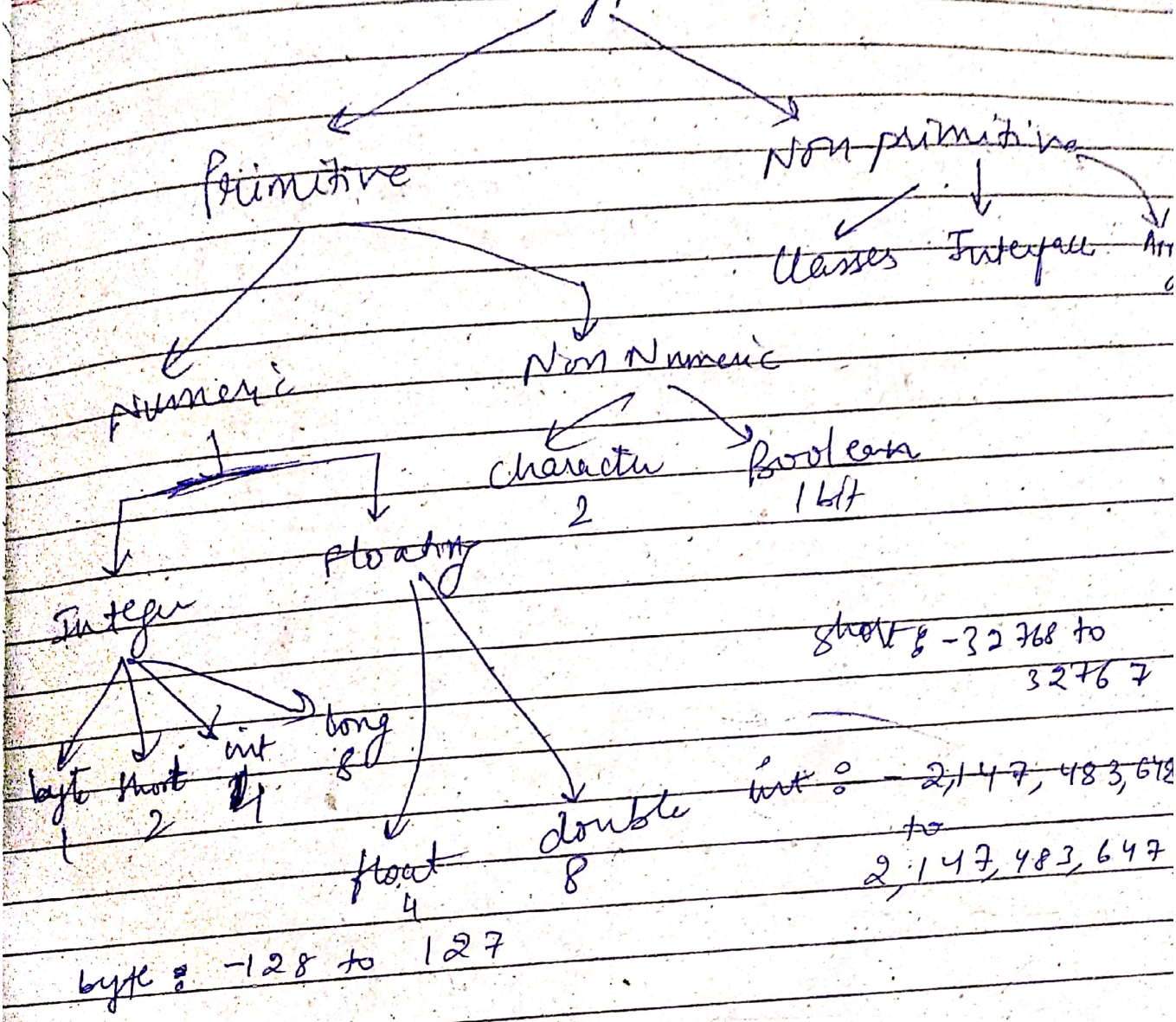
Datatypes



Java does not support the concept of unsigned datatypes.

Automatic conversions

When one type of data is assigned to another type of variable, an automatic type conversion take place if the following 2 conditions are meant.

1) The 2 types are compatible.
2) The destination type is larger than the source type.

When these 2 condition are met, a widening conversion takes place.

