JAMES GOSLING - Father of Java 19/05/2022

Name of the Experiment ..... Page No.:

Experiment No..... Date :

What is Java?

* General-purpose
* Object-Oriented
* Platform independent
* Concurrent (Multi-Threading)
* Very fast

} programming language.

Features

* familiar syntax → Syntax like C & C++
* Simple & Safe
  * C & C++ → Manipulate memory to free up space.
  * (Garbage collection) → Java → Automatic memory management
* Secure.
* Java API (Java libraries)
* FREE

Compilation

Machine language (Eg: 000000 00001 00010 00110 000000 100000)
add

* Computer understand instructions
* Program ~ set of instructions
* Instruction ~ 0s & 1s
* Machine language (machine code / native code)

Low level languages

Assembly language

* add, $s0, $s1, $s2

High level languages

* FORTRAN, C, C++, Java, C#
* Use english-like words, math notation, punctuations
* Hide low level details.

assembly language → assembler (It is a program) → Machine language

BIT

Source code → [compiler] → Target Language (Machine lang / other (Java bytecode)) → [CPU] → results

compilation

execution

target lang can be: machine code, bytecode, programming language

Core compilation Operations

→ Verifying syntax and semantics of source code.
→ Code optimization.
→ Generate Machine code.

Platform Dependency Reasons

OS:
- Executable file format ~ PE (windows) [Portable Executable]
  ELF (Linux) [Executable & linkable format]
- System calls, eg: file opening.

Hardware
Processor ~ x86 Vs ARM inst set.

Interpreter

source code → [compiler] → Machine code → [CPU] → results
(compile + execute)

source code → [Interpreter] → results

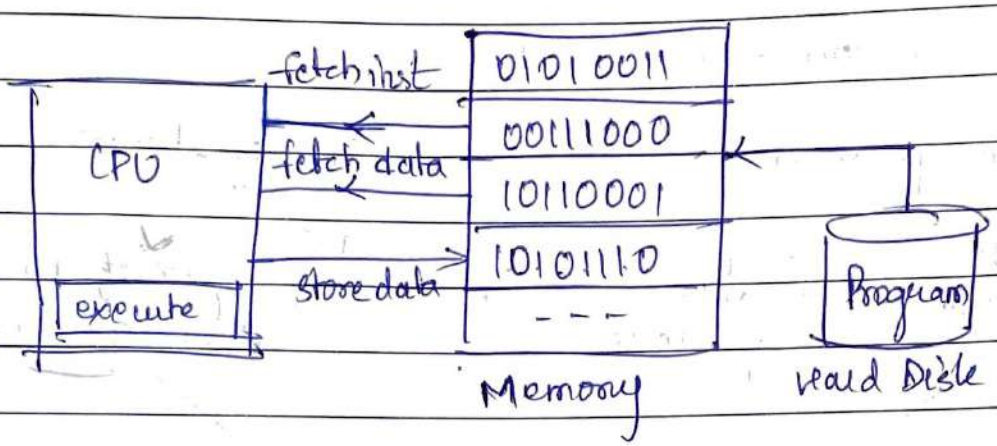
CPU → (Program) Interpreter ← code
results

Name of the Experiment ..... Page No.:

Experiment No..... Date :

* Interpreter is a Virtual Machine that simulates CPU
* fetch and execute cycle.

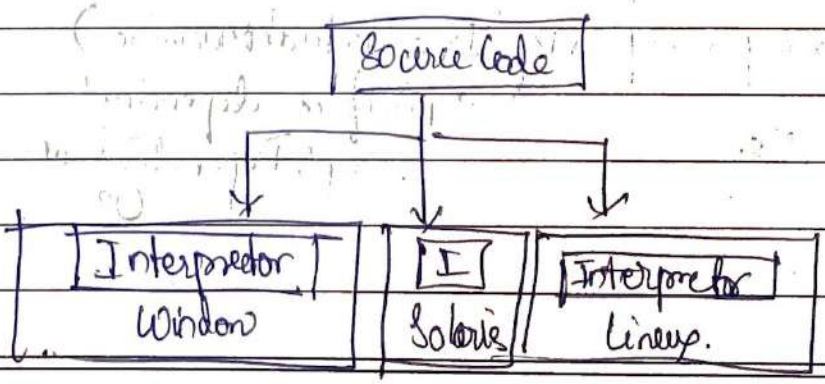
CPU's fetch and execute cycle.



Interpreter's fetch and execute.

* fetch next program statement } same as compiler.
  → understand the statement }
* Execute precompiled machine code in its library

Platform Independance ⟹



BIT

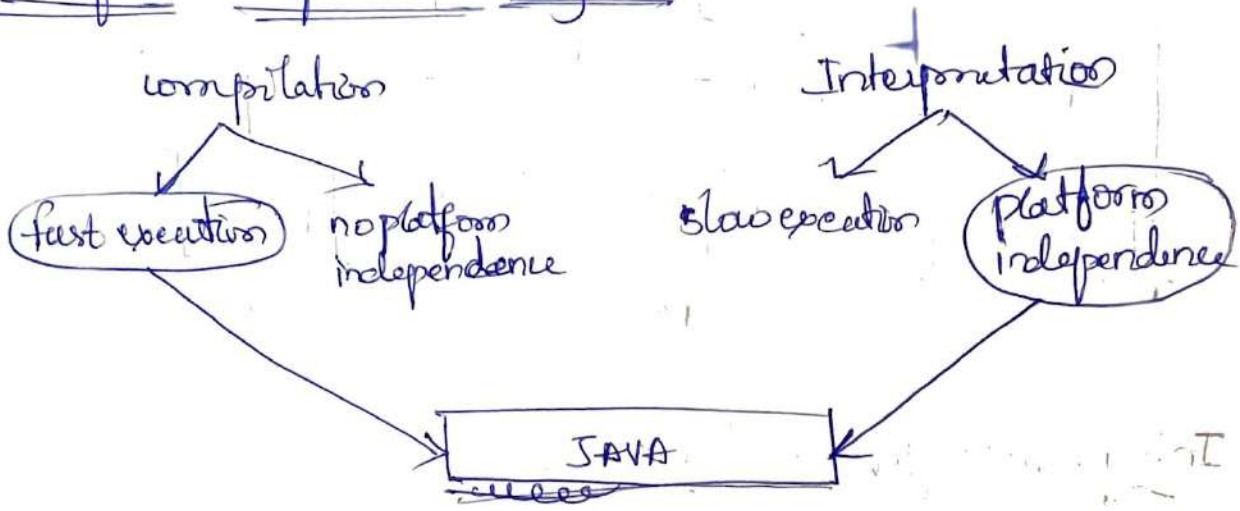
Advantages

* Platform independence
* No compilation step
* Easier to update

limitations

* Slower → source code is reinterpreted every time
  ↳ costly memory access.
* ~~Source~~ Interpreter is loaded into memory along with the source code

Platform Independance in java



source code
↓
Java Compiler
↓
Java bytecode
↓
Java interpreter JVM (Java Virtual Machine)
↓
results.

↳ Platform dependent
say: Dependent on OS

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Hello.java — Source code.

↓

compilation — javac Hello.java // command

↓

Hello.class (Java bytecode)

↓

JVM (Windows) | JVM (Linux) | JVM (OS X) | JVM (Solaris) — java Hello // command

JVM

JVM ⇒ Abstract Computing Machine

scala groovy

↓ ↓

Compiler

↓

Java(bytecode)

↓

JVM

↓

Results

* Instruction set → Java bytecode
* Manipulates memory at runtime (execution)

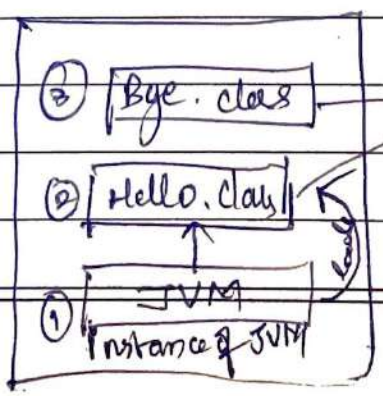
Responsibilities

* Loading & interpreting Java bytecode
* Security
* Automatic memory management

(X) What happens when we execute a Java code [JVM Instance] Hello.class.

java Hello →

java Bye →



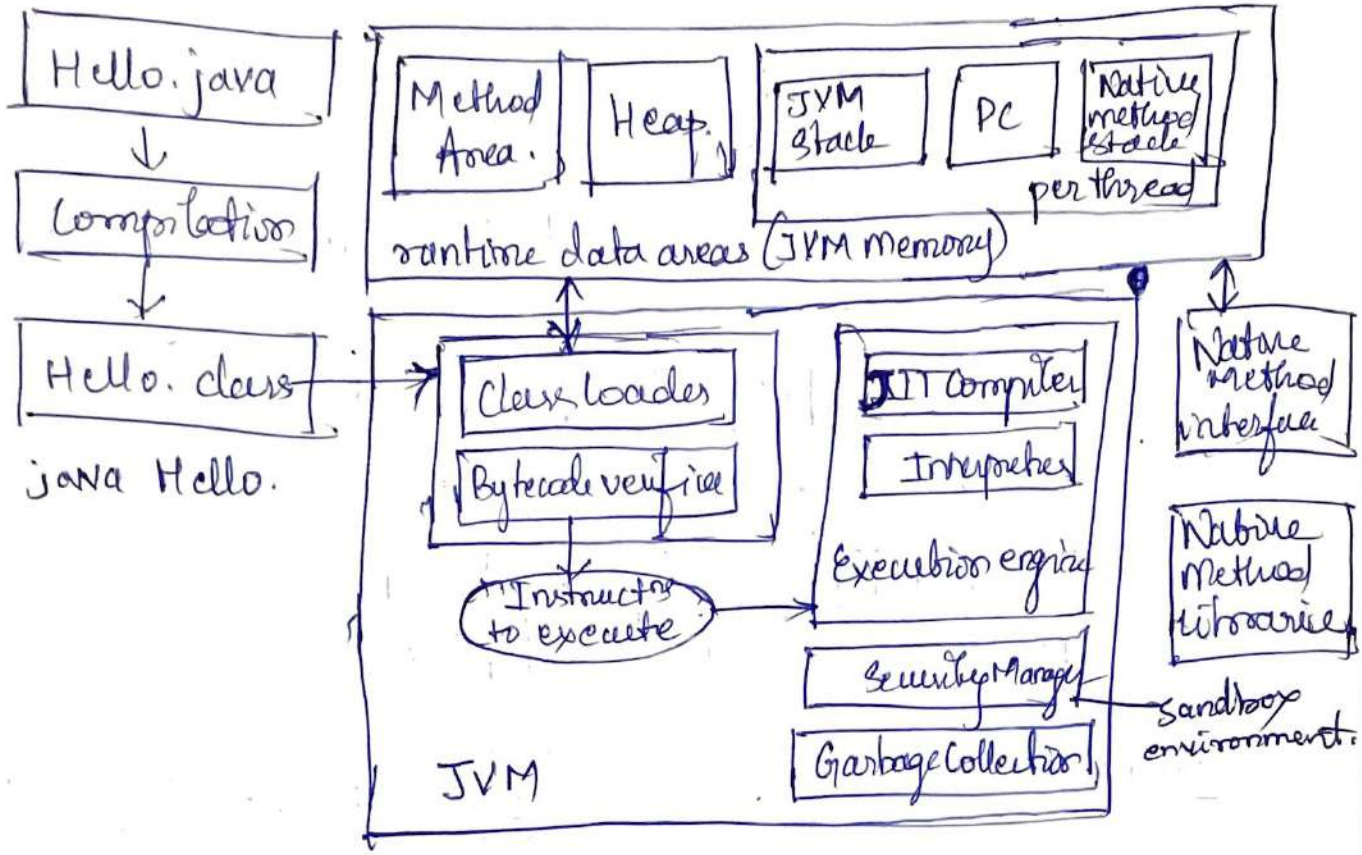
2 instances can be run

Source file creates an instance of JVM

i.e., .class file.

BIT

JVM Architecture



Performance

→ Bytecode interpretation is much faster.
* Java bytecode is compact, compiled and optimized

→ Just-in-time (JIT) compilation / dynamic compilation
* JVM identifies frequently executed bytecodes "hot spots"
* ~~JIT compiler~~ Pass it on to JIT compiler
* JIT compiler converts "hot spots" to machine code
* cache the machine code
* cached machine code → faster execution

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Java SE [Std Edition]

Java Software family

* Java Standard Edition (Java SE)
  - → Stand alone applicat^n for desktops & servers
  - → Ex:- Hospital management system.
* Java Enterprise Edition (Java EE) (built on Java SE)
  - → Large scale applicat^ns for servers
  - → Ex:- e-commerce website
  - → Built on Java SE.
* Java Micro Edition (Java ME) (built on Java SE)
  - → Applicat^ns for resource-constrained devices
  - → Uses subset of Java SE

Jakarta EE

Java SE

Specificat^ns

* Java language Specification (JLS) ⇒ Syntax & semantics of lang.
* Java Virtual Machine Specificat^n. ⇒ JVM
* Java API specification ⇒ Java Lib

Implement^ns

Open JDK

Oracle JDK | Oracle Open JDK (free) | Adopt open JDK | Amazon Corretto.

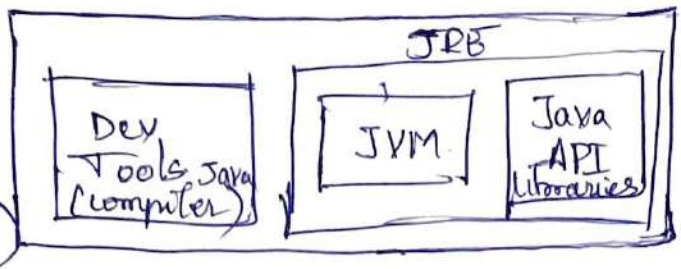
JDK (Java Development kit)

BIT

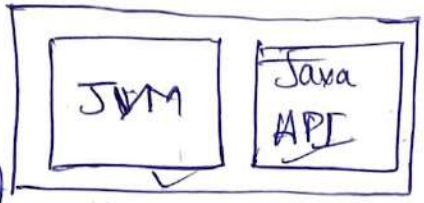
2) Open JG
D) Hotspot JVM

JSR - Java specifical° request ⇓ defines features of a specific JSR

JDK
⇓
Java SDK
(Software Dev Kit)



JRE
Java Runtime environment



Only to run the code / not to develop

JDK will have the specifications of Java SE which are been developed by different company.

All Java executable will be in "bin folder"

Environmental Variable

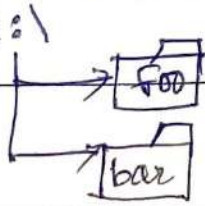
An environmental variable is simply a variable whose value is visible to entire environment with in the system something like a global variable.

Classpath (required only during command line useage)

It is a path on the file system for locating the java classes and can be more than 1 path too.

It is used both at compilation & execution time of java programs.

Ex:- ∘ java Hello classpath = C:\foo; C:\bar

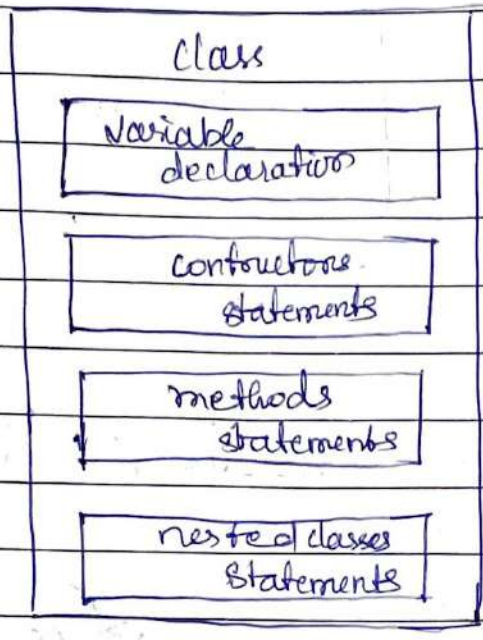


Hello.class & executes this path

Name of the Experiment ..... Page No.:

Experiment No..... Date :

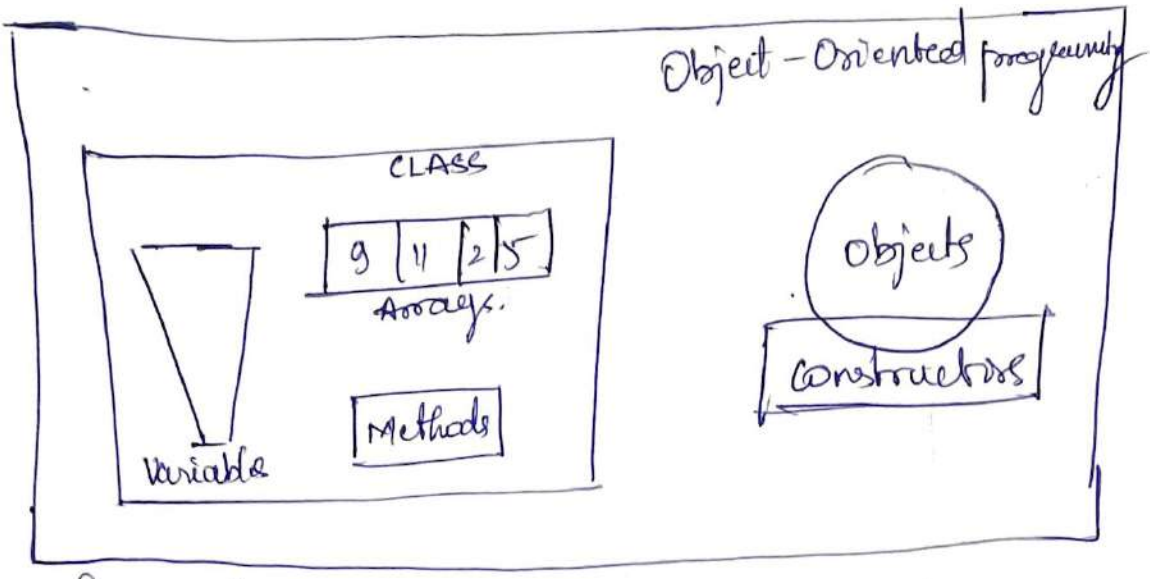
Java Program : Structure.



