Class & Objects

Class is an blueprint for creating objects which shares common characteristics and common properties.

- ~~Class~~ No memory is ~~declared~~ allocated when class is declared.
- It is a logical entity.
- ~~It can be~~

Objects is an instance of class that are created to use the attributes and methods of a class. Objects are characterized by three essential properties: state, behaviour and identity.

```
class Student {  // class Declaration.

}
// Object creation:
Student S1 = new Student ( );
```

State of an object is a value from its data ty
The identity of an objects distinguishes from each othe
Its useful to think of an object's identity as the
place where its value is stored in memory.
The behaviour of an object is the effect of data-
type operations.

The dot operator links the name of an object with the name of an instance variable.
In java, it is referred as a Separator.

The 'new' keyword dynamically allocates (during runtime) memory for an object and returns reference to it. This reference is, more or less, the address in memory of the object allocated by new. This reference is than stored in variable.
Thus. In Java, all class objects must be dynamically allocated.

```
Box mybox; // declare reference to object
mybox = new Box(); //allocate a box object.
```

Here, mybox in 1st line doesnot yet refer to an actual object. The next line allocates an object and assigns a reference it to mybox and use as object. But in reality, myBox simply holds memory address of actual Box object.
The key to Java's safety is that you cannot manipulate references as you can actual pointers.

Thus, you cannot cause an object reference to point to an arbitary memory location or manipulate it like an integer.

"How for"
Why we don't use Integer or charAt?:
Ans: Because a primitive data type are not implemented as object.

Box b1 = new Box()
Box b2 = b1;
Here, b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of original object. It simply refer to same object as b1. Thus any changes made to object ; b2 will be affected. When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Note

Bus bus = new Bus();
L.H.S → is processed by compiler.
R.H.S → is looked by JVM.

Types of Constructor

1) Default Constructor.
2) Parametrized Constructor
3) Copy Constructor.

1) Default:

- If no constructor is defined in class, Java Complier automatically prodives default constructor and initializes the object with default values, such as 0 for numbers and null for objects. → Implicit Default Constructor.

- If we define constructor with no parameters, its called an. Explicit default constructor.

Date / /
Page

3) Copy constructor.

Unlike other constructor copy constructor is passed with another object which copies the data available from the passed object to newly created object.

Note: In java, there is no such inbuilt copy Constructor available like in other programming languages such as c++, instead we can create our own copy constructor by passing the object of same class to other instance (object of class).

```
class Geek { string name; int id;

    Geek (String name, int id)
    { this.name = name;
      this.id = id;
    }

Geek (Geek obj))
{   this.name = obj.name.
    this.id = obj.id

}

}

class GFG
{ psvm(S[]args)
{   Geek g1 = new Geek("Prem",42);
    Sout (g1.name + " " + g1.id); // Prem 42.

    Geek g2 = new geek(g1);
    Sout (g2.name + " " + g2.id);
                // Copy Constructor Prem 42
}
}
```

Date / /
Page

Final keyword : (More concepts in Polymorphism)

Final keyword is used to restrict user and indicate that a variable, method or class cannot be modified or extended.

- Variables :- Since it can't be modified later, it should be initialized while declaring. or in the constructor.
- Methods :- When a method is final, it can't be overridden by a subclass.
- Class :- When a class is final, it cannot be extended by a subclass.
- Security :- The final can help improve Security by preventing malicious code from modifying sensitive data or behaviour.

Garbage collection :-

It is an automated process to free up memory space that has been allocated to objects no longer needed by the program
Before destroying an object, the GC calls finalize() method to perform cleanup activities.

```
protected void finalize() throws Throwable { Super.finalize(); }
```

Note

- finalize method is called by GC not JVM.
- default implementation of finalize() is empty. So, overriding it is recommended for resource cleanups.
- finalize method is called only once per object.
- If an uncaught exception is thrown by finalize(), the exception is ignored, & finalization of that object

Date / /
Page 000PL

Package:

A package is used to group related classes. Think of it as a folder in directory. We use packages to avoid name conflicts & to write better maintainable code.

import Java.util.Scanner :-

→ Java.util is a package and Scanner is a class of that pack

Static

It is used for a constant variable or method that is same for every instance of a class.

Characteristics :-

a. static variables and methods are allocated memory space only once during execution of the program. This memory space is shared among all instance of the class. It is useful for maintaining global state or shared functionality.

b. static members can be accessed without the need to create an instance of class. Hence. main() is declared as static bcoz it must be called before any object exist.

c. Static members are associated with class, not objects. This means changes to static member are reflected in all objects and that you can access static members using class name rather than object referenc

Eg. class A { static int count=0; }.

main()
{ A obj = new A(); Sout(obj.count); → is not good)

d. Static methods and variables cannot access non-static members of a class, as they are not associated with any particular instance of class.

e. Can be overloaded, but not overridden:-
Static methods can be overloaded. However, they cannot be overridden, as they are associated with class rather than with a particular instance of the class.

```
# class Test
{ static void m1() //static method.
    { sout("from m1");
    }
  psvm()
  { m1(); //calling m1 without
    }       creating object.
}
```

When a method/member is declared static, it can be accessed before any objects of its classes are created

When we declare a static block, that gets executed exactly only once, when the class is first loaded.

```
class Test
{ static int a = m1()
    static { sout("Inside static");}
    static int m1() { sout(from m1);
                      return 20;}
  psvm()
  { sout(value of a + a)
    sout(value of b + b)
  }
}
```

Flow of program

1 → Static Block.
2
3
4.

O/P
from m1
Inside static block.
Value of a 0

Date / /
Page

4. Static method can be accessed by static method only because static method does not require instance.

```
Class Test
{ psvm(.)          Test obj=new Test();
  { greeting(); //Not Allowed
    fun();      //Allowed.
  }

Static void fun()
  { Sout("fun");
  } greeting(); → Not Allowed bcoz. greeting is a non static method.

void greeting()
  { Sout("Namaste");
  } fun();
}
```

belongs / It depends on object.

Allowed.

You can't access non static stuff without referencing their instances in static method.

we can't use 'this' word inside static methods because from static (we can't use non-static.

a) (Inner) class inside a class should be static because it is dependent upon other class.

b) Single class which is independent of any class, can't be static.

Eg.

Example:

```
Class Test {
        // (b) → It is independent. So, it can't
    } be static.

public class Prem {
  static class Marks {                    (a)
    String name;
    void set() {this.name = name;}

public static void main(String [] args)
  { Prem a = new Prem("Yadav")
    { sout(a.name);
}
```

These both classes are dependent upon Prem class. So, it should be static & It does not require any instance of class.

Date / /
Page

Singleton Class:

A singleton Class is a class that allows only one instance of itself to be created and provides a global point of access to that instance. This is achieved by making the constructor private, so no other instances of the class can be created. and providing a static method that returns single instance of class. It is used for eg: logging or configuration system. It ensures that there is only one instance of class. which can be accessed globally.

Example:

```
public Class Singleton {
    private static Singleton instance;
    private Singleton() // constructor
    {
    }
    public static Singleton getInstance()
    { if (instance == null) //checks if instance is created or not
        instance = new Singleton();
      return Instance;
    }
    psvm()
    {
    Singleton s1 = Singleton.getInstance();
    Singleton s2 = Singleton.getInstance();
    // Since all reference variable are pointing to thas same one object.
    So, S1 & S2 both will be equal.
}
```

Inheritence

Date / /
Page 0op3

It is a mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class.

We need inheritance for:

→ Code Reusability

→ Method overriding: It is available only through Inheritence. With this, Java achieves Runtime polymorphism

→ Abstraction: The concept of abstract where we do not have to provide all details, is achieved through inheritence. Abstraction only show functionality to user.

• "extends" keyword is used for inheritance.

