



## ANSWER GUIDE

JAVA TEST - 08 May 2012

1. What will be the output of the program?

```
1. class A
2. {
3.     final public int GetResult(int a, int b) { return 0; }
4. }
5. class B extends A
6. {
7.     public int GetResult(int a, int b) {return 1; }
8. }
9. public class Test
10. {
11.     public static void main(String args[])
12.     {
13.         B b = new B();
14.         System.out.println("x = " + b.GetResult(0, 1));
15.     }
16. }
```

A. x = 0

B. x = 1

C. Compilation fails.

D. An exception is thrown at runtime.

**Explanation:** The code doesn't compile because the method `GetResult()` in `class A` is final and so cannot be overridden.

Read More: [Declarations & Access Control](#)

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

```
class SC2
{
    public static void main(String [] args)
    {
        SC2 s = new SC2();
        s.start();
    }

    void start()
    {
        int a = 3;
        int b = 4;
        System.out.print(" " + 7 + 2 + " ");
        System.out.print(a + b);
        System.out.print(" " + a + b + " ");
        System.out.print(foo() + a + b + " ");
        System.out.println(a + b + foo());
    }

    String foo()
    {
        return "foo";
    }
}
```

- A. 9 7 7 foo 7 7foo
- B. 72 34 34 foo34 34foo
- C. 9 7 7 foo34 34foo
- D. 72 7 34 foo34 7foo**

**Explanation:** Because all of these expressions use the **+** operator, there is no precedence to worry about and all of the expressions will be evaluated from left to right. If either operand being evaluated is a String, the **+** operator will concatenate the two operands; if both operands are numeric, the **+** operator will add the two operands.

Read More: [Operators & Assignments](#)

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

```
class BoolArray
{
    boolean [] b = new boolean[3];
    int count = 0;

    void set(boolean [] x, int i)
    {
        x[i] = true;
        ++count;
    }

    public static void main(String [] args)
    {
        BoolArray ba = new BoolArray();
        ba.set(ba.b, 0);
        ba.set(ba.b, 2);
        ba.test();
    }

    void test()
    {
        if ( b[0] && b[1] | b[2] )
            count++;
        if ( b[1] && b[(++count - 2)] )
            count += 7;
        System.out.println("count = " + count);
    }
}
```

- A. count = 0
- B. count = 2
- C. count = 3**
- D. count = 4

**Explanation:** The reference variables **b** and **x** both refer to the same boolean array. **count** is incremented for each call to the **set()** method, and once again when the first if test is **true**.

Because of the `&&` short circuit operator, `count` is not incremented during the second `if` test.

Read More: [Operators & Assignments](#)

4. Which two statements are equivalent?

1.  $3/2$
2.  $3 < 2$
3.  $3 * 4$
4.  $3 << 2$

A. 1 and 2

B. 2 and 3

C. 3 and 4

D. 1 and 4

#### Explanation:

(1) is wrong.  $3/2 = 1$  (integer arithmetic).

(2) is wrong.  $3 < 2 = \text{false}$ .

(3) is correct.  $3 * 4 = 12$ .

(4) is correct.  $3 << 2 = 12$ . In binary 3 is 11, now shift the bits two places to the left and we get 1100 which is 12 in binary ( $3 * 2 * 2$ ).

Read More: [Operators & Assignments](#)

## 5. Choose the logic.

```
public void foo( boolean a, boolean b)
{
    if( a )
    {
        System.out.println("A"); /* Line 5 */
    }
    else if(a && b) /* Line 7 */
    {
        System.out.println( "A && B");
    }
    else /* Line 11 */
    {
        if ( !b )
        {
            System.out.println( "notB" ) ;
        }
        else
        {
            System.out.println( "ELSE" ) ;
        }
    }
}
```

A. 1 and 2

B. 2 and 3

C. 3 and 4

D. 1 and 4

### Explanation:

Option C is correct. The output is "ELSE". Only when a is false do the output lines after 11 get some chance of executing.

Option A is wrong. The output is "A". When a is true, irrespective of the value of b, only the line 5 output will be executed. The condition at line 7 will never be evaluated (when a is true it will always be trapped by the line 12 condition) therefore the output will never be "A && B".

Option B is wrong. The output is "A". When **a** is true, irrespective of the value of **b**, only the line 5 output will be executed.

Option D is wrong. The output is "notB".

Read More: [Flow Control](#)

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

```
Float f = new Float("12");
switch (f)
{
    case 12: System.out.println("Twelve");
    case 0: System.out.println("Zero");
    default: System.out.println("Default");
}
```

A. Zero

B. Twelve

C. Default

D. Compilation fails.

### Explanation:

The **switch** statement can only be supported by integers or variables more "narrow" than an integer i.e. **byte**, **char**, **short**. Here a **Float** wrapper object is used and so the compilation fails.

