Declarations and Access modifiers:

first of all, just have a look at Java source file structure:

1. A java source file can contain any no of classes but atmost one class can be declared as public.
2. If ther is a public class, compulsory the name of the public class and name of source file must be matched, otherwise compile time error.
3. If there is no public class, any name we can given as java source file and there are no restrictions on file name.
4. suppose in our java file, if we have two A and B public classes and we save the java file as A.java, then we will get compile time error saying “class B is public, should be declared in a file named B.java”.
5. so, it is always a good practice to maintain only one class for one source file and we can maintain that class name as source file name.

case (i):

class Test

{

Public static void main(String args[])

{

ArrayList al=new ArrayList();

}

}

// C.T.E saying can’t find symbol ArrayList.

We can solve the above error in two ways.

1st way:

by taking fully qualified name as java.util.ArrayList al=new java.util.ArrayList();

it is fine but there is a problem in this approach. Using fully qualified name reduces the readability of the code. That’s why, just go for 2nd way.

2nd way:

Here we solve the above problem simply by using import statements.

import java.util.ArrayList;

class Test

{

Public static void main(String args[])

{

ArrayList al=new ArrayList();

}

}

Note: import statements simply acts as typing shortcut and increases the readability of the code.

Q: what is the difference between #include and import statement?

in the case of #include, all the required header files will be loaded at the beginning only. But in the case of import statement using, the specified classes won’t be loaded at the beginning, they load whenever we are using a particular class, then only corresponding class will be loaded i.e import statement follows load-on-fly.

types of import statements:

there are two types of import statements.

1. explicit class import i.e import java.util.ArrayList;
2. implicit class import i.e import java.util.\*;

1st type increases readability of the code, recommended to use for real time coding.

2nd type reduces readability of the code, not recommended to use.

Q: which of the following are valid import statements?

1. import java.util.\*;
2. import java.util.ArrayList;
3. import java.util;
4. import java.util.ArrayList.\*;

Answer: 1,2 are valid. 3,4 are invalid.

Case (i):

we can use fully qualified name when we are importing instead of import. It executes fine.

class Test extends java.rmi.UnicastRemoteObject

{

}

Case (ii):

import java.util.\*;

import java.sql.\*;

class Test

{

Public static void main(String args[])

{

Date d=new Date(); // C.T.E

}

}

C.T.E saying reference to Date is ambiguous( means confusion), because Date class is present in both util, sql packages.

Case (iii):

While resolving class names, compiler will always give the first preference to explicit class import, next to implicit class import.

import java.util.Date; // it is explicit class import

import java.sql.\*; // it is implicit class import

class Test

{

Public static void main(String args[])

{

Date d=new Date(); // it will be java.util.Date

System.out.println(d.getClass().getName());

}

}

o/p: java.util.Date

Note: whenever we are importing a package,its sub package classes won’t be imported.

Q: java.util.regex.Pattern is a class. to import the Pattern class, which of the following is correct?

1. import java.\*;
2. import java.util.\*;
3. import java.util.regex.\*;
4. all the above

Answer: 3 is correct.

Import is totally compile time issue, there is no impact on runtime.

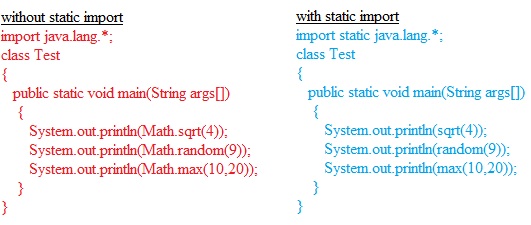
It is not required to import the following packages because they are available by default for every program.

1. lang package
2. default package ( i.e current directory, where java file places)

static import:

this concept is introduced in 1.5, according to static import, this import improves readability of the code, but according to worldwide experts, this import reduces readability of the code and increases the confusion of the code. Hence it is not recommended to use until and unless there is no specific requirement.

Whenever we are using static import, it is not required to use class name while accessing static members. See the following example for getting clarity



While resolving static members, compiler will always give precedence in the following order.

(i)current class static members

(ii)explicit static import

(iii)implicit static import

Example:

import static java.lang.Integer.MAX\_VALUE; --------2

import static java.lang.Byte.\*; ----------3

class Test

{

Static int MAX\_VALUE=9111; ----------1

Public static void main(String args[])

{

System.out.println(MAX\_VALUE);

}

}

o/p: 9111

if we comment line 1, then explicit static member 2 will be considered and o/p is 2147483647.

If we comment line 1 and 2, then o/p is 127.

