**-: Declaration & Access modifiers :-**

**Java source file structure:**

A java program can contain any number of classes but at most one class can be declared as public if there is a public class then name of the program and name of the public class must be matched. Otherwise we will get compile time error.

|  |
| --- |
| class A  {  }  class B  {  }  class C  {  } |

Case: 1

If there is no public class then we can use any name and there are no restriction.

A.Iava , B.Iava, c.Iava , Durga.Iava etc

Case: 2

If class B is public then the program name should be B.Java otherwise we will get compile time error saying : class B is public, should be declared in a file name B.java

Case: 3

If class B and class c declared as public and name of the program is B.java then we will get compile error saying: class C is public, should be declared in a file name C.java

Ex:

public class ConstructorOverloading {  
}  
class A{  
 public static void main(String[] args) {  
 System.*out*.println("A Main class ");  
 }  
}  
class B{  
 public static void main(String[] args) {  
 System.*out*.println("B Main class ");  
 }  
}  
class C{  
 public static void main(String[] args) {  
 System.*out*.println("C Main class ");  
 }  
}

Java A

O/P: A Main class

Java B

O/P: B Main class

Java C

O/P: C Main class

Java ConstructorOverloading

RE: NoSuchMethodError: main

**Conclusions:**

1. When ever we are compiling a java program for every class a separate .class file will be generated.
2. We can compile a java program(source file) but we can run a java .class.
3. Whenever we are executing a java class the corresponding class main method will be executed, if the doesn’t contain main method then we will get runtime exception saying NOSuchMethodError: main.
4. If the corresponding .class file not available then we will get runtime exception saying NoClassDefFounfError.
5. It is not recommended to declare multiple classes in a sing source file.
6. It is highly recommended to declare only class pre source file and name of the program we have to keep same as class name. the main advantages of this approach is readability and maintainability of the code will be improved.

**Import statement:**

class Test {

public static void main(String[ ] args){

ArrayList l = new ArrayList();

}

}

CE: cannot find symbol

Symbol : class ArrayList

Location: class Test

We can solve this problem by using fully qualified name.

i.e. **java.util.ArrayList l = new java.util.ArrayList();**

The problem with using fully qualified name every time is it increases length of the code and reduces readability.

We can solve this problem by using import statement. When ever we are writing import statement it is not required to use fully qualified every time we can use short name directly.

**import java.util.ArrayList;**

class Test {

public static void main(String[ ] args){

ArrayList l = new ArrayList(); // shortcut name : ArrayList

}

}

Hence import statement acts as typing short cut.

**Types of import statements:**

Two types of import statements.

1. Explicit class import Ex: import Java.util.ArrayList
2. Implicit class import Ex : import Java.util.\*

It is highly recommended to use explicit class import in program. It increases the readability of the code. Best suitable for hi-tech where readability is important.

Implicit import is not recommended to use because it reduces the readability of the code. Best suitable for amir-paid where typing is important.

**Case 1:**

Which of the following import statement are meaning full?

Import java.util.ArrayList; // valid

Import java.util.ArrayList.\*; // Invalid

Import java.util.\*; // valid

Import java.util; // In-valid

**Case 2:**

Consider the following code :

Class MyObject extends java.rmi.UnicastRemoteObject{

}

The code compiles fine even though we are not writing import statement. Because we used fully qualified name.

**Note:**  whenever we are using fully qualified name it is not required to use import statement. Similarly whenever we are writing import statement it is not required to use fully qualified name.

**Case 3:**

import java.util.\*;

import java.sql.\*;

class Test{

Public static void main(String[] args){

Date d = new Date();

}

}

CE: reference to Date is ambiguous

**Note:** even in the case of list also we may same ambiguity problem. Because it is available in both util and out package.

**Case 4:**

While resolving the class name compiler will always give the precedence in the following order.

Explicit class import

Classes present in current directory (default package)

Implicit class import

Ex:

import java.util.Date;

import java.sql.\*;

class Test{

Public static void main(String[] args){

Date d = new Date();

Sop(d.getClass().getName());

}

}

In the above example util package date got considered.

**Case 5:**

Whenever we are importing a java package all classes and interfaces present in the package by default available but not sub-packages classes.

If we want to use sub-package classes compulsory we have to until sub-package level.

Ex:

Java 🡪 util 🡪 regex 🡪 Pattern

To use pattern class in our which import statement is required.

import java.\*

import java.util.\*;

import java.util.regex.\*; // answer

No import statement required.

**Case 6:**

There are two package which are not required to use import package. All classes and interface are available to the program.

1. java.lang
2. default package(Current working directory)

**case 7:**

import statement is totally compile time related concept. if more number of imports then more will be compile time but there is no effect on run time/ execution time.

**Case 9**:

Difference c language #include and java language import statement.

In the case of c language #include all input output header files loaded at beginning only at the translation time. Hence it is static include.

But in case of java import statement no .class file will be included at the beginning. Whenever we are using a particular class then only a corresponding .class file will be loading. This is like dynamic include or load on demand or load on fly.