Read More: [Flow Control](#)

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

```
public class Test
{
    public static void aMethod() throws Exception
    {
        try /* Line 5 */
        {
            throw new Exception(); /* Line 7 */
        }
    }
}
```

```

    }
    finally /* Line 9 */
    {
        System.out.print("finally "); /* Line 11 */
    }
}
public static void main(String args[])
{
    try
    {
        aMethod();
    }
    catch (Exception e) /* Line 20 */
    {
        System.out.print("exception ");
    }
    System.out.print("finished"); /* Line 24 */
}
}

```

- A. finally
- B. exception finished
- C. finally exception finished**
- D. Compilation fails.

### Explanation:

This is what happens:

- (1) The execution of the `try` block (line 5) completes abruptly because of the `throw` statement (line 7).
- (2) The exception cannot be assigned to the parameter of any catch clause of the `try` statement therefore the `finally` block is executed (line 9) and "finally" is output (line 11).
- (3) The `finally` block completes normally, and then the `try` statement completes abruptly because of the `throw` statement (line 7).
- (4) The exception is propagated up the call stack and is caught by the catch in the main method (line 20). This prints "exception".
- (5) Lastly program execution continues, because the exception has been caught, and "finished" is output (line 24).

Read More: [Exceptions](#)

8. Which statement is true for the class `java.util.ArrayList`?

- A. The elements in the collection are ordered
- B. The collection is guaranteed to be immutable.
- C. The elements in the collection are guaranteed to be unique.
- D. The elements in the collection are accessed using a unique key.

**Explanation:**

Yes, always the elements in the collection are ordered.

Read More: [Objects & Collections](#)

9. Which is true about a method-local inner class?

- A. It must be marked `final`.
- B. It can be marked `abstract`.
- C. It can be marked `public`.
- D. It can be marked `static`.

**Explanation:**

Option B is correct because a method-local inner class can be `abstract`, although it means a subclass of the inner class must be created if the abstract class is to be used (so an abstract method-local inner class is probably not useful).

Option A is incorrect because a method-local inner class does not have to be declared `final` (although it is legal to do so).

C and D are incorrect because a method-local inner class cannot be made `public` (remember-you cannot mark any local variables as `public`), or `static`.

Read More: [Inner Classes](#)



10. Which of the following line of code is suitable to start a thread?

```
class X implements Runnable
{
    public static void main(String args[])
    {
        /* Missing code? */
    }
    public void run() {}
}
```

- A. Thread t = new Thread(X);
- B. Thread t = new Thread(X); t.start();
- C. X run = new X(); Thread t = new Thread(run); t.start();
- D. Thread t = new Thread(); x.run()

**Explanation:**

Option C is suitable to start a thread.

Read More: [Threads](#)

11. What will be the output of the program?

```
class MyThread extends Thread
{
    public static void main(String [] args)
    {
        MyThread t = new MyThread();
        t.start();
        System.out.print("one. ");
        t.start();
        System.out.print("two. ");
    }
}
```

```
public void run()
{
    System.out.print("Thread ");
}
}
```

- A. Compilation fails
- B. An exception occurs at runtime.
- C. It prints "Thread one. Thread two."
- D. The output cannot be determined.

### Explanation:

When the `start()` method is attempted a second time on a single Thread object, the method will throw an `IllegalThreadStateException` (you will not need to know this exception name for the exam). Even if the thread has finished running, it is still illegal to call `start()` again.

Read More: [Threads](#)

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

```
class MyThread extends Thread
{
    MyThread() {}
    MyThread(Runnable r) {super(r); }
    public void run()
    {
        System.out.print("Inside Thread ");
    }
}
class MyRunnable implements Runnable
{
    public void run()
    {
        System.out.print(" Inside Runnable");
    }
}
class Test
{
    public static void main(String[] args)
    {
```

```

        new MyThread().start();
        new MyThread(new MyRunnable()).start();
    }
}

```

A. Prints "Inside Thread Inside Thread"

B. Prints "Inside Thread Inside Runnable"

C. Does not compile

D. Throws exception at runtime

### Explanation:

If a Runnable object is passed to the Thread constructor, then the run method of the Thread class will invoke the run method of the Runnable object.

In this case, however, the run method in the Thread class is overridden by the run method in MyThread class. Therefore the run() method in MyRunnable is never invoked.

Both times, the run() method in MyThread is invoked instead.

Read More: [Threads](#)

13. What will be the output of the program?

```

class S implements Runnable
{
    int x, y;
    public void run()
    {
        for(int i = 0; i < 1000; i++)
            synchronized(this)
            {
                x = 12;
                y = 12;
            }
        System.out.print(x + " " + y + " ");
    }
    public static void main(String args[])
    {

```

```
s run = new s();  
Thread t1 = new Thread(run);  
Thread t2 = new Thread(run);  
t1.start();  
t2.start();  
}  
}
```

A. DeadLock

B. It prints 12 12 12 12

C. Compilation Error

D. Cannot determine output

**Explanation:**

The program will execute without any problems and print 12 12 12 12.

Read More: [Threads](#)

14. 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](#)

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

```
public class Test
{
    public static int y;
    public static void foo(int x)
    {
        System.out.print("foo ");
        y = x;
    }
    public static int bar(int z)
    {
        System.out.print("bar ");
        return y = z;
    }
    public static void main(String [] args )
    {
        int t = 0;
        assert t > 0 : bar(7);
        assert t > 1 : foo(8); /* Line 18 */
        System.out.println("done ");
    }
}
```

- A. bar
- B. bar done
- C. foo done

D. compilation fails.

### Explanation:

The `foo()` method returns void. It is a perfectly acceptable method, but because it returns void it cannot be used in an `assert` statement, so line 18 will not compile.

