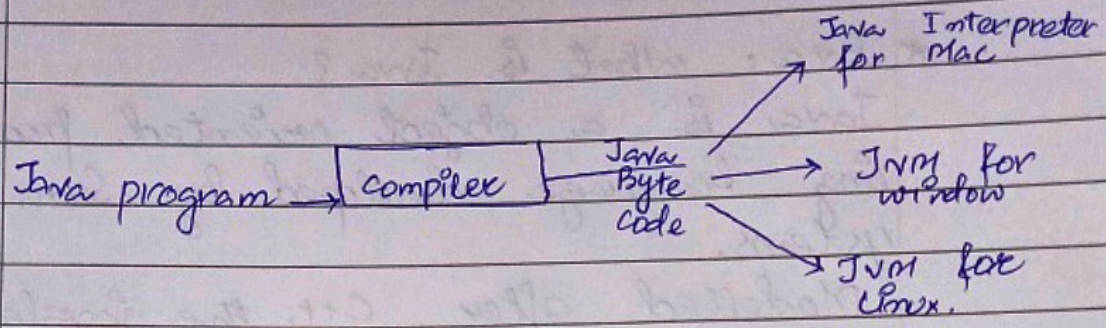
__/__/____

★ Java: What is Java?

- Java is a object oriented programming language developed by Sun. Microsystems.
- Modelled after C++, the java language was designed to be small, simple, and portable accross platforms & operating systems.
- It is also called Internet language. (Due to its platform independence it has become internet language).
- It is Secure programming language (JRE - Java Runtime environment).

- Java Virtual Machine (JVM):
- When Java program is compiled. It is converted to byte codes that are portable Machine language of a CPU architecture known as Java virtual machine (JVM)
- The machine language for JVM is called Java Byte code.
- JVM interprets & executes byte codes.

__/__/____



→ History of Java:

- Java was developed by James Gosling and his team at Sun microsystems in the mid 1990s.
- It was 1st named as Oak (Tree)
- It was released in 1995 as a programming language with promise of "Write once, Run anywhere" due to its platform independence.

- Fall 1995: Java 1.0 - the original release.
- Spring 1997: Java 1.1 - An update that improved support for graphical user interfaces (GUI)
- Summer 1998: Java 2 version 1.2 - A huge expansion, making the language a general purpose prog. lang.

__/__/____

- Fall 2000: Java 2 version 1.3 - A release for enhanced multimedia.
- Spring 2002: Java 2 version 1.4 - An upgrade for internet support, XML capabilities and text processing.
- Spring 2004: Java 2 version 5 - A release offering greater reliability & automatic data conversion.
- Winter 2006: Java 6 - An upgrade with a built-in database & web services support.
- Summer 2011: Java 7 - New core language improvements, memory management and Nimbus gui.

* Features of Java:
- Java is Platform Idependent: Write Once Run anywhere (WORA) due to its byte code & JVM.
- Object Oriented: Organize code into reusable objects & classes.
- Memory Mangement: Automatic garbage collection to manage memory efficiently.

__/__/____

- Security: JRE
- All the programs in Java are run under an area known as the Sand box.
- Built-in features like Sand boxing & Security managers.

- Multithreading:
Supports multithreaded programming which allows to write programs that do many things simultaneously.

- Exception Handling:
Effective mechanism for managing errors.
- Portability:
Code can run on different platforms without modification.
- Robust:
Java's robust nature is due to its strong type checking, exception handling mechanism and automatic memory management.

__/__/____

- Similarities:

1. Syntax Basis: Java C & C++ Share similar syntax roots.

2. Control Structures:
All three languages use common control structures like loops & conditionals.

3. Compiled languages:
Java C & C++ are compiled languages, where code is compiled before execution.

- Differences:

1. ~~Memory~~ Management
   - Java: Manual memory management with garbage collection.
   - ~~C/C++~~ C/C++ ~~Java~~

__/__/____

1. Memory Management:

- Java: Automatic memory management with garbage collection.
- C/C++: Manual memory management; allocate/deallocate memory.

2. Platform independence:

- Java: Designed for platform independence using the JVM.
- C/C++: Code needs to be recompiled for each platform.

3. Pointers:

- Java: No direct pointer manipulation; memory references are managed by the JVM.
- C/C++: Supports pointer arithmetic, allowing direct memory manipulation.

4. Portability:

- Java: Highly portable due to JVM. WORA
- C/C++: Code may need modification for different platforms.

__/__/____

5. Runtime Environment:

- Java: Requires the JRE to execute.
- C/C++: Executed directly by the operating system.

→ General Structure of Java Program:

Documentation Section → Suggested
/* set of comment lines */, //, /** ... */
Package Statement → optional
↳ Divide program or project in logical hierarchy
↳ namespace resolve → divide in folders.
↳ It informs the compiler that the class defined here belongs to this package. Ex. package Student;

Import Statement → optional
↳ like #include in C/C++
↳ Import JDK files already defined.
↳ programs itself.

Interface statement → optional

__/__/____

↳ used only when we wish to implement the multiple inheritance feature in program.

↳ There is no multiple inheritance in Java.

Class Definitions → Essential

↳ Primary & essential elements of Java.

Main method class → Essential

↳ Starting point of program.

→ Main:

C

```
void main()
{
  ---
}
```

C++

```
int main()
{
  ----
}
```

Java:

```
Class demo{
public static void main (String args[])
{
---
}
```

__/__/____

→ The keyword public indicates it is globally accessible.
→ The keyword Static ensures it is accessible even though no objects of the class exist.
  ↳ Persist value
  ↳ Class variable
  ↳ Co-ordinator → that our program will start here..

→ The keyword void indicates it doesn't return a value.
→ String args[] are command line arguments.

→ Save using classname.
  classname.java
  eg. demo.java.
→ If multiple classes exist - Save main method class name.

- Compile & Interpret Language:

demo.java ——compile——→ demo.class (bytecode)

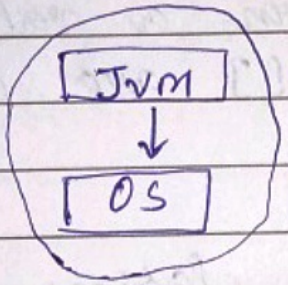
——execute——→ demo.exe (machine lang.)

__/__/____

- Javac demo.java
  - For compilation in command prompt.
  - Javac → compiler → generate bytecode
  - Itself a program.

→ Java bytecode move locally or through a network. (demo.class)

- Class loader – bring class files to JVM.
- Bytecode verifier – runtime errors.



- Interpreter – Line by line - compile
- Just in time compiler (JIT)
  - As a whole compiles → discards repitition.
  - Efficiency increased for a program
  - Invoked on demand.
  - Cannot invoke sometimes.

