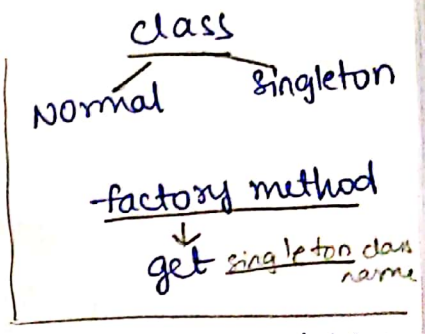
→ Syntax for finalize method.

```
Protected void finalize() throws throwable { }
```

```
class Object
{
  Protected void finalize() throws throwable
  {
    // null implementation
  }
}

Class Test extends Object
{
 // overriding the finalize()

Protected void finalize()
 or
 Public
 {
   clean up code - closing the connections.
 }
}
```

→ If we want to perform clean up activities like closing the connections then, we must override the finalize method of the object class. otherwise no need to override.

→ Just before destroying the object, the garbage collector has to execute either the finalize method of the object class or our class overriding finalize method.



Prog: // Finalize method.

```
class Test
{
Connection c = Connection.getConnection()
// connection - singleton class.
 File f = new File("abc.text"); // c & f are instance & ref variables.
 f.open()
// overriding finalize method
 Public void finalize()
 {
   System.out.println("finalize() method");
    c.close(); // connectivities get closed } Garbage collector will invoke.
    f.close(); // connectivities get closed. } No need to invoke it explicitly.
 }
 public static void main (string [] args)
 {
   Test t1 = new Test();
    t1 = null;
    System.gc();
    System.out.println("main method");
  }
}
```

- Test.java
- java Test