Top of Form

What will be the result of compiling and executing Test class?

1. public class Test {
2. public static void main(String[] args) {
3. byte b1 = (byte) (127 + 21);
4. System.out.println(b1);
5. }
6. }

* 

**148**

* 

**Compilation error**

* 

**-108**

**(Correct)**

* 

**128**

**Explanation**

**UKOCP53688:**

127 + 21 = 148 (int type: 32-bits) = 00000000 00000000 00000000 10010100

Above binary number is positive, as left most bit is 0.

Same binary number after type-casting to byte (8-bits): 10010100, negative number as left most bit is 1.

There is only one negative number in the option and hence -108 is the correct answer.

Binary 10010100 = -108.

Bottom of Form

Top of Form

Question 2: **Correct**

Question ID: UKOCP49189

Consider below code:

1. public class Test {
2. public static void main(String[] args) {
3. char c = 'Z';
4. long l = 100\_00l;
5. int i = 9\_2;
6. float f = 2.02f;
7. double d = 10\_0.35d;
8. l = c + i;
9. f = c \* l \* i \* f;
10. f = l + i + c;
11. i = (int)d;
12. f = (long)d;
13. }
14. }

Does above code compile successfully?

* 

**Yes**

**(Correct)**

* 

**No**

**Explanation**

**UKOCP49189:**

For readability purpose underscore (\_) is used to separate numeric values. This is very useful in representing big numbers such as credit card numbers (1234\_7654\_9876\_0987). long data can be suffixed by l, float by f and double by d. So first 5 variable declaration and assignment statements inside main(String []) method don't cause any compilation error.

Let's check rest of the statements:

l = c + i; => Left side variable 'l' is of long type and right side expression evaluates to an int value, which can easily be assigned to long type. No compilation error here.

f = c \* l \* i \* f; => Left side variable 'f' is of float type and right side expression evaluates to float value, which can easily be assigned to float type. Hence, it compiles successfully.

f = l + i + c; => Left side variable 'f' is of float type and right side expression evaluates to long value, which can easily be assigned to float type. Hence, no issues here.

i = (int)d; => double can't be assigned to int without explicit casting, right side expression `(int)d;` is casting double to int, so no issues.

f = (long)d; => double can't be assigned to float without explicit casting, right side expression `(long)d;` is casting double to long, which can easily be assigned to float type. It compiles successfully.

Bottom of Form

Top of Form

Question 3: **Correct**

**Question ID: UKOCP56713**

Given code:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. char c1 = 'a'; //ASCII code of 'a' is 97
6. int i1 = c1; //Line n1
7. System.out.println(i1); //Line n2
8. }
9. }

What is the result?

* 

**a**

* 

**97**

**(Correct)**

* 

**Line n1 causes compilation failure**

* 

**Line n1 causes runtime error**

**Explanation**

**UKOCP56713:**

Range of char data type is from 0 to 65535 and hence it can be easily assigned to int type. println() method is overloaded to accept char type and int type both. If char type value is passed, it prints char value and if int type value is passed, it prints int value.

As i1 is of int type, hence corresponding int value, which is 97, is printed on to the console.

Bottom of Form

Top of Form

Question 4: **Correct**

**Question ID: UKOCP79830**

Consider below statements:

1. int x = 5\_\_\_\_0;

2. int y = \_\_\_\_50;

3. int z = 50\_\_\_\_;

4. float f = 123.76\_86f;

5. double d = 1\_2\_3\_4;

How many statements are legal?

* 

**One statement only**

* 

**Two statements only**

* 

**Three statements only**

**(Correct)**

* 

**Four statements only**

* 

**All 5 statements**

**Explanation**

**UKOCP79830:**

For readability purpose underscore (\_) is used to separate numeric values. This is very useful in representing big numbers such as credit card numbers (1234\_7654\_9876\_0987). Multiple underscores are also allowed within the digits. Hence, `int x = 5\_\_\_\_0;` compiles successfully and variable x stores 50.

`float f = 123.76\_86f;` compiles successfully.

1\_2\_3\_4 is int literal 1234 and int can easily be assigned to double, hence `double d = 1\_2\_3\_4;` compiles successfully.

\_\_\_\_50 is a valid variable name, and as this variable is not available hence, int y = \_\_\_\_50; causes compilation error.

Underscores must be available within the digits. For the statement int z = 50\_\_\_\_; as underscores are used after the digits, hence it causes compilation error.

Bottom of Form

Top of Form

Question 5: **Correct**

**Question ID: UKOCP51023**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. byte b1 = 10; //Line n1
6. int i1 = b1; //Line n2
7. byte b2 = i1; //Line n3
8. System.out.println(b1 + i1 + b2);
9. }
10. }

What is the result of compiling and executing Test class?

* 

**Line n1 causes compilation error**

* 

**Line n2 causes compilation error**

* 

**Line n3 causes compilation error**

**(Correct)**

* 

**30 is printed on to the console**

**Explanation**

**UKOCP51023:**

Let us first check Line n1: byte b1 = 10;

Above statement compiles successfully, even though 10 is an int literal (32 bits) and b1 is of byte primitive type which can store only 8 bits of data.

Here java does some background task, if value of int literal can be easily fit to byte primitive type (-128 to 127), then int literal is implicitly casted to byte type.

So above statement is internally converted to:

byte b1 = (byte)10;

But if you specify any out of range value then it would not be allowed, e.g.

byte b = 128; // It would cause compilation failure as 128 is out of range value for byte type.

There is no issue with Line n2 as byte type (8 bits) can be easily assigned to int type (32 bits).

For line n3, `byte b2 = i1;`, expression on right hand side (i1) is neither a within range literal value nor constant expression, hence it causes compilation failure.

To compile successfully, this expression needs to be explicitly casted, such as: `byte b2 = (byte)i1;`

Bottom of Form

Top of Form

Question 6: **Correct**

**Question ID: UKOCP76324**

Range of short data type is from -32768 to 32767

Which of the following code segments, written inside main method will compile successfully?

Select 3 options.

* 

**short s1 = 10;**

**(Correct)**

* 

**short s2 = 32768;**

* 
  1. final int i3 = 10;
  2. short s3 = i3;

**(Correct)**

* 
  1. final int i4 = 40000;
  2. short s4 = i4;
* 
  1. final int i5 = 10;
  2. short s5 = i5 + 100;

**(Correct)**

* 
  1. final int m = 25000;
  2. final int n = 25000;
  3. short s6 = m + n;
* 
  1. int i7 = 10;
  2. short s7 = i7;

**Explanation**

**UKOCP76324:**

Let's check all the statements one by one:

short s1 = 10;

✓ Above statement compiles successfully, even though 10 is an int literal (32 bits) and s1 is of short primitive type which can store only 16 bits of data.

Here java does some background task, if value of int literal can be easily fit to short primitive type (-32768 to 32767), then int literal is implicitly casted to short type.

So above statement is internally converted to:

short s1 = (short)10;

short s2 = 32768;

✗ It causes compilation failure as 32768 is out of range value.

1. final int i3 = 10;
2. short s3 = i3;

✓ Above code compiles successfully. If you are working with final variable and the value is within the range, then final variable is implicitly casted to target type, as in this case i3 is implicitly casted to short.

1. final int i4 = 40000;
2. short s3 = i4;

✗ It causes compilation failure as 40000 is out of range value.

1. final int i5 = 10;
2. short s5 = i5 + 100;

✓ Above code compiles successfully. If you are working with constant expression and the resultant value of the constant expression is within the range, then resultant value is implicitly casted. In this case, resultant value 110 is implicitly casted.

1. final int m = 25000;
2. final int n = 25000;
3. short s6 = m + n;

✗ m + n is a constant expression but resultant value 50000 is out of range for short type, hence it causes compilation failure.

1. int i7 = 10;
2. short s7 = i7;

✗ Compilation error as i7 is non-final variable and hence cannot be implicitly casted to short type.

Bottom of Form

Top of Form

Question 7: **Correct**

**Question ID: UKOCP38050**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. boolean b1 = 0;
6. boolean b2 = 1;
7. System.out.println(b1 + b2);
8. }
9. }

What is the result of compiling and executing Test class?

* 

**0**

* 

**1**

* 

**true**

* 

**false**

* 

**Compilation error**

**(Correct)**

**Explanation**

**UKOCP38050:**In Java language, boolean type can store only two values: true and false and these values are not compatible with int type. Also + operator is not defined for boolean types. Hence, all the 3 statements inside main method causes compilation error.

Bottom of Form

Top of Form

Question 8: **Correct**

**Question ID: UKOCP50172**

Given code:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. Boolean b1 = Boolean.valueOf("tRuE");
6. Boolean b2 = Boolean.valueOf("fAlSe");
7. Boolean b3 = Boolean.valueOf("abc");
8. Boolean b4 = null;
9. System.out.println(b1 + ":" + b2 + ":" + b3 + ":" + b4);
10. }
11. }

What is the result?

* 

**false:false:false:null**

* 

**true:false:false:null**

**(Correct)**

* 

**false:false:true:null**

* 

**Compilation error**

**Explanation**

**UKOCP50172:**

Boolean.valueOf(String) method uses equalsIgnoreCase method to validate the passed String, so if passed String is "true" ('t', 'r', 'u' and 'e' can be in any case), then boolean value stored in Boolean object is true otherwise false.

b1 stores true, b2 stores false, b3 stores false and as b4 is of reference type, hence it can store null as well.

Output is: true:false:false:null

Bottom of Form

Top of Form

Question 9: **Correct**

**Question ID: UKOCP80827**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
5. private static void add(double d1, double d2) {
6. System.out.println("double version: " + (d1 + d2));
7. }
9. private static void add(Double d1, Double d2) {
10. System.out.println("Double version: " + (d1 + d2));
11. }
13. public static void main(String[] args) {
14. add(10.0, null);
15. }
17. }

What is the result of compiling and executing Test class?

* 

**Compilation error**

* 

**double version: 10.0**

* 

**Double version: 10.0**

* 

**An exception is thrown at runtime**

**(Correct)**

**Explanation**

**UKOCP80827:**

add(10.0, null); => Compiler can't convert null to double primitive type, so 2nd argument is tagged to Double reference type. So to match the method call, 10.0 is converted to Double object by auto-boxing and add(10.0, null); is tagged to add(Double, Double); method.

But at the time of execution, d2 is null so System.out.println("Double version: " + (d1 + d2)); throws NullPointerException.

Bottom of Form

Top of Form

Question 10: **Correct**

**Question ID: UKOCP73291**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. m(1);
6. }
8. private static void m(Object obj) {
9. System.out.println("Object version");
10. }
12. private static void m(Number obj) {
13. System.out.println("Number version");
14. }
16. private static void m(Double obj) {
17. System.out.println("Double version");
18. }
19. }

What is the result of compiling and executing Test class?

* 

**Compilation error**

* 

**Object version**

* 

**Number version**

**(Correct)**

* 

**Double version**

**Explanation**

**UKOCP73291:**

There are 3 overloaded method m. Note all the numeric wrapper classes (Byte, Short, Integer, Long, Float and Double) extend from Number and Number extends from Object.

Compiler either does implicit casting or Wrapping but not both. 1 is int literal, Java compiler can't implicit cast it to double and then box it to Double rather it boxes i to Integer and as Number is the immediate super class of Integer so Number version refers to Integer object.

Number version is printed on to the console.

Bottom of Form

Top of Form

Question 11: **Correct**

**Question ID: UKOCP18448**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. extractInt(2.7);
6. extractInt(2);
7. }
9. private static void extractInt(Double obj) {
10. System.out.println(obj.intValue());
11. }
12. }

What is the result of compiling and executing Test class?

* 

**2**

**2**

* 

**3**

**2**

* 

**Compilation error in main method**

**(Correct)**

* 

**Compilation error in extractInt method**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP18448:**

extractInt method accepts argument of Double type.

extractInt(2.7); => 2.7 is double literal, so Java compiler would box it into Double type. At runtime obj.intValue() would print int portion of the Double data, which is 2.

extractInt(2); => Java compiler either does implicit casting or Wrapping but not both. 2 is int literal, Java compiler can't implicit cast it to double and then box it to Double. So this statement causes compilation failure.

Bottom of Form

Top of Form

Question 12: **Correct**

**Question ID: UKOCP40378**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String [] arr = {"abc", "TrUe", "false", null, "FALSE"};
6. for(String s : arr) {
7. System.out.print(Boolean.valueOf(s) ? "T" : "F");
8. }
9. }
10. }

What is the result of compiling and executing Test class?

* 

**FTFFF**

**(Correct)**

* 

**FFFFF**

* 

**TTFTT**

* 

**TTTFT**

* 

**NullPointerException is thrown at runtime**

**Explanation**

**UKOCP40378:**

Boolean.valueOf(String s) returns true if passed String argument is not null and is equal, ignoring case, to the String "true". In all other cases it returns false.

Boolean.valueOf("abc") => false. As "abc".equalsIgnoreCase("true") is false.

Boolean.valueOf("TrUe") => true. As "TrUe".equalsIgnoreCase("true") is true.

Boolean.valueOf("false") => false. As "false".equalsIgnoreCase("true") is false.

Boolean.valueOf(null) => false. As passed argument is null.

Boolean.valueOf("FALSE") => false. As "FALSE".equalsIgnoreCase("true") is false.

Bottom of Form

Top of Form

Question 13: **Correct**

**Question ID: UKOCP17116**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. System.out.println( 53 / 2.0 );
6. System.out.println( 53 % 2.0 );
7. }
8. }

What will be the result of compiling and executing Test class?

* 

**26**

**1**

* 

**26.5**

**1.0**

**(Correct)**

* 

**26.0**

**1.0**

* 

**26.5**

**0.0**

**Explanation**

**UKOCP17116:**

As floating point numbers are used in the expression, hence result should be in floating point number.

Correct result is:

53 / 2.0 = 26.5

53 % 2.0 = 1.0

Bottom of Form

Top of Form

Question 14: **Correct**

**Question ID: UKOCP46906**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. System.out.println("Password" + 1 + 2 + 3 + 4);
6. }
7. }

What will be the result of compiling and executing Test class?

* 

**Password10**

* 

**Password19**

* 

**Password1234**

**(Correct)**

* 

**Password 10**

**Explanation**

**UKOCP46906:**

As expression contains + operator only, which is left to right associative. Let us group the expression.

"Password" + 1 + 2 + 3 + 4

= ("Password" + 1) + 2 + 3 + 4

= (("Password" + 1) + 2) + 3 + 4

= ((("Password" + 1) + 2) + 3) + 4

[Let us solve it now, + operator with String behaves as concatenation operator.]

= (("Password1" + 2) + 3) + 4

= ("Password12" + 3) + 4

= "Password123" + 4

= "Password1234"

Bottom of Form

Top of Form

Question 15: **Correct**

**Question ID: UKOCP38055**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. System.out.println(1 + 2 + 3 + 4 + "Running");
6. }
7. }

What will be the result of compiling and executing Test class?

* 

**10Running**

**(Correct)**

* 

**1234Running**

* 

**64Running**

* 

**10 Running**

**Explanation**

**UKOCP38055:**

As expression contains + operator only, which is left to right associative. Let us group the expression.

