Declaration and Access Modifiers

Agenda

1. Java source file structure

o Import statement

o Types of Import Statements

 Explicit class import

 Implicit class import

o 1.5 versions new features

o Static import

 Without static import

 With static import

o Explain about System.out.println statement ?

o Package statement

 How to compile package Program

 How to execute package Program

o Java source file structure

2. Class Modifiers

o Only applicable modifiers for Top Level classes

o Public Classes

o Default Classes

o Final Modifier

 Final Methods

 Final Class

o Abstract Modifier

 Abstract Methods

 Abstract class

o The following are the various illegal combinations for methods

o What is the difference between abstract class and abstract method ?

o What is the difference between final and abstract ?

o Strictfp

3. Member modifiers

o Public members

o Default member

o Private members

o Protected members

o Compression of private, default, protected and public

o Final variables

 Final instance variables

 At the time of declaration

 Inside instance block

 Inside constructor

 Final static variables

 At the time of declaration

 Inside static block

 Final local variables

o Formal parameters

o Static modifier

o Native modifier

 Pseudo code

o Synchronized

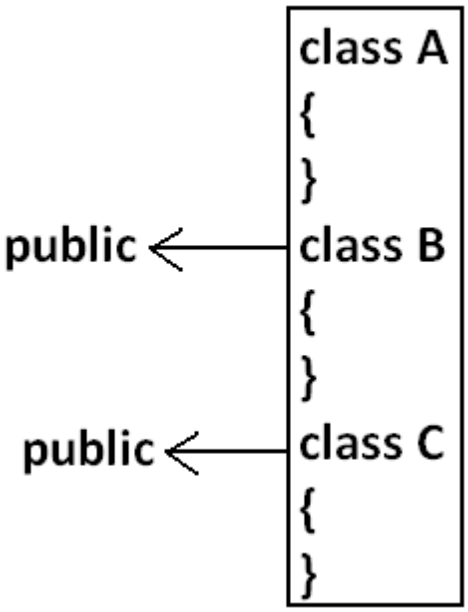
o Summary of modifier

Java source file structure:

 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 the name of the Program and name of the public class must be matched otherwise we will get compile time error".

 If there is no public class then any name we gives for java source file.

Example:



Case 1:

If there is no public class then we can use any name for java source file there are no restrictions.

Example:

A.java

B.java

C.java

Ashok.java

case 2:

If class B declared as public then the name of the Program should be B.java otherwise

we will get compile time error saying "class B is public, should be declared in a file

named B.java".

Case 3:

 If both B and C classes are declared as public and name of the file is B.java then

we will get compile time error saying "class C is public, should be declared in a file named C.java".

 It is highly recommended to take only one class for source file and name of the

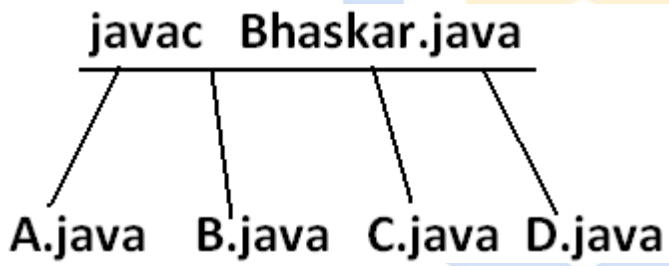
Program (file) must be same as class name. This approach improves readability and understandability of the code.

Example:

```
class A
{
public static void main(String args[]){
System.out.println("A class main method is executed");
}
}
class B
{
public static void main(String args[]){
System.out.println("B class main method is executed");
}
}
```

```
class C
{
public static void main(String args[]){
System.out.println("C class main method is executed");
}
}
class D
{
}
```

Output:




```
D:\Java>java A
A class main method is executed
D:\Java>java B
B class main method is executed
D:\Java>java C
C class main method is executed
D:\Java>java D
Exception in thread "main" java.lang.NoSuchMethodError: main
D:\Java>java Ashok
Exception in thread "main" java.lang.NoClassDefFoundError: Ashok
```

 We can compile a java Program but not java class in that Program for every class one dot class file will be created.

 We can run a java class but not java source file whenever we are trying to run a

class the corresponding class main method will be executed.

 If the class won't contain main method then we will get runtime exception saying

"main method not found,please define the main method as public static void main(String[] args)".

 If we are trying to execute a java class and if the corresponding .class file is not

available then we will get runtime execution saying "could not find or load main class:

Ashok".

Import statement:

```
class Test{
public static void main(String args[]){
ArrayList l=new ArrayList();
}
}
```

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:3: cannot find symbol
symbol : class ArrayList
location: class Test
ArrayList l=new ArrayList();
```

 We can resolve this problem by using fully qualified name "java.util.ArrayList l=new java.util.ArrayList();". But problem with using fully qualified name every time is it increases length of the code and reduces readability.

 We can resolve this problem by using import statements.

Example:

```
import java.util.ArrayList;
class Test{
public static void main(String args[]){
ArrayList l=new ArrayList();
}
}
```

Output:

```
D:\Java>javac Test.java
```

Hence whenever we are using import statement it is not require to use fully qualified

names we can use short names directly. This approach decreases length of the code and

improves readability.

Case 1: Types of Import Statements:

There are 2 types of import statements.

1) Explicit class import

2) Implicit class import.

Explicit class import:

Example: Import java.util.ArrayList

 This type of import is highly recommended to use because it improves readability of the code.

 Best suitable for Hi-Tech city /Industry level where readability is important.

Implicit class import:

Example: import java.util.*;

 It is never recommended to use because it reduces readability of the code.

 Best suitable for Awadhpuri /Institute (not for us)where typing is important.

Case 2:

Which of the following import statements are meaningful ?

import java.util; ×
import java.util.ArrayList.*; ×
import java.util.*; ✓
import java.util.ArrayList; ✓

Case 3:

consider the following code.

```
class MyArrayList extends java.util.ArrayList
{
}
```

 The code compiles fine even though we are not using import statements because we used fully qualified name.

 Whenever we are using fully qualified name it is not required to use import statement. Similarly whenever we are using import statements it is not require to
use fully qualified name.

Case 4:

Example:

```
import java.util.*;
import java.sql.*;
class Test
```

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

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:7: reference to Date is ambiguous,
both class java.sql.Date in java.sql and class java.util.Date in java.util
match
Date d=new Date();
```

Note: Even in the List case also we may get the same ambiguity problem because it is

available in both util and awt packages.

Case 5:

While resolving class names compiler will always gives the importance in the following order.

1. Explicit class import
2. Classes present in current working directory.
3. Implicit class import.

Example:

```
import java.util.Date;
import java.sql.*;
class Test
{
public static void main(String args[]){
```

Date d=new Date();

}}

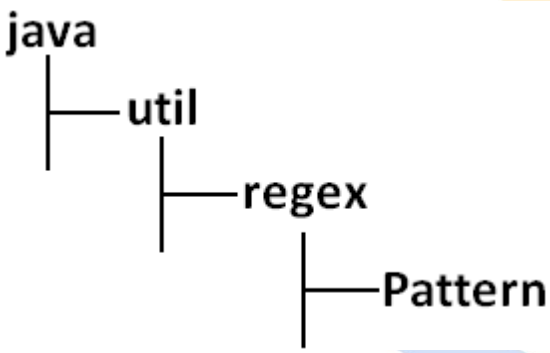
The code compiles fine and in this case util package Date will be considered.

