**Working with Arrays and Collections - 2 - Results**

Return to review

Chart

Pie chart with 4 slices.

End of interactive chart.

Attempt 2

All knowledge areas

All questions

Top of Form

Question 1: **Correct**

**Question ID: UKOCP57135**

For the given code snippet:

List<String> list = new /\*INSERT\*/();

Which of the following options, if used to replace /\*INSERT\*/, compiles successfully?

Select 2 options.

* 

**List<String>**

* 

**List<>**

* 

**ArrayList<String>**

**(Correct)**

* 

**ArrayList<>**

**(Correct)**

**Explanation**

**UKOCP57135:**

List is an interface so its instance can't be created using new keyword. List<String> and List<> will cause compilation failure.

ArrayList implements List interface, so it can be it can be used to replace /\*INSERT\*/. `List<String> list = new ArrayList<String>();` compiles successfully.

Starting with JDK 7, Java allows to not specify type while initializing the ArrayList. Type is inferred from the left side of the statement.

So `List<String> list = new ArrayList<>();` is a valid syntax starting with JDK 7.

Bottom of Form

Top of Form

Question 2: **Correct**

**Question ID: UKOCP26419**

Consider below code snippet:

1. public static void process(/\*INSERT\*/ list) {
2. list.add(100); //Line n2
3. int x = list.get(0); //Line n3
4. System.out.println(list.size() + ":" + x);
5. }

Which of the following options, if used to replace /\*INSERT\*/, compiles successfully?

* 

**java.util.List**

* 

**java.util.List<Integer>**

**(Correct)**

* 

**java.util.List<Object>**

* 

**java.util.List<int>**

**Explanation**

**UKOCP26419:**

Generic type can only be reference type and not primitive type, hence java.util.List<int> is not a valid syntax.

If you use raw type java.util.List or java.util.List<Object> then Line n3 will cause compilation failure as list.get(0) will return Object type. Object type cannot be converted to primitive type int.

java.util.List<Integer> is the only correct option left.

Bottom of Form

Top of Form

Question 3: **Correct**

**Question ID: UKOCP36682**

Consider below code snippet:

1. import java.util.ArrayList;
2. import java.util.List;
4. public class Test {
5. List list1 = new ArrayList<String>(); //Line n5
6. List<String> list2 = new ArrayList(); //Line n6
7. List<> list3 = new ArrayList<String>(); //Line n7
8. List<String> list4 = new ArrayList<String>(); //Line n8
9. List<String> list5 = new ArrayList<>(); //Line n9
10. }

Which of the following statements compile without any warning?

Choose 2 options.

* 

**Line n5**

* 

**Line n6**

* 

**Line n7**

* 

**Line n8**

**(Correct)**

* 

**Line n9**

**(Correct)**

**Explanation**

**UKOCP36682:**

Line n8's syntax was added in JDK 5 and it compiles without any warnings.

Line n9's syntax was added in JDK 7, in which type parameter can be ignored from right side of the statement, it is inferred from left side, so Line n9 also compiles without any warning.

Type parameter can't be removed from declaration part, hence Line n7 causes compilation error.

Both Line n5 and Line n6 are mixing Generic type with Raw type and hence warning is given by the compiler.

Bottom of Form

Top of Form

Question 4: **Correct**

**Question ID: UKOCP12254**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<Integer> list = new ArrayList<>();
9. byte b = 10;
10. list.add(b); //Line n1
11. var mul = list.get(0) \* list.get(0); //Line n2
12. System.out.println(mul);
13. }
14. }

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

* 

**Line n1 causes compilation error**

**(Correct)**

* 

**Line n2 causes compilation error**

* 

**An exception is thrown at runtime**

* 

**10**

* 

**100**

**Explanation**

**UKOCP12254:**

list is of Integer type and variable 'b' is of byte type.

At Line n1, b is auto-boxed to Byte and not Integer and List<Integer> can't store Byte objects, therefore Line n1 causes compilation error.

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

list.get(0) returns Integer and `list.get(0) \* list.get(0)` is evaluated to int, therefore variable 'mul' infers to int. Line n2 compiles successfully.

Bottom of Form

Top of Form

Question 5: **Correct**

**Question ID: UKOCP69107**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> list;
9. list = new ArrayList<>(); //Line n1
10. list.add("A");
11. list.add("E");
12. list.add("I");
13. list.add("O");
14. list.add("U");
15. list.addAll(list.subList(0, 4)); //Line n2
16. System.out.println(list);
17. }
18. }

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

* 

**Line n1 causes compilation error**

* 

**Line n2 causes compilation error**

* 

**An exception is thrown at runtime by Line n2**

* 

**[A, E, I, O, U]**

* 

**[A, E, I, O, U, A, E, I, O, U]**

* 

**[A, E, I, O, U, A, E, I, O]**

**(Correct)**

**Explanation**

**UKOCP69107:**

Starting with JDK 7, Java allows to not specify type while initializing the ArrayList. As variable list is of List<String> type, therefore type of ArrayList is considered as String. Line n1 compiles successfully.

sublist method is declared in List interface:

List<E> subList(int fromIndex, int toIndex)

fromIndex is inclusive and toIndex is exclusive

It returns a view of the portion of this list between the specified fromIndex and toIndex. The returned list is backed by this list, so non-structural changes in the returned list are reflected in this list and vice-versa.

If returned list (or view) is structurally modified, then modification are reflected in this list as well but if this list is structurally modified, then the semantics of the list returned by this method become undefined.

If fromIndex == toIndex, then returned list is empty.

If fromIndex < 0 OR toIndex > size of the list OR fromIndex > toIndex, then IndexOutOfBoundsException is thrown.

At Line n2, list.subList(0, 4) --> [A, E, I, O] (toIndex is Exclusive, therefore start index is 0 and end index is 3].

list.addAll(list.subList(0, 4)); is almost equal to list.addAll(5, [A, E, I, O]); => Inserts A at index 5, E takes index 6, I takes index 7 and O is placed at index 8. list --> [A, E, I, O, U, A, E, I, O]

Last statement inside main(String []) method prints [A, E, I, O, U, A, E, I, O] on to the console.

Bottom of Form

Top of Form

Question 6: **Correct**

**Question ID: UKOCP29919**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> list = new ArrayList<>();
9. list.add("C");
10. list.add("Z");
11. list.add("A");
12. list.add("R");
13. list.subList(1, 2).clear();
14. System.out.println(String.join("", list));
15. }
16. }

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

* 

**CZAR**

* 

**CAR**

**(Correct)**

* 

**CR**

* 

**Compilation error**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP29919:**

list --> [C, Z, A, R]

sublist method is declared in List interface:

List<E> subList(int fromIndex, int toIndex)

fromIndex is inclusive and toIndex is exclusive

It returns a view of the portion of this list between the specified fromIndex and toIndex. The returned list is backed by this list, so non-structural changes in the returned list are reflected in this list and vice-versa.

If returned list (or view) is structurally modified, then modification are reflected in this list as well but if this list is structurally modified, then the semantics of the list returned by this method become undefined.

If fromIndex == toIndex, then returned list is empty.

If fromIndex < 0 OR toIndex > size of the list OR fromIndex > toIndex, then IndexOutOfBoundsException is thrown.

list.subList(1, 2) --> [Z] (fromIndex is inclusive and endIndex is exclusive, so start index is 1 and end index is also 1)

list.subList(1, 2).clear(); => It removes element "Z" from the view and also from the ArrayList object referred by list. After this statement, list --> [C, A, R]

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.

`System.out.println(String.join("", list));` prints CAR on to the console.

Bottom of Form

Top of Form

Question 7: **Correct**

**Question ID: UKOCP33176**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> list = new ArrayList<>();
9. list.add("P");
10. list.add("O");
11. list.add("T");
13. List<String> subList = list.subList(1, 2); //Line n1
14. subList.set(0, "E"); //Line n2
15. System.out.println(String.join("", list));
16. }
17. }

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

* 

**PET**

**(Correct)**

* 

**POT**

* 

**Compilation error**

* 

**An exception is thrown by Line n2**

**Explanation**

**UKOCP33176:**

list --> [P, O, T]

sublist method is declared in List interface:

List<E> subList(int fromIndex, int toIndex)

fromIndex is inclusive and toIndex is exclusive

It returns a view of the portion of this list between the specified fromIndex and toIndex. The returned list is backed by this list, so non-structural changes in the returned list are reflected in this list and vice-versa.

If returned list (or view) is structurally modified, then modification are reflected in this list as well but if this list is structurally modified, then the semantics of the list returned by this method become undefined.

If fromIndex == toIndex, then returned list is empty.

If fromIndex < 0 OR toIndex > size of the list OR fromIndex > toIndex, then IndexOutOfBoundsException is thrown.

list.subList(1, 2) --> [O] (fromIndex is inclusive and endIndex is exclusive, so start index is 1 and end index is also 1). subList --> [O].

At Line n2, `subList.set(0, "E");` => sublist --> [E]. This change is also reflected in the backed list, therefore after this statement, list --> [P, E, T]

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.

`System.out.println(String.join("", list));` prints PET on to the console.

Bottom of Form

Top of Form

Question 8: **Correct**

**Question ID: UKOCP35358**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. class Student {
7. private String name;
8. private int age;
10. Student(String name, int age) {
11. this.name = name;
12. this.age = age;
13. }
15. public String toString() {
16. return "Student[" + name + ", " + age + "]";
17. }
18. }
20. public class Test {
21. public static void main(String[] args) {
22. List<Student> students = new ArrayList<>();
23. students.add(new Student("James", 25));
24. students.add(new Student("James", 27));
25. students.add(new Student("James", 25));
26. students.add(new Student("James", 25));
28. students.remove(new Student("James", 25));
30. for(Student stud : students) {
31. System.out.println(stud);
32. }
33. }
34. }

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

* 

**Student[James, 27]**

**Student[James, 25]**

**Student[James, 25]**

* 

**Student[James, 25]**

**Student[James, 27]**

**Student[James, 25]**

* 

**Student[James, 27]**

* 

**Student[James, 25]**

**Student[James, 27]**

**Student[James, 25]**

**Student[James, 25]**

**(Correct)**

**Explanation**

**UKOCP35358:**

Before you answer this, you must know that there are 5 different Student object created in the memory (4 at the time of adding to the list and 1 at the time of removing from the list). This means these 5 Student objects will be stored at different memory addresses.

remove(Object) method removes the first occurrence of matching object and equals(Object) method decides whether 2 objects are equal or not. equals(Object) method defined in Object class uses == operator to check the equality and in this case as 5 Student objects are stored at different memory location, hence not equal.

Nothing is removed from the students list, all the 4 Student objects are printed in the insertion order.

Bottom of Form

Top of Form

Question 9: **Correct**

**Question ID: UKOCP36689**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. class Employee {
7. private String name;
8. private int age;
10. Employee(String name, int age) {
11. this.name = name;
12. this.age = age;
13. }
15. public String toString() {
16. return "Employee[" + name + ", " + age + "]";
17. }
19. public boolean equals(Object obj) {
20. if(obj instanceof Employee) {
21. Employee emp = (Employee)obj;
22. if(this.name.equals(emp.name) && this.age == emp.age) {
23. return true;
24. }
25. }
26. return false;
27. }
28. }
30. public class Test {
31. public static void main(String[] args) {
32. List<Employee> employees = new ArrayList<>();
33. employees.add(new Employee("William", 25));
34. employees.add(new Employee("William", 27));
35. employees.add(new Employee("William", 25));
36. employees.add(new Employee("William", 25));
38. employees.remove(new Employee("William", 25));
40. for(Employee emp : employees) {
41. System.out.println(emp);
42. }
43. }
44. }

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

* 

**Employee[William, 27]**

**Employee[William, 25]**

**Employee[William, 25]**

**(Correct)**

* 

**Employee[William, 25]**

**Employee[William, 27]**

**Employee[William, 25]**

* 

**Employee[William, 27]**

* 

**Employee[William, 25]**

**Employee[William, 27]**

**Employee[William, 25]**

**Employee[William, 25]**

**Explanation**

**UKOCP36689:**

Before you answer this, you must know that there are 5 different Employee object created in the memory (4 at the time of adding to the list and 1 at the time of removing from the list). This means these 5 Employee objects will be stored at different memory addresses.

remove(Object) method removes the first occurrence of matching object and equals(Object) method decides whether 2 objects are equal or not. equals(Object) method has been overridden by the Employee class and equates the object based on their name and age.