1 + 2 + 3 + 4 + "Running"

= (1 + 2) + 3 + 4 + "Running"

= ((1 + 2) + 3) + 4 + "Running"

= (((1 + 2) + 3) + 4) + "Running"

Let us solve it now:

= ((3 + 3) + 4) + "Running"

= (6 + 4) + "Running"

= 10 + "Running"

[+ operator with String behaves as concatenation operator.]

= "10Running"

Bottom of Form

Top of Form

Question 16: **Correct**

**Question ID: UKOCP74139**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. System.out.println("1" + "2" + "3" == "1" + "2" + "3");
6. }
7. }

What will be the result of compiling and executing Test class?

* 

**Compilation error**

* 

**123**

* 

**true**

**(Correct)**

* 

**false**

**Explanation**

**UKOCP74139:**

Operator + has higher precedence over == and + operator is left to right associative, so let's group the given expression:

"1" + "2" + "3" == "1" + "2" + "3"

= ("1" + "2") + "3" == "1" + "2" + "3"

= ("1" + "2") + "3" == ("1" + "2") + "3"

= (("1" + "2") + "3") == (("1" + "2") + "3")

Let's solve it now:

= ("12" + "3") == (("1" + "2") + "3")

= "123" == (("1" + "2") + "3")

= "123" == ("12" + "3")

= "123" == "123"

= true

Please note that Strings computed by concatenation at compile time are referred by String Pool. Compile time String concatenation happens when both of the operands are compile time constants, such as literal, final variable etc. This means the result of constant expression is calculated at compile time and later referred by String Pool.

For the given question "123" is a String pool object and that is why true is returned by the given expression.

Bottom of Form

Top of Form

Question 17: **Incorrect**

**Question ID: UKOCP77648**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. System.out.println("Equals??? " + 10 != 5);
6. }
7. }

What will be the result of compiling and executing Test class?

* 

**Equals??? true**

* 

**Equals??? false**

**(Incorrect)**

* 

**Compilation error**

**(Correct)**

* 

**Equals??? 10 != 5**

**Explanation**

**UKOCP77648:**

Binary plus (+) has got higher precedence than != operator. Let us group the expression.

"Equals??? " + 10 != 5

= ("Equals??? " + 10) != 5

[!= is binary operator, so we have to evaluate the left side first. + operator behaves as concatenation operator.]

= "Equals??? 10" != 5

Left side of above expression is String, and right side is int. But String can't be compared to int hence compilation error.

Bottom of Form

Top of Form

Question 18: **Incorrect**

**Question ID: UKOCP56705**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. System.out.println("Result is: " + (10 != 5));
6. }
7. }

What will be the result of compiling and executing Test class?

* 

**Result is: true**

**(Correct)**

* 

**Result is: false**

**(Incorrect)**

* 

**Compilation error**

* 

**Result is: (10 != 5)**

**Explanation**

**UKOCP56705:**

"Result is: " + (10 != 5) [Nothing to evaluate at left side, so let's evaluate the right side of +, 10 != 5 is true.]

= "Result is: " + true [+ operator behaves as concatenation operator]

= "Result is: true"

Bottom of Form

Top of Form

Question 19: **Correct**

**Question ID: UKOCP51019**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String [] args) {
5. byte var = 127;
6. System.out.println(/\*INSERT\*/);
7. }
8. }

Range of byte data type is from -128 to 127.

Following options are available to replace /\*INSERT\*/:

1. var = var - 1

2. var = var + 1

3. ++var

4. --var

5. var \*= 2

6. var -= 10

7. var += 2

8. var

How many above options can be used to replace /\*INSERT\*/ (separately and not together) such that there is no compilation error?

* 

**One option only**

* 

**Two options only**

* 

**Three options only**

* 

**Four options only**

* 

**Five options only**

* 

**Six options only**

**(Correct)**

* 

**Seven options only**

* 

**All eight options**

**Explanation**

**UKOCP51019:**

var = var - 1: 'var - 1' results in int type and it cannot be assigned to byte type without explicit casting, hence it causes compilation error.

var = var + 1: 'var + 1' results in int type and it cannot be assigned to byte type without explicit casting, hence it causes compilation error.

Please note that implicit casting is added by the compiler for prefix, postfix and compound assignment operators.

++var: Compiler converts it to var = (byte) (var + 1) and hence it compiles successfully.

--var: Compiler converts it to var = (byte) (var - 1) and hence it compiles successfully.

var \*= 2: Compiler converts it to var = (byte) (var \* 2) and hence it compiles successfully.

var -= 10: Compiler converts it to var = (byte) (var - 10) and hence it compiles successfully.

var += 2: Compiler converts it to var = (byte) (var + 2) and hence it compiles successfully.

var: No issues at all, it also compiles successfully.

Hence, out of the given 8 options, 6 compiles successfully.

Bottom of Form

Top of Form

Question 20: **Correct**

**Question ID: UKOCP31889**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String [] args) {
5. String text = null;
6. /\*INSERT\*/
7. }
8. }

Following options are available to replace /\*INSERT\*/:

1. System.out.println(text.repeat(3));

2. System.out.println(null + null + null);

3. System.out.println(null + "null" + null);

4. System.out.println(text \*= 3);

5. System.out.println(text += "null".repeat(2));

6. System.out.println(text + text + text);

7. text += null; System.out.println((text.concat(null)));

How many of the above options can be used to replace /\*INSERT\*/ (separately and not together) such that nullnullnull is printed on to the console?

* 

**One option only**

* 

**Two options only**

* 

**Three options only**

**(Correct)**

* 

**Four options only**

* 

**Five options only**

* 

**Six options only**

* 

**All seven options**

**Explanation**

**UKOCP31889:**

You need to keep in mind an important point related to String Concatenation:

If only one operand expression is of type String, then string conversion is performed on the other operand to produce a string at run time.

If one of the operand is null, it is converted to the string "null".

If operand is not null, then the conversion is performed as if by an invocation of the toString method of the referenced object with no arguments; but if the result of invoking the toString method is null, then the string "null" is used instead.

Let's check all the options:

System.out.println(text.repeat(3));

✗ text refers to null, hence text.repeat(3) throws NullPointerException at runtime.

System.out.println(null + null + null);

✗ Compilation error as operator + is undefined for the arguments: null, null

System.out.println(null + "null" + null);

✓ If either one of the operand is of String type, then operator + acts as concatenation operator.

As multiple operators are used in above expression, so let's add parenthesis first.

= System.out.println((null + "null") + null); //Operator + is left to right associative

= System.out.println(("null" + "null") + null); //null is converted to "null" as it is used in String concatenation

= System.out.println("nullnull" + null);

= System.out.println("nullnull" + "null"); //null is converted to "null" as it is used in String concatenation

= System.out.println("nullnullnull");

It prints the expected output on to the console

System.out.println(text \*= 3);

✗ Compilation error as \*= operator is undefined for the arguments: String, int

System.out.println(text += "null".repeat(2));

✓ = System.out.println(text = text + "null".repeat(2));

= System.out.println(text = text + "nullnull");  //"null".repeat(2) returns "nullnull"

[Instance method 'repeat()' has been added to String class in Java 11 and it has the signature: `public String repeat(int count) {}`

It returns the new String object whose value is the concatenation of this String repeated 'count' times. For example,

"A".repeat(3); returns "AAA".]

= System.out.println(text = "null" + "nullnull"); //As text is null and used in String concatenation, so it is converted to "null"

= System.out.println(text = "nullnullnull");

"nullnullnull" is assigned to variable text and "nullnullnull" is printed on to the console as well.

System.out.println(text + text + text);

✓ Even though text refers to null but + operator is defined for the arguments: String, String. So, no compilation error.

Also as text is null and used in String concatenation, so it is converted to "null"

= System.out.println("null" + "null" + "null");

= System.out.println("nullnullnull");

It prints the expected output on to the console

text += null; System.out.println((text.concat(null)));

✗ First statement, `text += null;` is equals to `text = text + null`

As text is of String type, hence given operator behaves as String Concatenation operator and that is why null reference is converted to "null".

So, first statement translates to: `text = "null" + "null"` and that is why text refers to "nullnull".

Second statement, `System.out.println((text.concat(null)));`

= `System.out.println(("nullnull".concat(null)));`

If you see the source code of concat(String str) it uses str.length(); so invoking length() method on null reference causes NullPointerException.

Out of the 7 given options, only 3 will print expected output on to the console.

Bottom of Form

Top of Form

Question 21: **Correct**

**Question ID: UKOCP51498**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. class A {
4. public String toString() {
5. return null;
6. }
7. }
9. public class Test {
10. public static void main(String [] args) {
11. String text = null;
12. text = text + new A(); //Line n1
13. System.out.println(text.length()); //Line n2
14. }
15. }

What will be the result of compiling and executing Test class?

* 

**Line n1 causes compilation error**

* 

**Line n1 causes Runtime error**

* 

**Line n2 causes Runtime error**

* 

**0**

* 

**4**

* 

**8**

**(Correct)**

**Explanation**

**UKOCP51498:**

You need to keep in mind an important point related to String Concatenation:

If only one operand expression is of type String, then string conversion is performed on the other operand to produce a string at run time.

If one of the operand is null, it is converted to the string "null".

If operand is not null, then the conversion is performed as if by an invocation of the toString method of the referenced object with no arguments; but if the result of invoking the toString method is null, then the string "null" is used instead.

Let's check the expression of Line n1:

text = text + new A(); --> As text is of String type, hence + operator behaves as concatenation operator.

As text is null, so "null" is used in the Expression.

new A() represents the object of A class, so toString() method of A class is invoked, but as toString() method of A class returns null, hence "null" is used in the given expression.

So, given expression is written as:

text = "null" + "null";

text = "nullnull";

Hence, Line n2 prints 8 on to the console.

Bottom of Form

Top of Form

Question 22: **Correct**

**Question ID: UKOCP55855**

Consider below code fragment:

1. String text = "RISE ";
2. text = text + (text = "ABOVE ");
3. System.out.println(text);

What is the result?

* 

**RISE RISE ABOVE**

* 

**RISE ABOVE**

**(Correct)**

* 

**ABOVE ABOVE**

* 

**RISE ABOVE RISE**

**Explanation**

**UKOCP55855:**

Initially text refers to "RISE ".

Given statement:

text = text + (text = "ABOVE ");

text = "RISE " + (text = "ABOVE "); //Left operand of + operator is evaluated first, text --> "RISE "

text = "RISE " + "ABOVE "; //Right operand of + operator is evaluated next, text --> "ABOVE "

text = "RISE ABOVE "; //text --> "RISE ABOVE "

Hence `System.out.println(text);` print 'RISE ABOVE ' on to the console.

Bottom of Form

Top of Form

Question 23: **Correct**

**Question ID: UKOCP31888**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String [] args) {
5. StringBuilder sb = new StringBuilder("INHALE ");
6. String s = sb.toString() + (sb.append("EXHALE "));
7. System.out.println(s.strip().length());
8. }
9. }

What will be the result of compiling and executing Test class?

* 

**21**

* 

**20**

**(Correct)**

* 

**28**

* 

**27**

* 

**24**

* 

**23**

* 

**18**

* 

**17**

**Explanation**

**UKOCP31888:**

Initially sb refers to {"INHALE "}

Given Statement:

String s = sb.toString() + (sb.append("EXHALE "));

String s = "INHALE " + (sb.append("EXHALE ")); //Left operand of + operator is evaluated first, sb --> {"INHALE "}

String s = "INHALE " + {"INHALE EXHALE "}; //Right operand of + operator is evaluated next, sb --> {"INHALE EXHALE "}

String s = "INHALE " + "INHALE EXHALE "; //As left operand is of String type, so + operator behaves as concatenation operator and that is why toString() method on right operand (which is StringBuilder object referred by 'sb') is invoked.

String s = "INHALE INHALE EXHALE "; //String object referred by 's' has 21 characters.

strip() method of String class (available since Java 11) returns a string whose value is this string, with all leading and trailing white space removed.

To find out what is white space character in Java, check Character.isWhitespace​(int) method, you will find below:

A character is a Java whitespace character if and only if it satisfies one of the following criteria:

It is a Unicode space character (SPACE\_SEPARATOR, LINE\_SEPARATOR, or PARAGRAPH\_SEPARATOR) but is not also a non-breaking space ('\u00A0', '\u2007', '\u202F').

It is '\t', U+0009 HORIZONTAL TABULATION.

It is '\n', U+000A LINE FEED.

It is '\u000B', U+000B VERTICAL TABULATION.

It is '\f', U+000C FORM FEED.

It is '\r', U+000D CARRIAGE RETURN.

It is '\u001C', U+001C FILE SEPARATOR.

It is '\u001D', U+001D GROUP SEPARATOR.

It is '\u001E', U+001E RECORD SEPARATOR.

It is '\u001F', U+001F UNIT SEPARATOR.

White space character in java includes space character along with above characters.

As s refers to "INHALE INHALE EXHALE " [contains 21 characters], hence s.strip() creates a new String object by trimming trailing space character: "INHALE INHALE EXHALE" [contains 20 characters].

Hence, `System.out.println(s.strip().length());` prints 20 on to the console.

Bottom of Form

Top of Form

Question 24: **Correct**

**Question ID: UKOCP75007**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String [] args) {
5. boolean flag = false;
6. System.out.println((flag = true) | (flag = false) || (flag = true));
7. System.out.println(flag);
8. }
9. }

What will be the result of compiling and executing Test class?

* 

**true**

**false**

**(Correct)**

* 

**false**

**true**

* 

**true**

**true**

* 

**false**

**false**

* 

**Compilation error**

**Explanation**

**UKOCP75007:**

Given statement:

System.out.println((flag = true) | (flag = false) || (flag = true)); //flag = false

System.out.println(((flag = true) | (flag = false)) || (flag = true)); //bitwise inclusive OR | has higher precedence over logical OR ||. flag = false

|| has two operands, Left: ((flag = true) | (flag = false)) and Right: (flag = true). Left operand needs to be evaluated first.

System.out.println((true | (flag = false)) || (flag = true)); //flag = true

System.out.println((true | false) || (flag = true)); //flag = false

System.out.println(true || (flag = true)); //flag = false

|| is a short-circuit operator and as left operand evaluates to true, hence right operand is not evaluated.

Above statement prints true on to the console.

And

System.out.println(flag); prints false on to the console as flag variable is false.

Bottom of Form

Top of Form

Question 25: **Correct**

**Question ID: UKOCP55852**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String [] args) {
5. boolean status = true;
6. System.out.println(status = false || status = true | status = false);
7. System.out.println(status);
8. }
9. }

What will be the result of compiling and executing Test class?

* 

**true**

**false**

* 

**false**

**true**

* 

**true**

**true**

* 

**false**

**false**

* 

**Compilation error**

**(Correct)**

**Explanation**

**UKOCP55852:**

Given statement:

System.out.println(status = false || status = true | status = false);

As it contains multiple operators, hence let's group the operators first.

