Page No.:
Date:

Java invented cuz of web browsers

James Gosling invented JAVA to incorporate 3 things
1) Dynamic object model
2) Portable binary distribution
3) Safe execution environment

1) dynamic object model refers to dynamic (runtime decision) can be taken.
For eg if two class are written in same java file then the ".class" file of required class will only be loaded ~~on~~ into main memory whenever the object of that class will be created.

Java runs on both x64 & a64 machine

2) Portable binary distribution means ~~is~~ supercomputer java ~~is~~ support multiplatform except "ios" & ^ In java ".class" file can run on windows, macos or linux if all these machine have JVM installed that's why Java ~~is~~ supports multiplatform. (it's not platform independent it is OS independant) user talk to (Kernel) OS talk to hardware

3) Safe execution environment
Java provides safe execution environment
eg. int list = {1, 2, 3, 4}
list[20] = 56 // this will give error in java instead of overwritting memory.

Java is not just a programming language it is software enviroment that lets you implement "write once run anywhere" concept.
Java includes implementation of following :-
1) JVM creates virtual environment so that we can run java code (.class file) on any machine (windows, linux, macos)

Java invented (12 of web browsers)

James Gosling invented JAVA to incorporate 3 things
1) Dynamic object model
2) Portable binary distribution
3) Safe execution environment

1) dynamic object model refers to dynamic (runtime decision) can be taken.
For eg if two class are written in same java file then the ".class" file of required class will only be loaded ~~on~~ into main memory whenever the object of that class will be created

Java runson both x64 & a64 machine

2) Portable binary distribution means java ~~is~~ supports multiplatform except "ios" & supercomputer. In java ".class" file can run on windows, macos or linux if all these machine have JVM installed that's why Java ~~is~~ supports multiplatform. (it's not platform independent & it is OS independant) user $\xrightarrow{\text{talk to}}$ (Kernel OS) $\xrightarrow{\text{talk to}}$ hardware

3) Safe execution environment
Java provides safe execution environment
eg int list = {1, 2, 3, 4}
list [20] = 56 // this will give error in java instead of overwritting memory.

Java is not just a programming language it is software enviroment that lets you implement "write once run anywhere" concept.
Java includes implementation of following :-
1) JVM creates virtual environment so that we can run java code (.class file) on any machine (windows, linux, macos)

.class file is like english acceptable in all machines provided they all have JVM.

Page No.:
Date:
YOUVA

".class" file contain metadata (machine readable discription) and byte code (machine neutral instructions)

2) Java class library
It is collection of packages like util, lang.

3) Java programming language
i) ~~C is procedural language (functional progra.~~
~~C++ is both procedural (functional.~~
i) C supports functional programming
ii) C++ supports both OOPs and functional programming
iii) Java is primarily OOPs you have to write everything in class itself.
iv) Java is more consistent in the terms of syntax than cpp (cpp have different ~~methods~~ way to write particular code while java provides only one way to write that particalar code)
v) In C++ you can have primitive data type as pointers (reference) like int *a.
but in java this is not allowed for all 8 primitive types you are not allowed to access there reference (address).
"Primitive type are value type"

In cpp you can refer to ~~object~~ class using both objects or pointers of that object
but java wipes out this confusion by saying that "user-define type are reference type"
only userdefine class can have references but primitive type cannot.

vi) loosely typed language
eg. javaScript
a = "Hello" // doesn't need data type to be written

Typed language
eg. c, c++
int a = 10.5 // data type needs to be mentioned

Strictly type language
eg. java
int a = 10 // LHS = RHS
int a = 10.5 // error

* Hello World
Java is primarily object oriented that's why everything in java should be written inside the class.
public static void main (String args[])
we write main method inside class and ~~we call that class~~ interpreter call that class when we give command as "java classname"
we are calling main() from some other class which is in interpreter that's why it should be public.
It is static cuz we are not creating any instance of that class. we are calling main() using classname directly
void is written cuz ~~we are~~ main() is not returning anything
(String args[]) are ~~parame~~ arguments which takes multiple parameters of string type (we can pass array of string to main() method)

When function is written inside the class it is called method
all methods are function but not all functions method

data is information which can be digitalized (smell cannot be data cuz it cannot be digitalized) if we can digitalize smell can be use in zomato swiggy to smell food before ordering

System.out.println()

1) System is class from java.lang package
System.out is for monitor
System.in is for keyboard

printstream is static member of System class.
2) out is instance of printstream
3) All instance of printstream class contain println()

```
Vi Hello.java
javac Hello.java
java Greeter
```

Hello.java has Greeter class

```
class Greeter {
    p.s.v.m(String args[])
    {
        S.O.p ("Hello World");
    }
```

We compiled this program using filename i.e javac Hello.java. Which created .class file Greeter.class and we run this program using "java Greeter" command.

which means class name & file name need not to be same because for running the program we need to know only classname (execute bytecode .class while)

~~but~~ java can only have one public class and file name should be ~~of~~ same as public class in java.