3 matching Employee objects are found in the list and first matching list element is removed from the list. Remaining 3 list elements are printed in the insertion order.

Bottom of Form

Top of Form

Question 10: **Correct**

**Question ID: UKOCP59788**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. class Person {
7. private String name;
8. private int age;
10. Person(String name, int age) {
11. this.name = name;
12. this.age = age;
13. }
15. public String toString() {
16. return "Person[" + name + ", " + age + "]";
17. }
19. public boolean equals(Person obj) {
20. if(obj instanceof Person) {
21. Person person = (Person)obj;
22. if(this.name.equals(person.name) && this.age == person.age) {
23. return true;
24. }
25. }
26. return false;
27. }
28. }
30. public class Test {
31. public static void main(String[] args) {
32. List<Person> persons = new ArrayList<>();
33. persons.add(new Person("Liam", 25));
34. persons.add(new Person("Liam", 27));
35. persons.add(new Person("Liam", 25));
36. persons.add(new Person("Liam", 25));
38. persons.remove(new Person("Liam", 25));
40. for(Person person : persons) {
41. System.out.println(person);
42. }
43. }
44. }

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

* 

**Person[Liam, 27]**

**Person[Liam, 25]**

**Person[Liam, 25]**

* 

**Person[Liam, 25]**

**Person[Liam, 27]**

**Person[Liam, 25]**

* 

**Person[Liam, 27]**

* 

**Person[Liam, 25]**

**Person[Liam, 27]**

**Person[Liam, 25]**

**Person[Liam, 25]**

**(Correct)**

**Explanation**

**UKOCP59788:**

Before you answer this, you must know that there are 5 different Person object created in the memory (4 at the time of adding to the list and 1 at the time of removing from the list). This means these 5 Person objects will be stored at different memory addresses.

remove(Object) method removes the first occurrence of matching object and equals(Object) method decides whether 2 objects are equal or not. equals(Object) method has NOT been overridden by the Person class. In fact, equals(Person) is overloaded. But overloaded version is not invoked while equating the Person objects.

equals(Object) method defined in Object class is invoked and equals(Object) method defined in Object class uses == operator to check the equality and in this case as all the Person objects are stored at different memory location, hence not equal.

Nothing is removed from the persons list, all the 4 Person objects are printed in the insertion order.

Bottom of Form

Top of Form

Question 11: **Correct**

**Question ID: UKOCP68250**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> dryFruits = new ArrayList<>();
9. dryFruits.add("Walnut");
10. dryFruits.add("Apricot");
11. dryFruits.add("Almond");
12. dryFruits.add("Date");
14. for(String dryFruit : dryFruits) {
15. if(dryFruit.startsWith("A")) {
16. dryFruits.remove(dryFruit);
17. }
18. }
20. System.out.println(dryFruits);
21. }
22. }

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

* 

**[Walnut, Apricot, Almond, Date]**

* 

**[Walnut, Date]**

* 

**An exception is thrown at runtime**

**(Correct)**

* 

**Compilation error**

**Explanation**

**UKOCP68250:**

ConcurrentModificationException exception may be thrown for following condition:

1. Collection is being iterated using Iterator/ListIterator or by using for-each loop.

And

2. Execution of Iterator.next(), Iterator.remove(), ListIterator.previous(), ListIterator.set(E) & ListIterator.add(E) methods. These methods may throw java.util.ConcurrentModificationException in case Collection had been modified by means other than the iterator itself, such as Collection.add(E) or Collection.remove(Object) or List.remove(int) etc.

PLEASE NOTE: for-each loop internally implements Iterator and invokes hasNext() and next() methods.

For the given code, 'dryFruits' list is being iterated using for-each loop (internally as an Iterator).

hasNext() method of Iterator has following implementation:

1. public boolean hasNext() {
2. return cursor != size;
3. }

Where cursor is the index of next element to return and initially it is 0.

1st Iteration: cursor = 0, size = 4, hasNext() returns true. iterator.next() increments the cursor by 1 and returns "Walnut".

2nd Iteration: cursor = 1, size = 4, hasNext() returns true. iterator.next() increments the cursor by 1 and returns "Apricot". As "Apricot" starts with "A", hence dryFruits.remove(dryFruit) removes "Apricot" from the list and hence reducing the list's size by 1, size becomes 3.

3rd Iteration: cursor = 2, size = 3, hasNext() returns true. iterator.next() method throws java.util.ConcurrentModificationException.

If you want to successfully remove the items from ArrayList, while using Iterator or ListIterator, then use Iterator.remove() or ListIterator.remove() method and NOT List.remove(...) method. Using List.remove(...) method while iterating the list (using the Iterator/ListIterator or for-each) may throw java.util.ConcurrentModificationException.

Bottom of Form

Top of Form

Question 12: **Correct**

**Question ID: UKOCP80346**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.Iterator;
5. import java.util.List;

8. public class Test {
9. public static void main(String[] args) {
10. List<String> sports = new ArrayList<>();
11. sports.add("Windsurfing");
12. sports.add("Aerobics");
13. sports.add("Archery");
14. sports.add("Diving");
16. Iterator<String> iterator = sports.iterator();
17. while(iterator.hasNext()) {
18. String sport = iterator.next();
19. if(sport.startsWith("A")) {
20. sports.remove(sport);
21. }
22. }
24. System.out.println(sports);
25. }
26. }

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

* 

**[Windsurfing, Aerobics, Archery, Diving]**

* 

**[Windsurfing, Diving]**

* 

**An exception is thrown at runtime**

**(Correct)**

* 

**Compilation error**

**Explanation**

**UKOCP80346:**

ConcurrentModificationException exception may be thrown for following condition:

1. Collection is being iterated using Iterator/ListIterator or by using for-each loop.

And

2. Execution of Iterator.next(), Iterator.remove(), ListIterator.previous(), ListIterator.set(E) & ListIterator.add(E) methods. These methods may throw java.util.ConcurrentModificationException in case Collection had been modified by means other than the iterator itself, such as Collection.add(E) or Collection.remove(Object) or List.remove(int) etc.

For the given code, 'sports' list is being iterated using the Iterator<String>.

hasNext() method of Iterator has following implementation:

public boolean hasNext() {

    return cursor != size;

}

Where cursor is the index of next element to return and initially it is 0.

1st Iteration: cursor = 0, size = 4, hasNext() returns true. iterator.next() increments the cursor by 1 and returns "Windsurfing".

2nd Iteration: cursor = 1, size = 4, hasNext() returns true. iterator.next() increments the cursor by 1 and returns "Aerobics". As "Aerobics" starts with "A", hence sports.remove(dryFruit) removes "Aerobics" from the list and hence reducing the list's size by 1, size becomes 3.

3rd Iteration: cursor = 2, size = 3, hasNext() returns true. iterator.next() method throws java.util.ConcurrentModificationException.

If you want to remove the items from ArrayList, while using Iterator or ListIterator, then use Iterator.remove() or ListIterator.remove() method and NOT List.remove(...) method. Using List.remove(...) method while iterating the list (using the Iterator/ListIterator or for-each) may throw java.util.ConcurrentModificationException.

Bottom of Form

Top of Form

Question 13: **Correct**

**Question ID: UKOCP45103**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
5. import java.util.ListIterator;

8. public class Test {
9. public static void main(String[] args) {
10. List<String> objects = new ArrayList<>();
11. objects.add("Watch");
12. objects.add("Arrow");
13. objects.add("Anchor");
14. objects.add("Drum");
16. ListIterator<String> iterator = objects.listIterator();
17. while(iterator.hasNext()) {
18. if(iterator.next().startsWith("A")) {
19. iterator.remove();
20. }
21. }
23. System.out.println(objects);
24. }
25. }

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

* 

**[Watch, Arrow, Anchor, Drum]**

* 

**[Watch, Drum]**

**(Correct)**

* 

**An exception is thrown at runtime**

* 

**Compilation error**

**Explanation**

**UKOCP45103:**

If you want to remove the items from ArrayList, while using Iterator or ListIterator, then use Iterator.remove() or ListIterator.remove() method and NOT List.remove() method.

In this case ListIterator.remove() method is used. startsWith("A") returns true for "Arrow" and "Anchor" so these elements are removed from the list. In the output, [Watch, Drum] is displayed.

Bottom of Form

Top of Form

Question 14: **Correct**

**Question ID: UKOCP57561**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<StringBuilder> animals = new ArrayList<>();
9. animals.add(new StringBuilder("Walrus"));
10. animals.add(new StringBuilder("Anaconda"));
11. animals.add(new StringBuilder("Alligator"));
12. animals.add(new StringBuilder("Dog"));
14. for(int i = 0; i < animals.size(); i++) {
15. if(i == 0) {
16. animals.remove(new StringBuilder("Alligator"));
17. }
18. }
20. System.out.println(animals);
21. }
22. }

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

* 

**[Walrus, Anaconda, Alligator, Dog]**

**(Correct)**

* 

**[Walrus, Dog]**

* 

**An exception is thrown at runtime**

* 

**[Walrus, Anaconda, Dog]**

**Explanation**

**UKOCP57561:**

In this example, code is trying to remove an item from the list while iterating using traditional for loop so one can think that this code would throw java.util.ConcurrentModificationException.

But note, java.util.ConcurrentModificationException will never be thrown for traditional for loop. It is thrown for for-each loop or while using Iterator/ListIterator.

In this case `animals.remove(new StringBuilder("Alligator"));` will never remove any items from the list as StringBuilder class doesn't override the equals(Object) method of Object class.

StringBuilder instances created at "animals.add(new StringBuilder("Alligator"));" and "animals.remove(new StringBuilder("Alligator"));" are at different memory locations and equals(Object) method returns false for these instances.

Bottom of Form

Top of Form

Question 15: **Correct**

**Question ID: UKOCP78024**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.List;
5. import java.util.ListIterator;
7. public class Test {
8. public static void main(String[] args) {
9. List<String> list = Arrays.asList("T", "S", "R", "I", "F");
10. ListIterator<String> iter = list.listIterator(5);
11. while(iter.hasPrevious()) {
12. System.out.print(iter.previous());
13. }
14. }
15. }

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

* 

**IRST**

* 

**FIRST**

**(Correct)**

* 

**TSRIF**

* 

**Runtime Exception**

**Explanation**

**UKOCP78024:**

listIterator(index); method allows to have the starting point at any index. Allowed values are between 0 and size of the list.

ListIterater extends Iterator and can be used to iterate in both the directions. If you want to iterate backward then pass the no. of elements in the list to listIterator(index) method. In this case, 'list.listIterator(5);'.

To iterate backward, use hasPrevious() and previous() methods.

Bottom of Form

Top of Form

Question 16: **Correct**

**Question ID: UKOCP62915**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.List;
5. import java.util.ListIterator;
7. public class Test {
8. public static void main(String[] args) {
9. List<String> list = Arrays.asList("T", "S", "R", "I", "F");
10. ListIterator<String> iter = list.listIterator(2);
11. while(iter.hasNext()) {
12. System.out.print(iter.next());
13. }
14. }
15. }

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

* 

**IF**

* 

**RIF**

**(Correct)**

* 

**Runtime Exception**

**Explanation**

**UKOCP62915:**

listIterator(index); method allows to have the starting point at any index. Allowed values are between 0 and size of the list.

If next() method is called, then element at 'specified index' is returned and if previous() method is called, then element at 'specified index - 1' is returned.

Bottom of Form

Top of Form

Question 17: **Correct**

**Question ID: UKOCP21137**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. public class Test {
6. public static void main(String[] args) {
7. List<String> colors = new ArrayList<>();
8. colors.add("RED");
9. colors.add("GREEN");
10. colors.add("BLUE");
11. Iterator<String> iter = colors.iterator();
12. while(iter.hasNext()) {
13. iter.remove();
14. iter.next();
15. }
16. System.out.println(colors.size());
17. }
18. }

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

* 

**0**

* 

**2**

* 

**Runtime exception**

**(Correct)**

**Explanation**

**UKOCP21137:**

Iterator and ListIterater allow to remove elements while iterating. But next() should be called before remove().

