lgsi('geekathon');

# ANSWER GUIDE

JAVA TEST - 09 May 2012

## 1. Which of the following class level (nonlocal) variable declarations will not compile?

A. protected int a;

B. transient int b = 3;

C. private synchronized int e;

D. volatile int d;

**Explanation:**

Option C will not compile; the `synchronized` modifier applies only to methods.

Option A and B will compile because `protected` and `transient` are legal variable modifiers. Option D will compile because `volatile` is a proper variable modifier.

Read More: Declarations & Access Control

## 2. What will be the output of the program?

```
public class ArrayTest
{
    public static void main(String[ ] args)
    {
        float f1[ ], f2[ ];
        f1 = new float[10];
        f2 = f1;
```

```
        System.out.println("f2[0] = " + f2[0]);
    }
}
```

A. It prints f2[0] = 0.0

B. It prints f2[0] = NaN

C. An error at `f2 = f1;` causes compile to fail.

D. It prints the garbage value.

## Explanation:

Option A is correct. When you create an array (`f1 = new float[10];`) the elements are initialises to the default values for the primitive data type (float in this case - 0.0), so`f1` will contain 10 elements each with a value of 0.0. `f2` has been declared but has not been initialised, it has the ability to reference or point to an array but as yet does not point to any array. `f2 = f1;` copies the reference (pointer/memory address) of `f1` into`f2` so now `f2` points at the array pointed to by `f1`.

This means that the values returned by `f2` are the values returned by `f1`. Changes to `f1`are also changes to `f2` because both `f1` and `f2` point to the same array.

Read More: Declarations & Access Control

## 3. Which statement is true?

A. Class B'S constructor is `public`.

B. Class B'S constructor has no arguments.

C. Class B'S constructor includes a call to `this( )`.

D. None of these.

## Explanation:

Option B is correct. `Class B` inherits `Class A`'s constructor which has no arguments.

Option A is wrong. `Class B` inherits `Class A`'s constructor which uses default access.

Option C is wrong. There is just no implicit call to `this( )`.

Read More: Declarations & Access Control

## 4. Which two statements, added independently at beginning of the program, allow the code to compile?

```
/* Missing statements ? */
public class NewTreeSet extends java.util.TreeSet
{
    public static void main(String [] args)
    {
        java.util.TreeSet t = new java.util.TreeSet();
        t.clear();
    }
    public void clear()
    {
        TreeMap m = new TreeMap();
        m.clear();
    }
}
```

1. No statement is required
2. `import java.util.*;`
3. `import.java.util.Tree*;`
4. `import java.util.TreeSet;`
5. `import java.util.TreeMap;`

A. 1 only

B. 2 and 5

C. 3 and 4

D. 3 and 4

## Explanation:

(2) and (5). `TreeMap` is the only class that must be imported. `TreeSet` does not need an import statement because it is described with a fully qualified name.

(1) is incorrect because `TreeMap` must be imported. (3) is incorrect syntax for an import statement. (4) is incorrect because it will not import `TreeMap`, which is required.

Read More: Declarations & Access Control

## 5. What will be the output of the program?

```
class Two
{
    byte x;
}

class PassO
{
    public static void main(String [] args)
    {
        PassO p = new PassO();
        p.start();
    }

    void start()
    {
        Two t = new Two();
        System.out.print(t.x + " ");
        Two t2 = fix(t);
        System.out.println(t.x + " " + t2.x);
    }

    Two fix(Two tt)
    {
        tt.x = 42;
        return tt;
    }
}
```

A. null null 42

B. 0 0 42

C. 0 42 42

D. 0 0 0

## Explanation:

In the `fix()` method, the reference variable `tt` refers to the same object (class `Two`) as the `t` reference variable. Updating `tt.x` in the `fix()` method updates `t.x` (they are one in the same object). Remember also that the instance variable `x` in the `Two` class is initialized to 0.