In java constructor definition is always there when class is created. constructor is called when object is created

class provide support for following mechanism
i) Activation :- creating object of class is called activation.
class is passive entity & object is active entity.

ii) Binding
Variables declared in class are called 'field'. ~~These~~ value of these fields are stored in memory is called 'instance'. and ~~when~~ 'object' refer to these memory block.
Objects are handler of instance.
we have class called Fan

Fan	
rotating, speed	← Fields
Switch()	← methods
Regulate()	

Fan has two objects hall & kitchen
When we need to ~~call~~ perform switch() on hall object we write it as
hall.switch(); but switch() is written in class and hall only has reference of instance (address of instance) ∴ how switch() is gonna get perform on hall object. that's when ~~a~~ binding comes into the picture. in data binding hall.switch() get converted into Fan.switch(hall);
which means switch() is in class Fan: we write Fan.switch() and object hall is being passed as parameter to switch method. ∴ it becomes Fan.switch(hall);

object is used to bind instance and methods and the process is called binding

$\underbrace{\text{Myclass m}}_{\text{Object}} = \underbrace{\text{new Myclass()}}_{\text{instance}}$

Similarly if we want to perform regulate() operation on 'kitchen' object we write

kitchen.regulate(4)

but internally java does binding and it becomes

Fan.regulate(kitchen, 4);

∴ Process of converting ~~object.class()~~ object.method() into class.method(object) is called binding.

System.out.printf(" Float value = %f %n", f.floatV);

printf() lets you format output like C

println() lets you print plain text.

* Wrapper class

Java cannot give methods into primitive types that's why java has wrapper class of all primitive types.

Wrapper class contains parsing methods which converts String to its corresponding data type. ~~parseInt~~ Integer.parseInt(), Double.parseDouble() etc.

* Narrowing & widening

~~Float can fit into double but double can't be fit into float~~ ~~Va~~

Float can fit into double is called widening.

Double can't be fit into float is called narrowing and that's why we have to use typecasting.

* Solid principles of Data Abstraction

1) Single Responsibility Principle (SRP)

Eg.1 We have class called Student having instance fields ~~id~~ Roll_no and CGPA now Some Students work partime like out of 100 Student, 20 students are salaried So you cannot make one more field "Salary" in student Class Since it is not applied to ~~Student~~ all students

Students
- int Roll-no
- Float CGPA

✓

Students
- int Roll-no
- Float CGPA
- Float Salary

✗ since only some students are salaried we can't ~~of~~ have Salary as instance field.

Eg 2. We have class called Employee having instance field 'id', 'noOfDays', ~~and~~ 'tax()', and instance method as 'calculateSalary()', but Since we are working with 10 different country's and have empolyees from 10 different countrys ~~calculation~~ implementation ~~over~~ ~~tax~~ ~~of~~ 'tax()' instance ~~field~~ method will be different for each country. ∴ here tax is not identical, its behaviour is not identical So we also can't have 'tax()' in Employee class as instance ~~field~~ method.

If you want to have tax for different country the create class for every country which will have tax as instance ~~field~~ method and its 'calculateSalary()' instance method will be same for tax.

If you want to have Salary in student create whole another class as SalariedStudent and then we can have Salary as instance field ~~there in Student~~ class.

Fields of class should only be manipulated using methods of that class and all fields should be private.
↓
This methods are getters & setters of class.

* Open Closed Principle (OCP)

Open closed principle says instance fields of class should not be accessed outside the class. So all fields should be declared as private.

and instance methods should be defined to manipulate instance fields. ~~ohi~~

~~Static methods are used when you don't need to change state of objects~~

Open closed principle says ~~that~~

Open for extension closed for modification

~~Only public declarations should be change~~

public declarations of ~~class~~ methods ~~&~~ should not be changed.

We can add fields, change defination of methods, change private declarations but cannot change or modify fields and public declaration of methods.

This rules are made so that our code maintainance will be easy cuz in industry 80% of efforts goes into maintainance ∴ if code follows this principle it will be more maintainable.