In this case, remove() is called before next() and hence IllegalStateException is thrown at runtime.

Bottom of Form

Top of Form

Question 18: **Correct**

**Question ID: UKOCP29090**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. StringBuilder sb = new StringBuilder("Hello");
9. List<StringBuilder> list = new ArrayList<>();
10. list.add(sb);
11. list.add(new StringBuilder("Hello"));
12. list.add(sb);
13. sb.append("World!");
15. System.out.println(list);
16. }
17. }

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

* 

**[Hello, Hello, Hello]**

* 

**[HelloWorld!, Hello, Hello]**

* 

**[HelloWorld!, Hello, HelloWorld!]**

**(Correct)**

* 

**[HelloWorld!, HelloWorld!, HelloWorld!]**

**Explanation**

**UKOCP29090:**

ArrayList's 1st and 3rd items are referring to same StringBuilder instance referred by sb [sb --> {Hello}] and 2nd item is referring to another instance of StringBuilder.

sb.append("World!"); means sb --> {HelloWorld!}, which means 1st and 3rd items of ArrayList now refers to StringBuilder instance containing HelloWorld!

In the output, [HelloWorld!, Hello, HelloWorld!] is printed.

Bottom of Form

Top of Form

Question 19: **Correct**

**Question ID: UKOCP59750**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. String s = new String("Hello");
9. List<String> list = new ArrayList<>();
10. list.add(s);
11. list.add(new String("Hello"));
12. list.add(s);
13. s.replace("l", "L");
15. System.out.println(list);
16. }
17. }

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

* 

**[Hello, Hello, Hello]**

**(Correct)**

* 

**[HeLLo, Hello, Hello]**

* 

**[HeLLo, Hello, HeLLo]**

* 

**[HeLLo, HeLLo, HeLLo]**

**Explanation**

**UKOCP59750:**

ArrayList's 1st and 3rd items are referring to same String instance referred by 's' [s --> "Hello"] and 2nd item is referring to another instance of String.

String is immutable, which means s.replace("l", "L"); creates another String instance "HeLLo" but 's' still refers to "Hello" [s --> "Hello"].

[Hello, Hello, Hello] is printed in the output.

Bottom of Form

Top of Form

Question 20: **Correct**

**Question ID: UKOCP33124**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> list = new ArrayList<>();
9. list.add("ONE");
10. list.add("TWO");
11. list.add("THREE");
12. list.add("THREE");
14. if(list.remove(2)) {
15. list.remove("THREE");
16. }
18. System.out.println(list);
19. }
20. }

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

* 

**[ONE, TWO, THREE, THREE]**

* 

**[ONE, TWO, THREE]**

* 

**[ONE, TWO]**

* 

**Compilation error**

**(Correct)**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP33124:**

list.remove(Object) method returns boolean result but list.remove(int index) returns the removed item from the list, which in this case is of String type and not Boolean type and hence if(list.remove(2)) causes compilation error.

Bottom of Form

Top of Form

Question 21: **Incorrect**

**Question ID: UKOCP36634**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. Boolean [] arr = new Boolean[2];
9. List<Boolean> list = new ArrayList<>();
10. list.add(arr[0]);
11. list.add(arr[1]);
13. if(list.remove(0)) {
14. list.remove(1);
15. }
17. System.out.println(list);
18. }
19. }

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

* 

**Compilation error**

**(Incorrect)**

* 

**ArrayIndexOutOfBoundsException is thrown at runtime**

* 

**NullPointerException is thrown at runtime**

**(Correct)**

* 

**[true]**

* 

**[false]**

* 

**[]**

**Explanation**

**UKOCP36634:**

Default values are assigned to all array elements. As Boolean is of reference type, hence arr[0] = null and arr[1] = null. After addition list contains [null, null].

list.remove(0) removes the item at index 0 and returns the removed Boolean object referring to null. If expression can specify Boolean type, so no compilation error over here.

For the boolean expression of if-block, Java runtime tries to extract the stored boolean value using booleanValue() method, and this throws an instance of NullPointerException as booleanValue() method is invoked on null reference.

Bottom of Form

Top of Form

Question 22: **Correct**

**Question ID: UKOCP67376**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> list1 = new ArrayList<>();
9. list1.add("A");
10. list1.add("D");
12. List<String> list2 = new ArrayList<>();
13. list2.add("B");
14. list2.add("C");
16. list1.addAll(1, list2);
18. System.out.println(list1);
19. }
20. }

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

* 

**[A, B, C, D]**

**(Correct)**

* 

**[A, D, B, C]**

* 

**[A, D]**

* 

**[A, B, C]**

**Explanation**

**UKOCP67376:**

list1 --> [A, D],

list2 --> [B, C],

list1.addAll(1, list2); is almost equal to list1.addAll(1, [B, C]); => Inserts B at index 1, C takes index 2 and D is moved to index 3. list1 --> [A, B, C, D]

Bottom of Form

Top of Form

Question 23: **Correct**

**Question ID: UKOCP42074**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> list = new ArrayList<>();
9. list.add(0, "Array");
10. list.add(0, "List");
12. System.out.println(list);
13. }
14. }

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

* 

**[Array]**

* 

**[List]**

* 

**[Array, List]**

* 

**[List, Array]**

**(Correct)**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP42074:**

list.add(0, "Array"); means list --> [Array],

list.add(0, "List"); means insert "List" to 0th index and shift "Array" to right. So after this operation, list --> [List, Array]. In the console, [List, Array] is printed.

Bottom of Form

Top of Form

Question 24: **Correct**

**Question ID: UKOCP38817**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> list = new ArrayList<>();
9. list.add(0, "I");
10. list.set(0, "CAN");
12. System.out.println(list);
13. }
14. }

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

* 

**[I]**

* 

**[CAN]**

**(Correct)**

* 

**[I, CAN]**

* 

**[CAN, I]**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP38817:**

list.add(0, "I"); means list --> [I],

list.set(0, "CAN"); means replace the current element at index 0 with the passed element "CAN". So after this operation, list --> [CAN].

[CAN] is printed on to the console.

Bottom of Form

Top of Form

Question 25: **Correct**

**Question ID: UKOCP13516**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> list = new ArrayList<>(4);
9. list.add(0, "MOVE");
10. list.add(2, "ON");
12. System.out.println(list);
13. }
14. }

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

* 

**[MOVE, ON]**

* 

**[MOVE, null, ON, null]**

* 

**[null, null, null, null]**

* 

**[]**

* 

**Compilation error**

* 

**An exception is thrown at runtime**

**(Correct)**

**Explanation**

**UKOCP13516:**

ArrayList are different than arrays, though behind the scene ArrayList uses Object[] to store its elements.

There are 2 things related to ArrayList, one is capacity and another is actual elements stored in the list, returned by size() method. If you don't pass anything to the ArrayList constructor, then default capacity is 10 but this doesn't mean that an ArrayList instance will be created containing 10 elements and all will be initialized to null.

In fact, size() method will still return 0 for this list. This list still doesn't contain even a single element. You need to use add method or its overloaded counterpart to add items to the list. Even if you want to add null values, you should still invoke some methods, nothing happens automatically.

In this question, new ArrayList<>(4); creates an ArrayList instance which can initially store 4 elements but currently it doesn't store any data.

Another point you should remember for the certification exam: Addition of elements in ArrayList should be continuous. If you are using add(index, Element) method to add items to the list, then index should be continuous, you simply can't skip any index.

In this case, list.add(0, "MOVE"); adds "MOVE" to 0th index. so after this operation list --> [MOVE].  You can now add at 0th index (existing elements will be shifted right) or you can add at index 1 but not at index 2. list.add(2, "ON"); throws an instance of java.lang.IndexOutOfBoundsException.

Bottom of Form

Top of Form

Question 26: **Correct**

**Question ID: UKOCP35302**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
5. class Counter {
6. int count;
7. Counter(int count) {
8. this.count = count;
9. }
11. public String toString() {
12. return "Counter-" + count;
13. }
14. }
16. public class Test {
17. public static void main(String[] args) {
18. ArrayList<Counter> original = new ArrayList<>(); //Line n1
19. original.add(new Counter(10)); //Line n2
21. ArrayList<Counter> cloned = (ArrayList<Counter>) original.clone();
22. cloned.get(0).count = 5;
24. System.out.println(original);
25. }
26. }

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

* 

**[Counter-5]**

**(Correct)**

* 

**[Counter-10]**

* 

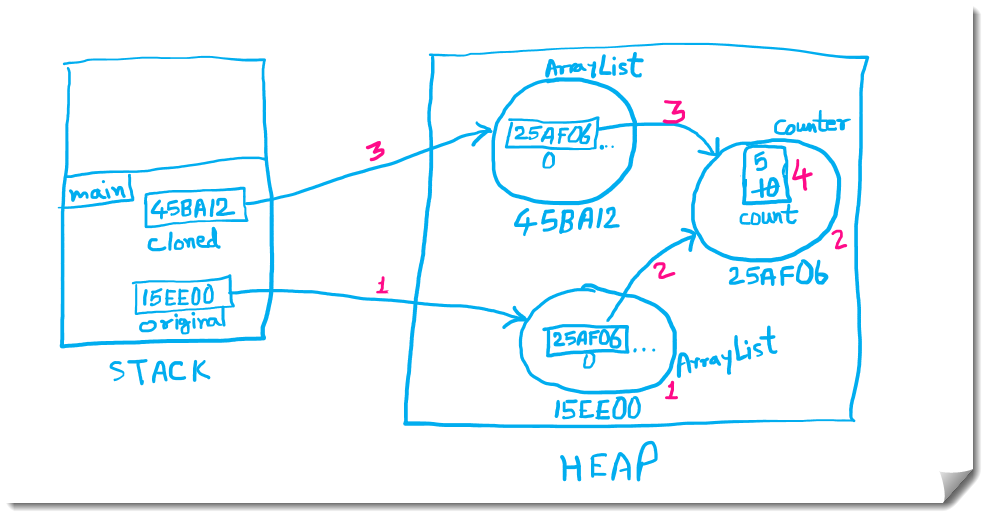
**Compilation error**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP35302:**



Let's see what is happening during execution:

main(String [] args) method goes on to the top of the STACK.

1. ArrayList<Counter> original = new ArrayList<>(); => It creates an ArrayList object [suppose at memory location 15EE00] and variable 'original' refers to it.

2. original.add(new Counter(10)); => It creates a Counter object [suppose at memory location 25AF06] and adds it as a first element of the ArrayList. This means element at 0th index of the ArrayList instance refers to Counter object at the memory location 25AF06.

3. ArrayList<Counter> cloned = (ArrayList<Counter>) original.clone(); => original.clone() creates a new array list object [suppose at memory location 45BA12] and then copies the contents of the ArrayList object stored at [15EE00]. So, cloned contains memory address of the same Counter object.

In this case, original != cloned, but original.get(0) == cloned.get(0). This means both the array lists are created at different memory location but refer to same Counter object.

4. cloned.get(0).count = 5; => cloned.get(0) returns the Counter object stored at the memory location 25AF06 and .count = 5 means change the value of count variable of the Counter object (stored at memory location 25AF06) to 5.

5. System.out.println(original); Prints the element of ArrayList 'original', which is: {25AF06} and toString() method prints: [Counter-5] as Counter object referred by [25AF06] is [Counter object (5)].