Read More: [Assertions](#)

## 16. Which of the following statements is true?

- A. In an `assert` statement, the expression after the colon ( `:` ) can be any Java expression.

B. If a `switch` block has no default, adding an `assert` default is considered appropriate.

C. In an `assert` statement, if the expression after the colon ( `:` ) does not have a value, the assert's error message will be empty.

D. It is appropriate to handle assertion failures using a `catch` clause.

### Explanation:

Adding an assertion statement to a `switch` statement that previously had no default case is considered an excellent use of the `assert` mechanism.

Option A is incorrect because only Java expressions that return a value can be used. For instance, a method that returns `void` is illegal.

Option C is incorrect because the expression after the colon must have a value.

Option D is incorrect because assertions throw errors and not exceptions, and assertion errors do cause program termination and should not be handled.

Read More: [Assertions](#)

17. which line is an example of an inappropriate use of assertions?

```
public class Test2
{
    public static int x;
    public static int foo(int y)
    {
        return y * 2;
    }
    public static void main(String [] args)
    {
        int z = 5;
        assert z > 0; /* Line 11 */
        assert z > 2: foo(z); /* Line 12 */
        if ( z < 7 )
            assert z > 4; /* Line 14 */

        switch (z)
```

```

    {
        case 4: System.out.println("4 ");
        case 5: System.out.println("5 ");
        default: assert z < 10;
    }

    if ( z < 10 )
        assert z > 4: z++; /* Line 22 */
    System.out.println(z);
}

```

A. Line 11

B. Line 12

C. Line 14

D. Line 22

### Explanation:

Assert statements should not cause side effects. Line 22 changes the value of `z` if the assert statement is `false`.

Option A is fine; a second expression in an assert statement is not required.

Option B is fine because it is perfectly acceptable to call a method with the second expression of an assert statement.

Option C is fine because it is proper to call an assert statement conditionally.

Read More: [Assertions](#)

18. What will be the output of the program?

```

public class NFE
{
    public static void main(String [] args)
    {
        String s = "42";
        try
        {
            s = s.concat(".5"); /* Line 8 */
        }
    }
}

```



```

        double d = Double.parseDouble(s);
        s = Double.toString(d);
        int x = (int) Math.ceil(Double.valueOf(s).doubleValue());
        System.out.println(x);
    }
    catch (NumberFormatException e)
    {
        System.out.println("bad number");
    }
}

```

- A. 42
- B. 42.5
- C. 43**
- D. bad number

### Explanation:

All of this code is legal, and line 8 creates a new String with a value of "42.5". Lines 9 and 10 convert the String to a double and then back again. Line 11 is fun! "Math.ceil()" 's argument expression is evaluated first. We invoke the `valueOf()` method that returns an anonymous Double object (with a value of 42.5). Then the `doubleValue()` method is called (invoked on the newly created Double object), and returns a double primitive (there and back again), with a value of (you guessed it) 42.5. The `ceil()` method converts this to 43.0, which is cast to an int and assigned to x.

Read More: [Java.lang Class](#)

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

```

public class Test138
{
    public static void stringReplace (String text)
    {
        text = text.replace ('j' , 'c'); /* Line 5 */
    }
    public static void bufferReplace (StringBuffer text)
    {
        text = text.append ("c"); /* Line 9 */
    }
    public static void main (String args[])

```

```

{
    String textString = new String ("java");
    StringBuffer textBuffer = new StringBuffer ("java"); /* Line 14 */
    stringReplace(textString);
    bufferReplace(textBuffer);
    System.out.println (textString + textBuffer);
}
}

```

- A. java
- B. javac
- C. javajavac
- D. Compile error

#### Explanation:

A string is immutable, it cannot be changed, that's the reason for the `StringBuffer` class. The `stringReplace` method does not change the string declared on line 14, so this remains set to "java".

Method parameters are always passed by value - a copy is passed into the method - if the copy changes, the original remains intact, line 5 changes the reference i.e. text points to a new `String` object, however this is lost when the method completes. The `textBuffer` is a `StringBuffer` so it can be changed.

This change is carried out on line 9, so "java" becomes "javac", the text reference on line 9 remains unchanged. This gives us the output of "javajavac"

Read More: [Java.lang Class](#)

20. What will be the output of the program (in jdk1.6 or above)?

```

public class BoolTest
{
    public static void main(String [] args)
    {
        Boolean b1 = new Boolean("false");
        boolean b2;
        b2 = b1.booleanValue();
        if (!b2)
        {

```

```
        b2 = true;
        System.out.print("x ");
    }
    if (b1 & b2) /* Line 13 */
    {
        System.out.print("y ");
    }
    System.out.println("z");
}
```

A. z

B. x z

C. y z

D. Compilation fails.

Read More: [Java.lang Class](#)

**REMEMBER: PREPARATION IS KEY**