__/__/____

→ Java is usually called pure OOP but is not actually.

like int a;
4 but it is not necessary to make object here.

→ Small talk - Pure OOP.

→ NO. linking in Java.
→ Class loader loads class files in JVM.
→ Bytecode verifier checks runtime errors.

→ Interpreter - Line by line

→ JIT (Just in time) - compiles as a whole & discards repetition.
. Efficiency increased for a program.
. Invoked on demand
. Cannot invoke sometimes.

→ Java is not purely OOP.

→

__/__/____

- Lexical Issues:

- Whitespaces:
  - White spaces in Java may be:
    - A space
    - A newline
    - A tab.

- Identifiers:
  - Used for naming variable names.
  - Can start with:
    - _ (underscore)
    - $ (dollar symbol)
    - Uppercase or lowercase alphabets.

- Literals:
  - It is a constant created in java.
    78 is an integer
    "me" is a string.

- Comments:
  - Three types:
  - Single line
    - //

__/__/____

- Multiline
  /* ...
  ...
  */
- Documentation
  /**
  ....
  ....
  */

- Seperators:
  - () → Parenthesis
  - {} → Braces
  - [] → Brackets
  - ; → Semicolon
  - , → Comma
  - . → Period.

- Data types:
- Primitive / Simple:

1. Integers:
   - long 64 bit
   - Int 32 bit
   - short 16 bit
   - byte 8. bit

2. Floating
   - double 64 Bit
   - float 32 bit

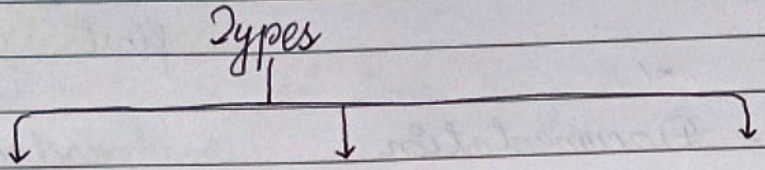
3. Characters:
   - char 16 Bit

4. Boolean
   - boolean 1 (not fixed)

Data type	Default value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String	null
boolean	false.

__/__/____

- Variables:



Local	Instance	Class
• variable that is declared inside the method.	• variable that is declared inside the class but outside method. It is not declared Static.	• A variable that is declared st- ated is called Static (also class) variable. It c- an't be local.

```
class myfirstprog
{
public static void main (String args[])
 {
   int a = 10; // local variable
   System.out.println("Hello world");
 }
}
```

- System: Imported class.
  Standard Input & output defined.
- Out: Object

__/__/____

- println: method
  ↳ nextline.

- Access modifiers can be used for local variables (public, protected, private).
- No default value for local variables.

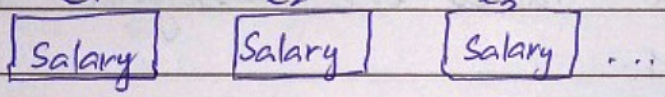
- Instance:
  Inside class outside the method.
- These are not declared static.
- Access modifiers can be given for instance variables.
- default values are assigned to instance variables.
  ↳ To prevent errors by ensuring they always have a value before being used in code.
- Instance variables are used for temporary storage.

- Class variables:
  - Also known as stack variables because these belong to whole class.
  - Declared like instante variables but

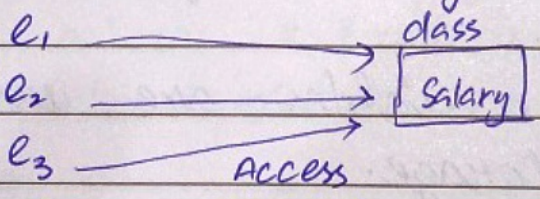
with static keyword.
- per class - only one copy of class variable.

- Static Memory?
- It is created when program starts & destroyed when program stops.
- Java has automatic garbage collection.

eg: double Salary;
e1 = new employee();
e2 = ...
...



Static double Salary;



- If one will modify salary it will be modified for all.

__/__/____

- Java Operators:

1. Arithmatic - (+, -, /, %, *)
2. Relation Operators - (==, !=, >, <, >=, <=)
3. Logical operators - (&&, ||, !)
4. Bitwise operators - (&, |, ^, <<, >>)
5. Assignment operators - (=, +=, -=, *=, /=, %=)
6. Miscellaneous operators - (?:) → conditional or Terneny.

```
- void main()
{
    int a = 10;
    int b = 200;
    printf("%d", a++);   //output : 10
    printf("%d", --b);   // output : 199
    int x = a++;         // x = 12
    printf("%d", x);     // output : 12
     int y = --a;        // y = 10.
     y = a++;            // y = 11.
     printf("%d", y);    // output = 11
}
```

- Conditional or Miscelleneous operator (:?)
- Also known as ternary operator.

__/__/____

- used instead of if else.
- 3 operands
- used to evaluate boolean expression.

```
if (a < b)
{
   if (a)
     { ... }
   else
     {
       if (b) { .. }
     }
```

→ (a>b) ? a : b

- Precedence:

10 + 2 * 3 + 5 * 4 + 12 / 6 + 5 * 20 / 2
= 10 + 6 + 20 + 2 + 100 % 2
= 88

```
class event
{
  public static void main (String args[])
  {
    int x = 10;
    if (x % 2 == 0)
  {
```

__/__/____

```
    System.out.println("No. is even");
}
else
    System.out.println("No. is odd");
}
```

- if, if else, if else if ladder, nested if all are same as in C/C++.
- Switch statement:
  - break - optional
  - default - optional
- Rules to apply switch statement
- variable (either integer, convertable integer)

- Reading Input from user:

- In java we can read input from user using Scanner class.
- System.out → output
- System.in → standard input stream.

```
Scanner obj = new Scanner(System.in);
```

__/__/____

```
class demo
{
    String name;
    int age;
    float marks;
    public static void main (String args[])
    {
        demo d = new demo(); //default constructor call.
        d.name = "Shifa";
        d.age = 30;
        d.marks = 55;
        Scanner sc = new Scanner(System.in)
        System.out.println("Enter your name");
        d.name = sc.nextLine();
        d.age = sc.nextInt();
    }
}
```

nextInt
nextLine
nextFloat
nextDouble

__/__/____

- For each loop:
  - Only in Java
  - Mainly for array traversal.

```
int a[] = {10, 20, 30, 40, 50};
int i;
for (i = 0; i < 5; i++)
    System.out.println(a[i]);
```

or

```
for (int x : a)   { x has same datatype as of a.
    System.out.println(x);
```

- Header files in Java:
  - as we write #include in c/c++
  - In Java we write import java....
  - eg: import java.lang.* - for S.O.P.
    import java.util.* - for Scan.

__/__/____

Unit 2

- Why we use class?