Bottom of Form

Top of Form

Question 27: **Correct**

**Question ID: UKOCP72714**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
5. public class Test {
6. public static void main(String[] args) {
7. ArrayList<Integer> original = new ArrayList<>(); //Line n1
8. original.add(10); //Line n2
10. ArrayList<Integer> cloned = (ArrayList<Integer>) original.clone();
11. Integer i1 = cloned.get(0);
12. ++i1;
14. System.out.println(cloned);
15. }
16. }

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

* 

**[11]**

* 

**[10]**

**(Correct)**

* 

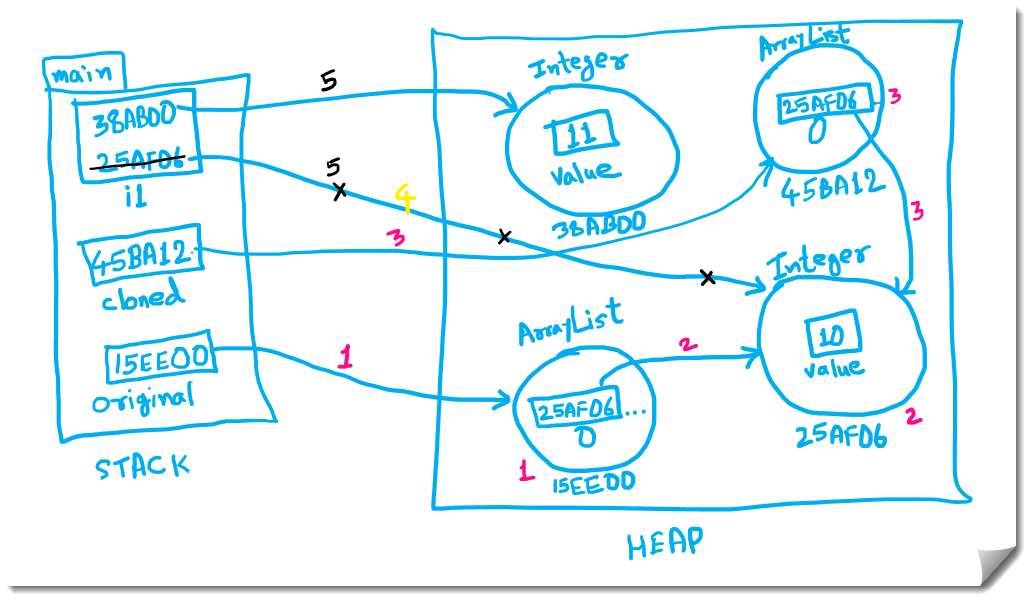
**Compilation error**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP72714:**



Let's see what is happening during execution:

main(String [] args) method goes on to the top of the STACK.

1. ArrayList<Integer> original = new ArrayList<>(); => It creates an ArrayList object [suppose at memory location 15EE00] and variable 'original' refers to it.

2. original.add(new Integer(10)); => It creates an Integer object [suppose at memory location 25AF06] and adds it as a first element of the ArrayList. This means element at 0th index of the ArrayList instance refers to Integer object at the memory location 25AF06.

3. ArrayList<Integer> cloned = (ArrayList<Integer>) original.clone(); => original.clone() creates a new array list object [suppose at memory location 45BA12] and then copies the contents of the ArrayList object stored at [15EE00]. So, cloned contains memory address of the same Integer object.

In this case, original != cloned, but original.get(0) == cloned.get(0). This means both the array lists are created at different memory location but refer to same Integer object.

4. Integer i1 = cloned.get(0); => cloned.get(0) returns the Integer object stored at the memory location 25AF06 and variable 'i1' refers to it.

5. ++i1; => As Integer object is immutable, hence ++i1; creates a new Integer object with value 11 and suppose this newly created Integer object is stored at memory location 38AB00. This means variable 'i1' stops referring to Integer object at the memory location 25AF06 and starts referring to Integer object at the memory location 38AB00.

Cloned list stays intact and still refers to Integer object at memory location 25AF06.

6. System.out.println(cloned); => Prints [10] on to the console as 'cloned' contains an element which refers to Integer object containing value 10.

Bottom of Form

Top of Form

Question 28: **Incorrect**

**Question ID: UKOCP10003**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> trafficLight = new ArrayList<>();
9. trafficLight.add(1, "RED");
10. trafficLight.add(2, "ORANGE");
11. trafficLight.add(3, "GREEN");
13. trafficLight.remove(Integer.valueOf(2));
15. System.out.println(trafficLight);
16. }
17. }

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

* 

**Compilation error**

* 

**An exception is thrown at runtime**

**(Correct)**

* 

**[RED, GREEN]**

* 

**[RED, ORANGE]**

**(Incorrect)**

* 

**[RED, ORANGE, GREEN]**

**Explanation**

**UKOCP10003:**

There is no element at index 0 so call to add element at index 1, "trafficLight.add(1, "RED");" throws an instance of java.lang.IndexOutOfBoundsException.

`trafficLight.remove(Integer.valueOf(2));` matches with trafficLight.remove(Object) and hence no compilation error.

Bottom of Form

Top of Form

Question 29: **Correct**

**Question ID: UKOCP35306**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> days = new ArrayList<>();
9. days.add("SUNDAY");
10. days.add("SUNDAY");
11. days.add("MONDAY");
12. System.out.println(days.size());
13. days.clear();
14. System.out.println(days.size());
15. }
16. }

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

* 

**3**

**3**

* 

**3**

**0**

**(Correct)**

* 

**2**

**0**

* 

**2**

**2**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP35306:**

ArrayList can have duplicate elements, so after addition, list is: [SUNDAY, SUNDAY, MONDAY]. days.size() returns 3, so 3 is printed on to the console.

days.clear(); removes all the elements from the days list, in fact days list will be empty after successful execution of days.clear();

2nd System.out.println statement prints 0 on to the console.

Bottom of Form

Top of Form

Question 30: **Correct**

**Question ID: UKOCP33129**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<StringBuilder> gemStones = new ArrayList<>();
9. gemStones.add(new StringBuilder("Sapphire"));
10. gemStones.add(new StringBuilder("Emerald"));
11. gemStones.add(new StringBuilder("Ruby"));
13. if(gemStones.contains(new StringBuilder("Sapphire"))) {
14. gemStones.add(new StringBuilder("Diamond"));
15. }
17. System.out.println(gemStones.size());
18. }
19. }

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

* 

**4**

* 

**3**

**(Correct)**

* 

**Compilation error**

* 

**Runtime exception**

**Explanation**

**UKOCP33129:**

Three StringBuilder objects [{"Sapphire"}, {"Emerald"}, {"Ruby"}] are added to the gemStones list.

StringBuilder class doesn't override equals(Object) method and hence `gemStones.contains(new StringBuilder("Sapphire"))` returns false. Code inside if-block is not executed and days.size() returns 3.

Bottom of Form

Top of Form

Question 31: **Correct**

**Question ID: UKOCP71868**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<Character> list = new ArrayList<>();
9. list.add(0, 'V');
10. list.add('T');
11. list.add(1, 'E');
12. list.add(3, 'O');
14. if(list.contains('O')) {
15. list.remove(3);
16. }
18. for(char ch : list) {
19. System.out.print(ch);
20. }
21. }
22. }

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

* 

**Compilation error**

* 

**Runtime error**

* 

**VET**

**(Correct)**

* 

**VTE**

* 

**VTEO**

* 

**VETO**

**Explanation**

**UKOCP71868:**

list.add(0, 'V'); => char 'V' is converted to Character object and stored as the first element in the list. list --> [V].

list.add('T'); => char 'T' is auto-boxed to Character object and stored at the end of the list. list --> [V,T].

list.add(1, 'E'); => char 'E' is auto-boxed to Character object and inserted at index 1 of the list, this shifts T to the right. list --> [V,E,T].

list.add(3, 'O'); => char 'O' is auto-boxed to Character object and added at index 3 of the list. list --> [V,E,T,O].

list.contains('O') => char 'O' is auto-boxed to Character object and as Character class overrides equals(String) method this expression returns true. Control goes inside if-block and executes: list.remove(3);.

list.remove(3); => Removes last element of the list. list --> [V,E,T].

for(char ch : list) => First list item is Character object, which is auto-unboxed and assigned to ch. This means in first iteration ch = 'V'; And after this it is simple enhanced for loop. Output is VET.

Bottom of Form

Top of Form

Question 32: **Incorrect**

**Question ID: UKOCP22470**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<Character> list = new ArrayList<>();
9. list.add(0, 'E');
10. list.add('X');
11. list.add(1, 'P');
12. list.add(3, 'O');
14. if(list.contains('O')) {
15. list.remove('O');
16. }
18. for(char ch : list) {
19. System.out.print(ch);
20. }
21. }
22. }

What is the result?

* 

**Compilation error**

* 

**Runtime error**

**(Correct)**

* 

**EPX**

**(Incorrect)**

* 

**EXP**

* 

**EPXO**

* 

**EXPO**

**Explanation**

**UKOCP22470:**

list.add(0, 'E'); => char 'E' is converted to Character object and stored as the first element in the list. list --> [E].

list.add('X'); => char 'X' is auto-boxed to Character object and stored at the end of the list. list --> [E,X].

list.add(1, 'P'); => char 'P' is auto-boxed to Character object and inserted at index 1 of the list, this shifts X to the right. list --> [E,P,X].

list.add(3, 'O'); => char 'O' is auto-boxed to Character object and added at index 3 of the list. list --> [E,P,X,O].

list.contains('O') => char 'O' is auto-boxed to Character object and as Character class overrides equals(String) method this expression returns true. Control goes inside if-block and executes: list.remove('O');.

remove method is overloaded: remove(int) and remove(Object). char can be easily assigned to int so compiler tags remove(int) method. list.remove(<ASCCI value of 'O'>); ASCCI value of 'A' is 65 (this everybody knows) so ASCII value of 'O' will be more than 65.

list.remove('O') throws runtime exception, as it tries to remove an item from the index greater than 65 but allowed index is 0 to 3 only.

Bottom of Form

Top of Form

Question 33: **Correct**

**Question ID: UKOCP21142**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. interface Sellable {}
7. abstract class Animal {}
8. class Mammal extends Animal{}
9. class Rabbit extends Mammal implements Sellable{}
11. public class Test {
12. {
13. List<Animal> list = new ArrayList<>();
14. list.add(new Rabbit());
15. }
16. {
17. List<Animal> list = new ArrayList<>();
18. list.add(new Mammal());
19. }
20. {
21. List<Mammal> list = new ArrayList<>();
22. list.add(new Rabbit());
23. }
24. {
25. List<Sellable> list = new ArrayList<>();
26. list.add(new Mammal());
27. }
28. {
29. List<Sellable> list = new ArrayList<>();
30. list.add(new Rabbit());
31. }
32. }

Which of the following statements is true?

* 

**Only one initializer block causes compilation error**

**(Correct)**

* 

**Two initializer blocks cause compilation error**

* 

**Three initializer blocks cause compilation error**

* 

**Four initializer blocks cause compilation error**

* 

**Five initializer blocks cause compilation error**

**Explanation**

**UKOCP21142:**

Even though code seems to be checking the knowledge of ArrayList but it actually checks the knowledge of Polymorphism.

List<Sellable> list = new ArrayList<>(); is valid statement and list can accept any object passing instanceof check for Sellable type.

Rabbit implements Sellable hence new Rabbit() can be added to list.

But as Mammal doesn't implement Sellable hence new Mammal() can't be added to list.

Other initializer blocks can be verified on similar lines. So there is only one initializer block, which causes compilation error.

Bottom of Form

Top of Form

Question 34: **Correct**

**Question ID: UKOCP45572**

Below is the code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. abstract class Animal {}
7. class Dog extends Animal{}
9. public class Test {
10. public static void main(String [] args) {
11. List<Animal> list = new ArrayList<Dog>();
12. list.add(0, new Dog());
13. System.out.println(list.size() > 0);
14. }
15. }

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

* 

**true**

* 

**false**

* 

**Compilation error**

**(Correct)**

* 

**Runtime exception**

**Explanation**

**UKOCP45572:**

List is super type and ArrayList is sub type, hence List l = new ArrayList(); is valid syntax.

Animal is super type and Dog is sub type, hence Animal a = new Dog(); is valid syntax. Both depicts Polymorphism.