**Static Import:**

Introduced in 1.5 version. According to SUN usage of static import reduces length of the code and improves readability.

But according to world wide programming expert(like us) and reduces readability. Hence if there is no specific requirement then it is not recommended to use static import.

Usually we can static members by using class name but when ever we are using static import we can static member directly without class name.

Ex: Without static Import

class Test{

p s v m (){

Sop(Math.sqrt(4));

Sop(Math.max(4,8));

Sop(Math.random());

}

}

Ex: With static Import

import static java.lang.Math.\*; // we can use explicit for each method also

// import static java.lang.Math.sqrt;

Class Test{

p s v m (){

Sop(sqrt(4));

Sop(max(4,8));

Sop(random());

}

}

* Explain about System.out.println():

class Test{

static String s = “java”;

}

Test.s.length();

Test : is a class name.

s : is a static variable present in Test class of the type java.lang.String

length() : is a method present in String class

Similarly we can take a scenario for System.out.println():

class System{

static PrintStream out;

}

System.out.println();

System : is a class in java.lang.pkg.

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

Println() : it is a method present in PrintStream class

**With static import:**

out is a static variable present in system class hence we can access by using class name System but whenever we are writing static import it is not required to use class name we can access out directly.

import static java.lang.System.out;

class Test{

p s v m ( ){

out.println(“Hello ”);

out.println(“Hi”);

}

}

Ex:

import static java.lang.Integer.\*;

import static java.lang.Byte.\*;

public class Test{

p s v m( ){

Sop(MAX\_VALUE);

}

}

// CE : reference to MAX\_VALUE is ambiguous (Integer & Byte both contain MAX\_VALUE)

While resolving static member compiler will always consider the priority in following order:

1. Current class static member
2. Explicit static import
3. Implicit static import

Ex:

import static java.lang.Integer.MAX\_VALUE;

import static java.lang.Byte.\*;

public class Test{

static int MAX\_VALUE = 234;

p s v m( ){

Sop(MAX\_VALUE);

}

}

O/P: 234

Which of the following import statements are valid ?

import java.lang.Math.\*; // In- valid

import static java.lang.Math.\*; //Valid

import java.lang.Math.sqrt; // In-valid

import static java.lang.Math.sqrt(); // In-valid

import java.lang.Math.sqrt.\*; // In-valid

import static java.lang.Math.sqrt; //Valid

import java.lang; // In-valid

import static java.lang; // In-valid

import java.lang.\*; //Valid

import static java.lang.\*; // In-valid

Two packages contains a class or interface with same name is very rare and hence ambiguity problem is also very rare in normal input.

But two classes contain a variable or method with same name is vary common and hence ambiguity problem is also very common problem in static input.

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

Difference between normal import and static import:

|  |  |
| --- | --- |
| Normal Import | Static Import |
| 1. We can use normal import classes and interfaces of a particular package. 2. Whenever we are using normal import it is not required to use fully qualified name and we can use short name directly. | 1. It is used to import static members of a particular class or interface. 2. When ever we are writing static import it is not required to use class name to access static member and we and use directly. |

**Packages:**

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

Ex: All classes and interfaces which are required for database operation are grouped into a single package which is nothing but java.sql package.

Ex:2 All classes and interfaces which are required for file io(input output) operation grouped into a separate package which is nothing but java.io package.

The main advantages of package are:

* To resolve naming conflicts (i.e. unique identification of components).
* It improves modularity of the application.
* It improves maintainability of the application.
* It provides security for our components.

**Class level Modifier:**

Whenever we are writing our own classes we have to provide some information about our class to the JVM like.

* Whether this class cn be accessible from any where or not.
* Whether child class creation is possible or not.
* Whether object creation is possible or not etc.

We can specify this information by using appropriate modifier.

The only applicable modifiers for top level classes are:

public

default

final

abstract

strictfp

But for inner classes the applicable modifiers are

public

default

final

abstract

strictfp

private

protected

static

Ex:

private class Test{

p s v m( ){

Sop(“Hello”); // CE: modifier private not allowed here

}

}

**Access specifier Vs Access Modifiers:**

public , private , protected ,default are considered as specifiers except these remaining are considered as modifier. But this rule is applicable only for old languages like c++ but not for java. In java all are considered as modifiers only there is no word like specifier.

**public classes**:

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

Ex:

package pack1;

public class A{

public void m1(){

Sop(“Hello”);

}

}

package pack2;

import pack1.A;

class B{

public static void main(String[ ] args){

A a = new A();

a.m1();

}

}

O/P: Hello

If class A is not public then while compiling B class we will get compile time error saying : pack1.A is not public in pack1: can’t be accessed from outside of package.

**default classes:**

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

**final Modifier:**

final is a modifier applicable for classes, method, and variables.

**final() Method:**

whatever methods parent has by default available to the through inheritance. If the child not satisfied with parent method implementation then child is allowed to redefine that method based on his requirement. This process is called overriding.

If the parent class method I declared as final then we can’t override that method in the child class because its implementation is final.