- It is a new data type.

```
class demo
    demo d;
    ↑
    new datatype
```

- It gives a template.
- It has a logical existance.

class = template, new data type, blue print of an object.

- Syntax of class:

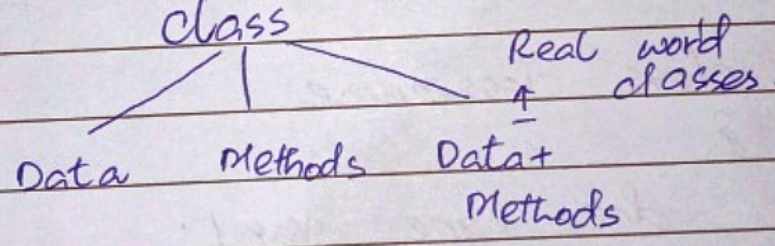
```
class class_name
{
    data_type var1;
    datatype var2;
    returntype method name (paramdevs)
    {
        ...
    }
    n methods;
}
```

__/__/____

- It is not important every class has a main method.
- If there are 10 classes in a program
- only one will have main method which is beginning of the program.
- The class containing main method program will be saved with that class name.

- Every class file is created when we compile.

Javac demo.java




```
Class Box //Object
{
    double length;
    double breadth;
    double height;
}
```

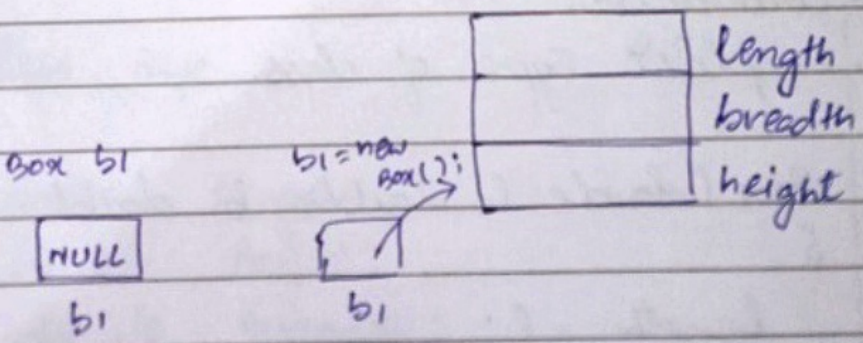
__/__/__

```
class Box_demo
{
public static void main(String args[])
{
Box b1; // Reference
S.o.P(b1); // Compile time error
b1 = new Box(); // memory allocated
}
}
```

Encapsulating main & Starting point of execution.

```
b1.length = 10;
b1.breadth = 20;
b1.height = 15;
double vol = b1.length * b1.breadth * b1.height;
S.o.P("volume of box b1 =" +vol);
```

Save as Box_demo.java.



- Local variables hide instance variables.

```
void set_dim(double length, double breadth, double height)
{
    this.length = length;
    this.breadth = breadth;
    this.height = height;
}
```

- this overcomes it → it refers to current object.

```
Box b1 = new Box();
```

↳ constructor - as object created initialise it.

- Constructor:
. Implicit type of class.

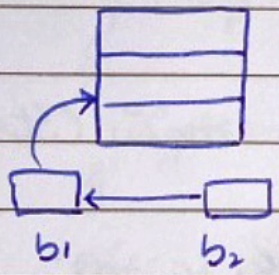
```
Box(double l, double b, double h)
{
    length = l;
    breadth = b;
    height = h;
}
}//Box class end
```

__/__/____

```
class demo{
public static void main (String args[])
{
  Box b1 = new Box (10,5,20);
  Box b2 = new Box ( ); //error because
                        box constructor is specified.
}
```

- if b2 then we have to declare a constructor declared with no arguments.

- Constructor overloading:
  - Can be multiple constructors.




```
Class Box
{
  double length;
  double height;
  double breadth;
  Box (double length, double Breadth, double height)
  {
```

```
this.length = length;
this.height = height;
this.breadth = breadth;
}
Box(double length) // cube
{
  this.length = length;
  breadth = length;
  height = length;
}
Box();
}

class Box_demo{
public static void main(String args[])
{
Box b1 = new Box(10,20,30);
Box b2 = new Box(10);
Box b3 = new Box();
b3.length = -1;
b3.breadth = -1;
b3.height = -1;
}
}
```

Stack:

```
class stack
{
  int arr[] = new int [10];
  int n = arr.length;
  int tos;
  stack()
  {
    tos = -1; }
  public static void push(int item)
  {
    if (top == n)
    {
      System.out.println("overflow");
    }
    else
    {
      arr[++top] = item;
    }
  }
  static int pop()
  {
    if (tos == -1)
    {
```

```
        System.out.println ("Underflow");
    }
    else
    {
        int item = arr[tos];
        tos--;
        return item;
    }
}
static void display( )
{
    int i;
    if (tos == -1)
    {
        S.O.P("Underflow ... empty stack");
    }
    else
    {
        for (i=tos; i>=0; i++)
        {
            S.O.P("Stack arr[i]+" ");
        }
    }
```

__/__/____

- Queue Implementation -

↳ Method Overloading:

- Polymorphism:

↳ One thing can be implemented in multiple ways.

abs[-1] } → will work on any numeric data type.
sqrt[ ] }

- Parameters should be different for overloaded method.

eg: Sum (2,3)
Sum (2,3,5)
sum (2.9, 1.6)

- Same name, same class, but different number of arguments or data type of arguments.
- Then we can say method is overloaded.

```
class dem {
  void test ()
```

```
{ System.out.println("No. arg. method");
}
void test(int a)
{
  System.out.println("Single arg. test method");
}
void test(int a, int b)
{
  S.o.p("Two arg. test method." + a + b);
}
double test(double a)
{
  S.o.p("Return singl arg. test method");
  return a;
}
}
class overloaddemo{
psvm(String args[])
{
  demo d = new demo();
  d.test();
  d.test(2,3);
  d.test(100);
  double x = d.test(20.9);
}
}
```

__/__/____

- return type if different - can't determine overloading.

- No. of arguments
- Type of arguments
- Automatic type conversion (Implicit conv.)

Sum method - no. of args, type of args, Implicit.

```
class Test {
  void meth (int i, int j)
  {
     i = i * 10;
     j = j + 2;
  }
}

class test_demo {
public static void main (String args[])
{
   int a = 10, b = 20;
   Test obj = new Test ();
   obj.meth (a, b);
   System.out.println ("value of a & b" + a + b);
}
} // output - 10 20 - call by value
```

__/__/____

- Call by reference:

. object pass as parameter.

```
class Test
{
  Int a, b;
  Test (int i, int j)
  {
     a = i;
     b = j;
  }
  void meth (Test ob)
  {
     ob.a = ob.a * 10;
     ob.b = ob.b * 2;
  }
}
class Test_demo {
public static void main (String args[])
{
  int i=10, j=20;
  Test t1 = new Test (i, j);
  S.O.P("value of a = " + t1.a + "b = " + t1.b);
```

t1.meth(t1);
System.out.println("value of a=" + t1.a + "b=" + t1.b);
}
}

Swapping variables - using call by reference

- Default & public:
. Default is like public.
. Difference is if program is in different packages, public is accessed default is not.