But in generics syntax, Parameterized types are not polymorphic, this means ArrayList<Animal> is not super type of ArrayList<Dog>. Remember this point. So below syntaxes are not allowed:

ArrayList<Animal> list = new ArrayList<Dog>(); OR List<Animal> list = new ArrayList<Dog>();

Bottom of Form

Top of Form

Question 35: **Incorrect**

**Question ID: UKOCP44241**

Below is the code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String [] args) {
8. List<Integer> list = new ArrayList<Integer>();
9. list.add(2);
10. list.add(1);
11. list.add(0);
13. list.remove(list.indexOf(0));
15. System.out.println(list);
16. }
17. }

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

* 

**Compilation error**

* 

**An exception is thrown at runtime**

* 

**[1, 0]**

**(Incorrect)**

* 

**[2, 1]**

**(Correct)**

* 

**[2, 1, 0]**

**Explanation**

**UKOCP44241:**

list.add(2); => 2 is auto-boxed to Integer and added to the list.

list.add(1); => 1 is auto-boxed to Integer and added to the list.

list.add(0); => 0 is auto-boxed to Integer and added to the list.

remove method of List interface is overloaded: remove(int) and remove(Object).

indexOf method accepts argument of Object type, in this case list.indexOf(0) => 0 is auto-boxed to Integer object so no issues with indexOf code. list.indexOf(0) returns 2 (index at which 0 is stored in the list). So list.remove(list.indexOf(0)); is converted to list.remove(2);

remove(int) version is matched, it's a direct match so compiler doesn't do auto-boxing in this case. list.remove(2) removes the element at index 2, which is 0.

Hence in the output, you get [2, 1].

Bottom of Form

Top of Form

Question 36: **Correct**

**Question ID: UKOCP21146**

Below is the code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String [] args) {
8. List<Integer> list = new ArrayList<Integer>();
10. list.add(27);
11. list.add(27);
13. list.add(227);
14. list.add(227);
16. System.out.println(list.get(0) == list.get(1));
17. System.out.println(list.get(2) == list.get(3));
18. }
19. }

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

* 

**false**

**false**

* 

**false**

**true**

* 

**true**

**true**

* 

**true**

**false**

**(Correct)**

**Explanation**

**UKOCP21146:**

This is bit tricky. Just remember this:

Two instances of following wrapper objects, created through auto-boxing will always be same, if their primitive values are same:

Boolean,

Byte,

Character from \u0000 to \u007f (7f equals to 127),

Short and Integer from -128 to 127.

For 1st statement, list.add(27); => Auto-boxing creates an integer object for 27.

For 2nd statement, list.add(27); => Java compiler finds that there is already an Integer object in the memory with value 27, so it uses the same object.

That is why `System.out.println(list.get(0) == list.get(1));` returns true.

For 3rd statement, list.add(227); => Auto-boxing creates an integer object for 227.

For 4th statement, list.add(227); => As 227 is greater than 127, hence auto-boxing creates another integer object for 227.

As both the objects are created at different memory locations, hence `System.out.println(list.get(2) == list.get(3));` returns false.

Bottom of Form

Top of Form

Question 37: **Incorrect**

**Question ID: UKOCP23323**

Below is the code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<Integer> list = new ArrayList<>();
9. list.add(100);
10. list.add(200);
11. list.add(100);
12. list.add(200);
13. list.remove(100);
15. System.out.println(list);
16. }
17. }

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

* 

**[200, 100, 200]**

**(Incorrect)**

* 

**[100, 200, 200]**

* 

**[200, 200]**

* 

**[200]**

* 

**Compilation error**

* 

**Exception is thrown at runtime**

**(Correct)**

**Explanation**

**UKOCP23323:**

List cannot accept primitives, it can accept objects only. So, when 100 and 200 are added to the list, then auto-boxing feature converts these to wrapper objects of Integer type.

So, 4 items gets added to the list. One can expect the same behavior with remove method as well that 100 will be auto-boxed to Integer object.

But remove method is overloaded in List interface: remove(int) => Removes the element from the specified position in this list.

and remove(Object)  => Removes the first occurrence of the specified element from the list.

As remove(int) version is available, which perfectly matches with the call remove(100); hence compiler does not do auto-boxing in this case.

But at runtime remove(100) tries to remove the element at 100th index and this throws IndexOutOfBoundsException.

Bottom of Form

Top of Form

Question 38: **Correct**

**Question ID: UKOCP78034**

Below is the code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<Integer> list = new ArrayList<>();
9. list.add(15);
10. list.add(25);
11. list.add(15);
12. list.add(25);
13. list.remove(Integer.valueOf(15));
15. System.out.println(list);
16. }
17. }

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

* 

**[25, 15, 25]**

**(Correct)**

* 

**[15, 25, 25]**

* 

**[25, 25]**

* 

**[25]**

* 

**Compilation error**

* 

**Exception is thrown at runtime**

**Explanation**

**UKOCP78034:**

List cannot accept primitives, it can accept objects only. So, when 15 and 25 are added to the list, then auto-boxing feature converts these to wrapper objects of Integer type.

So, 4 items gets added to the list: [15, 25, 15, 25].

`list.remove(Integer.valueOf(15));` matches with list.remove(Object) and this removes the first occurrence of 15 from the list, which means the 1st element of the list.

After removal list contains: [25, 15, 25].

NOTE: String class and all the wrapper classes override equals(Object) method, hence at the time of removal when another instance is passed [Integer.valueOf(15)], there is no issue in removing the matching item.

Bottom of Form

Top of Form

Question 39: **Correct**

**Question ID: UKOCP44247**

Below is the code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> fruits = new ArrayList<>();
9. fruits.add("apple");
10. fruits.add("orange");
11. fruits.add("grape");
12. fruits.add("mango");
13. fruits.add("banana");
14. fruits.add("grape");
16. if(fruits.remove("grape"))
17. fruits.remove("papaya");
19. System.out.println(fruits);
20. }
21. }

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

* 

**An exception is thrown at runtime**

* 

**Compilation error**

* 

**[apple, orange, mango, banana]**

* 

**[apple, orange, mango, banana, grape]**

**(Correct)**

* 

**[apple, orange, grape, mango, banana, grape]**

**Explanation**

**UKOCP44247:**

remove(Object) method of List interface removes the first occurrence of the specified element from the list, if it is present. If this list does not contain the element, it is unchanged. remove(Object) method returns true, if removal was successful otherwise false.

Initially list has: [apple, orange, grape, mango, banana, grape].

fruits.remove("grape") removes the first occurrence of "grape" and after the successful removal, list has: [apple, orange, mango, banana, grape].

fruits.remove("grape") returns true, control goes inside if block and executes fruits.remove("papaya");

fruits list doesn't have "papaya", so the list remain unchanged. In the console, you get: [apple, orange, mango, banana, grape].

Bottom of Form

Top of Form

Question 40: **Correct**

**Question ID: UKOCP55371**

Below is the code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> places = new ArrayList<>();
9. places.add("Austin");
10. places.add("Okinawa");
11. places.add("Giza");
12. places.add("Manila");
13. places.add("Batam");
14. places.add("Giza");
16. if(places.remove("Giza"))
17. places.remove("Austin");
19. System.out.println(places);
20. }
21. }

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

* 

**An exception is thrown at runtime**

* 

**Compilation error**

* 

**[Okinawa, Manila, Batam]**

* 

**[Austin, Okinawa, Giza, Manila, Batam, Giza]**

* 

**[Austin, Okinawa, Manila, Batam, Giza]**

* 

**[Austin, Okinawa, Manila, Batam]**

* 

**[Okinawa, Manila, Batam, Giza]**

**(Correct)**

* 

**[Okinawa, Giza, Manila, Batam]**

**Explanation**

**UKOCP55371:**

remove(Object) method of List interface removes the first occurrence of the specified element from the list, if it is present. If this list does not contain the element, it is unchanged. remove(Object) method returns true, if removal was successful otherwise false.

Initially list has: [Austin, Okinawa, Giza, Manila, Batam, Giza]. places.remove("Giza") removes the first occurrence of "Giza" and after the successful removal, list has: [Austin, Okinawa, Manila, Batam, Giza]. places.remove("Giza") returns true, control goes inside if block and executes places.remove("Austin");

places list contains "Austin", so after the removal list has: [Okinawa, Manila, Batam, Giza].

Bottom of Form

Top of Form

Question 41: **Correct**

**Question ID: UKOCP53190**

Below is the code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
5. public class Test {
6. public static void main(String[] args) {
7. var list = new ArrayList<>();
8. list.add(null);
9. list.add(null);
10. list.add(null);
11. System.out.println(list.remove(0) + ":" + list.remove(null));
12. }
13. }

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

* 

**true:true**

* 

**true:false**

* 

**null:true**

**(Correct)**

* 

**null:null**

* 

**NullPointerException is thrown at runtime**

**Explanation**

**UKOCP53190:**

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

Given statement:

var list = new ArrayList<>(); => list refers to an ArrayList of Object type, because Generic type is not defined on the right side.

It is possible to add null to ArrayList instance.

Initially list has 3 elements: [null, null, null].

remove(int) returns the deleted member of the list. In this case `list.remove(0);` returns null as null was deleted from the 0th index. So, list is left with 2 elements: [null, null].

remove(Object) returns true if deletion was successful otherwise false. In this case `list.remove(null)` removes first null from the list and returns true and list is left with just one element: [null].

Hence, the output is: 'null:true'.

Bottom of Form

Top of Form

Question 42: **Correct**

**Question ID: UKOCP83848**

Below is the code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. public class Test {
6. public static void main(String[] args) {
7. var list = new ArrayList<>(); //Line n1
8. list.add("TAKE");
9. list.add("THE");
10. list.add("RISK");
12. System.out.println(String.join(".", list)); //Line n2
13. }
14. }

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

* 

**Compilation error at Line n1**

* 

**Compilation error at Line n2**

**(Correct)**

* 

**An exception is thrown at runtime**

* 

**TAKE.THE.RISK.**

* 

**TAKE.THE.RISK**

* 

**TAKETHERISK**

* 

**TAKETHERISK.**

**Explanation**

**UKOCP83848:**

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, list refers to an ArrayList of Object type, because Generic type is not defined on the right side.

Three String objects are added to the list.

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.

Now if you look at the 2nd overloaded method, 2nd parameter is: 'Iterable<? extends CharSequence>', which means if you are passing List, then List<String>, List<StringBuilder> & List<StringBuffer> are possible arguments.

But raw List and List<Object> won't work. As ArrayList instance created at Line n1 is of Object type, hence Line n2 causes compilation error.

If you change Line n1 to: List<String> list = new ArrayList<>();

then output will be: TAKE.THE.RISK

Bottom of Form

Top of Form

Question 43: **Correct**

**Question ID: UKOCP11341**

Below is the code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. public class Test {
6. public static void main(String[] args) {
7. var list = new ArrayList<>(); //Line n1
8. list.add(7);
9. list.add(14);
10. list.add(21);
12. var sum = 0; //Line n2
13. for(int i : list){ //Line n3
14. sum += i;
15. }
16. System.out.println(sum);
17. }
18. }

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

* 

**Compilation error at Line n1**

* 

**Compilation error at Line n2**

* 

**Compilation error at Line n3**

**(Correct)**

* 

**42**

**Explanation**

**UKOCP11341:**

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, list refers to an ArrayList of Object type, because Generic type is not defined on the right side.

Three Integer objects are added to the list.

At Line n2, 'sum' infers to int.

At Line n3, as list is of ArrayList of Object type, hence each element of the list is considered as Object type and Object cannot be assigned to int.

Line n3 causes compilation error.

Bottom of Form

Top of Form

Question 44: **Correct**

**Question ID: UKOCP34441**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List list = new ArrayList<String>();
9. list.add(1);
10. list.add("2");
11. list.forEach(System.out::print);
12. }
13. }

Which of the following is correct?

Select 2 options.

* 

**Code compiles without any errors and warnings**

* 

**Code compiles with some warnings**

**(Correct)**

* 

**Exception is thrown at runtime**

* 

**12 is displayed on to the console**

**(Correct)**

**Explanation**

**UKOCP34441:**

Compiler warning for unchecked call to add and forEach.

list can store all objects and when each element is passed to System.out.print() method, toString() method for passed element is invoked.

Both Integer and String class overrides toString() method and hence 12 is printed on to the console.