**final class:**

If a class is declared as final we can’t extend functionality of that class i.e. we can’t child class for that class hence inheritance is not possible for final classes.

Ex:

final class P{  
}

class c extends P{

}

// CE: can’t inherit from final P

**Note:** Every method present in side final class is always final by default. But every variable present in final class need not be final.

The main advantage of final keyword is we can achive security and we can provide unique implementation.

But the main disadvantages of final keyword is we are missing key benefits of opps:

Inheritance(because of final class)

Polymorphism (because of final method)

**abstract modifiers:**

abstract is a modifier applicable for classes and methods but not for variables.

**abstract method:**

even though we don’t know about implementation still we can declare a method by abstract modifier i.e. for abstract method only declaration is available but not implementation. Hence abstract method declaration should ends with semicolon.

Ex: public abstract void m1(); // Valid

public abstract void m1(){ } // In-valid

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

Ex:

abstract class vehicle{

abstract public int getNumberWheel();

}

class Bus extends vehicle{

public int getNumberWheel(){

return 7;

}

}

class auto extends vehicle{

public int getNumberWheel(){

return 3;

}

}

By declaring abstract method in the parent class we can provide guide line to the child class such that which method compulsory child has to implement.

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

The following are various illegal combination of modifiers for method with respect to abstract.

abstract – final

abstract – native

abstract – synchronized

abstract – static

abstract – private

abstract – strictfp

Ex: abstract final void m1();

\\ CE: illegal combination of modifiers : abstract and final

**abstract class:**

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

Ex:

abstract class Test{

p s v m (){

Test t = new Test(); // Test is abstract; can’t be instantiated

}

}

class Bus extends vehicle{

public int getNumberWheel(){

}

**abstract class vs abstract method:**

* If a class contains at least one abstract method then compulsory we have to declare that class as abstract otherwise we will get compile time error.

If a class contains at least one abstract method then implementation is not complete and hence it is not recommended to create object. To restrict object instantiation compulsory we should declare the class as abstract.

* Even though class does not contain any abstract method still we can declare the class as abstract if we don’t want instantiation i.e. abstract class can contain 0 number of abstract class also.

If we are extending abstract class then for each and every abstract method of parent class we should provide implementation otherwise we have to declare child class as abstract.

In this case next level child class is responsible to provide implementation.

abstract class p{  
 public abstract void m1();

public abstract void m2();  
}

class c extends p{

public void m1(){ }

}

//CE: c is not abstract and doesn’t override in p

**final vs abstract:**

* abstract methods compulsory we should override in child classes to provide implementation. Where as we can’t override final method. Hence final abstract combination is illegal combination for methods.
* For final classes we can’t create child class where as for abstract we should create to provide implementation hence final abstract combination is illegal for classes.
* Abstract class can contain final method but final class can’t contain abstract method.

Ex1:

abstract class Test{

public final void m1(){ } // Valid

}

Ex2:

final class Test{

public abstract void main(); //In-valid

}

Int is highly recommended to use abstract modifier because it promotes several opps feature like inheritance and polymorphism.

**strictfp modifier (strict floating point):**

Introduced in 1.2 version.

We can use strictfp for classes and methods but not for variable.

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

**strictfp method:**

If a method declared as strictfp all floating point calculation in that method has to follow IEEE-754 standard so that we will get platform independent results.

Abstract modifier never talks about implementation where as strictfp method always talks about implementation. Hence abstract-strictfp is illegal for methods.

**strictfp class:**

If a class declared as strictfp then every floating point calculation present in concrete method has to follow IEE-754 standard so that we will get platform independent result.

We can declare abstract strictfp combination for classes i.e. abstract strictfp combination is illegal for classes but illegal for methods.

Ex:

abstract strictfp class Test{

} // Valid

Ex:

abstract strictfp void m1(); //CE: illegal combination of modifiers : abstract and static

**Member Modifiers:** (method or variable level modifier)

If a member declared as public then we can access that member from any where but corresponding class should be visible i.e. before checking member visibility we have to check class visibility.

Ex:

package pack1;

class A{

public void m1(){

Sop(“A class method”);

}

}

package pack2;

import pack1.A;

class B{

public static void main(String[ ] args){

A a = new A();

a.m1();

}

}

CE: pack1.A is not public in pack1; can’t be accessed from outside package

In the above example even though m1() method is public we can’t access from outside package because corresponding class is not public. i.e. if both class and method are public then only we can access that method from outside package.

**Default members:**

If a member declared as default then we can access that member only within the current package from outside of the package we cant access. Hence default package is also known as package level access.

**Private members:**

If a member is private then we can access that member only within the class i.e. from outside of the class we can’t access.

Abstract method should be available to the child classes to provide implementation. Where as private methods are not available to the child methods to provide implementation. Hence private abstract combination is illegal for methods.

**Protected members**: (the most miss understood modifier in java)

If a member declared as protected then we can access that anywhere within the current package but only in child classes of outside package.

Protected = default + kid

Continued from lecture-36