- Constructors:

- Constructors can be:
. with no arguments.
. Parameterized constructors.
. Copy constructors.

eg: class Box{
double length;
double breadth;
double height;
Box(){
length = -1;

```
    breadth = -1;
    height = -1;
}
Box( double l, double b, double h )
{
    length = l;  breadth = b;  height = h;
}
Box (double x)
{
    length = x;  breadth = x;  height = x;
}
Box( Box ob)
{
    length = ob.length;
    breadth = ob.breadth;
    height = ob.height;
} //Copy Constructor
```

- Check whether if value of two objects is equal or not.

```
class test
{
    int a, b;
    test (int i, int j)
    { a = i;
      b = j; }
```

__/__/____

```
Boolean equals( test obj )
{
  if (obj.a == a && obj.b == b)
  {
    return true;
  }
  else
    { return false; }
}
class demo{ psvm(String args[]){
  test t1 = new test(10,20);
  test t2 = new test (100,200);
  test t3 = new test (10,20);
  System.out.println(t1.equals(t2));
  System.out.println(t1.equals(t3));
 }
}
```

// Return Object

return type object

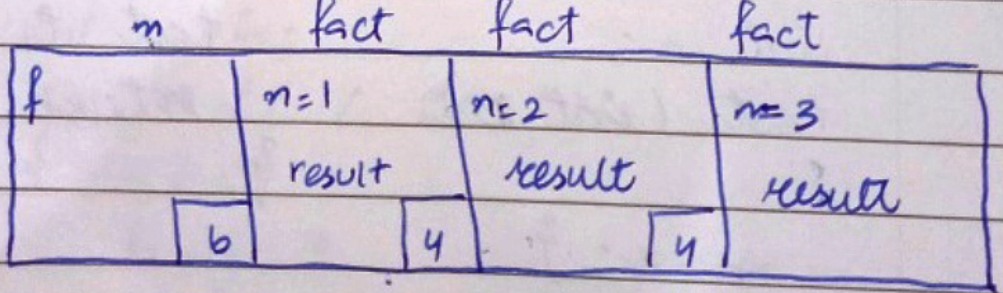
```
class test
{
  int a;
  test (int x)
  {
    a = x; }
```

```
test incrementTen( )
{
  test t1 = new test(a+10);
  return t1;
}
```

__/__/____

⇒ Recursion:

```
class factorial
{
  1 int fact(int n)
    {
      2 int result;
      3 if(n==1)
          return 1;
      4 else
          5 result = fact(n-1)*n;
            return result;
    }
}
class demo{
public static void main(String args[])
{
a- factorial f = new factorial();
b- System.out.println("Factorial of 3=" +
                      f.fact(3));
}
}
```



__/__/____

- Iterative is faster than recursive.
- Stack overrun (if necessary..)
  ↳ memory out of bound error

Print an array using recursion - H/W...

```
class demo{
  int a;
  private int b;
  public int c;
  demo ( )
  {
    int x;
    b = x; }
  void set (int x)
  {
    b = x;
  }
  int get ( )
  {
    return b;
  }
}
```

```
class demo_main {
  psvm (String args[])
  {
    demo d = new demo();
    d.a = 10;
  ×d.b = 20; → d.set(20);
    d.c = 30;
    int res = d.a + d.b + d.c;
                ↓d.get();
    S.o.p (res);
  }
}
```

- Private members can be accessed through methods.

__/__/____

- gets & sets used.

- Final Keyword:

  final int a = 10; → constant.

- The variable ~~what~~ with which final is used is as would as constant. & can't change its value all over program.
- Declaring that variable as a constant.

- Why is main static?
- so that we don't need to make object to access main.
- It is starting point of program.

- Class demo{
  static int a=10;
  Static int b;
  static
  {
    System.out.println("Static block executed");
    b= a* 20;
  }

```
Static ← void display()
       {
         System.out.println("value of a & b = "+a+
                                    "  "+b);
       }
public static void main(String args[])
{
    demo d = new demo();
  d.display();
          ↳ new → add - display();
}
// output
1. Static Block executed
   value of a & b = 10  200.
4 Both executed with same output.
```

✓- **Static:**

- class - combination Data + Methods.
- Access data directly (i.e., it belong to class not objects then we use static keyword.

```
Static int a;
Static int a=10;
```
} → both are valid.

```
class demo{
Static int a;  // If we want to
Static int b;  // initialize in a
               // specific method →
               // Static block
Static void meth()
{
  System.out.println("value of a="+a);
  System.out.println("value of b="+b);
}
Static    // used to initialise static
{         // variables
  System.out.println("Static block exe");
  b=a*10;
}
public Static void main(String args[])
{
 x demo d=new demo(); x
   d1.meth(); x → can call it directly
   meth();
}
}
```

- Execution starts from main only if there is no other static variable in the program.

__/__/____

- If there is any other static variable ist that will be executed.
eg: Static block in previous prog. will be executed ist.

- When a member is declared static it can be accessed before any object of its class & or without reference to any object.

- Methods declared static have certain restrictions:
  - They can only directly call other static methods.
  - The can only directly access static data.
  - They cannot refer to this in any way.

* Nested Class:
- class enclosed within class.

```
class A {
    var 1;          // 1 can be accessed by
    class B {}      B but 2 can't be
        var 2; }    accessed by A.
}
```

__/__/____

- Inner class can access variables of outer class but not vice versa.
- Inner class cannot exist independently.
- Inner class can have no existance without outer class.

Ex:

```
Class outer{          // Static outer_x.
  int outer_x=100;
  void show()
  {
    inner in = new inner();
    in.display();
  }
  class inner          // → Static class inner ↓ here.
  {
    void display()
    {
      S.o.p("value of outer var = "+outer_x);
    }
  }
}
```

→ nested class.

error on static → Static method can only access static variables/members

__/__/____

```
class demo{
  public static void main (String args[])
  {
     outer obj= new outer();
     obj.show();
  }
}
```

- we can make object of inner class in outer class, but can't access inner class variables.
- Nested class can be static or non static.
- Static & Inner class.

System.out.println("value of a =" + a);

→ " " → String, + → concatenated with

→ Here is no limit of arguments to a function.

- Passing variable number of arguments:

- Java provides feature where we can pass variable number of arguments.

__/__/____

eg: void show(int ... v) → show();
show(10);
show(10,20);

↓

0 to any no. of arguments.