Bottom of Form

Top of Form

Question 45: **Correct**

**Question ID: UKOCP58431**

Consider below code snippet:

1. import java.util.\*;
3. class Father {}
5. class Son extends Father {}
7. class GrandSon extends Son {}
9. abstract class Super {
10. abstract List<Father> get();
11. }
13. class Sub extends Super {
14. /\*INSERT\*/
15. }

And the definitions of get() method:

1. List<Father> get() {return null;}

2. ArrayList<Father> get() {return null;}

3. List<Son> get() {return null;}

4. ArrayList<Son> get() {return null;}

5. List<GrandSon> get() {return null;}

6. ArrayList<GrandSon> get() {return null;}

7. List<Object> get() {return null;}

8. ArrayList<Object> get() {return null;}

How many definitions of get() method can replace /\*INSERT\*/ such that there is no compilation error?

* 

**One definition**

* 

**Two definitions**

**(Correct)**

* 

**Three definitions**

* 

**Four definitions**

* 

**Five definitions**

* 

**Six definitions**

* 

**Seven definitions**

* 

**Eight definitions**

**Explanation**

**UKOCP58431:**

There are 2 rules related to return types of overriding method:

1. If return type of overridden method is of primitive type, then overriding method should use same primitive type.

2. If return type of overridden method is of reference type, then overriding method can use same reference type or its sub-type (also known as covariant return type).

ArrayList is a subtype of List, hence overriding method can use List<Father> or ArrayList<Father> as return type. Definitions 1 and 2 are valid.

Please note: even though Son is a subtype of Father, List<Son> is not subtype of List<Father>. Hence definitions 3 and 4 are NOT valid.

On similar lines, even though GrandSon is a subtype of Father, List<GrandSon> is not subtype of List<Father>. Hence definitions 5 and 6 are also NOT valid.

List<Object> is not subtype of List<Father>, definition 7 is NOT valid.

ArrayList<Object> is not subtype of List<Father>, definition 8 is also NOT valid.

Bottom of Form

Top of Form

Question 46: **Correct**

**Question ID: UKOCP68235**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. String[] names = { "Smith", "Brown", "Thomas", "Taylor", "Jones" };
9. List<String> list = new ArrayList<>();
10. for (int x = 0; x < names.length; x++) {
11. list.add(names[x]);
12. switch (x) {
13. case 2:
14. continue;
15. }
16. break;
17. }
18. System.out.println(list.size());
19. }
20. }

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

* 

**0**

* 

**1**

**(Correct)**

* 

**2**

* 

**3**

* 

**4**

* 

**5**

* 

**None of the other options**

**Explanation**

**UKOCP68235:**

break; and continue; are used inside for-loop, hence no compilation error.

In 1st iteration, x = 0. "Smith" is added to the list. There is no matching case found, hence control just goes after the switch-case block and executes break; statement, which takes the control out of the for loop. `System.out.println(list.size());` is executed and this prints 1 on to the console.

Bottom of Form

Top of Form

Question 47: **Correct**

**Question ID: UKOCP42067**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<Integer> list = Arrays.asList(0,2,4,6,8);
9. list.replaceAll(i -> i + 1);
10. System.out.println(list);
11. }
12. }

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

* 

**[0, 2, 4, 6, 8]**

* 

**[1, 3, 5, 7, 9]**

**(Correct)**

* 

**Runtime Exception**

* 

**Compilation error**

**Explanation**

**UKOCP42067:**

replaceAll(UnaryOperator<E> operator) is a default method available in List interface, it replaces each element of this list with the result of applying the operator to that element.

list.replaceAll(i -> i + 1); => Adds 1 to each element of the list. Result is [1, 3, 5, 7, 9].

Bottom of Form

Top of Form

Question 48: **Correct**

**Question ID: UKOCP70542**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.Arrays;
5. import java.util.List;
7. public class Test {
8. public static void main(String[] args) {
9. List<Integer> list = new ArrayList<>(Arrays.asList(1,2,3,4,5,6,7,8,9,10));
10. list.removeIf(i -> i % 2 == 1);
11. System.out.println(list);
12. }
13. }

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

* 

**Compilation Error**

* 

**Runtime Exception**

* 

**[2, 4, 6, 8, 10]**

**(Correct)**

* 

**[1, 3, 5, 7, 9]**

**Explanation**

**UKOCP70542:**

Arrays.asList(...) method returns a list backed with array, so items cannot be added to or removed from the list.

But if this list is passed to the constructor of ArrayList, then new ArrayList instance is created which copies the elements of passed list and elements can be added to or removed from this list.

List<Integer> list = new ArrayList<>(Arrays.asList(1,2,3,4,5,6,7,8,9,10)); => [1,2,3,4,5,6,7,8,9,10].

list.removeIf(i -> i % 2 == 1); => [2,4,6,8,10]. Remove the element for which passed Predicate is true.

Bottom of Form

Top of Form

Question 49: **Correct**

**Question ID: UKOCP58435**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List list = new ArrayList<Integer>();
9. list.add(1);
10. list.add(2);
11. list.add("3"); //Line n1
12. list.removeIf(i -> i % 2 == 1); //Line n2
13. System.out.println(list);
14. }
15. }

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

* 

**Compilation error at Line n1**

* 

**Compilation error at Line n2**

**(Correct)**

* 

**Runtime Exception**

* 

**[2]**

**Explanation**

**UKOCP58435:**

list is of List (raw) type. So, it can accept any object. Line n1 doesn't cause any compilation error.

As list is raw list, which means it is of Object type, hence in Predicate's lambda expression, i is of Object type. Modulus operator (%) cannot be applied to Object type. So, Line n2 causes compilation error.

NOTE: This questions checks whether you can find out the issues when raw and generic types are mixed.

Bottom of Form

Top of Form

Question 50: **Incorrect**

**Question ID: UKOCP33596**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. List<String> list = Arrays.asList("ALL", "IS");
9. list.add("WELL");
10. list.forEach(System.out::print);
11. }
12. }

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

* 

**ALLIS**

* 

**ALLISWELL**

**(Incorrect)**

* 

**WELLALLIS**

* 

**Compilation error**

* 

**Runtime exception**

**(Correct)**

**Explanation**

**UKOCP33596:**

You cannot add or remove elements form the list returned by Arrays.asList(T...) method but elements can be re-positioned.

list.add("WELL"); throws UnsupportedOperationException at runtime.

Bottom of Form

Top of Form

Question 51: **Correct**

**Question ID: UKOCP38804**

Given code:

1. package com.udayankhattry.ocp;
3. import java.util.List;
5. public class Test {
6. public static void main(String[] args) {
7. var list1 = List.of(); //Line n1
8. var list2 = List.of("A"); //Line n2
9. list1 = list2; //Line n3
10. System.out.println(list1.size()); //Line n4
11. }
12. }

What is the result?

* 

**0**

* 

**1**

* 

**An exception is thrown at runtime**

* 

**Compilation error**

**(Correct)**

**Explanation**

**UKOCP38804:**

There are 12 overloaded of methods in List interface:

List.of()

List.of(E e1)

List.of​(E e1, E e2)

List.of(E e1, E e2, E e3)

List.of(E e1, E e2, E e3, E e4)

List.of(E e1, E e2, E e3, E e4, E e5)

List.of(E e1, E e2, E e3, E e4, E e5, E e6)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)

List.of(E... elements)

These create an unmodifiable list containing specified elements.

These methods throw NullPointerException - if an element is null

For Line n1, list1 is of List<Object> type as of() method has no argument.

For Line n2, list2 is of List<String> type as of() method specify single argument of String type.

In generics syntax, parameterized types are not polymorphic, this means List<Object> is not super type of List<String>. Therefore, Line n3 causes compilation error.

Bottom of Form

Top of Form

Question 52: **Correct**

**Question ID: UKOCP59767**

Given code:

1. package com.udayankhattry.ocp;
3. import java.util.List;
5. public class Test {
6. public static void main(String[] args) {
7. var list1 = List.of("A", "B");
8. var list2 = List.of("B", "C");
9. list1.retainAll(list2);
10. System.out.println(list1.size());
11. }
12. }

What is the result?

* 

**1**

* 

**2**

* 

**3**

* 

**4**

* 

**An exception is thrown at runtime**

**(Correct)**

**Explanation**

**UKOCP59767:**

There are 12 overloaded of methods in List interface:

List.of()

List.of(E e1)

List.of​(E e1, E e2)

List.of(E e1, E e2, E e3)

List.of(E e1, E e2, E e3, E e4)

List.of(E e1, E e2, E e3, E e4, E e5)

List.of(E e1, E e2, E e3, E e4, E e5, E e6)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)

List.of(E... elements)

These create an unmodifiable list containing specified elements.

These methods throw NullPointerException - if an element is null

Calling any mutator method on the List(add, addAll, set, remove, removeIf, removeAll, retainAll) will always cause UnsupportedOperationException to be thrown.

As, List object referred by 'list1' variable is unmodifiable, hence `list1.retainAll(list2);` throws UnsupportedOperationException at runtime.

Bottom of Form

Top of Form

Question 53: **Correct**

**Question ID: UKOCP37953**

Given code:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. var list = new ArrayList<String>(); //Line n1
9. list.add("A");
10. list.add("M");
11. var list\_of\_list = List.of(list); //Line n2
12. list\_of\_list.get(0).set(1, "N"); //Line n3
13. list\_of\_list.get(0).forEach(System.out::print); //Line n4
14. }
15. }

What is the result?

* 

**Compilation error at Line n2**

* 

**An exception is thrown by Line n3**

* 

**AM**

* 

**AN**

**(Correct)**

**Explanation**

**UKOCP37953:**

There are 12 overloaded of methods in List interface:

List.of()

List.of(E e1)

List.of​(E e1, E e2)

List.of(E e1, E e2, E e3)

List.of(E e1, E e2, E e3, E e4)

List.of(E e1, E e2, E e3, E e4, E e5)

List.of(E e1, E e2, E e3, E e4, E e5, E e6)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)

List.of(E... elements)

These create an unmodifiable list containing specified elements.

These methods throw NullPointerException - if an element is null

Calling any mutator method on the List(add, addAll, set, remove, removeIf, removeAll, retainAll) will always cause UnsupportedOperationException to be thrown. But if the list element of unmodifiable list itself is mutable, then contents of element can be changed.

Line n1 creates an ArrayList object and variable 'list' infers to ArrayList<String>.

Next two statements add "A" and "M" to the ArrayList object referred by 'list'.

At Line n2, 'list\_of\_list' infers to List<ArrayList<String>> and instance referred by 'list\_of\_list' is unmodifiable. So, if you invoke any mutator method on 'list\_of\_list' reference, you will get UnsupportedOperationException. But object referred by 'list\_of\_list' contains modifiable List object, hence Line n3 doesn't throw any exception. Rather, it changes 2nd element from "M" to "N".

Line n4 prints AN on to the console.

Bottom of Form

Top of Form

Question 54: **Correct**

**Question ID: UKOCP15704**

Given code:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.List;
6. public class Test {
7. public static void main(String[] args) {
8. var list1 = Arrays.asList('A', 'B', 'D');
9. var list2 = List.of('A', 'B', 'C');
10. /\*INSERT\*/
11. System.out.println(list1.equals(list2));
12. }
13. }

And below options:

1.

list2.set(2, 'D');

2.

list1.set(2, 'C');

3.

1. list1.remove('D');
2. list2.remove('C');

4.

list2 = list1;

How many of the above options can replace /\*INSERT\*/ (separately and not together) such that output is true?

* 

**None of the options is valid**

* 

**Only one option is valid**

* 

**Two options are valid**

**(Correct)**

* 

**Three options are valid**

* 

**All four options are valid**

**Explanation**

**UKOCP15704:**

Arrays.asList(...) method returns a list backed with array, so elements cannot be added to or removed from the list. But list elements can be replaced.

Variable 'list1' infers to List<Character> type and refers to List object containing 3 elements ['A', 'B', 'D'], though elements cannot be added/removed from this List object but elements can be replaced.

List.of(...) method creates an unmodifiable list containing specified elements. Calling any mutator method on the List(add, addAll, set, remove, removeIf, removeAll, retainAll) will always cause UnsupportedOperationException to be thrown.

Variable 'list2' infers to List<Character> type and refers to unmodifiable List object containing 3 elements [A', 'B', 'C'].

According to the Javadoc of equals(Object) method of Map interface, it compares the specified object with this list for equality. Returns true if and only if the specified object is also a list, both lists have the same size, and all corresponding pairs of elements in the two lists are equal. (Two elements e1 and e2 are equal if Objects.equals(e1, e2).) In other words, two lists are defined to be equal if they contain the same elements in the same order.

