1-) Question

Given:

```
public class A {
public static void main(String[] args) {
String[] horses = new String[5];
horses [4] = null;
for(int i = 0; i < horses.length; i++) {
if(i < args.length)
horses[i] = args [i];
System.out.print(horses[i].toUpperCase()+" ");
}
}}
```

And, if the code compiles the command line:

java A hope all is well

What is the result?

a) HOPE ALL IS WELL
b) HOPE ALL IS WELL null
c) An exception is thrown with no other output.
d) HOPE ALL IS WELL, and then a NullPointerException
e) It prints: 'HOPE ALL IS WELL' and then a ArrayIndexOutOfBoundsException.
f) Compiler Error.

1- Answer

d) is correct.

It prints: 'HOPE ALL IS WELL' and then a throws a *NullPointerException*.

Let's step through the code to see how it runs:

===Code Step Through===

```
1: public class A {
2: public static void main(String[] args) {
3: String[] horses = new String[5];
4: horses [4] = null;
5: for(int i = 0; i < horses.length; i++) {
6: if(i < args.length)
7: horses[i] = args [i];
8: System.out.print(horses[i].toUpperCase()+" ");
9: }
10: }}
```

The user runs the program and inputs their arguments as follows:

java A ***hope all is well***

This results in the array String[] 'args' getting populated with the following strings:

args[0] = "hope"
args[1] = "all"
args[2] = "is"
args[3] = "well"

=======?=======

2-) Question

Given:

```
1: class A {
2: int size;
3: A(int s) { size = s; }
4: }
5: public class B {
6: public static void main(String[] args) {
7: A b1 = new A(5);
8: A[] ba = go(b1, new A(6));
9: ba[0] = b1;
10: for(A b : ba) System.out.print(b.size + " ");
11: }
12: static A[] go(A b1, A b2) {
13: b1.size = 4;
14: A[] ma = {b2, b1};
15: return ma;
16: }}
```

What is the result?

a) 4 4

b) 5 4

c) 6 4

d) 4 5

e) 5 5

f) Compilation fails

2-) Answer, A is correct

4-)

```
public class Test {

    public static void main(String[] args) {

        String[][] chs = new String[2][];
        chs[0] = new String[2];
        chs[1] = new String[5];
        int i = 97;

        for (int a = 0; a < chs.length; a++) {
            for (int b = 0; b < chs.length; b++) {
                chs[a][b] = "" + i;
                i++;
            }
        }

        for (String[] ca : chs) {
            for (String c : ca) {
                System.out.print(c + " ");
            }
            System.out.println();
        }
    }
}
```

What is the result?

A. 97 9899 100 null null null

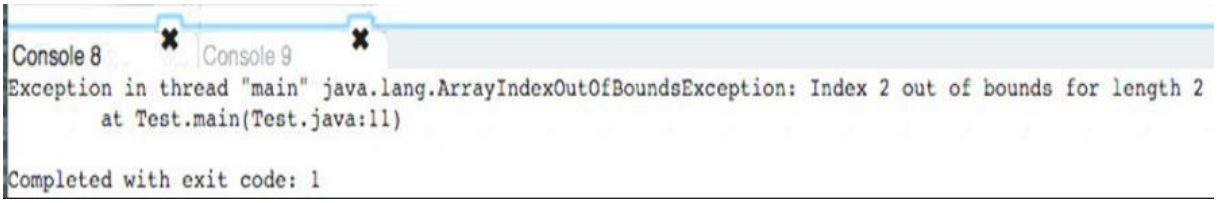
B. 97 9899 100 101 102 103

C. Compilation fails.

D. A NullPointerException is thrown at runtime.

E. An ArrayIndexOutOfBoundsException is thrown at runtime.

4-) Answer E is Correct



5-) Question 5

```
StringBuilder h4 = new StringBuilder("Hello");
h4.reverse();
System.out.println(h4);
```

What is the result?

a) Compile and print "olleh"
b) Compile and print "olleH"
c) Compile and print HERE
d) None of the answers are correct

5-)Answer

b) is correct.

The program will compile and print "olleH"

```
StringBuilder h4 = new StringBuilder("Hello");
h4.reverse();
System.out.println(h4);
```

The above outputs: "olleH"

6-) Question

Given:

```
public class A {
public static void main(String[] args){
byte[ ] [ ] ba = {{1,2,3,4}, {1,2,3} };
System.out.println(ba[1].length+" "+ba.length);
}}
```

What is the result?

a) 2 4
b) 2 7
c) 3 2
d) 3 7
e) 4 2
f) 4 7
g) Compilation fails

6-) Answer

c) is correct.

A **two-dimensional** array (2D) is an array comprised of two one-dimensional arrays.

The length of the 2D array referenced by '*ba*' is 2 because it contains 2 one-dimensional arrays.

Array indexes are *zero-based,* so ba[1] refers to ba's second array.

Arrays, or indeed variables, of primitive type *byte* can *store* int values.

===Code Step Through===

```
1: public class A {
2: public static void main(String[] args){
3: byte[ ] [ ] ba = {{1,2,3,4}, {1,2,3} };
4: System.out.println(ba[1].length+" "+ba.length);
5: }}
```

On the line 3, a two-dimensional array of type byte is created containing two one-dimensional arrays.

The first one-dimensional array {1,2,3,4} in this two-dimensional array is referred to by **ba**[0]:

ba[0] = {1,2,3,4}

The second one-dimensional array {1,2,3} in this two-dimensional array is referred to by **ba**[1] i.e.

ba[1] = {1,2,3}

Looking at the two-dimensional array again

If you wish to access '1' highlighted in bold below:

byte[ ] [ ] ba = {{**1**,2,3,4}, {1,2,3} };

You would call: **ba[0][0]**

If you wish to access 2 highlighted in bold below:

byte[ ] [ ] ba = {{1,**2**,3,4}, {1,2,3} };

You would call: **ba[0][1]**

To access any of the elements within this two-dimensional array 'ba' you would use the following 2D array 'address calls':

ba[0][0] = 1
ba[0][1] = 2
ba[0][1] = 3
ba[0][2] = 4
ba[1][0] = 1
ba[1][1] = 2
ba[1][2] = 3

=======2=======

```
1: public class A {
2: public static void main(String[] args){
3: byte[ ] [ ] ba = {{1,2,3,4}, {1,2,3} };
4: System.out.println(ba[ 1 ].length+" "+ba.length);
5: }}
```

On the line 4, ba[1].**length** retrieves the length of the *second* one-dimensional array of the 2D array 'ba'.