main() method

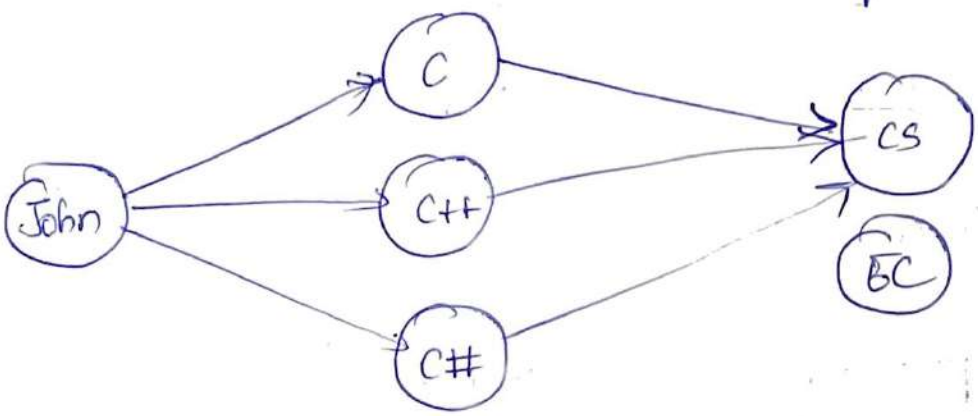
→ The program starts with main() method
→ Java HelloWorld
  * JVM loads bytecodes of HelloWorld.class into memory
  * invokes main()
→ Must be declared as public, static and void
→ From main() we invoke other code
→ Program ends with main() method

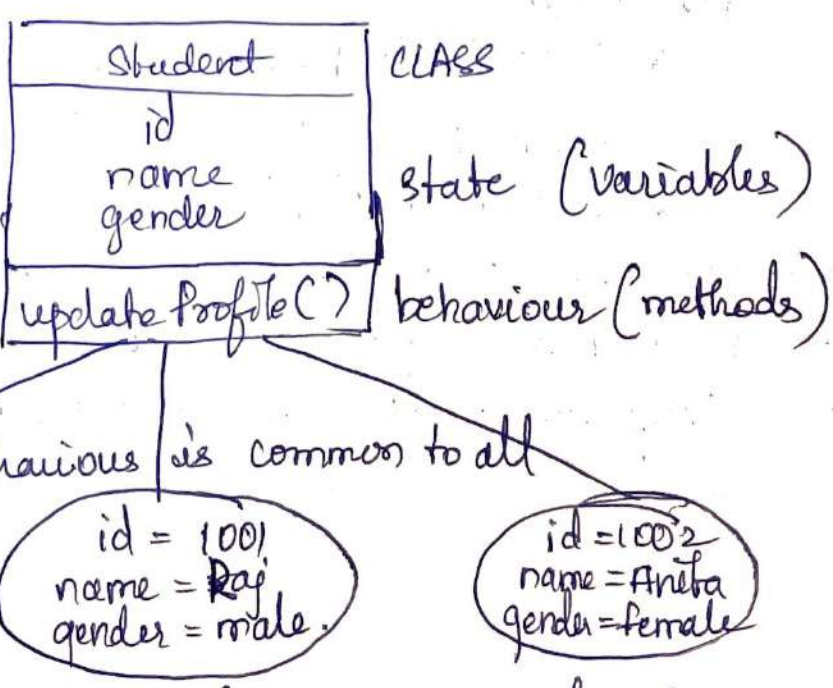
20/05/2022



Class & Objects

Ex Student register course offered by department





Name of the Experiment ..... Page No.:

Experiment No..... Date :

Ex

```
class Student {
    //Variables declared
    int id;
    String name;
    String gender;

    //method definitions
    boolean updateProfile (String newName) {
        name = newName;
        return true;
    }
}.
class StudentTest {
    public static void main (String[] args) {
        //(1) Creating a new student object
        Student s = new Student();

        //(2) Setting the student's state
        s.id = 1000;
        s.name = "joan";
        s.gender = "male";

        //(3) updating profile with correct name
        s.updateProfile ("John");
    }
}.
```

BIT

Naming rules

a) First character => letter, underscore, $
   - post 1st character => letter, underscore, $, numbers.
b) No reserved keywords. (around 50)
c) Camle casing.

System.out.print(" ");

System → class
out → variable that corresponds to an object through that object we are invoking print method.

Variables (stores state information)

[Global int = if variable not defined, the default value will be = 0]

primitive variable
int (id) = 1000; // Holds Integer
String name = "John"; // Holds characters
Student (s) = new Student(); // Holds object (Student Object)
object reference

* Variables are static in nature hence
  JAVA IS A STATICALLY TYPED LANGUAGE → static type checking (done during compilation)
* DYNAMICALLY TYPED LANGUAGE (JAVASCRIPT) → Dynamic type checking (done at runTime)

Declaration :- <type> <name> [= literal (or) expression];
raw data

literal → raw data.

Ex :- int id = 1000; → literals.
boolean flag = true;
String name = "John"

Name of the Experiment ..... Page No.:

Experiment No..... Date :

expression → evaluated to single value

Ex: * int id = x; (say x = 1)
* int id = x+y;
* Student s = new Student()

} declarat" statements

can appear anywhere in the classes.

* Variable ~ Something whose value can be changed.
* Assignment statement are used to update a value.

Ex:- <name> = literal (or) expression

→ Count = 23;

→ count = x+y;

* Cannot appear at class-levels // only in the methods.

Assignment

1) Create a class → CurrencyConverter

US dollar Global

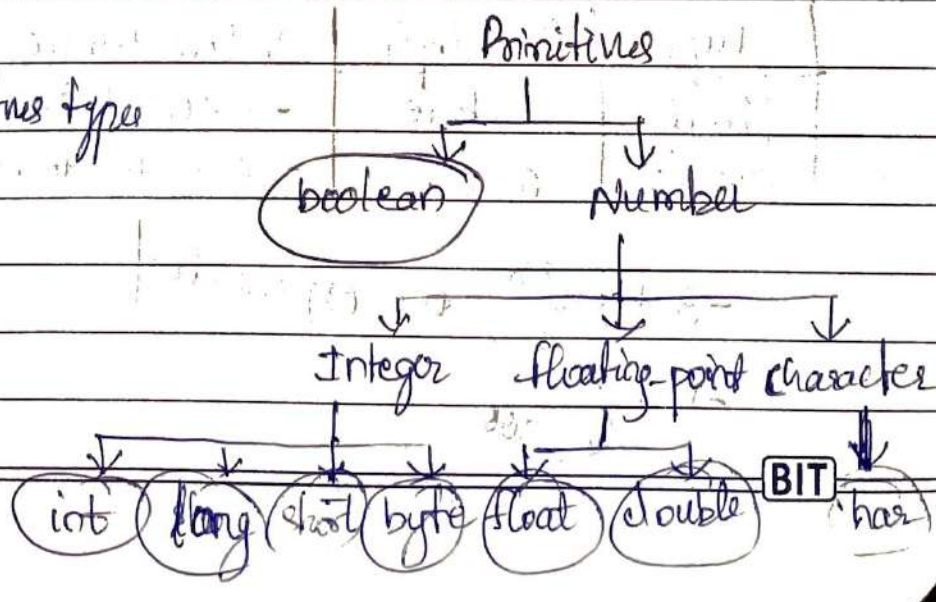
~~US dollar~~ = 63 Rupees. = 1 US dollar

~~Uti~~ 3 dirham = 1 US dollar

Datatypes

Primitive Types

There are 8 primitives types



Integer datatypes (Represent - sign 2's complement repres)

* Whole numbers (or) fixed-point number.

  Ex:- 65, -1000 etc

* byte, short, int, long

Type	Bit depth	Value Range	default value
byte	8 bits	-2^7 to 2^7-1	0
short	16 bits	-2^{15} to $2^{15}-1$	0
int	32 bits	-2^{31} to $2^{31}-1$	0
long	64 bits.	-2^{63} to $2^{63}-1$	0.

There are only 2 literals in Integer.
i.e., int literal, long literal.

int intHexa = 0x0041 ; = $16^0 \times 1 + 16^1 \times 4 + 0 + 0$

int intBinary = 0b01000001 ; $2^0 \times 1 + 2^0 \times 0 + 0 + 0 + 0 + 0 + 2^6 \times 1$

int intOctal = 0101 ; $8^0 \times 1 + 8^1 \times 0 + 8^2 \times 1 + 8^3 \times 0$

Floating point datatypes (Binary floating point format / IEEE 754 floating point scheme)

* Real numbers, Ex:- 3.14, -0.014
* float, double

Type	Bit depth	Value Range	Default	Precision
float	32 bits	-3.4538 to 3.4538	0.0f	6-7 decimal digits
double	64 bits	-1.76308 to 1.76308	0.0d	15-16 decimal digits

-3.4×10^{38} E = exponential

-1.7×10^{308}

$\updownarrow$

1.7×10^{308}

Name of the Experiment ..... Page No.: ☐

Experiment No..... Date : ☐☐☐☐☐☐

* Usually stick with int & double.
* Use byte, short, float only if memory saving is important

Ex : float gpa = 3.8f → must. (if not throws compilation error)

double gpa = 3.8

by default it takes double

hence it throws compilation error if "f" is not included during the declaration of float variable.

Binary floating point format not accurate

say • SOP(0.1+0.2) o/p = 0.30000000000000004

(Absolute accuracy is involved) Hence use BigDecimal class to get accurate results.

Ex :- BigDecimal first = new BigDecimal("0.1");
BigDecimal second = new BigDecimal("0.2");
SOP(first.add(second));

Character Datatypes

* Single letter characters Ex :- 'A', '0', '$'
* char
  → char degree = 'B';
* Data rep - 16 bit unsigned integers. (+ve only)

BIT

Type	Bit depth	Value range	Default
char	16 bits	0 to $2^{16}-1$ 0 to 65535	'\u0000'

⇒ Unicode ⇒ Rep all characters with corresponding number integer. i.e., in hexadecimal format called "code point" and then to integer

(UTF-16) encoding scheme.

⇒ 'B' ⇒ 0042 ⇒ 0000 0000 0100 0010 (66)

4 2

< 'B' , \u0042 , 66 >

this rep is called as unicode escape sequence

Other ways of declaring char variables

By. char charInt = 65; // o/p = A
char charHex = 0x0041; // o/p = A
char charBinary = 0b0100_0001; // o/p = A

vice versa is also possible

int intChar = 'A'; // o/p = 65

∴ char variable ← char literal ← unicode escape seq.
↖ int literal

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Boolean datatypes

* By default = it is false
* Bit depth = JVM specific (1 byte / 4 bytes) etc.
* Values = True / false
* Widely used in control statements.

Ex : boolean internation = true;

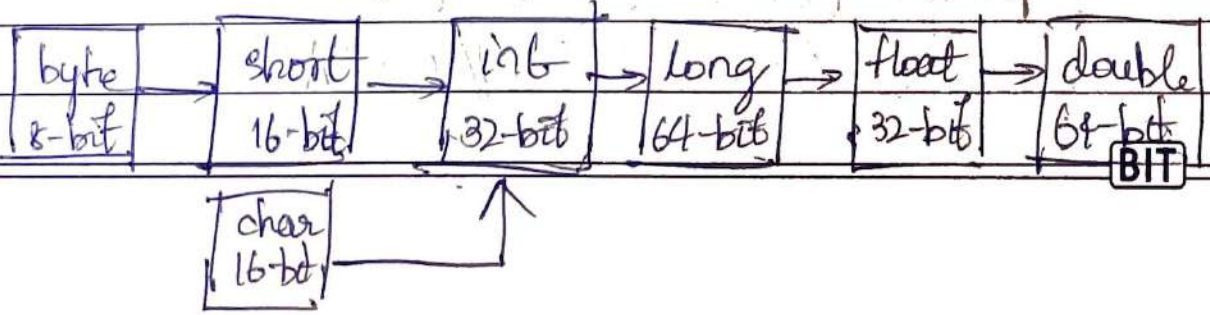
Variable - Type casting

* Assign variable or literal of one type to variable of another type.
  - → int ← long
  - → int ← byte.
* Only numeric - to - numeric casting is possible
* Cannot cast to boolean or vice versa.
* Implicit (or) explicit casting.

Implicit :-

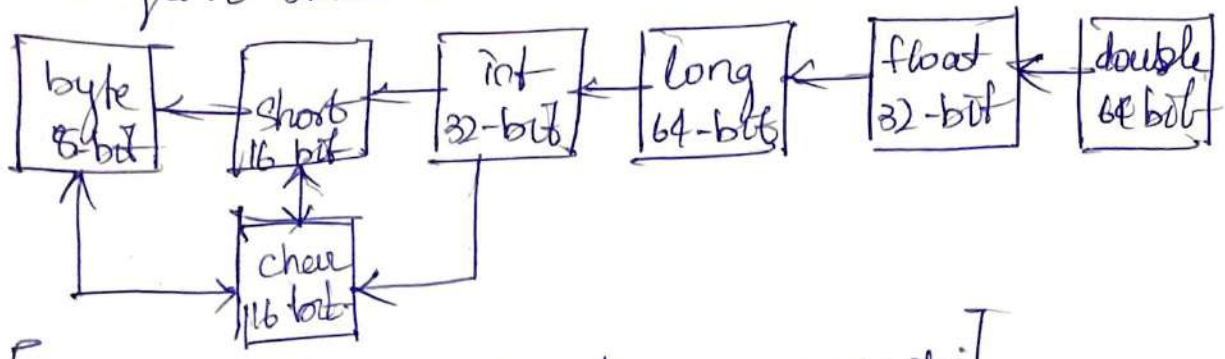
* Smaller to larger ~ widening conversation

Ex:- int x = 65;
long y = x; (implicit casting by compiler)



Explicit Casting

* Larger to smaller ~ narrowing conversion.



[casting with char is always explicit]

Ex:- long y = 42;
int x = (int) y;

byte b = 65
char c = (char) b; // c = 'A' (widening & Narrowing)

char c = 65; // c = 'A'
short s = 'A'; // s = 65

} within the range.

// out of range assignments.

* byte narrowedByte = (byte) 123456 // 64

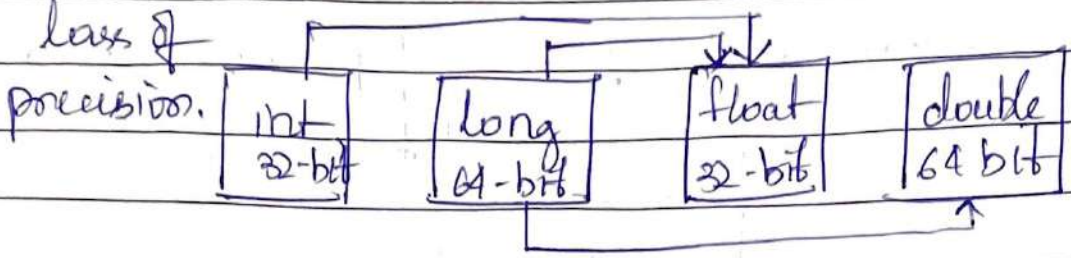
// Truncation.

* floating-point to integer/char will always truncate
* int x = (int) 3.14f; // x = 3
* int y = (int) 0.9; // x = 0
* char c = (char) 65.5; // c = 'A'

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Information loss Implicit casting




```
int oldVal = 1234567890;
float f = oldVal; // implicit cast
int newVal = (int) f ; //(1234567936)
```

Casting Use - Cases.

Implicit Casting

```
float f1 =3.133f;
float f2 = 4.135f
go(f1, f2)   // done by compiler itself.
```

```
go(double d1, double d2)
{ --- }.
```

Explicit casting

```
double avg = (2+3)/2 ; //2.0, not 2.5
double avg = (double) (2+3)/2 ;
```

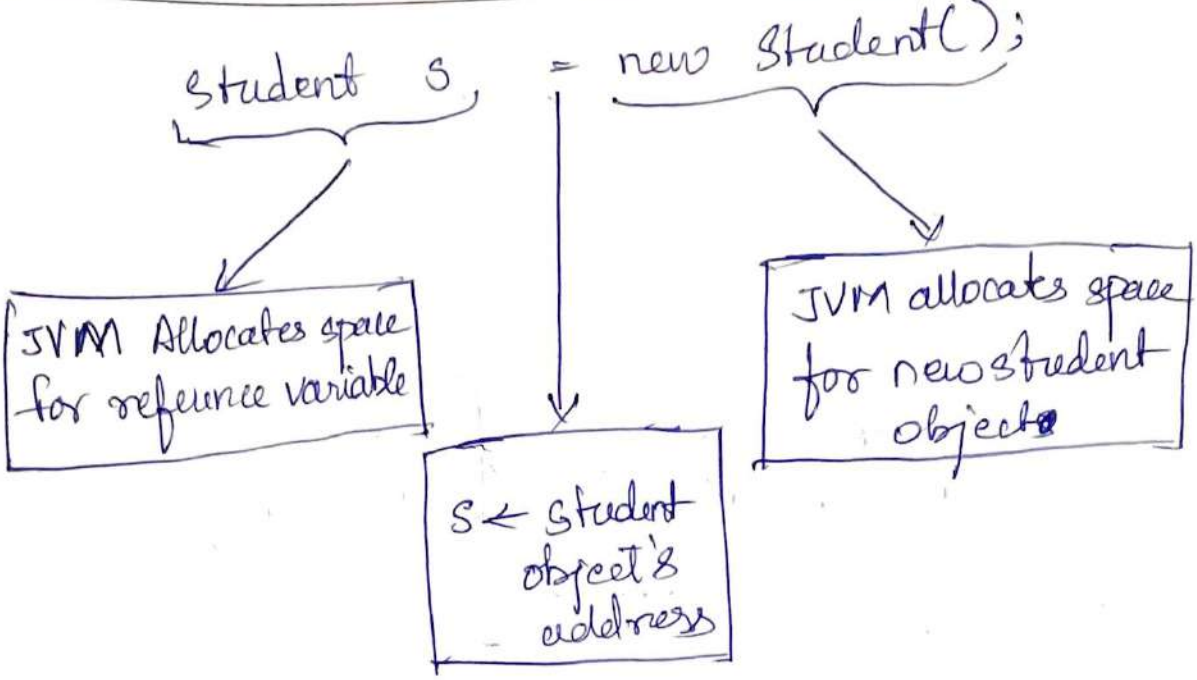
when the is issue with the range of the value.