Case 6:

Whenever we are importing a package all classes and interfaces present in that package

are by default available but not sub package classes.

Example:



To use pattern class in our Program directly which import statement is required ?

Case7:

In any java Program the following 2 packages are not required to import because these are available by default to every java Program.

1. java.lang package

2. default package(current working directory)

Case 8:

"Import statement is totally compile time concept" if more no of imports are there then

more will be the compile time but there is "no change in execution time".

New Features of Java 1.5 Versions---

1. For-Each

2. Var-arg

3. Queue

4. Generics

5. Auto boxing and Auto unboxing

6. Co-varient return types

7. Annotations

8. enum

9. static import

10. String builder

Static import:=

This concept introduced in 1.5 versions. According to sun people static import improves readability of the code but according to worldwide Programming exports (like us) static

imports creates confusion and reduces readability of the code. Hence if there is no

specific requirement never recommended to use a static import.

Usually we can access static members by using class name but whenever we are using

static import it is not require to use class name we can access directly.

```
class Test
{
public static void main(String args[]){
System.out.println(Math.sqrt(4));
System.out.println(Math.max(10,20));
System.out.println(Math.random());
```

```
}}
```

Output:

```
D:\Java>javac Test.java
D:\Java>java Test
2.0
20
0.841306154315576
```

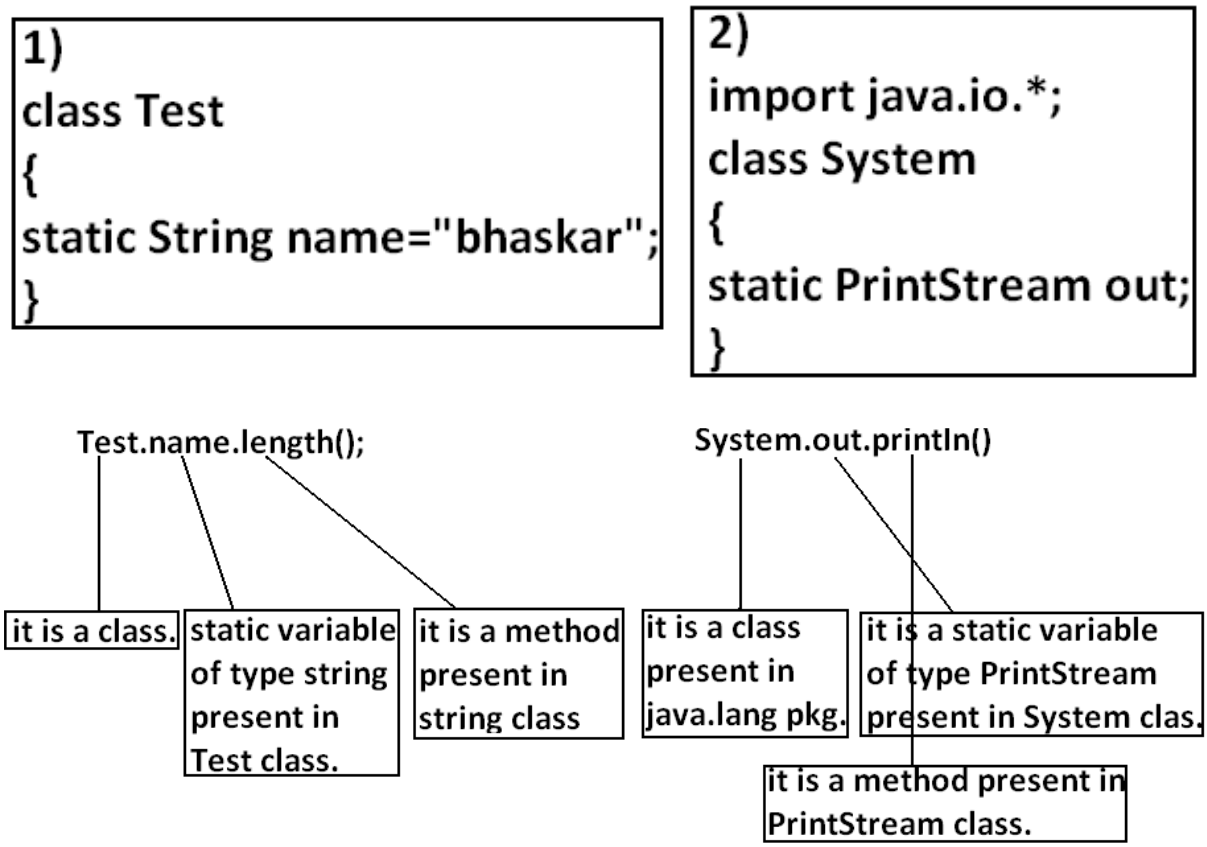
```
import static java.lang.Math.sqrt;
import static java.lang.Math.*;
class Test
{
public static void main(String args[]){
System.out.println(sqrt(4));
System.out.println(max(10,20));
System.out.println(random());
}}
```

Output:

```
D:\Java>javac Test.java
D:\Java>java Test
2.0
20
0.4302853847363891
```

Explain about System.out.println statement ?

Example 1 and Example 2:



Example 3:

```
import static java.lang.System.out;
class Test
{
public static void main(String args[]){
out.println("hello");
out.println("hi");
}}
```

Output:

```
D:\Java>javac Test.java
D:\Java>java Test
hello
hi
```

Example 4:

```
import static java.lang.Integer.*;
import static java.lang.Byte.*;
class Test
{
public static void main(String args[]){
System.out.println(MAX_VALUE);
}}
```

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:6: reference to MAX_VALUE is ambiguous,
both variable MAX_VALUE in java.lang.Integer and variable MAX_VALUE in
java.lang.Byte match
System.out.println(MAX_VALUE);
```

Note: Two packages contain a class or interface with the same is very rare hence ambiguity problem is very rare in normal import.

But 2 classes or interfaces can contain a method or variable with the same name is very common hence ambiguity problem is also very common in static import.

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

1. Current class static members

2. Explicit static import

3. implict static import.