As the total number of elements in the second 1D array is **3**, this equates to the length of the said array and as a consequence, 3 is what gets printed.

=======3=======

```
1: public class A {
2: public static void main(String[] args){
3: byte[ ] [ ] ba = {{1,2,3,4}, {1,2,3} };
4: System.out.println(ba[1].length+" "+ba.length);
5: }}
```

On the line 4, ba.**length** retrieves the *length* of the 2D array 'ba' i.e.

ba.**length** retrieves the *total* number of 1D arrays that comprise of the 2D array 'ba'.

As the total number of elements (1D arrays) in the 2D array is 2, this equates to the length of the said array and as a consequence, 2 is what gets printed.

Final output:

3 2

7-) Question

Given:

```
import java.time.LocalDate;
import java.time. Period;
public class A {
public static void main(String[] args) {
LocalDate date1 = LocalDate.of(1977, 10, 15);
Period z = ____ .____(date1, LocalDate.now());
System.out.println("You lived for:");
System.out.println("days: "+z.getDays());
System.out.println("Months: "+z.getMonths());
System.out.println("Years: "+z.getYears());
}}
```

What code inserted above, will calculate the amount of days months and years from the date 15 October 1977?

a) Period.between
b) LocalDate.between
c) Period.getPeriod
d) DateTimes.getPeriod
e) None of the answers are correct

7-)Answer

a) is correct.

The code 'Period.between' will *calculate the amount of days months and years* from the date 15 October 1977

```
import java.time.LocalDate;
import java.time.Period;
public class A {
public static void main(String[] args) {
LocalDate date1 = LocalDate.of(1977, 10, 15);
Period z = Period.between(date1, LocalDate.now() ) ;
System.out.println("You lived for:");
System.out.println("days: "+z.getDays());
System.out.println("Months: "+z.getMonths());
System.out.println("Years: "+z.getYears());
}}
```

The above will output:

You lived for:

days: 16
Months: 1
Years: 41

Note:

java.time.Period

This class is used to create *immutable* objects that represent a period of time between two dates.

e.g.

"one year, two months, and three days from/after today"

8-)Question

You are developing a banking module. You have developed a class named ccMask that has a maskcc method. Given the code fragment:

```
class CCMask {
    public static String maskCC(String creditCard) {
        String x = "XXXX-XXXX-XXXX-";
        //line n1
    }

    public static void main(String[] args) {
        System.out.println(maskCC("1234-5678-9101-1121"));
    }
}
```

You must ensure that the maskcc method returns a string that hides all digits of the credit card number except the four last digits (and the hyphens that separate each group of four digits).

Which two code fragments should you use at line n1, independently, to achieve this requirement? (Choose two.)

☐ A)
```
StringBuilder sb = new StringBuilder(creditCard);
sb.substring(15, 19);
return x + sb;
```

☐ B)
```
return x + creditCard.substring(15, 19);
```

☐ C)
```
StringBuilder sb = new StringBuilder(x);
sb.append(creditCard, 15, 19);
return sb.toString();
```

☐ D)
```
StringBuilder sb = new StringBuilder(creditCard);
StringBuilder s = sb.insert(0, x);
return s.toString();
```

A. Option A

B. Option B

C. Option C

D. Option D

8-) Answer B and D are Correct

9-) Question

```
class Employee {
    private String name;
    private int age;
    private int salary;


    public Employee(String name, int age) {
        setName(name);
        setAge(age);
        setSalary(2000);
    }


    public Employee(String name, int age, int salary) {
        this(name, age);
        setSalary(salary);
    }


    //getter and setter methods for attributes go here

    public void printDetails() {
        System.out.println(name + " : " + age + " : " + salary);
    }
}
```

Test.java:

```
class Test {
    public static void main(String[] args) {
        Employee e1 = new Employee();
        Employee e2 = new Employee("Jack", 50);
        Employee e3 = new Employee("Chloe", 40, 5000);

        e1.printDetails();
        e2.printDetails();
        e3.printDetails();
    }
}
```

Which is the result?

A Compilation fails in the Employee class.

B

```
null : 0: 0
Jack : 50 : 0
Chloe : 40 : 5000
```

C

```
null : 0 : 0
Jack : 50 : 2000
Chloe : 40 : 5000
```

D Compilation fails in the Test class.

E Both the Employee class and the Test class fail to compile.

A. Option A

B. Option B

C. Option C

D. Option D

E. Option E

9-) Answer D is Correct

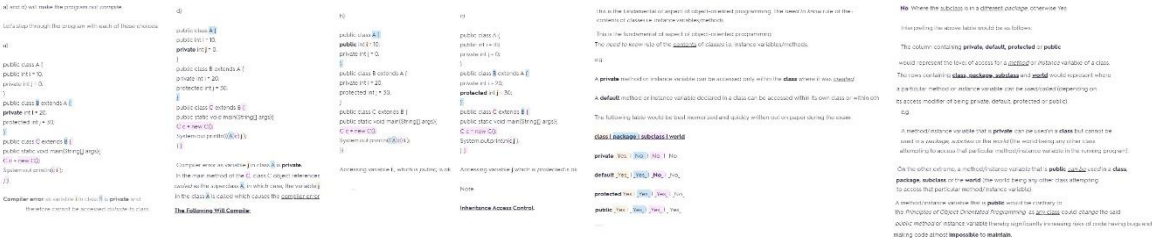
10-) Question

```
public class A {
public int i = 10;
private int j = 0;
}
public class B extends A {
private int i = 20;
protected int j = 30;
}
public class C extends B {
public static void main(String[] args){
C c = new C();
//insert code//
}}
```

Which lines of code, inserted above will make the program to not compile? (Choose all that apply)

a) System.out.println(c.i);
b) System.out.println(((A)c).i);
c) System.out.println(c.j);
d) System.out.println(((A)c).j);

10-)Answer A and D are Correct



11-)Question

```
class A {
int k =1;
}
class B extends A {
int k =2;
}
class C extends B{
int k =3;
public static void main(String[] args) {
C c = new C();
System.out.println(((A)c).k);
}}
```

What is the result? (Choose all that apply)

a) compiler error
b) runtime error
c) compile and print 1
d) compile and print 2
e) compile and print 3
f) none of the above

11-) Answer

c) is correct.

The program will compile and print : 1

```
1: class A {
2: int k = 1;
3: }
4: class B extends A {
5: int k =2;
6: }
7: class C extends B{
8: int k =3;
9: public static void main(String[] args) {
10: C c = new C();
11: System.out.println(((A)c).k);
12: }}
```

On the line 11, the c reference, which is *casted* as A,
will retrieve the the variable **k** in class A on the line 2 and not the
variable the sub class C.