Reference Variables

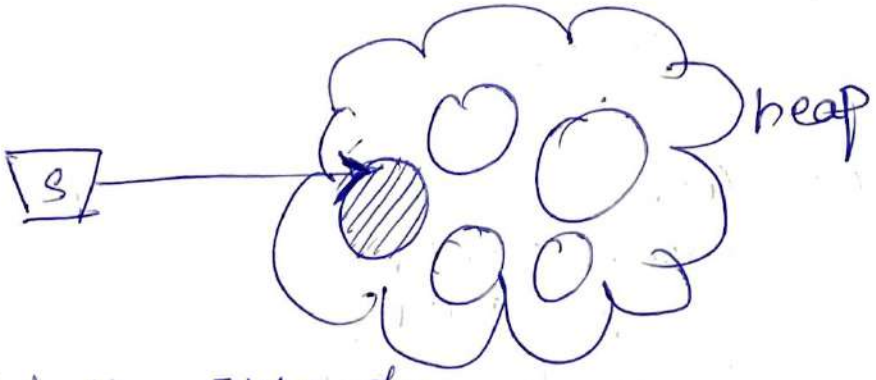
Student (s) = new Student()

objref ⇒ holds reference to a student obj in memory.

BIT



All objects are stored in "Heap memory"



* Bit depth ~ JVM specific
  (size of the obj doesn't matter)

* default value for obj ref = null;
  Ex Student s;

* s.updateProfile(); // nullPointerException.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Statements

* Command to be executed (ends with ;)
  - → Declare a variable
  - → change variable value
  - → Invoke a method
* change program state
* Involves one or more expressions.
* Expression ~ evaluated to single value
  - → Involves literals, variables, operators and method calls,
* Example
  - → int count = x * getCount();
  - → x, getCount(), x * getCount(), count = x * getCount();

compound exp

compound exp.

Types

* Declaration statements ex - int count = 25;
* Expression statements.
  - → count = 25; // assignment statement
  - → getCount(); // method invocation statement
  - → count ++; // increment statement
* Control flow statements.

```
if (count < 100) {
    ---
}
```

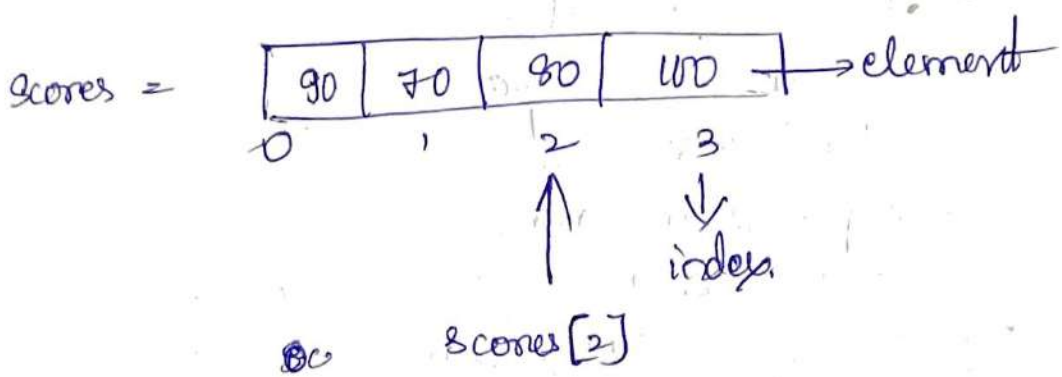
class level

not at class level

BIT

ARRAYS

Arrays is a container "object" that holds fixed# values of single type.

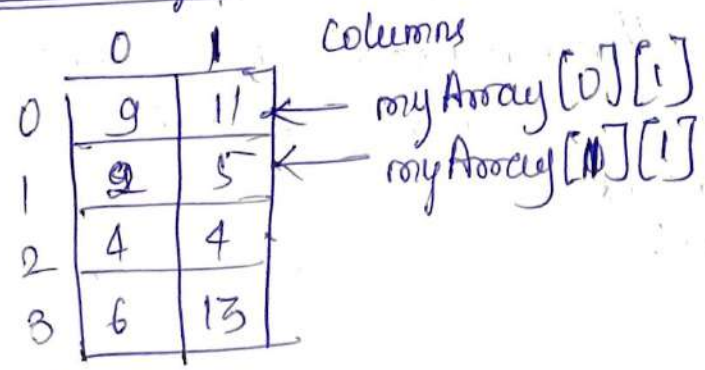


Ex :- 1) int[] scores = new int[4];
scores[0] = 95;
scores[1] = 90;
scores[2] = 85;
scores[3] = 100;

2) int[] scores = new int[] {90, 95, 85, 100};

3) int[] score = {90, 95, 85, 100};

2D Arrays



Name of the Experiment ..... Page No.:

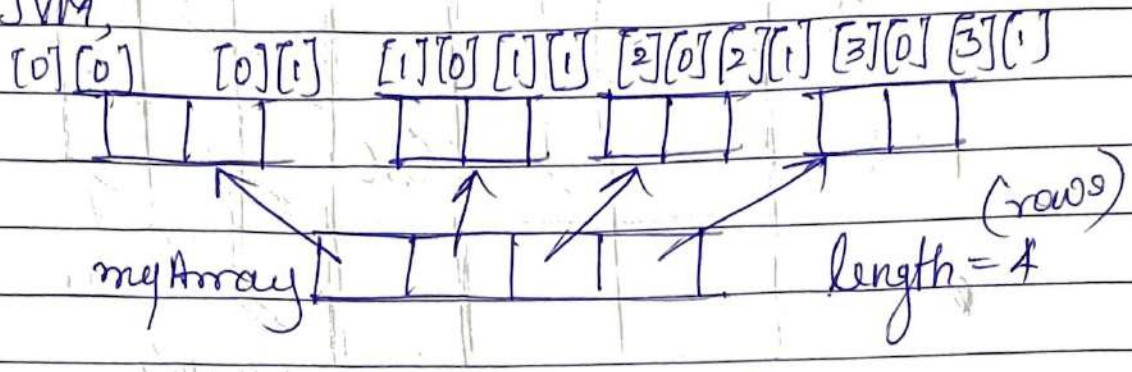
Experiment No..... Date :

Creating 2D Array

rows columns.

1) `int[][] myArrays = new int[4][2]`

in JVM,



type[] myArray → array of type
int[] myArray → array of int
int[][] myArray → array of array of int

Initialization → 1) `myArrays[0][0] = 9;`
`myArrays[0][1] = 2;`

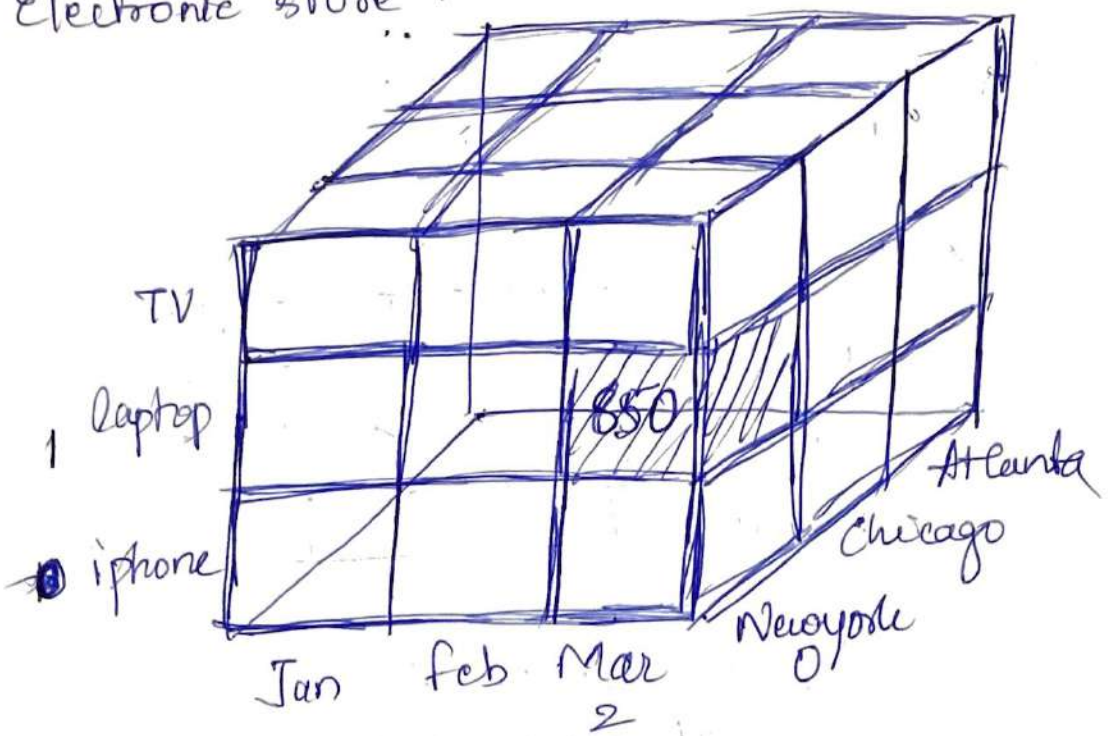
2) `int[][] myArray = new int[][]{{9,11},{2,5},{4,4},{6,13}};`

Array operations

* length
  → myArray.length → 4
  → myArray[0].length → 2
* `int[] row = myArray[2]`

3D Arrays

Electronic store ~ Sales Data



myArray[0][2][1] = 850

Initialization is same as 2D arrays.

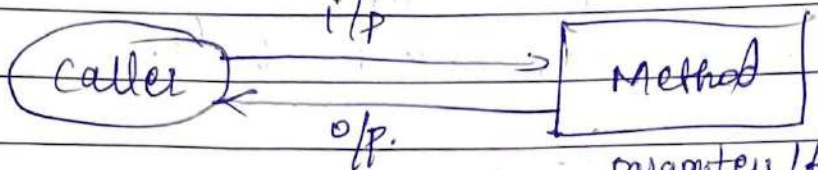
21/05/2022

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Methods (behaviours of a class) (Reusability)

* Self-contained logic that can be used many times.
* Can receive input & generate output



Syntax def

parameters / formal parameters.

```
returnType methodName(type param1, type param2, ...)
{ ...
  return someValue;
}
```

signature.

match

Syntax invoke

```
type var = methodName(arg1, arg2, ...)
```

arguments / actual parameters.

returntype

* Void → Nothing to return.
  → Optional return; as last statement
* must be primitive, array, class, interface (or) void
* Other than void → must return a value.

Method Types

There are 2 types → instance methods, static methods

1) Object-level methods - // behaviour of objs
   * Invocation : objectRef . methodName()
   * Affect object state
     → Instance variables
     → other instance methods.

2) static methods -
   * Keyword static in declaration
   * class level methods.
   * No access to state (instance variable / methods)
     → Serve as utility methods ex : sum(double x, double y)
     → Can access static variable.
     → Can directly access other static methods.
   * Invocation : className . methodName());
   * main method is static.

Passing data to methods

* Pass by Value
* Pass by Reference

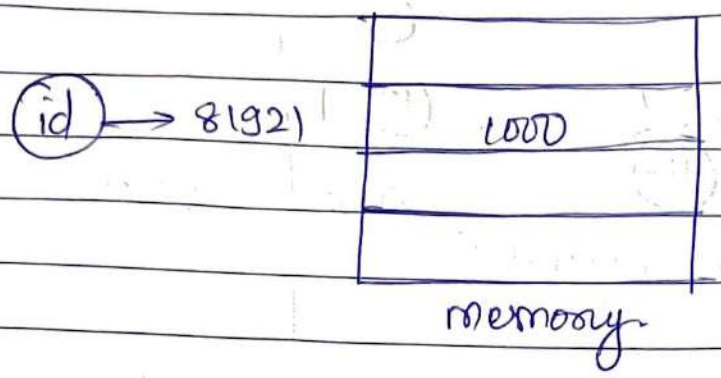
Primitives in Memory

int id = 1000;

id → <logical name, memory address, value>

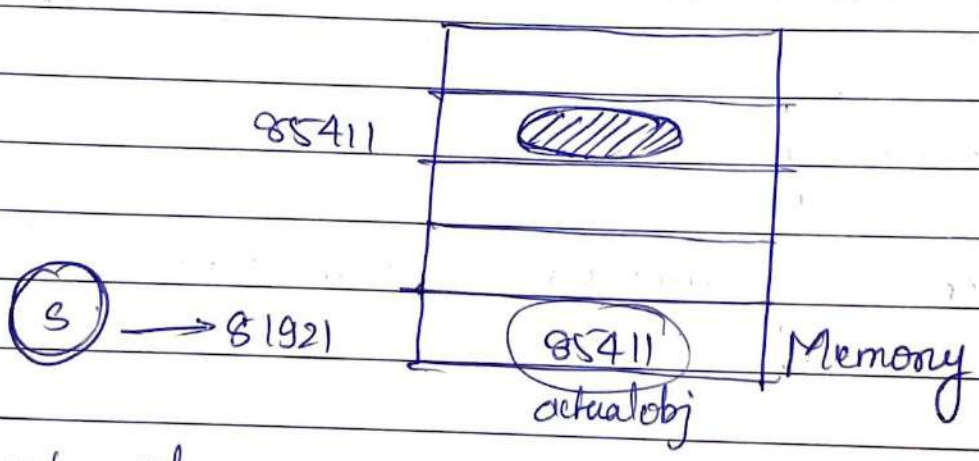
Name of the Experiment ..... Page No.:

Experiment No..... Date :



Object ref is memory

Student s = new Student();

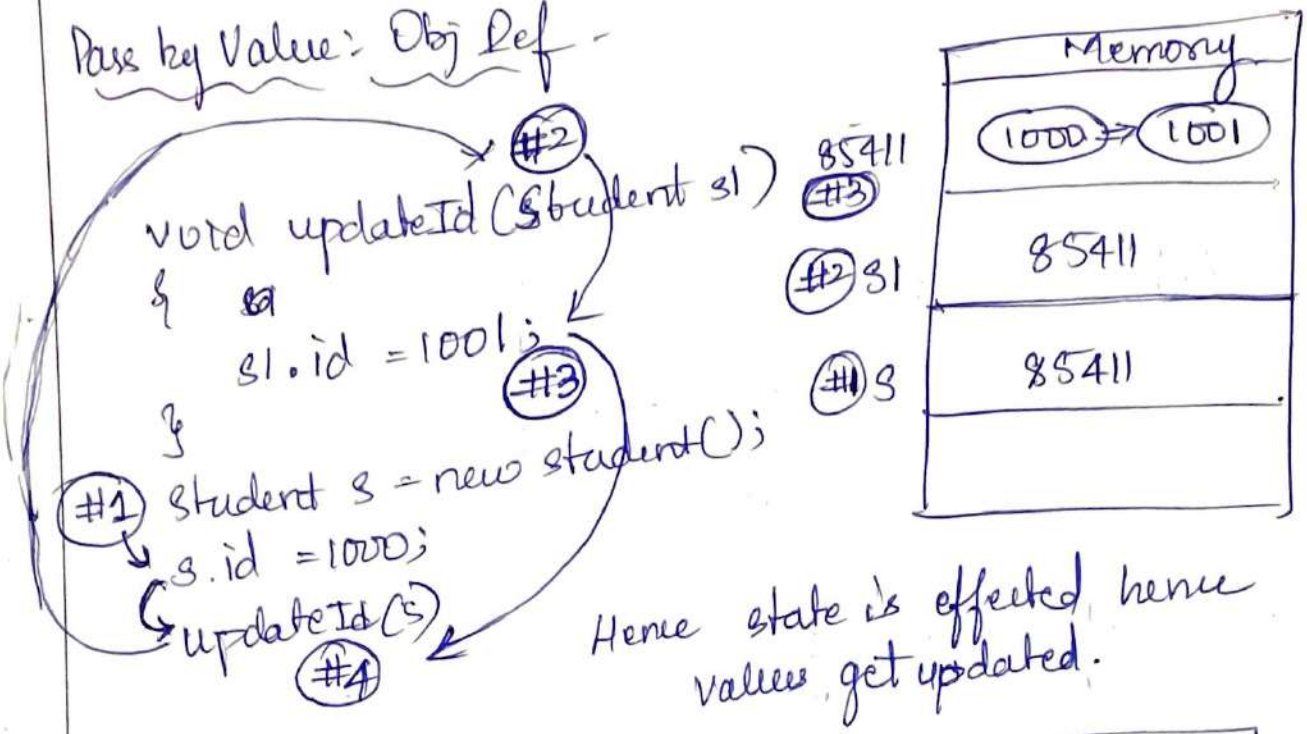


Pass by value

Arguments are passed to the method parameter.