Some tips in static import:

In static import, we can import particular member field (i.e variable) or method, but not all.

import static java.lang.Integer.MAX\_VALUE; // it is valid

import static java.lang.Integer.bitCount; // it is valid

import static java.lang.Integer.\*; // it is valid

import static java.lang.Integer; // it is invalid.

In our general import, we can’t import particular member field (i.e variable) or method, but all.

import java.lang.Integer.MAX\_VALUE; // it is invalid

import java.lang.Integer.\*; // it is invalid

import java.lang.Integer; // it is valid.

packages:

it is an encapsulation mechanism to group related .class files and interfaces into a single module. The main purpose of the package statement is to resolve naming conflicts.

There is universally accepted naming convention to name the packages.

Package name should be always in lower case.

>javac Test.java

The generated .class file will be placed in current working directory(default package).

>javac -d . Test.java

>javac -d C: Test.java

Destination to place generated .class files will be placed in the corresponding package structure.

Case (i):

Package com.sharingfolder.gutta;

Class Test

{

}

To compile this, we need to create first the package directory structure i.e

gutta folder in sharingfolder in com folder and we should place the Test.java file and then we have to compile.

The alternative way is compiling by the following command.

>javac -d . Test.java

This command automatically creates the related directory structure and places the compiled java file in com.sharingfolder.gutta folder.

If the related folder is already available, then it uses that folder to place the .class file.

to run this file

>java com.sharingfolder.gutta.Test

conclusions on packages:

1.for any java source file, there should be atmost one package statement.

package pack1;

package pack2; // C.T.E

class Test

{

}

2.package statement should be the first statement in the java file otherwise we will get error.

Example:

import java.util.\*;

package pack2; // C.T.E

class Test

{

}

Q: which of the following is a valid sequence of statements in java source file?

1. import, package, class
2. package, import, class
3. import, class, interface
4. package, interface, class

Answer: 2,3,4 are correct. 1 is not correct.

Q:which of the following are valid java source files?

1. empty file
2. file contains only one statement which is import statement
3. file contains only one statement which is package statement

Answer: 1,2,3 are correct.

class level modifiers:

1. whenever we are writing our own java classes, compulsory we have to provide the following information to the jvm.
2. Whether this class is accessible from anywhere or not.
3. Shall I accept to create an object of this class or not.
4. Shall I accept to create child class or not.

The only applicable modifiers for top level (i.e outer) classes are

1. public
2. default
3. final
4. abstract
5. strictfp

if we use any other modifier, we will get compile time error.

The only applicable modifiers for inner classes are

1. public
2. default
3. final
4. abstract
5. strictfp
6. private
7. protected
8. static

public class (i.e for outer, inner classes):

if a class is declared as public, then we can access that class from anywhere.

Note: public class name and the java file name should be the same. Otherwise we will get error.

So we can write only one public class per a java file.

default class(i.e for outer, inner classes):

if a class is declared as default, then we can access that class only in the current package. But not from outside package. i.e the default class is also known as “package level access”.

final class(i.e for outer, inner classes):

final is the modifier applicable for classes, methods and variables.

If a class is declared as final, then we are not allowed to create the child class for it.

abstract class(i.e for outer, inner classes):

abstract is the modifier applicable only for classes,methods but not for variables.

If a class is declared as abstract, it can hold all abstract methods or all concrete methods or both.

If a class contains at least one abstract method, then it should be considered as abstract class.

Note: we can’t create object for abstract class.

abstract class A

{

}

A a1=new A(); // C.T.E saying A is abstract, can’t be instantiated.

strictfp class(i.e for outer, inner classes):

strictfp modifier is applicable for classes, methods but not for variables.

If a class is declared as strictfp, then all concrete methods of that class has to follow IEEE745 standard for floating point arithmetic.

private (only for inner class):

the private inner class object scope is within the outer class.

protected (only for inner class):

the protected inner class object scope is within the current package and its child class which is in other package.

static (only for inner class):

the static inner class object will be created when the outer class is loading.

member modifiers(i.e method modifier, instance variable modifier):

a class’s instance variable (i.e field), method are called as members of the class.

member modifier means the modifiers which are applicable for a class’s member variable, method.

public member(i.e public method, public instance variable):

if a member is declared as public, then we can access that member from anywhere, but corresponding class must be visible.