- Internally v is an array where our arguments get read.

```
void show(int ... v)
{
  System.out.println("no. of args = " + v.length);
} //count no. of args passed.
class demo_var{
  static void show(int ... v)
  {
    S.o.p("No. of args to show = " + v.length);
  }
  static void show(boolean ... x)
  {
    S.o.p("No. of args to boolean show" + x.length);
  }
  Static void show(String s, int ... a)
  {
    S.o.p("value of s = " + s);
    S.o.p("No. of args = " + a.length);
  }
```

__/__/____

```
public static void main(String args[])
{
  Boolean x = true;
  Show(10, 20);
  Show( );  // ambiguity → error
  Show(x);        ↳ so we avoid using
  Show("a", (2,13));   variable length args.
}
```

- Constructor chain:

• Calling a constructor within a constructor.

```
class Box {
  double width;
  double height;
  double length;
  Box( ){                    this(10);  → constructor chain
    width = length = height = -1;
  }
  Box(double x)
  {                          this(x, x, x);  → call this constructor
    width = length = height = x;
  }
  Box(double d, double l, double h)
```

```
{
  width = d; length = l; height = h;
}
Box(Box ob)
{
    width = ob.width;
    length = ob.length;
    height = ob.height;
}
public static void main(String arg[])
{
    Box b = new Box();
    Box b1 = new Box(40);
    Box b2 = new Box(5, 10, 15);
    Box b3 = new Box(6);
}
}
```

Imp ★ Garbage collection:

- Garbage collection is automatic in Java.
- There is no concept of Destructor in Java.
- Automatic garbage collection in JVM

__/__/____

→ class System contains method gc → which is garbage collector (System.gc).

- What is freed by Garbage Collector?

1. Unreferenced objects:
Or unreached objects are freed.

2. We can make objects unreachable:

⇒ t_1 = NULL.

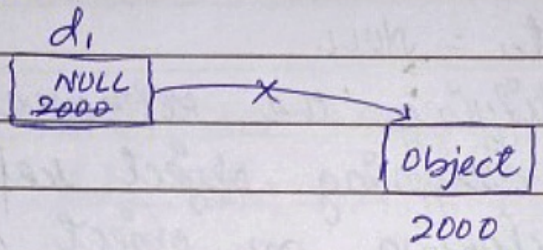
i) Nullifying the reference
ii) Re-assigning object reference.
iii) Declaring an object within a method or a Block.
iv) Island of isolation.

i) Nullifying the references:

```
class demo{
    String obj_name;
    demo(String name)
    {
        obj_name = name;
    }
```

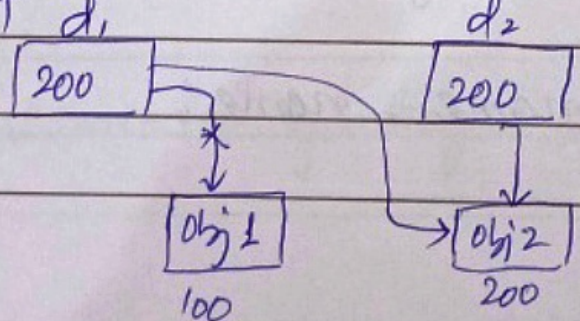
```
public static void main(String args[])
{
    Demo d1 = new Demo("object");
    :
    d1 = null; // eligible for garbage collection
}
}
```

- Automatically collected (eligible) → out of scope.



ii) Reassigning Object references:

```
public static void main (String args[])
{
    show();
    Demo d1 = new Demo("object 1");
    Demo d2 = new Demo("object 2");
    d1 = d2; // reassigning
}
```



__/__/____

iii) Declaring an object within a method or block:

```
static void show()
{
  demo temp = new demo("Temporary object");
  System.out.println("Inside show()");
}
```

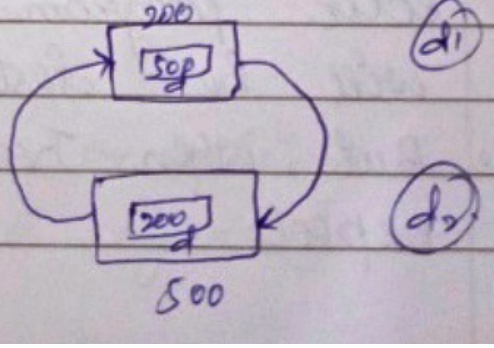
- when an object goes out of scope it is eligible for garbage collection.

iv) Island of Isolation:

```
class demo
{ demo d;
  public static void main(String args[])
  {
    demo d1 = new demo();
    demo d2 = new demo();
    d1.d = d2;
    d2.d = d1;
    d1 = NULL ; d2 = NULL;
  }
}
```

// Both objects will become unreachable & Both will be eligible for garbage collection.

Prequisite of Island of Isolation.



__/__/____

- System.gc(): Request to JVM to collect garbage.
- May or may not execute sometimes because it is automatic.

- Finalize Method:

```
protected void finalize() throws Throwable
{
   // No code.
}
```

- Whenever System.gc is called before that it will call finalize method.

- Ways To request JVM to run GC:

- Once we make an object eligible for gc it will not destroy immediately by the gc.
- Whenever JVM runs the garbage collector program then only the object will be destroyed.
- But when JVM runs GC we can't expect.

__/__/____

- We can also request a JVM to run gc by using system.gc method.
- System class contains the static method gc for requesting JVM to run gc.

- Finalize:
  - Based on requirement we can overwrite finalize method for cleanup activities.

eg: object class finalize has an empty implementation, thus it is recommended to overwrite finalize method.

  - To dispose of system resources.
  - Finalize method is called by Garbage collector not by JVM, but GC is a module of JVM.
  - And finalize method ~~not~~ is invoked only once for an object (can't be invoked more than once)
  - If it throws an exception and its uncaught, it is ignored and finali-

__/__/____

zation of that object is terminated.

- Final activities - finalize method.

★ Program to count number of employees.

```
class employee
{
    String name;
    int id;
    int age;
    static int nextId = 1;
    employee (String s, int a)
    {
        name = s;
        age = a;
        id = nextId++;
    }
public static void main (String args[])
{
    employee e1 = new employee("xyz", 23);
    employee e2 = new employee("AB", 20);
    employee e3 = new employee("CD", 25);
    e1.showNextId();
```

e2. Shownextid();
employee e4 = new employee("AAA", 19);
employee e5 = new employee("BBB", 18);
e4. Shownextid();
e4 = NULL;
e5 = NULL;
System.gc();
System.runFinalization(); // explicitely call Finalize method.
}

void Shownextid() // before main method
{
System.out.println("Next available ID =" + nextid);
}
protected void finalize() throws Throwable
{ // Before main
nextId--; // To decrement nextid & unreference interns
}

__/__/____

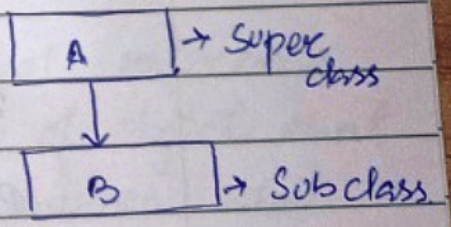
Inheritance:

	Default	Private	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same Package Subclass	Yes	No	Yes	Yes
Same Package non-subclass	Yes	NO	Yes	Yes
Different Package Subclass	NO	NO	Yes	Yes
Different Package non-Subclass	NO	NO	NO	Yes.