* Primitive argument ~ value is primitive
* Obj ref argument ~ value is memory address.

Pass by Value: Obj Ref -



Hence state is effected hence values get updated.

JAVA IS ALWAYS PASS BY VALUE

Method Overloading

Methods having same names in the class but having different parameters.

* Must change parameter list.
  → #parameters or parameter types or both must vary
* changing only return type doesn't matter
* Applies to instance & static methods.

Ex :-

Valid
```
void updateProfile(int newId){}
void updateProfile(int newId, char gender){}
void updateProfile(char gender, int newId){}
void updateProfile(short newId){}
```

Name of the Experiment ..... Page No.:

Experiment No. ..... Date :

void updateProfile(int newId) {}

Ex:- boolean updateProfile(int newId) {}
void updateProfile(int id) {}
String void updateProfile(int newId) {}

duplicate

compiler error.

Method : Varargs (basically an array)

* Before Java 5 ~ fixed number of arguments.
* After intro of varargs ~ variable length arguments.
* Last parameter can take variable numbers of arguments
  → Can be the only parameter.

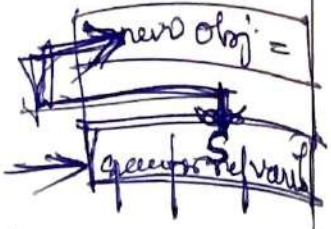
Syntax: Three dots following parameter type
→ foo(boolean flag, int... items)

Invocation: Array : foo(true, new int[]{1,2,3})
Comma-separated arguments : foo(true, 1, 2, 3)
Omitted : foo(true) ⇒ foo(true, null)
compiler does it

Restrictions ⇒
* Must be last parameter
  foo(int... items, boolean flag) ✗ error
* Only one varargs parameter can be used
* provides simpler & flexible invocation.

BIT

Varargs & Overload Methods



* Invalid overload example
  foo (boolean flag, int... items)
  foo (boolean flag, int[] items)
* Varargs method will be matched last.
* Demo : basics\BasicsDemo.varargsOverload()

CONSTRUCTOR

* By default constructor is created by compiler implicitly
* main goal – Initialize object state

Student s = new Student();

- Student s → Allocates space for reference variable.
- = → s ← Student object's address
- new Student(); → Allocate space for new student object
- Student() invokes constructor

Syntax : className(type param1, type param2, ...) {
  ...
}

{ Has No Return Type
  Can have Varargs.

Ex :-
```
class Student{
    int id;
    Student(int newId){
        id = newId;
    }
}
```

Student s = new Student(); X comp error
Student s = new Student(100);

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Constructor Overloading

* Same name with different parameters.
* Simplifies object creation statements.
* Object can be created from any of the constructors in a class

this() is used to invoke an overloaded constructor (no type)
to avoid duplication in code.

should be 1st statement

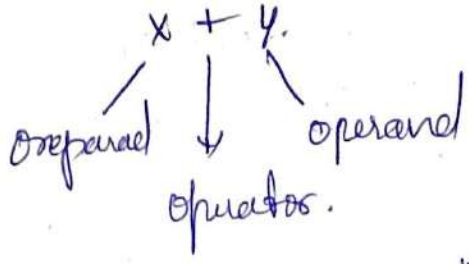
* Must be 1st statement inside constructor
* Only one - per - constructor
* No recursive invocation
* No instance variable as arguments.
* Cannot pass instance variable.

23/05/2022

Method building blocks :- Operators & control flow Statements

Operators

Performs operat° on its operands and produces a result



Types → Assignment
Arithmetic
Comparison
Logical
Bitwise
Bitshift
InstanceOf

Based on no of operands

* Unary.
  → pre-inc/dec Eg. -x;
  → post-in/dec Eg. x++;
* Binary
  → x+y
* Ternary
  → (?:) (T) (F)

Ex:- (x>3)?x:0

Arithmetic Operator

* Addition (+) Eg., 5+2 = int i
* Subtraction (-) Eg; int i = 5-2
* Multipl° (*)
* Div (/) ⇒ Quotient
* Modulus (%) ⇒ Remainder

~ Unary plus = +5;
~ String concatnat°

~ Unary minus; -x;

to check if the number is even (or) odd.

APPLIES ALL PRIMITIVES.

Name of the Experiment ..... Page No.: ☐

Experiment No..... Date : ☐☐☐☐☐☐☐☐

Shorthand Operators

* Pre & Post increment/decrement
  - → Applies to addition & subtraction
  - → ++ (or) --
  - → Increment/decrement by 1 ; eg:-, x++ [x = x+1]

* Compound Arithmetic Assignment operators
  - → Applies to all arithmetic operators
  - → +=, -=, *=, /=, %=
  - → x += 5 ; [x = x+5]

Example

Post x++; // 6.

Pre ++x; // 6

int y = x++ ; // y = 5, x = 6

↓

int y = x;

x = x+1;

int y = ++x; // y = 6, x = 6

↓

x = x+1;

int y = x;

Arithmetic operators rule.

Operator precedence.

- → Rule 1 ⇒ *, /, % higher precedence
  - ⇒ +, -
- → Rule 2 ⇒ Operators in same group are evaluated left to right

(15+9) - 3) + (2*5)

→

BIT

Use parenthese to change evaluation order

((5+9) - (3+2) * 5)

Operand promotion

Operands smaller than int are promoted to int.

Same Type Operators

If both operands are int / long / float / double, then the operation is carried in that type and evaluated to a value of that type.

5 + 6 → 11

1/2 → 0, not 0.5

Mixed type operators

If operands belong to different types, then smaller type is promoted to larger type

Order of promotion : int → long → float → double.

1/2.0 or 1.0/2 → 1.0/2.0 → 0.5

char + float → int + float → float + float → float

9/5 * 20.1 → (9/5) * 20.1 → 1 * 20.1 → 1.0 * 20.1

→ 20.1

"Type of final result will be of largest data type"

Name of the Experiment ..... Page No.: ☐

Experiment No..... Date : ☐☐☐☐☐☐

Comparision Operators

!= Not equal to, <, >, >=, <=

== ⇒ equal to. !=, etc.

Logical Operators

Logical AND = &&

Logical OR = ||

Logical Not = !

Truth table

x	1	1	0	0
y	1	0	1	0
x && y	1	0	0	0
x || y	1	1	1	0
!x	0	0	1	1

Short Circuit Operators ~ &&, ||

&&

if left operand is "false", return "false".

- Conditional AND.

||

left operand is "true", return true.

- Conditional OR.

&& - prevents NullPointerException.

Precedence

1) !

2) &&

3) ||

BIT

Grouping & evaluation

1) a) Left-associativity ~ order of grouping
   b) Associativity
   $(a \&\& b) \&\& c = a \&\& (b \&\& c)$
   [Applies for both && and ||]

2) a) Operator precedence of logical operator:-
   Helps with only grouping operations.
   Not order of execution. (! > && > ||)

Ex:- $A \&\& B \| C \&\& D = (A \&\& B) \| (C \&\& D)$
$A \&\& B \&\& C \| D = ((A \&\& B) \&\& C) \| D)$

3) Operator Precedence across logical, comparison and arithmetic operators.

! > arithmetic > comparison > &&, ||

Bitwise Operators

1) Operate on individual bits of operands
2) Operands:
   Integer primitives ~ operands promot° rule applies
   Boolean (rare)

→ Used mainly in embedded systems
→ Used in Hash tables.
→ Compression & encryption

Name of the Experiment ..... Page No.: ☐

Experiment No..... Date : ☐☐/☐☐/☐☐☐☐

Operators.

Bitwise AND (&)

→ Return 1 if both input bits are 1.

→ let x = 1, y = 3.

(x & y) → 1

```
00000000 00000000 00000000 00000001 (x)
00000000 00000000 00000000 00000011 (y)
00000000 00000000 00000000 00000001
```

Bitwise OR (|)

→ Return 1 if one of the bits is 1.

→ let x = 1, y = 3

(x | y) → 3.

```
00000000 00000000 00000000 00000001 (x)
00000000 00000000 00000000 00000011 (y)
00000000 00000000 00000000 00000011
```

Bitwise XOR (^)

→ Return 1 if only if one of the input bits is 1 but not both

(x ^ y) → 2

(x ^ y)

```
00000000 00000000 00000000 00000010 (2)
```

BIT

Bitwise NOT (~)

→ Inverts bits
→ Let x = 1
→ ~x → -2
→ ~(00000000 00000000 00000000 00000001) =
11111111 11111111 11111111 11111110

Non short circuit Operators

→ & and |
→ Always checks both operands.

Compound Bitwise Assignment

→ operand1 = operand1 & operand2
operand1 &= operand2

Ex :- boolean b = true
b &= false; // Assigns false

→ Operand1 |= operand2
→ Operand1 ^= operand2

Name of the Experiment ..... Page No.:

Experiment No. ..... Date :

Bit Shift Operators

→ shifts bits.
→ Operands ~ integer primitives

→ << (left shift)
→ >> (signed right shift)
→ >>> (Unsigned right shift).

<< (left shift (LSB → zeros)

→ left shifts left operand by # bits specified on the right.
→ Ex:- (6)

6 → 00000000 00000000 00000000 00000110

(12) ← 6<<1 → 00000000 00000000 00000000 00001100

→ Inserts zeros at lower-order bits.
→ Same as multiplicat° by powers of 2

6<<1 → 6 * 2^1 → 12
6<<3 → 6 x 2^3 → 48

8 x 2^3 =

BIT

Unsigned Right Shift (>>>)

→ Right shifts left operand by # bits specified on right

→ Inserts zeros at MSB.

Ex:-

(12) → 00000000 00000000 00000000 00001100

(6) ← 12>>>1 → 00000000 00000000 00000000 00000110

→ Same as div by powers of 2

12 >>> 1 ⇒ 12/2¹ → 6.

Signed RS operator

→ Same as >>>, but preserved with MSB for sign indication (sign bit).

Compound bit shift Assignment

→ Operand 1 = operand 1 << operand 2
operand 1 <<= operand 2

→ Operand 1 = operand 1 >>> operand 2
operand 1 >>>= operand 2

→ Operand 1 = operand 1 >> operand 2
operand 1 >>= operand 2

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Control - flow Statements

if-else

```
if(cond)
{
   // code
}
else if(cond)
{
   // code
}
else
{
}
```

last

nested if

```
if(cond)
{
   if(cond)
   {
      if(cond)
      { // code
      }
      else{}
   }
   else{}
}
```

Switch statement

→ Can be used as an alternative to if-statement

Ex:-

```
int month = 3;
switch(month){
case1 : SOP("Jan");
        break;
Case2 : SOP("Feb");
        break;
Case3 : SOP("March");
        break;
default : SOP("NA");
}
```

BIT

Switch exp type

→ Integer, e.g. : 7, x, x+y
Cannot be long and above
→ String (since Java 7)
→ enum

other than these 3, the compiler throws error.

Case Lable Restrictions

→ Must be within range of datatype of switch expression.
→ Constant expression : value known at compile time.
→ Value must be unique
→ Cannot be null.

Range Ex :-

byte month = 3;
Switch(month) {
Case 1 : SOP ("Jan");
break;
Case 128 : SOP ("Feb");
break;

-127 to 126.
Error

Where can't we used Switch?
→ Cannot be used where there are more than one condition to test.
→ Tests other than equality eg., month >=3

Name of the Experiment ..... Page No.:

Experiment No..... Date :

When

→ Switch expression is not integer, string or enum. Cannot be used.

Where is switch preefered?

→ Readability

→ Intent

⇒ Switch deliberatly states that only one variable is involved.

→ Speed.

→ Faster than if becauses conditions & cases are known during compile time itself.

Ternary Statement (Alternative to if else)

→ Shorthand for if-else with single statement

Syntax → result = (boolean-exp) ? true-exp : false-exp ;

Ex min = (x < y) ? x : y ;

→ Improves code reability.

Ex Smart string construction.

greeting = "Hello " + (user.isMale() ? "Mr." : "Ms.") + user.name();

≠ Cannot be an expression statement / compiler error.

BIT

For statement (optional)

```
for (initializat° ; condit° / expression ; expression-list);
{
    //code
}
```

Initializat° - declarat° statement

```
for (int i = 0; ; )
for (int i = 0; j = 1; ; )
for (int i = 0, int j = 1; ; ) // invalid
for (int i = 1, double d = 1.0; ; ) // invalid
```

Initializat° - Expression statements

```
for (i = 1, j = 2; ; )
for (i = 1, double d = 10.0; ; ) // invalid
for (i++; ; )
for (SOP(i); ; )
for (SOP(i), i++; ; )
```

Initializat° Condition Expression

Must be boolean.

Optional

→ if omitted, a true is assumed i.e., infinite loop

```
for (int i = 0; ; )i++)
    SOP(i);
}
```

Name of the Experiment ..... Page No.:

Experiment No..... Date :

enhanced for Loop

actual for loop

```
iArray = new int[]{0,1,2,3,4,5,6,7,8,9};
for(int i=0;i<iArray.length;i++)
{
    SOP(iArray[i]+" ");
}
```

↙ same as this

enhanced for loop

```
for(int i : iArray) ⟹ directly access array elements.
{
    SOP(i+" ");
}
```

Variable Scope

→ Class level variable
* Entire class
* Cannot be assigned to variables declared before it

→ Local variable
* Within methods/constructor/ with Control flow Stm

→ Shadowing class level Variable

BIT

while statement

```
while (boolean cond?)
{
    // code.
}
```

```
while
{
  // code
} do(cond?)
```

Break statement

Exits immediately enclosing switch / loop

Ex:-
```
for() {
    break;
}
exits for.
```

```
for() {
  if() {
    break;
  }
}
exists for.
```

```
for() {
  for() {
    break;
  }
}
exits inner for.
```

```
for() {
  switch() {
    case : break;
  }
}
exits switch.
```

Cannot be used in if statement directly

```
if() {
    break;   ✗ invalid
}
```

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Labelled break statement

∴ Ex:- Label : block statement
if() { // block1
} else { // block2
}.

break label;

x: if() {
break x; // works fine.
}

Continue Statement

→ Used with only loops.
→ Continues with next iteration of innermost loop.

Ex:- while (cond^n - exp) {
if (cond^n) {
continue; // goes to next iteration
}

}

(or) for

Cannot be used in if
and labeled continue statements can be achieved but only for "for" (or) "while" only.

not if

BIT

Recursion

The method that invokes itself is called as recursion technique.

Factorial (But slower)

⇒

```
static int factorial(int n){
if (n==0){
    return 1;
}
return n* factorial (n-1);
}.

public static void main (String[] args)
{
    SOP (factorial(4));
}.
```

Where can we use recursion?

→ Problem addressed via similar sub-problems

Eg:- Binary search, Towers of Hanoi, Word frequency count

```
int a[] = {4, 19, 24, 34, 55, 65, 71, 83, 91};
binarySearch (a, 0, 8, 65);
```

24/05/2022

Name of the Experiment ..... Page No.:

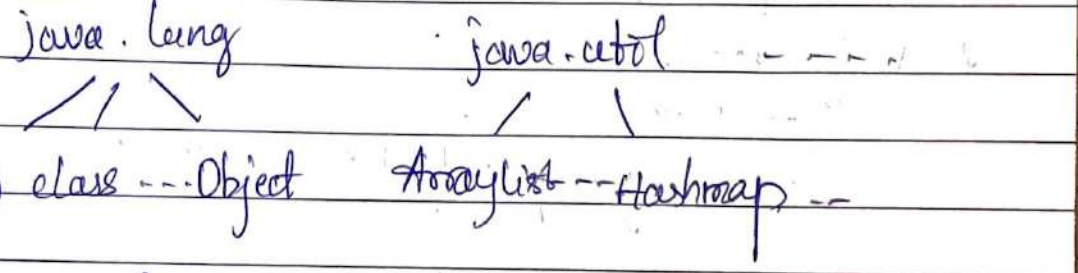
Experiment No..... Date :

Package & Information hiding

Java API

* Library of well-tested classes
* Java 8 ~ 4240 classes
* Developed by experts
* Used by millions of programmers
* Part of both JDK and JRE

Packages



Why?

* Meaningful organisation.
* Name scoping ⇒ java.util.Date ≠ java.sql.Date
* Help in Security.

Important Packages

* java.lang ~ Fundamental classes
* java.util ~ Data structures
* java.io ~ Reading & writing
* java.net ~ Networking
* java.sql ~ Databases.

BIT

3rd Party API

Big Data (Apache Hadoop)
Data Mining (Weka, Apache Mahout)
Database (Hibernate)
Search (Apache Solr)
Parsing (JDM, Jackson, Google gson)
Core (Apache commons, Google Guava)
Web framework (Spring).

Accessing Packages

Accessing classes

* Same package → direct access
* Diff pack
  → import
  → fully-qualified class name.

Ex:- import java.util.ArrayList.