Case (i):

package pack1;

class A

{

public void fun()

{

System.out.println(“observe carefully”);

}

}

package pack2;

class B

{

public static void main(String args[])

{

A a1=new A();

a1.fun();

}

} C.T.E saying can’t find symbol a1

Note:

even though fun() is public, we can’t access from A, because class A is not public. So before checking member visibility, compulsory we have to check class visibility.

default member(i.e default method, default instance variable):

if a member is declared as default, then we can access that member only in the current package. Hence default access is also known as package level access.

private member(i.e private method, private instance variable):

if a member is declared as private then we can access that member only within the class.

protected member(i.e protected method, protected instance variable):

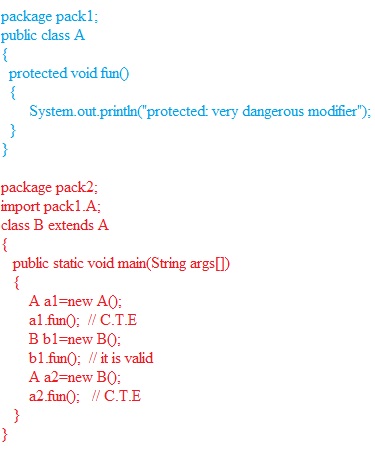
if a member is declared as protected then we can access that member in the current package and its child class which is in other package. i.e protected = default + kids

Note:

Parent class protected members can be accessible either by using parent class reference or child class reference within the current package. But

from out side package, parent class protected members can be accessible by using child class reference only. If we try to access parent class protected members by using parent class reference, then we will get compile time error.

See the following example



final member(i.e final method, final instance variable):

final method:

generally whatever parent has are by default available to child through inheritance if the child is not satisfied with parent class implementation, then child can overload the parent defined implementation in its own way.

But in some cases to achieve security, we can stop the child class from overloading the parent defined implementation. It is possible by assigning parent class method with final modifier.

final methods can’t be overridden.

Example:

class Parent

{

public final void fun()

{

}

}

class Child extends Parent

{

public void fun() // C.T.E saying overridden method is final.

{

}

}

final instance variable:

generally for static and instance variable, jvm will always provide default value. It is not required to perform explicit initialization. But for the local variables, jvm won’t provide any default values. Compulsory we should perform initialization before using that variable.

Note: if instance variable is declared as the final, then compulsory we should perform initialization whether you are using it or not. See the following example

class Test

{

final int i; // C.T.E saying variable i might not have been initialized.

}

For the final instance variable, compulsory we should perform initialization before constructor completion i.e the following are the various places to initialize

1. at the time of declaration i.e

final int i=10;

1. final int i;

{

i=10;

}

1. within the constructor

final int i;

Test()

{

i=10;

}

1. if declare anywhere else, we will get compile time error.

class Test

{

final int i;

public void fun()

{

i=10; // C.T.E saying can’t assing a value to final variable.

}

}

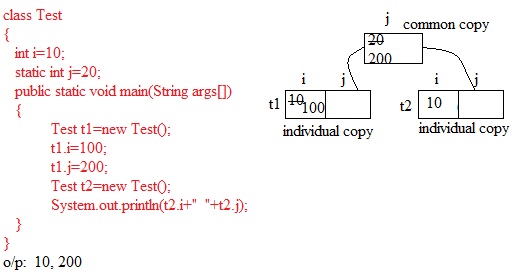
static member(i.e static method, static instance variable):

static is the modifier, applicable for methods, variables, inner classes.

static instance variable:

In the case of instance variables, for every object a separate copy will be created but in the case of static variables only one copy will be created per class and shared by all objects of that class.

See the following example



Note:

If we do any change in static variable, it will be reflected on all objects.

If we do any change in instance variable, it will be reflected on that object only, but no impact on other objects.

Note:

Just to avoid doing changes in static variable, most of the cases static variables are declared as final.

Note:

from static area, we can access static members directly but not instance members directly.

But from instance area, we can access both static & instance members directly. Here directly means without the object help.

static method:

1. for the static methods, compulsory implementation must be available. Hence static methods can’t be declared as abstract.
2. Overloading concept is applicable for static method.
3. Inheritance concept is applicable for static method. See the following example

Sample.java

class Parent

{

public static void main(String args[])

{

System.out.println(“Parent”);

}

}

class Child extends Parent

{

}

>javac Sample.java

>java Parent // o/p: parent

>java Child // o/p: parent

4. overriding concept is not applicable for static methods, alternately we have method hiding concept.

abstract member(i.e abstract method only,but not for variable):

abstract method:

even though we don’t know about implementation, but we can declare methods with abstract keyword i.e abstract methods can have only declarations but not implementation.

abstract method declaration should compulsory end with semicolon.

public abstract void fun(); // it is valid abstract method

public abstract void fun() { }; // it is invalid abstract method

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

Through abstract keyword, we can provide guidelines to the child classes.

for any class, if we don’t want instantiation, such type of classes, we have to declare with abstract modifier, i.e for abstract classes instantiation is not possible.

Strictfp member(i.e strictfp method only, but not for variable):

if a method is declared as strictfp, all floating point calculations in that method has to follow IEEE756 statndard. So that we will get platform independent results.