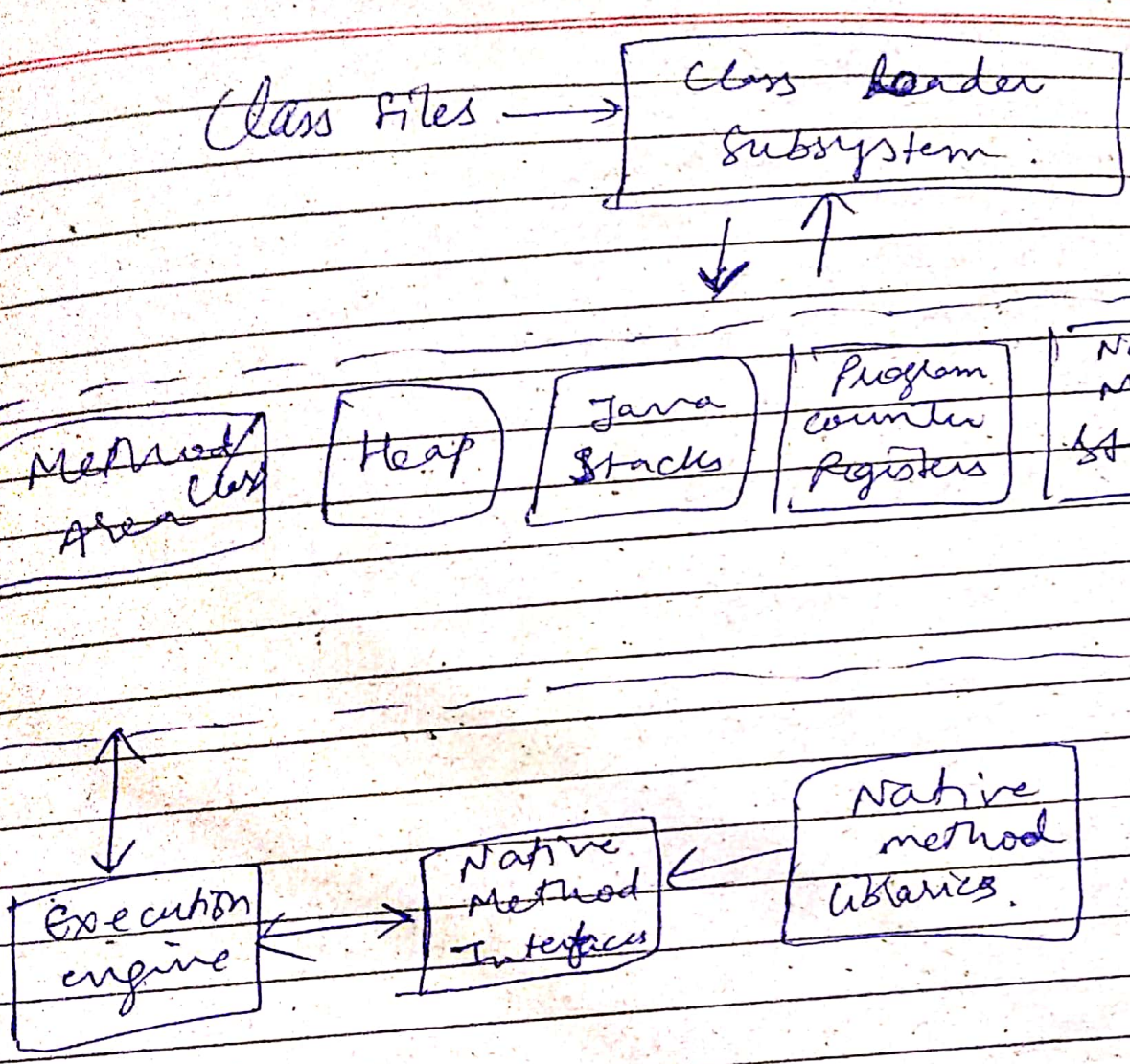
Casting Incompatible types

If you want to assign an integer value to a byte variable, this type of conversion will not be performed automatically. It is also called a narrowing conversion. So, we have to use the type casting explicitly.

JVM Architechture

The class loader subsystem will take a .class file as the input & perform the following operations.

1) It is responsible for loading the .class file into the JVM.
2) Before loading the bytecode into the JVM, it will verify whether the bytecode is valid or not.



This verification will be done by bytecode verifier.

3) If the bytecode is valid then the memory for bytecode will be allocated in different areas. The different areas into which the bytecode is loaded are called as runtime data areas.

Method Area / Class Area: JVM method area can be used for storing the class code & the method code. All classes bytecode is loaded & stored in this runtime area & all static variables are created in this area.

Heap Memory: JVM heap area can be used for storing all the objects that are created.

It is the main memory of JVM, all objects of classes & non static variables memory are created in this runtime area.

Program counter register: It will contain the address of next next instruction that have to be executed.

Java Native Stack: It is used for storing non Java codes available in the application.

Execution engine: The execution engine of JVM is responsible for executing the program & it contains 2 parts,
i) Interpreter
ii) JIT compiler

DATE / /
PAGE

~~i) Interpreter~~ ⊗ The java code will be executed by both the interpreter & JIT compiler simultaneously which will reduce the execution time & also provide high performance.

JIT compiler compiles the part of the bytecode that have similar functionality & hence reduce the amount of time needed for compilation.

Java

Java stack :- It can be used for storing the information of the method ie, under the execution. The java stack can be considered as the combination of stack frame in whose every frame will contain the state of a single method.

In the run time area all java methods are executed. In this run time JVM by default it creates 2 threads i) Main method thread
ii) Garbage collection thread.

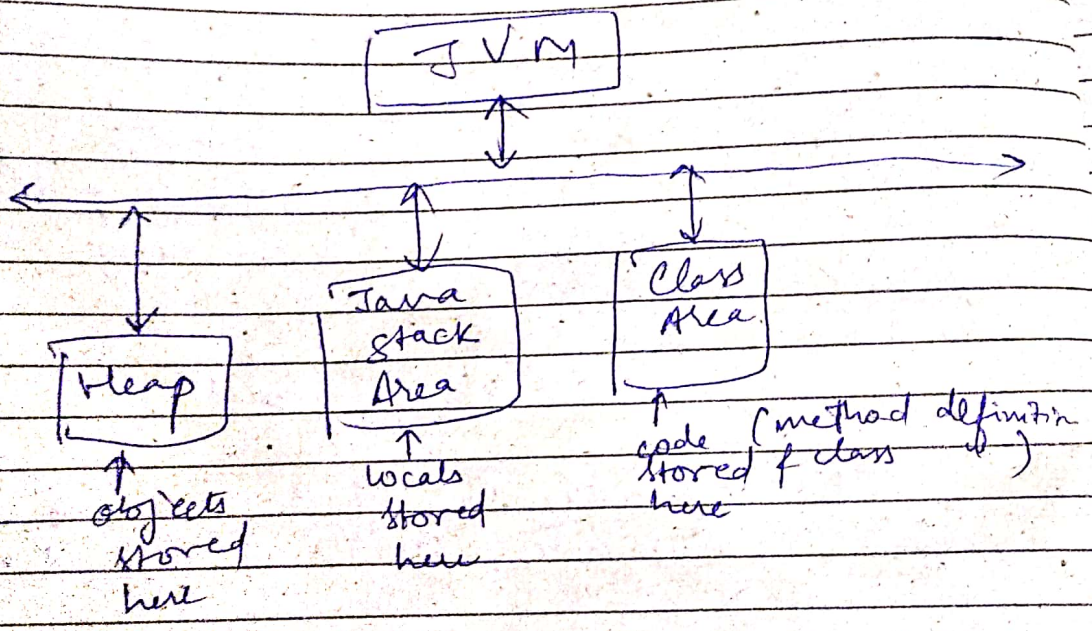
Here main thread is responsible to execute java method start with the main method.