* Import single class
  → Explicit import (or Single-Type-Import)
* Import multiple classes
  → Seperate explicit import
  → import (or Import-On-Demand)

* import

Eg. import java.util.* → imports all class

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Alternative to import (Fully Qualified class Name)

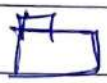
java.util.ArrayList list = new java.util.ArrayList();

java.lang → imported by default

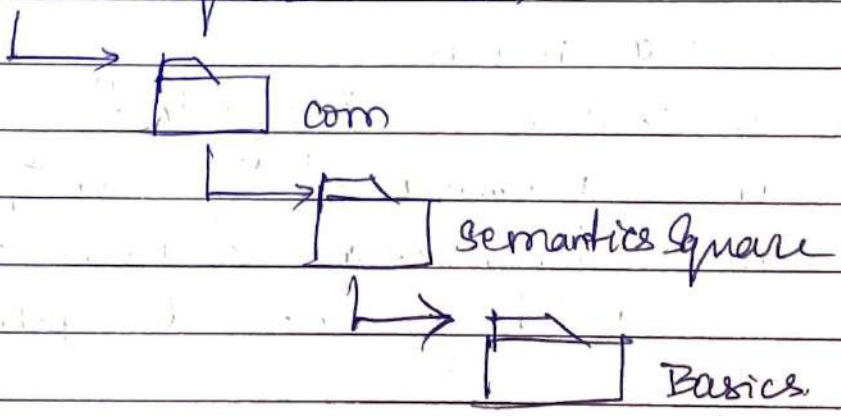
Creating Packages

Set-up Matching directory structure

* basics



* com.semanticssquare.basics;



Package statement

* package package-name;

  package com.semanticssquare.basics;

* Must be first statement above any import.

Package Naming

(Item 56) - All rules related to naming

* lowercase alphabets, rarely digits.
* Short ~ generally, less than 8 characters.
* Meaningful abbreviations, eg. util for utilities
* Acronyms are fine., e.g., awt for Abstract Windowtoolkit
* Generally, should be single word.
* never start with java or javax

Access levels

Accessibility for classes

* Inside package
* Inside & outside package - "public" access modifier

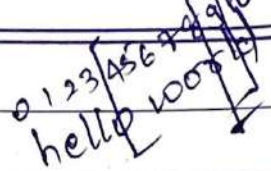
for class members.

* Inside class. ~ "private"
* Inside package only "package-private" default.
* Inside package only + any subclass. - "protected"
* Inside & out package - "public".

Name of the Experiment ..... Page No.:

Experiment No..... Date :

String



* Object of Class java.lang.String

```
String s = new String(); // empty string
String s = new String("hello!");
//(Accepts String Objects)

char[] cArray = {'h', 'e', 'l', 'l', 'o', '!'};
String s = new String(cArray);

String s = "hello!"; // String literal.
```

* String class internally uses character array to store text.
* Java uses UTF-16 for characters
* String is a sequence of unicode characters
* String is immutable

Str Obj = immutable sequence of unicode characters.

Speciality
* String literal
* + Operator ⇒ Concatination.
* String pool ∞ saves memory

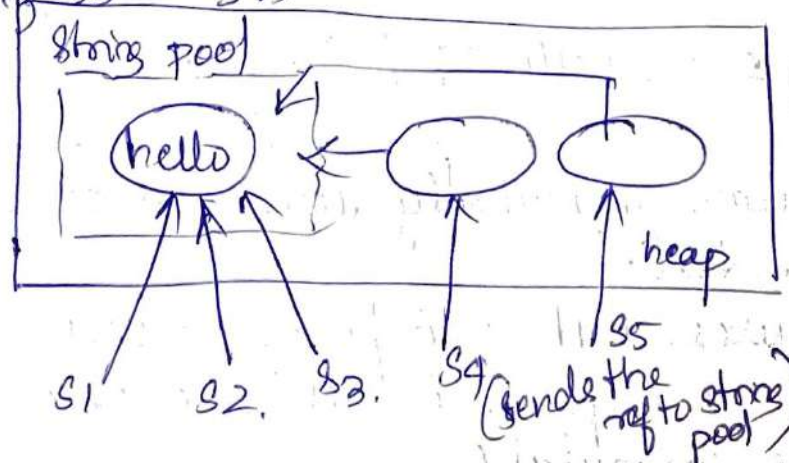
String class – Common operations

* Comparing
* Searching
* Examining individual characters
* Extracting substrings
* Case translation
* Replace
* Split

String Pool { Interning } ⇒ (String literal) (vs) (String new).

```
String s1 = "hello";
String s2 = "hello";
String s3 = s1;
```

} can appear anywhere like in diff packages / class / etc.




```
String s4 = new String("hello");
String s5 = new String("hello");
```

s1 == s2 ? (True) | s4 == s5 ? (false)

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Backend by JVM

* Create new String object with given literal
* Invoke intern()
*

If (string in string pool)
    return existing reference
else
    add to string pool & return reference

String interning

* The result of concatinat° is also pooled.
    ⇒ interened.

String Immutability

Values can never be changed.

String s1 = new String ("abcd");
s1 = new String ("1234"); // above obj is abandoned.

String Concatinat° (+ operator)

* String s = "hello" + "World!";
  String s = "hello" + "world!" + "125"; "hello world! 125"
String s = "hello" + "world!" + 125;
String s = "hello" + "world!" + 125 + 25.5 ~ "helloworld!12525.5
String s = "hello" + "world!" + 125 + 25.5 + = "150.5hello World!"

BIT

Concatinat can also be done by

* String Builder // Mutable
* String Buffer -

54. - StringBuilder sb = new StringBuilder();

```
sb.append("hello");
sb.append("world!");
String s = sb.append("Good").append("morning")
           .toString();
```

returns ref to same obj

* Other methods are : length, delete, insert, reverse, replace.
* Not synchronised.

String Buffer

* Synchronised ~ Slow.
* Obsolete - same as String Builder
* API compatible with String Builder -

Escape Sequence

* Character preceded by \
* To use special characters in strings & character literals.

Name of the Experiment ..... Page No.: ☐

Experiment No. ..... Date : ☐☐☐☐☐☐☐☐

Escape sequences

- \" ~ double quote
- \' ~ single quote
- \n ~ new line
- \t ~ tab
- \\ ~ backslash.
- \r ~ carriage return
- \b ~ back space
- \f ~ formfeed.

} only valid backslash

Static Initializing

* Initialization needs multiple lines
  → Populating a data structure
  → Initialization with error handling

Ex: `static HashMap map = new HashMap();`

```
static {
    map.put("John", "111-222-3333");
    map.put("Anita", "222-333-4444");
}
```

Ex: `static Stuff staff`

```
static {
    try {
        staff = getStuff();
    } catch(Exception e) { ... }
}
```

BIT

final variable

* Cannot be changed.
* Implies constant
  - → primitive - value is const
  - → Ref variable - ref is const, not obj content.
* Do not get a default value.
* Used with instance, local / static.
* Must be initialised in → Declat°
  - → Constructor
  - → Instance initializer.

* Naming convention —
  - → All Caps with underscore seperating words
  - → Private static final int COPY_THRESHOLD = 10;

Constant variable.

public static final double PI = 3.14159265358979323846

* Compile time constants.
* Computer optimizat°.

int x = Math.PI → int x = 3.14159265358979323846;

Stored in .class file.

* It should be declared as a final.
* primitive (or) string type
* Initialised in declaration statement
* Initialized with compile-time const expression.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

```
final int x = 23;
final String x = "hello";
final int x = 23+5;
final String x = "hello" + "world!";

final int z=5;
final int x = 23+z; // z is hard-wired.
```

Invalid

```
int z = 5;
final int x = 23+z ; // Not a const variable.

public class Test {
    final int x;
    public Test(){
        x = 23;          // Not a const var as the decret not done at first.
    }
}
```

Boxed Primitives.

int ~ Integer	boolean ~ Boolean
long ~ Long	char ~ Character.
byte ~ Byte	
short ~ Short	
float ~ Float	
double ~ Double.	

BIT

Ex :- Integer boxed = new Integer(25); // boxing

Auto boxing

Integer boxed = 25;

Now the computer does the auto boxing part

→ Integer boxed = new Integer(25);

Auto unboxing:

int j = boxed;

int j = boxed.intValue();

→ No autoboxing for Arrays.

Class Organisation

1) Variables ~ static followed by instance
2) Static initializers
3) Static nested classes
4) Static methods.
5) Instance initializers
6) Constructors
7) Instance nested classes
8) Methods

Name of the Experiment ..... Page No.: ☐

Experiment No..... Date : ☐☐☐☐☐☐☐☐

Class Size

* The Single Responsibility principle
* Helps create better abstractions
* Helps in having fewer lines of code.
* Helps make loosely coupled classes

* less than 2000 lines.

Methods

small & focused.

* should do only one thing.

Group methods with similar functionality.

Local variables

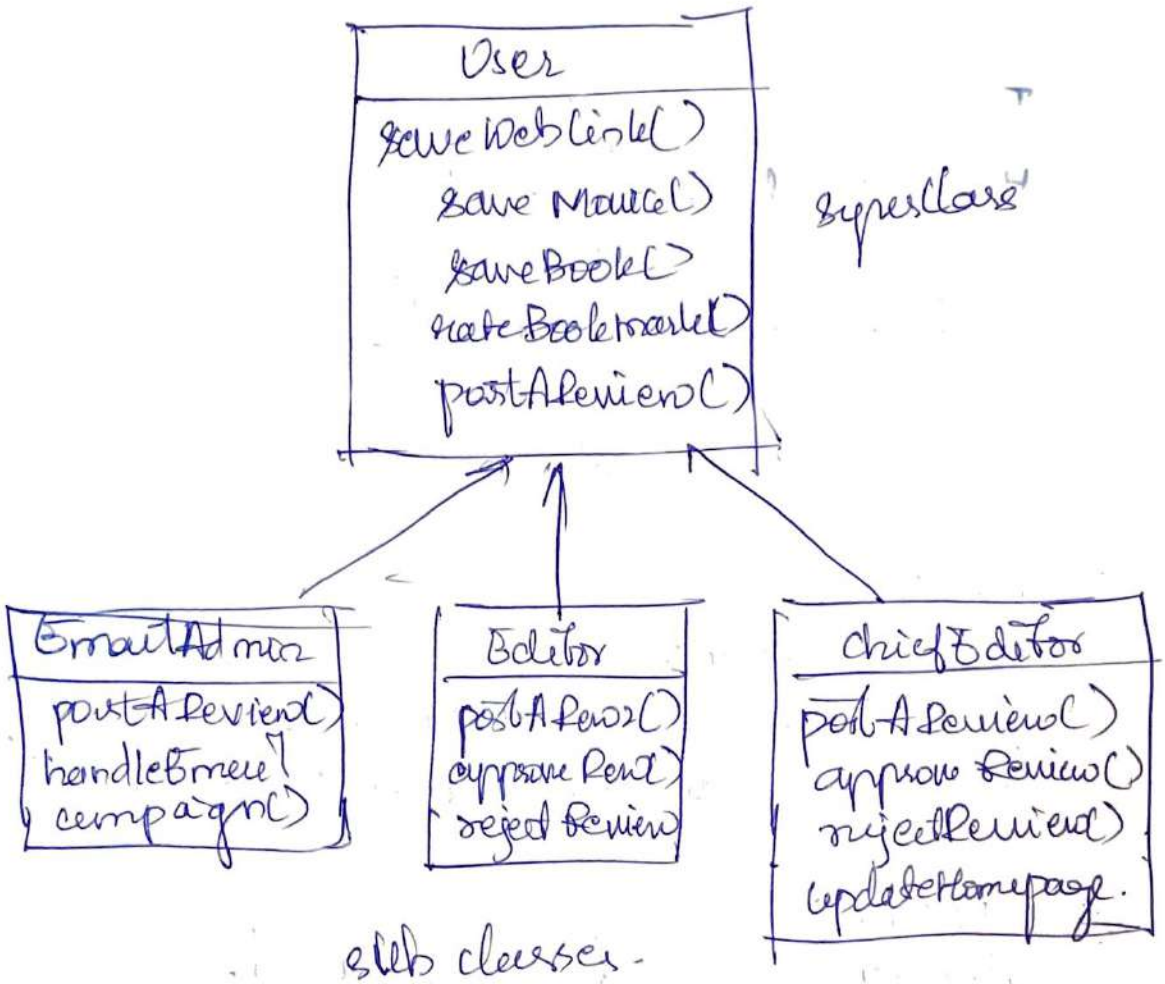
declare where it is first used.

Prefer "for" over "while" loops

25/05/2022

Inheritance

* Fundamental feature of OOP.



subclasses inherit features from superclasses.

Subclasses

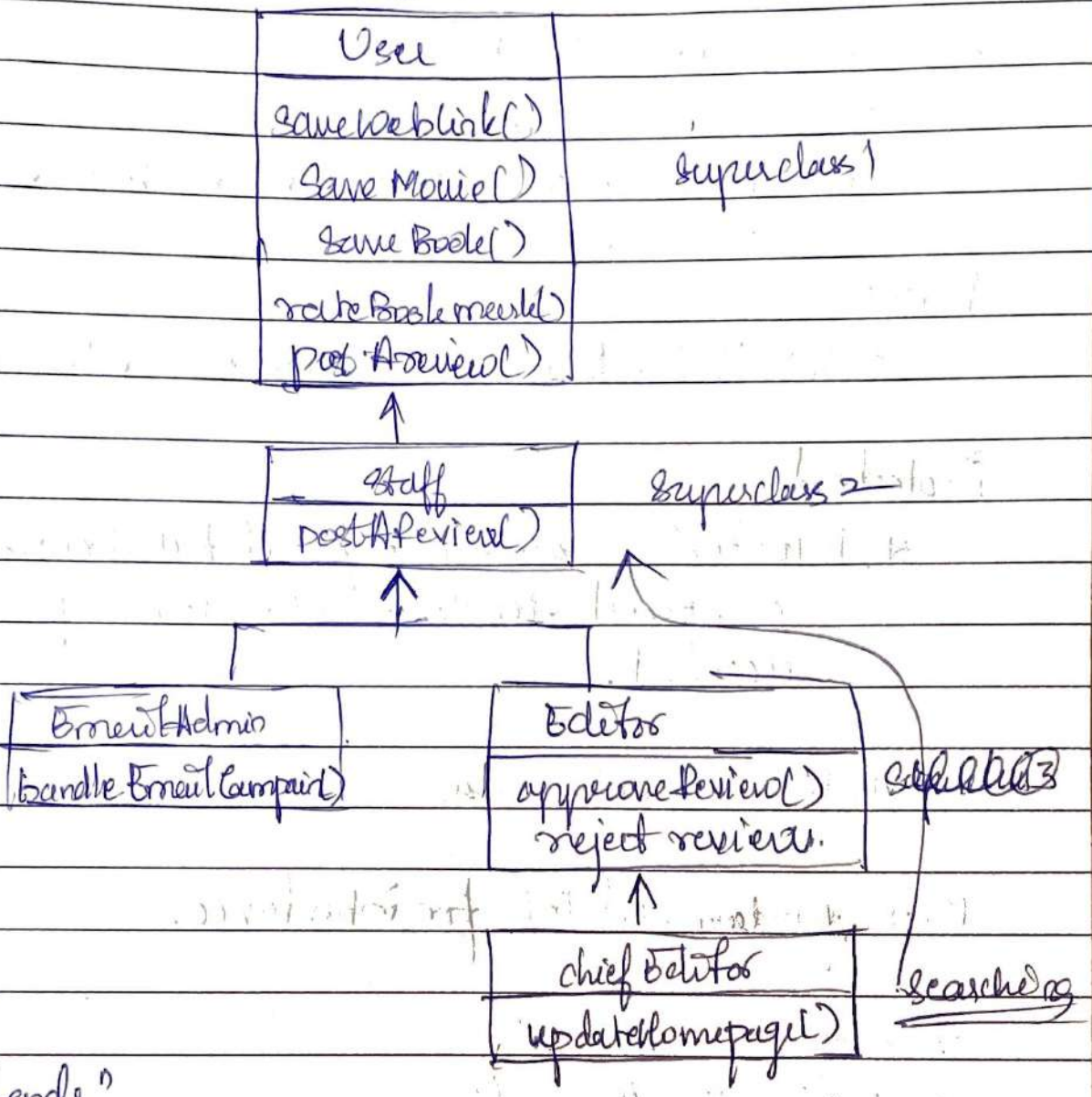
* Specialized versions of super classes.
  - → Inherit members
  - → Add new members
  - → Override superclass methods

subclass ⟵ Superclass + subclass capabilities.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Superclass → supertype (or) base class.
subclass → subtype (or) derived class.



"extends"

Ex:- class User{}
class Staff extends User{}
class EmailAdmin extends Staff{}
class Editor extends Staff{}
class ChiefEditor extends Editor{}

A class can extend from only one class.

BIT

Inheritance Accessibility.

Private.
* Not inherited

Default (only in package)
* Inherited if from family
* Only family can access inherited members.

Public
* Inherited - Anyone can access inherited members.

Protected
* Inherited - same as default but 1 extra feature is a special subclasses outside the package can be accessed.

IS-A Test for inheritance

Most fundamental test for inheritance.

* Staff IS-A User
* Editor IS-A Staff
* Editor IS-A User
* Chief Editor IS-A User
* Surgeon IS-A Doctor
* Triangle IS-A Shape.

Name of the Experiment ..... Page No.: ☐

Experiment No..... Date : ☐☐☐☐☐☐☐☐

HAS-A Test

* Bookmark IS-A Review ~ Nope
* Review IS-A Bookmark ~ Nope.
* Bookmark HAS-A Review ~ Yes. // composition.
* Bathroom IS-A Tub ~ Nope.
* Tub IS-A Bathroom ~ Nope
* Bathroom HAS-A Tub ~ Yes.