When an object is created, during inheritence, only the object of subclass is created, not the superclass.

Explanation to points from next page @ & (b)

(a) boxWeight object is created here (in heap memory) and its reference is stored in box5 of type box class. (in stack memory). child class is referred by parent class. `box box5 = new boxWeight (1,2,3,4);`. Hence, its allowed in Java.

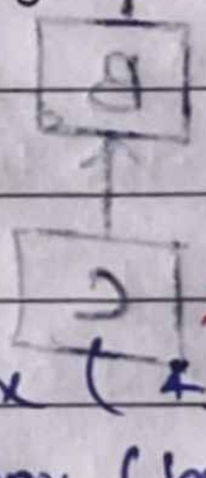
(b) // boxWeight box6 = new box (9,8,7);
There are many variables in both parent and child. You are given access to variables that are in reference type (boxWeight). This means the ones we are trying to access should be initialized but here object itself is of parent class. So, we can't call parent constructor because parent class has no idea about weight variable of child class.

Parent class

```
Public class box {          // Parent class
    double l,w,h;
    public box ()
    { this.l = -1; this.w = -1; this.h = -1
    }
    public box (double l, double w, double h)
    { this.l = l
      this.w = w
      this.h = h.
    }
    public box (box old) { this.l = old.l; & so on }
}
```

child class

```
public class boxWeight extends box {   // child class
    double weight; // Data Member
    public boxWeight () { this.weight = -1; } // Constructor
    public boxWeight (double l, double w, double h, double weight)
    { super(l, w, h);
      sout ("Inside child class");
      this.weight = weight;
    }
}
```



Driver program

```
public class Main {
psvm () {
    box box1 = new box (4.6, 3.2, 8.8);
    box box2 = new box (box1);
    boxWeight boxB = new boxWeight (2,4,6,8);   // l w h weight
    ⓐ box box5 = new boxWeight (1,2,3,4)
    ⓑ // boxWeight box6 = new box (9,8,7);
}
```

Date / /
Page

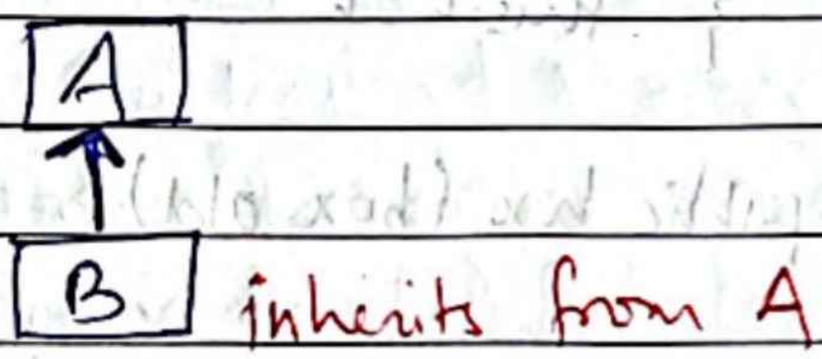
RANKA

Types of Inheritence

- follow GFG for examples.

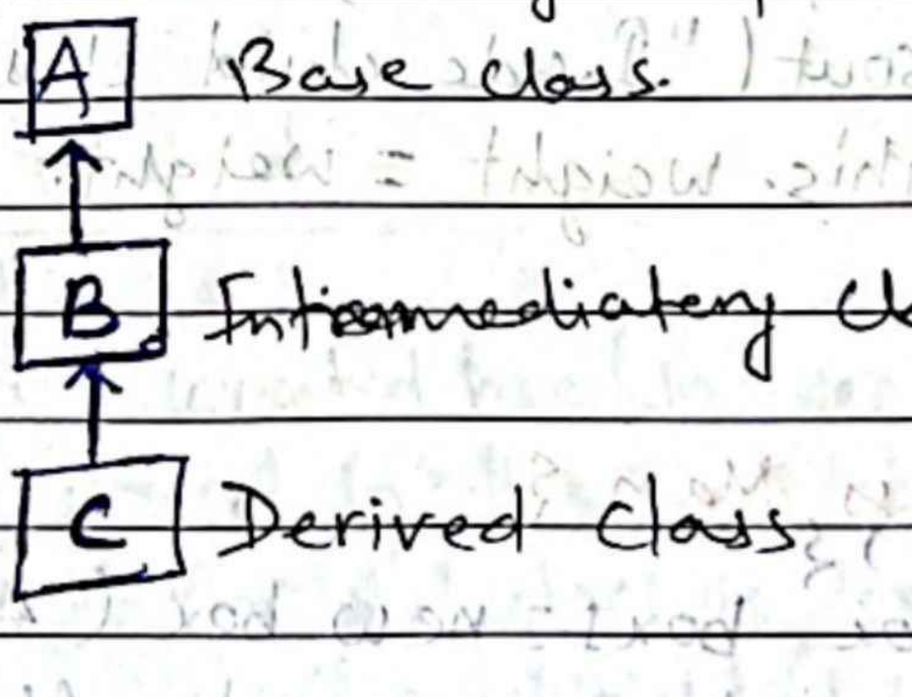
① Single level Inheritence.

Here a subclass is derived from only one super class. It inherits properties and behaviour of single parent class. Previous Eg. was an example of single inheritence.



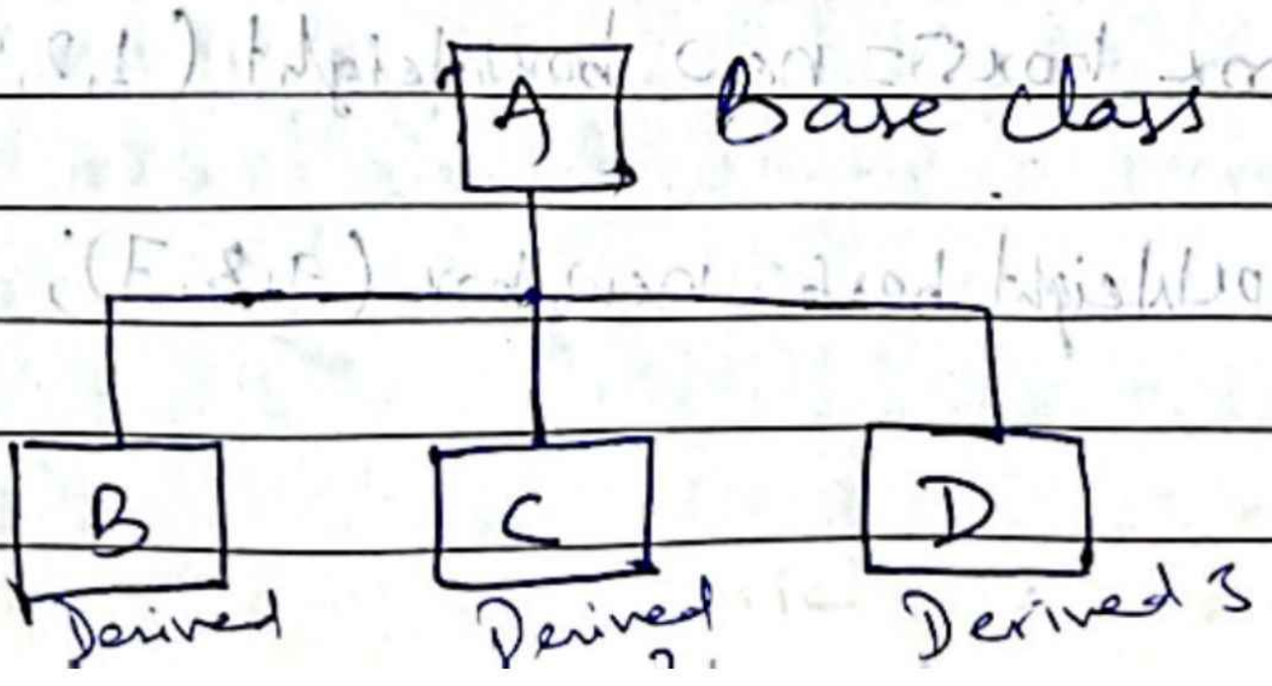
② Multilevel Inheritence

A derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In Java, a class cannot directly access the grandparent's members.