Read More: Operators & Assignments

## 6. What will be the output of the program?

```
public class If2
{
    static boolean b1, b2;
    public static void main(String [] args)
    {
        int x = 0;
        if ( !b1 ) /* Line 7 */
        {
            if ( !b2 ) /* Line 9 */
            {
                b1 = true;
                x++;
                if ( 5 > 6 )
                {
                    x++;
                }
                if ( !b1 )
                    x = x + 10;
                else if ( b2 = true ) /* Line 19 */
                    x = x + 100;
                else if ( b1 | b2 ) /* Line 21 */
                    x = x + 1000;
            }
        }
        System.out.println(x);
    }
}
```

A. 0

B. 1

C. 101

D. 111

## Explanation:

As instance variables, `b1` and `b2` are initialized to false. The if tests on lines 7 and 9 are successful so `b1` is set to true and `x` is incremented. The next if test to succeed is on line 19 (note that the code is not testing to see if `b2` is true, it is setting `b2` to be true). Since line 19 was successful, subsequent else-if's (line 21) will be skipped.

Read More: Flow Control

## 7. What will be the output of the program?

```
public class If1
{
    static boolean b;
    public static void main(String [] args)
    {
        short hand = 42;
        if ( hand < 50 & !b ) /* Line 7 */
            hand++;
        if ( hand > 50 );     /* Line 9 */
        else if ( hand > 40 )
        {
            hand += 7;
            hand++;
        }
        else
            --hand;
        System.out.println(hand);
    }
}
```

A. 41

B. 42

C. 50

D. 51

## Explanation:

In Java, boolean instance variables are initialized to `false`, so the if test on line 7 is true and hand is incremented. Line 9 is legal syntax, a do nothing statement. The else-if is true so hand has 7 added to it and is then incremented.

Read More: Flow Control

## 8. What will be the output of the program?

```
int i = 0;
while(1)
{
    if(i == 4)
    {
        break;
    }
    ++i;
}
System.out.println("i = " + i);
```

A. i = 0

B. i = 3

C. i = 4

D. Compilation fails.

## Explanation:

Compilation fails because the argument of the `while` loop, the condition, must be of primitive type boolean. In Java, 1 does not represent the true state of a boolean, rather it is seen as an integer.

Read More: Flow Control

## 9. What will be the output of the program?

```
int i = 0, j = 5;
tp: for (;;)
    {
        i++;
        for (;;)
        {
            if(i > --j)
            {
                break tp;
            }
        }
        System.out.println("i =" + i + ", j = " + j);
```

A. i = 1, j = 0

B. i = 1, j = 4

C. i = 3, j = 4

D. Compilation fails.

**Explanation:**

If you examine the code carefully you will notice a missing curly bracket at the end of the code, this would cause the code to fail.

Read More: Flow Control

## 10. What will be the output of the program?

```
int I = 0;
label:
    if (I < 2) {
    System.out.print("I is " + I);
    I++;
    continue label;
}
```

A. I is 0

B. I is 0 I is 1

C. Compilation fails.

D. None of the above

## Explanation:

The code will not compile because a `continue` statement can only occur in a looping construct. If this syntax were legal, the combination of the `continue` and the `if`statements would create a kludgey kind of loop, but the compiler will force you to write cleaner code than this.

Read More: Flow Control

## 11. What will be the output of the program?

```
try
{
    int x = 0;
    int y = 5 / x;
}
catch (Exception e)
{
    System.out.println("Exception");
}
catch (ArithmeticException ae)
{
    System.out.println(" Arithmetic Exception");
}
System.out.println("finished");
```

A. finished

B. Exception

C. Compilation fails.

D. Arithmetic Exception

## Explanation:

Compilation fails because `ArithmeticException` has already been caught.`ArithmeticException` is a subclass of `java.lang.Exception`, by time the`ArithmeticException` has been specified it has already been caught by the`Exception` class.