The garbage collection thread is responsible to destroy all unused objects from the heap area.

Java stack store the frames. It hold the local variable & partial result. Each thread has a private JVM stack created at the same time as the thread is created.

A new frame is created each time a method is invoked. A frame is destroyed when its method invokation completes.

Organisation of JVM.



Class Area : All Java methods are compiled into bytecodes which are stored in this area.

Heap : Every time new is invoked, memory or instance of class is allocated in this area.

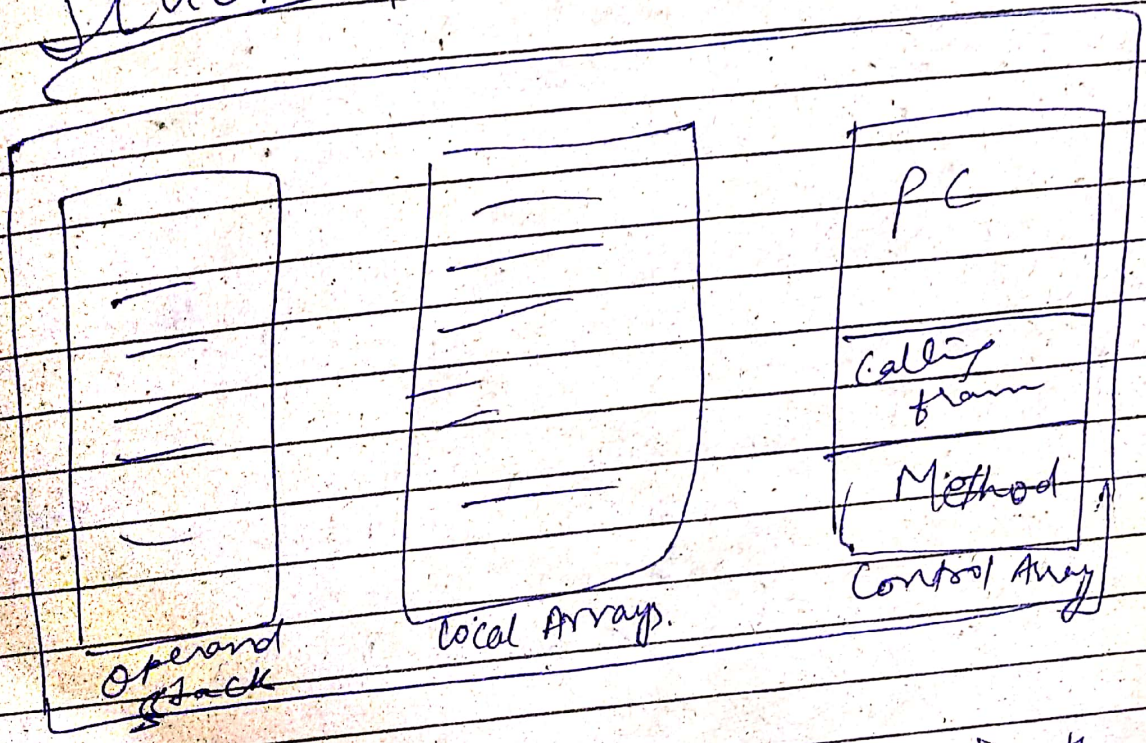
Java Stack : Everytime a method is invoked a stack frame is created & pushed into the Java stack. When

JVM is a stack machine.

DATE / /
PAGE

the method is terminated, this method is popped off from the stack. The top most frame of Javastack always belong to the method i.e; currently being executed by the JVM

Stack Frame



There are 3 areas contained in the stack frame :

1) The control array contains a reference to the current method frame & reference to the calling frame & a program counter which is pointing to the next byte code instruction to be executed.

2) The local array contains all local variables & parameters of the currently executing parameter.

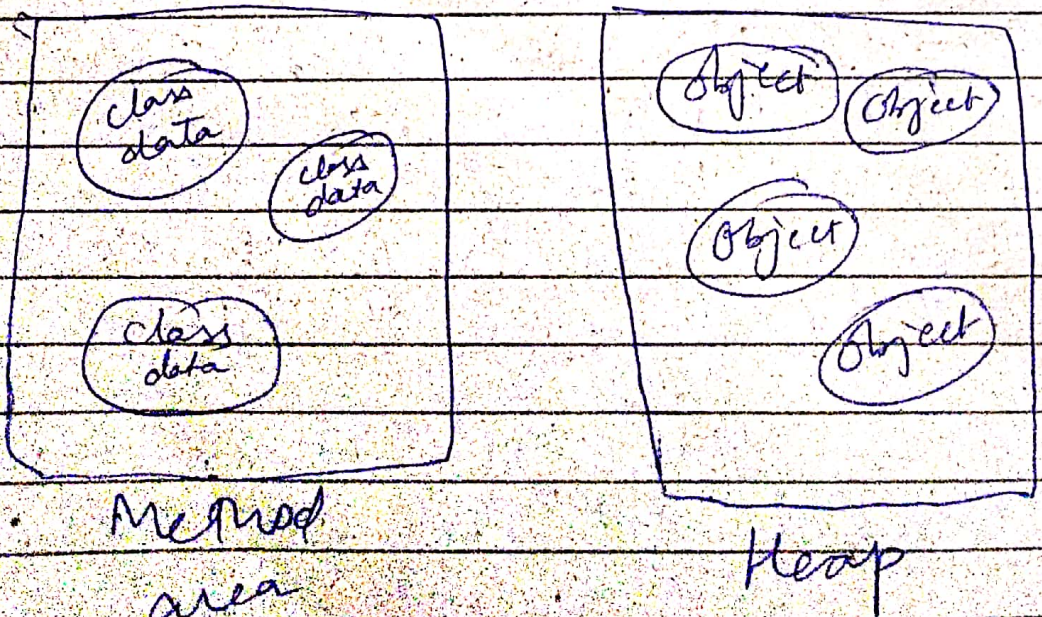
3) Each method get its own operand stack where it can store the intermediate value of the calculation ~~There are 2~~ +

