OBJECT ORIENTED PROGRAMMING

- Classes and Objects:

A class is a template for an object, and an object is an instance of a class.

When you declare an object of a class you are creating an instance of a class.

A class creates a new data type that can be used to create objects.

- Properties of Objects:

Objects are characterized by three essential properties: state, identity and behaviour.

The state of an object is a value from its data type

The identity of an object distinguishes one object from another, often represented by the place where stored in memory.

The behaviour of an object is the effect of data-type operations.

- Creating and allocating objects

The 'new' keyword dynamically allocates memory for an object and returns a reference to it.

The dot operator links the name of the object with the name of an instance variable.

- Declaring and Allocating objects in Java.

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

The reference is stored in the variable, and all class objects in Java must be dynamically allocated.

Java safety lies in the inability to manipulate reference like pointers.

classname class-var = new classname();
class-var is the variable of the class type and classname is the name of the class being initiated.
Java's primitive types are not implemented as objects for efficency reasons
The 'new' keyword allocates memory for an object during runtime.

NOTE:
Bus bus = new Bus();
LHS (reference i.e bus) is looked by compiler and RHS (object i.e new Bus()) is looked by JVM

- 'this' keyword
  Used inside any method to refer to the current object
  Always a reference to object on which method was invoked.

- 'final' keyword
  A field can be declared as final, preventing its contents from being modified
  Requires initializing a final field when declared.
  Common convention - Use all uppercase identifiers for final fields.
  eg:
  final int FILE_OPEN = 2;
  Guarantees immutability only for instance variables of primitive types, not reference types

- 'finalize()' method
  Allows performing code actions when an object to be reclaimed by the garbage collector

Define finalization actions by implementing finalize()
method.

```
protected void finalize(){
    // finalize code over here.
}
```

- Constructors
  Automatically called when an object is created, before the new operator completes
  No return type, not even void
  Implicit return type is the class type itself.

```
Box mybox1 = new Box();
// new Box() is calling the Box() constructor.
```

- Inheritance and constructors
  Constructor of the base class with no argument gets automatically called in the derived class constructor
  Default constructor present in any class, whether declared or not.
  If a derived class doesn't have a default constructor, JVM invokes its default constructor and calls the super class constructors
  If the class has parameterized constructor, the derived class constructor should explicitly call the parameterized super class constructor.

```
class Base {
    Base() {
        System.out.println("Base class constructor called");
    }
}
```

```
class Derived extends Base{
    Derived(){
        System.out.println("Derived class constructor
        called");
    }
}

public class Main{
    public static void main(String[]args){
        Derived d = new Derived();
    }
}
```

Output-

Base class constructor called

Derived class constructor called.

- Packages: Organization and Visibility

Packages are containingers for classes, prouding compart-
-mentalization for class names.

They prevent naming collisions and keep namespaces organized.

Packages are stored in hierarchial mannen and are explicitly imported into new class definitions.

- Package as Naming and Visibility control Mechanism

The package serves as both a naming and visibility control mechanism

Example: package MyPackage creates a package named MyPackage.

- Package Storage

  Java uses file system directories to store packages
  class files for classes in a package must be stored in a directory matching the package name.
  Case sensitivity is crucial, the directory name must match the package name.

- Reflecting Package Hierarchy in File System

  A package hierarchy in code must be mirored on the file system.
  Example: A package declared as 'package java.awt.image;' needs to be stored in java\awt\image in a Windows environment.

- Determining Package locations

  The Java run time system determines package locations in three ways

  Default : Uses the current working directory as the starting point
  CLASSPATH : Specifying directory paths by setting the CLASSPATH environmental variable.
  -classpath : Using the -classpath option with java and javac to specify class paths.

- Static

  When a member is declared static, it can be accessed before any objects of its class are created and without reference to any object
  Static members include both methods and variables
  Example: 'main()' method is declared as static because it must be called before any objects exist
  A static method in Java belongs to class not to object

- Static method characteristics

A static method can access only static data; it cannot access non static data (instance variable

Non static method belongs to static instance, and in an static context, you need to explicitly mention an object reference to access them.

A static method can call only static methods and cannot call a non-static method directly.

- Static Block Initialization

Static blocks can be used for computations to initialize static variables

A static block is executed exactly once when the class is first loaded.

Example:

```
class UseStatic {
    static int a = 3;
    static int b;
    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
    static {
        System.out.println("Static block initialized");
        b = a * 4;
    }
    public static void main(String[] args) {
        meth(42);
    }
}
```

Output:

Static block initialized

x = 42

a = 3

b = 12

NOTES:

Static methods cannot refers to 'this' or 'super' keyword.

Static inner classes can have static variables.