System.out.println(status = false || status = (true | status) = false); //Bitwise inclusive OR | has highest precedence over logical or || and assignment =

For assignment operator to work, left operand must be variable but in above case, `(true | status) = false` causes compilation failure as left operand (true | status) evaluates to a boolean value and not boolean variable.

Bottom of Form

Top of Form

Question 26: **Correct**

**Question ID: UKOCP51494**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String [] args) {
5. var a = 3; //Line n1
6. var b = 5; //Line n2
7. var c = 7; //Line n3
8. var d = 9; //Line n4
9. boolean res = --a + --b < 1 && c++ + d++ > 1;
10. System.out.printf("a = %d, b = %d, c = %d, d = %d, res = %b", a, b, c, d, res);
11. }
12. }

What will be the result of compiling and executing Test class?

* 

**a = 2, b = 4, c = 7, d = 9, res = false**

**(Correct)**

* 

**a = 2, b = 4, c = 8, d = 10, res = false**

* 

**a = 2, b = 4, c = 7, d = 9, res = true**

* 

**a = 2, b = 4, c = 8, d = 10, res = true**

* 

**a = 3, b = 5, c = 8, d = 10, res = false**

* 

**a = 3, b = 5, c = 8, d = 10, res = true**

**Explanation**

**UKOCP51494:**

Local variable Type inference was added in JDK 10.

Reserved type name var is allowed in JDK 10 onwards for local variable declarations with initializers, enhanced for-loop indexes, and index variables declared in traditional for loops. For example,

var x = "Java"; //x infers to String

var m = 10; //m infers to int

The identifier var is not a keyword, hence var can still be used as variable name, method name or package name but it cannot be used as a class or interface name.

At Line n1, a infers to int type.

At Line n2, b infers to int type.

At Line n3, c infers to int type.

At Line n4, d infers to int type.

Given expression:

--a + --b < 1 && c++ + d++ > 1;

--a + --b < 1 && (c++) + (d++) > 1; //postfix has got highest precedence

(--a) + (--b) < 1 && (c++) + (d++) > 1; //prefix comes after postfix

{(--a) + (--b)} < 1 && {(c++) + (d++)} > 1; //Then comes binary +. Though parentheses are used but I used curly brackets, just to explain.

[{(--a) + (--b)} < 1] && [{(c++) + (d++)} > 1]; //Then comes relational operator (<,>). I used square brackets instead of parentheses.

This expression is left with just one operator, && and this operator is a binary operator so works with 2 operands, left operand [{(--a) + (--b)} < 1] and right operand [{(c++) + (d++)} > 1]

Left operand of && must be evaluated first, which means [{(--a) + (--b)} < 1] must be evaluated first.

[{2 + (--b)} < 1] && [{(c++) + (d++)} > 1]; //a=2, b=5, c=7, d=9

[{2 + 4} < 1] && [{(c++) + (d++)} > 1]; //a=2, b=4, c=7, d=9

[6 < 1] && [{(c++) + (d++)} > 1];

false && [{(c++) + (d++)} > 1];

&& is short circuit operator, hence right operand is not evaluated and false is returned.

Output of the given program is: a = 2, b = 4, c = 7, d = 9, res = false

Bottom of Form

Top of Form

Question 27: **Correct**

**Question ID: UKOCP50163**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. int m = 20;
6. int var = --m \* m++ + m-- - --m;
7. System.out.println("m = " + m);
8. System.out.println("var = " + var);
9. }
10. }

What will be the result of compiling and executing Test class?

* 

**m = 25**

**var = 363**

* 

**m = 363**

**var = 363**

* 

**m = 18**

**var = 363**

**(Correct)**

* 

**Compilation error**

**Explanation**

**UKOCP50163:**

Given statement:

int var = --m \* m++ + m-- - --m;

int var = --m \* (m++) + (m--) - --m; // Postfix operator has higher precedence than other available operators

int var = (--m) \* (m++) + (m--) - (--m); //Then comes prefix operators

int var = ((--m) \* (m++)) + (m--) - (--m); //\* comes next

int var = (((--m) \* (m++)) + (m--)) - (--m); //+ is next

Right hand side is left with just one operator '-', it is a binary operator, hence let's solve the left hand side first.

int var = ((19 \* (m++)) + (m--)) - (--m); //m = 19

int var = ((19 \* 19) + (m--)) - (--m); //m = 20

int var = (361 + 20) - (--m); //m = 19

int var = 381 - (--m); //m = 19

int var = 381 - 18; //m = 18

int var = 363 // m = 18

So,

m = 18

var = 363

Bottom of Form

Top of Form

Question 28: **Correct**

**Question ID: UKOCP75006**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. int x = 7;
6. boolean res = x++ == 7 && ++x == 9 || x++ == 9;
7. System.out.println("x = " + x);
8. System.out.println("res = " + res);
9. }
10. }

What will be the result of compiling and executing Test class?

* 

**x = 10**

**res = true**

* 

**x = 9**

**res = true**

**(Correct)**

* 

**x = 10**

**res = false**

* 

**Compilation error**

**Explanation**

**UKOCP75006:**

Given statement:

boolean res = x++ == 7 && ++x == 9 || x++ == 9;

boolean res = (x++) == 7 && ++x == 9 || (x++) == 9; // Postfix operator has higher precedence than other available operators

boolean res = (x++) == 7 && (++x) == 9 || (x++) == 9; //Then comes prefix operators

boolean res = ((x++) == 7) && ((++x) == 9) || ((x++) == 9); //== operator comes next

boolean res = (((x++) == 7) && ((++x) == 9)) || ((x++) == 9); //&& has higher precedence over ||

Right hand side is left with just one operator '||', it is a binary operator, hence let's solve the left hand side first.

boolean res = ((7 == 7) && ((++x) == 9)) || ((x++) == 9); //x = 8

boolean res = (true && ((++x) == 9)) || ((x++) == 9); //x = 8

boolean res = (true && (9 == 9)) || ((x++) == 9); //x = 9

boolean res = (true && true) || ((x++) == 9); //x = 9

boolean res = true || ((x++) == 9); //x = 9, || is a short-circuit operator, given expression evaluates to true without evaluating `((x++) == 9)`

boolean res = true; //x = 9

So,

x = 9

res = true

Bottom of Form

Top of Form

Question 29: **Correct**

**Question ID: UKOCP79843**

Given code:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String [] args) {
5. int a = 1000;
6. System.out.println(-a++);
7. }
8. }

What is the result?

* 

**Compilation error**

* 

**-1000**

**(Correct)**

* 

**-1001**

* 

**999**

* 

**-999**

**Explanation**

**UKOCP79843:**

First add parenthesis (round brackets) to the given expression: -a++.

There are 2 operators involved. unary minus and Postfix operator. Let's start with expression and value of a.

-a++; [a = 1000].

-(a++); [a = 1000] Postfix operator has got higher precedence than unary operator.

-(1000); [a = 1001] Use the value of a (1000) in the expression and after that increase the value of a to 1001.

-1000; [a = 1001] -1000 is printed on to the console.

Bottom of Form

Top of Form

Question 30: **Incorrect**

**Question ID: UKOCP10968**

What will be the result of compiling and executing Test class?

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String [] args) {
5. int i = 2;
6. boolean res = false;
7. res = i++ == 2 || --i == 2 && --i == 2;
8. System.out.println(i);
9. }
10. }

* 

**2**

**(Incorrect)**

* 

**3**

**(Correct)**

* 

**1**

* 

**Compilation error**

**Explanation**

**UKOCP10968:**

i++ == 2 || --i == 2 && --i == 2; [Given expression].

(i++) == 2 || --i == 2 && --i == 2; [Postfix has got higher precedence than other operators].

(i++) == 2 || (--i) == 2 && (--i) == 2; [After postfix, precedence is given to prefix].

((i++) == 2) || ((--i) == 2) && ((--i) == 2); [== has higher precedence over && and ||].

((i++) == 2) || (((--i) == 2) && ((--i) == 2)); [&& has higher precedence over ||].

Let's start solving it:

((i++) == 2) || (((--i) == 2) && ((--i) == 2)); [i=2, res=false].

(2 == 2) || (((--i) == 2) && ((--i) == 2)); [i=3, res=false].

true || (((--i) == 2) && ((--i) == 2)); [i=3, res=false].  || is a short-circuit operator, hence no need to evaluate expression on the right.

res is true and i is 3.

Bottom of Form

Top of Form

Question 31: **Incorrect**

**Question ID: UKOCP51495**

Consider below code snippet:

1. int i = 10;
2. System.out.println(i > 3 != false);

What is the result?

* 

**Compilation error**

* 

**true**

**(Correct)**

* 

**false**

**(Incorrect)**

* 

**null**

**Explanation**

**UKOCP51495:**

Given Expression:

i > 3 != false

It has 2 operators > and !=. > has higher precedence over !=, hence given expression can be written as:

(i > 3) != false

Let's solve above expression:

true != false

true

Hence true is printed on to the console.

Bottom of Form

Top of Form

Question 32: **Correct**

**Question ID: UKOCP66527**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String [] args) {
5. boolean flag1 = true;
6. boolean flag2 = false;
7. boolean flag3 = true;
8. boolean flag4 = false;
10. System.out.println(!flag1 == flag2 != flag3 == !flag4); //Line n1
11. System.out.println(flag1 = flag2 != flag3 == !flag4); //Line n2
12. }
13. }

What will be the result of compiling and executing Test class?

* 

**Line n1 causes compilation error**

* 

**Line n2 causes compilation error**

* 

**true**

**true**

* 

**true**

**false**

* 

**false**

**true**

**(Correct)**

* 

**false**

**false**

**Explanation**

**UKOCP66527:**

Let's solve the expression at Line n1:

!flag1 == flag2 != flag3 == !flag4

(!flag1) == flag2 != flag3 == (!flag4) //Logical NOT has got highest precedence among given operators

((!flag1) == flag2) != flag3 == (!flag4) //== and != have same precedence and left to right associative, grouping == first

(((!flag1) == flag2) != flag3) == (!flag4) //grouping != next

Above expression is left with single operator ==, whose left side is: (((!flag1) == flag2) != flag3) and right side is: (!flag4). As == is a binary operator, so left side is evaluated first.

((false == flag2) != flag3) == (!flag4) //!flag1 is false

((false == false) != flag3) == (!flag4) //flag2 is false

(true != flag3) == (!flag4) //(false == false) evaluates to true

(true != true) == (!flag4) //flag3 is true

false == (!flag4) //(true != true) evaluates to false

false == true //!flag4 is true

false //(false == true) evaluates to false

Hence, false is printed on to the console.

Let's solve the expression at Line n2:

flag1 = flag2 != flag3 == !flag4

flag1 = flag2 != flag3 == (!flag4) //Logical NOT has got highest precedence among given operators

flag1 = (flag2 != flag3) == (!flag4) //== and != have same precedence and left to right associative, grouping == first

flag1 = ((flag2 != flag3) == (!flag4)) //grouping == next

Above expression is left with single assignment operator =, whose right side needs to be evaluated first

flag1 = ((false != flag3) == (!flag4)) //flag2 is false

flag1 = ((false != true) == (!flag4)) //flag3 is true

flag1 = (true == (!flag4)) //(false != true) evaluates to true

flag1 = (true == true) //!flag4 is true

flag1 = true //(true == true) evaluates to true

true is assigned to flag1 and true is also printed on to the console

One suggestion: In the real exam, if you find a question containing multiple expressions, then first check if there is any compilation error or not. If there is no compilation error in all the expressions, then only solve the expressions.

Bottom of Form

Top of Form

Question 33: **Correct**

**Question ID: UKOCP74152**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String [] args) {
5. var val = 9;
6. System.out.println(val += 10 - -val-- - --val);
7. }
8. }

What will be the result of compiling and executing Test class?

* 

**21**

**(Correct)**

* 

**22**

* 

**23**

* 

**24**

* 

**25**

* 

**26**

* 

**Compilation error**

**Explanation**

**UKOCP74152:**

Given expression:

val += 10 - -val-- - --val

val = val + 10 - -val-- - --val

val = val + 10 - -(val--) - --val //Postfix operator has higher precedence than other available operators

val = val + 10 - (-(val--)) - (--val) //Unary minus and prefix operator has same preference

val = (val + 10) - (-(val--)) - (--val) // + and - have same preference and both are left associative, hence grouping + first.

val = ((val + 10) - (-(val--))) - (--val) //Grouping - next

Expression on right side of assignment operator has 2 parts: Left: ((val + 10) - (-(val--))) and Right: (--val). Expression on Left side needs to be evaluated first.

val = ((9 + 10) - (-(val--))) - (--val) //val=9

val = (19 - (-(val--))) - (--val) //val=9

val = (19 - (-9)) - (--val) //val=8

val = 28 - (--val) //val=8

val = 28 - 7 //val=7

val = 21

21 is assigned to val and 21 is printed on to the console as well.

Bottom of Form

Top of Form

Question 34: **Correct**

**Question ID: UKOCP22952**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. var str = "Have Faith!"; //Line n1
6. System.out.println(str.length() + " : " + str.charAt(10)); //Line n2
7. }
8. }

What will be the result of compiling and executing Test class?

* 

**11 : !**

**(Correct)**

* 

**10 : !**

* 

**An exception is thrown at runtime**

* 

**11 : h**

* 

**10 : h**

* 

**Compilation error**

**Explanation**

**UKOCP22952:**

Local variable Type inference was added in JDK 10.

Reserved type name var is allowed in JDK 10 onwards for local variable declarations with initializers, enhanced for-loop indexes, and index variables declared in traditional for loops. For example,

var x = "Java"; //x infers to String

var m = 10; //m infers to int

In Line n1, str infers to String type.

String class has length() method, which returns number of characters in the String object. So length() method returns 11.

String class has charAt(int index) method, which returns character at passed index. str.charAt(10) looks for character at index 10. index starts with 0. ! sign is at index 10.

Hence LIne n2 prints 11 : ! on to the console.

For more information on local variable type inference, please check the URL: http://openjdk.java.net/jeps/286

Bottom of Form

Top of Form

Question 35: **Correct**

**Question ID: UKOCP42545**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String s1 = "OCP";
6. String s2 = "ocp";
7. System.out.println(/\*INSERT\*/);
8. }
9. }

Which of the following options, if used to replace /\*INSERT\*/, will compile successfully and on execution will print true on to the console?

Select 2 options.

