**Declarations & Access Modifiers**

Topics

1. Java Source File Structures

2. Class level Modifiers

3. Member level Modifiers

4. Interfaces

**1. Java Source File Structures**

> A Java program can contain any no. of classes but at most one class can be declared as public. If there is a public class, then name of the program & name of the public class must be matched otherwise we will get compile time error.

Case 1: - If there is no public class then we can use any name & there are no restrictions like A.java, B.java, C.java etc.

Case 2: - If Class B is public then name of the program should B.java otherwise we will get compile time error.

Case 3: - If Class B & C declared as public & name of the program is B.java then we will get compile time error.

> Whenever we’re compiling a java program, for every class present in that program, a separate .class file will be generated.

> We can compile a java program (Java source file) but we can run only a java .class file.

> Whenever we’re executing a java class, the corresponding class main method will be executed. If the class doesn’t contain main () method then we will get RuntimeException: **NoSuchMethodError**: main

> If corresponding .class file not available then we will get RuntimeException: **NoClassDefFoundError**

> It is recommended to declare only one class per single source file & keep name of program same as class name. The advantage of this approach is readability & maintainability of the code will be improved.

**> Import Statement**

- For using a class in Java, we need to import the required classes into our class

> One way is to use fully Qualified Name. But the problem with fully qualified name is it increases the length of the code & reduces readability.

> Another way is using “import” statement. When using “import” statement, we don’t need to write fully qualified name of the required class, just use short name directly.

> Types of Import statements

a) Explicit class import

b) Implicit class import

a) Explicit class import

> It is highly recommended to use explicit class import because it improves readability of the code.

e.g. import java.util.ArrayList;

b) Implicit class import

> It is not recommended to use because it reduces readability of the code.

e.g. import java.util.\*;

> While resolving class names, compiler will always give precedence in the following order for normal import not static import.

1. Explicit class import //Highest priority

2. Classes present in Current working directory (CWD) or default package.

3. Implicit class import // Lowest priority

> Whenever we’re importing a java package, all classes & interfaces present in that package by default available but not sub-package class, compulsory we should write import statement until sub-package level.

> All classes & interfaces present in the following packages are by default available to every java program; hence we’re not required to write import statement

a) java.lang package

b) default package i.e. current working directory

> Import statements is totally compile time related concept. If more no. of imports then more will be the compile time but there is no affect on execution time. (runtime)

Q. Difference between C language #include & java language import statement ?

Ans: In case of C language #include, all input, output, header files will be loaded at the beginning only (at translation time); hence it is **static include**.

But In case of Java import statement, no .class file will be loaded at the beginning. Whenever we require a particular class then only corresponding .class file will be loaded. This is like **dynamic include** or **Load on Demand** or **Load on Fly**.

**> Static Import**

- If there is no specific requirement then it is not recommended to use static import as it creates confusion & reduces readability.

- Usually we can access static members by using class name but whenever we’re writing static import we can access static members directly without class name.

e.g.

|  |  |
| --- | --- |
| **Without static import** | **With static import** |
| class Test {  public static void main(String[]] args) {  System.out.println(Math.sqrt(4));  System.out.println(Math.max(10, 20));  }  } | Import static java.lang.Math.\*;  class Test {  public static void main(String[]] args) {  System.out.println(sqrt(4));  System.out.println(max(10, 20));  }  } |

Q. Explain about System.out.println

Ans:- class System {

Static PrintStream out;

..

..

}

i.e. **System** is a class present in java.lang package.

**out** is a static variable present in System class of type PrintStream.

**println ()** is a method present in PrintStream class.

> “out” is a static variable present in “System” class hence to access out we need to use class name “System”. But whenever we’re writing static import, it is not required to use class name & we can access “out” directly.

|  |  |
| --- | --- |
| **Without static import** | **With static import** |
| class Test {  public static void main(String[]] args) {  System.out.println(“Hi”);  }  } | Import static java.lang.Sytem.out;  class Test {  public static void main(String[]] args) {  out.println(“Hi”);  }  } |

> While resolving static members, compiler will always consider the precedence in the following order

1. Current class static members // Highest Priority

2. Explicit static import

3. Implicit static import // Lowest Priority

> Difference between Normal import & static import

|  |  |  |
| --- | --- | --- |
| **No** | **Normal import** | **Static import** |
| 1. | Explicit import Syntax:  import *package\_name*.**class\_name;**  e.g.  import *java.util*.**ArrayList**; | Explicit static import Syntax:  import static *package\_name*.**class\_name**.static\_member**;**  e.g.  import static *java.lang*.**Math**.sqrt; |
| 2. | Implicit import Syntax:  import *package\_name*.**\*;**  e.g.  import *java.util*.**\***; | Implicit static import Syntax:  import static *package\_name*.**class\_name**.\***;**  e.g.  import static *java.lang*.**Math**.\*; |

**> Problems with Static import**

- 2 packages contain a class or interface with same name is very rare & hence ambiguity problem is also very rare in normal import. But 2 classes or interfaces contain a variable or method with same name is very common & hence ambiguity problem is also very common problem in static import.