If `ArithmeticException` appears before `Exception`, then the file will compile. When catching exceptions the more specific exceptions must be listed before the more general (the subclasses must be caught before the superclasses).

### Read More: Exceptions

## 12. Which statement is true?

A. A try statement must have at least one corresponding catch block.

B. Multiple catch statements can catch the same class of exception more than once.

C. An Error that might be thrown in a method must be declared as thrown by that method, or be handled within that method.

D. Except in case of VM shutdown, if a `try` block starts to execute, a corresponding finally block will always start to execute.

## Explanation:

A is wrong. A `try` statement can exist without catch, but it must have a `finally`statement.

B is wrong. A `try` statement executes a block. If a value is thrown and the try statement has one or more catch clauses that can catch it, then control will be transferred to the first such catch clause. If that `catch` block completes normally, then the `try` statement completes normally.

C is wrong. `Exceptions` of type `Error` and `RuntimeException` do not have to be caught, only checked exceptions (`java.lang.Exception`) have to be caught. However, speaking of `Exceptions`, `Exceptions` do not have to be handled in the same method as the throw statement. They can be passed to another method.

If you put a `finally` block after a `try` and its associated `catch` blocks, then once execution enters the `try` block, the code in that `finally` block will definitely be executed except in the following circumstances:

1. An exception arising in the finally block itself.
2. The death of the thread.
3. The use of `System.exit()`
4. Turning off the power to the CPU.

I suppose the last three could be classified as VM shutdown.

Read More: Exceptions

## 13. Which statement is true?

```
class Test1
{
    public int value;
    public int hashCode() { return 42; }
}
class Test2
{
    public int value;
    public int hashcode() { return (int)(value^5); }
}
```

A. `class Test1` will not compile.

B. The `Test1 hashCode()` method is more efficient than the `Test2 hashCode()` method.

C. The `Test1 hashCode()` method is less efficient than the `Test2 hashCode()` method.

D. `class Test2` will not compile.

### Explanation:

The so-called "hashing algorithm" implemented by `class Test1` will always return the same value, 42, which is legal but which will place all of the hash table entries into a single bucket, the most inefficient setup possible.

Option A and D are incorrect because these classes are legal.

Option B is incorrect based on the logic described above.

Read More: Objects & Collections

## 14. What will be the output of the program?

```
public class Foo
{
    Foo()
    {
        System.out.print("foo");
    }

class Bar
{
    Bar()
    {
        System.out.print("bar");
    }
    public void go()
    {
        System.out.print("hi");
    }
} /* class Bar ends */

    public static void main (String [] args)
    {
        Foo f = new Foo();
        f.makeBar();
    }
    void makeBar()
    {
        (new Bar() {}).go();
    }
}/* class Foo ends */
```

A. Compilation fails.

B. An error occurs at runtime.

C. It prints "foobarhi"

D. It prints "barhi"

### Explanation:

Option C is correct because first the `Foo` instance is created, which means the `Foo`constructor runs and prints "foo". Next, the `makeBar()` method is invoked which creates a `Bar`, which means the `Bar` constructor runs and prints "bar", and finally the `go()`method is invoked on the new `Bar` instance, which means the `go()` method prints "hi".

Read More: Inner Classes

## 15. At what point is the Bar object, created on line 6, eligible for garbage collection?

```
class Bar { }
class Test
{
    Bar doBar()
    {
        Bar b = new Bar(); /* Line 6 */
        return b; /* Line 7 */
    }
    public static void main (String args[])
    {
        Test t = new Test();  /* Line 11 */
        Bar newBar = t.doBar();  /* Line 12 */
        System.out.println("newBar");
        newBar = new Bar(); /* Line 14 */
        System.out.println("finishing"); /* Line 15 */
    }
}
```

A. after line 12

B. after line 14

C. after line 7, when `doBar()` completes

D. after line 15, when `main()` completes

### Explanation:

Option B is correct. All references to the `Bar` object created on line 6 are destroyed when a new reference to a new `Bar` object is assigned to the variable `newBar` on line 14. Therefore the `Bar` object, created on line 6, is eligible for garbage collection after line 14.