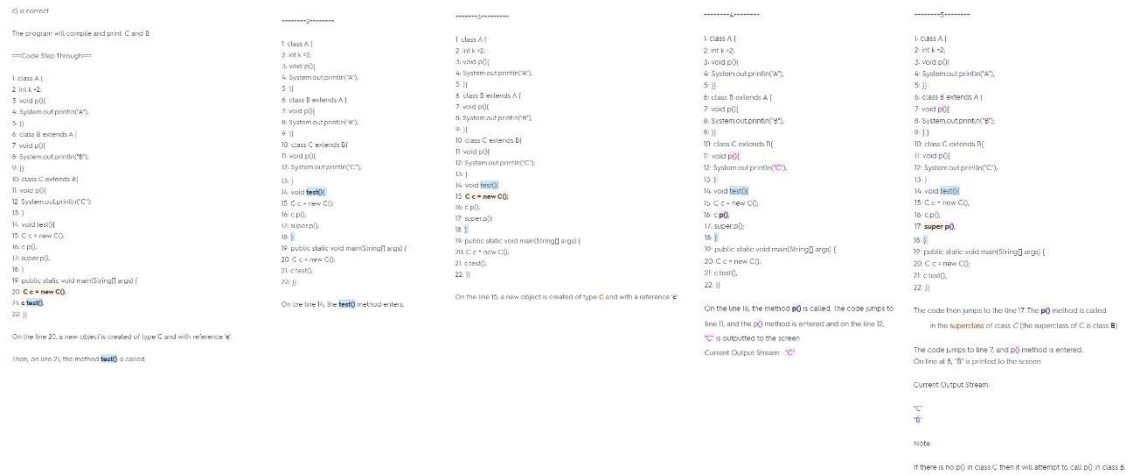
12-)Question

```
class A {
int k =2;
void p(){
System.out.println("A");
}}
class B extends A {
void p(){
System.out.println("B");
}}
class C extends B{
void p(){
System.out.println("C");
}
void test(){
C c = new C();
c.p();
super.p();
}
public static void main(String[] args) {
C c = new C();
c.test();
}}
```

What is the result? (Choose all that apply)

a) compiler error
b) runtime error
c) compile and print C and B
d) none of the above

12-)Answer, C is Correct



c) is correct

The program will compile and print. C and B

===Code Step Through===

```
1: class A {
2: int k =2;
3: void p(){
4: System.out.println("A");
5: }}
6: class B extends A {
7: void p(){
8: System.out.println("B");
9: }}
10: class C extends B{
11: void p(){
12: System.out.println("C");
13: }
14: void test(){
15: C c = new C();
16: c.p();
17: super.p();
18: }
19: public static void main(String[] args) {
20: C c = new C();
21: c.test();
22: }}
```

On the line 20, a new object is created of type C and with reference 'c'.

Then, on line 21, the method **test()** is called.

---------2---------

```
1: class A {
2: int k =2;
3: void p(){
4: System.out.println("A");
5: }}
6: class B extends A {
7: void p(){
8: System.out.println("B");
9: }}
10: class C extends B{
11: void p(){
12: System.out.println("C");
13: }
14: void test(){
15: C c = new C();
16: c.p();
17: super.p();
18: }
19: public static void main(String[] args) {
20: C c = new C();
21: c.test();
22: }}
```

On the line 14, the **test()** method enters.

---------3---------

```
1: class A {
2: int k =2;
3: void p(){
4: System.out.println("A");
5: }}
6: class B extends A {
7: void p(){
8: System.out.println("B");
9: }}
10: class C extends B{
11: void p(){
12: System.out.println("C");
13: }
14: void test(){
15: C c = new C();
16: c.p();
17: super.p();
18: }
19: public static void main(String[] args) {
20: C c = new C();
21: c.test();
22: }}
```

On the line 15, a new object is created of type C and with a reference 'c'.

---------4---------

```
1: class A {
2: int k =2;
3: void p(){
4: System.out.println("A");
5: }}
6: class B extends A {
7: void p(){
8: System.out.println("B");
9: }}
10: class C extends B{
11: void p(){
12: System.out.println("C");
13: }
14: void test(){
15: C c = new C();
16: c.p();
17: super.p();
18: }
19: public static void main(String[] args) {
20: C c = new C();
21: c.test();
22: }}
```

On the line 16, the method **p()** is called. The code jumps to line 11, and the p() method is entered and on the line 12, "C" is outputted to the screen.

Current Output Stream: "C"

---------5---------

```
1: class A {
2: int k =2;
3: void p(){
4: System.out.println("A");
5: }}
6: class B extends A {
7: void p(){
8: System.out.println("B");
9: }}
10: class C extends B{
11: void p(){
12: System.out.println("C");
13: }
14: void test(){
15: C c = new C();
16: c.p();
17: super.p();
18: }
19: public static void main(String[] args) {
20: C c = new C();
21: c.test();
22: }}
```

The code then jumps to the line 17. The **p()** method is called in the superclass of class C (the superclass of C is class **B**).

The code jumps to line 7, and p() method is entered.
On line at 8, "B" is printed to the screen

Current Output Stream:

"C"
"B"

Note:

If there is no p() in class C then it will attempt to call p() in class B.

13-)Question

```
class A {
public String mA() {
return "mA()";
}}
class B extends A{
public String mA() {
return "mB()";
}
public static void main(String[] args) {
A a = new B();
System.out.println(a.mA());
}}
```

What is the result? (Choose all that apply)

a) This is an example of method overloading
b) This is an example of method overriding
c) compile and print mB()
d) none of the above

13-) Answer

b) and c) are correct.

This is an example of method overriding (which occurs at runtime). The output of this program would be: 'mB()'

An overriding method has the exact same:

- Method name
- Parameter list
- Return type

```
class A {
public String mA() {
return "mA()";
} }
class B extends A{
public String mA() {
return "mB()";
}
public static void main(String[] args) {
A a = new B();
System.out.println(a.mA());
}}
```

14-) Question

Given:

```
3. public class B extends A {
4. public static String sing() { return "fa"; }
5. public static void main(String[] args) {
6. B t = new B();
7. A s = new B();
8. System.out.println(t.sing () + " " + s.sing());
5. }
10. }
11. class A {
12. public static String sing() { return "la"; } }
```

What is the result?