Q: abstract method versus abstract class

1. If a class contain at least one abstract method then compulsory corresponding class should be declared as abstract, otherwise we get compile time error.
2. even though class doesn’t contain any abstract method, still we can declare that class with abstract modifier. Ex: HttpServlet

case(i):

class A

{

public void fun(); // C.T.E saying missing method body or declare abstract

}

Case(ii):

class A

{

public abstract void fun() { } // C.T.E saying abstract methods can’t have body

}

Case(iii):

class A

{

public abstract void fun(); // C.T.E saying A is not abstract & doesn’t override abstract method in A.

}

Case(iv):

abstract class A

{

public abstract void fun1();

public abstract void fun2();

}

class B extends A

{

}

C.T.E saying B is not bstract doesn’t override abstract method fun1() in A

Case(v):

abstract class A

{

public abstract void fun1();

public abstract void fun2();

}

class B extends A

{

Public void fun1(){ }

}

C.T.E saying B is not bstract doesn’t override abstract method fun2() in A

Case(vi):

abstract class A

{

public abstract void fun1();

public abstract void fun2();

}

abstract class B extends A

{

Public void fun1(){ }

}

The above code is valid. No errors.

Note: child class is responsible to provide implementation for every abstract method of parent otherwise we have to declare child class as abstract. Violation leads to compile time error.

Note: abstract method never talks about implementation. If any modifier talks about implementation, then it is considered as illegal combination with abstract.

The following is the list of various illegal combination for methods

abstract with (final, synchronized, native, static, strictfp, private)

comparison between abstract and final

|  |  |
| --- | --- |
| Abstract | Final |
| for abstract classes compulsory we should create child class to provide implementation for abstract method. | But for final classes, we can’t create child class. hence final abstract combination is illegal for classes. |
| abstract method, compulsory we should override in child class to provide implementation. | but final methods can’t be overridden. Hence abstract final combination is illegal for methods. |

Note:

A final class can’t have abstract methods but an abstract class can contain final methods.

|  |  |
| --- | --- |
| final class Test  {  abstract void fun();  }  // the above is invalid. | abstract class Test  {  final void fun() { }  }  // the above is valid. |

Note:

abstract & strictfp is legal combination for classes, but illegal for methods.

Ex:

abstract strictfp class Test

{

} // it is valid

Ex: abstract strictfp void fun(); // it is invalid.

Note: abstract methods can’t be declared as strictfp

Q: private & abstract combination is illegal for methods. Why?

private methods are not visible in child class but abstract method should be visible in child for

getting implementation. Hence private abstract combination is illegal for method.

Comparison between public, protected, default, private

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Visibility | private | default | protected | Public |
| Within the class | Yes | Yes | Yes | Yes |
| Within the package | No | Yes | Yes | Yes |
| From child class of ouside package | No | No | Yes,but we should use child reference.that’s why protected is dangerous. | Yes |
| From non child class of outside package | No | No | No | Yes |

Note:

The most accessible modifier is public and most restricted modifier is private.

data members are recommended to declare with private and methods are recommended to declare with public modifier.

final static variables:

if the static variable declared as final compulsory we should perform initialization whether we are using or not. This should be done before class loading completion.

The following are various possibilities.

1. At the time of declaration

final static int i=10;

1. inside static block

final static int i;

static

{

i=10;

}

1. if we perform initialization anywhere else, we will get compile time error.

final local variables

even though local variable declared as final, it is not required to perform initialization until we are not using the variable. But before we use the local variable, we have to perform initialization. See the following example

class Test

{

public static void main(String args[])

{

final int i;

i=10;

System.out.println( i );

}

}

Note: the only applicable modifier for the local variable is final. If we declare any other modifier we will get compile time error.

Case(i):

The variables inside a method signature are called formal parameters.

The formal parameters declared inside a method are acted as local variables of the method. Hence we can declare a formal parameter as final. So once we declare a formal parameter as final, then we can’t change its value inside the method. For clarity see the following

class Test

{

public static void main(String args[])

{

fun(10,20);

}

public static void fun( final int i, int j)

{

i=100; // C.T.E saying can’t assign a value to final variable

j=200;

}

}

Q:which of the following are valid combinations?

1. int i=10;
2. static int i=10;
3. public void fun()

{

System.out.println(i);

}

1. public static void main(String args[])

{

System.out.println(i);

}

1. 1 & 3
2. 1 & 4
3. 2 & 3
4. 2 & 4
5. 1 & 2
6. 3 & 4
7. 2 & 3 & 4

Answer: a, c, d, g are correct. b,e,f are wrong.

T H E E N D