OOPs in JAVA

Classes in JAVA

A java program can have multiple classes but can contain ATMOST one public class.

If there is a public class, the name of java program must be the same as the public class. If there is no public class then any java file name is acceptable.

There is no relation between the presence of main method in the class and file name.

Importing Packages

```
ArrayList L = new ArrayList();   //error occurs
java.util.ArrayList L = new ArrayList();  //Fully qualified name
import java.util.ArrayList
ArrayList l = new ArrayList();   //no error
```

Explicit and implicit import:

```
import java.util.ArrayList  //explicit import – Recommended
import java.util.*;         //implicit import – Readability is low
```

Packages that need not be imported, available by default

`java.lang` [Contains String class, Exception]

`default` [classes present in current working directory]

NOTE: While importing a package, all class and interfaces present in the imported class is imported but the subpackages are NOT imported.

Packages in JAVA

Packages is an encapsulation mechanism to group related classes and interfaces into a single unit.

Advantages: 1. Naming conflicts – two packages can have classes with same class

2. Modularity – Similar classes and interfaces grouped in one package.

3. Maintainability improves

4. Security

NOTE: Always making your classes a part of a package is a good practice.

How to write packages in JAVA

```
package com.SS.demo;          //domain name in reverse
```

When we compile a program, the .class is placed in cwd, but if we want the class file to be placed in the package folder we need to use `-d .` while compiling.

```
javac -d . Test.java
javac -d <destination_directory> <source_files>
java com.SS.demo.Test   //running the class file Test
```

NOTE: 1. Any java program can contain zero or one package statement is allowed.

2. The first non-comment statement in java must be package statement only.

Template of a java program

```
package statement
import statements
class/interface/enum
```

Class Level Modifiers

For top level classes (outer classes)

`abstract` – Objects can't be created

`final` – Child class not possible

`strictfp` –

`<default>` - no modifiers

For Inner class (All outer classes modifiers can be used)

`Private, protected, static`

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

NOTE: While accessing protected members in child class, we must use child class reference only.

Abstract Modifier (Applicable for methods & classes)

Abstract Methods: Methods that ONLY have declaration and NO implementation.

The declaration ends with a semi-colon (;).

```
public abstract void method_name();   //declaring a abstract method
```

NOTE: If the class contains at least one abstract method, the class must also be declared abstract. An abstract class can contain no abstract methods.

Abstract Classes: Classes that are partially implemented.

Objects cannot be created & no methods can be called from class.

Can contain zero number of abstract methods.

The child class is responsible for providing implementation for all methods in abstract parent class.

Interface: Any service requirement specification (SRS).

Any contract between client and service provider

By public and abstract class. (not applicable in newer versions).

Implementation of interface: All methods in an interface are abstract and public by default, even if we don't mention it. But while implementing the method, we must mention `public` access modifier.

All methods declared in the interface must be provided with implementation in the implemented class, or declare the class as `abstract`.

All variables defined in an interface are public and final by default.

```
interface interf
{
    void m1();
}

public class A implements interf
{
    public void m1()
    {

    }
}
```

Abstract Class	Interface
Abstract class can **have abstract and non- abstract** methods.	Interface can have **only abstract** methods. Since Java8, it can have default and static methods also.
Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance.**
Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
An **abstract class** can be extended using keyword extends.	An **interface class** can be implemented using keyword implements.
A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

Default methods were introduced in Java 8 to provide a mechanism for adding new methods to existing interfaces without breaking compatibility with classes that already implement those interfaces. Prior to Java 8, adding a new method to an existing interface would have required all implementing classes to provide an implementation for that method, which could potentially be impractical or infeasible, especially in large codebases or libraries where numerous implementing classes exist.

Static Methods in an interface:

```
interface MathOperations {
    static int add(int a, int b) {
        return a + b;    }

    static int subtract(int a, int b) {
        return a - b;    } }

public class Main {
    public static void main(String[] args) {
```

```
int sum = MathOperations.add(5, 3);
int difference = MathOperations.subtract(5, 3);
System.out.println("Sum: " + sum); // Output: 8
System.out.println("Difference: " + difference); // Output: 2    }  }
```

Since Java 9, private methods too in interfaces can have a body. These private methods can be used to encapsulate common code shared by default methods within the interface.

OOPs Concepts

1. **Data hiding**: Restricting access of data and functions from members outside particular class (declaring a variable private, etc).
2. **Abstraction**: Hiding internal implementation only highlighting set of offered services (using interfaces).
   Advantages: 1. Security 2. Easy enhancement 3. Improved Maintainability 4. Improved Modularity
3. **Encapsulation**: The process of grouping data members and corresponding methods (class). It should compulsorily have data hiding and abstraction. It hides data behind methods. But it increases code length and reduces performance.
   Advantages: 1. Security 2. Easy Enhancement 3. Maintainability 4. Modularity
4. **Inheritance**: Is a relationship

Tightly Encapsulated Class: Every variable in the class and all variables in parent/super class is `private`, 100% data hiding.