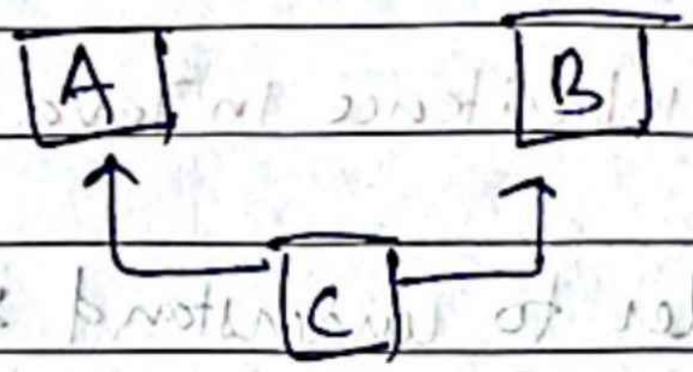
③ Hierarchial Inheritence.

Here, one class serves as super class for more than one subclass.



(4) Multiple Inheritence (Through Interface)

One class can have more than one superclass and inherit features from all parent class.
Java does not support multiple inheritance.
In Java, we can achive it only through interfaces.



Eg. interface one { public void print_geek(); }

interface two { public void print_for(); }

interface Three extends one, two {
public void print_geek();
}

class child implements Three
{ @override public void print_geek()
{ sout("Geeks");
}
public void print_for()
{ sout("for");
}

Pubic class Main()
{ psvm()
{
child c = new child();
c.print_geek();
c.print_for();
c.print_geek();
}
}

(5) Hybrid Inheritence

It is a mix of two or more types of inheritence.
In Java, we can achive it through interfaces.

Disadvantages of inheritence in Java.

a) Complexity: (harder to understand & more complex if inheritence hierarchy is deep & multiple inheritence is used:)

b) Tight Coupling: Inheritence creates a tight coupling blw superclass & subclass, making it difficult to make changes without affecting the subclass.

Conclusion

a) Default superclass
Except object class, which has no superclass, every class has one and only one direct superclass (single inheritence)
In absence of any other explicit superclass, every class is implictly a subclass of Object class.

b) Superclass can only be one.
A superclass can have any no. of subclasses.
but a subclass can have only one superclass because Java does not support multiple inheritences with classes.

c) Inheriting constructors.
A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructor are not members so they are not inherited by subclass, but constructor of superclass can be invoked from subclass.

d) Private member Inheritence:
A subclass doesnot inherit the private members of its parent class. However, if the superclass has public or protected methods (like getters & setters) for accessing its private fields, these can also be used by subclays.

(c) 'Super' keyword :-

Super(l, w, h);
- It calls parent class constructor and used to initialize value in parent class
- It should be used at first . No other operations are allowed before this.
- super.h will access value from parent class but,
- If a child class would have same variable 'h'. then it would be accessible as this.h.

↓

- Most common use is to eliminate the confusion b/w super class and subclasses that have methods with same name.

- If super() is not used in subclass constructor, then the default or parameterless constructor of each superclass will be executed.

Polymorphism

Polymorphism refers to the ability of a message to be displayed in more than one form. It allows objects to behave differently based on their specific class type.

Types of polymorphism

(1) Compile-Time Polymorphism.
It is also known as static polymorphism. It is achieved by function or operator overloading.

Method overloading in Java means there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by changes in the number of arguments or/and a change in arguments. Eg. int add (int a, int b) { }
double add (int a, int b, int c) { }

Subtypes of Compile time polymorphism
- Function overloading
- Operator Overloading (C++ only)
- Template (C++ only)

Date / /
Page

(2) Runtime Polymorphism:-

In Java, it is known as "Dynamic Method Dispatch."
It is a process in which a function call to the overridden method is resolved at runtime. It is achieved by method overriding.

On the other hand, method overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

```
class shape{
    void shape()
    {
        sout("Inside parent shape")
    }
    void area()
    {
        sout("Inside area in parent");
    }
}
```

```
void area(float radius
{
sout("Area: πr²");
}
```

```
class circle extends shape{
    void area()
    {
        sout("Inside area in child class");
    }
```

Date / /
Page

Advantages of polymorphism

→ Increases code reusability by allowing objects to be treated as objects of a common class.

→ Improves readability and maintainability of code by reducing the amount of code that needs to be written and maintained.

→ Supports dynamic binding, enabling correct method to be called at runtime, based on the actual class of the object.

→ Enables objects to be treated as a single type, making it easier to write generic code that can handle objects of different types.

Disadvantages of polymorphism

→ Can make it more difficult to understand behaviour of an object, especially if the code is complex.

→ This may lead to performance issues, as polymorphic behaviour may require additional computations at runtime.

Parent obj = new child();

Here, which method will be called depends upon
This is called upcasting

↓

overriding works

Early and Late Binding

```
Public class NewClass{
    public static class superclass{
    } static void print(){ sout("Inside superclass");
    public static class subclass extends superclass.
    { //@override
      static void print(){
          sout("Inside subclass");
    }
    psvm(){
    {  Superclass A= new superclass();
       Superclass B = new subclass();
    }
}
```

It is for late binding
Eg. It allows child method to override parent method.

a) **Early Binding**
The binding which can be resolved at compile time by complier. It is known as early binding.
Binding of all static, private and final methods is done at compile-ti

O/P: Inside superclass
Inside superclass

(if @override was not there in the code & print method should be static
else. child method will override parent

b) **Late Binding / Dynamic Binding :-**
In it the complier doesn't decide the method to be called.
Overriding is the best example for late binding.
In overriding, both parent and child class have same method.

O/P Inside superclass
Inside subclass

[Due to override, child method override parent method.
~~parent method~~ child method ↙

Note: ~~Static methods cannot be annotated with~~

Date / /
Page

RANKA

Final Keyword Uses

a) We use final to prevent overriding.

↳ To disallow a method from being overridden, specify final as a modifier at start of declaration.

Methods declared as final can sometimes provide performance enhancement: The complier is free to inline calls to them because it knows they will not be overridden by a subclass. When a small final method is called, often the Java complier can copy bytecode for subroutine directly inline with the compiled code of calling method. thus eliminating costly overhead associated with a method call.

Inlinig is an option only for final methods. Normally, Java resolves calls to method dynamically at runtime. This is called late binding. However since final methods can't be overridden, a call to one can be resolved at compile time. This is called early binding.

b) To prevent Inheritence

Preceeding the class declaration with final, we can prevent a class from being inherited.

Note:- Declaring a class as final implicitly declares all of its methods as final.

It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon subclasses to provide complete implementation.

c) Although static methods can be inherited. there is no point in overriding them in child classes because the method in parent class will always run no matter from which object you call it. That's why Static Interface methods cannot be inherited because method will run from parent interface and no matter if we are allowed to override them, they will always run the method in parent interface

Note:- Polymorphism does not apply to instance variables.

Note

Overriding deals with object. Static does not deals with object. So Thus, static can't be overridden

Encapsulation

Encapsulation is a principle that combines data and methods in a class. It allows implementation details to be hidden while exposing a public interface for interaction. Ex.

```
class programmer
{
    private string name;
getter ⟶ public string getName() { return name;}
setter ⟶ public void setName(String name){ this.name=name;}
}

public class Test
{ PSVM(){
    programmer p= new Programmer();
    p.setName("Prem");
    sout (p.getName());
}
```

Here, class restricts direct access to it from outside. This encapsulation mechanism protects internal state of the programmer object and allows for better control and flexibility in how name attribute is accessed/modifie

It is wrapping of data under single unit that binds together code and the data it manipulates.