- Inheritance → code reusability.
- Packages in java → Folders
  ↳ Hierarchy of programs.

```
class A
{
    int i, j;
    void showij()
    {
        System.out.println("value of i & j"
                + i + "  " + j);
    }
```

```
}
class B extends A
{
  int k;
  void showk()
  {
    System.out.println("value of k=" +k);
  }
  void sum()
  {
    System.out.println("Sum = " + (i+j+k));
  }
}
class Demo{
public static void main(String args[])
{
  A superob = new A();
  superob.i = 10;
  superob.j = 20;
  superob.showij();
  B subob = new B();
   subob.i = 5;
   subob.j = 2;
   subob.k = 10;
    subob.sum();   // output 17
}}
```



- class BoxWeight extends Box

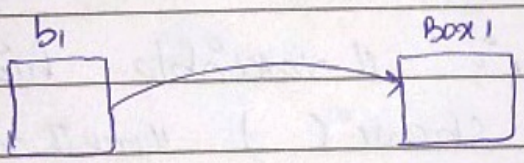
```
{
    double weight;
    BoxWeight(double l, double w, double h,
              double wt)
    {
        length = l;
        width = w;
        height = h;        } → super(l, w, h);
        weight = wt;
    }

    BoxWeight ( )
    {                          → super ( );
        length = width = height = weight = 1;
    }

    BoxWeight (double x, double wt )
    {
        length = width = height = x;  → Super(x);
        weight = wt;
    }

    BoxWeight (BoxWeight ob)
    {
        length = ob.length;
        height = ob.height;    } → super(ob)
        width = ob.width;
```

```
    weight = ob.weight;
  }
}
class demo{
public static void main(String args[])
{
  BoxWeight Box1 = new BoxWeight(10,20,30,40);
  BoxWeight Box2 = new BoxWeight(5,2,3,0);
/* Box b1 = new Box(10,20,30); */
/* b1 = Box1; */
// b1.vol  ← System.out.println("volume of box 1="+ Box1.vol());
//b1.weight ←System.out.println("weight of box 1 ="+ Box1.weight);
  System.out.println("volume of box2 ="+ Box2.vol());
  System.out.println("weight of box2 ="+ Box2.weight);
}
}
```



/*-

i) Note that it is the type of reference variable and not the type of object that it refers to, that determines what members can be accessed.
i.e. when a reference to a subclass object is assigned to a superclass reference variable, you will have access

only to those parts of object defined by superclass.

→ Super keyword

```
class A
{
  int i;
  void showi()
  {
    S.o.p("value of i="+i);
  }
}
class B extends A
{
  int i; // variable hiding
  void showi() // method hiding
  {
    super.showi();
    S.o.p("value of i in subclass="+i);
  }
  B(int y)
  {
    i=y;
  }
}
```

```
A(int x)
{
  i=x;
}
```

```
B(int y, int a)
{
  super(a);
  i=y;
}
```

__/__/____

```
class demo
{
  public static void main (String args[])
  {
    A obj1 = new A();
    A.i = 20;
    A.showi();
    B bi = new B(); → new B(10);
    bi.i = 10; // value of i in Subclass
    Si.showi();
  }
}
```

overriding → everything is same → name & args & Subclass is overriding Superclass method.
↳ overloading not same.

- Constructor chainings and method overriding can be resolved using Super keyword.

- Class A
{
  A()
  {
    S.O.P("Inside A's constructor");
  }
}
class B extends A

```
{
  B()
  {
    S.O.p("Inside B's constructor");
  }
}
class C extends B
{
   C()
   {
     S.O.p("Inside C's constructor");
   }
}
class demo
{
public static void main(String args[])
{
   C obj1 = new C();
} // when subclass constructor is called
} every superclass constructor will be
   called.
```

Superclass → A – Inside A's const.

↓

B – Inside B's Const.

↓

C – Inside C's const.

__/__/____

- Super is always 1st statement in constructor.
- In case of inheritance the constructors are called in order of their derivation.

- H/w → create a multiple hierarchy

Box
↓
weight
↓
Shipment

→
```
class A{
  void callme()
  {
    S.o.p("Inside A");
  }
}
class B extends A
{
  void callme()
  {
    S.o.p("Inside B");
  }
}
class C extends A{
  void callme(){
    S.o.p("Inside C");
  }
}
```

```
class demo
{
psvm(String args[])
{
  A a = new A();
  B b = new B();
  C c = new C();
  a.callme();
  b.callme();
  c.callme();
}
}
```

```
A r; // superclass
r = a;
r.callme();
r = b;
r.callme();
r = c;
r.callme();
```

→ Dynamic method dispatch.

- Determined at runtime which method will be called → Dynamic Method Dispatch.

= Method Overriding:

- Also known as runtime polymorphism.

Runtime polymorphism

one interface multiple forms.

eg: Figure

area → dimensions

Triangle circle square rectangle

__/__/____

```
- class figure                    //abstract class figure
{
  double dim1, double dim2;
  figure(double d1, double d2)
  {
    dim1 = d1;
    dim 2 = d2;
  }
  void area() // useless.        //abstract void area()
  {
    S.o.p ("Area not defined");
  }
}
class cube extends figure
{
  double height;
  cube(double length, double breadth, double h)
  {
    dim1 = length;
    dim2 = breadth;
    height = h;
  }
  void areea( )
  {
    S.o.p ("Area of cube is = "+(length * breadth *
                                 height));
  }
```

```
}
class rectangle extends figure
{
  rectangle(double a, double b)
  {
    super(a, b);
  }
  void area()
  {
    S.o.p("Area of rectangle = "+(dim1*dim2));
  }
}
class triangle extends figure
{
  triangle(double x, double y)
  {
    super(x,y);
  }
  void area()
  {
    S.o.p("Area of triangle = "+(dim1*dim2)/2);
  }
}
class figimplementation
{
```

__/__/____

```
public static void main(String args[])
{
    Figure f;
    rectangle r = new rectangle(10, 20);
    f = r;
    f.area();
    triangle t = new triangle(5, 7);
    f = t;
    f.area();
}
} // Runtime Polymorphism.
```

- Abstract area(); - its subclass defines it.
- Abstract class → if a class contains one or more than one abstract method then class is also declared abstact.
- No object for Abstract class.