If we follow this ~~rules~~ principles of data abstraction then we have limitations. that's why OOP gives concept like method overloading, Inheritance and so on.

behaviour = methods
State = fields

* Enums

Enum is special data type that enables for a variable to be ~~written~~ set of predifined constants and they ~~must~~ should be in capital letters

eg.
```
enum Directions {
        NORTH, SOUTH, EAST, WEST;
}
```

* Variadic functions

If a function can take multiple arguments (variable number of arguments) for eg. average(), min(), max().

When we are dealing with variadic function we use "variable Argument" or "varargs" (...) it allows us to accept zero or multiple arguments. it is helpful if you don't know how many arguments you will have to pass to method.

1) It is always used at the end of ~~parameter~~ argument list.
2) It is iterated using for...each loop
3) and its length ~~is~~ can be taken using "length property

for eg.
```
public static double average(double First,
  double second, double...remaining) {
    double total = First + Second
    for (value : remaing) {
        total += value;
    }
    return total /(2 + remaining.length);
}
```

* Pass by value & Pass by reference

Function doesn't allows us to return multiple values but if we ~~wan't~~ want to have multiple values from function we can use class.

In java only primitive data types can be can be "pass by value" and only user defined types can be "pass by reference"

In pass by value copy of variable is being changed but in pass by reference directly address of variable is being passed so changes are done on original variable.

```
myFunc(double d) {
Sop("double" = %f", d); d = 20;
}

main() {
double d = 10;
MyFunc(d); Sop("dob = %f", d);
}
```

pass by value

o/p :- 10
10

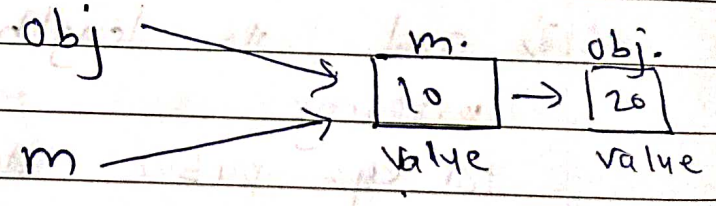
myFunc → d = [10] → [20]

main → d = [10]

```
my Func(Myclass obj) {
Sop("double" = %f, obj.value)
}

main() {
Myclass m = new Myclass();
m.value = 10
myFunc(m.)
Sop("double = %f", obj.value);
```

o/p :- 10
20



12/12

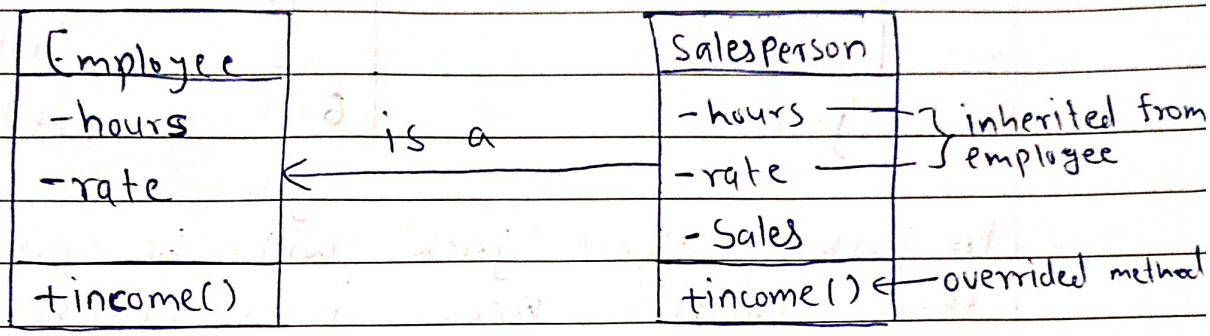
* Subtype - Polymorphism

Subtype :- Stands for Substitute type

it means "make Single function work with different objects".

If classes are sharing common behaviour (methods)(income) the polymorphism lets you implement a function (tax) which can operate on these classes

- private
+ public




```
income() {
// logic
}
```

```
income() {
super.income()
//additional logic
}
```

```
class program {
double tax(Employee emp){
double i = emp.income();
return 0.15 * i;
}
p.s.v.m() {
Employee jack = new Employee();
tax(jack); // When this goes to tax it is Employee emp = new Employee
Salesperson jill = new Salesperson();
tax(jill); // When this goes to tax it is Employee emp = ~~New Salesperson~~ new Salesperson();
}
}
```

This is called dispatch.

* In p.s.v.m() we are calling tax method by passing objects of employee as well as salesperson how compiler is resolving that at runtime?

→ When we use 'new' keyword to create object in memory apart from fields of class one extra block is created for that object and reference to that class is stored

eg.