There are 2 types of processors:

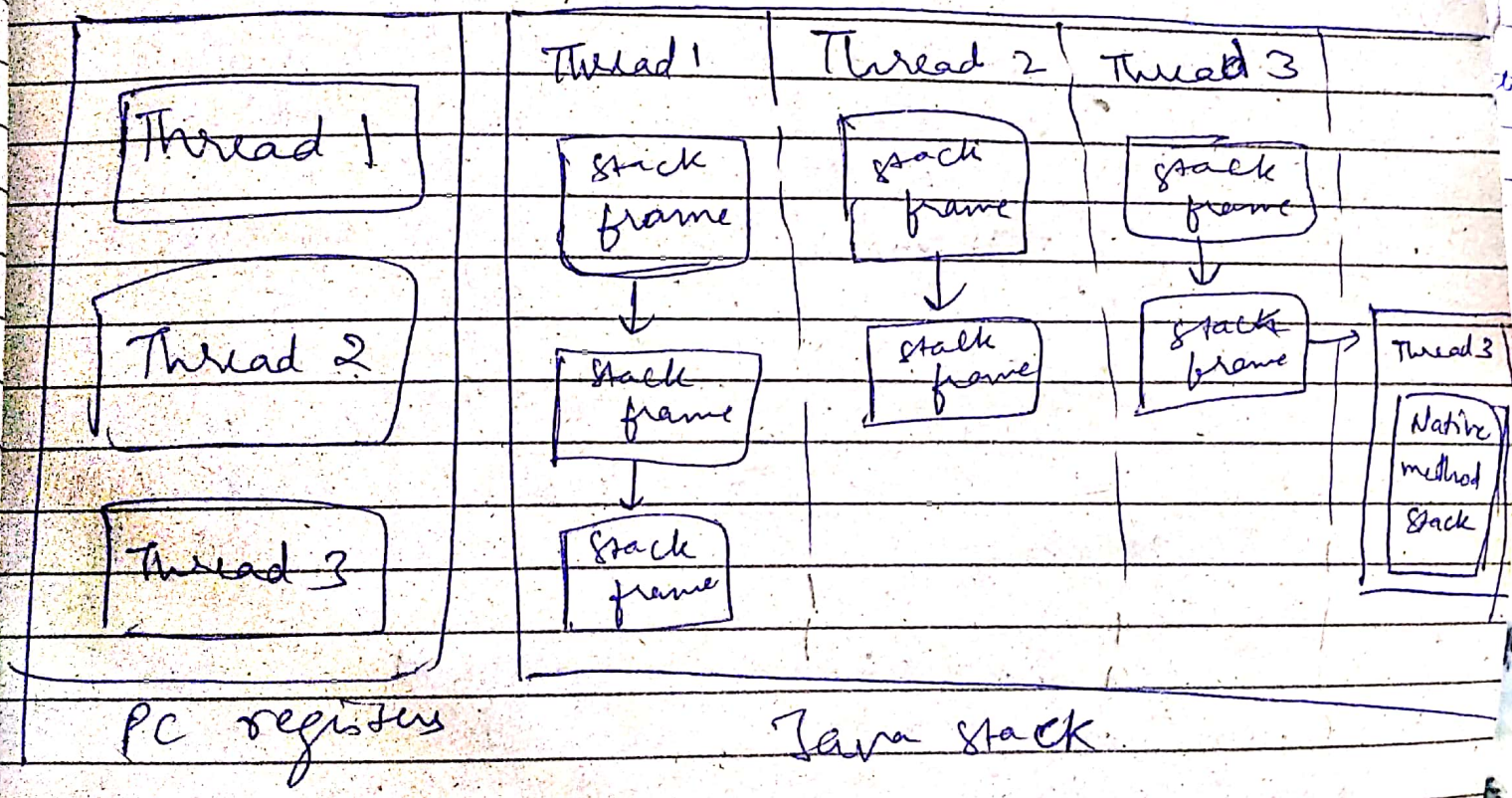
1) Register Machine : It stores the intermediate results of lengthy calculations in the register.

2) Stack machine : store the intermediate results on the stack.

The JVM is a stack machine



Run time data area exclusive to each thread



Some run time data area are shared among all of an application thread & others are unique to the individual thread.

Each instance of JVM have one method area & one heap area. These areas are shared by all the threads running inside the virtual machine as each new thread comes into existance, it get its own pc register and Java stack. If the thread is executing a Java method the value of pc registers indicate the next instruction to be executed. A thread Java stack store the state of Java method invocation for the thread. The state of Java method invocation include its local variables, the parameter with which it was invoked its return value & intermediate calculation.

The state native method invocation is stored in the native method stack.