```
//import static java.lang.Integer.MAX_VALUE;------> line2
import static java.lang.Byte.*;
class Test
{
//static int MAX_VALUE=999;------> line1
public static void main(String args[])throws Exception{
System.out.println(MAX_VALUE);
}
}
```

Example:

 If we comet line one then we will get Integer class MAX_VALUE 2147483647.

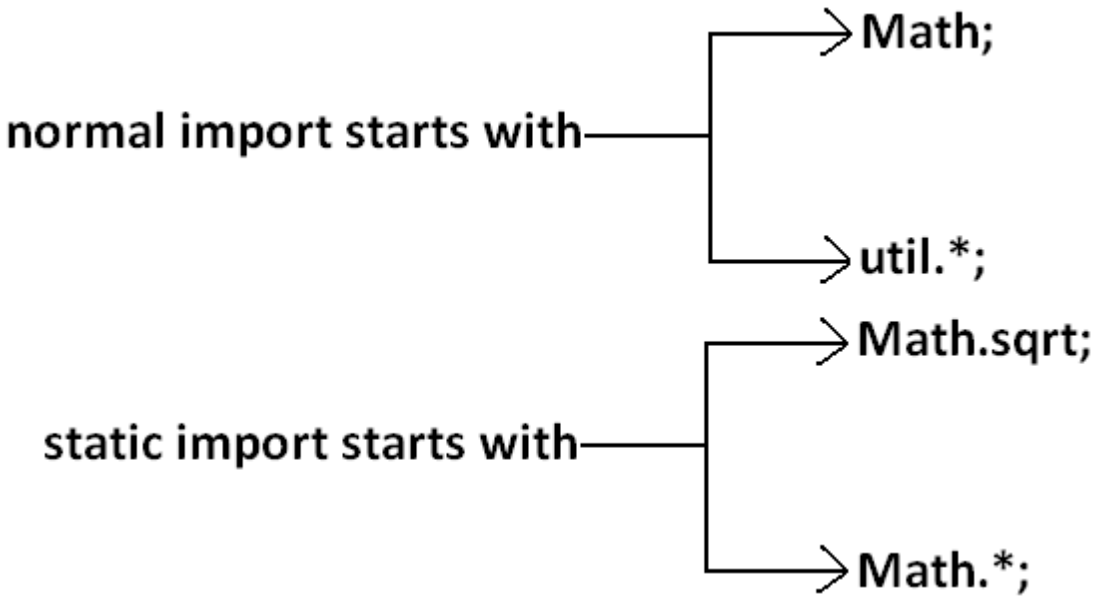
 If we comet lines one and two then Byte class MAX_VALUE will be considered

1. 127.

2. Which of the following import statements are valid ?

1.import java.lang.Math.*; ×
2.import static java.lang.Math.*; ✓
3.import java.lang.Math; ✓
4.import static java.lang.Math; ×
5.import static java.lang.Math.sqrt.*; ×
6.import java.lang.Math.sqrt; ×
7.import static java.lang.Math.sqrt(); ×
8.import static java.lang.Math.sqrt; ✓

Diagram:



Usage of static import reduces readability and creates confusion hence if there is no

specific requirement never recommended to use static import.

What is the difference between general import and static import ?

 We can use normal imports to import classes and interfaces of a package.

whenever we are using normal import we can access class and interfaces directly

by their short name it is not require to use fully qualified names.

 We can use static import to import static members of a particular class.

whenever we are using static import it is not require to use class name we can access static members directly.

Package statement:

It is an encapsulation mechanism to group related classes and interfaces into a single

module.

The main objectives of packages are:

 To resolve name confects.

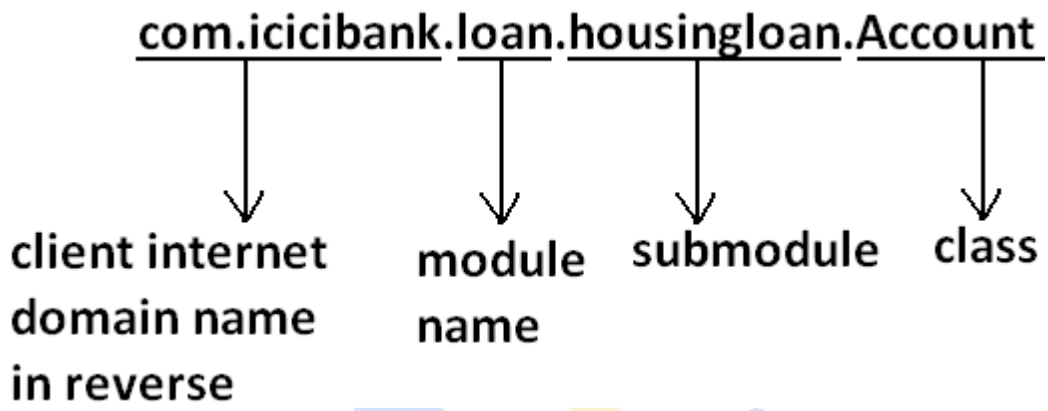
 To improve modularity of the application.

 To provide security.

 There is one universally accepted naming conversion for packages that is to use

internet domain name in reverse.

Example:



How to compile package Program:

Example:

```
package com.ByteCodejobs.itjobs;
class Jobs
{
public static void main(String args[]){
    System.out.println("package demo");
}
}
```

Javac KnpJobs.java generated class file will be placed in current working directory.

Diagram:

 Javac -d . Jobs.java

 -d means destination to place generated class files "." means current working directory.

 Generated class file will be placed into corresponding package structure.

Diagram:

 If the specified package structure is not already available then this command itself will create the required package structure.

 As the destination we can use any valid directory.

  If the specified destination is not available then we will get compile time error.

Example:

Diagram:

If the specified destination is not available then we will get compile time error.

Example:

D:\Java>javac -d z: Jobs.java

If Z: is not available then we will get compile time error.

How to execute package Program:

D:\Java>java com.ByteCodejobs.itjobs.KnpJobs

At the time of execution compulsory we should provide fully qualified name.

Conclusion 1:

In any java Program there should be at most one package statement that is if we are

taking more than one package statement we will get compile time error.

Example:

```
package pack1;
package pack2;
class A
{
}
```

Output:

Compile time error.

```
D:\Java>javac A.java
A.java:2: class, interface, or enum expected
package pack2;
```

Conclusion 2:

In any java Program the 1st non comment statement should be package statement [if it

is available] otherwise we will get compile time error.

Example:

```
import java.util.*;
package pack1;
class A
{
}
```

Output:

Compile time error.

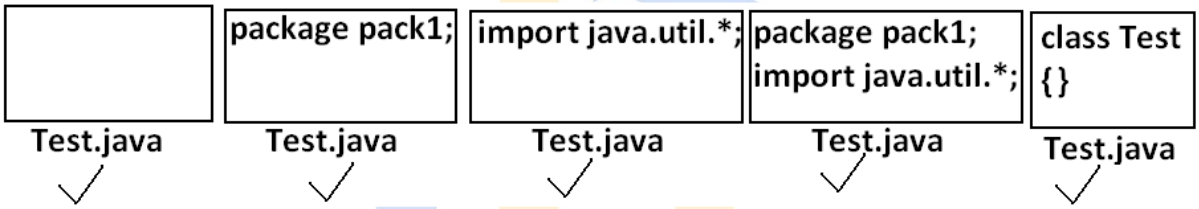
```
D:\Java>javac A.java
```

A.java:2: class, interface, or enum expected

package pack1;

Java source file structure:

All the following are valid java Programs.



Note: An empty source file is a valid java Program.

Class Modifiers

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

Like

1. Whether this class can be accessible from anywhere or not.
2. Whether child class creation is possible or not.
3. Whether object creation is possible or not etc.

We can specify this information by using the corresponding modifiers.

The only applicable modifiers for Top Level classes are:

1. Public
2. Default
3. Final
4. Abstract
5. Strictfp

If we are using any other modifier we will get compile time error.

Example:

```
private class Test
{
public static void main(String args[]){
int i=0;
for(int j=0;j<3;j++)
{
i=i+j;
}
System.out.println(i);
}}
```

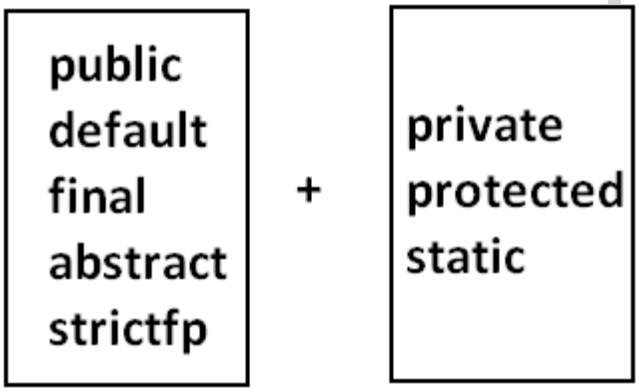
OUTPUT:

Compile time error.