Implementation.

1. In it, variable or data of a class are hidden from any other class and can be accessed only through any member function of its own class.

2. A private class can hide its members or methods from end user, using abstraction to hide implementation details, by combining data hiding and abstraction.

3. It can be achieved by declaring all variables in class as private and writing public methods in the class to set and get the values of variables.
4. It is more defined with getter and setter method.

Abstraction

It is the process of hiding implementation details and only showing the essential functionality or features to the user. This helps simplify the system by focusing on what an object does rather than how it does. Ex. Tv control remote

Tvremote - It simplifies the interaction with a Tv by hiding the complexity behind simple buttons and symbols. making it easy without needing to understant the technical details of how Tv functions.

```
abstract class Geek
{   abstract void turnOn();
    astract void turnOff();
}
class TVRemote extends Geek
{ @Override void turnOn()
                { sout ("Tv Turned on");}
  @override void turnoff()
                { sout ("Tv turned off");}
}

Public class Main
{ psvm(){
    Geek remote = new TvRemote();
    remote.turnOn();
    remote.turnoff();
  }
}
```

In Java, We can achieve abstraction by interfaces and abstract classes.

Abstract Classes & Abstract Methods :-

1. An abstract class is a class that is declared with an abstract keyword.
2. An abstract method is a method that is declared without implementation.
3. An abstract class may or may not have all abstract methods.
4. An abstract method must always be redefined in subclass, thus making overriding compulsory or making subclass itself abstract.
5. Any class that contain one or more abstract methods must also declared with an abstract keyword.
6. There can be no object of abstract class i.e. abstract class can't be directly instantiated with the new operator.
7. An abstract class can have parametrized constructor and default constructor is always present in an abstract class.

Algorithm to implement abstraction :-

1. Determine the class or interface that will be part of abstraction.
2. Create an abstract class that defines common behaviours and properties of these classes.
3. Define abstract methods inside abstract class that they do not have any implementation
4. Implement concrete class that extends to abstract class.
5. Override abstract methods in concrete classes to provide specific implementations.
6. Use concrete classes to contain program logic.

Access Modifier

Access Modifier helps to restrict the scope of class, constructor, variable, method or data member. It provides security, accessibility etc. to the user depending upon access modifier used with the element.

Types of Access Modifier:- Default, Public, Protected & Private

1) **Default**

When no access modifier is specified for a class, method, or data member, it is said to be having the default access modifier by default. The default access modifier are accessible only within the same package.

2) **Private:-**

The methods or data members declared as private are accessible only within the class they are declared.

- Any other class of same package will not be able to access these members
- Top-level classes or interfaces can not be declared as private because:-
  - private means "Only visible within enclosing class".
  - protected means "only visible within enclosing class and any subclasses".

3) **Protected**

Methods or data members declared as protected are accessible within the same package or subclasses in different package

Only subclass can access protected members in base class. when it in different package.

4) Public Access Modifier.

It has widest scope.
Classes, methods or data members that are declared as public are accessible from everywhere in the program.

Comparison Table of Access Modifier

	Class	Package	Subclass (same pkg)	Subclass (dif. pkg)	World (diff pkg. not subclass)
public	+	+	+	+	+
Protected	+	+	+	+	
Default	+	+	+		
Private	+				

Inbuilt Packages

Packages are used to avoid naming conflict and to control the access of class, interface, subclasses etc.

Some inbuilt packages are:-

1. Sql → provides classes for accessing & processing data stored in a database.

2. lang → Contains classes and fundamental to the design of Java. Classes like String, System, Math, etc

3. util → Contains collections framework, properties, random no. generation classes like Arraylist, linkedlis, Hashmap Calendar, Date.

4. net → provides classes for implementing networking apps. like Authenticator, HTTPCookie, Socket, URL, etc.

5. io → provides classes for Input/output operation. like BufferReader, file, outputstream, Serializable etc.

6. awt → contains classes for creating user interfaces for painting graphics & images. Classes like button, color, event, font, Graphics etc.

Date / /
Page

Interfaces

It is an abstract type used to specify the behaviour of a class. It contain static constants and abstract method.

- It is a mechanism to achieve abstraction.
- By default, variables in an interface are public, static and final.
- It is used to achieve abstraction and multiple inheritence.
- It is also used to achieve loose coupling.
- Interfaces primarily define methods and other classes must implement.
- It represents the Is-A relationship.

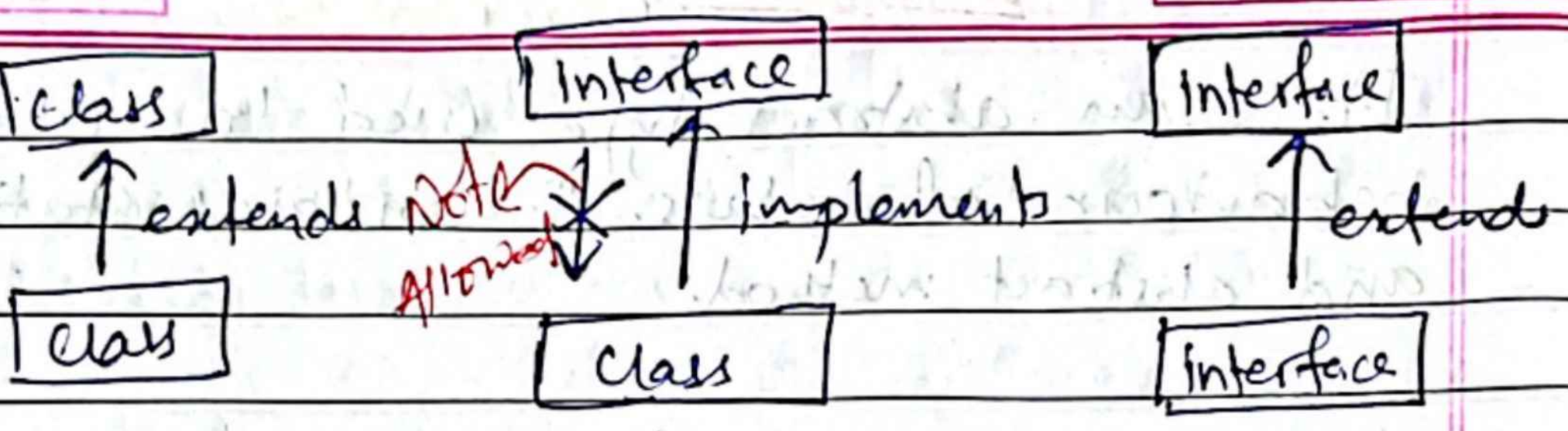
Note:

In Java, abstract keyword applies only to classes and methods, indicating they can't be instantiated directly and must be implemented.

When we decide on a type of entity by its behaviour and not via attribute we should define it as an interface.

Using, interface, we can get total abstraction. That means all methods in an interface with an empty body and are public and all fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement interface, use 'implement' keyword.

A class can implement an interface, but (reverse) interface implementing a class is not allowed.



Class	Interface :
1. You can instantiate variable and create objects.	→ You must initialize variables as they are final but you can't create an object.
2. It can contain concrete (with implementation) methods.	→ It can't contain concrete (with implementation) methods.
3. Access modifiers used are public, private and protected.	→ Only one specifier is used :- public

New features Added in JDK 8

1. Default Method.

Interfaces can define methods with default implementation. Useful for adding new methods to interfaces without breaking existing implementation.

2. Static method.

Interfaces can now include static methods. These methods are directly called using interface name and are not inherited by implementing classes.