* 

**s1.equals(s2)**

* 

**s1.equals(s2.toUpper())**

* 

**s2.equals(s1.toLower())**

* 

**s1.length() == s2.length()**

**(Correct)**

* 

**s1.equalsIgnoreCase(s2)**

**(Correct)**

* 

**s1.contentEquals(s2)**

**Explanation**

**UKOCP42545:**

Let's check all the statements one by one:

s1.equals(s2): equals(String) method of String class matches two String objects and it takes character's case into account while matching. Alphabet A in upper case and alphabet a in lower case are not equal according to this method. As String objects referred by s1 and s2 have different cases, hence output is false.

s1.equals(s2.toUpper()): Compilation error as there is no toUpper() method available in String class. Correct method name is: toUpperCase().

s2.equals(s1.toLower()): Compilation error as there is no toLower() method available in String class. Correct method name is: toLowerCase().

s1.length() == s2.length(): length() method returns the number of characters in the String object. s1.length() returns 3 and s2.length() also returns 3, hence output is true.

s1.equalsIgnoreCase(s2): Compares s1 and s2, ignoring case consideration and hence returns true.

s1.contentEquals(s2): String class contains two methods: contentEquals(StringBuffer) and contentEquals(CharSequence). Please note that String, StringBuilder and StringBuffer classes implement CharSequence interface, hence contentEquals(CharSequence) method defined in String class cab be invoked with the argument of either String or StringBuilder or StringBuffer. In this case, it is invoked with String argument and hence it is comparing the contents of two String objects. This method also takes character's case into account while matching. As String objects referred by s1 and s2 have different cases, hence output is false.

Bottom of Form

Top of Form

Question 36: **Correct**

**Question ID: UKOCP42543**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String fName = "Joshua";
6. String lName = "Bloch";
7. System.out.println(fName = lName);
8. }
9. }

What will be the result of compiling and executing Test class?

* 

**Compilation error**

* 

**false**

* 

**true**

* 

**None of the other options**

**(Correct)**

**Explanation**

**UKOCP42543:**

Both fName and lName are of String reference type. fName refers to "Joshua" and lName refers to "Bloch".

In System.out.println() statement, we have used assignment operator (=) and not equality operator (==). So result is never boolean.

fName = lName means copy the contents of lName to fName.

As lName is referring to "Bloch" and so after the assignment, fName starts referring to "Bloch" as well.

System.out.println() finally prints the String referred by fName, which is "Bloch".

As this option is is not available, hence correct answer is "None of the other options"

Bottom of Form

Top of Form

Question 37: **Correct**

**Question ID: UKOCP89634**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String str = "Think"; //Line n3
6. change(str); //Line n4
7. System.out.println(str); //Line n5
8. }
10. private static void change(String s) {
11. s.concat("\_Twice"); //Line n9
12. }
13. }

What will be the result of compiling and executing Test class?

* 

**Think**

**(Correct)**

* 

**\_Twice**

* 

**Think\_Twice**

* 

**None of the other options**

**Explanation**

**UKOCP89634:**

At Line n3, String reference variable 'str' refers to String object "Think".

When change(String) method is invoked by Line n4, both variables 's' and 'str' refer to the same String object "Think".

As String is immutable, so Line 9 doesn't modify the passed object, instead it creates a new String object "Think\_Twice".

But this newly created object is not referred and hence is a candidate for GC.

When control goes back to calling method main(String[]), 'str' still refers to "Think".

Line n5 prints "Think" on to the console.

Bottom of Form

Top of Form

Question 38: **Correct**

**Question ID: UKOCP17121**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String [] args) {
5. String text = "ONE ";
6. System.out.println(text.concat(text.concat("ELEVEN ")).trim());
7. }
8. }

What will be the result of compiling and executing Test class?

* 

**ONE ELEVEN**

* 

**ONE ONE ELEVEN**

**(Correct)**

* 

**ONE ELEVEN ONE ELEVEN**

* 

**ONE ELEVEN ONE**

**Explanation**

**UKOCP17121:**

Given statement:

System.out.println(text.concat(text.concat("ELEVEN ")).trim()); //'text' refers to "ONE "

System.out.println(text.concat("ONE ELEVEN ").trim()); //As String is immutable, hence there is no change in the String object referred by 'text', 'text' still refers to "ONE "

System.out.println(("ONE ONE ELEVEN ").trim()); //'text' still refers to "ONE "

System.out.println("ONE ONE ELEVEN"); //trim() method removes the trailing space in this case

ONE ONE ELEVEN is printed on to the console.

Bottom of Form

Top of Form

Question 39: **Correct**

**Question ID: UKOCP88788**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String text = " BE YOURSELF! "; //Contains multiple space characters
6. System.out.println(text./\*INSERT\*/);
7. }
8. }

Which of the following options, if used to replace /\*INSERT\*/, will compile successfully and on execution will print "BE YOURSELF!" on to the console?

Select 3 options.

* 

**trim()**

**(Correct)**

* 

**strip()**

**(Correct)**

* 

**stripLeading().stripTrailing()**

**(Correct)**

* 

**ltrim().rtrim()**

* 

**trimLeading().trimTrailing()**

* 

**trimBoth()**

**Explanation**

**UKOCP88788:**

Javadoc of trim() method states that it returns a string whose value is this string, with all leading and trailing space removed, where space is defined as any character whose codepoint is less than or equal to 'U+0020' or '\u0020' (the space character).

As the developer comment in above code states that 'text' refers to String object containing multiple space characters, hence text.trim() returns "BE YOURSELF!" after removing all leading and trailing space characters. Hence trim() is correct option.

stripLeading(), stripTrailing() and strip() were added in Java 11.

stripLeading() returns a string whose value is this string, with all leading white space removed.

stripTrailing() returns a string whose value is this string, with all trailing white space removed.

strip() returns a string whose value is this string, with all leading and trailing white space removed.

Point to note here is that trim() method works with space, where space is defined as any character whose codepoint is less than or equal to 'U+0020' or '\u0020' (the space character) but strip(), stripLeading() and stripTrailing() works with white space.

To find out what is white space character in Java, check Character.isWhitespace​(int) method, you will find below:

A character is a Java whitespace character if and only if it satisfies one of the following criteria:

It is a Unicode space character (SPACE\_SEPARATOR, LINE\_SEPARATOR, or PARAGRAPH\_SEPARATOR) but is not also a non-breaking space ('\u00A0', '\u2007', '\u202F').

It is '\t', U+0009 HORIZONTAL TABULATION.

It is '\n', U+000A LINE FEED.

It is '\u000B', U+000B VERTICAL TABULATION.

It is '\f', U+000C FORM FEED.

It is '\r', U+000D CARRIAGE RETURN.

It is '\u001C', U+001C FILE SEPARATOR.

It is '\u001D', U+001D GROUP SEPARATOR.

It is '\u001E', U+001E RECORD SEPARATOR.

It is '\u001F', U+001F UNIT SEPARATOR.

White space character in java includes space character along with above characters.

Hence, text.strip() will successfully remove all the leading and trailing white spaces and will return "BE YOURSELF!".

text.stripLeading().stripTrailing() will first remove leading white space and then trailing white space and will return "BE YOURSELF!".

String class doesn't have methods with the names: ltrim(), rtrim(), trimLeading(), trimTrailing() and trimBoth(), hence these will cause compilation error.

Bottom of Form

Top of Form

Question 40: **Correct**

**Question ID: UKOCP88304**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test extends String {
4. @Override
5. public String toString() {
6. return "TEST";
7. }
9. public static void main(String[] args) {
10. Test obj = new Test();
11. System.out.println(obj);
12. }
13. }

What will be the result of compiling and executing Test class?

* 

**TEST**

* 

**Output string contains @ symbol**

* 

**Exception is thrown at runtime**

* 

**Compilation error**

**(Correct)**

**Explanation**

**UKOCP88304:**

String is a final class so it cannot be extended.

Bottom of Form

Top of Form

Question 41: **Correct**

**Question ID: UKOCP30560**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String s1 = new String("Java"); //Line n3
6. String s2 = "JaVa"; //Line n4
7. String s3 = "JaVa"; //Line n5
8. String s4 = "Java"; //Line n6
9. String s5 = "Java"; //Line n7
11. int i = 1; //Line n9
13. }
14. }

How many String objects are there in the HEAP memory, when control is at Line n9?

* 

**2**

* 

**3**

**(Correct)**

* 

**4**

* 

**5**

**Explanation**

**UKOCP30560:**

String s1 = new String("Java"); --> Creates 2 objects: 1 String Pool and 1 non-pool. 's1' refers to non-pool object.

String s2 = "JaVa"; --> Creates 1 String pool object and 's2' refers to it.

String s3 = "JaVa"; --> Doesn't create a new object, 's3' refers to same String pool object referred by 's2'.

String s4 = "Java"; --> Doesn't create a new object, s4 refers to String Pool object created at Line n3.

String s5 = "Java"; --> Doesn't create a new object, s5 also refers to String Pool object created at Line n3.

So, at Line n9, 3 String objects are available in the HEAP memory: 2 String pool and 1 non-pool.

Bottom of Form

Top of Form

Question 42: **Correct**

**Question ID: UKOCP58910**

A bank's swift code is generally of 11 characters and used in international money transfers.

An example of swift code: ICICINBBRT4

ICIC: First 4 letters for bank code

IN: Next 2 letters for Country code

BB: Next 2 letters for Location code

RT4: Next 3 letters for Branch code

Which of the following code correctly extracts country code from the swift code referred by String reference variable swiftCode?

* 

**swiftCode.substring(4, 6);**

**(Correct)**

* 

**swiftCode.substring(5, 6);**

* 

**swiftCode.substring(5, 7);**

* 

**swiftCode.substring(4, 5);**

**Explanation**

**UKOCP58910:**

substring(int beginIndex, int endIndex) is used to extract the substring. The substring begins at "beginIndex" and extends till "endIndex - 1".

Country code information is stored at index 4 and 5, so the correct substring method to extract country code is: swiftCode.substring(4, 6);

Bottom of Form

Top of Form

Question 43: **Incorrect**

**Question ID: UKOCP52338**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String str = " "; //single space
6. boolean b1 = str.isEmpty();
7. boolean b2 = str.isBlank();
8. System.out.println(b1 + " : " + b2); //Line n1
10. str.strip(); //Line n2
11. b1 = str.isEmpty();
12. b2 = str.isBlank();
13. System.out.println(b1 + " : " + b2); //Line n3
14. }
15. }

What will be the result of compiling and executing Test class?

* 

**false : true true : true**

* 

**false : false**

**true : true**

* 

**true : false**

**true : false**

**(Incorrect)**

* 

**false : true**

**false : true**

**(Correct)**

* 

**false : true**

**true : false**

**Explanation**

**UKOCP52338:**

isEmpty() method of String class returns true if and only if length() is 0.

isBlank() method of String class (available since Java 11) returns true if the string is empty or contains only white space codepoints, otherwise false.

strip() method of String class (available since Java 11) returns a string whose value is this string, with all leading and trailing white space removed.

To find out what is white space character in Java, check Character.isWhitespace​(int) method, you will find below:

A character is a Java whitespace character if and only if it satisfies one of the following criteria:

It is a Unicode space character (SPACE\_SEPARATOR, LINE\_SEPARATOR, or PARAGRAPH\_SEPARATOR) but is not also a non-breaking space ('\u00A0', '\u2007', '\u202F').

It is '\t', U+0009 HORIZONTAL TABULATION.

It is '\n', U+000A LINE FEED.

It is '\u000B', U+000B VERTICAL TABULATION.

It is '\f', U+000C FORM FEED.

It is '\r', U+000D CARRIAGE RETURN.

It is '\u001C', U+001C FILE SEPARATOR.

It is '\u001D', U+001D GROUP SEPARATOR.

It is '\u001E', U+001E RECORD SEPARATOR.

It is '\u001F', U+001F UNIT SEPARATOR.

White space character in java includes space character along with above characters.

Let's check the code:

'str' refers to " " and its length is 1.

`str.isEmpty()` returns false, as length() is not 0.

`str.isBlank()` returns true, as it contains only white space character.

So, Line n1 prints false : true on to the console.

At Line n2, `str.strip();` returns an empty string "". As String is immutable, hence a new String object is created and 'str' still refers to " ".

As, there is no change in 'str', Line n3 also prints false : true on to the console.

Bottom of Form

Top of Form

Question 44: **Correct**

**Question ID: UKOCP62170**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String s1 = "1Z0-819";
6. String s2 = "1Z0-819" + "";
7. System.out.println(s1 == s2);
8. }
9. }

What will be the result of compiling and executing Test class?

* 

**1Z0-819**

* 

**true**

**(Correct)**

* 

**false**

* 

**Compilation error**

**Explanation**

**UKOCP62170:**

Please note that Strings computed by concatenation at compile time, will be referred by String Pool during execution. Compile time String concatenation happens when both of the operands are compile time constants, such as literal, final variable etc.

For the statement, String s2 = "1Z0-819" + "";, `"1Z0-819" + ""` is a constant expression as both the operands "1Z0-819" and "" are String literals, which means the expression `"1Z0-819" + ""` is computed at compile-time and results in String literal "1Z0-819".

So, during compilation, Java compiler translates the statement

String s2 = "1Z0-819" + "";

to

String s2 = "1Z0-819";

As "1Z0-819" is a String literal, hence at runtime it will be referred by String Pool.

When Test class is executed,

s1 refers to "1Z0-819" (String Pool object).

s2 also refers to same String pool object "1Z0-819".

s1 and s2 both refer to the same String object and that is why s1 == s2 returns true.

Please note that Strings computed by concatenation at run time (if the resultant expression is not constant expression) are newly created and therefore distinct.

For below code snippet:

String str1 = "1Z0-819";

String str2 = str1 + "";

System.out.println(str1 == str2);

Output is false, as str1 is a variable and `str1 + ""` is not a constant expression, therefore this expression is computed only at runtime and a new non-pool String object is created.

Bottom of Form

Top of Form

Question 45: **Correct**

**Question ID: UKOCP55846**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. final String fName = "James";
6. String lName = "Gosling";
7. String name1 = fName + lName;
8. String name2 = fName + "Gosling";
9. String name3 = "James" + "Gosling";
10. System.out.println(name1 == name2);
11. System.out.println(name2 == name3);
12. }
13. }

What will be the result of compiling and executing Test class?

* 

**true**

**true**

* 

**true**

**false**

* 

**false**

**false**

* 

**false**

**true**

**(Correct)**

**Explanation**

**UKOCP55846:**

Please note that Strings computed by concatenation at compile time, will be referred by String Pool during execution. Compile time String concatenation happens when both of the operands are compile time constants, such as literal, final variable etc.

Whereas, Strings computed by concatenation at run time (if the resultant expression is not constant expression) are newly created and therefore distinct.

fName is a constant variable and lName is a non-constant variable.

`fName + lName` is not a constant expression and hence the expression will be computed at run-time and the resultant String object "JamesGosling" will not be referred by String Pool.

As fName is constant variable and "Gosling" is String literal, hence the expression `fName + "Gosling"` is a constant expression, therefore expression is computed at compile-time and results in String literal "JamesGosling".

So, during compilation, Java compiler translates the statement

String name2 = fName + "Gosling";

to

String name2 = "JamesGosling";

As "JamesGosling" is a String literal, hence at runtime it will be referred by String Pool.

So, at runtime name1 and name2 refer to different String object and that is why name1 == name2 returns false.

`"James" + "Gosling"` is also a constant expression and hence Java compiler translates the statement

String name3 = "James" + "Gosling";

to

String name3 = "JamesGosling";

This means at runtime, variable 'name3' will refer to the same String pool object "JamesGosling", which is referred by variable 'name3'.

So, name2 and name3 refer to same String object and that is why name2 == name3 returns true.