You can't override inherited static methods. Overriding occurs at runtime and static methods are resolved during compile time.

Static interface methods are not inherited by implementing classes or sub-interfaces.

- Singleton class

A singleton class is designed to allow only one instance of itself and provides a global point to access this instance.

This is achieved by making the constructor private, preventing the creation of additional instances. Instead a static method is provided to return the single instance of the class

- Implementing a Singleton class

1. Declare a private static variable to hold single instance of the class
2. Make the constructor private to prevent the creation of additional instances.
3. Provide a public static method to return the single instance of the class, creating if necessary.

- Example 1: Eager Initialization.

```
public class Singleton {
    public static final Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton getInstance() {
        return instance;
    }
}
```

- Example 2: Lazy Initialization

```
public class Singleton {
    public static Singleton instance;
    private Singleton() { }
    public static synchronized Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return singleinstance;
    }
}
```

- Inheritance

Inheritance is an mechanism in which one object aquires all the properties and behaviours of a parent object.
You inherit a class by incorporating its definition into another using 'extends' keyword.

```
class Subclass extends Superclass {

}
```

You can only specify only one superclass by any subclass, and multiple inheritance is not supported.

A subclass includes all members of its superclass but cannot access private members of the superclass.

- Superclass variable and subclass object

When an reference variable to a subclass object is assigned to a superclass reference variable, you can only access the members defined in the superclass.

```
Superclass ref = new subclass();
```

Here ref can only access methods available in Superclass.

- super keyword

'super' is used when a subclass needs to refer to its immediate superclass

'super' has two forms: calling the superclass constructor and accessing a hidden superclass member.

Example

```
class BoxWeight extends Box1
    double weight;
    BoxWeight(double w, double h, double d, double m){
        super(w,h,d); // call superclass constructor
        weight = m;
    }
}
```

Here, super(w,h,d) calls the constructor of the superclass Box

Using super to access hidden member:

```
super.member;
```

Here, 'member' can be method or an instance variable.

It is used when subclass members hide members by the samename in superclass

M T W T F S S

Page No.

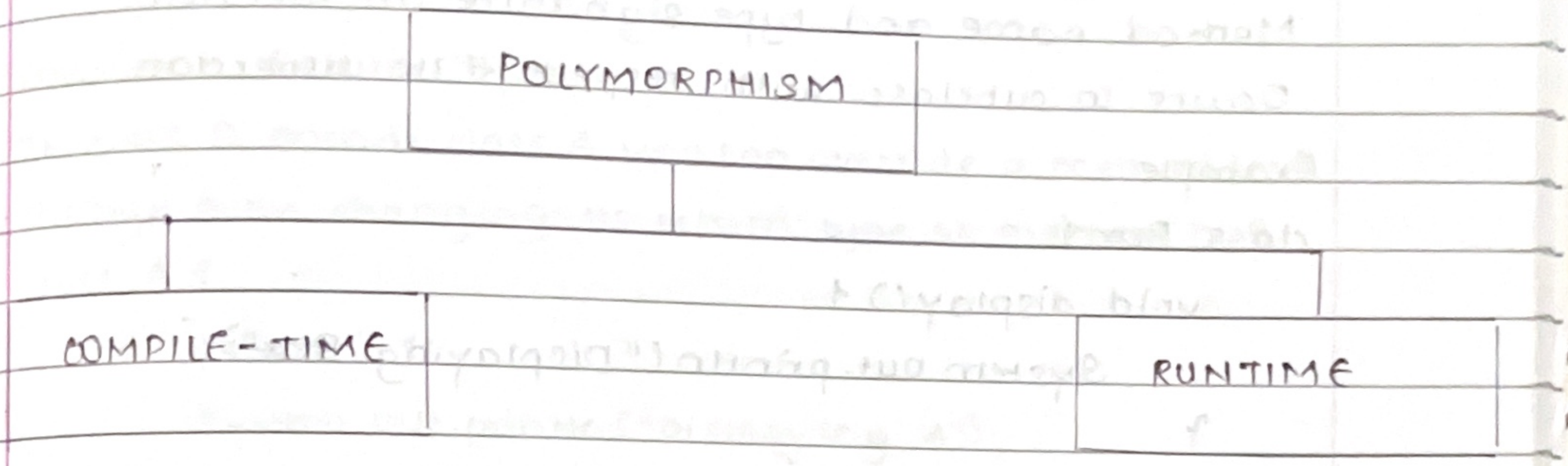
Date:

YOUVA

1. Polymorphism

Polymorphism in Java is a concept by which we can perform a single action in different ways.

Polymorphism is derived from 2 greek words: poly means many and morphs means forms. So polymorphism means many forms.



Method Overloading

Method overloading allows defining multiple methods in the same class with the same name but with different parameter declarations.