A. fa fa
B. fa la
C. la la
D. Compilation fails
E. An exception is thrown at runtime

14-) Answer

B is correct.

The program will print: *fa la*

The code is correct, but *polymorphism* doesn't apply to static methods.

===Code Step Through===

```
3. public class B extends A {
4. public static String sing() { return "fa"; }
5. public static void main(String[] args) {
6. B t = new B();
7. A s = new B();
8. System.out.println(t.sing () + " " + s.sing());
5. }
10. }
11. class A {
12. public static String sing() { return "la"; } }
```

On the line 6, new object is created of type B and which has a reference '**t**'.
On the line 7, a new object is created of type B and which has a reference '**s**' (of type **A**)

========2========

```
3. public class B extends A {
4. public static String sing() { return "fa"; }
5. public static void main(String[] args) {
6. B t = new B();
7. A s = new B();
8. System.out.println(t.sing() + " " + s.sing());
5. }
10. }
11. class A {
12. public static String sing() { return "la"; } }
```

On the line 8, t.**sing()** is called which calls the *static* method of class B on line 4.
On line 4, string "fa" is return to the main() method and gets printed to the screen.

========3========

```
3. public class B extends A {
4. public static String sing() { return "fa"; }
5. public static void main(String[] args) {
6. B t = new B();
7. A s = new B();
8. System.out.println(t.sing() + " " + s.sing());
5. }
10. }
11. class A {
12. public static String sing() { return "la"; } }
```

On the line 8, s.**sing()** is called which calls the *static* method of class A.
On line 4, string "la" is return to the main() method and gets printed to the screen.

Final Ouptut:

The program compiles and prints: "*fa la*"

Polymorphism:

Polymorphism is the ability of a class instance to *behave* as if it were an *instance* of another class in its inheritance tree, most often one of its ancestor classes.

Consider the following code which has a *superclass* name Animal and which has two *subclasses* Pig and Chicken.

In the main method, an array of Animals stores a number of animal objects names.

When the code is run, the compiler knows ("*polymorphically*") that inside *animalFarm*[0], a Pig object is stored.

The compiler calls the method *getNoise()* from the Pig class and prints "Oink" to the screen.

The same is for *animalFarm*[1].

This is an example of Polymorphism.

```
class Animal {
public String getNoise() {
return "Noise";
}
}
public class Pig extends Animal {
public String getNoise(){ return "Oink"; }
}
}
class Chicken extends Animal {
public String getNoise() { return "Cluck Cluck";
}
public static void main(String[] args) {
Animal [] animalFarm = {new Pig(),new Chicken()};
System.out.println(animalFarm[0].getNoise());
System.out.println(animalFarm[1].getNoise());
}
```

15-)Question

22. Given the code fragment:

```
public static void main(String[] args) {
    int ans;
    try {
        int num = 10;
        int div = 0;
        ans = num / div;
    } catch (ArithmeticException ae) {
        ans = 0;                                    // line n1
    } catch (Exception e) {
        System.out.println("Invalid calculation");
    }
    System.out.println("Answer = " + ans);    // line n2
}
```

What is the result?

A. Answer = 0

B. Invalid calculation

C. Compilation fails only at line n1.

D. Compilation fails only at line n2.

E. Compilation fails at line n1 and line2.

15-)Answer,

Answer: A

Explanation::

```
1
2  public class Test {
3    public static void main(String[] args) {
4      int ans;
5      try {
6        int num = 10;
7        int div = 0;
8        ans = num / div;
9      } catch (ArithmeticException ae) {
10       ans = 0;
11     } catch (Exception e) {
12       System.out.println("Invalid calculation");
   variable ans might not have been initialized
14     System.out.println("Answer = " + ans);  //line n2
15   }
16 }
```

16-)Question

```
public class Locomotive {
Locomotive() { main("hi");
}
public static void main(String[] args) {
System.out.print("2 ");
}
public static void main(String args) {
System.out.print("3 " + args);
}}
```

What is the result? (Choose all that apply.)

A. 2 will be included in the output
B. 3 will be included in the output
C. hi will be included in the output
D. Compilation fails
E. An exception is thrown at runtime

16-) Answer

A is correct.

The number **2** will be included in the output.
This question tests your knowledge of **overloaded methods**.

```
public class Locomotive {
Locomotive() { main("hi");
}
public static void main(String[] args) {
System.out.print("2");
}
public static void main(String args) {
System.out.print("3 " + args);
} }
```

It's legal to overload main() since **no instances** of Locomotive are created, the constructor does not run and the overloaded version of main () does not run.

17-)Question

```
public class A {
{
System.out.println("Non static block A");
}
static {
System.out.println("static block A");
}
}
public class B extends A{
static {
System.out.println("static block B");
}
{
System.out.println("Non static block B");
}
public static void main(String[] args){
B b = new B();
}}
```

What is the output of the above code?

What is the result? (Choose all that apply)

a)

static block A
static block B
Non static block A
Non static block B

b)

Non static block A
Non static block B
static block A
static block B

c)

static block A
Non static block A
Non static block B
static block B

d) none of the above

17-)Answer A is Correct

a) is correct.

The program will output:

static block A
static block B
Non static block A
Non static block B

This question tests your knowledge of static and nonstatic initialization blocks.

Static Initialization Blocks

The Java language includes *static initialization* blocks. Think of it as a *constructor* for *static variables*.

Static initialization blocks gives you a chance to *initialize* the *variables* before anyone uses them.

static{} block is executed when the class is loaded.

Within the static parenthesis **{}** 'block', there would be the static code (usually static variables) that would be the same for all instances.

Note: Starting at the top of the class hierarchy, all static blocks get executed first in sequential order down the class hierarchy.

This is followed by the same procedure for nonstatic blocks.

Nonstatic Initialization Blocks

Similar to static blocks, Nonstatic blocks are *simply parenthesis* **{}** which would also contain code:

{ }

The non-static initialization block would contain code (usually *instance variables*) that would need to be initialized first when an object is instantiated.

non-static blocks of code can access the *instance variables* of an object. This is not the same for static blocks of code.

static blocks are stored in *class memory*. This means that the code within the static block can never be changed when the application executes.

This saves memory and makes the code more efficient.

Nonstatic blocks are stored in a heap memory.

Let's step through the code:

Code Step Through

```
1: public class A {
2: {
3: System.out.println("Non static block A");
4: }
5: static {
6: System.out.println("static block A");
7: }
8: }
9: public class B extends A{
10: static {
11: System.out.println("static block B");
12: }
13: {
14: System.out.println("Non static block B");
15: }
16: public static void main(String[] args){
17: B b = new B();
18: }}
```

On the line 17, a new object of type B is instantiated.