```
D:\Java>javac Test.java
Test.java:1: modifier private not allowed here
private class Test
```

But For the inner classes the following modifiers are allowed.

Diagram:



What is the difference between access specifier and access modifier ?

 In old languages 'C' (or) 'C++' public, private, protected, default are considered as access specifiers and all the remaining are considered as access modifiers.

 But in java there is no such type of division all are considered as access modifiers.

Public Classes:

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

package or outside the package.

Example:

Program1:

```
package pack1;
public class Test
{
public void methodOne(){
System.out.println("test class methodone is executed");
}}
```

Compile the above Program:

```
D:\Java>javac -d . Test.java
```

Program2:

```
package pack2;
import pack1.Test;
class Test1
{
public static void main(String args[]){
Test t=new Test();
t.methodOne();
}}
```

OUTPUT:

D:\Java>javac -d . Test1.java

D:\Java>java pack2.Test1

Test class methodone is executed.

If class Test is not public then while compiling Test1 class we will get compile time error

saying pack1.Test is not public in pack1; cannot be accessed from outside package.

Default Classes:

If a class declared as the default then we can access that class only within the current

package hence default access is also known as "package level access".

Example:

Program 1:

```
package pack1;
class Test
{
public void methodOne(){
System.out.println("test class methodone is executed");
}}
```

Program 2:

```
package pack1;
import pack1.Test;
class Test1
{
public static void main(String args[]){
Test t=new Test();
```

t.methodOne();

}}

OUTPUT:

D:\Java>javac -d . Test.java

D:\Java>javac -d . Test1.java

D:\Java>java pack1.Test1

Test class methodone is executed

Final Modifier:

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

Final Methods:

 Whatever the methods parent has by default available to the child.

 If the child is not allowed to override any method, that method we have to declare with final in parent class. That is final methods cannot overridden.

Example:

Program 1:

```
class Parent
{
public void property(){
System.out.println("cash+gold+land");
}
public final void marriage(){
System.out.println("subbalakshmi");
}}
```

Program 2:

```
class child extends Parent
{
public void marriage(){
```

System.out.println("Thamanna");

}}

OUTPUT:

Compile time error.

D:\Java>javac Parent.java

D:\Java>javac child.java

child.java:3: marriage() in child cannot override marriage() in Parent;

overridden method is final

public void marriage(){

Final Class:

If a class declared as the final then we cann't creates the child class that is inheritance

concept is not applicable for final classes.

Example:

Program 1:

final class Parent

{

}

Program 2:

class child extends Parent

{

}

OUTPUT:

Compile time error.

D:\Java>javac Parent.java

D:\Java>javac child.java

child.java:1: cannot inherit from final Parent

class child extends Parent

Note: Every method present inside a final class is always final by default whether we are

declaring or not. But every variable present inside a final class need not be final.

Example:

```
final class parent
{
static int x=10;
static
{
x=999;
}}
```

The main advantage of final keyword is we can achieve security.

Whereas the main disadvantage is we are missing the key benefits of oops: polymorsim (because of final methods), inheritance (because of final classes) hence if

there is no specific requirement never recommended to use final keyboard.

Abstract Modifier:

Abstract is the modifier applicable only for methods and classes but not for variables.

Abstract Methods:

Even though we don't have implementation still we can declare a method with abstract

modifier.

That is abstract methods have only declaration but not implementation.

Hence abstract method declaration should compulsory ends with semicolon.

Example:

public abstract void methodOne(); ⟶ valid
public abstract void methodOne(){} ⟶ invalid

Child classes are responsible to provide implementation for parent class abstract
methods.

Example:

Program:

```
abstract class vehicle
{
public abstract int getNoOfWheels();
}
```

vehicle.java

```
class Bus extends vehicle
{
public int getNoOfWheels(){
return 7;
}
}
```

Bus.java

```
class Auto extends vehicle
{
public int getNoOfWheels(){
return 3;
}
}
```

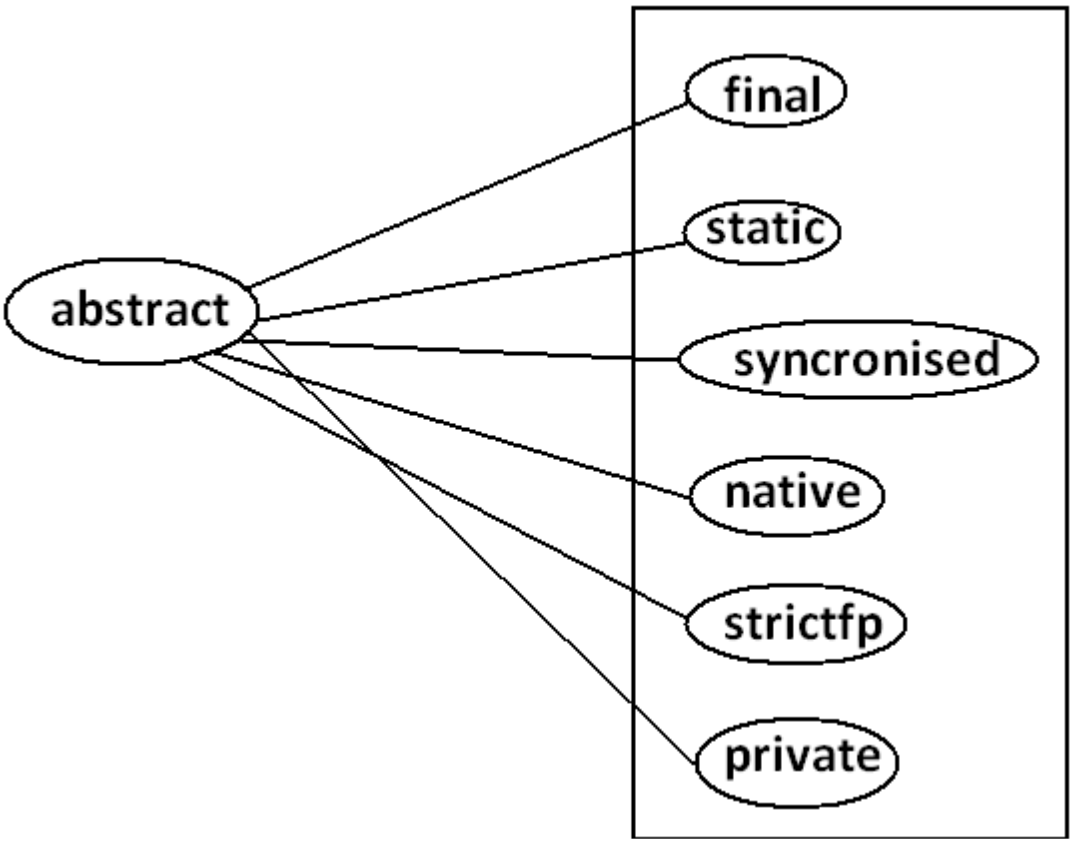
Auto.java

 The main advantage of abstract methods is , by declaring abstract method in parent class we can provide guide lines to the child class such that which methods they should compulsory implement.

 Abstract method never talks about implementation whereas if any modifier talks
about implementation then the modifier will be enemy to abstract and that is always illegal combination for methods.

The following are the various illegal combinations for methods.

Diagram:



All the 6 combinations are illegal.

For any java class if we are not allow to create an object such type of class we have to

declare with abstract modifier that is for abstract class instantiation is not possible.

Example:

```
abstract class Test
{
```

```
public static void main(String args[]){
Test t=new Test();
}}
```

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:4: Test is abstract; cannot be instantiated
Test t=new Test();
```

What is the difference between abstract class and abstract method ?