Date / /
Page

3. Extending Interfaces

One interface can inherit another by use of keyword extends. When a class implements an interface that inherits from another interface it must provide an implementation for all methods by the interface inheritance chain.

Eg.

```
interface A { void method1(); void method2(); }
Interface B extends A
{ void method3();}

Class GFG implements B
{
   void method1() { sout("M1"); }
   void method2() { sout("M2"); }
   void method3() { sout("M3"); }
}
Psvm()
{  GFG x = new GFG();
   x.Method1();
   x.Method2();
   x.Method3();
}
```

Advantages of Interfaces

1) Without bothering about the implementation part, we can achieve the security of the implementation

2) In Java, multiple inheritence are not allowed, however you can use an interface to make use of it as you can implement more than one interface.

Date / /
Page

RANKA

New Features Added in interface in Java 9.

→ From Java 9 onwards, interfaces can also contain static methods, private methods and private static methods

Important points:

1. We can't create an instance (interface can't be instantiated) of the interface but we can make the reference of it that refers to object of its implementing class.

2. A class can implement more than one interface.

3. An interface can extend to another interface or interface (more than one interface)

4. A class that implements the interface must implement all the methods in the interface.

5. All the methods are public and abstract. And all the fields are public, static and final.

6. It is used to achieve multiple inheritence.

7. It is also used to achieve loose coupling.

Inside the

8. Interface not possible to declare instance variables because by default variable are public static final

9. Inside interface main method & constructors are not allowed.

10. Inside interface static, final & private method declarations are not possible.

Date / /
Page

Tagged Interface are interfaces without any methods they serve as a marker without any capabilities

Annotations in Java

1. It starts with @..
2. Annotations do not change the action of compiled program.
3. It helps to associate metadata to the program elements i.e. instance variables, constructors, methods, classes etc.
4. Annotations are not pure comments as they can change the way a program is treated by the complier.
5. Annotations basically are used to provide additional information, so ~~should~~ could be an alternative to XML and Java marker interfaces

Java.lang.annotation.Annotation

(Built-in) Standard Annotations	Custom Annotations
General purpose Annotations	**Meta Annotations**
@Override	@Inherited
@Deprecated	@Documented
@SafeVarArgs	@Target
@SupressWarnings	@Retention
@FunctionalInterface	@Repeatable.

Note This program throws Compliar errors because we have mentioned one.

Categories:-

1) Marker Annotations.
The only purpose is to mark a declaration. These annotations contain no members and do not consist of any data. Thus, its presence as an annotation is sufficient. Since, the marker interface contain no members, simply determining whether it is present or absent is sufficient. @ Ex. @Override.
Ex- @TestAnnotation()

2) Single Value Annotations.
These annotations contain only one member and allow a shorthand form of specifying the value of the member. We only need to specify the value for that member when the annotation is applied and don't need to specify the name of the member. However, in order to use this shorthand, the name of member must be a value.
Ex- @TestAnnotation("Testing");

3) Full Annotations
These Annotations consists of multiple data members, names, values etc
Ex- @TestAnnotation(owner="Rahul", value="GFG"

4) Type Annotations:-
These annotations can be applied to any place where a type is being used. For eg. we can annotate the return type of a method. These are declared annotated with @Target annotation:

5) Repeating Annotation:-
It can be applied to a single item more than once. For an annotation to be repeatable, it must be annotated with @Repeatable annotation. Its value field specifies the container type for repeatable annotation. The container is specified as an annotation whose value field is an array of the repeatable annotation type.

Hence, to create a repeatable annotation, firstly the container annotation is created, and then annotation type is specified as an argument to the @Repeatable annotation.

Pre-Defined / Stantard Annotation

1) • 4 are imported from java.lang.annotation
:- @Retention, @Documented, @Target, @Inherited

2) 3 are included in Java.lang :-
@Deprecated, @Override, @SupressWarnings.

Generics

Generic means parametrized types. It allows type like Integer, stringe etc. or user defined types to be a parameter to methods, classes and interfaces. Using Generics, it is possible to creat classes that work with different data types. An entity such as a class, interface or method that operates on a parametrized type is a generic entity.

Why Generics?

Ans The object is the superclass of all other classes. and Object reference can refer to any object. These features lakk type safety. Generics add that type of safety feature.
Ex- Hashset, Arraylist, Hashmap etc

Types of Java Generics

1. Generic Method.
A generic method has type parameters that are cited by actual type. ~~parameter section. There can be more than one type of parameter, separated by a comma.~~ The complier takes care of type safety which enables programmers to code easily since they do not have to perform long, individual type casting.

2. Generic classes
A generic class is implemented like non-generic class. difference is that it contains type parameter section. There can be more than one parameter, separated by comma.

To create an instance of generic class,

BaseType <Type> obj = new BaseType <Type> ();

Note:- In parameter, we can't use primitive data types like int, char or double.

But, primitive type arrays can be passed to type parameter because arrays are reference type.

Error ← Test<int> obj = new Test<int>(20);

ArrayList<int[]> al = new ArrayList<>();

Benefits

1) Code Reuse :-

2) Type Safety:

```
psvm()
{
    Arraylist al = new Arraylist();
    al.add("Prem");
    al.add(20);
    String s3 = (String) al.get(1);
}
```

Compiler allows this

This gives error in general Arraylist without any specified type.

So, we use,

Arraylist <String> al = new Arraylist <String> ();

al.add(20);

This will give error.

3) Individual Type casting is not needed.
Refer to (2) example, there S3 assigned (String) al.get(1) → Here it is typecasted.

So, if we already know our list holds only String value, we do not need to type cast.

4) Generics promote code reusability.
For eg. We want to sort array elements of various data types like int, char, etc for this, we can create a single common function for diffent data types using Generics.

5) Implementing Generic Algorithms
By using generics, we can implement algorithm that work on different type of objects, and at the same time, they are type-safe too.

extends Comparable
Wild card generics Interfaces

Lambda Expressions

Lambda Expressions represents instances of functional interface with a single abstract method. They provide concise way to express instances of single method interfaces using a block of code.
Functionalities.

a) Functional Interfaces : It is an interface that contains only one abstract method.
b) Code as Data : Treat functionality as method argument
c) Class Independence :- Create functions without defining a class.
e) Pass and Execute : Pass lambda expressions as objects and execute on demand.

Structure :-

```
(int arg1, String arg2) -> System.out("Two arguments" +
{                                      arg1 + " " + arg2);}
```

Argument List — Arrow Token — Body of Lambda Expression.

Exception Handling

It allows developers to manage runtime errors effectively by using mechanism like try-catch block, finally block, throwing Exceptions, custom Exception Handling etc.

An exception is an unwanted or unexpected event that occurs during the execution of a program i.e. at runtime) and disrupts the normal flow of program's instructions.
Ex- accessing invalid index, divide by zero. or trying to open file that dont exist.