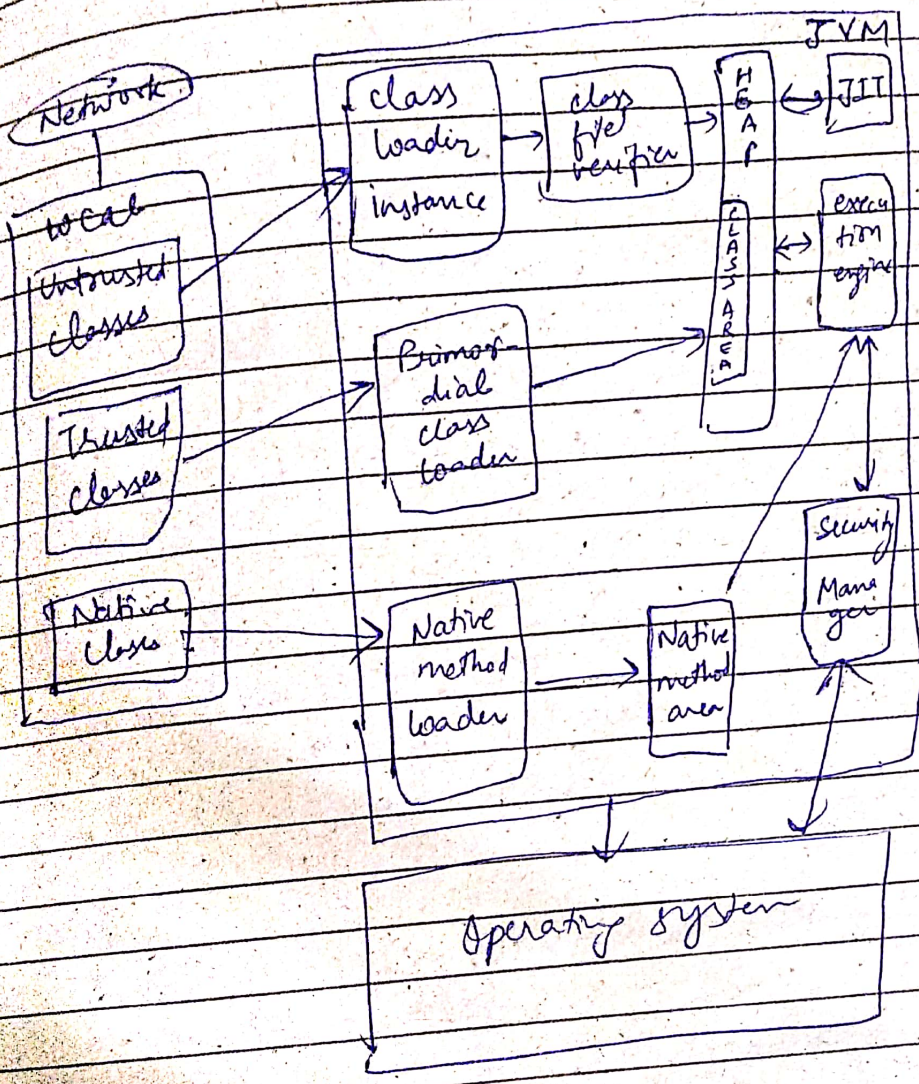
These areas are private to the owning thread. No thread can access the pc register or Java stack of another thread. So from these data area are exclusive to each thread.

JVM is a runtime env. in which our code runs.

Java

DATE: / /
PAGE

Components of Java or execution environment of JVM



Class loader : It locates or search the class file and load the classes into the JVM.

Primordial class loader : It loads the trusted classes (system classes found on the boot class path) & it is an integral part of the JVM.

Class loader Instance : It loads the untrusted classes from the local file system or from the network (internet) (computer) & passes them to the class file verifier.

class file verifier : It checks the untrusted class file & also it checks the size & structure of the class file & also check run time characteristics (stack overflow, divide by zero etc.)

Native method loader : It loads the native code which is stored inside the native method area for easy access.

Heap : It contains memory which is used to store objects during the execution.

Execution engine : It is a virtual processor that executes the byte code. It has virtual registers, stacks etc. It performs memory management, thread manager, call to native method etc.

Security Manager : It enforces access control at run time. Application developer can also implement their own security manager.

JVM Instruction Set

A JVM instruction consists of an opcode specifying the operation to be performed, followed by zero or more operand and the embodying value to be operated upon.

The JVM checks that the code satisfies the static & structural constraints at linking time using a class file verifier.

Constant pool : A class file keeps all its symbolic references in one place which is at the constant pool. Each class file has a constant pool & each class or interface loaded by the JVM has an internal version of its constant pool which is known as run time constant pool.

Types of JVM Instruction set:

(1) Stack operation & constants can be pushed onto the stack either by loading them from the constant pool with the LDC instruction or with special shortcut instruction where operand is encoded into the instruction (iconst 0, bipush)

istore_1 pops the value from the top of operand stack & store it as local variables.

(2) Arithmetic op: The instruction set of JVM distinguish its operand type & using the different instructions to operate on values of specific types. e.g. ADD, SUB, MUL.

(3) Control flow Instructions: There are branch instructions like if_icmpeq, which compare two integers for equality. There is also a jsr instructions (Jump to subroutine) & RET pair of instructions that is used to implement the finally

Scanned by CamScanner

clause of a try catch block.

④ Field access : The value of an instance field may be retrieved with getfield & return with putfield. For static fields there are getstatic & putstatic counter parts.

⑤ Method invocation : Static method may be called using invoke static method/ instruction.

super class methods & private methods are invoked with invoke special instruction.

⑥ Object allocation : Class instances are allocated with the new instruction. Arrays of basic types like int[] or int[][] with the newarray.

⑦ Conversion & type checking : For stack operands of basic type there exists casting operations like f2i which converts a float value to an integer. The validity of a typecast

may be checked with a check cast instruction :

Format of instruction description

1) Operation - Here mnemonic = opcode. Its opcode is in the numeric representation is given in both decimal & hexadecimal forms. Only the numeric representation is actually present in the JVM code in a class file.