- **Abstract:**

1. Any class that contains one or more abstact methods must also be declared as abstract.

2. An abstract class cannot be directly instenteated with a new operator. (i.e. object can't ~~by~~ be created but reference can be).

3. You cannot ~~as~~ declare abstract constructor or abstract static methods.

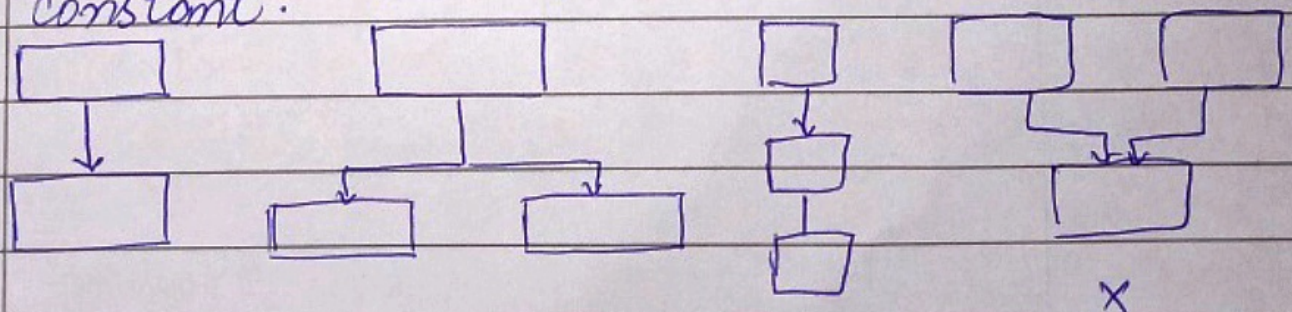
4. Any subclass of an abstract class must either implement all the abstract methods in the superclass or be declared abstract itself. (Although this is permissible).

```
- class A                    // Final class A
  {
Final ← void meth()          // preventing this method
  {                                from being overwritten.
    S.o.p("Inside Super class");
  }
}
class B extends A  // error
{
  void meth()      // error
  {
```

```
    S.o.p("Inside Subclass");
  }
}
class demo
{
  public static void main(String args[]);
  {
    A a;
    B b = new B();
    a = b;
    b.meth();
  }
}
```

Final

- Final keyword is used to prevent method over riding.
- Final keyword with class → used to prevent inheritance.
- Final → instance variable → becomes constant.



__/__/____

- Java doesn't support multiple inheritance.
- Java provides concept of interface.

- Interface:
  - It says what a class should do
    - Not how it is done.
- Interfaces are similar to classes but they lack instance variable & their methods donot have any body.
- One class can have any number of interfaces, each class that includes an interface must include the definitions for all methods in that interface.
- All methods and variables in the interface are implicitly public.

__/__/____

- Java doesn't support multiple inheritance.
- Java provides concept of interface.

- Interface:
  - It says what a class should do
    - Not how it is done.
- Interfaces are similar to classes but they lack instance variable & their methods donot have any body.
- One class can have any number of interfaces, each class that includes an interface must include the definitions for all methods in that interface.
- All methods and variables in the interface are implicitly public.

- Interface demo{
  final int x = 10;
  void callme(); // what a subclass should do
}

- The methods that implement an interface must be declared public.

```
- Interface demo
{
    // void show(); {...} // Possible to add
       void callme(int x)   more methods
}

class demonew implements demo
{
  public void callme(int a)
  {
    System.out.println("Inside Subclass value
                        = " + a);
  }
class main_demo
{
  public static void main (String args[])
  {
    demo d;
    demonew d1 = new demonew();
    d = d1;
    d.callme(10);
  }
}
```

__/__/____

- class A extends B implements C, D, E...

- We can at a time implement as well as extend.

- HLW - Stack - fixed
  ↳ Growable
  ] - same push pop operations.

★ Exception Handling:

- Error - program can't handle - eg. out of mem.
- Exception - Handle using program.
  ↳ Both are unwanted.

- Exception
  -It is an object in Java.

- Class demo
  {
    public static void main (String args[])
    {
      int a = 42;
      int b;
      b = a/0; // Exception division by 0
    }

__/__/__

- It creates an exception object & throws it
  ↓ thrown
  If there is no handler
  JRE → default handler
  ↓
  Prints stack trace.

- Arithmetic Exception : / by 0
  demo.main(demo.Java:7)
  ↳ Line number
  ↓
  Stack trace.

→ Class demo
{
  Static void meth()
  {
    int a=42;
    int b;
    b=a/0;
  }
  Public Static void main(String args[])
  {
    meth();
  }
}

__/__/____

Stack Trace

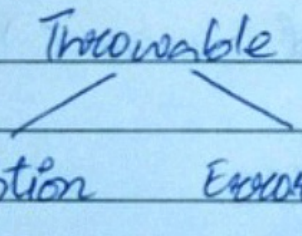
Arithmetic Exception: / by 0
at demo.meth (demo.java:3)
at demo.main (demo.java:7)

- Keywords used in Exception Handling:

1. try.
2. Catch
3. finally
4. Throw
5. throws.

Why Exception Handling:
- Stop Abnormal Termination
- Handle in our own way.

- Exception is an object.
  - It is subclass of Throwable class.



- Try:
  - that goes in it which raises some exception.
  - i.e., code to be monitored for exception.

__/__/____

- **Catch:**
→ It is handler for exception raised.

- **Finally:**
→ Code which will be executed in any scenario.

- class Demo
{
public static void main (String args[])
{
try{
int a = 42;
int b;
b = a/0; }
catch(ArithmeticException e)
{
System.out.println("Exception caught" + e);
}
System.out.println("outside catch");
}
}

Manual → Handler

NO stack trace will be printed.

→ If handler is found then it will execute

__/__/____

if not it will not execute
It will go to default handler.

- Throw:
Null Ptr exception
↳ To create exception & throw it manually

- Throws:
- When a method throws an exception object.
- But we are not handling in that method.
- But we are handling it where we call it.

- Division by 0 & array out of bounds error

```
class demo
{
public static void main (String args[])
{
  try {
     int a = 10;
     int b;
```

```
        b=a/0;
    }
    catch(ArithmeticException e)
    {
      System.out.println("Exception caught "+e);
    }
    try
    {
      int arr[5];
      int arr[] = {1,2,3,4,5};
      arr[6] = 10;
    }
    catch (ArrayOutOfBoundsException e)
    {
      System.out.println("exception caught"+e)
    }
    finally {
      System.out.println("Errors caught");
    }
  }
}
- Class demo
{
  psvm (String args[])
  {
```

```
try{
  int a = args.length;
  int b = 10/a; // No exception if args given.
  int c[] = new int[4];
  c[5] = 10;
}
catch (ArithmeticException e1)
{
  System.out.println(e1);
}
catch( ArrayIndexOutOfBoundsError e2)
{
  System.out.println(e2);
}
}
```

→ We can write multiple catch with one try.