 If a class contain at least on abstract method then compulsory the corresponding class should be declare with abstract modifier. Because implementation is not complete and hence we can't create object of that class.

 Even though class doesn't contain any abstract methods still we can declare the class as abstract that is an abstract class can contain zero no of abstract methods also.

Example1: HttpServlet class is abstract but it doesn't contain any abstract method.

Example2: Every adapter class is abstract but it doesn't contain any abstract method.

Example1:

```
class Parent
{
public void methodOne();
}
```

Output:

Compile time error.

D:\Java>javac Parent.java

Parent.java:3: missing method body, or declare abstract

public void methodOne();

Example2:

```
class Parent
{
public abstract void methodOne(){}
}
```

Output:

Compile time error.

Parent.java:3: abstract methods cannot have a body

public abstract void methodOne(){}

Example3:

```
class Parent
{
public abstract void methodOne();
}
```

Output:

Compile time error.

D:\Java>javac Parent.java

Parent.java:1: Parent is not abstract and does not

override abstract method methodOne() in Parent

class Parent

If a class extends any abstract class then compulsory we should provide implementation

for every abstract method of the parent class otherwise we have to declare child class as

abstract.

Example:

```
abstract class Parent
{
public abstract void methodOne();
public abstract void methodTwo();
}
class child extends Parent
{
public void methodOne(){}
}
```

Output:

Compile time error.

```
D:\Java>javac Parent.java
Parent.java:6: child is not abstract and does not
override abstract method methodTwo() in Parent
class child extends Parent
```

If we declare class child as abstract then the code compiles fine but child of child is

responsible to provide implementation for methodTwo().

What is the difference between final and abstract ?

 For abstract methods compulsory we should override in the child class to provide implementation. Whereas for final methods we can't override hence abstract final combination is illegal for methods.

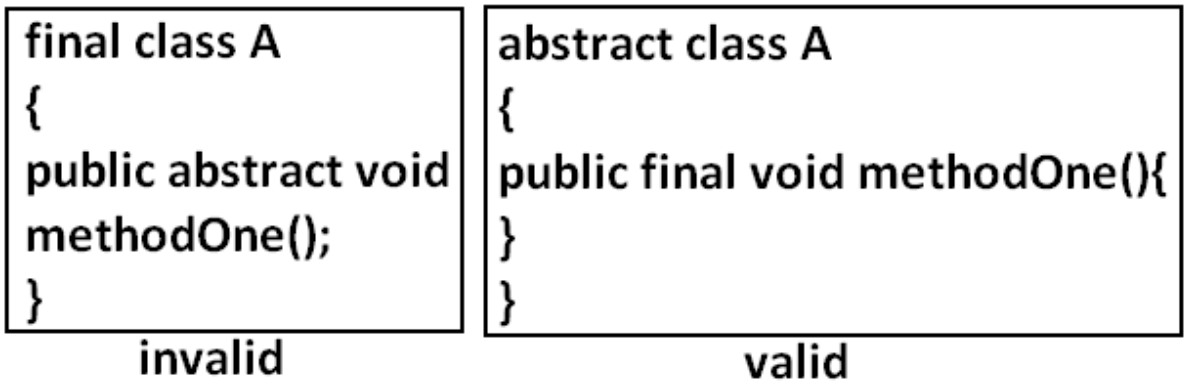
 For abstract classes we should compulsory create child class to provide

implementation whereas for final class we can't create child class. Hence final abstract combination is illegal for classes.

 Final class cannot contain abstract methods whereas abstract class can contain

final method.

Example:



Note:

Usage of abstract methods, abstract classes and interfaces is always good Programming

practice.

Strictfp:

 strictfp is the modifier applicable for methods and classes but not for variables.

 Strictfp modifier introduced in 1.2 versions.

 Usually the result of floating point of arithmetic is varing from platform to platform , to overcome this problem we should use strictfp modifier.

 If a method declare as the Strictfp then all the floating point calculations in that

method has to follow IEEE754 standard, So that we will get platform independent results.

Example:

```
System.out.println(10.0/3);
```

P4	P3	IEEE754
3.33333333333333	3.333333	3.333

If a class declares as the Strictfp then every concrete method(which has body) of that

class has to follow IEEE754 standard for floating point arithmetic, so we will get platform independent results.

What is the difference between abstract and strictfp ?

 Strictfp method talks about implementation where as abstract method never talks about implementation hence abstract, strictfp combination is illegal for methods.

 But we can declare a class with abstract and strictfp modifier simultaneously. That is abstract strictfp combination is legal for classes but illegal for methods.

Example:

```
public abstract strictfp void methodOne(); (invalid)

abstract strictfp class Test  (valid)
{
}
```

Member modifiers:

Public members:

If a member declared as the public then we can access that member from anywhere

"but the corresponding class must be visible" hence before checking member visibility

we have to check class visibility.

Example:

Program 1:

```
package pack1;
class A
{
public void methodOne(){
System.out.println("a class method");
}}
```

D:\Java>javac -d . A.java

Program 2:

```
package pack2;
import pack1.A;
class B
{
public static void main(String args[]){
A a=new A();
a.methodOne();

}}
```

Output:

Compile time error.

D:\Java>javac -d . B.java

B.java:2: pack1.A is not public in pack1;

cannot be accessed from outside package

import pack1.A;

In the above Program even though methodOne() method is public we can't access from
class B because the corresponding class A is not public that is both classes and methods
are public then only we can access.

If a member declared as the default then we can access that member only within the
current package hence default member is also known as package level access.

Example 1:

Program 1:

```
package pack1;
class A
{
void methodOne(){
System.out.println("methodOne is executed");
}}
```

Program 2:

```
package pack1;
import pack1.A;
class B
{
public static void main(String args[]){
A a=new A();
a.methodOne();
}}
```

Output:

D:\Java>javac -d . A.java

D:\Java>javac -d . B.java

D:\Java>java pack1.B

methodOne is executed

Example 2:

Program 1:

```
package pack1;
class A
{
void methodOne(){
System.out.println("methodOne is executed");
}}
```

Program 2:

```
package pack2;
import pack1.A;
class B
{

public static void main(String args[]){
A a=new A();
a.methodOne();
}}
```

Output:

Compile time error.

D:\Java>javac -d . A.java

D:\Java>javac -d . B.java

B.java:2: pack1.A is not public in pack1; cannot be accessed from outside package

import pack1.A;

Private members:

 If a member declared as the private then we can access that member only with in

the current class.

 Private methods are not visible in child classes where as abstract methods should

be visible in child classes to provide implementation hence private, abstract combination is illegal for methods.

Protected members:

 If a member declared as the protected then we can access that member within

the current package anywhere but outside package only in child classes.

Protected=default+kids.

 We can access protected members within the current package anywhere either

by child reference or by parent reference

 But from outside package we can access protected members only in child classes

and should be by child reference only that is we can't use parent reference to call

protected members from outside package.