Bottom of Form

Top of Form

Question 46: **Correct**

**Question ID: UKOCP33616**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. final int i1 = 1;
6. final Integer i2 = 1;
7. final String s1 = ":ONE";
9. String str1 = i1 + s1;
10. String str2 = i2 + s1;
12. System.out.println(str1 == "1:ONE");
13. System.out.println(str2 == "1:ONE");
14. }
15. }

What will be the result of compiling and executing Test class?

* 

**true**

**true**

* 

**true**

**false**

**(Correct)**

* 

**false**

**false**

* 

**false**

**true**

**Explanation**

**UKOCP33616:**

Please note that Strings computed by concatenation at compile time, will be referred by String Pool during execution. Compile time String concatenation happens when both of the operands are compile time constants, such as literal, final variable etc.

Whereas, Strings computed by concatenation at run time (if the resultant expression is not constant expression) are newly created and therefore distinct.

For the statement, String str1 = i1 + s1;, i1 is a final variable of int type and s1 is a final variable of String type. Hence, `i1 + s1` is a constant expression which is computed at compile-time and results in String literal "1:ONE".

This means during compilation, Java compiler translates the statement

String str1 = i1 + s1;

to

String str1 = "1:ONE";

As "1:ONE" is a String literal, hence at runtime it will be referred by String Pool.

On the other hand, for the statement, String str2 = i2 + s1;, `i2 + s1` is not a constant expression because i2 is neither of primitive type nor of String type, hence it is computed at run-time and returns a non-pool String object "1:ONE".

As, str1 refers to String Pool object "1:ONE", hence `str1 == "1:ONE"` returns true, whereas str2 refers to non-Pool String object "1:ONE" and hence `str2 == "1:ONE"` returns false.

Bottom of Form

Top of Form

Question 47: **Correct**

**Question ID: UKOCP52377**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String javaworld = "JavaWorld";
6. String java = "Java";
7. String world = "World";
8. java += world;
9. System.out.println(java == javaworld);
10. }
11. }

What will be the result of compiling and executing Test class?

* 

**JavaWorld**

* 

**Java**

* 

**World**

* 

**true**

* 

**false**

**(Correct)**

**Explanation**

**UKOCP52377:**

Please note that Strings computed by concatenation at compile time, will be referred by String Pool during execution. Compile time String concatenation happens when both of the operands are compile time constants, such as literal, final variable etc.

Whereas, Strings computed by concatenation at run time (if the resultant expression is not constant expression) are newly created and therefore distinct.

`java += world;` is same as `java = java + world;` and `java + world` is not a constant expression and hence is calculated at runtime and returns a non pool String object "JavaWorld", which is referred by variable 'java'.

On the other hand, variable 'javaworld' refers to String Pool object "JavaWorld".

As both the variables 'java' and 'javaworld' refer to different String objects, hence `java == javaworld` returns false.

Bottom of Form

Top of Form

Question 48: **Correct**

**Question ID: UKOCP78966**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. m1(null);
6. }
8. static void m1(CharSequence s) {
9. System.out.println("CharSequence");
10. }
12. static void m1(String s) {
13. System.out.println("String");
14. }
16. static void m1(Object s) {
17. System.out.println("Object");
18. }
19. }

What will be the result of compiling and executing Test class?

* 

**Compilation Error**

* 

**CharSequence**

* 

**String**

**(Correct)**

* 

**Object**

**Explanation**

**UKOCP78966:**

Method m1 is overloaded to accept 3 different parameters: String, CharSequence and Object.

String implements CharSequence and Object is the super Parent class in Java. There is no conflict among the overloaded methods for the call m1(null) as it is mapped to the class lowest in hierarchy, which is String class. Hence, output will be "String".

Now if you add one more overloaded method, `static void m1(StringBuilder s) {...}` in the Test class, then `m1(null);` would cause compilation error as it would match to both m1(StringBuilder) and m1(String) methods. So m1(null) in that case would be ambiguous call and would cause compilation error.

For the same reason, System.out.println(null); causes compilation error as println method is overloaded to accept 3 reference types Object, String and char [] along with primitive types.

System.out.println(null); matches to both println(char[]) and println(String), so it is an ambiguous call and hence the compilation error.

Bottom of Form

Top of Form

Question 49: **Correct**

**Question ID: UKOCP76305**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. /\*INSERT\*/ x = 7, y = 200;
6. System.out.println(String.valueOf(x + y).length());
7. }
8. }

Which of the following options, if used to replace /\*INSERT\*/, will compile successfully and on execution will print 3 on to the console?

Choose 3 options.

* 

**byte**

* 

**short**

**(Correct)**

* 

**int**

**(Correct)**

* 

**long**

**(Correct)**

* 

**float**

* 

**double**

* 

**var**

**Explanation**

**UKOCP76305:**

Local variable Type inference was added in JDK 10.

Reserved type name var is allowed in JDK 10 onwards for local variable declarations with initializers, enhanced for-loop indexes, and index variables declared in traditional for loops. For example,

var x = "Java"; //x infers to String

var m = 10; //m infers to int

The identifier var is not a keyword, hence var can still be used as variable name, method name or package name but it cannot be used as a class or interface name.

Compound declarations are allowed in Java for primitive type and reference type but not for var type. Hence, var cannot be used to replace /\*INSERT\*/.

Range of byte data type is from -128 to 127, hence if byte is used to replace /\*INSERT\*/, then y = 200 would cause compilation error as 200 is out of range value for byte type. Hence, byte cannot be used to replace /\*INSERT\*/.

short, int, long, float & double can replace /\*INSERT\*/ without causing any error. x + y will evaluate to 207 for short, int and long types whereas, x  + y will evaluate to 207.0 for float and double types.

String class has overloaded valueOf methods for int, char, long, float, double, boolean, char[] and Object types. valueOf method returns the corresponding String object and length() method returns number of characters in the String object.

So, `String.valueOf(x + y).length()` in case of short, int and long returns 3, on the other hand, in case of float and double it would return 5.

Hence, only 3 options (short, int and long) print expected output on to the console.

For more information on local variable type inference, please check the URL: http://openjdk.java.net/jeps/286

Bottom of Form

Top of Form

Question 50: **Incorrect**

**Question ID: UKOCP11824**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String str = "\*";
6. /\*INSERT\*/
7. System.out.println(str);
8. }
9. }

Which of the following options, if used to replace /\*INSERT\*/, will compile successfully and on execution will print \*\*\*\*\* on to the console?

Select 2 options.

* 

**str.repeat(5);**

**(Incorrect)**

* 

**str = str.repeat(5);**

**(Correct)**

* 

**String.repeat(str, 5);**

* 

**str = String.repeat(str, 5);**

* 

**str \*= 5;**

* 

**str = str \* 5;**

* 
  1. for(var var = 0; var < 4; var++) {
  2. str += "\*";
  3. }

**(Correct)**

* 
  1. for(int i = 0; i < 5; i++) {
  2. str += "\*";
  3. }

**(Incorrect)**

**Explanation**

**UKOCP11824:**

Instance method 'repeat()' has been added to String class in Java 11 and it has the signature: `public String repeat(int count) {}`

It returns the new String object whose value is the concatenation of this String repeated 'count' times. For example,

"A".repeat(3); returns "AAA".

In this question, 'str' refers to "\*". Let's check all the options:

`str.repeat(5);`: returns a new String Object "\*\*\*\*\*" but as String is immutable, so there is no change is the String object referred by 'str'. It will not give expected output.

`str = str.repeat(5);`: returns a new String Object "\*\*\*\*\*" and 'str' refers to this newly created object. It will print the expected output.

`String.repeat(str, 5);`: repeat is an instance method and not static. It causes compilation error.

`str = String.repeat(str, 5);`: repeat is an instance method and not static. It causes compilation error.

`str \*= 5;`: Operator '\*=' is not defined for String type and hence it causes compilation error.

`str = str \* 5;`: Operator '\*' is not defined for String type and hence it causes compilation error.

`for(var var = 0; var < 4; var++) { str += "\*"; }`:

Local variable Type inference was added in JDK 10.

Reserved type name var is allowed in JDK 10 onwards for local variable declarations with initializers, enhanced for-loop indexes, and index variables declared in traditional for loops. For example,

var x = "Java"; //x infers to String

var m = 10; //m infers to int

The identifier var is not a keyword, hence var can still be used as variable name, method name or package name but it cannot be used as a class or interface name.

It is a valid loop syntax in which var infers to int type, loop executes 4 times and at the end 'str' refers to "\*\*\*\*\*". It will also print expected output on to the console.

for(int i = 0; i < 5; i++) { str += "\*"; }: Loop syntax is correct but this loop executes 5 times and at the end 'str' refers to "\*\*\*\*\*\*". It will not print expected output.

Bottom of Form

Top of Form

Question 51: **Correct**

**Question ID: UKOCP26089**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. final String str = "+";
6. System.out.println(str.repeat(2) == "++");
7. }
8. }

What will be the result of compiling and executing Test class?

* 

**true**

* 

**false**

**(Correct)**

* 

**+**

* 

**++**

**Explanation**

**UKOCP26089:**

Instance method 'repeat()' has been added to String class in Java 11 and it has the signature: `public String repeat(int count) {}`

It returns the new String object whose value is the concatenation of this String repeated 'count' times. For example,

"A".repeat(3); returns "AAA".

repeat(count) method uses constructor of String class to construct the new Object and hence newly created String object is a non-pool object.

"++" is a String literal, so it is a pool object. Hence, `str.repeat(2) == "++"` evaluates to false.

Bottom of Form

Top of Form

Question 52: **Correct**

**Question ID: UKOCP74125**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String str = "PANIC";
6. StringBuilder sb = new StringBuilder("THET");
7. System.out.println(str.replace("N", sb)); //Line n1
8. }
9. }

What will be the result of compiling and executing Test class?

* 

**PANIC**

* 

**PATHETIC**

**(Correct)**

* 

**Line n1 causes compilation error**

* 

**Line n1 throws error at runtime**

**Explanation**

**UKOCP74125:**

String class has following two overloaded replace methods:

1. public String replace(char oldChar, char newChar) {}:

Returns a string resulting from replacing all occurrences of oldChar in this string with newChar. If no replacement is done, then source String object is returned. e.g.

"Java".replace('a', 'A') --> returns new String object "JAvA".

"Java".replace('a', 'a') --> returns the source String object "Java" (no change).

"Java".replace('m', 'M') --> returns the source String object "Java" (no change).

2. public String replace(CharSequence target, CharSequence replacement) {}:

Returns a new String object after replacing each substring of this string that matches the literal target sequence with the specified literal replacement sequence. e.g.

"Java".replace("a", "A") --> returns new String object "JAvA".

"Java".replace("a", "a") --> returns new String object "Java" (it replaces "a" with "a").

"Java".replace("m", "M") --> returns the source String object "Java" (no change).

As String, StringBuilder and StringBuffer all implement CharSequence, hence instances of these classes can be passed to replace method. Line n1 compiles successfully and on execution replaces "N" with "THET", and hence Line n1 prints PATHETIC on to the console.

Bottom of Form

Top of Form

Question 53: **Correct**

**Question ID: UKOCP20300**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. boolean flag1 = "Java" == "Java".replace('J', 'J'); //Line n1
6. boolean flag2 = "Java" == "Java".replace("J", "J"); //Line n2
7. System.out.println(flag1 && flag2);
8. }
9. }

What will be the result of compiling and executing Test class?

* 

**Line n1 causes compilation error**

* 

**Line n2 causes compilation error**

* 

**true**

* 

**false**

**(Correct)**

**Explanation**

**UKOCP20300:**

String class has following two overloaded replace methods:

1. public String replace(char oldChar, char newChar) {}:

Returns a string resulting from replacing all occurrences of oldChar in this string with newChar. If no replacement is done, then source String object is returned. e.g.

"Java".replace('a', 'A') --> returns new String object "JAvA".

"Java".replace('a', 'a') --> returns the source String object "Java" (no change).

"Java".replace('m', 'M') --> returns the source String object "Java" (no change).

2. public String replace(CharSequence target, CharSequence replacement) {}:

Returns a new String object after replacing each substring of this string that matches the literal target sequence with the specified literal replacement sequence. e.g.

"Java".replace("a", "A") --> returns new String object "JAvA".

"Java".replace("a", "a") --> returns new String object "Java" (it replaces "a" with "a").

"Java".replace("m", "M") --> returns the source String object "Java" (no change).

For Line n1, as both oldChar and newChar are same, hence source String ("Java") is returned by `"Java".replace('J', 'J');` without any change. flag1 stores true.

For Line n2, even though target and replacement are same but as "J" is found in the source String, hence a new String object "Java" is returned by `"Java".replace("J", "J");` after replacing "J" with "J". flag2 stores false.

flag1 && flag2 evaluates to false.

Bottom of Form

Top of Form

Question 54: **Correct**

**Question ID: UKOCP89618**

Consider below code fragment:

1. String place = "MISSS";
2. System.out.println(place.replace("SS", "T"));

What is the output?

* 

**MIST**

* 

**MITS**

**(Correct)**

* 

**MISSS**

* 

**MIT**

**Explanation**

**UKOCP89618:**

According to Javadoc, replace(CharSequence target, CharSequence replacement) method of String class returns a new String object after replacing each substring of this string that matches the literal target sequence with the specified literal replacement sequence. The replacement proceeds from the beginning of the string to the end, for example, replacing "aa" with "b" in the string "aaa" will result in "ba" rather than "ab".

"MISSS".replace("SS", "T"); returns "MITS".

Bottom of Form

Top of Form

Question 55: **Correct**

**Question ID: UKOCP38071**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String str = "ALASKA";
6. System.out.println(str.charAt(str.indexOf("A") + 1));
7. }
8. }

What will be the result of compiling and executing Test class?

* 

**A**

* 

**L**

**(Correct)**

* 

**S**

* 

**K**

* 

**Runtime error**

**Explanation**

**UKOCP38071:**

`int indexOf(String str)` method of String class returns the index within this string of the first occurrence of the specified substring. e.g. "Java".indexOf("a") returns 1.

`char charAt​(int index)` method of String class returns the char value at the specified index. e.g. "Java".charAt(2) returns 'v'.

Let's check the given expression:

str.charAt(str.indexOf("A") + 1)

= "ALASKA".charAt("ALASKA".indexOf("A") + 1)

= "ALASKA".charAt(0 + 1) //"ALASKA".indexOf("A") returns 0.

= "ALASKA".charAt(1)

= 'L'

Hence, L is printed on to the console.

Bottom of Form

Top of Form

Question 56: **Correct**

**Question ID: UKOCP62143**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String [] arr = {"1st", "2nd", "3rd", "4th", "5th"};
6. String place = "faraway";
7. System.out.println(arr[place.indexOf("a", 3)]); //Line n1
8. }
9. }

What will be the result of compiling and executing Test class?

* 

**1st**

* 

**3rd**

* 

**5th**

* 

**2nd**

* 

**4th**

**(Correct)**

* 

**An exception is raised by Line n1**

**Explanation**

**UKOCP62143:**

`int indexOf​(String str, int fromIndex)` method of String class returns the index within this string of the first occurrence of the specified substring, starting at the specified index. e.g.