Let's check all the options:

1.

list2.set(2, 'D');

✗ List object referred by 'list2' is unmodifiable, hence it will throw UnsupportedOperationException.

2.

list1.set(2, 'C');

✓ In List object referred by 'list1', elements can be replaced, hence this will change the element at index 2 from 'D' to 'C'. So, list1 would refer to [A', 'B', 'C']. Line n2 will print true.

3.

list1.remove('D');

list2.remove('C');

✗ list1.remove('D'); will throw UnsupportedOperationException.

4.

list2 = list1;

✓ 'list2' will refer to same List object referred by 'list1'. Line n2 will print true.

Bottom of Form

Top of Form

Question 55: **Incorrect**

**Question ID: UKOCP32269**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. class Student {
6. private String name;
7. private int age;
9. Student(String name, int age) {
10. this.name = name;
11. this.age = age;
12. }
14. public String toString() {
15. return "Student[" + name + ", " + age + "]";
16. }
18. public boolean equals(Object obj) {
19. if(obj instanceof Student) {
20. Student stud = (Student)obj;
21. return this.name.equals(stud.name) && this.age == stud.age;
22. }
23. return false;
24. }
25. }
27. public class Test {
28. public static void main(String[] args) {
29. Set<Student> students = new HashSet<>();
30. students.add(new Student("James", 20));
31. students.add(new Student("James", 20));
32. students.add(new Student("James", 22));
34. System.out.println(students.size());
35. }
36. }

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

* 

**3**

**(Correct)**

* 

**2**

**(Incorrect)**

* 

**Runtime Exception**

**Explanation**

**UKOCP32269:**

HashSet makes use of hashCode to find out the correct bucket, it then makes use of equals(Object) method to find out duplicate objects.

Student class correctly overrides equals(Object) method but it doesn't override hashCode() method. This means you get different hashCode for different objects.

HashSet in this case cannot find out duplicate Student objects and 3 Student objects are added to the Set.

System.out.println(students.size()); => Prints 3 on to the console.

To avoid duplicate in the given Set, override hashCode() method in Student class:

1. public int hashCode() {
2. return name.hashCode() + age;
3. }

Bottom of Form

Top of Form

Question 56: **Correct**

**Question ID: UKOCP22481**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. class Student {
6. private String name;
7. private int age;
9. Student(String name, int age) {
10. this.name = name;
11. this.age = age;
12. }
14. public int hashCode() {
15. return name.hashCode() + age;
16. }
18. public String toString() {
19. return "Student[" + name + ", " + age + "]";
20. }
22. public boolean equals(Object obj) {
23. if(obj instanceof Student) {
24. Student stud = (Student)obj;
25. return this.name.equals(stud.name) && this.age == stud.age;
26. }
27. return false;
28. }
29. }
31. public class Test {
32. public static void main(String[] args) {
33. Set<Student> students = new TreeSet<>();
34. students.add(new Student("James", 20));
35. students.add(new Student("James", 20));
36. students.add(new Student("James", 22));
38. System.out.println(students.size());
39. }
40. }

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

* 

**2**

* 

**3**

* 

**Runtime Exception**

**(Correct)**

**Explanation**

**UKOCP22481:**

TreeSet requires you to provide either Comparable or Comparator. If you don't provide Comparator explicitly, then for natural ordering your class should implement Comparable interface.

Student class doesn't use Comparable, hence ClassCastException is thrown at runtime.

Bottom of Form

Top of Form

Question 57: **Correct**

**Question ID: UKOCP78487**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. class Student {
6. private String name;
7. private int age;
9. Student(String name, int age) {
10. this.name = name;
11. this.age = age;
12. }
14. public int hashCode() {
15. return name.hashCode() + age;
16. }
18. public String toString() {
19. return "Student[" + name + ", " + age + "]";
20. }
22. public boolean equals(Object obj) {
23. if(obj instanceof Student) {
24. Student stud = (Student)obj;
25. return this.name.equals(stud.name) && this.age == stud.age;
26. }
27. return false;
28. }
30. public String getName() {return name;}
32. public int getAge() {return age;}
34. public static int compareByName(Student s1, Student s2) {
35. return s1.getName().compareTo(s2.getName());
36. }
37. }
39. public class Test {
40. public static void main(String[] args) {
41. Set<Student> students = new TreeSet<>(Student::compareByName);
42. students.add(new Student("James", 20));
43. students.add(new Student("James", 20));
44. students.add(new Student("James", 22));
46. System.out.println(students.size());
47. }
48. }

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

* 

**1**

**(Correct)**

* 

**2**

* 

**3**

* 

**Runtime Exception**

**Explanation**

**UKOCP78487:**

TreeSet requires you to provide either Comparable or Comparator. NOTE: To be used with TreeSet, it is not needed to override equals(Object) and hashCode() methods.

But in real world projects, it is a good practice to override hashCode() and equals(Object) methods for the classes to be used in Collection framework.

In this case 'new TreeSet<>(Student::compareByName);' provides the instance of Comparator<Student> type. Which compares the names only. All 3 Student objects have same name and hence only first Student object was added to this set.

Bottom of Form

Top of Form

Question 58: **Incorrect**

**Question ID: UKOCP47782**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. public class Test {
6. public static void main(String[] args) {
7. Set<Character> set = new TreeSet<>(Arrays.asList('a','b','c','A','a','c'));
8. set.stream().forEach(System.out::print);
9. }
10. }

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

* 

**abcAac**

**(Incorrect)**

* 

**Aaabcc**

* 

**Aabc**

**(Correct)**

* 

**abc**

**Explanation**

**UKOCP47782:**

TreeSet requires you to provide either Comparable or Comparator. If you don't provide Comparator explicitly, then for natural ordering your class should implement Comparable interface.

Character and all wrapper classes implement Comparable interface, hence Characters are sorted in ascending order. Uppercase characters appears before lowercase characters.

Set doesn't allow duplicate, hence output will always be: 'Aabc'.

Bottom of Form

Top of Form

Question 59: **Correct**

**Question ID: UKOCP57587**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. public class Test {
6. public static void main(String[] args) {
7. Set<String> set = new TreeSet<>(Arrays.asList(null,null,null));
8. long count = set.stream().count();
9. System.out.println(count);
10. }
11. }

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

* 

**0**

* 

**1**

* 

**3**

* 

**Runtime Exception**

**(Correct)**

**Explanation**

**UKOCP57587:**

TreeSet cannot contain null values. Hence, 'new TreeSet<>(Arrays.asList(null,null,null));' throws NullPointerException.

Bottom of Form

Top of Form

Question 60: **Correct**

**Question ID: UKOCP82983**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. public class Test {
6. public static void main(String[] args) {
7. Set<String> set = new HashSet<>(Arrays.asList(null,null,null));
8. long count = set.stream().count();
9. System.out.println(count);
10. }
11. }

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

* 

**0**

* 

**1**

**(Correct)**

* 

**3**

* 

**Runtime Exception**

**Explanation**

**UKOCP82983:**

HashSet cares about uniqueness and allows 1 null value.

Bottom of Form

Top of Form

Question 61: **Correct**

**Question ID: UKOCP57102**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. public class Test {
6. public static void main(String[] args) {
7. NavigableSet<String> set = new TreeSet<>(Arrays.asList("red", "green", "blue", "gray"));
8. System.out.println(set.ceiling("gray"));
9. System.out.println(set.floor("gray"));
10. System.out.println(set.higher("gray"));
11. System.out.println(set.lower("gray"));
12. }
13. }

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

* 

**gray**

**gray**

**green**

**blue**

**(Correct)**

* 

**green**

**blue**

**gray**

**gray**

* 

**green**

**blue**

**green**

**blue**

* 

**gray**

**gray**

**gray**

**gray**

**Explanation**

**UKOCP57102:**

new TreeSet<>(Arrays.asList("red", "green", "blue", "gray")); => [blue, gray, green, red].

set.ceiling("gray") => Returns the least value greater than or equal to the given value, 'gray'.

set.floor("gray") => Returns the greatest value less than or equal to the given value, 'gray'.

set.higher("gray") => Returns the least value strictly greater than the given value, 'green'.

set.lower("gray") => Returns the greatest value strictly less than the given value, 'blue'.

Bottom of Form

Top of Form

Question 62: **Correct**

**Question ID: UKOCP44279**

Consider below code:

1. package com.udayankhattry.ocp;
3. import java.util.Set;
5. public class Test {
6. public static void main(String[] args) {
7. Set<Integer> set = Set.of(10, null, 20, 40, null); //Line n1
8. System.out.println(set.size());
9. }
10. }

What is the result?

* 

**Compilation error**

* 

**IllegalArgumentException is thrown at runtime**

* 

**NullPointerException is thrown at runtime**

**(Correct)**

* 

**3**

* 

**4**

* 

**5**

**Explanation**

**UKOCP44279:**

There are 12 overloaded of methods in Set interface:

Set.of()

Set.of(E e1)

Set.of​(E e1, E e2)

Set.of(E e1, E e2, E e3)

Set.of(E e1, E e2, E e3, E e4)

Set.of(E e1, E e2, E e3, E e4, E e5)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)

Set.of(E... elements)

These create an unmodifiable set containing specified elements.

These methods throw NullPointerException - if an element is null or IllegalArgumentException - if the elements are duplicates

As, while executing Line n1, null is found as the argument to of method, therefore NullPointerException is thrown.

Bottom of Form

Top of Form

Question 63: **Correct**

**Question ID: UKOCP11352**

Given code:

1. package com.udayankhattry.ocp;
3. import java.util.Set;
5. public class Test {
6. public static void main(String[] args) {
7. Set<String> set = Set.of("A", "E", "I", "I", null, "O", "U");
8. System.out.println(set.size());
9. }
10. }

What is the result?

* 

**Compilation error**

* 

**IllegalArgumentException is thrown at runtime**

**(Correct)**

* 

**NullPointerException is thrown at runtime**

* 

**5**

* 

**6**

* 

**7**

**Explanation**

**UKOCP11352:**

There are 12 overloaded of methods in Set interface:

Set.of()

Set.of(E e1)

Set.of​(E e1, E e2)

Set.of(E e1, E e2, E e3)

Set.of(E e1, E e2, E e3, E e4)

Set.of(E e1, E e2, E e3, E e4, E e5)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)

Set.of(E... elements)

These create an unmodifiable set containing specified elements.

These methods throw NullPointerException - if an element is null or IllegalArgumentException - if the elements are duplicates

As, while executing Line n1, duplicate element "I" is processed first, therefore IllegalArgumentException is thrown.

Bottom of Form

Top of Form

Question 64: **Correct**

**Question ID: UKOCP25508**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Set;
5. public class Test {
6. public static void main(String[] args) {
7. var set1 = Set.of("1", "2", "3");
8. var set2 = Set.of("3", "1", "2");
9. var set3 = Set.of("1", "2", "3");
10. System.out.println(set1.equals(set2) + ":" + set1.equals(set3));
11. }
12. }

What is the result?

* 

**true:true**

**(Correct)**

* 

**false:false**

* 

**true:false**

* 

**false:true**

**Explanation**

**UKOCP25508:**

There are 12 overloaded of methods in Set interface:

Set.of()

Set.of(E e1)

Set.of​(E e1, E e2)

Set.of(E e1, E e2, E e3)

Set.of(E e1, E e2, E e3, E e4)

Set.of(E e1, E e2, E e3, E e4, E e5)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)

Set.of(E... elements)

These create an unmodifiable set containing specified elements.

These methods throw NullPointerException - if an element is null or IllegalArgumentException - if the elements are duplicates

According to the Javadoc of equals(Object) method of Set interface, it compares the specified object with this set for equality. Returns true if the specified object is also a set, the two sets have the same size, and every member of the specified set is contained in this set (or equivalently, every member of this set is contained in the specified set).