Example:

Program 1:

```
package pack1;

public class A

{
```

```
protected void methodOne(){
System.out.println("methodOne is executed");
}}
```

Program 2:

```
package pack1;
class B extends A
{
public static void main(String args[]){
A a=new A();
a.methodOne();
B b=new B();
b.methodOne();
A a1=new B();
a1.methodOne();
}}
```

Output:

```
D:\Java>javac -d . A.java
D:\Java>javac -d . B.java
D:\Java>java pack1.B
methodOne is executed
methodOne is executed
methodOne is executed
```

Example 2:

```
package pack2;
import pack1.A;
public class C extends A
{
public static void main(String args[]){
A a=new A();
a.methodOne();
C c=new C();
c.methodOne();
A a1=new B();
a1.methodOne();
}
}
```

output:
compile time error.
D:\Java>javac -d . C.java
C.java:7: methodOne() has protected access in pack1.A
a.methodOne();

 The least accessible modifier is private.

 The most accessible modifier is public.

Private<default<protected<public

Recommended modifier for variables is private where as recommended modifier for

methods is public.

Final variables:

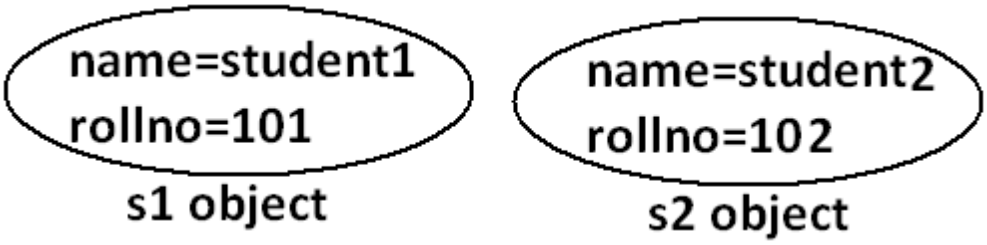
Final instance variables:

 If the value of a variable is varied from object to object such type of variables are

called instance variables.

 For every object a separate copy of instance variables will be created.

DIAGRAM:



For the instance variables it is not required to perform initialization explicitly jvm will

always provide default values.

Example:

```
class Test
{
int i;
public static void main(String args[]){
Test t=new Test();
System.out.println(t.i);
}}
```

Output:

```
D:\Java>javac Test.java
D:\Java>java Test
0
```

If the instance variable declared as the final compulsory we should perform initialization explicitly and JVM won't provide any default values.

whether we are using or not otherwise we will get compile time error.

Example:

Program 1:

```
class Test
{
int i;
}
```

Output:

```
D:\Java>javac Test.java
D:\Java>
```

Program 2:

```
class Test
{
final int i;
}
```

Output:

```
Compile time error.
D:\Java>javac Test.java
Test.java:1: variable i might not have been initialized
class Test
```

Rule:

For the final instance variables we should perform initialization before constructor completion. That is the following are various possible places for this.

1) At the time of declaration:

Example:

```
class Test
{
final int i=10;
```

```
}
```

Output:

```
D:\Java>javac Test.java

D:\Java>
```

2) Inside instance block:

Example:

```
class Test
{
final int i;
{
i=10;
}}
```

Output:

```
D:\Java>javac Test.java
D:\Java>
```

3) Inside constructor:

Example:

```
class Test
{
final int i;
Test()
{
i=10;
}}
```

Output:

```
D:\Java>javac Test.java
D:\Java>
```

If we are performing initialization anywhere else we will get compile time error.

Example:

```
class Test
{
final int i;
public void methodOne(){
i=10;
}}
```

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:5: cannot assign a value to final variable i
i=10;
```

Final static variables:

 If the value of a variable is not varied from object to object such type of variables

is not recommended to declare as the instance variables. We have to declare those variables at class level by using static modifier.

 In the case of instance variables for every object a seperate copy will be created

but in the case of static variables a single copy will be created at class level and shared by every object of that class.

 For the static variables it is not required to perform initialization explicitly jvm will always provide default values.

Example:

```
class Test
{
```

```
static int i;
public static void main(String args[]){
System.out.println("value of i is :"+i);
}}
```

Output:

```
D:\Java>javac Test.java
D:\Java>java Test
Value of i is: 0
```

If the static variable declare as final then compulsory we should perform initialization explicitly whether we are using or not otherwise we will get compile time error.(The JVM won't provide any default values)

Example:

```
class Test
{
static int i;
}
```

valid

```
class Test
{
final static int i;
}
```

invalid

```
output:
D:\Java>javac Test.java
Test.java:1: variable i might not have been initialized
class Test
```

Rule:

For the final static variables we should perform initialization before class loading completion otherwise we will get compile time error. That is the following are possible

places.

1) At the time of declaration:

Example:

```
class Test
{
final static int i=10;
}
```

Output:

```
D:\Java>javac Test.java
D:\Java>
```

2) Inside static block:

Example:

```
class Test
{
final static int i;
static
{
i=10;
}}
```

Output:

Compile successfully.

If we are performing initialization anywhere else we will get compile time error.

Example:

```
class Test
{

final static int i;
public static void main(String args[]){
```

```
i=10;
}}
```

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:5: cannot assign a value to final variable i
i=10;
```

Final local variables:

 To meet temporary requirement of the Programmer sometime we can declare the variable inside a method or block or constructor such type of variables are called local variables.

 For the local variables jvm won't provide any default value compulsory we should perform initialization explicitly before using that variable.

Example:

```
class Test
{
public static void main(String args[]){
int i;
System.out.println("hello");
}}
```

Output:

```
D:\Java>javac Test.java
D:\Java>java Test
Hello
```

Example:

```
class Test
```

```
{
public static void main(String args[]){
int i;
System.out.println(i);
}}
```

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:5: variable i might not have been initialized
System.out.println(i);
```

Even though local variable declared as the final before using only we should perform initialization.

Example:

```
class Test
{
public static void main(String args[]){
final int i;
System.out.println("hello");
}}
```

Output:

```
D:\Java>javac Test.java

D:\Java>java Test
hello
```

Note: The only applicable modifier for local variables is final if we are using any other modifier we will get compile time error.

Example:

```
class Test
{
public static void main(String args[])
{
private int x=10;————(invalid)
public int x=10; ————(invalid)
volatile int x=10;————(invalid)
transient int x=10; ——— (invalid)
final int x=10;————(valid)
}
}
```

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:5: illegal start of expression

private int x=10;

 For instance and static variables JVM will provide default values but if instance and static declared as final JVM won't provide default value compulsory we should perform initialization whether we are using or not .

 For the local variables JVM won't provide any default values we have to perform explicitly before using that variables , this rule is same whether local

variable final or not.

 Static is the modifier applicable for methods, variables and blocks.

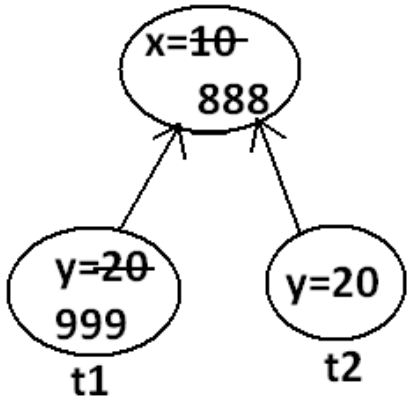
 We can't declare a class with static but inner classes can be declaring as the static.

 In the case of instance variables for every object a separate copy will be created

but in the case of static variables a single copy will be created at class level and shared by all objects of that class..

Example:

```
class Test{
static int x=10;
int y=20;
public static void main(String args[]){
Test t1=new Test();
t1.x=888;
t1.y=999;
Test t2=new Test();
System.out.println(t2.x+"....."+t2.y);
}
}
```



Output:

D:\Java>javac Test.java

D:\Java>java Test

888.....20

 Instance variables can be accessed only from instance area directly and we can't

access from static area directly.

 But static variables can be accessed from both instance and static areas directly.

1) Int x=10;

2) Static int x=10;

3) Public void methodOne(){

System.out.println(x);

}

4) Public static void methodOne(){

System.out.println(x);

}

Which are the following declarations are allow within the same class simultaneously ?

a) 1 and 3

Example:

```
class Test
{
int x=10;
public void methodOne(){
System.out.println(x);
}}
```

Output:

Compile successfully.

b) 1 and 4

Example:

```
class Test
{
int x=10;
```

```
public static void methodOne(){
System.out.println(x);
}}
```

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:5: non-static variable x cannot be referenced from a static
context
System.out.println(x);
```

c) 2 and 3

Example:

```
class Test
{
static int x=10;
public void methodOne(){
System.out.println(x);
}}
```

Output:

Compile successfully.

d) 2 and 4

Example:

```
class Test
{
static int x=10;
public static void methodOne(){
System.out.println(x);
}}
```

Output:

Compile successfully.

e) 1 and 2

Example:

```
class Test
{
int x=10;
static int x=10;
}
```

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:4: x is already defined in Test
static int x=10;
```

f) 3 and 4

Example:

```
class Test{
public void methodOne(){
System.out.println(x);
}
public static void methodOne(){
System.out.println(x);
}}
```

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:5: methodOne() is already defined in Test
```

public static void methodOne(){

For static methods implementation should be available but for abstract methods

implementation is not available hence static abstract combination is illegal for methods.

case 1:

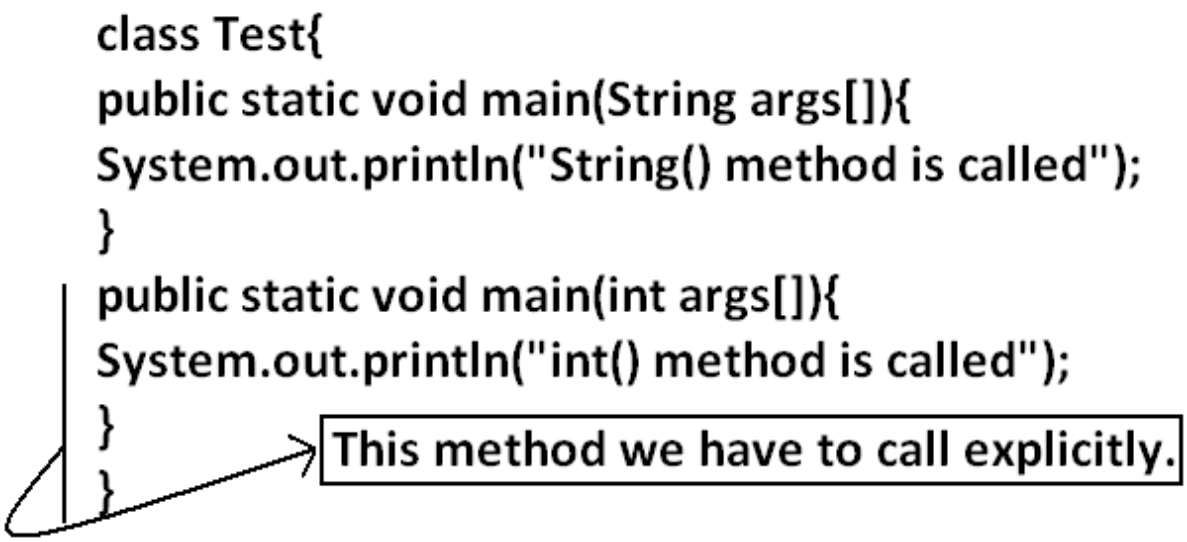
Overloading concept is applicable for static method including main method also.But

JVM will always call String[] args main method .

The other overloaded method we have to call explicitly then it will be executed just like

a normal method call .

Example:



Output :

String() method is called

case 2:

Inheritance concept is applicable for static methods including main() method hence

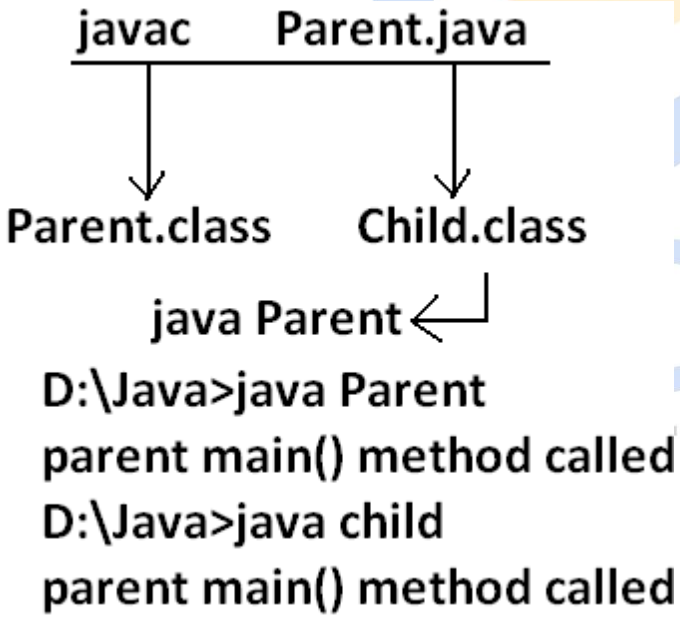
while executing child class, if the child doesn't contain main() method then the parent

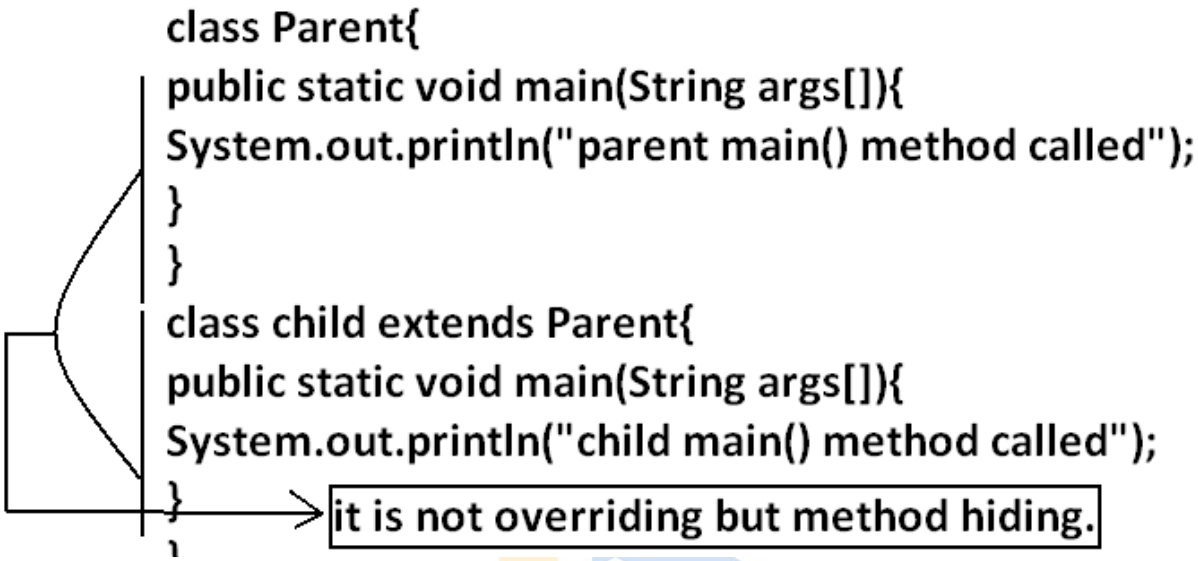
class main method will be executed.