"alaska".indexOf("a", 1) returns 2

"alaska".indexOf("a", 2) returns 2

"alaska".indexOf("a", 3) returns 5

In the given question, 'arr' refers to a String array of size 5. Element at index 0 refers to "1st", element at index 1 refers to "2nd" and so on.

Let's solve the given expression of Line n1:

arr[place.indexOf("a", 3)]

= arr["faraway".indexOf("a", 3)] //Starts looking for "a" from index 3 of the given String "faraway" and "a" is found at index 3.

= arr[3]

= "4th" //Array element at index 3 refers to "4th".

Hence, 4th is printed on to the console.

Bottom of Form

Top of Form

Question 57: **Correct**

**Question ID: UKOCP66503**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String word = "REBUS";
6. /\* INSERT \*/
7. System.out.println(word);
8. }
9. }

Following options are available to replace /\*INSERT\*/:

1. word = word.substring(2);

2. word = word.substring(2, 4);

3. word = word.substring(2, 5);

4. word = word.replace("RE", "");

5. word = word.substring(2, 6);

6. word = word.delete(0, 2);

How many of the above options can be used to replace /\*INSERT\*/ (separately and not together) such that given command prints BUS on to the console?

* 

**One option only**

* 

**Two options only**

* 

**Three options only**

**(Correct)**

* 

**Four options only**

* 

**Five options only**

* 

**All 6 options**

**Explanation**

**UKOCP66503:**

substring(int beginIndex, int endIndex) method of String class extracts the substring, which begins at the specified beginIndex and extends to the character at index endIndex - 1.

This method throws IndexOutOfBoundsException if the beginIndex is negative, or endIndex is larger than the length of this String object, or beginIndex is larger than endIndex. e.g.

"freeway".substring(4, 7) returns "way"

"freeway".substring(4, 8) throws IndexOutOfBoundsException

substring​(int beginIndex) method of String class extracts the substring, which begins with the character at the specified index and extends to the end of this string.

This method throws IndexOutOfBoundsException if beginIndex is negative or larger than the length of this String object. e.g.

"freeway".substring(4) returns "way"

"freeway".substring(8) throws IndexOutOfBoundsException

replace(CharSequence target, CharSequence replacement) method of String class returns a new String object after replacing each substring of this string that matches the literal target sequence with the specified literal replacement sequence. e.g.

"Java".replace("a", "A") --> returns new String object "JAvA".

Let's check all the given options:

"REBUS".substring(2); [begin = 2, end = 4 (end of the string)], returns "BUS" and hence it is a correct option.

"REBUS".substring(2, 4); [begin = 2, end = 3 (endIndex - 1)], returns "BU" and hence it is incorrect option.

"REBUS".substring(2, 5); [begin = 2, end = 4 (endIndex - 1)], returns "BUS" and hence it is a correct option.

"REBUS".replace("RE", ""); It replaces "RE" with empty string "" and returns "BUS", so it is also a correct option.

"REBUS".substring(2, 6); Length of "REBUS" = 5 and endIndex = 6, which is greater than 5, hence it will thrown IndexOutOfBoundsException at runtime. Incorrect option

"REBUS".delete(0, 2); Compilation error as delete(...) method is not available in String class, it is part of StringBuilder class. Incorrect option.

So, total 3 options will replace /\*INSERT\*/ to print BUS on to the console.

Bottom of Form

Top of Form

Question 58: **Correct**

**Question ID: UKOCP67831**

Consider below 4 code snippets:

1.

1. String [] arr1 = {null, null};
2. System.out.println("1. " + String.join("::", arr1));

2.

1. String [] arr2 = {};
2. System.out.println("2. " + String.join("-", arr2));

3.

1. String [] arr3 = null;
2. System.out.println("3. " + String.join(".", arr3));

4.

System.out.println("4. " + String.join(".", null));

Which of the following statements are correct about above snippets?

Select 4 options.

* 

**Snippet 1 compiles successfully**

**(Correct)**

* 

**Snippet 2 compiles successfully**

**(Correct)**

* 

**Snippet 3 compiles successfully**

**(Correct)**

* 

**Snippet 4 compiles successfully**

* 

**Snippet 1 throws runtime exception**

* 

**Snippet 2 throws runtime exception**

* 

**Snippet 3 throws runtime exception**

**(Correct)**

* 

**Snippet 4 throws runtime exception**

**Explanation**

**UKOCP67831:**

Static overloaded method join(...) was added in JDK 1.8 and has below declarations:

1. public static String join(CharSequence delimiter, CharSequence... elements) {...}: It returns a new String composed of copies of the CharSequence elements joined together with a copy of the specified delimiter.

For example,

String.join(".", "A", "B", "C"); returns "A.B.C"

String.join("+", new String[]{"1", "2", "3"}); returns "1+2+3"

String.join("-", "HELLO"); returns "HELLO"

If delimiter is null or elements refer to null, then NullPointerException is thrown. e.g.,

String.join(null, "A", "B"); throws NullPointerException

String [] arr = null; String.join("-", arr); throws NullPointerException

But if single element is null, then "null" is considered. e.g.,

String str = null; String.join("-", str); returns "null"

String.join("::", new String[] {"James", null, "Gosling"}); returns "James::null::Gosling"

2. public static String join​(CharSequence delimiter, Iterable<? extends CharSequence> elements) {...}: It returns a new String composed of copies of the CharSequence elements joined together with a copy of the specified delimiter.

For example,

String.join(".", List.of("A", "B", "C")); returns "A.B.C"

String.join(".", List.of("HELLO")); returns "HELLO"

If delimiter is null or elements refer to null, then NullPointerException is thrown. e.g.,

String.join(null, List.of("HELLO")); throws NullPointerException

List<String> list = null; String.join("-", list); throws NullPointerException

But if single element is null, then "null" is considered. e.g.,

List<String> list = new ArrayList<>(); list.add("A"); list.add(null); String.join("::", list); returns "A::null"

Please note: String.join("-", null); causes compilation error as compiler is unable to tag this call to specific join(...) method. It is an ambiguous call.

Based on above points:

Snippet 1 compiles successfully and on execution prints "1. null::null" on to the console.

Snippet 2 compiles successfully and on execution prints "2. " on to the console.

Snippet 3 compiles successfully and on execution throws NullPointerException as elements refer to null

Snippet 4 causes compilation error as `String.join(".", null)` is an ambiguous call and compiler is not sure about which overloaded join(...) method should be tagged for this call.

Bottom of Form

Top of Form

Question 59: **Correct**

**Question ID: UKOCP66500**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String str = "USER GENERATED";
6. str = String.join("-", str.split(" "));
7. System.out.println(str);
8. }
9. }

What is the result of compiling and executing Test class?

* 

**USER GENERATED**

* 

**USER-GENERATED**

**(Correct)**

* 

**-**

* 

**Compilation error**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP66500:**

To know more about join and split methods of String class, please check the URLs:

https://udayankhattry.com/join-string/

https://udayankhattry.com/split-string/

Let's solve the given expression:

Variable 'str' refers to "USER GENERATED".

str = String.join("-", str.split(" "));

str = String.join("-", "USER GENERATED".split(" "));

str = String.join("-", ["USER","GENERATED"]); //Splits on the basis of single space " ".

str = "USER-GENERATED";

`System.out.println(str);` prints USER-GENERATED on to the console.

Bottom of Form

Top of Form

Question 60: **Correct**

**Question ID: UKOCP16285**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String str = "BEVERAGE";
6. String [] arr = str.split("E", 3);
7. System.out.println(String.join(".", arr));
8. }
9. }

What is the result of compiling and executing Test class?

* 

**BEVERAGE**

* 

**B.VERAGE**

* 

**B.V.RAGE**

**(Correct)**

* 

**B.V.RAG.**

* 

**B.V.RAG..**

* 

**Compilation error**

**Explanation**

**UKOCP16285:**

To know more about join and split methods of String class, please check the URLs:

https://udayankhattry.com/join-string/

https://udayankhattry.com/split-string/

Let's solve the given expression:

Variable 'str' refers to "BEVERAGE".

str.split("E", 3); returns ["B","V","RAGE"] as pattern is applied 3 - 1 = 2 times.

String.join(".", arr); = String.join(".", ["B","V","RAGE"]); returns "B.V.RAGE" and last statement prints B.V.RAGE on to the console.

Bottom of Form

Top of Form

Question 61: **Correct**

**Question ID: UKOCP10981**

Given the code fragment:

1. String txt = "an";
2. System.out.println(txt.split("an").length);

What is the result?

* 

**0**

**(Correct)**

* 

**1**

* 

**2**

* 

**3**

* 

**Compilation error**

**Explanation**

**UKOCP10981:**

Instance overloaded method split(...) has below declarations:

1. public String[] split​(String regex, int limit) {...}: It returns the String[] after splitting this string around matches of the given regex.

The limit parameter controls the number of times the pattern is applied and therefore affects the length of the resulting array.

A. If the limit is positive then the pattern will be applied at most (limit - 1) times, the array's length will be no greater than limit, and the array's last entry will contain all input beyond the last matched delimiter. For example:

"BAZAAR".split("A", 1); returns ["BAZAAR"] as pattern is applied 1 - 1 = 0 time.

"BAZAAR".split("A", 2); returns ["B","ZAAR"] as pattern is applied 2 - 1 = 1 time.

"BAZAAR".split("A", 3); returns ["B","Z","AR"] as pattern is applied 3 - 1 = 2 times.

"BAZAAR".split("A", 4); returns ["B","Z","","R"] as pattern is applied 4 - 1 = 3 times.

"BAZAAR".split("A", 4); returns ["B","Z","","R"] as pattern needs to be applied 5 - 1 = 4 times but it is applied 3 times (which is max).

":".split(":", 2); returns ["",""] as pattern is applied (2 - 1) time and there is an empty space ("") before and after ":", so resulting array contains two empty strings.

B. If the limit is zero then the pattern will be applied as many times as possible, the array can have any length, and trailing empty strings will be discarded. For example,

"BAZAAR".split("A", 0); returns ["B","Z","","R"].

":".split(":", 0); returns blank array as both the empty strings are discarded.

C. If the limit is negative then the pattern will be applied as many times as possible and the array can have any length. For example,

"BAZAAR".split("A", -10); returns ["B","Z","","R"] as pattern is applied max times, which is 3.

":".split(":", -2); returns ["",""] as pattern is applied max times, which is 3.

2. public String[] split​(String regex) {...}: It returns the String[] after splitting this string around matches of the given regular expression.

This method works as if by invoking the two-argument split method with the given expression and a limit argument of zero. Trailing empty strings are therefore not included in the resulting array.

"BAZAAR".split("A"); returns ["B","Z","","R"] as this statement is equivalent to "BAZAAR".split("A", 0);.

":".split(":"); returns blank array as this statement is equivalent to ":".split(":", 0);.

Let's solve the given expression based on above points:

txt.split("an").length

="an".split("an").length

=[].length //"an".split("an") returns blank array as both the empty strings are discarded.

=0

Hence, 0 is printed on to the console.

Bottom of Form

Top of Form

Question 62: **Correct**

**Question ID: UKOCP67836**

Given code:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String str1 = "Book"; //Line n1
6. String str2 = new String(new char[] {'B', 'o', 'o', 'k'}); //Line n2
7. String str3 = new String("Book"); //Line n3
8. System.out.println((str1 == str2) + ":" + (str1 == str2.intern())); //Line n4
9. System.out.println((str2 == str3) + ":" + (str2.intern() == str3.intern())); //Line n5
10. }
11. }

What is the result?

* 

**false:true**

**false:false**

* 

**false:false**

**false:true**

* 

**false:false**

**false:false**

* 

**false:true**

**false:true**

**(Correct)**

* 

**false:true**

**true:true**

* 

**true:true**

**true:true**

* 

**false:false**

**true:true**

**Explanation**

**UKOCP67836:**

Line n1 creates a String Pool instance "Book". Variable 'str1' refers to String Pool instance "Book".

At Line n2, variable 'str2' refers to non-pool instance "Book". Therefore, str1 == str2 evaluates to false.

At Line n3, variable 'str3' refers to another no-pool instance "Book". Therefore, str2 == str3 evaluates to false.

According to Javadoc, when the intern() method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

As, at Line n1, "Book" object is already referred by String Pool, therefore, str2.intern() and str3.intern() would refer to same String Pool object.

Line n4 prints false:true and Line n5 prints false:true.

Bottom of Form

Top of Form

Question 63: **Incorrect**

**Question ID: UKOCP44745**

Given code:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String s1 = new String("ALL IS WELL"); //Line n1
6. String s2 = new String("ALL IS WELL"); //Line n2
7. System.out.println(s1 == s2); //Line n3
8. }
9. }

Which of the following changes, done independently, allows the code to compile and on execution prints true?

Select 3 options.

* 

**Replace Line n1 with: String s1 = "ALL IS WELL";**

**Replace Line n2 with: String s2 = "ALL IS WELL";**

**(Correct)**

* 

**Replace Line n1 with: String s1 = new String("ALL IS WELL").intern();**

**Replace Line n2 with: String s2 = new String("ALL IS WELL").intern();**

**(Correct)**

* 

**Replace Line n2 with: String s2 = s1.toString();**

**(Correct)**

* 

**Replace Line n3 with: System.out.println(s1.toString() == s2.toString());**

**(Incorrect)**

**Explanation**

**UKOCP44745:**

s1 refers to non-pool String object and s2 refers to different non-pool String object and that is why s1 == s2 evaluates to false.

Let's check all the options one by one:

Replace Line n1 with: String s1 = "ALL IS WELL";

Replace Line n2 with: String s2 = "ALL IS WELL";

✓  "ALL IS WELL" is String literal and hence represents String-Pool object. Both 's1' and 's2' refer to same String object.

Replace Line n1 with String s1 = new String("ALL IS WELL").intern();

Replace Line n2 with String s2 = new String("ALL IS WELL").intern();

✓  According to Javadoc, when the intern() method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

After intern() method is invoked, both 's1' and 's2' refer to same String object.

Replace Line n2 with: String s2 = s1.toString();

✓  's1' refers to non-pool String object.

Below is the definition of toString() method of String class:

public String toString() {

    return this;

}

It returns the same instance, on which toString() method is invoked. Hence, `s1.toString();` returns the same object referred by s1. s2 and s1 both refer to same non-pool String object.

Replace Line n3 with: System.out.println(s1.toString() == s2.toString());

✗  `s1.toString();` returns the same String object referred by s1 and `s2.toString();` returns the same String object referred by s2. As 's1' and 's2' refer to different String objects, hence  `s1.toString() == s2.toString()` evaluates to false.

Bottom of Form

Top of Form

Question 64: **Correct**

**Question ID: UKOCP78958**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder("Hakuna"); //Line n3
6. change(sb); //Line n4
7. System.out.println(sb); //Line n5
8. }
10. private static void change(StringBuilder s) {
11. s.append("\_Matata"); //Line n9
12. }
13. }

What will be the result of compiling and executing Test class?