Inheritance

Also known as is-a relationship.

Enables code reusability.

Can call both methods in parent and child class.

Parent reference can be used to hold child class object. But can only call methods in parent class.

```
A b = new B();  //A is parent and B is child class. Can call methods only in A
```

Advantages of Inheritance

All Java API classes are based on inheritance only.

`Object` class is the parent (root) class for all java classes.

Types of inheritance

1. Single inheritance: One parent and child class
2. Multiple Inheritance (X): Multiple parent classes and one child class. (Diamond access problem/Ambiguity problem).
3. Multi-level inheritance: Parent classes in multilevel.
4. Hierarchical inheritance: One parent, multiple child classes.
5. Hybrid Inheritance (X): Group of inheritance used together.
6. Cyclic Inheritance (X): Inheriting classes in a cyclic fashion.

Note: Implementing multiple interfaces is possible on one class is possible as interfaces have only declaration and no implementation.

Inheriting properties of 2 interfaces to one child interface is also possible.

```
interface child_inter extends interf, interf2
{

}
```

Note: If a class Abc is NOT extending any other class, then the Abc is direct child of Object class, but if Abc is extending a class then it is an indirect child of Object class. Object will be the parent of Abc's parent class.

Note: Method Signature: Method name, argument types (with order) of a method.

In languages like C++ even return type is part of method signature but not in Java.

Compiler maintains a method table for each class, table has argument types and method names. methodName(<datatype1>, <datatype2>).

METHOD	PURPOSE
Object clone()	Creates a new object that is same as the object being cloned
boolean equals(Object ob)	Determines whether one object is equal to another
protected void finalize()	Called before an unused object is recycled
final class getClass()	Obtains the class of an object at runtime
int hashCode()	Returns the hashcode associated with the invoking object
void notify()	Resumes execution of a thread waiting on the invoking object
void notifyAll()	Resumes execution of all threads waiting on the invoking object
String toString()	Returns a string that describes the object

Polymorphism

Method Overloading: Both methods having same name but different argument types.

In overloading, method resolution is taken care by compiler based on reference type not based on runtime object. Hence overloading is also called as **Compile time polymorphism/static polymorphism/early binding**.

```
public class A
{
    static void m1(Object a)
    {
        System.out.println("Object arg");
    }

    static void m1(int… a)   //var-args always gets least priority,< general
method
    {
        System.out.println("Int arg");
    }

    static void m1(float a)
    {
        System.out.println("Float arg");
    }

    public static void main(String[] args)
    {
        m1(2,3,4,5);  //Int arg
        m1(2.2f);     //Float arg
        m1('a');      //Int arg (Character promoted to int)
        m1(2.2);      //Object arg (Double promoted to parent class Object as
no double arg type)    }}
```

Program2:

```
public class B
```

```
{
    static void mn(int a)
    {
        System.out.println("Int arg");
    }
    static void mn(int a, float b)
    {
        System.out.println("Int, float arg");
    }
    static void mn(float a, int b)
    {
        System.out.println("float, Int arg");
    }

    public static void main(String[] args)
    {
        mn(10);       //Int arg
        mn(20, 20f);  //Int, float arg
        mn(20, 20);   //ERROR: reference to mn is ambiguous

    }}
```

Program3: Method resolution based on reference type NOT based on runtime object.

```
class Parent{}

class Child extends Parent{}

public class test
{
    public void m1(Parent a)
    {
        System.out.println("Parent type");
    }

    public void m1(Child a)
    {
        System.out.println("Child Type");
    }

    public static void main(String[] args)
    {
        test t = new test();
        Parent obj = new Child();

        t.m1(new Parent()); //Parent type
        t.m1(new Child());  //child type
        t.m1(obj);          //Parent type, method resolution based on reference
type not based on runtime object
    }
```

```
}
```

Method Overriding: Child/Subclass has same method as the parent class.