========2========

```
1: public class A {
2: {
3: System.out.println("Non static block A");
4: }
5: static {
6: System.out.println("static block A");
7: }
8: }
9: public class B extends A{
10: static {
11: System.out.println("static block B");
12: }
13: {
14: System.out.println("Non static block B");
15: }
16: public static void main(String[] args){
17: B b = new B();
18: }}
```

On the line 17, the B constructor will first automatically call the superclass A constructor using implicitly *super()*.
Class A does not have an *explicit constructor* so the *default constructor* is provided implicitly.

========3========

```
1: public class A {
2: {
3: System.out.println("Non static block A");
4: }
5: static {
6: System.out.println("static block A");
7: }
8: }
9: public class B extends A{
10: static {
11: System.out.println("static block B");
12: }
13: {
14: System.out.println("Non static block B");
15: }
16: public static void main(String[] args){
17: B b = new B();
18: }}
```

On the line 5, inside the superclass constructor A(){} calls **static block** first which outputs to the screen:
static block A

========4========

```
1: public class A {
2: {
3: System.out.println("Non static block A");
4: }
5: static {
6: System.out.println("static block A");
7: }
8: }
9: public class B extends A {
10: static {
11: System.out.println("static block B");
12: }
13: {
14: System.out.println("Non static block B");
15: }
16: public static void main(String[] args){
17: B b = new B();
18: }}
```

The the next static block down the code is called.

This outputs to the screen:

static block A
static block B

========5========

```
1: public class A {
2: {
3: System.out.println("Non static block A");
4: }
5: static {
6: System.out.println("static block A");
7: }
8: }
9: public class B extends A{
10: static {
11: System.out.println("static block B");
12: }
13: {
14: System.out.println("Non static block B");
15: }
16: public static void main(String[] args){
17: B b = new B();
18: }}
```

After all *static blocks* have been executed, the *nonstatic blocks* get executed next from the *top down* in the class hierarchy.
This outputs next:

static block A
static block B
Non static block A

========6========

```
1: public class A {
2: {
3: System.out.println("Non static block A");
4: }
5: static {
6: System.out.println("static block A");
7: }
8: }
9: public class B extends A {
10: static {
11: System.out.println("static block B");
12: }
13: {
14: System.out.println("Non static block B");
15: }
16: public static void main(String[] args){
17: B b = new B();
18: } }
```

The next nonstatic block in the class hierarchy is executed which is located between the lines 13 and 15.

After the execution of the above code, the following is a final snapshot of what is outputted to the screen:

static block A
static block B
Non static block A
Non static block B

18-) Question

```
interface RemoteControl1
{ int curChannel=1;
public void turnOnTV();
}
interface RemoteControl2
{ int curChannel=2;
public void turnOnTV();
}

class RemoteControler implements RemoteControl1, RemoteControl2 {
//insert code//
}
```

What code inserted above will cause it to compile and run fine? (Choose all that apply)

a)

```
public void turnOnTV(){}
public void turnOnTV(){}
```

b)

```
public void turnOnTV();
```

c)

```
public void turnOnTV<RemoteControl1>(){}
public void turnOnTV()<RemoteControl2>{}
```

d) none of the above

18-) Answer

b) is correct.

The program was compiled with the following method inserted:

public void turnOnTV();

Let's look at the full program below:

```
1: interface RemoteControl1
2: { int curChannel=1;
3: public void turnOnTV();
4: }
5: interface RemoteControl2
6: { int curChannel=2;
7: public void turnOnTV();
8: }
9: class RemoteControler implements RemoteControl1, RemoteControl2 {
10: public void turnOnTV(){}
11: }
```

On the line 10, the ***turnOnTV()*** method is required to be implemented for two interfaces which is *perfectly fine*.

The Following Will Not Compile:

a)

```
public void turnOnTV(){}
public void turnOnTV(){}
```

The above will give a **compiler error** as you cannot implement two methods with the *same name* and parameter signature

c)

```
public void turnOnTV<RemoteControl1>(){}
public void turnOnTV()<RemoteControl2>{}
```

The above will give **syntax errors**

19-) Question

Which of the following are false? (Choose all that apply)

a) A subclass can call the superclass constructor
b) A subclass cannot call it superclass constructor
c) A constructor can be private
d) A constructor cannot be final, static or abstract
e) A constructor can access static and nonstatic members of the class.

19-) Answer

a) is not true.

A **subclass** cannot call the *superclass* constructor.

The following are true:

b) A *subclass* cannot call it *superclass* constructor
c) A constructor can be *private*
d) A constructor cannot be *final, static* or *abstract*
e) A constructor can access static and nonstatic *members* of the class.

20-) Question

Given this code for the classes MyException and Test:

```
public class MyException extends RuntimeException {}

public class Test {
    public static void main(String[] args) {
       try {
           method1();
       }
       catch (MyException ne) {
           System.out.print("A");
       }
    }
    public static void method1() { // line n1
        try {
           throw 3 > 10 ? new MyException() : new IOException();
       }
        catch(IOException ie) {
            System.out.println("I");
        }
       catch (Exception re) {
           System.out.print("B");
       }
    }
}
```

What is the result?

A. A

B. AB

C. A compile time error occurs at line n1.

D. B

E. I

20-) Answer C is Correct

21-) Question

```
int[] intArr = {15, 30, 45, 60, 75};
intArr[2] = intArr[4];
intArr[4] = 90;
```

What are the values of each element in intArr after this code has executed?

A. 15, 60, 45, 90, 75

B. 15, 90, 45, 90, 75

C. 15, 30, 75, 60, 90

D. 15, 30, 90, 60, 90

E. 15, 4, 45, 60, 90

21-) Answer C is Correct

22-) Question

```
public class A {
public static void main(String[] args) {
StringBuilder sb = new StringBuilder();
String s = new String();
for(int i = 0; i < 1000; i++) {
s = " " + i ;
sb.append(s);
}
}}
```

If the garbage collector does NOT run while this code is executing, approximately how many objects will exist in memory when the loop is complete?

a) Less than 10
b) About 1000
c) About 2000
d) About 3000
e) About 4000

22-) Answer B is Correct

b) is correct.

➤ **StringBuilders** are *mutable* (changeable), so all of the append() invocations are acting on the same StringBuilder *object* over and over.

➤ **Strings**, however, are *immutable* (not changeable), so every String concatenation operation results in a new String *object*.

➤ Also, the string " " is created once and reused in every loop iteration.