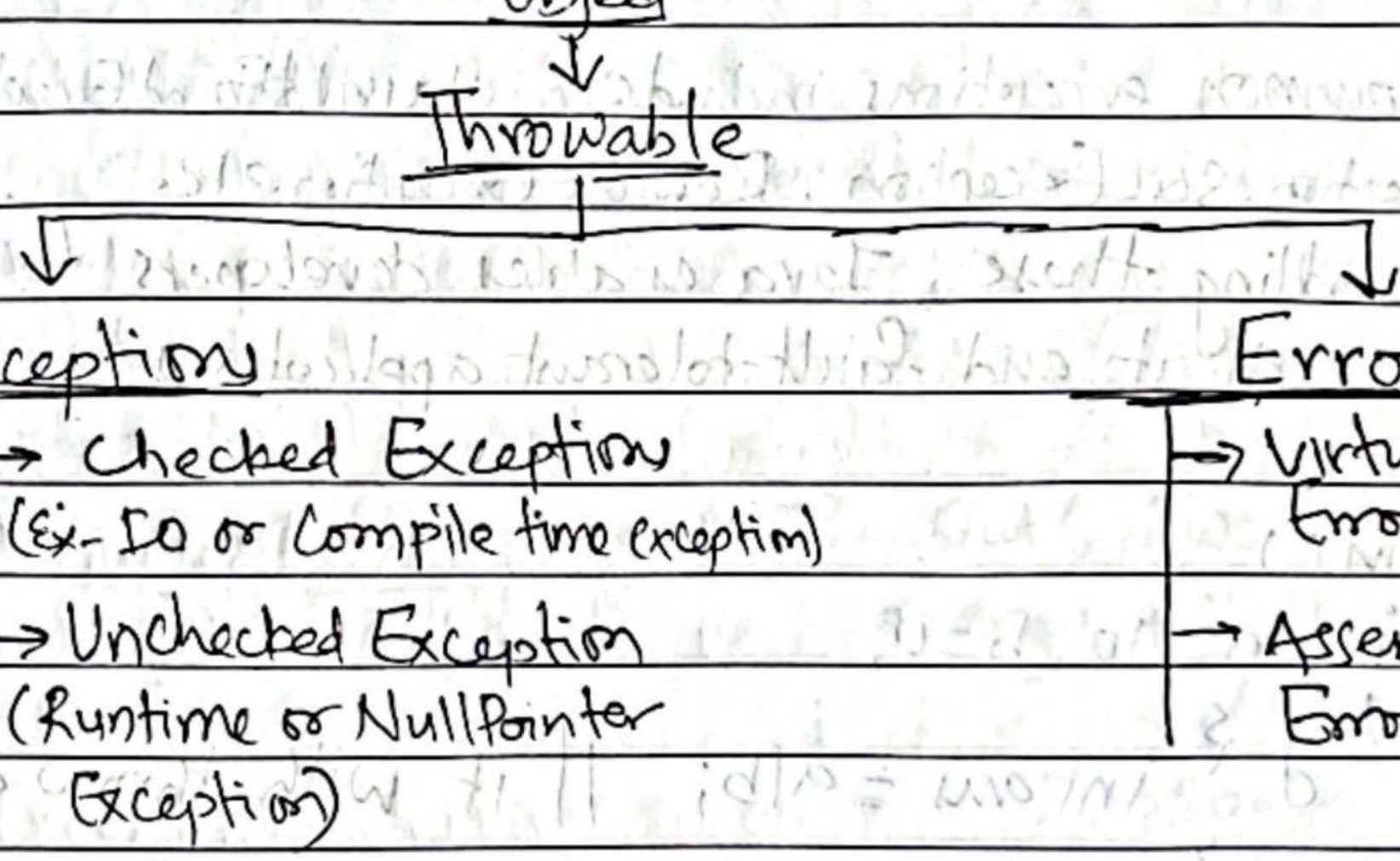
Other common exceptions include: ClassNotFoundException, IOException, SQLException, RemoteException etc.
By handling these, Java enables developers to create robust and fault-tolerant applications.

Eg.

```
psvm()
{ int a=10, b=0;
  try {
      int ans = a/b; // it with throw exception.
      }
  Catch(ArithmeticException e)
  { sout("Can't divide by zero");
  }  or, sout(e.getMessage());
finally
  { sout("Program continues after handling exception");
  }
```

Java Exception Hierarchy

All exceptions and error types are subclasses of class Throwable, which is the base class of hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user program should catch. NullPointerException is an example of such exception.
Error is used by JVM to indicate errors having to do with the run-time environment itself (JRE). StackOverflowError is example of it.



Exceptions can occur due to several reasons

- → Invalid User Input
- → Device Failure
- → Loss of Network Connection
- → Physical Limitations (out of memory)
- → Code Errors
- → Out of Bounds
- → Null Reference
- → Type mismatch
- → Database & Arithmetic error

	Error	Exception
→	It Indicates serious problem that a reasonable application should not try to catch.	→ It indicates condition that a reasonable application might try to catch.
→	Caused by issues with JVM or hardware.	→ Caused by condition in the program such as invalid input or logic errors
→	Out of Memory Error StackOverflow Error	→ IO Exception NullPointer Exception.

	throw	throws
1. Point of Usage.	It is used inside function when it is required to throw an Exception logically.	It is used in function signature when function has some statement that can lead to exceptions.
2. Exceptions Thrown	It is used to throw an exception explicitly. It can throw only one exception at a time.	It can be used to declare multiple exceptions, separated by a comma. Whichever exception occurs, if match with declared ones is thrown automatically.
3. Syntax	'throw' keyword includes instance of exception to be thrown. Syntax wise throw keyword is followed by instance variable.	'throws' keyword includes the class name of Exception to be thrown. Syntax wise 'throws' keyword is followed by Exception class name.
4. Propagation of exceptions	throw keyword is used to propagate unchecked exceptions that are not checked using throws keyword.	throws keyword is used to propagate the checked Exceptions only.

Date / /
Page

RANKA

Types

① **Builtin Exception**
These are predefined exception classes provided by Java to handle common errors during program execution.

1.1) **Checked Exceptions:-**
These are called compile-time exceptions because these exceptions are checked at compile-time by complier.

a) **ClassNotFound Exception :-**
Throws when program tries to load a class at runtime but class in not found because its not present at current location or it is missing from projeet.

b) **InterruptedException:**
throws when a thread is paused or another threads interrupts it.

c) **IOException:-** throws when 'input/output operation fails.

d) **Instantiation Exception**
throw when program tries to create a object but fails because class is abstract, an interface or has no default constructor.

e) **SQLExceptim:** throws when there's an error with database

f) **fileNotfound Exceptim.**
throws when programs tries to open a file that does not exist.

Date / /
Page

1.2) Unchecked Exceptions:-

Its opposite to checked exceptions. The complier will not check these exceptions at compile-time. Simply, if a program throws an unchecked exceptim and even if we didn't handle or declare it, the program will not give compilation error. Eg.

a) ArithmeticException :- thrown when there is illegal math operation. Eg. divide by zero.

b) ClassCastExceptim:- throw when you try to cast an object to a class it does not belongs to.

c) NullPointerException:- throws when you try to use null object. (eg accessing its methods or fields).

d) ArrayIndexOutOfBoundsException:- invalid throws when we try to access array element with index

e) ArrayStoreException:- When you store an object of wrong type in an array.

f) IllegalThreadStateException:-
thrown when a thread operation is not allowed in its current state.

Date / /
Page

② User-Defined Exception

Methods to print Exception information.

i. printStackTrace(.)
↳ prints full stack trace of exception incl. name, message and location of error.

ii. toString( )
↳ prints exception information in format of Name of the exception

iii.) getMessage ( ) → prints description of exception.

a) Try-Catch Block

```
try {
    // code that may throw exception
}
catch ( ExceptionType e) {
    // code to handle exception.
}
```