Runtime polymorphism/Dynamic Polymorphism/Late binding

```
class Parent{
    void m1()
    {
        System.out.println("Parent");
    }}

class Child extends Parent{
    void m1()
    {
        System.out.println("Child");
    }}

public class test
{

    public static void main(String[] args)
    {
        Parent p = new Parent();
        Child c = new Child();
        Parent obj = new Child();

        p.m1();   //Parent
        c.m1();   //Child
        obj.m1(); //Child, Method resolution by JVM based on runtime object }}
```

Program 2:

```
class Parent{
    void m1()
    {
        System.out.println("Parent");
    }}

class Child extends Parent{
    void m1(int a)
    {
        System.out.println("Child");
    }}

public class test
{

    public static void main(String[] args)
    {
        Parent p = new Parent();
        Child c = new Child();
```

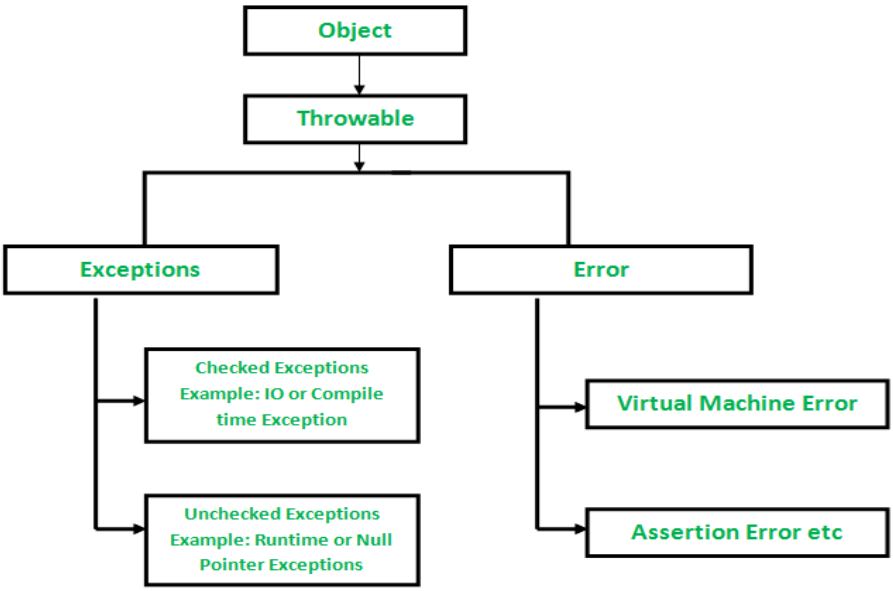
```
Parent obj = new Child();

p.m1();   //Parent
c.m1();   //Parent
obj.m1(); //Parent
c.m1(10); //Child, maybe method overloading    }}
```

Rules of method overriding

1. Method signature (method name, arg list) must be same
2. Return type must also be same (until version 1.4, from 1.5 co-variant datatypes also allowed. Child (object) datatypes allowed in child class, not applicable to primitive types).
3. If a method is private in parent class, then the method is not visible to the child class, hence in this case it will not be considered as overriding.
4. **NOTE**: If parent class method is final it cannot be overridden, but the method that is overriding can be final.
5. **NOTE**: If a parent has non-abstract method, it can be overridden by an abstract method in child class.
6. Synchronized, Native, strictfp methods can also be overridden.
7. **NOTE**: While overriding we cannot reduce the scope of access modifier, but the scope can be increased. Overriding private methods not possible.

Throwable Class (Error & Exception)



NOTE: Runtime exception and its child classes & Error and its child classes are unchecked exception.

8. If child class throws checked exception, the parent must throw the same checked exception. However, child can throw unchecked exceptions.
    i. A child need not always throw the checked exception that the parent throws.
    ii. Child method can throw the child exception of the exception thrown in parent class, vice versa not allowed.

Instance methods are related to objects, but static methods are class level.

9. **NOTE:** An instance method should be overridden with instance method only.
   Static methods cannot be overridden. If we try to override static methods, it is method hiding (NOT overriding). It hides the implementation of super class method.

   Method Hiding: Taken care by compiler based on reference type.
   Compile time/static/ polymorphism or late binding.
   Overriding: Taken care by JVM based on object.
   Runtime/Dynamic polymorphism or late binding.
10. Var-arg methods: var-arg methods have the least priority.
    If the name of the methods is the same but the arguments is different, it is overloading not overriding.

Variable Hiding (overriding not possible in variables)

Variable resolution is taken care by compiler based on reference type, if we define the same variable in parent and child class, the variable in parent class is hidden.

Difference between Overloading & Overriding

Feature	Overloading	Overriding
Signature	Different parameter lists	Same signature (name, parameters, return type)
Resolution	Compile time (based on arguments)	Runtime (based on object's dynamic type)
Applicability	Methods in the same class	Methods in a subclass and its superclass
Purpose	Provide multiple ways to call the same functionality with different arguments	Allow specialization of inherited behavior in subclasses

Object Typecasting (Rules)

```
A b = (C) d;
```

1. Type of d and C must have a relationship (Checked by Compiler).
2. A should be parent or same type of C (Reference type must be parent or same), checked by compiler.
3. The runtime object time of d must be same as C or child of C (Checked by JVM).

NOTE: while typecasting, the reference type is changed not the internal object type, a new object is NOT created.