2) Operand stack - It is a longer description detailing the constraints on the operand stack contents of constant pool entries, the operation performed that contain the type of the results etc

3) Linking exception - If any linking exception may be thrown by the execution of this instruction, they all are set off one to align in the order in which they must be thrown.

Java

4) Run time exception - It any run time exception like ÷ 0. can be thrown by the execution of an instruction they are set off one to align in the order in which they must be thrown.

Why JVM is known as an Emulator & Interpreter.

It is a hardware or software that enables one computer system to behave like another computer system or an emulated program is a simulation program that run on some existing system to make that system appear to be a something else.

JVM is a program that executes on wide variety of different computer systems.

It simulates the execution of virtual process whose instruction set is defined in JVM. This emulation allow pgm written in Java programing lang. to be executed on any processor for

which JVM has been written.

Java interpreter must also make our computer of the bytecode file to believe that they are running on a real m/c. It does this by acting as a intermediatory b/w the virtual m/c & the real m/c, & then the Java interpreter execute the code.

Java

Java Security

Different expectations of the term SECURITY might let us to expect that Java programs would be

1) safe from malevolent or malicious program. & Programs should not be allowed to harm a users computing environment. This includes trojan horses as well as harmful programs that can replicate themselves (computer viruses)

2) Non-intrusive & Programs should be prevented from discovering private information on the host computer or the host's computer network.

3) Authenticated & The identity of parties involved in the program should be verified.

4) Encrypted & Data that the program sends and receives should be encrypted.

5) Audited : Potentially sensitive operations should always be logged.

6) well defined : A well defined security specs. should be followed.

7) Verified : The rules of the operation should be set & verified.

8) Well behaved : Program should be prevented from consuming the too many resources.

Security Promises of The JVM.

1) Every object is constructed/created exactly once before it is used.

2) Every object is an instance of exactly one class, which does not change throughout the life of the Object.

3) If a field or method is marked private, then the only code that ever accesses it, is found within the class itself.

4) Field & methods marked protected are used only by code that participates in the implementation of the class.

5) Every local variable is initialised before it is used.

6) It is impossible to underflow or overflow the stack

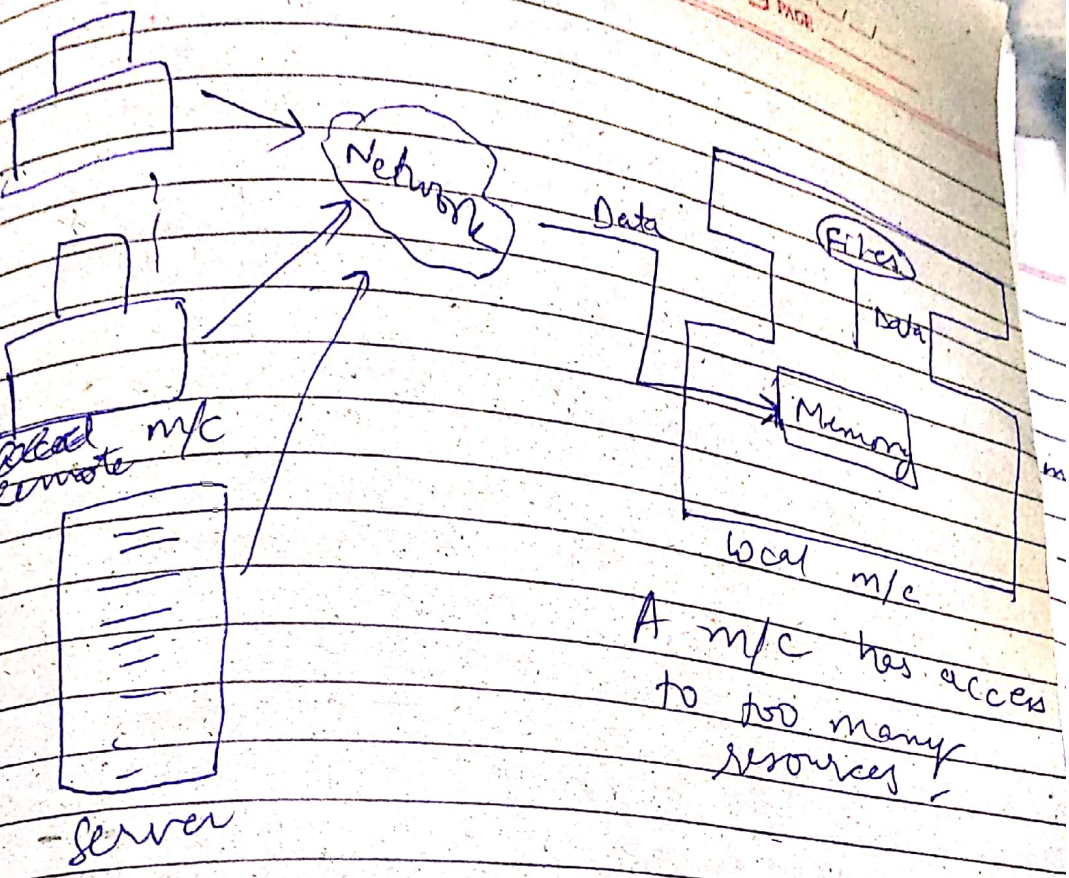
7) It is impossible to read or write past the end of the array or before the beginning of the array.

8) To change the length of array is impossible.

9) final methods cannot be overriden & final classes cannot be subclassed.

10) Attempts to use a null reference as the receiver of a method invocation or the source of a field causes a null pointer exception to be thrown.

Java Sandbox Security Model



A m/c has access to too many resources!