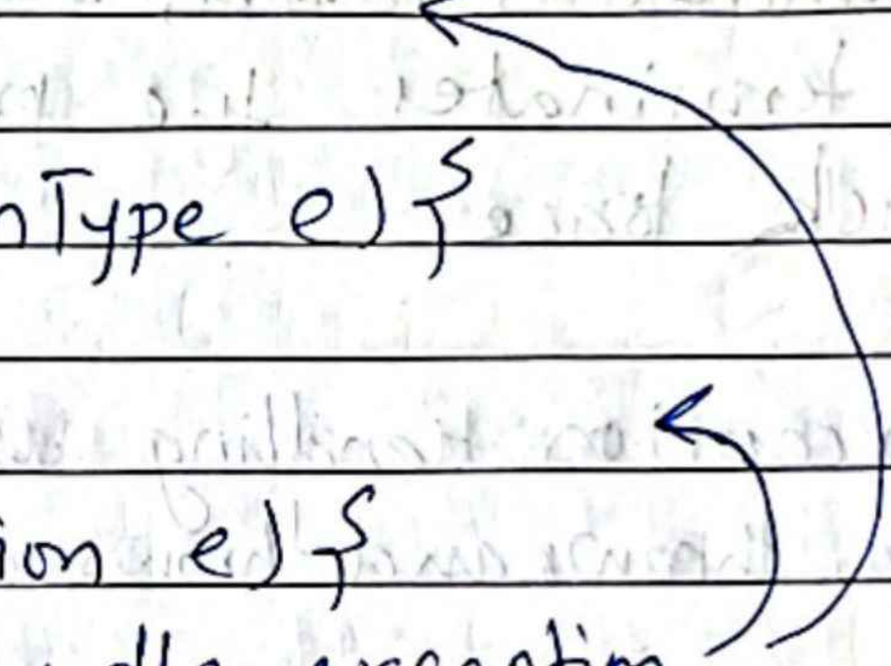
b) finally block.
↳ finally block is used to execute important code regardless of whether an exception occurs or not. It always executes after try-catch block. - It is also known as resource cleanup.

```
code of a +

finally {
    // cleanup Code.
}
```

c) Handling Multiple Exception.

```
try {
    // code that may throw exception
}
Catch (ExceptionType e) {
    //
}
catch (ExceptionType e) {
    //
}
catch (Exception e) {
    // code to handle exceptim
}
finally {
    // clean up code
}
```



How JVM handle Exception :-

Default Exception Handling :- When an Exception occurs, JVM creates an exception object containing exception name, description and program state. Creating Exception object and handling it in the run-time system is called throwing an exception. There might be a list of method that had been called to get to the method where an exceptim occured. The ordered list of methods is called call Stack.

a) The run-time system searches call stack for exception handler.

b) It starts searching from the method where exception occured and proceeds backward through call stack.

c) If a handler is found, exception is passed to it.

d) If no handler is found, default exception handler terminates the program and prints the stack trace.

Customized Exception Handling

Java Exception Handling uses five keywords :- try, catch, throws and throws and finally.

We can throw exceptions manually with throw. and methods must declare exceptions they can throw using throws.

Advantages of Exception Handling

a) Provision to complete program Execution.
b) Easy Identification of program code and Error-Handling code.
c) Propagation of Errors
d) Meaningful Error Reporting
e) Identifying Error Types.

Object Cloning

Object cloning in Java refers to creating an exact copy of an object. The clone() method in Java is used to clone an object. It creates a new instance of the class of current object and initializes all its field with exactly the contents of corresponding fields of this object.

```
class Human implements Cloneable
{   int age; String name;
    public Human(int age, String name)
    {   this.age = age;
        this.name = name;
    }

    public Human (Human other)
    {   this.age = other.age;
        this.name = other.name;
    }

    public Object clone() throws CloneNotSupportedException
    {   return super.clone(); }
}
public class objectCloning
{   psvm()
    {   Human prem = new Human(32, "Prem Yadav");
        Human another = new Human(prem);
        // It takes a lot of time, so we will use
        Object Cleaning method known as Cloneable.

        Human twin = (Human) prem.clone();
        sout(twin.age);
    }   sout(twin.name);

}
```

Date / /
Page

RANKA

Advantages of clone() method.

1) If we use assignment operator to assign an object reference to another reference variable then it will point to the same address location of old object and no new copy of object will be created. Due to this, any changes in the reference variable will be reflected in original object.

2) If we use a copy constructor, then we have to copy all the data over explicitly i.e we have to reassign all the fields of the class in the constructor explicitly. But in the clone method, this work of creating a new copy is done by the method itself. So to avoid extra processing we use object cloning.

Note: We can use the Assignment operator to create a copy of the reference variable. This creates a shallow copy, means both variables will refer to the same object in memory. This is not the same as cloning, where a new object is created.

Collections

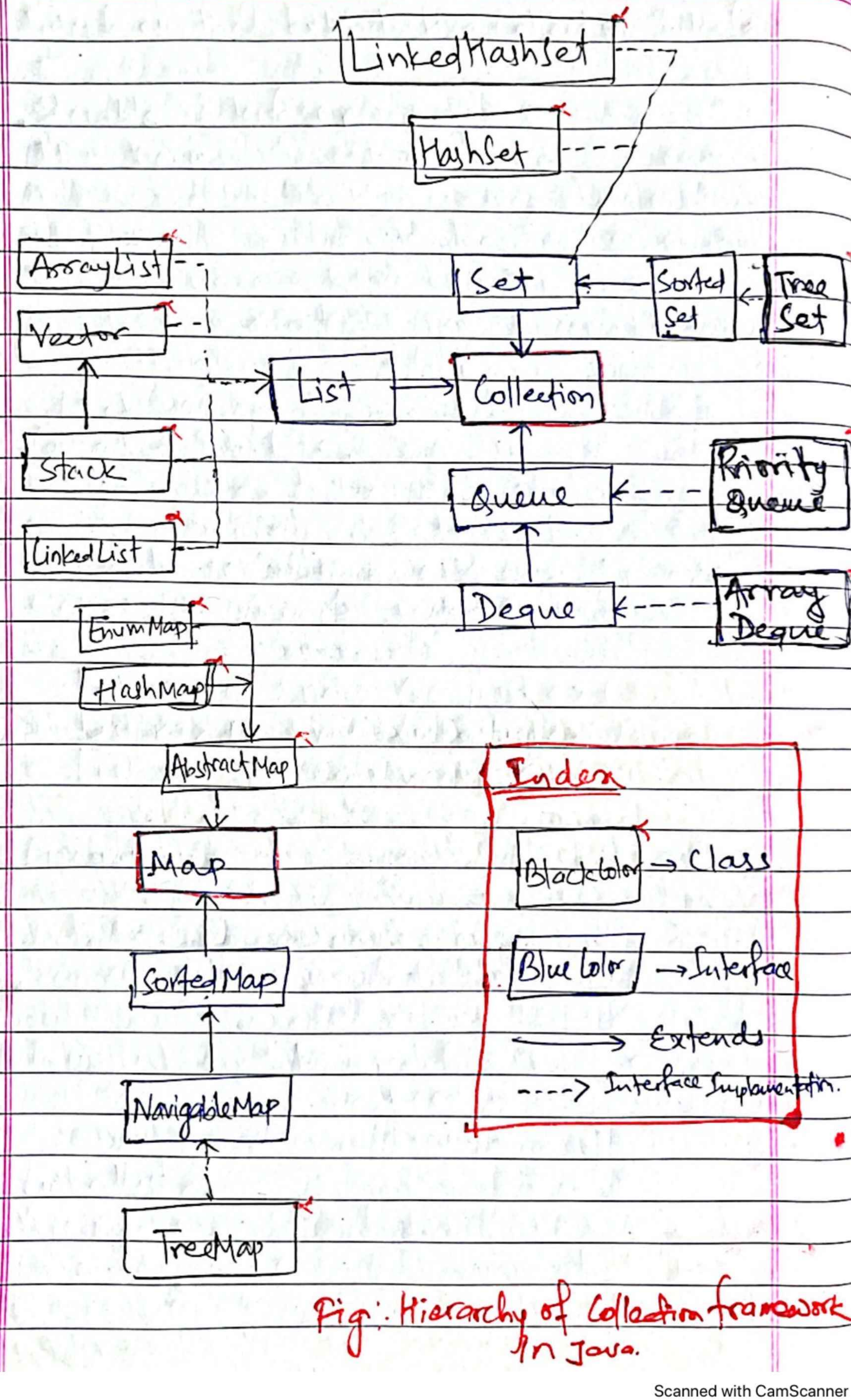


Fig: Hierarchy of Collection framework in Java.