* 

**Hakuna**

* 

**\_Matata**

* 

**Hakuna\_Matata**

**(Correct)**

* 

**None of the other options**

**Explanation**

**UKOCP78958:**

At Line n3, 'sb' refers to StringBuilder object {"Hakuna"}.

When change method is called, both the variables 's' and 'sb' refer to same StringBuilder object {"Hakuna"}.

Line n9 modifies the passed object and appends "\_Matata" to it. As a result, 's' now refers to "Hakuna\_Matata" and 'sb' also refers to "Hakuna\_Matata".

So, when control goes back to calling method main(String[]), Line n5 prints "Hakuna\_Matata" on to the console.

Bottom of Form

Top of Form

Question 65: **Correct**

**Question ID: UKOCP83084**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder("HavePatience");
6. sb.delete(4, 5).insert(4, " P").toString().toUpperCase();
7. System.out.println(sb);
8. }
9. }

What will be the result of compiling and executing Test class?

* 

**HAVE PTIENCE**

* 

**HAVE PATIENCE**

* 

**Have Patience**

**(Correct)**

* 

**Have Ptience**

* 

**HavePatience**

* 

**HAVEPATIENCE**

* 

**Haveatience**

* 

**HAVEATIENCE**

**Explanation**

**UKOCP83084:**

sb --> {"HavePatience"}

sb.delete(4, 5) --> {"Haveatience"}

sb.insert(4, " P") -> {"Have Patience"}

sb.toString() -> Creates a new String object "Have Patience"

"Have Patience".toUpperCase() -> Creates another String object "HAVE PATIENCE" but the String object is not referred and used.

Method invocation on sb modifies the same object, so after insert(4, " P") method invocation, 'sb' refers to {"Have Patience"}.

StringBuilder class overrides toString() method, which returns the contents of StringBuilder object in the form of String.

`System.out.println(sb);` invokes the sb.toString() method, which returns "Have Patience" and the same is printed on to the console.

Bottom of Form

Top of Form

Question 66: **Correct**

**Question ID: UKOCP56726**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. String str = "Game on"; //Line n1
6. StringBuilder sb = new StringBuilder(str); //Line n2
8. System.out.println(str.contentEquals(sb)); //Line n3
9. System.out.println(sb.contentEquals(str)); //Line n4
10. System.out.println(sb.equals(str)); //Line n5
11. System.out.println(str.equals(sb)); //Line n6
12. }
13. }

Which of the following statements is correct?

* 

**Only one statement causes compilation error**

**(Correct)**

* 

**Two statements cause compilation error**

* 

**Three statements cause compilation error**

* 

**Four statements cause compilation error**

* 

**No compilation error**

**Explanation**

**UKOCP56726:**

There are no issues with Line n1 and Line n2, both the statements compile successfully.

String class contains contentEquals(CharSequence) method. Please note that String, StringBuilder and StringBuffer classes implement CharSequence interface, hence contentEquals(CharSequence) method defined in String class cab be invoked with the argument of either String or StringBuilder or StringBuffer.

At Line n3, `str.contentEquals(sb)` is invoked with StringBuilder argument and hence it compiles fine. On execution it would compare the contents of String object and the passed StringBuilder object. As both the String object and StringBuilder object contains same content "Game on", hence on execution, Line n3 will print true.

contentEquals method is not available in StringBuilder class and hence Line n4 causes compilation error.

equals method declared in Object class has the declaration: `public boolean equals(Object)`. Generally, equals method is used to compare different instances of same class but if you pass any other object, there is no compilation error. Parameter type is Object so it can accept any Java object.

`str.equals(sb)` => It compiles fine, String class overrides equals(Object) method but as 'sb' is of StringBuilder type so `str.equals(sb)` would return false at runtime.

`sb.equals(str)` => It also compiles fine, StringBuilder class doesn't override equals(Object) method. So Object version is invoked which uses == operator, hence `sb.equals(str)` would return false as well at runtime.

Bottom of Form

Top of Form

Question 67: **Correct**

**Question ID: UKOCP75465**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder("Dream BIG");
6. String s1 = sb.toString();
7. String s2 = sb.toString();
9. System.out.println(s1 == s2);
10. }
11. }

What will be the result of compiling and executing Test class?

* 

**Compilation error**

* 

**true**

* 

**false**

**(Correct)**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP75465:**

toString() method defined in StringBuilder class doesn't use String literal rather uses the constructor of String class to create the instance of String class.

So both 's1' and 's2' refer to different String instances even though their contents are same. s1 == s2 returns false.

Bottom of Form

Top of Form

Question 68: **Correct**

**Question ID: UKOCP73287**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder("Breathe Deeply");
6. String str1 = sb.toString();
7. String str2 = "Breathe Deeply";
9. System.out.println(str1 == str2);
10. }
11. }

What will be the result of compiling and executing Test class?

* 

**Compilation error**

* 

**true**

* 

**false**

**(Correct)**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP73287:**

toString() method defined in StringBuilder class doesn't use String literal rather uses the constructor of String class to create the instance of String class.

So both 'str1' and 'str2' refer to different String instances even though their contents are same. str1 == str2 returns false.

Bottom of Form

Top of Form

Question 69: **Correct**

**Question ID: UKOCP74132**

Consider below code:

1. //Test.java
2. package com.udayankhattry.ocp;
4. class SpecialString {
5. String str;
6. SpecialString(String str) {
7. this.str = str;
8. }
9. }
11. public class Test {
12. public static void main(String[] args) {
13. Object [] arr = new Object[4];
14. for(int i = 1; i <=3; i++) {
15. switch(i) {
16. case 1:
17. arr[i] = new String("Java");
18. break;
19. case 2:
20. arr[i] = new StringBuilder("Java");
21. break;
22. case 3:
23. arr[i] = new SpecialString("Java");
24. break;
25. }
26. }
27. for(Object obj : arr) {
28. System.out.println(obj);
29. }
30. }
31. }

What will be the result of compiling and executing Test class?

* 

**Java**

**Java**

**Java**

* 

**Java**

**Java**

**<Some text containing @ symbol>**

* 

**Java**

**<Some text containing @ symbol>**

**<Some text containing @ symbol>**

* 

**null**

**Java**

**Java**

**Java**

* 

**null**

**Java**

**Java**

**<Some text containing @ symbol>**

**(Correct)**

* 

**null**

**Java**

**<Some text containing @ symbol>**

**<Some text containing @ symbol>**

* 

**Java**

**Java**

**Java**

**null**

* 

**Java**

**Java**

**<Some text containing @ symbol>**

**null**

* 

**Java**

**<Some text containing @ symbol>**

**<Some text containing @ symbol>**

**null**

* 

**null**

**Java**

**null**

**null**

**Explanation**

**UKOCP74132:**

Variable 'arr' refers to an Object array of size 4 and null is assigned to all 4 elements of this array.

for-loop starts with i = 1, which means at 1st index String instance is stored, at 2nd index StringBuiler instance is stored and at 3rd index SpecialString instance is stored. null is stored at 0th index.

So, first null will be printed on to the console.

String and StringBuilder classes override toString() method, which prints the text stored in these classes. SpecialString class doesn't override toString() method and hence when instance of SpecialString is printed on to the console, you get: <fully qualified name of SpecialString class>@<hexadecimal representation of hashcode>.

Therefore output will be:

null

Java

Java

<Some text containing @ symbol>

Bottom of Form

Top of Form

Question 70: **Correct**

**Question ID: UKOCP20791**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder(100);
6. System.out.println(sb.length() + ":" + sb.toString().length());
7. }
8. }

What will be the result of compiling and executing Test class?

* 

**100:100**

* 

**100:0**

* 

**16:16**

* 

**16:0**

* 

**0:0**

**(Correct)**

**Explanation**

**UKOCP20791:**

`new StringBuilder(100);` creates a StringBuilder instance, whose internal char array's length is 100 but length() method of StringBuilder object returns the number of characters stored in the internal array and in this case it is 0. So, `sb.length()` returns 0.

sb.toString() is the String representation of StringBuilder instance and in this case as there are no characters inside the StringBuilder object, hence `sb.toString()` returns an empty String "", so `sb.toString().length()` also returns 0.

Output is 0:0.

Bottom of Form

Top of Form

Question 71: **Correct**

**Question ID: UKOCP55873**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder("Friends are treasures");
6. sb.delete(0, 100);
7. System.out.println(sb.length());
8. }
9. }

What will be the result of compiling and executing Test class?

* 

**21**

* 

**0**

**(Correct)**

* 

**16**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP55873:**

'delete' method accepts 2 parameters: delete(int start, int end), where start is inclusive and end is exclusive.

This method throws StringIndexOutOfBoundsException for following scenarios:

A. start is negative

B. start is greater than sb.length()

C. start is greater than end

If end is greater than the length of StringBuilder object, then StringIndexOutOfBoundsException is not thrown and end is set to sb.length().

So, in this case, `sb.delete(0, 100);` is equivalent to `sb.delete(0, sb.length());` and this deletes all the characters from the StringBuilder object.

Hence, `System.out.println(sb.length());` prints 0 on to the console.

Bottom of Form

Top of Form

Question 72: **Correct**

**Question ID: UKOCP78974**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder(5);
6. sb.append("0123456789");
7. sb.delete(8, 1000);
8. System.out.println(sb);
9. }
10. }

What will be the result of compiling and executing Test class?

* 

**Compilation error**

* 

**An exception is thrown at runtime**

* 

**01234567**

**(Correct)**

* 

**89**

**Explanation**

**UKOCP78974:**

`new StringBuilder(5);` creates a StringBuilder instance, whose internal char array's length is 5 but the internal char array's length is adjusted when characters are added/removed from the StringBuilder instance. `sb.append("0123456789");` successfully appends "0123456789" to the StringBuilder's instance referred by sb.

delete method accepts 2 parameters: delete(int start, int end), where start is inclusive and end is exclusive.

This method throws StringIndexOutOfBoundsException for following scenarios:

A. start is negative

B. start is greater than sb.length()

C. start is greater than end

If end is greater than the length of StringBuilder object, then StringIndexOutOfBoundsException is not thrown and end is set to sb.length().

So, in this case, `sb.delete(8, 1000);` is equivalent to `sb.delete(8, sb.length());` and this deletes characters at 8th index (8) and 9th index (9). So remaining characters are: "01234567".

StringBuilder class overrides toString() method, which prints the text stored in StringBuilder instance. Hence, `System.out.println(sb);` prints 01234567 on to the console.

Bottom of Form

Top of Form

Question 73: **Correct**

**Question ID: UKOCP52364**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder();
6. System.out.println(sb.append("").append("").append("").length());
7. }
8. }

What will be the result of compiling and executing Test class?

* 

**0**

**(Correct)**

* 

**1**

* 

**2**

* 

**3**

**Explanation**

**UKOCP52364:**

As "" is empty string, hence nothing is appended to the StringBuilder instance and length() method returns 0.

Bottom of Form

Top of Form

Question 74: **Correct**

**Question ID: UKOCP23824**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder();
6. System.out.println(sb.append(null).length());
7. }
8. }

What will be the result of compiling and executing Test class?

* 

**1**

* 

**4**

* 

**Compilation error**

**(Correct)**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP23824:**

'append' method is overloaded in StringBuilder class: append(String), append(StringBuffer) and append(char[]) etc. In this case compiler gets confused as to which method `append(null)` can be tagged because String, StringBuffer and char[] are not related to each other in multilevel inheritance. Hence `sb.append(null)` causes compilation error.

Bottom of Form

Top of Form

Question 75: **Incorrect**

**Question ID: UKOCP44744**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder("TOMATO");
6. System.out.println(sb.reverse().replace("O", "A")); //Line n1
7. }
8. }

What will be the result of compiling and executing Test class?

* 

**TOMATO**

* 

**TAMATO**

* 

**TAMATA**

* 

**OTAMOT**

* 

**OTAMAT**

* 

**ATAMAT**

**(Incorrect)**

* 

**Compilation error**

**(Correct)**

**Explanation**

**UKOCP44744:**

sb --> {"TOMATO"}

sb.reverse() --> {"OTAMOT"}. reverse() method returns a StringBuilder object.

replace method of StringBuilder class accepts 3 arguments: `replace(int start, int end, String str)`. At Line n1, replace("O", "A") method accepts 2 arguments and hence it causes compilation error.

Bottom of Form

Top of Form

Question 76: **Correct**

**Question ID: UKOCP16290**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder("ELECTROTHERMAL"); //Line n1
6. sb.setLength(7); //Line n2
7. System.out.print(sb.toString().strip()); //Line n3
8. System.out.print(":"); //Line n4
9. sb.setLength(14); //Line n5
10. System.out.println(sb.toString().strip()); //Line n6
11. }
12. }

What will be the result of compiling and executing Test class?

* 

**THERMAL:ELECTROTHERMAL**

* 

**ELECTRO:ELECTROTHERMAL**

* 

**ELECTROTHERMAL:THERMAL**

* 

**ELECTROTHERMAL:ELECTRO**

* 

**ELECTRO:ELECTRO**

**(Correct)**

* 

**ELECTRO:THERMAL**

* 

**THERMAL:ELECTRO**

* 

**THERMAL:THERMAL**

**Explanation**

**UKOCP16290:**

strip() method of String class (available since Java 11) returns a string whose value is this string, with all leading and trailing white space removed.

To find out what is white space character in Java, check Character.isWhitespace​(int) method, you will find below:

A character is a Java whitespace character if and only if it satisfies one of the following criteria:

It is a Unicode space character (SPACE\_SEPARATOR, LINE\_SEPARATOR, or PARAGRAPH\_SEPARATOR) but is not also a non-breaking space ('\u00A0', '\u2007', '\u202F').

It is '\t', U+0009 HORIZONTAL TABULATION.

It is '\n', U+000A LINE FEED.

It is '\u000B', U+000B VERTICAL TABULATION.

It is '\f', U+000C FORM FEED.

It is '\r', U+000D CARRIAGE RETURN.

It is '\u001C', U+001C FILE SEPARATOR.

It is '\u001D', U+001D GROUP SEPARATOR.

It is '\u001E', U+001E RECORD SEPARATOR.

It is '\u001F', U+001F UNIT SEPARATOR.

White space character in java includes space character along with above characters.

At Line n1, 'sb' refers to StringBuilder object {'E','L','E','C','T','R','O','T','H','E','R','M','A,'L'}. So length of this StringBuilder object is 14.

At Line n2, `sb.setLength(7);` sets the length of StringBuilder object referred by 'sb' to 7 (it is reducing the length of StringBuilder object from 14 to 7), hence 'sb' refers to {'E','L','E','C','T','R','O'}. Last 7 characters are gone.

There is no white space character in above StringBuilder object, hence at Line n3 `sb.toString()` returns "ELECTRO" and strip() method at Line n3 returns "ELECTRO".

"ELECTRO" is printed on to the console.

When code reaches Line n4, console shows "ELECTRO:"

At Line n5, `sb.setLength(14);` sets the length of StringBuilder object referred by 'sb' to 7 (it is increasing the length of StringBuilder object by 7 by filling available space with null characters '\u0000'). Now 'sb' refers to StringBuilder object {'E','L','E','C','T','R','O','\u0000','\u0000','\u0000','\u0000','\u0000','\u0000,'\u0000'}.