The idea behind this model is that when you allow a program to be hosted on your computer, you want to provide an environment where the program can be run & limit the resources that can be accessed by the applet, ie, you want to confine the program play area within the certain boundaries. Java sandbox model is responsible for protecting a non-trusted resources & it does so at a no-of levels.

Sandboxes are used when executable code comes from the unknown or untrusted resource & allow the user to run the untrusted code safely.

Resources
1) Internally the user m/c has access to its local memory to computer's RAM.

2) Externally it has access to its own file system & to other m/cs on the local network.

3) For running an applet it also has access to a web server which may be on its local network or may be on the internet.

4) Data flow from ~~this entire model through the network~~ from one user m/c through the network & possibly to the disk.

5) Each of these resources need to be protected & those protection form the basis of Java security model.

Java.

Objective
Need & scope
Future scope
Usefulness

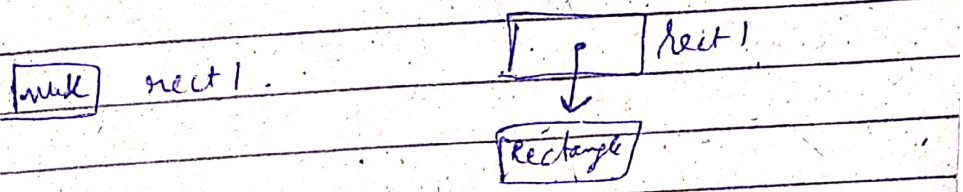
DATE: / /
PAGE

Class defines the state & behaviour of the basic program component known as Object

class create object and objects use methods to communicate b/w them. In java programming the data items are called fields & the ()s are called methods.

Creating the objects.

```
rectangle rect1;
rect1 = new Rectangle()
```



An object in java is essentially a block of memory that contains space to store all the instance variables. Creating an object is also known as instantiating an object.

Objects in java are created using the new operator. The new operator creates an object of the specified class & returns a reference to that object.

The 1st statement declares a variable to hold the object reference & the 2nd one actually assigns

Java supports a spl. type of method called a constructor that enables an object to initialize itself when it is created.

~~Java~~ constructors have the same name as the class itself. They do not specify a return type not even a void because they return the instance of the class itself.

Method Overloading

In java programming it is possible to define 2 or more methods within the same class that shares the same name as long as their parameter declarations are different. In this case the methods are said to be overloaded methods & the process is known as method overloading.

Java

DATE: __/__/____
PAGE ____

Nesting of Methods

```
class nesting
{ int m, n;

  nesting (int x, int y)
  {
    m = x;
    n = y; }

  int largest ()
  {
  if (m >= n)
     return (m);
     else
        return (n);
  }

  void display ()
  {
    int large = largest ()
    System.out.print ("Large = " + large);
  } }

class nesting test
{ psvm (String xy[])
  { nesting nest = new nesting (50, 40);
    nest.display ();
  }}
```

DATE:
PAGE

WKT, a method of a class can be called only by an object of that class or by the class itself using the static method. However, there is an exception to this. A method can be called by using only its name by another method of the same class. This is known as nesting of methods. In simple words, the calling of one method inside the another method of the same class is known as nesting of methods.

The variables & methods that are declared inside the class are known as instance variables & instance methods. This is busy everytime the class is instantiated, a new copy of each of them is created.

They are accessed using the objects with dot operator.

Static members: When we want to define a member that is common to all the objects & accessed without using a particular object, that member belongs to the class as a whole rather than the objects created from the class. The member that are declared static is also known as class variables & class methods.

```
class usestatic
{ static int a=5;
  static int b;

  static void meth (int x)
  { sop ("x = " + x);
    sop ("a = " + a);
    sop ("b = " + b);
  }

static
{ sop ("static block initialized");
  b = a * 4;
}

public static void main(String args[])
{ meth (42); }
```

Static method have several restrictions:

1) They can only call other static methods.
2) They can only access static data.
3) They cannot refer to this or super key-word in any way.

~~Using~~

Using object as parameter

```
class test
{ int a, b;
  test(int i, int j)
  {
   a = i;
   b = j;
  }
  boolean equals(test o)
  {
   if (o.a == a && o.b == b)
       return true;
   else
       return false;
  }

class Pass Ob
  { PSVM(String [] args)
  {
   test obj1 = new test(100, 22)
   test obj2 = new test(100, 22)
   test obj3 = new test(-1, -1)

   SOP("obj1 == obj2" + obj1.equals(obj2))
   SOP("obj1 == obj3" + obj1.equals(obj3))
  }
}
```

Java

Argument passing using call by value & call by reference

In call by value approach, it copies the value of an argument into the formal parameters of the calling () or subroutine. Therefore any changes made to the parameter of the subroutine have no effect on the argument. In call by reference approach, a reference to an argument not the value of the argument is passed to the parameter inside the subroutine, this reference is used to access the actual arguments specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

```
class test
{ void m1(int i, int j)
    { i *= 2;
      j /= 2;
    }
}

class callbyvalue
{ ps vm()
  {
```

Test ob = new Test ( );
int a = 15, b = 20;
SOP ("a & b before call " + a + " " + b);
ob.ml (a, b);
SOP ("a & b after call " + a + " " + b);
}
}

class Test
{ int a, b;
void m1 (Test o)
{
o.a *= 2;
o.b /= 2;
}
}

Test (int i, int j)
{
a = i;
b = j;
}

class callbyreference
{
PSVM ( )
{
Test ob = new Test (15, 20);
SOP ("a & b bfor call " + ob.a + " "
+ ob.b);

ob.m1(ob);
SOP("a & b after call "+ob.a+ob.b);
}
}

Returning Objects

class Test
{ int a;
Test(int i)
{
a = i; }

Test incrByTen()
{ Test temp = new Test(a+10);
return temp;
}
}

class AceOb
{ PSVM()
{ Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
SOP(ob1.a);
SOP(ob2.a);
ob2 = ob2.incrByTen();
ob2 = ob2.incrByTen();
SOP(ob2.a);
}}

Recursion

It is a process in which a () calls itself again & again until it meets a certain condition.