Date / /
Page

RANKA

Methods of Collection interface.

1) add (object)
2) addAll (Collection c) → adds all elements in the Collection c
3) clear ( ) → Remove all elements
4) contains (Object o) → returns true if element is present.
5) containsAll (Collection c) → return true if the collection contains all of the elements in the given collection.
6) equals (object o) →
7) hashCode ( ) → returns hash code value for this Collection
8) isEmpty ( ) .
9) iterator ( ) → returns an iterator over elements in this collection
10) max ( ) → return max. value present in the collection.
11) parallelStream ( ) → .
12) remove (object o) → removes first occurance.
13) removeAll (collection c) → remove all elements of mentioned in given collection from the collection.
14) removeIf (predicate filter)
15) retainAll (Collection c) → retain only the elements in the collection that are contained in specified collection.
16) size ( )
17) Spliterator ( ) → Creates a Spliterator over elements in this collection.
18) Stream ( ) → returns a sequential stream with this collection as its source.
19) toArray ( ) → return an array containing all elements in this collection.

Date / /
Page

RANKA

Interfaces that Extend Java Collection Interface

① *Iterable Interface*

It is the root for entire collection framework. The collection extends the iterable interface. The main function is to provide iterator for the collections. Thus, this interface contains only one abstract method which is the iterator.

```
Iterator iterator();
```

② ~~Collection~~ List *Collection Interface*

It is dedicated to the data of list type in which we can store all ordered collection of objects. The list interface is implemented by various classes like ArrayList, Vector, Stack etc.

Eg.

```
List <T> al = new ArrayList <> ();
List <T> li = new LinkedList <> ();
List <T> v = new vector <> ();
```

a) ArrayList.

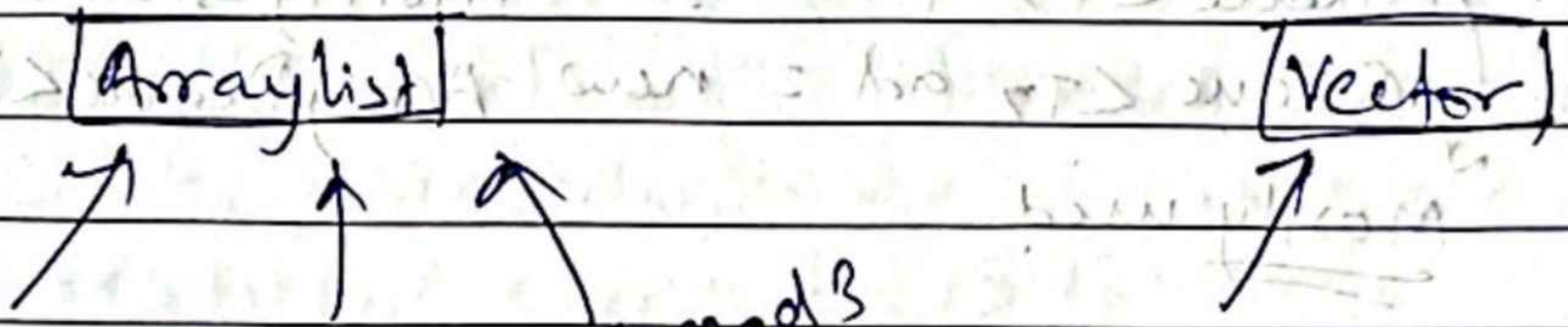
It provides us dynamic arrays in Java. It is slower but helpful in programs where lot of manipulations in the array is needed. The size of ArrayList automatically increases if collection grows & also shrinks. ArrayList can't be used for primitive types like int, char etc. So, we need to use wrapper class.

b) LinkedList

It is an implementation of LinkedList Data structure where elements are stored in contiguous locations and every element is separate object. The elements are linked using pointer and addresses. Each element is called node.

c) Vector

A vector provides us dynamic arrays. It is slower. It is similar for Arraylist in terms of implementation However, primary difference between a vector and ArrayList is that a vector is synchronized and Arraylist is not-synchronized.



d) Stack

Stack class models and implements stack data structure. The major operations are:- push, pop, empty, search and peek.

(3) Collection Interface

This interface extends Iterable Interface. It contains all basic methods like adding data, removing, clearing etc. All these methods are implemented in this Interface because these methods are implemented by all the classes irrespective of their style of implementation. In short, this interface builds a foundation on which classes are implemented.

Date / /
Page

RANKA

④ Queue Interface

This interface maintains FIFO (first in first out) order. There are various classes like PriorityQueue, ArrayDeque etc. Since, all classes implement queue, we can instantiate a queue object with any of these classes.

Eg. Queue<T> pq = new PriorityQueue<>();
Queue<T> ad = new ArrayDeque<>();

Mostly used

a) PriorityQueue

It is used when objects are supposed to be processed based on priority. It follows FIFO, but sometimes elements of queue are needed to be processed according to priority and this class is used in these cases.

It is based on priority heap. The elements of priority queue are ordered according to the natural ordering, or by a comparator provided at queue construction time, depending upon which constructor is used.

⑤ Deque Interface

also known as double ended queue, a data structure where we can add and remove data from both ends of the queue. This interface extends queue interface. It is implemented by ArrayDeque.

a) ArrayDeque

It provides us with a way to apply resizable array. It have no capacity restriction and they grow as necessary to support usage.

The basic methods are clear(), arr.addFirst(), arr.addLast();

⑥ Stack

Note: Stack is a subclass of vector and legacy class. It is thread-safe which might be overhead in an environment where thread safety is not needed. An alternate to stack is to use ArrayDeque which is not thread-safe and has faster array implem.

⑥ *Set Interface*

It is an unordered collection of objects in which duplicate values can't be stored. It is implemented by various classes like HashSet, TreeSet, LinkedHashSet etc. Eg.

```
Set<T> hs = new HashSet<>();
Set<T> lhs = new LinkedHashSet();
Set<T> ts = new TreeSet();
```

a) HashSet

The objects that we insert into HashSet do not guranteed to be inserted in same order. The objects are inserted based on Hashcode. It also allows insertion of NULL elements.

Eg: main()

```
{ Hashset<String> hs = new Hashset<String>();
hs.add("Hello"); hs.add("How"); hs.add("are");

Iterator<String> itr = hs.iterator();
while(itr.hasNext())
{
  sout(itr.next());
}
```

o/p

are
hello
How.

b) LinkedHashSet.

Similar to HashSet. Difference is that this uses doubly linked list to store data and retains ordering of elements.

⑦ Sorted Set Interface

Difference is that this interface has extra methods that maintain ordering of elements. It is implemented by TreeSet.

Eg. SortedSet <T> ts = new Treeset<>();

a) Treeset

TreeSet class uses a Tree for storage. The ordering of elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. It can also be ordered by a comparator provided at set creation, depending upon which constructor is used.

⑧ Map Interface

Map is a data structure that supports key-value pair for mapping the data. This interface does not support duplicate keys because same key can't have multiple mappings, however it allows duplicate values in different keys.

Map Interface is implemented by HashMap, TreeMap etc. Eg.

Map <T> hm = new Hashmap <>();

Map <T> tm = new Treemap <>();

Date / /
Page

a) HashMap

To access a value in a HashMap, we need to know its key. It uses a technique called Hashing. Hashing is a technique of converting large String to small String that represents the same String so that the indexing and search operations are faster. It also uses HashMap internally.

Eg. psvm()

```
{
    Hashmap<Integer, String> hm = new Hashmap
                                  <Integer, String>();
    hm.put(1, "Prem");
    hm.put(2, "Kumar");
    hm.put(3, "Yadav");
    Sout(hm.get(1);

    for (Map.Entry<Integer, String> e : hm.entryset())
    Sout(e.getkey + "  " + e.getValue());
```