===Code Step Through===

```
1: public class A {
2: public static void main(String[] args) {
3: StringBuilder sb = new StringBuilder();
4: String s = new String();
5: for(int i = 0; i < 1000; i++) {
6: s = " " + i ;
7: sb.append(s);
8: }
9: }}
```

On the line 3, a StringBuilder object with reference **sb** is created and with a *unique arbitrary address* assigned to it e.g. *@9982*

A pseudocode description of this object in memory as outlined below:

```
sb → Object StringBuilder(Memory Address @9982)
{
""
}
```

=======2=======

```
1: public class A {
2: public static void main(String[] args) {
3: StringBuilder sb = new StringBuilder();
4: String s = new String();
5: for(int i = 0; i < 1000; i++) {
6: s = " " + i ;
7: sb.append(s);
8: }
9: }}
```

On the line 4, a new string *object* is created with reference **s** in the **String Constant Pool**. It would have an *arbitrary address* in memory also.

A pseudocode description of the current objects in memory is outlined below:

```
sb → Object StringBuilder(Memory Address @9982)
{
""
}

s → Object String(Memory Address @3324)
{
""
}
```

=======3=======

```
1: public class A {
2: public static void main(String[] args) {
3: StringBuilder sb = new StringBuilder();
4: String s = new String();
5: for(int i = 0; i < 1000; i++) {
6: s = " " + i ;
7: sb.append(s);
8: }
9: }}
```

On the line 5, the for loop block is executed and the *int variable* **i** is initialised to **0**.

At this point in the code, a pseudocode description of the current objects and variables in memory would look like this.

```
i = 0

sb → Object StringBuilder(Memory Address @9982)
{
""
}

s → Object String(Memory Address @3324)
{
""
}
```

=======4=======

```
1: public class A {
2: public static void main(String[] args) {
3: StringBuilder sb = new StringBuilder();
4: String s = new String();
5: for(int i = 0; i < 1000; i++) {
6: s = " " + i ;
7: sb.append(s);
8: }
9: }}
```

On the line 5, a *comparison* is made between the *current value* stored in the **variable** **i** and the number **1000**.

If the current value stored in the variable **i** is less than **1000**, the *postfix* expression **i++** is executed and the for block is entered.

Looking at the current variables and objects in memory above, we can see that the *current* value stored in variable **i** is **0** and as 0 is less than **1000**:

- The *postfix* expression i++ is *executed*.
- The *for block* is entered.

Note: The postfix expression i++ will not **increment** (by 1) *until* the control flow of the code gets to *line 8* (i.e. *the end of the for bock current iteration*)

=======5=======

```
1: public class A {
2: public static void main(String[] args) {
3: StringBuilder sb = new StringBuilder();
4: String s = new String();
5: for(int i = 0; i < 1000; i++) {
6: s = " " + i ;
7: sb.append(s);
8: }
9: }}
```

On the line 6, the current value of **i** is *appended* to an empty String " " which results in a new String *object* being created (" 0") with its own unique arbitrary memory address e.g. *&5643*.

The String reference **'s'** no longer references the String object would memory address *@3324* and instead, is reassigned to the new String object with memory address *&5643*.

Remember, Strings are **immutable**, therefore they cannot change that is why a new string object is created above with memory address *&5643*. This is important to understand for this line of the code.

The following outlines a pseudocode description of the current objects and variables in memory when line 6 is executed:

```
i = 0

sb → Object StringBuilder(Memory Address @9982)
{
""
}
Object String(Memory Address @3324)
{
""
}
s → Object String(Memory Address &5643)
{
" 0"
}
```

=======6=======

```
1: public class A {
2: public static void main(String[] args) {
3: StringBuilder sb = new StringBuilder();
4: String s = new String();
5: for(int i = 0; i < 1000; i++){
6: s = " " + i ;
7: sb.append(s);
8: }
9: }}
```

On the line 7, the contents of the String object referenced by **'s'** (with memory address *&5643*) is appended to the StringBuilder object referenced by **'sb'** (with memory address *@9982*).

As StringBuilder is *mutable*, a new object is not created.

The following pseudocode description outlines the current objects and variables in memory at this particular point of the code:

```
i = 0

sb → Object StringBuilder(Memory Address @9982)
{
" 0"
}
Object String(Memory Address @3324)
{
""
}
s → Object String(Memory Address &5643)
{
" 0"
}
```

=======7=======

```
1: public class A {
2: public static void main(String[] args) {
3: StringBuilder sb = new StringBuilder();
4: String s = new String();
5: for(int i = 0; i < 1000; i++) {
6: s = " " + i ;
7: sb.append(s);
8: }
9: }}
```

On the line 8, the **i++** gets **incremented** by **1** and the flow control jumps back up to line 5.

The following pseudocode description outlines the current objects and variables in memory at this particular point of the code:

```
i = 1

sb → Object StringBuilder(Memory Address @9982)
{
" 0"
}
Object String(Memory Address @3324)
{
""
}
s → Object String(Memory Address &5643)
{
" 0"
}
```

=======8=======

```
1: public class A {
2: public static void main(String[] args) {
3: StringBuilder sb = new StringBuilder();
4: String s = new String();
5: for(int i = 0; i < 1000; i++){
6: s = " " + i ;
7: sb.append(s);
8: }
9: }}
```

On the line 5, the expression **i < 1000** is *executed*.

Looking at the current variables and objects in memory above, we can see that the *current value* stored in variable **i** is **1** and as 1 is less than **1000**:

- The *postfix* expression i++ is executed.
- The *for* block is entered.

Note: The postfix expression i++ will not *increment* (by 1) until the control flow of the code gets to line 8 (i.e. the end of the for bock current iteration)

=======9=======

```
1: public class A {
2: public static void main(String[] args) {
3: StringBuilder sb = new StringBuilder();
4: String s = new String();
5: for(int i = 0; i < 1000; i++) {
6: s = " " + i ;
7: sb.append(s);
8: }
9: }}
```

On the line 6, the *current value* of **i** is **appended** to an empty String " " which results in a new String *object* being created (" 1") with its own unique arbitrary memory address e.g. **&2232**.

The following outlines a pseudocode description of the current objects and variables in memory when line 6 is executed.

```
sb → Object StringBuilder(Memory Address @9982)
{
" 0"
}
Object String(Memory Address @3324)
{
""
}
Object String(Memory Address &5643)
{
" 0"
}
s → Object String(Memory Address &2232)
{
" 1"
}
```

The String reference **'s'** no longer references the String object with memory address &5643 and instead, is reassigned to the new String object with memory address **&2232**.