Java decides which method to execute based on parameter used in the class. Automatic type conversion may be employed to resolve method calls.

```
class OverloadDemo {
    void test(double a) {
        System.out.println(a);
    }
}

class Overload {
    public static void main(String[]args) {
        OverloadDemo od = new OverloadDemo();
        int i = 88;
        od.test(i); //Invokes test(double) with automatic
        od.test(123.2); type conversion.
    }
}
```

Method Overriding

In a class hierarchy when a method in the subclass has the same name and type signature as a method in its superclass, the method in the subclass is said to override the method in the superclass.

Overriding occurs when:

Method name and type signature are identical.

Occurs in subclass with respect to its superclass.

Example

```
class Box {
    void display() {
        System.out.println("Displaying Box");
    }
}
class SubBox extends Box {
    //Method overriding
    void display() {
        System.out.println("Displaying SubBox");
    }
}
```

Here SubBox overrides the display() method of its superclass Box

Dynamic Method Dispatch

It is a mechanism by which a call to overriden method is resolved at runtime, not compile time.

Key Points:

- A superclass reference variable can refer to a subclass object
- The determination of which version of an overriden method to execute is based on the type of object being refered at runtime.

Example:

```
Box myBox = new SubBox();
myBox.display(); // Calls the overriden method in SubBox
```

Here, myBox is of type Box, but it refers to an object of type SubBox. The defined display method called will be the one defined in SubBox because its determined at runtime.

Covariant Return Type.

If class B extends class A you can override a method in A through B by changing its return type of method to B.

```
class A {
    A display() {
        System.out.println("Displaying A");
        return this;
    }
}

class B extends A {
    B display() {
        System.out.println("Displaying B");
        return this;
    }
}
```

Here, the return type of the display method in B is B which is a subtype of the return type of the display method in A.

Encapsulation

Encapsulation in Java is a process of wrapping code and data together in a single unit

We can create a fully encapsulated class in Java by making all data members of class private. Now we can use setters and getters to set and get data.

Example:

```
package com.example;
public class Student
    // private data member
    private String name;
    // getter method for name
    public String getName(){
        return name;
    }
    // setter method for name.
    public void setName(String name){
        this.name = name;
    }
}
```

- Abstraction

Abstraction is the process of hiding the implementation details and showing only functionality to the user. Abstraction lets you focus on what the object does instead of how it does it.

There are two ways to achieve abstraction.

1. Abstract class
2. Interface

- Abstract class

A class which is declared as abstract is known as abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated. It can have final methods which will force sub class not to change body of the method.

Example:

Parent.java.

```
public abstract class Parent{
    abstract void career(String name);
}
```

Son.java

```
public class Son extends Parent{
    @Override
    void career(String name){
        System.out.println("The career is :" + name);
    }
}
```

Main.java

```
public class Main{
    public static void main(String[]args){
        S.career("Doctor");
    }
}
```

Output:

The career is : Doctor

- Interface

An interface in Java is a blueprint of a class. It has static constants and abstract methods

There can be only abstract methods in Java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.

Example:

Engiene.java.

```
public interface Engine {
        void start();
        void acc();
        void stop();
}
```

Car.java

```
public class Car implements Engine {
    @Override
    public void start(){
        System.out.println("Start");
    }
    @Override
    public void acc(){
        System.out.println("Accelerate");
    }
    @Override
    public void stop(){
        System.out.println("stop");
    }
}
```

- Access control

Access to ~~control~~ class members is determined by access modifiers, including public, private, protected and default. These modifiers control accessibility within its class, its subclasses, the same package and the wider world.

	CLASS	PACKAGE	SAME PKG SUBCLASS	OTHER PKG SUBCLASS	WORLD DIFF PKG AND NOT SUB CLASS
PUBLIC	✓	✓	✓	✓	✓
PROTECTED	✓	✓	✓	✓	
DEFAULT	✓	✓	✓		
PRIVATE	✓				

- Public : Accessible everywhere
- Protected: Accessible within the class, its subclasses and the same package
- Default : Accessible within the same package only
- Private : Accessible only within the class

- Object class

The Object class is the parent class of all the classes in java, it is topmost class of java.

Methods of object class

public final Class getClass()

returns the class object of the object. The class class can further be used to get metadata of this class

public int hashCode()

returns the hashcode number of this object

public boolean equals (Object obj)

compares the given object to this object

public String toString()

returns the string representation of this object

- Annotations

Annotations is a tag that represents the metadata i.e attached with class, interface, methods or field to indicate some additional information which can be used in java compiler and JVM.

Built-in Annotations

@Override

Checks that the method is an override. Causes a compilation error if the method is not found in one of the parent classes or implemented interfaces.