→ Nested try:

```
try{
  int a = args.length;
  int b = 10/a;
  try{
    int c[] = new int[4];
    c[5] = 10; }
```

```
catch (ArrayIndexOutOfBoundsException e1)
{
    System.out.println(e1);
}
{
catch (ArithmeticException e2)
{
    System.out.println(e2);
}
```

- Manually Exception Handling:

- we can do manually exception handling using Throw Throwable instance.

eg:

ArrayOutOfBoundsException
new nullpointer Exception.

- It is possible for our program to throw an exception explicitly. using throw statement.

Throw Throwable instance

It should be an object of Throwable class or subclass.

- Primitive types such as int or char

__/__/____

as as well as non Throwable classes such as string cannot be used as exceptions.

```
class demo
{
  try{
      throw new NullPointerException;
  }
  catch (NullPointerException e)
  {
    S.o.p(" Exception Caught = " + e);
  } // throw e;
}
```

→ Manually creating an exception & throwing it.

```
// catch (NullPointerException e1)
// { S.o.p ("Recaught the exception=" + e1);
// }}.
```

- class demo{

```
{
static void meth() throw ArrayIndexOutOfBoundsException
```

```
{
throw new ArrayIndexOutOfBoundException;
}
public static void main(String args[])
{
  try{
    meth();
  }
  catch(ArrayIndexOutOfBoundsException e)
  {
    System.out.println(e);
  }
}
```

- Custom Exception:

```
class myexception extends Exception
{
  int a;
  myexception(int x)
  {
    a = x;
  }
  public string to string(){
    return("My exception[" + a + "]"); }
```

```
// Throws Myexception.
class demo{
  static void compute(int a)
{
  if (a > 20)
     throw new myexception(a);
  else
     S.o.p("value less than 20");
}
public static void main (String args[])
{
  try{
     compute(10);
     compute(20);
  }
  catch(myexception e)
  {
    S.o.p("Exception caught" +e);
  }
}
```

→ Testing method is an overriden method that display the value of the exception.

★ Chained Exceptions:

- The chained exceptions feature allows you to associate another exception with in exception.
- This second exception describes the cause of the is exception.

eg:

- A situation where a method throws an arithmetic exception because of an attempt to divide by zero.
However the actual cause of the problem was the I/O error that occured.

Throwable (Throwable Cause Exe)

- CauseExe is the underline reason that an exception occured.

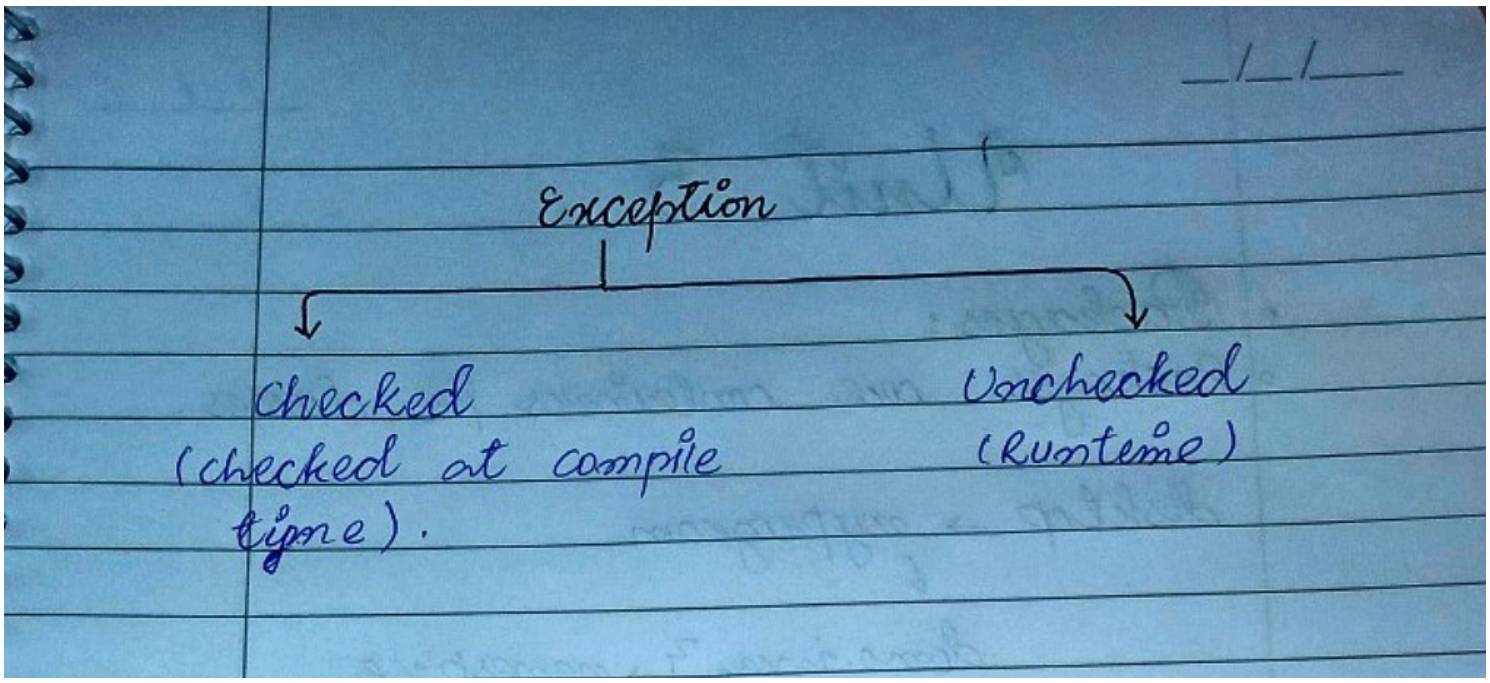
→ getCause
→ init Cause.

- The getcause method returns the exception that underlines the current exception.

__/__/____

- The initCause method associates an exception within invoking exception.

Eg:
```
class demo{
  static void meth()
  {
    NullPointerException e = new NullPointerEx-
                             ception("Top Layer");
    e.initCause(new ArithmeticException("Cause"));
    throw e;
  }
  public static void main(String args[])
  {
    try{
      meth();
    }
    Catch(" NullPointerException e)
    {
      System.out.println("Exception Caught"+e);
      System.out.println("Actual cause of exception"
                         +e.getCause());
    }
```


Exception
Checked
(checked at compile time).
Unchecked
(Runtime)