Employee.class	reference
186	hours
52	rate

jack

Salesperson.class	reference
186	hours
52	rate
8800	sales

jill

For jack :- object "jack" will be passed to tax() there compiler knows it is of the type Employee then when it comes to JVM, JVM will check in its object (Jack) and there "Jack" have reference to class Employee means ~~tmp~~ income() of employee will be called.

For jill :- object "jill" will be passed to tax() there compiler knows it is of type Salesperson but since it is subclass of employee compiler will say ok go ahead then JVM will check object jill and there "jill" will have reference to class Salesperson means income() of Salesperson will be called. and since we are deciding which income should get called at runtime we call this machanism as "dynamic binding"

When we write tax() we followed liskov's principle here.

* In method overloading : Signature is same of both class (in subclass & superclass) but implementation is different.

* It is advisable to call constructor of superclass ~~advisa~~ explicitely in subclass's constructor even if compiler calls it automatically.

* Abstraction :-

If Someone asks to draw animal everyone will have their own idea of animal like cat, dog, tiger etc. So we can say animal is abstract idea just like a vehical or courage or love they are abstract.

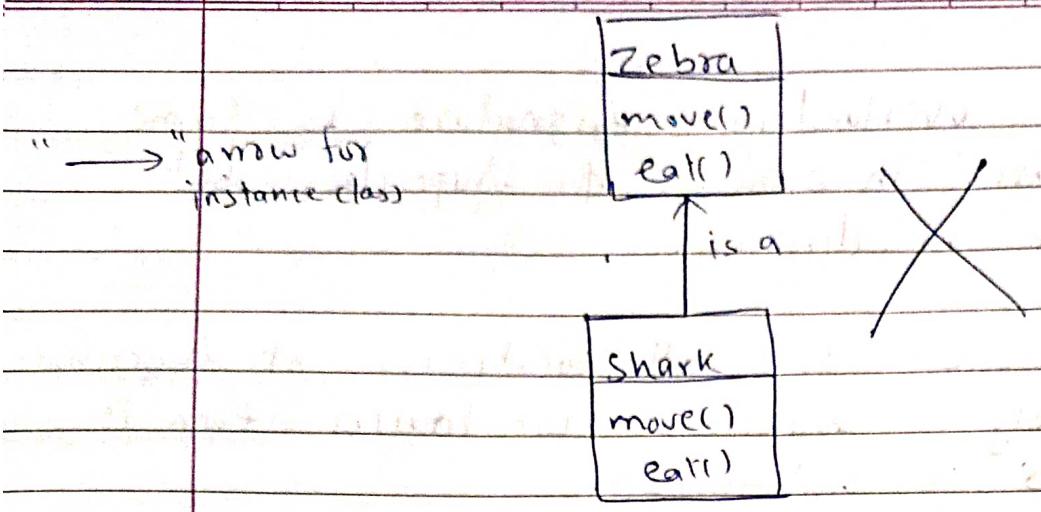
Since they are abstract you cannot draw them therefore you cannot create their object.

Now, every animal can move() and eat() but a cat might move ~~and~~ and eat differntly than a duck then here move() & eat() are pure methods. this pure ~~with~~ methods will be contained by abstract class or interface and they are meant to be inherited by subclass (cat, duck)

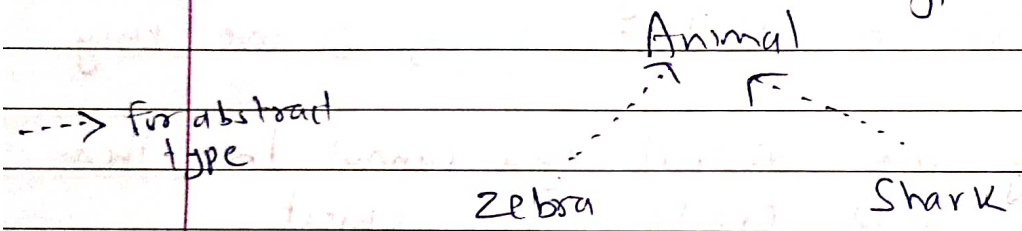
4) Interface seggregation principle

Suppose we have two class called zebra and shark and both have methods called move() and eat(). Since they share common behaviour but you cannot say that zebra is a shark and you cannot implement inheritance between them

Object is class in java.lang package which is mother (root) of every class. it has .toString() method which is generally overridden to add more meaning (functionality) to



instead of this since we know move() & eat() are pure methods and they are implemented by each animal we can create animal as an abstract type and make Zebra & Shark inherit this abstract type



5) Dependancy Inversion Principle
instances of abstract type should be activated using class which inherits this abstract type.

* Packages
If you want to have same class but in different context then create package to store class in differnt package

HR ← Package → Payroll
⇓ ⇓
employee ← Class → Employee