Polymorphism

Defining contract

Classes defines contract

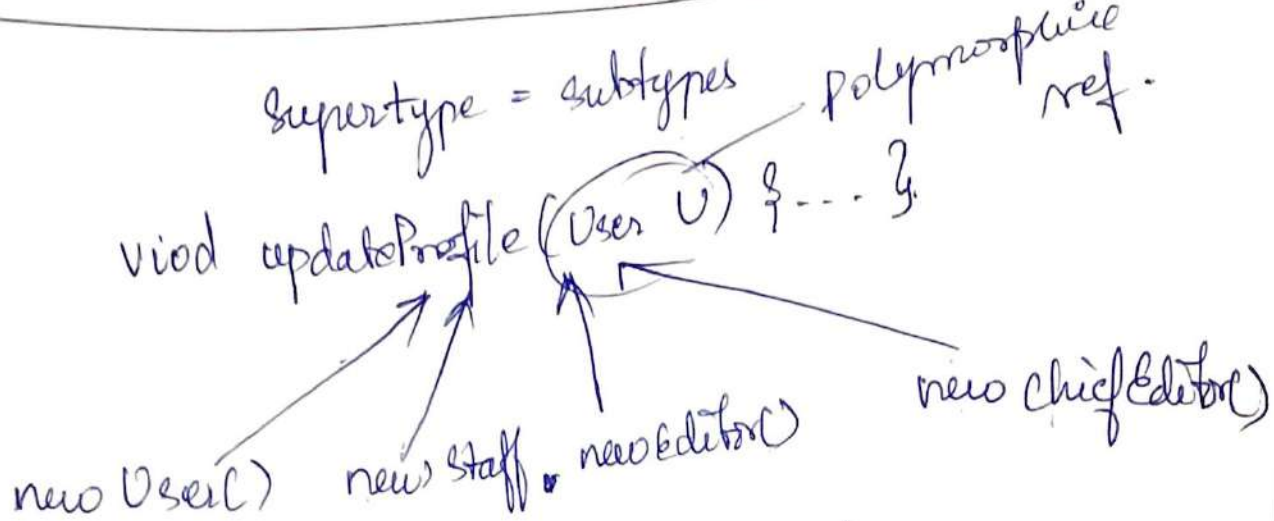
↓

"I have these kinds of methods".

Supertype defines common protocol

↓

"Myself & my subtypes have these kinds of methods..."



* flexible code.
* clean code.

Here "User u" can take any form i.e. parameter this is called polymorphism.

Reference type & actual object type can be different.

User user = new Editor()

↓ Ref type

↓ Object type

Method invocation

* Compiler uses ref type to decide whether a method can be invoked.

```
User user = new Editor();
user.approveReview(); // compile error
```

* JVM uses object type to decide which method is invoked.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Casting of objects & instanceof Operator

Primitives Casting.

Implicit : float f = 25

Explicit : int x = (int) 42L;

Implicit Casting

void updateProfile (User u) {...}.

new User() new Staff() new Editor() new chiffEdi()

Explicit casting

Subclasses - specific methods are no longer valid.

```
Staff s = new Editor();
s.approveReview(); // compiler error.
((Editor) s).approveReview();
```

i.e.,

```
void approveReview (Staff s) {
    s.approveReview(); // compiler error.
}
void approveReview (Staff s) {
    ((editor)s).approveReview(); // runtime error.
}
approveReview (new Staff());
```

BIT

Instance of Operator.

objRef instanceof aClass.

* Objet's type is what matters.
* Instance of subclass is also an instance of superclass.

```
User u = new User()
Staff s = new Staff();
User us = new Staff();
```

u instance of User → true
u instance of Staff → false
s instance of Staff → True.
s instance of User → True.
us instance of User → True
us instance of Staff → True.

Exp casting,

```
void approveReview (Staff s){
    if(s.instanceof Editor){
        ((Editor).s).approveReview();
    }
}.
approveReview (new Staff());
approveReview (new Editor());
approveReview (new ChiefEditor());
```

Name of the Experiment ..... Page No.: ☐

Experiment No..... Date : ☐☐☐☐☐☐☐☐

Type safety

* Prevents type errors. – (undesirable program Error)
  → invoking non-existent method on an obj.
  staff.approveReview()

* Buffer overflow ≈ out-of-bound writes.
  char data[6] = "hello"; [h|e|l|l|o|\0|A]
  data[6] = 'p' = [h|e|l|l|o|\0|p]
  data corrupted undesirable.

enforcement check
→ Compile time (Static type checking) ≈ C, Pascal
→ Runtime (dynamic-type checking) ≈ Ruby, Python
→ Both ✓ (in Java).

Static-type checking

* int i = 233.3;
  Sol^n ⇒ int i = (int) 233.3;

* Staff s = new Editor();
  Sol^n ⇒ s.approveReview();
  Sol^n ⇒ ((Editor) s).approveReview();

* Generics
    ⇒ detects at compile time itself.

Dynamic type checking

ClassCastException

```
void approveReview(Staff s){
    ((Editor)s).approveReview(); //Runtime exception
}
approveReview(new Staff());
```

Method Overloading

* Redefine behaviour of superclass method.
    → Add new behaviour.
    → Extend behaviour
    → Providing better implementation eg. override bad code.

Supertype define contract & common protocol

↓ method overriding

agreeing to fulfill the contract.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Rules

Rule 1 = Same parameters + compatible return types.
Rule 2 = Can't be less accessible.

Rule 1

* Return type must be same (or) subclass type.

Incompatible return type → compiles error

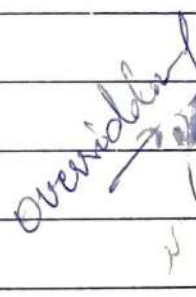
Rule 2

* Can't be less accessible.
* Access level must be same or friendlier.

Super keyword

Access superclass method by the subclass.
* If it is not overridden ~ direct or super
* If it is overridden ~ super.
* Can't use in static method * Can Access hidden field.

Ex:- class Staff extends User {
void passAReview() {
super.passAReview();
// behaviour extension code
}
}



*

Method Binding

Binding a method call to -
method declaration and its implementation

↓ method signature binding

↓ method implementation binding.

*Method signature binding

* compiler checks if reference type has compatible method
* Yes * Writes method signature details into bytecode.
* Applies to both instance and static methods

implementation

* If it is static method - By compiler based on ref type
* If it is Instance method - By JVM based on obj type.

Static methods → compile-time (or) early binding
& fields, Instance variable.

Instance methods → runtime (or) late binding.

What cannot be overridden

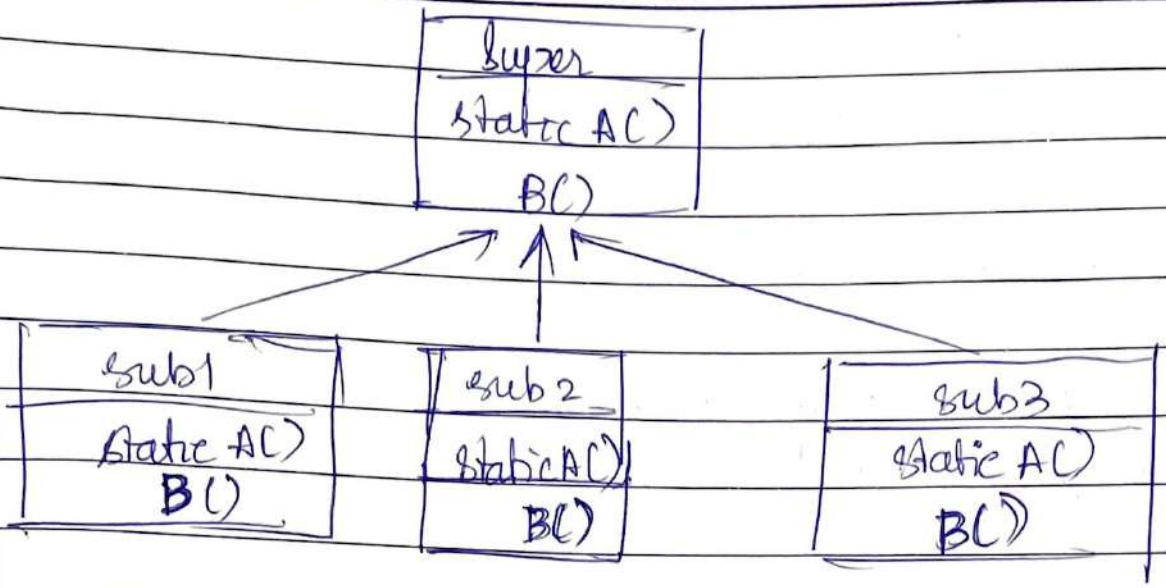
* final methods cannot be overridden

```
final returnType methodName() {}
```

* fields. ~ Instance & static
* Static methods.

Name of the Experiment ..... Page No.:

Experiment No..... Date :




```
void foo (Super s) {
    s.A(); // Resolved at compile time based on ref type
    s.B(); // Resolved at runtime based on object type.
}
```

Static methods & overriding

* Can't use super keyword - Qualify with superclass name
* Can't hide instance method. - Static means no state.
* Can't be overriden by instance method

Object class

java.lang.Object.

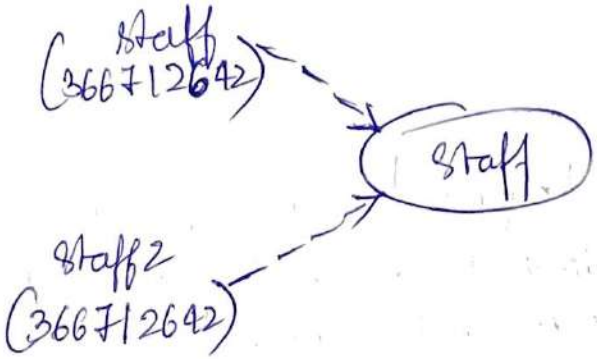
⇒ Mother of all classes, i.e., superclass of everything

Main purpose of Obj class

* Act as polymorphic type
* Includes core methods.

Core methods

* toString() ⇒ Returns String rep of obj.
* hashCode() ⇒ Returns obj's hashcode (memory address in hexadecimal)



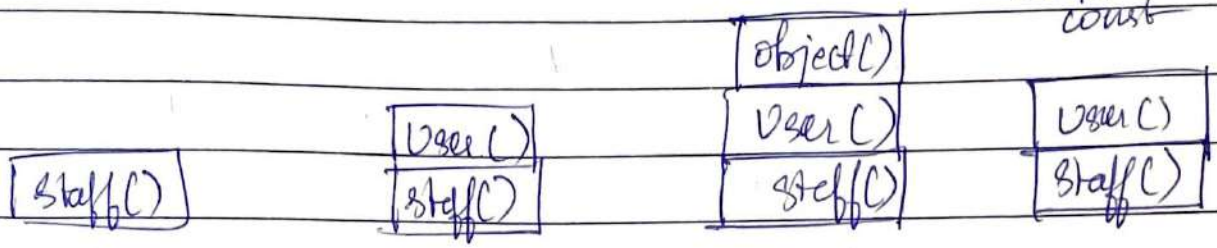
* equals(Object) ⇒ Tests object equality, uses == operator
* getclass() ⇒ returns class object.
  class obj → class name, superclass name, method name.
* clone() ⇒ protected method that returns a copy of this object

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Constructors Chaining

1 new Staff() 2. Staff invokes User's const 3. User invokes Obj's const 4. Obj completes Execn goes back to user const



* super keyword can be used to invoke superclass const.
* Superclass statement must be 1st or else this() is used.
* this() or super() but never both.
* with overloaded const, last invoked will calles superclass const
* If super() is not provided, compiler inserts super().

Preventing Inheritance

* Final class
* private Constructor.

Abstract classes & methods

* Signifies abstractness
* Non-instantiable, but defines common protocol for all subclasses..

Abstract class

```
abstact class Bookmarke{
    ...
}                    // Abstract classes should be extended.
```

Abstract method

```
abstract void iskidfriendlyEligible();
```

* Must be overridden!
* Cannot be dec as static.
* ~~Abstract~~

Specifics

* Abstract method → Abstract class.
* Abstract class → {abstract, concrete} methods.
    * Need not have abstract methods ∵ not typical.

Abstract subclass

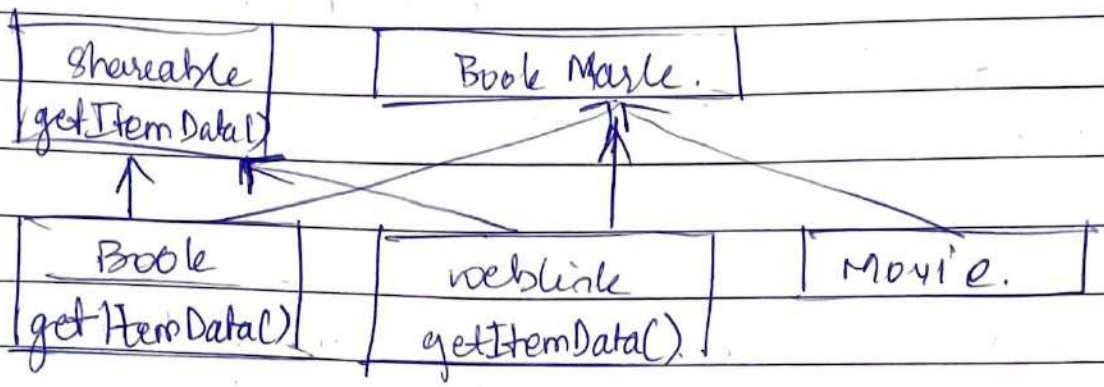
* Need not override abstract methods.
* Can override methods.
* Can define abstract & concrete methods.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

* Concrete subclasses must override unimplemented/ abstract methods.

Multiple Inheritance



// Java doesn't support multiple inheritance

Interfaces

Supertype defines common protocol

↓ contract

"Myself & any subtypes have these kinds of methods..."

Supertype defines contract.

class → public / protected methods

abstract class → public / protected (abstract) methods } state

Interface → mostly only public abstract methods.

(No state) (Non-instantiable) → pure contract

Syntax :-

```
public interface InterfaceName {
    static final fields.
    abstract methods // main.
    default methods
    static methods
    Nested types.
}
```

```
public interface Shareable {
    String getItemData();
}
```

* public & abstract by default
* Variables are public, static & final by default
* All members are public by default.
* Members can't be private & protected.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Implementing Interface.

```
public class Book extends Bookmark implements Shareable{
    public String getItemDetail(){...}
}.

public class Weblink extends Bookmark implements Shareable{
    public String getItemDetail(){...}
}.
```

Interface is Reference type

```
Shareable obj = new Book();
Bookmark obj = new Book();
Shareable obj = new Movie();

Shareable obj = new Shareable();
```
(And Never an Object type)

→ Interface can extend interface

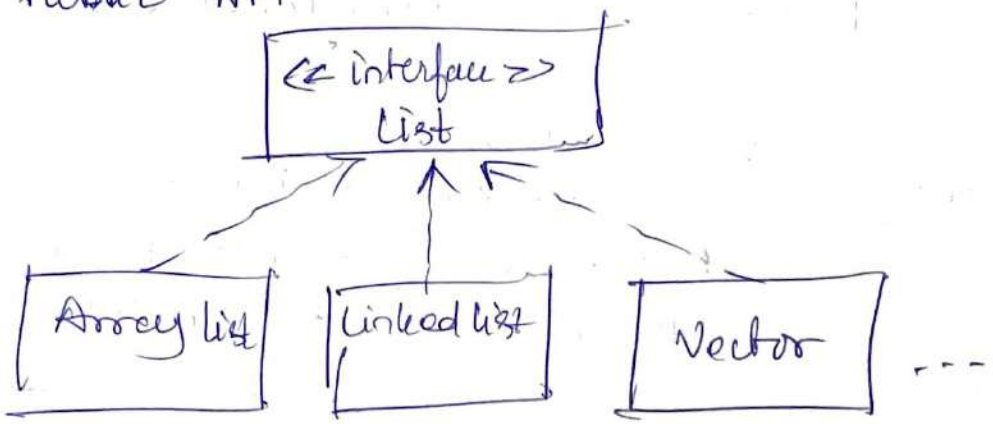
Ex:- `public interface List extends Collection{ .... }.`

Interfaces Classification

* Representative interfaces
* Mixins.

→ Define representative behaviour of subclasses.
→ Include one (or) more implementations.
→ Public API - rarely sub-classed outside API

Example Public API



Mixins

* Define additional capability of subclasses.
* Very generic - subclasses can come from anywhere

Marker Interface

* No methods // Empty Body
* Merely "marks" a class as having some property.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Ex:- java.util.RandomAccess

↓ ——→ fast random access.

java.io.Serializable

↓ ——→ Serializability.

~~Example 2 — Cloneable interface~~

Static methods in interfaces.

* Static keyword
* Not inherited.
* Invoked only via interface name.

26/05/2022

JVM Internals

JVM ⇒ Abstract Computing Machine.

Instruction set → Java Bytecode
Manipulates memory at runtime.