- Usage of static import reduces readability & creates confusion & hence if there is no specific requirement then it is not recommended to use static import.

**> packages**

- It is an encapsulation mechanism to group related classes & interfaces into a single unit, which is nothing but package.

e.g. All classes & interfaces which are useful for **file operations** are grouped in a separate package which is nothing but **java.io** package.

- Advantages of packages are

a) To resolve naming conflicts (i.e. unique identification of our component)

b) It improves modularity of the application.

c) It improves maintainability of the application.

d) It provides security of our component.

- There is one universally accepted naming convention for packages i.e. to use internet domain name in reverse.

i.e. **com.icicibank .loan .housing .Account**

- In any java source file, there can only be atmost one package statement.

- In any java program, the first non-comment statement should be package statement.

- The following is valid Java source file structure (Order of statements is important). Also any empty source file is a valid java program.

|  |
| --- |
| package statement; // at most one  import statement; // any number  class/interface/enum declaration // any number |

2. Class level Modifiers

> Whenever we’re writing our own classes we have to provide some information about our class to the JVM like

a) Whether this class can be accessible from anywhere or not.

b) Whether child class creation is possible or not

c) Whether object creation is possible or not. Etc.

we can specify all these information by using appropriate modifier.

>The only possible modifiers for top – level classes & Inner classes are

|  |  |
| --- | --- |
| Top level classes  1. public  2. <default>  3. final  4. abstract  5. strictfp | Inner Classes  1. public  2. <default>  3. final  4. abstract  5. strictfp  6. private  7. protected  8. static |

> Access Specifier Vs Access Modifier

- There is no word like Specifier in Java but in old languages, private, protected, public, default are considered as Access specifiers.

**1. public Classes**

> If a class is declared as public then we can access that class from anywhere.

**2. default Classes**

> If a class declared as default then we can access that class only within the current package i.e. from outside package we can’t access, hence default class is also known as package level access.

**3. final Modifier**

> final is the modifier applicable for classes, methods & variables.

**a) final Method**

> Whatever methods parent has by default available to the child through inheritance. If the child is not satisfied with parent method implementation then child is allowed to redefine that method based on its requirements. This process is called **Overriding**.

> So, if a parent class method is declared as final, then the child class can’t override that method because its implementation is final.

**b) final class**

> If a class is declared as final, we can’t extend functionality of that class i.e. we can’t create child class for that class i.e. Inheritance is not possible for final classes.

> Every method present inside final class is always final by default but every variable present inside final class need not to be final.

> The main **advantage** of final keyword is we can achieve security & we can provide unique implementation.

> But the main **disadvantage** of final keyword is we’re missing key benefits of OOPs: inheritance (because of final classes) & Polymorphism (because of final methods). Hence if there is no specific requirement then it is not recommended to use final keyword.

**4. abstract Modifier**

> abstract is the modifier applicable for classes & methods only not to variables.

**a) Abstract method**

> Even though we don’t know about implementation still we can declare a method with abstract modifier i.e. for abstract methods only declaration is available but not implementation.

e.g. public abstract void m1 (); // correct

public abstract void m1 () { } // wrong

> Child class is responsible to provide implementation for parent class abstract methods.

> Abstract method never talks about implementation if any modifier talks about implementation then it forms illegal combination with abstract modifier.

**b) Abstract class**

> For any java class, if we are not allowed to create an object (because of partial implementation), such type of class should be declared with abstract modifier i.e. for abstract classes, instantiation is not possible.

**Note**:

> If a class contains at least one abstract method then compulsory we should declare class as abstract otherwise we will get compile time error.

> Even though a class doesn’t contain any abstract method still we can declare the class as abstract if we don’t want instantiation i.e. abstract class can contain zero number of abstract methods also.

e.g. HttpServlet class

> If a class extends an abstract class then the child class has to provide implementation for each & every abstract method or we should declare the child class as abstract (in this case, next level child class is responsible to provide implementation).

> Abstract class can contain final method.

> It is highly recommended to use abstract modifier because it promotes several OOPs features like inheritance & polymorphism.

**5. strictfp [strict floating point]**

> We can declare/use strictfp for classes & methods but not for variables.

> Usually, the result of floating point arithmetic is varied from platform to platform. If we want platform independent results for floating point arithmetic, then we should go for strictfp modifier.

a) strictfp method

> If a method declared as strictfp, all floating point calculations in that method has to follow IEEE 754 Standard, so that we will get platform independent results.

> We can’t declare abstract – strictfp combination for methods.

b) strictfp class

> If a class is declared as strictfp then every floating point calculation present in every concrete method has to follow IEEE 754 Standard so that we will get platform independent results.

> We can declare abstract – strictfp combination for classes.

**3. Member level Modifiers**