@Deprecated

Marks the method as obsolete. causes a compile warning if the method is used.

@SuppressWarnings

Instructs the compiler to supress the compile time warnings specified in the annotation parameters

@SafeVarags

Indicates that the annoted method or constructor is safe for use with varargs arguments

@FunctionalInterface

Indicates that the annoted annoted interface is functional interface

@Retention

Indicates how long an annotation is shared in compiled class file

@Documented

Indicates that the annotated element should be documented

@Target

Indicates where an annotation can be used.

@Inherited

Indicates that an annotation should be inherited by subclasses.

Custom Annotation

Java custom Annotation also known as user defined annotations are easy to create and use. The @interface element is used to declare an annotation.

```
@interface MyNotation { }
```

Points to remember for Java custom annotation signature.

- Method should not have any throw clauses.
- Method should return of of the following types: String, class, enum or an array of these types
- Method should not have any parameters
- @ should be attached just before the interface keyword to define the annotation.
- It may assign the default value to the method.

Types of Annotations:

4. Marker Annotation

An annotation that has no method is called a marker Annotation

```
@interface MyNotation{}
```

The @Override and @Deprecated are example of marker notations.

2. Single-Value Annotation

An annotation that has one method is called a single-value annotation

```
@interface MyAnnotation{
        int value();
}
```

We can provide the default value as well

```
@interface MyAnnotation{
        int value() default 0;
}
```

How to apply Single-Vawe Annotation

```
@MyAnnotation(Value=10)
```

3. Multi-Value Annotation

An annotation that has more than one method is called a Multi-Value Annotation

```
@interface MyNotation{
        int value1();
        String value2();
        String value3();
```

We can prouide default value as well

```
@interface MyNotation {
        int value1() default 1;
        String value2() default "";
        String value3() default "";
```

How to apply Multi-level Annotation

```
@MyNotation(value1=10, value2="ABC", value3="XYZ")
```

- Generics

The Java Generics programming is introduced in J2SE 5 to deal with type safety objects. It makes code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e non-generic. Now generic force the Java programmer to store a specific type of objects.

Advantage

1. Type - Safety
2. Type casting is not required
3. Compile - Time checking

Syntax

Class or Interface <Type>

Example

ArrayList<String>

- Generic class

A class that can refer to any type is known as generic class. Here we are using Type parameter to create generic class of specific type

```
class MyGen<T> {
    T obj;
    void add(T obj) {
        this.obj = obj;
    }
    T get() {
        return obj;
    }
}

class Test {
    public static void main(String[] args) {
        MyGen<Integer> m = new MyGen<Integer>();
        m.add(2);
        //m.add("A"); // Compile time error
        System.out.println(m.get());
    }
}
```

Output

2

Type Parameters

1. T → Type
2. E → Element
3. K → Key
4. N → Number
5. V → Value

Generic Method

We can create a generic method that can accept any type of arguments. Here the scope of argument is limited to method where it is declared.

```
public class Test {
    public static <E> void printArray (E[] elements) {
        for (E element : elements) {
            System.out.println (element);
        }
        System.out.println();
}
public static void main (String [] args) {
    Integer[] intArray = {1,2,3};
    character[] charArray = {'A', 'B', 'C'};
    printArray (intArray);
    printArray (charArray);
    }
}
```

Output

```
1
2
3

A
B
C
```

- Wildcards

The ? symbol represents the wildcard elements. It means any type.

Eg: If we write <? extends Number>, it means any child class of Number eg. Integer, float and double. Now we can call the method of Number class through any child class object.

We can use a wildcard as a type of a parameter, field return type, or local variable. However it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a supertype

```
abstract class Shape {
        abstract void Shape() { drawI(); }
}
class Circle extends Shape {
        void draw() {
            System.out.println("drawing circle");
        }
}
class Rectangle extends Shape {
        void draw() {
            System.out.println("drawing rectangle");
        }
}
class Test {
    public static void drawShapes(List<? extends Shape> lists){
        for (Shape s : lists)
            s.draw();
    }
}
```

```
public static void main (String [] args) {
    List<Rectangle> list1 = new List<Rectangle>();
    list1.add(new Rectangle());
    list <Circle> list2 : new ArrayList<Circle>();
    list2.add(new circle());
    list2. add(new circle());

    drawShapes(list1);
    drawShapes(list2);
  }
}
```

Output

drawing rectangle
drawing circle
drawing circle

- Lambda Expressions

A lambda expression is a short block of code which takes in parameters and return a value. Lambda expression are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Syntax

parameter → expression

(parameter1, parameter2) → expression {code block}

Example

```
interface funInterface {
    void abstractfun(int x);
}
```