Responsibilities

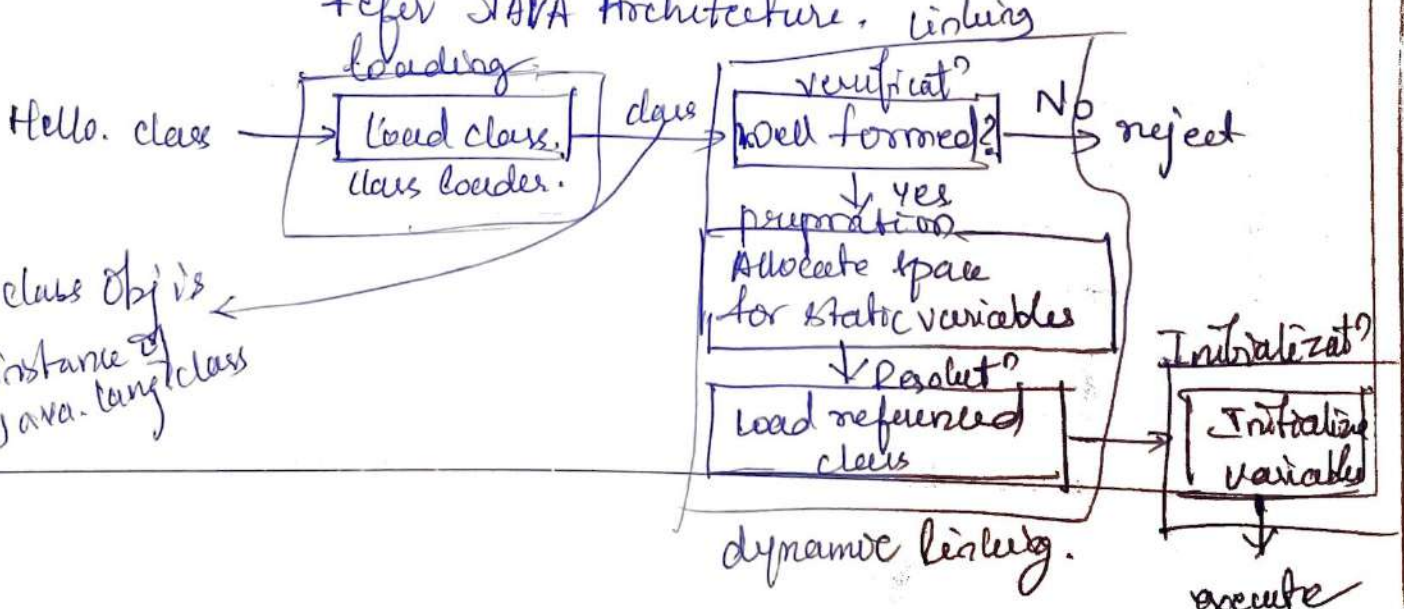
* Loading & interpreting bytecode
* Security
* Automatic memory management.

Performance

* Bytecode interpretation is much faster.
  → Java bytecode is compact, compiled & optimized.
* Just-in-Time (JIT) compilation.

Lifetime of a Type

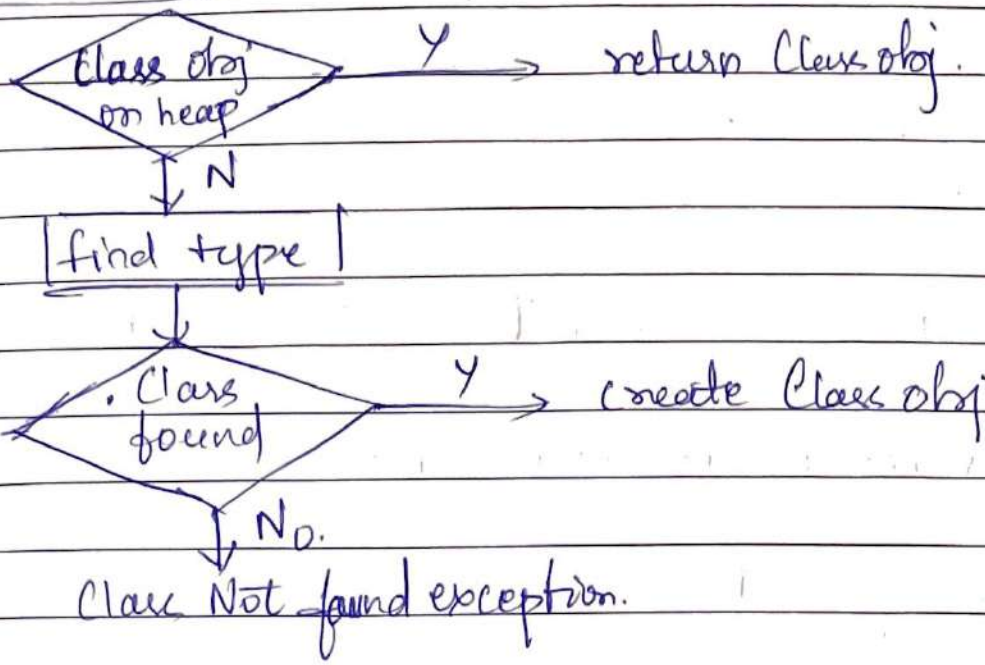
Refer JAVA Architecture.



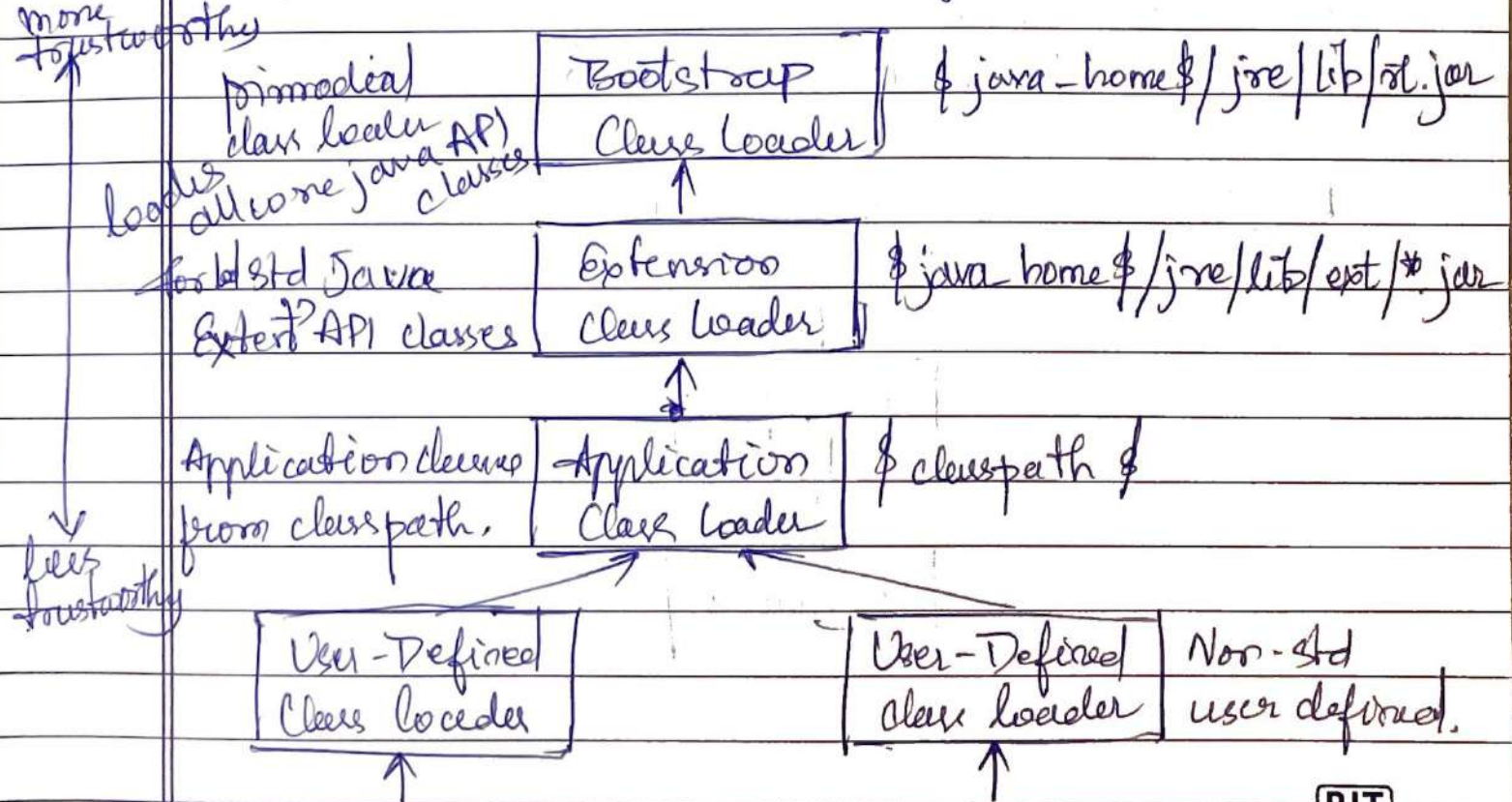
Name of the Experiment ..... Page No.:

Experiment No..... Date :

Class Loading - Loading type



Class loaders ≈ Parent - Delegation



First time Loading of class

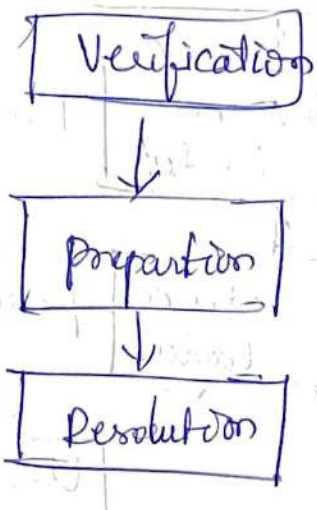
* new instance is created
* Invoking static method (Java 8) ← for interface.
* Accessing static field - Compile-time constants
* Subclass is loaded (OR) Sub-interface is loaded
* Run from command like
* Reflection.

JVM uses Class object to create a new instance.

Class obj contains meta-information?

* String getName();
* Class getSuperClass();
* boolean isInterface();
* Class[] getInterfaces();
* ClassLoader getClassLoader();

Linking

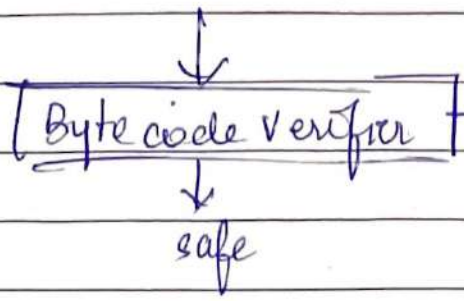


Name of the Experiment ..... Page No.:

Experiment No..... Date :

Verificat°

* Checks for conformity with language rules if it is downloaded from internet / malfound.



* final classes are not sub-classed
* final methods are not over-ridden
* No illegal method overloading
* byte integrity
  ⇒ jump inst. (if condition) does not send VM beyond end of method.

Preprat°

Storage allocat° for static fields
&
Initialize with default values
↓
Out of memory Error if no space.

Instance Creation:

Instance fields are also initialized (after static fields).

BIT

Resolution

Resolving symbolic references & loads them.

- ↳ logical references
- ↳ stored in constant pool
- ↳ At runtime, replaced with direct references.

Resolution checks
1) permissions
2) Correctness

Dynamic linking ⟹ 1) Eager loading ~ after preparation
2) Lazy loading ~ after initialization (on-demand)

Reflection (Meta info of class, obj, methods & Interface)

* Introspect known/unknown code - (class/methods/interface at runtime)
* Affect ~~before~~ runtime behaviour due to 'introspect'

Use-cases

* Class browsers in IDE
* Processing annotation eg. JUnit Testcases.
* Dynamic proxies.

Accessing Class Object

* Object Ref. getClass()
* Class.forName(String className)
* Class literals.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

a) * Class clazz = "foo".getClass(); → Class for String.
* Set set = new HashSet();
Class clazz = set.getClass(); → Class for HashSet.

b) Using Class.forName.
* Argument is fully qualified class Name
* Doesn't work with primitives.

Ex Class.forName("java.lang.String");
class.forName("[D"); → Array of primitive double
Class.forName("[Ljava.lang.String;"); // 1D array for String
Class.forName("[[Ljava.lang.String;") // 2D Array for String.

c) Class literals.
* Append .class to type name
* Works with classes, interfaces, arrays & primitives
* Examples.
String.class
boolean.class
int[][][].class
void.class.

d) Boxed Primitives have TYPE field.
* Class clazz = Boolean.TYPE; → boolean.class.
* Void.TYPE → void.class
* .class is preferred.

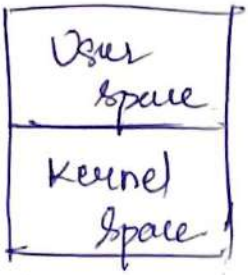
BIT

Runtime Data Areas

Method Area → To hold class data

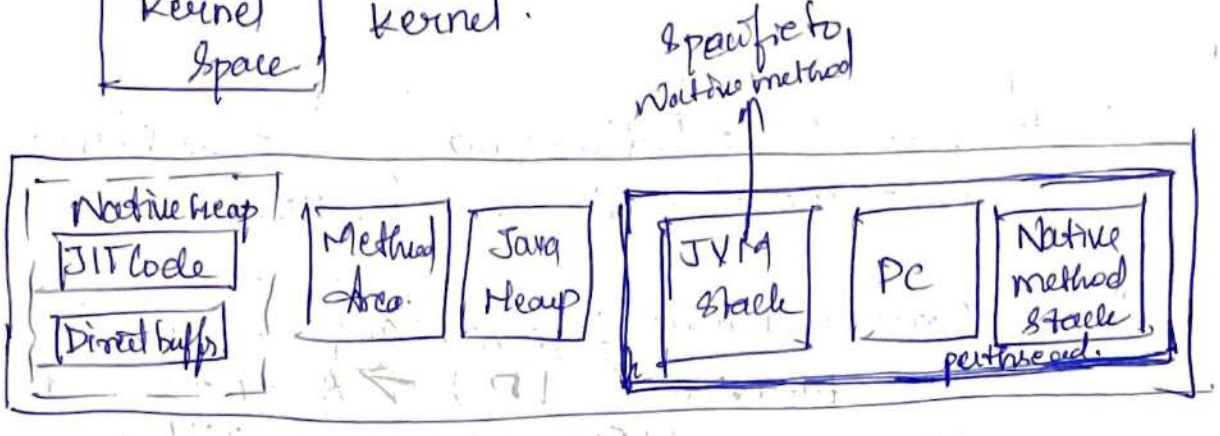
Heap → To hold class object

Process Virtual Address Space



User Programs.

kernel.



Native Heap = User space - Java Heap

Java Heap → Java Objects (include class obj, arrays)

Method Area → Class data eg. method bytecode.

Stack → method invocation state, local variable info.

Method Table inside Method Area

Used during invocation of instance methods.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Garbage Collection

* Automatic Memory Management done by JVM component called Garbage Collector.

* Reclaims dead object ~ no memory leaks
* Never reclaim referenced objects ~ no dangling ref.

(Abandoned Obj)

* Going out of scope Ex:- void go() {
  Book b = new Book();
  }
  Once the go() ends, the obj will be eligible for GC.

* Book b = new Book(); Assigning new Object
  b = new Book();

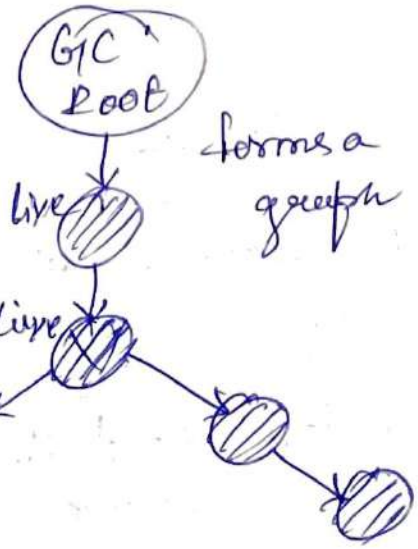
* Book b = new book(); Assigning null.
  b = null;

Specs

* Built-In / add-on
* Runs in background
* No guarantee when gc run (can invoke manually - System.gc() (or) Runtime.getRuntime().gc()) BIT
* Can cause pauses (stop-the-world) in applications.

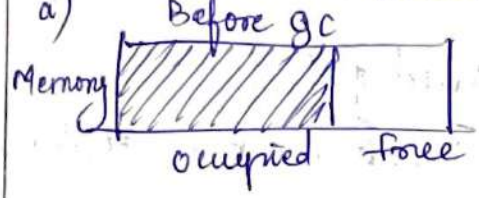
GC Algorithms

* Mark & sweep
  - a) identify live obj

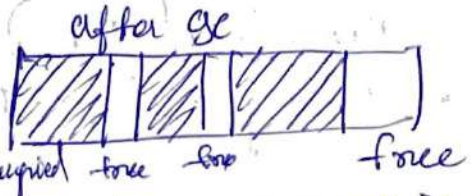


b) Sweep traces the graph and checks for the obj that are not "live".

a)



b)

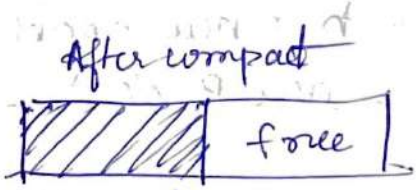


problem is memory is "fragmented"

* Mark - Sweep - Compact

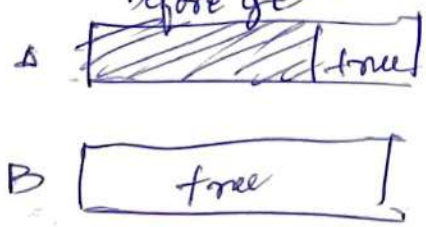
Same as above

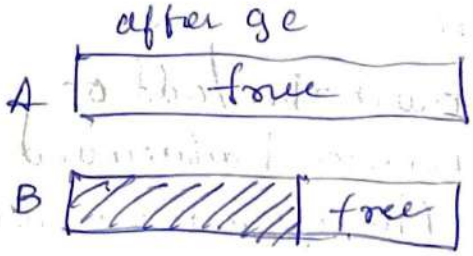
a) → b) → c)



↑ pause time.