At Line n6, `sb.toString()` returns corresponding String object "ELECTRO       ". As '\u0000' is white space character, hence "ELECTRO       ".strip() returns "ELECTRO". Line n6 prints "ELECTRO" on to the console.

Hence, the output is ELECTRO:ELECTRO.

Bottom of Form

Top of Form

Question 77: **Correct**

**Question ID: UKOCP20799**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder(20); //Line n1
6. sb.append("A".repeat(25)); //Line n2
7. System.out.println(sb.toString().length()); //Line n3
9. sb.setLength(10); //Line n4
10. System.out.println(sb.toString().length()); //Line n5
11. }
12. }

What will be the result of compiling and executing Test class?

* 

**20**

**20**

* 

**25**

**10**

**(Correct)**

* 

**20**

**10**

* 

**25**

**25**

* 

**10**

**10**

**Explanation**

**UKOCP20799:**

`new StringBuilder(20);` creates a StringBuilder instance, whose capacity (internal char array's length) is 20. This initial capacity is not fixed and changes on adding / removing characters to the StringBuilder object.

Instance method 'repeat()' has been added to String class in Java 11 and it has the signature: `public String repeat(int count) {}`

It returns the new String object whose value is the concatenation of this String repeated 'count' times. For example,

"A".repeat(3); returns "AAA".

Line n2 successfully appends 25 A's to the StringBuilder object.

Line n3 prints 25 on to the console.

At Line n4, `sb.setLength(10);` sets the length of StringBuilder object referred by 'sb' to 10 (it is reducing the length of StringBuilder object from 25 to 10), hence 'sb' refers to StringBuilder object containing 10 A's (last 15 A's are gone).

Line n5 prints 10 on to the console.

Bottom of Form

Top of Form

Question 78: **Correct**

**Question ID: UKOCP75460**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder("B"); //Line n1
6. sb.append(sb.append("A")); //Line n2
7. System.out.println(sb); //Line n3
8. }
9. }

What will be the result of compiling and executing Test class?

* 

**B**

* 

**BA**

* 

**AB**

* 

**BAB**

* 

**ABA**

* 

**ABAB**

* 

**BABA**

**(Correct)**

* 

**ABBA**

* 

**Compilation error at Line n2**

**Explanation**

**UKOCP75460:**

At Line n1:

sb --> {"B"}

append(...) method in StringBuilder class is overloaded to accept various arguments and 2 such arguments are String and CharSequence. It's return type is StringBuilder and as StringBuilder class implements CharSequence interface, hence 'sb.append("A")' can easily be passed as and argument to sb.append(...) method. Line n2 compiles successfully.

At Line n2:

sb.append(sb.append("A")); //sb --> {"B"}

sb.append({"BA"}); //sb --> {"BA"}

{"BABA"}

Hence, Line n3 prints BABA

Bottom of Form

Top of Form

Question 79: **Correct**

**Question ID: UKOCP43413**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. static String var = "FRIENDS"; //Line n1
5. public static void main(String[] args) {
6. int var = (var = Test.var.length()); //Line n2
7. System.out.println(var); //Line n3
8. }
9. }

What will be the result of compiling and executing Test class?

* 

**Line n1 causes compilation error**

* 

**Line n2 causes compilation error**

* 

**Line n3 causes compilation error**

* 

**It compiles successfully and on execution prints FRIENDS on to the console**

* 

**It compiles successfully and on execution prints 7 on to the console**

**(Correct)**

**Explanation**

**UKOCP43413:**

Local variable Type inference was added in JDK 10.

Reserved type name var is allowed in JDK 10 onwards for local variable declarations with initializers, enhanced for-loop indexes, and index variables declared in traditional for loops. For example,

var x = "Java"; //x infers to String

var m = 10; //m infers to int

The identifier var is not a keyword, hence var can still be used as variable name, method name or package name but it cannot be used as a class or interface name. Line n1 compiles successfully as static variable 'var' is of String type and refers to "FRIENDS".

At Line n2:

int var = (var = Test.var.length()); //Local variable 'var' is declared of int type and shadows static String variable 'var'

=> int var = (var = 7); //Test.var refers to "FRIENDS" and "FRIENDS".length() = 7

=> int var = 7; //7 is assigned to var

=> 7 is again assigned to var

Line n3 prints 7 on to the console.

Bottom of Form

Top of Form

Question 80: **Correct**

**Question ID: UKOCP21636**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. var x = "100"; //Line n1
6. var y = 100; //Line n2
7. System.out.println(x + y); //Line n3
8. }
9. }

What is the result of compiling and executing Test class?

* 

**Only Line n1 causes compilation error**

* 

**Only Line n2 causes compilation error**

* 

**Only Line n3 causes compilation error**

* 

**Only Line n1 and Line n3 cause compilation error**

* 

**Line n1, Line n2 and Line n3 cause compilation error**

* 

**Code compiles successfully and prints 200 on to the console**

* 

**Code compiles successfully and prints 100100 on to the console**

**(Correct)**

**Explanation**

**UKOCP21636:**

Local variable Type inference was added in JDK 10.

Reserved type name var is allowed in JDK 10 onwards for local variable declarations with initializers, enhanced for-loop indexes, and index variables declared in traditional for loops. For example,

var x = "Java"; //x infers to String

var m = 10; //m infers to int

The identifier var is not a keyword, hence var can still be used as variable name, method name or package name but it cannot be used as a class or interface name.

For Line n1, x infers to String type.

For Line n2, y infers to int type.

As, x is of String type and y is of int type, hence x + y results in "100100".

For more information on local variable type inference, please check the URL: http://openjdk.java.net/jeps/286

Bottom of Form

Top of Form

Question 81: **Incorrect**

**Question ID: UKOCP66992**

Which of the following code segments, written inside main method will compile successfully?

Select 3 options.

* 

**var [] arr1 = new String[2];**

* 
  1. var num;
  2. num = 10;

**(Incorrect)**

* 

**final var str = "Hello";**

**(Correct)**

* 

**var arr2 = {1, 2, 3};**

* 
  1. for (var i = 0; i <2; i++) {
  2. System.out.println(i);
  3. }

**(Correct)**

* 
  1. String [] arr = {"A", "E", "I", "O", "U"};
  2. for(var x : arr) {
  3. System.out.println(x);
  4. }

**(Correct)**

* 

**var msg = null;**

* 

**private var y = 100;**

**(Incorrect)**

**Explanation**

**UKOCP66992:**

Local variable Type inference was added in JDK 10.

Reserved type name var is allowed in JDK 10 onwards for local variable declarations with initializers, enhanced for-loop indexes, and index variables declared in traditional for loops. For example,

var x = "Java"; //x infers to String

var m = 10; //m infers to int

The identifier var is not a keyword, hence var can still be used as variable name, method name or package name but it cannot be used as a class or interface name.

But there are certain restrictions on using var type:

Declaration and initialization should be in one statement, hence `var num; num = 10;` won't compile.

var is not allowed as an element type of an array, hence `var [] arr1 = new String[2];` won't compile.

Explicit target-type is needed for the array initializer, if you use var type. Hence, `var arr2 = {1, 2, 3};` won't compile.

If you provide the target-type, `var arr2 = new int[]{1, 2, 3};`, then there is no compilation error and in this case var infers to int [] type.

If initializer is of null type, then it is not possible to infer the target type, hence `var msg = null;` won't compile.

Also note that, var type cannot be the target type of lambda expressions and method references.

`final var str = "Hello";` will compile successfully as 'str' infers to String type and final modifier is allowed for local variables.

`private var y = 100;` will not compile as private modifier is not allowed for local variables.

var can easily be used for enhanced for-loop indexes, and index variables declared in traditional for loops, hence both the loop code segments will compile successfully.

For more information on local variable type inference, please check the URL: http://openjdk.java.net/jeps/286

Bottom of Form

Top of Form

Question 82: **Correct**

**Question ID: UKOCP38066**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
5. public class Test {
6. private var place = "Unknown"; //Line n1
7. public static final var DISTANCE = 200; //Line n2
9. public static void main(String[] args) {
10. var list1 = new ArrayList<>(); //Line n3
12. var list2 = new ArrayList(); //Line n4
14. var lambda1 = () -> System.out.println("Hello"); //Line n5
16. var var = 100; //Line n6
17. }
18. }

Which of the following statements are correct?

Select 3 options.

* 

**Line n1 compiles successfully**

* 

**Line n2 compiles successfully**

* 

**Line n3 compiles successfully**

**(Correct)**

* 

**Line n4 compiles successfully**

**(Correct)**

* 

**Line n5 compiles successfully**

* 

**Line n6 compiles successfully**

**(Correct)**

**Explanation**

**UKOCP38066:**

Local variable Type inference was added in JDK 10.

Reserved type name var is allowed in JDK 10 onwards for local variable declarations with initializers, enhanced for-loop indexes, and index variables declared in traditional for loops. For example,

var x = "Java"; //x infers to String

var m = 10; //m infers to int

As the name suggests, Local variable Type inference is applicable only for local variables and not for instance or class variables. Hence, Line n1 and Line n2 cause compilation error.

For Line n3, 'list1' infers to ArrayList<Object> type and for Line n4, 'list2' infers to ArrayList. Both the lines n3 and n4 compile successfully.

var type cannot be the target type of lambda expressions and method references. Hence, Line n5 causes compilation error.

The identifier var is not a keyword, hence var can still be used as variable name, method name or package name but it cannot be used as a class or interface name. So, Line n6 compiles successfully.

For more information on local variable type inference, please check the URL: http://openjdk.java.net/jeps/286

Bottom of Form

Top of Form

Question 83: **Incorrect**

**Question ID: UKOCP63002**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. operate(10 , 20);
6. operate("A", "B");
7. operate('A', 1); //ASCII code of 'A' is 65
8. }
10. public static void operate(var v1, var v2) {
11. System.out.println(v1 + v2);
12. }
13. }

What is the result of compiling and executing above code?

* 

**30**

**AB**

**66**

**(Incorrect)**

* 

**30**

**AB**

**A1**

* 

**1020**

**AB**

**A1**

* 

**1020**

**AB**

**66**

* 

**Compilation error**

**(Correct)**

* 

**Runtime error**

**Explanation**

**UKOCP63002:**

Local variable Type inference was added in JDK 10.

Reserved type name var is allowed in JDK 10 onwards for local variable declarations with initializers, enhanced for-loop indexes, and index variables declared in traditional for loops. For example,

var x = "Java"; //x infers to String

var m = 10; //m infers to int

The identifier var is not a keyword, hence var can still be used as variable name, method name or package name but it cannot be used as a class or interface name. But var type cannot be used as method parameters or method return type. Hence 'operate(var v1, var v2)' causes compilation error.

For more information on local variable type inference, please check the URL: http://openjdk.java.net/jeps/286

Bottom of Form

Top of Form

Question 84: **Incorrect**

**Question ID: UKOCP76309**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. System.out.println(add(90, 7));
6. System.out.println(add('a', 1)); //ASCII code for 'a' is 97 and 'b' is 98
7. }
9. public static var add(int v1, int v2) {
10. return v1 + v2;
11. }
12. }

What is the result of compiling and executing above code?

* 

**97**

**98**

**(Incorrect)**

* 

**a**

**b**

* 

**a**

**98**

* 

**97**

**b**

* 

**Compilation error**

**(Correct)**

* 

**Runtime error**

**Explanation**

**UKOCP76309:**

Local variable Type inference was added in JDK 10.

Reserved type name var is allowed in JDK 10 onwards for local variable declarations with initializers, enhanced for-loop indexes, and index variables declared in traditional for loops. For example,

var x = "Java"; //x infers to String

var m = 10; //m infers to int

The identifier var is not a keyword, hence var can still be used as variable name, method name or package name but it cannot be used as a class or interface name. But var type cannot be used as method parameters or method return type. Hence 'var add(int v1, int v2)' causes compilation error.

For more information on local variable type inference, please check the URL: http://openjdk.java.net/jeps/286

Bottom of Form

Top of Form

Question 85: **Incorrect**

**Question ID: UKOCP30582**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. var m = 10; //Line n1
6. var n = 20; //Line n2
7. /\*INSERT\*/ p = m = n = 30; //Line n3
8. System.out.println(m + n + p); //Line n4
9. }
10. }

Which of the following options, if used to replace /\*INSERT\*/, will compile successfully and on execution will print 90 on to the console?

Select 3 options.

* 

**byte**

**(Incorrect)**

* 

**short**

**(Incorrect)**

* 

**int**

**(Correct)**

* 

**long**

**(Correct)**

* 

**float**

* 

**double**

* 

**var**

**(Correct)**

**Explanation**

**UKOCP30582:**

Local variable Type inference was added in JDK 10.

Reserved type name var is allowed in JDK 10 onwards for local variable declarations with initializers, enhanced for-loop indexes, and index variables declared in traditional for loops. For example,

var x = "Java"; //x infers to String

var m = 10; //m infers to int

At Line n1, m infers to int type as 10 is an int literal.

At Line n2, n infers to int type as 20 is an int literal.

var type can easily replace /\*INSERT\*/ and in this case p would infer to int type.

As m and n are non-final variables, hence /\*INSERT\*/ cannot be replaced with byte or short because int variable cannot be implicitly casted to byte or short types.

int, long, float and double types can also replace /\*INSERT\*/ without causing any error but for float and double type output will be 90.0 where as for int, long and var type output will be 90.

Hence, only 3 options (int, long and var) print expected output on to the console.

For more information on local variable type inference, please check the URL: http://openjdk.java.net/jeps/286

Bottom of Form

Top of Form

Question 86: **Correct**

**Question ID: UKOCP10996**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String [] args) {
5. var x = 7.85; //Line n1
6. var y = 5.25f; //Line n2
7. var a = (int)x + (int)y; //Line n3
8. var b = (int)(x + y); //Line n4
9. System.out.println(a + b);
10. }
11. }

What will be the result of compiling and executing Test class?

* 

**Compilation error at Line n3**

* 

**Compilation error at Line n4**

* 

**24**

* 

**25**

**(Correct)**

* 

**26**

**Explanation**

**UKOCP10996:**

Local variable Type inference was added in JDK 10.

Reserved type name var is allowed in JDK 10 onwards for local variable declarations with initializers, enhanced for-loop indexes, and index variables declared in traditional for loops. For example,

var x = "Java"; //x infers to String

var m = 10; //m infers to int

The identifier var is not a keyword, hence var can still be used as variable name, method name or package name but it cannot be used as a class or interface name.

At Line n1, x infers to double type as 7.85 is double literal.

At Line n2, y infers to float type as 5.25f is a float literal.

Line n3:

var a = (int)x + (int)y;

var a = (int)7.85 + (int)5.25;

var a = 7 + 5;

var a = 12; //a infers to int type

Line n4:

var b = (int)(x + y);

var b = (int)(7.85 + 5.25);

var b = (int)(13.10);

var b = 13; //b infers to int type

Hence, `System.out.println(a + b);` prints 25 on to the console.

Bottom of Form

Retake test

**Continue**