Q1) private class Test{  
}

Q2) For outer class, modifiers allowed are: public, default, strictfp, abstract, final

For inner class, modifiers used are: public, default, protected, private, strictfp, abstract, final, static

Q3) public abstract void m1{}

o/p: CE

Q4) Abstract keyword cannot be used with final, static, synchronized, native, strictfp or private keyword.

**abstract** **class** A {

**protected** **abstract** **void** main();

}

HttpServlet does not contain any abstract method, still it is declatred abstract. If in a class, even a single method is abstract, declare the class also as abstract.

Q5) abstract final class A{

}

CE: abstract methods need to be overridden but final class/method cannot be overridden

Strictfp: used for methods and classes, not for variables.If strictfp, all the floating point calculation in that method has to follow IEEE754 standard. If class is strictfp, all concrete methods in the class has to follow IEEE754 standard to make it platform independent. Hence abstract strictfp is not valid for methods, but valid for classes.

Protected: default+ child classes.

If within the same package, protected can be accessed by both child as well as parent reference. Whereas if outside the package, we need only child reference.Parent reference won’t work. Even only the reference of parent won’t work.

Q6)

package Day2;

public class Date extends Test2{

public static void main(String args[]){

Test2 obj = new Date();

obj.setA(5);

System.out.println(obj.a);

}

}

package Day1;

public class Test2{

protected int a;

protected void setA(int a){

if(a>0)

this.a=a;

System.out.println(this.a);

}

}

CE

Q) class Test{

final int x;

}

CE

For final variables, instantiation should be done before construction of an object.

Q) package Day1;

public class Test2{

private final int x;

{ x=10;

}

Test2(){

System.out.println(x);

}

public static void main(String args[]){

Test2 obj= new Test2();

}

}

o/p: 10

Instance block is also called before constructor.

**p**ublic class Test2{

private static int x;

{ System.out.println("In instance bliock");

x=10;

}

static{

System.out.println("In static bliock");

x=15;

}

Test2(){

System.out.println("In constructor");

}

public static void main(String args[]){

System.out.println("main called");

Test2 obj= new Test2();

System.out.println("main ended!");

System.out.println(obj.x);

}

}

o/p: In static bliock

main called

In instance bliock

In constructor

main ended!

10

Q) class Test{

final int x;

{

x=10;

}}

No error

Q) class Test{

final int x;

Test{

x=10;}}

No error

For static variables, we don’t need to initialize them explicitly, but if static final, we must else CE. For local variables, even if not final, initialization shud be performed before they are used.

Q) class Test{

static int x;

}

o/p Compiles fine

Q) Class Test{

final static int x;

}

O/p: CE

Q) class Test{

Public void main(){  
int x;

Sysout(“hello”);

}}

o/p: Compiles fine

Q) Q) class Test{

Public void main(){  
int x;

Sysout(x);

}}

o/p: CE, x may not hv been initialized.

For local variables, only final modifier is allowed else CE. If we declare formal parameter of a method as final, we cannot assign it a value, else CE.

Q) class Test{

p.s.v.main(String args[]){

m(10,20);

}

Private void m(int x, final int y){

x=10;

y=100;

}}

CE

Static: can be used with variables, methods and inner classes. In a static method, we can’t access instance members, else CE. Inheritance concept is valid for static methods including main, hence while executing child class, if it does not have a main method, parent class main method will be called. Static method don’t override but they hide the parent method.

Q) class P{

Main(){

Sysout(“parent class”);}}

Class C extends P{

}

Javac p.java java P: parent class

Javac C.java, java C:

Run c class: o/p: parent class.

Native modifier: Only for methods, not for classes or variables.These can be implemented in other languages, hence called foreign methods. For native methods, implementation is already provided in other language, hence we don’t provide body. Just a semi-colon. But use of native methods, make it platform dependent.

Q) class Test{

public native void m1(){

}}

CE: native method cannot have a body.

Q) package Day1;

public class Test2{

public static void main(String args[]){

}

native void m1(); //can be private too.

}

class T1{

public static void main(String args[]){

Test2 obj= new Test2();

obj.m1();

}

}

Synchronized: Only for methods and blocks. Not for classes/ variables.

Transient: Only to variables. Not for method/ classes. At the time of serialization, value is not saved. At time of serialization, default value will be saved for the transient variable.

Volatile: Only for variables. For every thread, a separate local copy is created for a thread. Just before thread terminates, it updates the master copy with it’s value. Final and volatile keywords are not used together as for final value should not change but we use volatile to track changes.

Only variables can be transient and volatile.

Only methods can be native & synchronized.

Interfaces: Contract b/w client and service provider. Achieve security as implementation is hidden. Since interface methods are by-default public and abstract, the class implementing these methods should also declare them as public. Else CE. A class can extend 1 class, An interface can extend any number of interfaces. An interface method cannot be final, private, protected, native, synchronized, strictfp. Every interface variable is public, static, final. These cannot be private, protected, default, transient, volatile. For variables, initialization should be performed at the time of declaration else CE. In implementation classes, we can access these variables but cannot change values.

Q) public class Test2{

void m1(){}

}

class T1 extends Test2{

@Override

private void m1(){}

}

o/p: CE

Q) class X implements Y extends Z{  
}

o/p: CE

Q) interface X{

int x;

{

x=10;

}}

o/p: CE

Q) class Test2 implements Y {

public static void main(String args[]){

x=3;}}

interface Y{

static int x=8;

}

o/p: CE

If a class implements 2 interfaces with same method and diff. Return types, u cannot implement them.

class Test2 implements Y,Z {

@Override

public void m1() {

}}

interface Y{

void m1();

}

interface Z{

int m1();

}

o/p CE

Q) class Test2 implements Y,Z {

@Override

public int m1() {

return x;

}

}

interface Y{

int x=99;

}

interface Z{

int x=100;

int m1();

}

o/p: CE, x is ambiguous

Q) class Test2 implements Y,Z {

@Override

public int m1() {

return Y.x;

}

}

interface Y{

int x=99;

}

interface Z{

int x=100;

int m1();

}

o/p: Compiles fine. The naming conflict is removed by using interface name with variable.

Marker Interface: Tag interface/ ability interface: No methods.eg: serializable, cloneable, RandomAccess, SingleThreadModel. If we want to create our own marker interface, jvm will have to be customized.

Adapter class: A class which implements interface and provides empty implementation methods.If we extend adapter class instead of interface, we’ll not hv to implement all methods.

Eg: interface x{

Public void add();

}

Abstract class X{

Public void add(){}

}

Interfaces cannot have constructors, static blocks and initialization blocks. Abstract class can have constructors.