Option A is wrong. This actually protects the object from garbage collection.

Option C is wrong. Because the reference in the `doBar()` method is returned on line 7 and is stored in `newBar` on line 12. This preserver the object created on line 6.

Option D is wrong. Not applicable because the object is eligible for garbage collection after line 14.

Read More: Garbage Collections

## 16. When is the Demo object eligible for garbage collection?

```
class Test
{
    private Demo d;
    void start()
    {
        d = new Demo();
        this.takeDemo(d); /* Line 7 */
    } /* Line 8 */
    void takeDemo(Demo demo)
    {
        demo = null;
        demo = new Demo();
    }
}
```

A. After line 7

B. After line 8

C. After the `start()` method completes

D. When the instance running this code is made eligible for garbage collection.

### Explanation:

Option D is correct. By a process of elimination.

Option A is wrong. The variable `d` is a member of the `Test` class and is never directly set to null.

Option B is wrong. A copy of the variable `d` is set to null and not the actual variable `d`.

Option C is wrong. The variable `d` exists outside the `start()` method (it is a class member). So, when the `start()` method finishes the variable `d` still holds a reference.

Read More: Garbage Collections

17. After line 8 runs. how many objects are eligible for garbage collection?

```
public class X
{
    public static void main(String [] args)
    {
        X x = new X();
        X x2 = m1(x); /* Line 6 */
        X x4 = new X();
        x2 = x4; /* Line 8 */
        doComplexStuff();
    }
    static X m1(X mx)
    {
        mx = new X();
        return mx;
    }
}
```

A. 0

B. 1

C. 2

D. 3

## Explanation:

By the time line 8 has run, the only object without a reference is the one generated as a result of line 6. Remember that "Java is pass by value," so the reference variable `x` is not affected by the `m1()` method.

Read More: Garbage Collections

## 18. What causes compilation to fail?

```
public class Test
{
    public void foo()
    {
        assert false; /* Line 5 */
        assert false; /* Line 6 */
    }
    public void bar()
    {
        while(true)
        {
            assert false; /* Line 12 */
        }
        assert false;  /* Line 14 */
    }
}
```

A. Line 5

B. Line 6

C. Line 12

D. Line 14

## Explanation:

Option D is correct. Compilation fails because of an unreachable statement at line 14. It is a compile-time error if a statement cannot be executed because it is unreachable. The question is now, why is line 20 unreachable? If it is because of the assert then surely line 6 would also be unreachable. The answer must be something other than assert.

Examine the following:

A while statement can complete normally if and only if at least one of the following is true:

- The `while` statement is reachable and the condition expression is not a constant expression with value true.

-There is a reachable break statement that exits the `while` statement.

The while statement at line 11 is infinite and there is no break statement therefore line 14 is unreachable. You can test this with the following code:

```
public class Test80
{
   public void foo()
   {
      assert false;
      assert false;
   }
   public void bar()
   {
      while(true)
      {
         assert false;
         break;
      }
      assert false;
   }
}
```

Read More: Assertions

## 19. What will be the output of the program?

```
public class SqrtExample
{
    public static void main(String [] args)
    {
        double value = -9.0;
        System.out.println( Math.sqrt(value));
    }
}
```

A. 3.0

B. -3.0

C. NaN

D. Compilation fails.

**Explanation:**

The `sqrt()` method returns `NaN` (not a number) when it's argument is less than zero.

Read More: Java.lang Class

20. What will be the output of the program?

```
String a = "newspaper";
a = a.substring(5,7);
char b = a.charAt(1);
a = a + b;
System.out.println(a);
```

A. apa

B. app

C. apea

D. apep

Read More: Java.lang Class

REMEMBER: PREPARATION IS KEY