As you can see, the for loop will continue to execute until the expression **i < 1000** is no longer true.

Therefore, when "**i**" finally increments to 1000, there will be approximately **1000** new *String* objects *created* in memory.

Note:

To make Java more *memory efficient*, the Java Virtual Machine sets aside a special area of memory called the **String Constant Pool**.

When the compiler encounters a String literal (e.g. "Hello"), it **checks** the pool to see if the same *identical* String literal already **exists**.

If a match is found, the **reference** to the new literal is *directed* to the existing String, and no new String literal object is created.

This is why string objects are **immutable** i.e. they cannot be *changed* and are *final* constants.

The key concept for you to understand is that once a String object is created, it can never be *changed*.

23-) Question

Which of the following are true?

a) StringBuilder class methods are not synchronized
b) StringBuffer class methods are synchronized
c) StringBuilder class methods are synchronized
d) StringBuffer class methods are not synchronized

23-) Answer

b) is correct.

StringBuffer class methods are synchronized.

The **Java synchronized** keyword is an essential tool in concurrent programming in **Java**.

Its overall purpose is to only allow one thread at a time into a particular section of code thus allowing us to protect, for example, *variables or data* from being *corrupted* by *simultaneous modifications* from different threads.

24-) Question

Consider the following code:

```
import java.util.AbstractList;
import java.util.ArrayList;
public class A {
public static void main(String args[])
{
AbstractList<String> list = new ArrayList<String>();
list.add('Hi');
list.add(5);
System.out.println(list);
}}
```

What is the result?

a) Compiler error
b) Runtime error
c) Compile and print:

```
Hi
5
```

d) None of the answers are correct.

24-) Answer

a) is correct.

As stated, the program will give a **compiler error**.

===Code Step Through===

```
1: import java.util.AbstractList;
2: import java.util.ArrayList;
3: public class A {
4: public static void main(String args[])
5: {
6: AbstractList<String> list = new ArrayList<String>();
7: list.add('Hi');
8: list.add(5);
9: System.out.println(list);
10: }
11: }
```

On the line 7, a *primitive* char '**Hi**' is stored in an ArrayList.

Likewise, on the line 8, the *primitive* int variable **5** is attempted to be stored in the ArrayList.

You cannot store *primitives* in an *ArrayList*.

Therefore, a **compiler error** occurs on the line 7 and also 8

The above shows that *ArrayList* is *type safe* as it is expecting an object on the line 7 and 8.

25-) Question

Given that addIt() returns an int, which are valid Predicate lambdas?

a) x, y -> 7 < 5
b) x -> { return addIt(2, 1) >5; }
c) x -> return addIt(2, 1) > 5;
d) x -> { int y = 5; int z = 7; addIt(y, z) > 8; }
e) (MyClass x) -> 7 > 13
f) x -> { int y = 5; int z = 7; return addIt(y, z) >8; }
g) (MyClass x) -> 5+4

25-) Answer

b), e) and f) are correct.

b) x -> **{ return** addIt(2, 1) >5; **}**

According to the rules of lambda expressions, the keyword '**return**' must be *wrapped* in brackets '**{}**'.

The method addIt(2, 1) passes in two ints into its parameters, it *adds these numbers and returns 3.*

The number 3 is then compared to see if it is greater than the number 5.

This expression resolves to a boolean false.

e) **(** MyClass x **)** -> 7 > 13

According to the rules of lambda expressions, the lambda single parameter *can be* cast using parenthesis.

f) x -> **{** int y = 5; int z = 7; return **addIt**(y, z) >8; **}**

According to the *rules of lambda expressions,* variables can be declared *inside* and *outside* the **block**.

The method **addIt(**5, 7**)** passes in two ints into its parameters, it adds these numbers and returns 12.

The number 12 is then compared to see if it is greater than the number 8.

This expression resolves to a boolean true

Incorrect Answers:

a) **x, y** -> 7 < 5

Incorrect syntax: **x, y** -> 7 < 5

The java lambda uses a Single Parameter. In the above, it has *two parameters* which gives a **compiler error**.

c) x -> **return** addIt(2, 1) > 5;

The '**return**' keyword must be *encapsulated* in a code block using '**{ }**'. Therefore the above code give a **compiler error**.

d) x -> **{** int y = 5; int z = 7; addIt(y, z) > 8; **}**

A '**return**' keyword must be *encapsulated* in a code block using '**{}**'. As this keyword is missing in the above expression, it gives **compiler error**.

g) (MyClass x) ->**5+4**

According to the rules of lambda expressions, the body of the lambda expression must return a **boolean**.

In the expression above, it does not resolve to a *boolean* but simply *adds* two numbers together. This gives a **compiler error**.

26-) Question

Given:

```
1: import java.util.function.Predicate;
2: class A {
3: public static void main(String[] args) {
4: A a1 = new A();
5: a1.go(INSERT CODE HERE);
6: }
7: void go(Predicate<A> a){
8: A a2 = new A();
9: System.out.println(a.test(a2));
10: }
11: static int adder(int x, int y){return x+y;}
12: }
```

Which code, inserted on line 5 above, will compile and run fine?

a) p->7<4
b) p<->5<6
c) u->{return adder(2, 1) >2;}
d) None of the answers are correct

26-) Answer

c) is correct.

Looking at the highlighted aspects of the correct answer:

u-> **{return** adder(2, 1) >2; **}**

The syntax rules for Java Lambda states that the **body** of a Lambda expression can be a code *block* surrounded by curly brackets **{}** ending with a *return statement*.

In the above Lambda expression, the body

'**{return** adder(2, 1) >2;**}**' fulfils this syntax rule. Note that the adder() method call returns a boolean which is then compared with the number 2.

Incorrect Answers:

a) *p->7<4*

The syntax rules for Java Lambda states that the **body** of a Lambda expression must return a *boolean*.

In the above Lambda expression, the body '**7<4**' resolves to a boolean which is 'false'.

b) *p<->5<6*

Looking at the highlighted aspects of the program:

The Java Lambda **arrow** *token* is written as **->**.

The symbol '<->' is not recognised in Java and gives a **compiler error**.

Note:

Introduction to Simple Lambdas

In this section we're going to outline a basic introduction to Lambdas. Firstly

The benefits of Lambdas:

- Easier to Read Code
- More Concise Code
- Faster Running Code (More Efficient).

The core aspect of lambdas is that you can pass *complex expressions* (calculations that result in a boolean) as an *argument* into a *method*. This is as *opposed* to the traditional manner of *passing variables* into a *method* i.e. *primitive types* or *object type* variables.