Example:

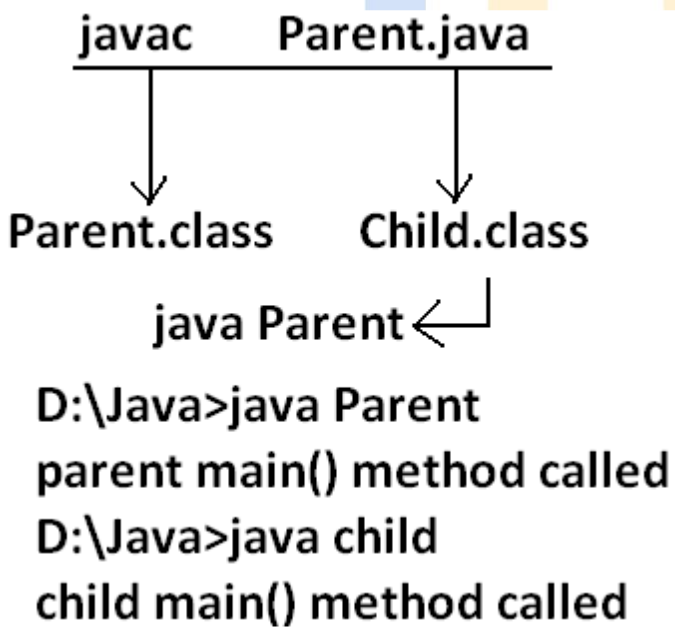
```
class Parent{
public static void main(String args[]){
System.out.println("parent main() method called");
}
}
class child extends Parent{
}
```

Output:





Output:



 It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.

Native modifier:

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

 The methods which are implemented in non java are called native methods or foreign methods.

The main objectives of native keyword are:

 To improve performance of the system.

 To use already existing legacy non-java code.

 To achieve machine level communication(memory level - address)

 Pseudo code to use native keyword in java.

For native methods implementation is already available and we are not responsible to
provide implementation hence native method declaration should compulsory ends with
semicolon.

 For native methods implementation is already available where as for abstract methods implementation should not be available child class is responsible to provide that, hence abstract native combination is illegal for methods.

 We can't declare a native method as strictfp because there is no guaranty whether the old language supports IEEE754 standard or not. That is native strictfp combination is illegal for methods.

 For native methods inheritance, overriding and overloading concepts are applicable.

 The main advantage of native keyword is performence will be improves.

 The main disadvantage of native keyword is usage of native keyword in java breaks platform independent nature of java language.

Transient modifier:

1. Transient is the modifier applicable only for variables but not for methods and

classes.

2. At the time of serialization if we don't want to serialize the value of a particular

variable to meet the security constraints then we should declare that variable with transient modifier.

3. At the time of serialization jvm ignores the original value of the transient variable and save default value that is transient means "not to serialize".

4. Static variables are not part of object state hence serialization concept is not applicable for static variables duo to this declaring a static variable as transient there is no use.

5. Final variables will be participated into serialization directly by their values due

to this declaring a final variable as transient there is no impact.

Summary of modifier:

t

Java Modifiers Important Questions

1) For the Top Level Classes which Modifiers are Allowed?

2) Is it Possible to Declare a Class as static, private, and protected?

3) What are Extra Modifiers Applicable for Inner Classes when compared with Outer

Classes?

4) What is a final Class?

5) Explain the Differences between final, finally and finalize?

6) Is Every Method Present in final Class is final?

7) Is Every Variable Present Inside a final Class is final?

8) What is abstract Class?

9) What is abstract Method?

10) If a Class contain at least One abstract Method is it required to declared that Class

Compulsory abstract?

11) If a Class doesn't contain any abstract Methods is it Possible to Declare that Class as

abstract?

12) Whenever we are extending abstract Class is it Compulsory required to Provide

Implementation for Every abstract Method of that Class?

13) Is final Class can contain abstract Method?

14) Is abstract Class can contain final Methods?

15) Can You give Example for abstract Class which doesn't contain any abstract Method?

16) Which of the following Modifiers Combinations are Legal for Methods?

- public – static
- static – abstract
- abstract - final
- final - synchronized
- synchronized - native
- native – abstract

17) Which of the following Modifiers Combinations are Legal for Classes?

- public - final
- final - abstract
- abstract - strictfp

 strictfp – public

18) What is the Difference between abstract Class and Interface?

19) What is strictfp Modifier?

20) Is it Possible to Declare a Variable with strictfp?

21) abstract - strictfp Combination, is Legal for Classes OR Methods?

22) Is it Possible to Override a native Method?

23) What is the Difference between Instance and Static Variable?

24) What is the Difference between General Static Variable and final Static Variable?

25) Which Modifiers are Applicable for Local Variable?

26) When the Static Variables will be Created?

27) What are Various Memory Locations of Instance Variables, Local Variables and

Static Variables?

28) Is it Possible to Overload a main()?

29) Is it Possible to Override Static Methods?

30) What is native Key Word and where it is Applicable?

31) What is the Main Advantage of the native Key Word?

32) If we are using native Modifier how we can Maintain Platform Independent Nature?

33) How we can Declare a native Method?

34) Is abstract Method can contain Body?

35) What is synchronized Key Word where we can Apply?

36) What are Advantages and Disadvantages of synchronized Key Word?

37) Which Modifiers are the Most Dangerous in Java?

38) What is Serialization and Explain how its Process?

39) What is Deserialization?

40) By using which Classes we can Achieve Serialization and Deserialization?

41) What is Serializable interface and Explain its Methods?

42) What is a Marker Interface and give an Example?

43) Without having any Method in Serializable Interface, how we can get Serializable

Ability for Our Object?

44) What is the Purpose of transient Key Word and Explain its Advantages?

45) Is it Possible to Serialize Every Java Object?

46) Is it Possible to Declare a Method, a Class with transient?

47) If we Declare Static Variable with transient is there any Impact?

48) What is the Impact of declaring final Variable a transient?

49) What is volatile Variable?

50) Is it Possible to Declare a Class OR a Method with volatile?

51) What is the Advantage and Disadvantage of volatile Modifier?

Note :

1. The modifiers which are applicable for inner classes but not for outer classes are

private, protected, static.

2. The modifiers which are applicable only for methods native.

3. The modifiers which are applicable only for variables transient and volatile.

4. The modifiers which are applicable for constructor public, private, protected, default.

5. The only applicable modifier for local variables is final.

6. The modifiers which are applicable for classes but not for enums are final , abstract.

7. The modifiers which are applicable for classes but not for interface are final.

50, Lakhanpur Housing Society, Near Lucky Restaurant, Vikas Nagar 208024
Contact: 9794687277, 8707276645
Web: www.bytecode.co.in, Email: bytecodeitsolutions@gmail.Com