The function which calls itself is called recursive ().

```
class factorial
{ int fact (int n)
  { int result;
    if (n == 1)
      return 1;
    else
      result = n * fact (n-1);
    return result;
  }
}

class recursion
{ PSVM ()
  { factorial f1 = new factorial();

    SOP (f1.fact(3));
    SOP (f1.fact(4));
  }}
```

Nested & Inner classes

It is possible to define a class within another class & such classes are known as nested classes.

The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within the class A, then B class does not exist independently of A.

A nested class has access to the members including the private members of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

```
class outer{ {
    int outer_x = 100;
    void test()
    { Inner inner = new Inner();
      inner.display();
    }

class Inner
    { void display()
      { sop("display outer x "+outer_x);
    } } }
```

```
class Innerclassdemo
{
  psvm()
  {
    outer outer=new Outer();
    outer.test();
  }
}
```

Java

DATE: / /
PAGE

Dynamic Method Dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time rather than the compile time. When an overridden method is called through a superclass reference, java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.

∴ This determination is made at the run time & It is called run time polymorphism.

In another words it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed. ∴ If a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable different versions of the methods are executed.

If super constructor is not used then the default or parameterless constructor of each super class will be executed.

```
final int size = 20;
class A
  { final void m1()
      { SOP("Final method");
      }
  }
class B extends A
  { void m1()
      { SOP("Hello");
      }
  }
final class A
  {
  }
class B extends A
```

Using final keyword to prevent method overiding.

Java supports the concept of finalization which is opposite to initialisation. A garbage collection system automatically free up the memory resources used by the objects. But objects may hold other non object resources such as file descriptors or window system ~~encrypt~~ used. The garbage

collector cannot free these resources. In order to free these resources we use a finalize() method.

Java calls this finalizer method, whenever it is about to reclaim the space for that object.

Abstract method & Abstract classes

Use of wrapper class

DATE / /
PAGE

String builder is identical to string buffer with the one imp difference that string builder is not synchronized which means that it is not thread safe.

The adv of string builder is faster performance. However in case you are using multi threading, you must use string buffer rather than string builder.

Java

DELTA Page No.
Date / /

Controls in AWT

Control are the components that allow a user to interact with your application in the various ways. For eg., a commonly used control is the push button.

The AWT supports following type of controls:

1) Label
2) Push button
3) Check box
4) Choice list
5) List
6) Scroll bar
7) Text editing (Text field & text area)

These controls are the subclasses of component class.

Adding & Removing controls

Component add (Component obj)

void remove (Component obj);

removeAll ( )

Label ( ) throws HeadlessException

The add & remove method are defined by the container class.

Headless Exception occurs when an attempt is made to instantiate a GUI component in a non interactive environment.

Label

Ls is a passive component.

Label() // default constructor

Label (String str)

Label (String str, int how)

↳ 3 integer constants can be passed
Label.LEFT
Label.RIGHT
Label.CENTRE

void setText (String str)
String getText()

void setAlignment (int how)
int getAlignment()

```
class Label extends Applet
{
  public void init()
  {
    Label one = new Label("One");
    Label two = new Label("Hello");
    add(one);
    add(two);
  }
}
```

Buttons

```
Button()                    // default constructor
Button(String str)
void setLabel(String str)
String getLabel()
```

```
public class ButtonDemo extends Applet
                implements ActionListener
{ String msg = " ";

Button yes, no, maybe;

public void init()
{ yes = new Button("Yes");
  no = new Button("No");
  maybe = new Button("Undecided");

add(yes);
add(no);
add(maybe);

yes.addActionListener(this);
no.addA"  "  (this);
maybe." "  " ; }

public void actionPerformed(ActionEvent ae)
{ String str = ae.getActionCommand();
  if (str equals("Yes"))
  msg = "You passed Yes"; }
  else if (str equals("No"))
  { msg = "You passed No";
  }
  else {
  msg = "You passed undecided";
  } }
```

repaint();
public void paint (Graphics g)
{
g.drawString ("msg", 6, 100);
}}

Checkbox

Checkbox()

Checkbox(String str)

Checkbox(String str, boolean true)

boolean getState()
void setState(boolean true)
String getLabel()
void setLabel(String str)

public void init()
{ Checkbox winXP = new Checkbox("Windows", true);

Checkbox("Linux", null, true);

Checkbox("Mac OS");

DELTA Page No.
Date / /

checkbox(String str, CheckboxGroup cbg, boolean true)

checkbox win, linux, vista;

checkboxGroup cbg;

public void init() {

win = new Checkbox("XP", cbg, false);

linux = " " ("Linux", cbg, true);

Vista = " " ("Vista", cbg, false);

Choice & List (Pop up list of items)

Choice
- only default constructor can be made.

List
- List()
- List(int numRows)
- List(int numRows, boolean multipleSelect)

scrollbar()

scrollbar(int style)
- Scrollbar.HORIZONTAL
- Scrollbar.VERTICAL

DELTA Page No.
Date / /

TextField()

TextField(int numchars)

TextField(String str)

" (String str, int numchar)

TextArea

TextArea()

TextArea(int numlines, int nbm.chars)
TextArea(String str)

TextArea(String str, int numLines, int numchars)

TextArea(String str, int numLines, int numchars
int SBars)

scrollbars_BOTH

SCROLLBARS_NONE

SCROLLBARS_HORIZONTAL_ONLY

SCROLLBARS_VERTICAL_ONLY