So instead of having a Java class that contains many *different methods* that perform *separate and distinct responsibilities* (functions), you could have one *method* that can perform **ALL** of those *separate and distinct responsibilities*.

The *benefit* would be in the area of *memory efficiency* so that, instead of having *multiple* different *methods* allocated memory, you could have *one* method allocated *memory*.

Also, the class containing the lambda expression would make the class *more cohesive* in performing a particular task.

Predicate Interface

For the OCA exam, you will only need to know about the Predicate *interface* and its *method* **test()**.

Predicate, in general meaning, is a statement about something that is either true or false. In programming, predicates represent single *argument functions* that *return* a boolean value.

The Predicate interface is located here:

java.util.function.Predicate

The **test()** method *evaluates* a predicate on the given argument '**t**'. Its signature is as follows:

boolean test(T **t**)

Where **parameter t** represents the *input* argument which will result in the boolean value true being returned if the input argument matches the predicate, otherwise false.

There is a simple example of the use of the Predicate interface and its method test():

```
import java.util.function.Predicate;
public class Main {
public static void main(String[] args) {
Predicate<String> i = (s)-> s.length() > 5;
System.out.println( i.test("java2s.com "));
}
```

As you can see in the above code, the Java Lambda expression is: **s.length() > 5** and this is 'tested' utilising the test() method with a string literal "**java2s.com** ".

As you can see in the above code, the Java Lambda expression is: **s.length() > 5** and this is 'tested' utilising the test() method with a string literal "**java2s.com** ".

As "**java2s.com** " has a length greater than 5, '*true*' is printed to the screen.

In summary, for the OCA exam, you will be tested on the very *basics* of Java lambdas e.g. on the *syntax* of Java Lambda expressions, the use of the Predicate interface and also its *test()* method.

The Lambda *questions* for the OCA exam may also be *mixed* with other aspects that will be tested e.g. *objects* and *data structures* like a array lists.

Syntax Rules for Predicate Lambda

➤ The predicate **parameter** can be just a *variable name* or it can be the *type* followed by the variable name, all in parethesis.

e.g.

```
s.go((x) -> adder(5,1) < 7);
s.go(u -> adder(6,2) < 9);
```

➤ The body must return a **boolean**.

e.g.

```
s.go((x) -> 3 < 7);
s.go(u -> adder(6,2) < 9);
```

➤The body can be a **single** *expression* which cannot have a *return statement*.

e.g.

```
s.go((x) -> 3 < 7);
```

➤ The *body* can be a code block surrounded by *curly brackets* containing one or more valid statements, each ending with a semicolon and the block must end with a return statement.

e.g.

```
s.go(u -> { int x = 1; return adder(x,2) < 9 } );
```

27-) Question

Given:

```
import java.time. ____ ;
import java.time.ZoneId;
public class A {
public static void main(String[] args) {
ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Europe/Dublin");
____ now1 = ____ .now(zone1);
____ now2 = ____ .now(zone2);
System.out.println("Berlin Time: "+now1);
System.out.println("Dublin Time: "+now2);
}}
```

What class, inserted in the above empty spaces, will return the current local time for Berlin and Dublin?

a) DateTime
b) LocalDateTimes
c) LocalTime
d) DateTimes
e) LocalDate

27-) Answer

c) is correct.

The class LocalTime will return the current local time for Berlin and Dublin.

```
import java.time.LocalTime;
import java.time.ZoneId;
public class A {
public static void main(String[] args) {
ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Europe/Dublin");
LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);
System.out.println("Berlin Time: "+now1);
System.out.println("Dublin Time: "+now2);
}}
```

The above will output:

Berlin Time: 11:51:19.238
Dublin Time: 10:51:19.254

Note:

java.time.LocalTime

This class is used to create immutable objects each which represents a *specific time for certain area* (time-zone) in the world.

e.g.

Berlin Time Now: 11:28:04.865
Dublin Time Now: 10:28:04.865

28-) Question

Which of the following classes are not part of the Calendar collection?

a) LocalDateTime
b) LocalDate
c) LocalTime
d) DateTimeFormatter
e) TimeDateFormatter
f) TemporalAmount

28-) Answer

e) is correct.

There is no such class as *TimeDateFormatter*.

Calendar Collection

It would be advisable to memorize certain classes of the Calendar collection as these will pop up in the exam.

Below is a suggested abbreviation to help memorize some of the core classes of the calendar collection:

LLL-DPT

➤ java.time.***L****ocalDateTime*

This class is used to create immutable objects each of which represents a specific date and time.

➤ java.time.***L****ocalDate*

This class is used to create immutable objects each of which represents a specific date.

➤ java.time.***L****ocalTime*

This class is used to create immutable objects each of which represents a specific time.

➤ java.time.format.***D****ateTimeFormatter*

This immutable class is used by the class above to format date/time objects for output and to parse input strings and convert them to date/time objects.

➤ java.time.***P****eriod*

This class is used to create immutable objects that represent a period of time for example "one year, two months, and three days"

➤ java.time.temporal.***T****emporalAmount*

This **interface** is implemented by the Period class.

29-) Question

```
import java.time.LocalDate;
import java.time. Period;
public class A {
public static void main(String[] args) {
LocalDate date1 = LocalDate.of(1977, 10, 15);
Period z = ____ .____(date1, LocalDate.now());
System.out.println("You lived for:");
System.out.println("days: "+z.getDays());
System.out.println("Months: "+z.getMonths());
System.out.println("Years: "+z.getYears());
}}
```

What code inserted above, will calculate the amount of days months and years from the date 15 October 1977?

a) Period.between
b) LocalDate.between
c) Period.getPeriod
d) DateTimes.getPeriod
e) None of the answers are correct

29-) Answer

a) is correct.

The code 'Period.between' will *calculate the amount of days months and years* from the date 15 October 1977

```
import java.time.LocalDate;
import java.time.Period;
public class A {
public static void main(String[] args) {
LocalDate date1 = LocalDate.of(1977, 10, 15);
Period z = Period.between(date1, LocalDate.now() ) ;
System.out.println("You lived for:");
System.out.println("days: "+z.getDays());
System.out.println("Months: "+z.getMonths());
System.out.println("Years: "+z.getYears());
}}
```

The above will output:

You lived for:

days: 16
Months: 1
Years: 41

Note:

java.time.Period

This class is used to create *immutable* objects that represent a period of time between two dates.

e.g.

"one year, two months, and three days from/after today"