* Mark & copy. ⇒ (MSC + Mark & copy)

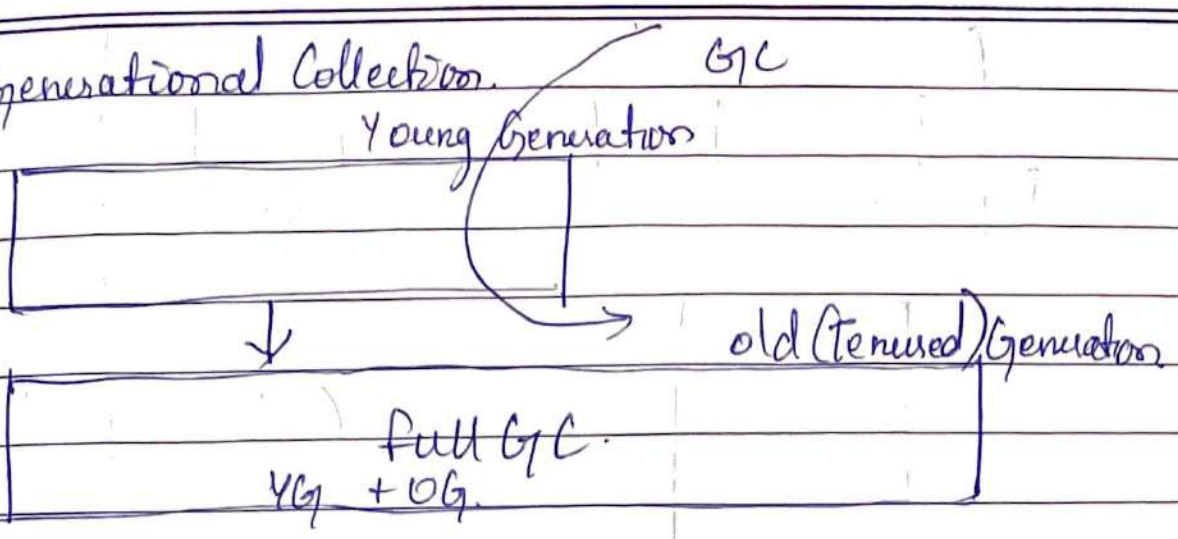




Name of the Experiment ..... Page No.:

Experiment No..... Date :

* Generational Collection.



The objs are 1st put to young gen bucket memory if the those objs sustain for more than 15 GC cycles the that would be promoted to old gen bucket memory

GC types.

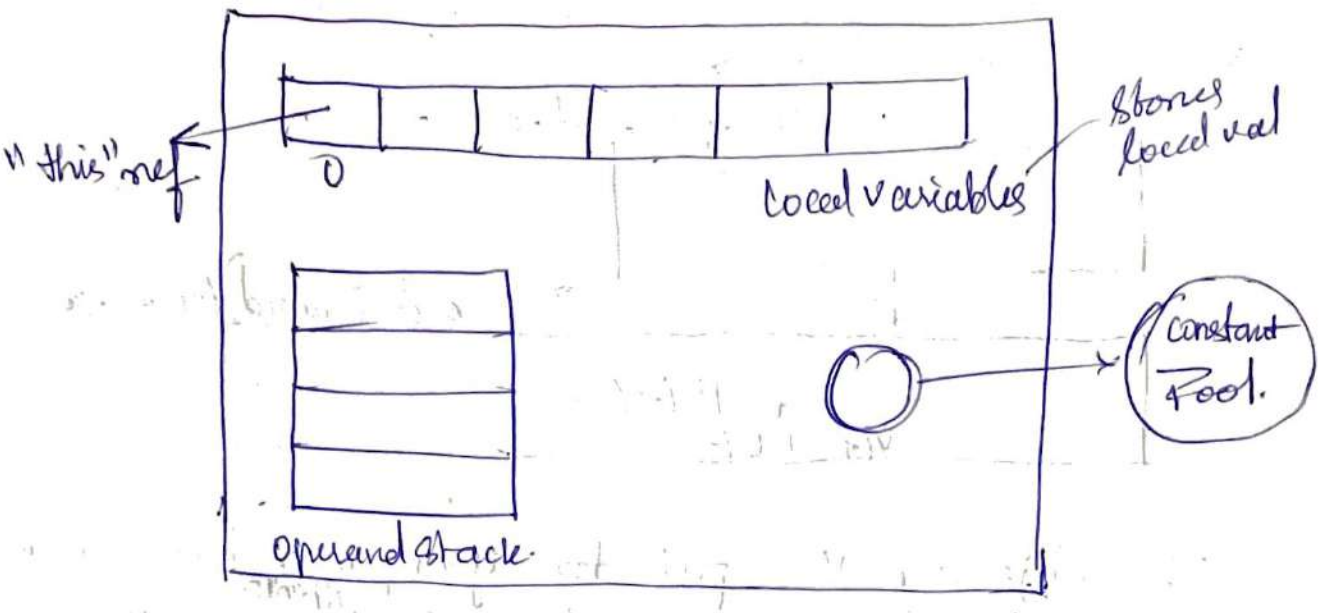
	Young	Old
Serial GC	Mark & Copy	Mark - Sweep - Compact
Parallel GC	Mark & Copy	"
CMS	Mark & Copy	Concurrent Mark Sweep

multi-threaded

STACK

* LIFO → Last In first Out.
* One stack per thread.

Stack frame Contents



Operand Stack

* similar to registers in CPU.
* Central focus of JVM's inst set
* Stores values returned by invoked methods.

```
int y = 1;
int z = 3;
int x = y+z;
```

(bytecode)

```
iconst_1
istore_1
iconst_3
istore_2
iload_1
iload_2
iadd
istore_3
```

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Exceptions

Object of class java.lang.Throwable.

Exceptions are classified into 2 type i.e, Checked & Unchecked.

Checked exceptions ⇒ Server problem, Connectivity problem (or) Database etc.

Compiler guarantees –

* Specify in throws clause.
* exception handling via try/catch.

Unchecked exceptions

Programming flaws OR System Errors.

↓ ↓

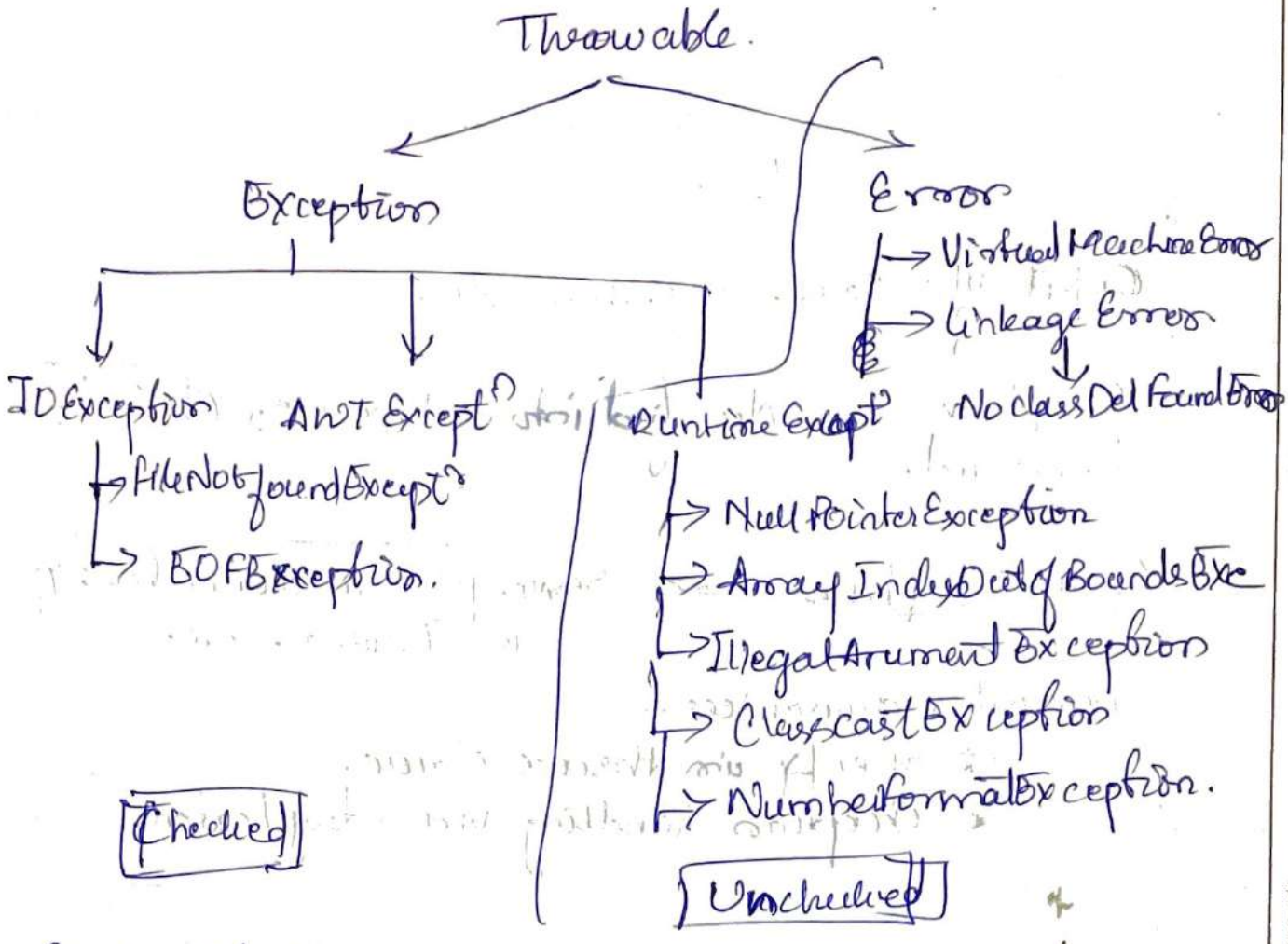
Runtime exceptions Errors.

Minimized at development & testing. Rare.

Generally, shouldn't be caught. Almost never

NOT CHECKED BY COMPILER.

BIT



finally Block

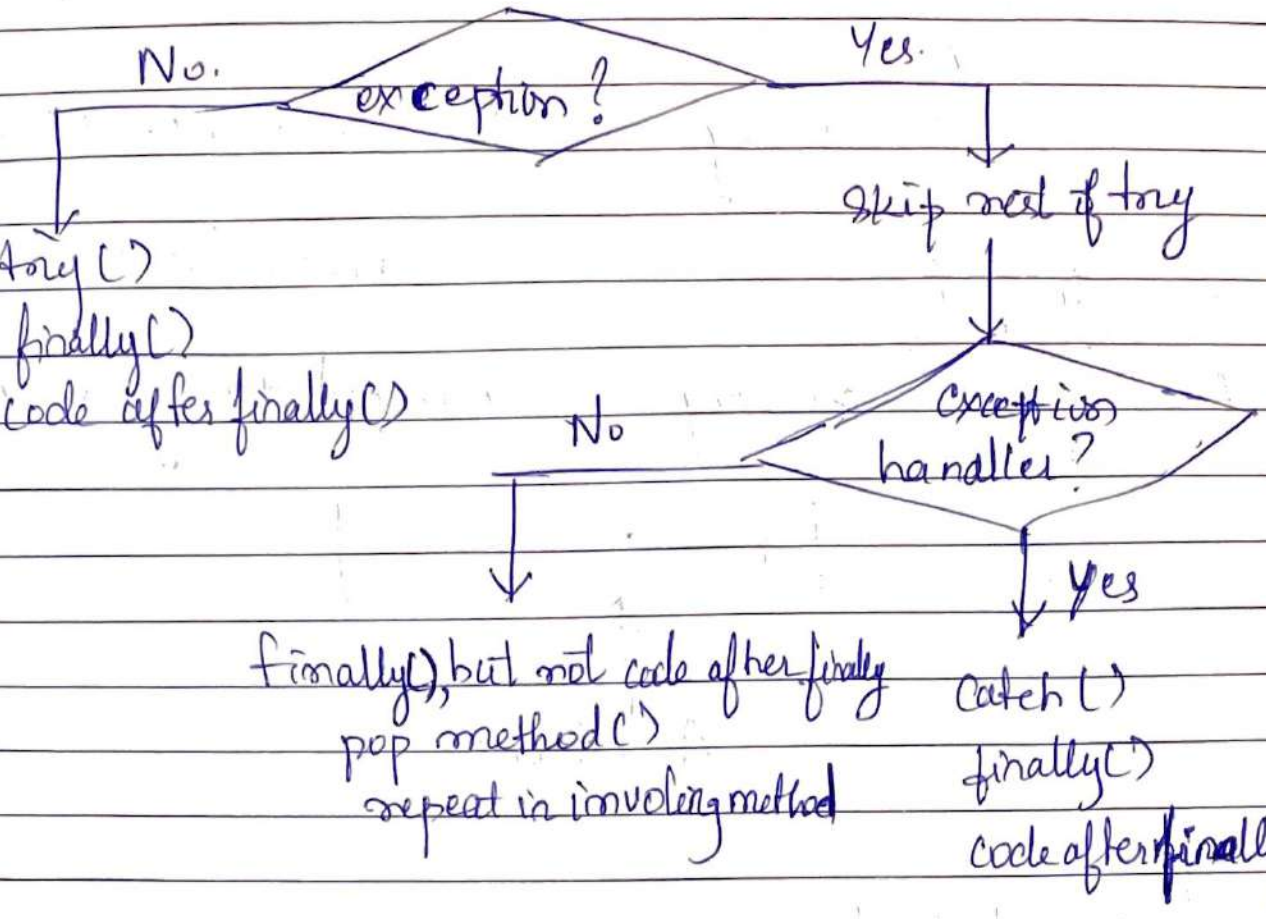
* Meant to close all the Exception.

```
try{
  ...
}
catch(Except° e){
}
catch(Except° e){
}
finally{
  obj.close();
}
```

at last

Name of the Experiment ..... Page No.:

Experiment No..... Date :



Try with resources

→ implicitely final.

try(FileInputStream in = new FileInputStream(fileName)){
    //Read data.
} catch (FileNotFoundException e){
    ...
} catch (IOException e){
    ...
}

General syntax

try(java.lang.AutoCloseable){
    ...
}

Assertions

It is all about detecting the errors during developing time & testing time.

Software Reliability = Robustness (exceptions) + Correctness (assertions)

during development few are made

Assumptions → wrong → detected during development

boolean expression

↑ test

Assertions

Syntax

* Assert boolean-expression
  - → assert ref! = null;
  - → assert index > 0;

  } AssertionError & terminates the program immediately

* assert boolean-expression : message;
  - → assert index > 0 : "index is -ve";

Name of the Experiment ..... Page No.: ☐

Experiment No..... Date : ☐☐☐☐☐☐☐☐

Benefits

* Effective in detecting bugs during development
  assertion → obviously true.
* Serve as documentation.
* assertions are like active comment.
* Can be used anywhere and moreover can be used to validate ~~pass~~ method parameters.
  → public methods ∼ exceptions
  → Assertions are disabled by default → Throws unloved AssertionError
  → non-public methods ∼ assertions.
  → Comes error from within, helps the developer to check all the methods.
* Granular level testing unlike JUnit.

Enabling & disabling Assertions

* Disabled by default
* -ea (or -enableassertions) // command line.
* -ea & -da at class & package levels.

enable → java -ea:com.semantics square.Hello MyClass
disable → java -ea -da:com.semantics square.Hello.MyClass
→ java -ea -da:com.semantics square... MyClass
(at package level)

Java IO Packages

[destinal? is also called as sink]

java.io - Stream IO
java.nio - New IO.

Decorator Pattern.

Stream IO

1st set of packages
Byte Stream
* images

2nd set of packages.
Character Streams
* text files. (R & W)

Character Encoding

* files - binary & text
* fundamentally, all files are binary
* Look alike to hardware.
* Software makes distinction.
  → Text files ∞ text processing s/w, e.g., notepad, eclipse
  → Images ∞ image processing s/w, e.g., windows photo editor
* Computer use hexadecimal for binary rep.

i.e., 'a' → 61 (hex) → 01100001
(6, 1)

↓

Something else in an image

More characters → more bytes needed.

BMP

1 - 16 bit code point is used.
i.e., Hexadecimal code.

Non-BMP

2 - 16 bit code point is used.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Processing text is complex → # langues + # ways of process

↓

encoding scheme.

Every file uses some encoding scheme

↓

character ←→ # hexadecimal # ←→ binary

Encoding 'a' → 61 (hex code) → 01100001

Decoding: 01100001 → 61 → 'a'

Examples :- ASCII, UCS-2, UTF-16, UTF-32.

Implement some character set.

* Unicode is now used for encoding & decoding

UTF - Unicode Transformat' format.

Unicode

* Maintained by Unicode consortium
* Backwards compatible with 7-bit ASCII
* UTF-16 → each char 16 bits.
* Covers over 120k characters & 129 scripts.

* Initial version assumed 16 bits for 65535 characters
* Basic Multilingual Plane
* Non BMP around 15%

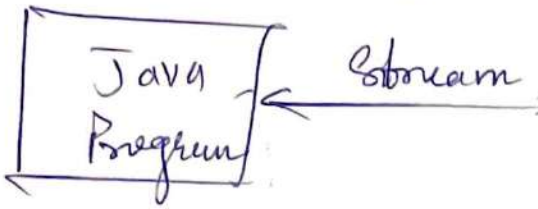
// Always decode using same or compatible encoding scheme //.

BIT

Stream IO

(reading & writing are handled by streams)

Stream is a connection b/w Java Program & destination/source.



Stream is a connection b/w Java Program & data source/sink & it is specific.

- Input stream ⇒ To read data from source we use i/p stream
- Output stream ⇒ To write data to the destination, we use o/p stream.

While working with stream, There are 3 operations involved.

1) Open stream
2) Read/write data depending on open stream if it is input/output.
3) Close stream. (sockets/filehandlers ⇒ OS)

Closing streams free up system resources & this would help in avoiding resource leaks.

Ex:-

```
FileInputStream in = null;
try {
    in = new FileInputStream (filename); //open stream
    // read data.
} catch (FileNotFoundException e) {
    ...
} finally {
    try {
        if (in != null)
            in.close();
    } catch (IOException e) {...}
}
```

close stream should always be in a finally block to free up system resource