All three sets contain same elements "1", "2" and "3" and hence the output is true:true

Bottom of Form

Top of Form

Question 65: **Correct**

**Question ID: UKOCP67352**

Given code:

1. package com.udayankhattry.ocp;
3. import java.util.Set;
5. public class Test {
6. public static void main(String[] args) {
7. Set<Character> set = Set.of('A', 'F', 'I', 'O', 'U'); //Line n1
8. set.remove(Character.valueOf('F')); // Line n2
9. System.out.println(set.size());
10. }
11. }

What is the result?

* 

**5**

* 

**4**

* 

**An exception is thrown at Line n1**

* 

**An exception is thrown at Line n2**

**(Correct)**

**Explanation**

**UKOCP67352:**

There are 12 overloaded of methods in Set interface:

Set.of()

Set.of(E e1)

Set.of​(E e1, E e2)

Set.of(E e1, E e2, E e3)

Set.of(E e1, E e2, E e3, E e4)

Set.of(E e1, E e2, E e3, E e4, E e5)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)

Set.of(E... elements)

These create an unmodifiable set containing specified elements.

These methods throw NullPointerException - if an element is null or IllegalArgumentException - if the elements are duplicates

Calling any mutator method on the Set(add, addAll, remove, removeAll, retainAll) will always cause UnsupportedOperationException to be thrown.

As, Set object referred by 'set' variable is unmodifiable, hence Line n2 throws UnsupportedOperationException at runtime.

Bottom of Form

Top of Form

Question 66: **Correct**

**Question ID: UKOCP33145**

Given code:

1. package com.udayankhattry.ocp;
3. import java.util.List;
4. import java.util.Map;
5. import java.util.Set;
7. public class Test {
8. public static void main(String[] args) {
9. var list = List.of("A", "E", "I", "O", "U"); //Line n1
10. var set1 = Set.copyOf(list); //Line n2
12. var map = Map.of(1, "U", 2, "O", 3, "I", 4, "E", 5, "A"); //Line n3
13. var set2 = Set.copyOf(map.values()); //Line n4
15. System.out.println(set1.equals(set2)); //Line n5
16. }
17. }

What is the result?

* 

**Compilation error at Line n2**

* 

**Compilation error at Line n4**

* 

**An exception is thrown by Line n2**

* 

**An exception is thrown by Line n4**

* 

**true**

**(Correct)**

* 

**false**

**Explanation**

**UKOCP33145:**

Variable 'list' refers to unmodifiable List object containing 5 elements ["A", "E", "I", "O", "U"].

Variable 'map' refers to unmodifiable Map object containing 5 pairs [(1, "U"), (2, "O"), (3, "I"), (4, "E"), (5, "A")]

According to the Javadoc of copyOf method:

Returns an unmodifiable Set containing the elements of the given Collection. The given Collection must not be null, and it must not contain any null elements. If the given Collection contains duplicate elements, an arbitrary element of the duplicates is preserved. If the given Collection is subsequently modified, the returned Set will not reflect such modifications.

It throws NullPointerException if passed argument is null, or if it contains any nulls.

Variable 'set1' refers to unmodifiable Set Object containing 5 elements ["A", "E", "I", "O", "U"], order of elements is not significant.

At Line n4, map.values() returns Collection<String> object containing 5 elements ["U", "O", "I", "E", A"] and this object is passed as and argument to the copyOf method. Variable 'set2' refers to unmodifiable Set Object containing 5 elements ["U", "O", "I", "E", A"], order of elements is not significant.

According to the Javadoc of equals(Object) method of Set interface, it compares the specified object with this set for equality. Returns true if the specified object is also a set, the two sets have the same size, and every member of the specified set is contained in this set (or equivalently, every member of this set is contained in the specified set).

Both the sets contain same 5 elements "A", "E", "I", "O", U" and hence the output is true

Bottom of Form

Top of Form

Question 67: **Incorrect**

**Question ID: UKOCP70994**

Given code:

1. package com.udayankhattry.ocp;
3. import java.util.Set;
5. public class Test {
6. public static void main(String[] args) {
7. var sb1 = new StringBuilder("A");
8. var sb2 = new StringBuilder("B");
9. var set1 = Set.of(sb1, sb2); //Line n1
10. var set2 = Set.copyOf(set1); //Line n2
11. System.out.println((set1 == set2) + ":" + set1.equals(set2)); //Line n3
12. }
13. }

What is the result?

* 

**true:true**

**(Correct)**

* 

**true:false**

* 

**false:false**

**(Incorrect)**

* 

**false:true**

**Explanation**

**UKOCP70994:**

There are 12 overloaded of methods in Set interface:

Set.of()

Set.of(E e1)

Set.of​(E e1, E e2)

Set.of(E e1, E e2, E e3)

Set.of(E e1, E e2, E e3, E e4)

Set.of(E e1, E e2, E e3, E e4, E e5)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)

Set.of(E... elements)

These create an unmodifiable set containing specified elements.

These methods throw NullPointerException - if an element is null or IllegalArgumentException - if the elements are duplicates

Variable 'set1' refers to unmodifiable Set object containing two StringBuilder objects.

According to the Javadoc of copyOf method:

Returns an unmodifiable Set containing the elements of the given Collection. The given Collection must not be null, and it must not contain any null elements. If the given Collection contains duplicate elements, an arbitrary element of the duplicates is preserved. If the given Collection is subsequently modified, the returned Set will not reflect such modifications.

It throws NullPointerException if passed argument is null, or if it contains any nulls.

NOTE: If the given Collection is an unmodifiable Set, calling copyOf will generally not create a copy.

`Set.copyOf(set1);` returns the same Set object referred by variable 'set1'. Hence, at Line n2, variable 'set2' refers to same Set object referred by 'set1'.

Therefore set1 == set2 evaluates to true and also set1.equals(set2) evaluates to true.

Output is true:true

Bottom of Form

Top of Form

Question 68: **Correct**

**Question ID: UKOCP58403**

Given code:

1. package com.udayankhattry.ocp;
3. import java.util.Set;
5. public class Test {
6. public static void main(String[] args) {
7. var set1 = Set.of(new StringBuilder("GOD"), new StringBuilder("IS"), new StringBuilder("GREAT"));
8. var set2 = Set.of(new StringBuilder("GOD"), new StringBuilder("IS"), new StringBuilder("GREAT"));
9. System.out.println((set1 == set2) + ":" + set1.equals(set2));
10. }
11. }

What is the result?

* 

**true:true**

* 

**true:false**

* 

**false:false**

**(Correct)**

* 

**false:true**

**Explanation**

**UKOCP58403:**

There are 12 overloaded of methods in Set interface:

Set.of()

Set.of(E e1)

Set.of​(E e1, E e2)

Set.of(E e1, E e2, E e3)

Set.of(E e1, E e2, E e3, E e4)

Set.of(E e1, E e2, E e3, E e4, E e5)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)

Set.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)

Set.of(E... elements)

These create an unmodifiable set containing specified elements.

These methods throw NullPointerException - if an element is null or IllegalArgumentException - if the elements are duplicates

Variable 'set1' refers to unmodifiable Set object containing 3 StringBuilder objects.

Variable 'set2' refers to another unmodifiable Set object containing 3 StringBuilder objects.

Therefore set1 == set2 evaluates to false.

According to the Javadoc of equals(Object) method of Set interface, it compares the specified object with this set for equality. Returns true if the specified object is also a set, the two sets have the same size, and every member of the specified set is contained in this set (or equivalently, every member of this set is contained in the specified set).

To check the equality of the Elements, equals(Object) method of the elements is invoked. StringBuilder class doesn't override equals(Object) method and hence two StringBuilder objects containing same texts are not considered equal. Therefore, `set1.equals(set2)` evaluates to false.

Output is false:false

Bottom of Form

Top of Form

Question 69: **Correct**

**Question ID: UKOCP35329**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. enum TrafficLight {
6. RED, YELLOW, GREEN
7. }
9. public class Test {
10. public static void main(String[] args) {
11. Map<TrafficLight, String> map = new TreeMap<>();
12. map.put(TrafficLight.GREEN, "GO");
13. map.put(TrafficLight.RED, "STOP");
14. map.put(TrafficLight.YELLOW, "READY TO STOP");
16. for(String msg : map.values()) {
17. System.out.println(msg);
18. }
19. }
20. }

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

* 

**STOP**

**READY TO STOP**

**GO**

**(Correct)**

* 

**GO**

**STOP**

**READY TO STOP**

* 

**GO**

**READY TO STOP**

**STOP**

* 

**Printing order cannot be predicted**

**Explanation**

**UKOCP35329:**

TreeMap is the sorted map on the basis on natural ordering of keys (if comparator is not provided).

enum TrafficLight is used as a key for TreeMap. The natural order for enum elements is the sequence in which they are defined. Value corresponding to 'RED' is printed first, followed by value corresponding to 'YELLOW' and finally value for 'GREEN' is printed.

Bottom of Form

Top of Form

Question 70: **Correct**

**Question ID: UKOCP11355**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. enum TrafficLight {
6. RED, YELLOW, GREEN
7. }
9. public class Test {
10. public static void main(String[] args) {
11. Map<TrafficLight, String> map = new TreeMap<>();
12. map.put(TrafficLight.GREEN, "GO");
13. map.put(TrafficLight.RED, "STOP");
14. map.put(TrafficLight.YELLOW, "STOP IN 3 Seconds");
15. map.put(TrafficLight.YELLOW, "READY TO STOP");
17. for(String msg : map.values()) {
18. System.out.println(msg);
19. }
20. }
21. }

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

* 

**STOP**

**READY TO STOP**

**STOP IN 3 Seconds**

**GO**

* 

**STOP**

**STOP IN 3 Seconds**

**READY TO STOP**

**GO**

* 

**STOP**

**READY TO STOP**

**GO**

**(Correct)**

* 

**STOP**

**STOP IN 3 Seconds**

**GO**

* 

**Printing order cannot be predicted**

**Explanation**

**UKOCP11355:**

TreeMap is the sorted map on the basis on natural ordering of keys (if comparator is not provided).

enum TrafficLight is used as a key for TreeMap. The natural order for enum elements is the sequence in which they are defined.

A map doesn't allow duplicate keys. 'map.put(TrafficLight.YELLOW, "READY TO STOP");' replaces the previous value corresponding to 'TrafficLight.YELLOW' with the new value 'READY TO STOP'.

Value corresponding to 'RED' is printed first, followed by value corresponding to 'YELLOW' and finally value for 'GREEN' is printed.

Bottom of Form

Top of Form

Question 71: **Correct**

**Question ID: UKOCP38833**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. public class Test {
6. public static void main(String[] args) {
7. Map<Integer, String> map = new LinkedHashMap<>();
8. map.put(null, "zero");
9. map.put(1, "one");
11. System.out.println(map);
12. }
13. }

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

* 

**{null=zero, 1=one}**

**(Correct)**

* 

**{1=one, null=zero}**

* 

**Order cannot be predicted**

* 

**Runtime Exception**

**Explanation**

**UKOCP38833:**

HashMap and LinkedHashMap can accept 1 null key but TreeMap cannot accept null keys.

LinkedHashMap by default keeps an insertion order so every time you iterate the map, you get same result.

Output will always be: {null=zero, 1=one}

Bottom of Form

Top of Form

Question 72: **Correct**

**Question ID: UKOCP82978**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. public class Test {
6. public static void main(String[] args) {
7. NavigableMap<Integer, String> map = new TreeMap<>();
8. map.put(25, "Pune");
9. map.put(32, "Mumbai");
10. map.put(11, "Sri Nagar");
11. map.put(39, "Chennai");
13. System.out.println(map.headMap(25, true));
14. }
15. }

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

* 

**{11=Sri Nagar}**

* 

**{11=Sri Nagar, 25=Pune}**

**(Correct)**

* 

**{25=Pune, 32=Mumbai, 39=Chennai}**

* 

**{32=Mumbai, 39=Chennai}**

**Explanation**

**UKOCP82978:**

TreeMap is sorted map based on the natural ordering of keys. So, map has entries: {11=Sri Nagar, 25=Pune, 32=Mumbai, 39=Chennai}.

headMap(K toKey, boolean inclusive) => returns the map till toKey, if inclusive is true. Hence the output is: {11=Sri Nagar, 25=Pune}.

For the exam, you should know some of the methods from NavigableMap map. Below are the method calls and outputs for the map object used in this example:

//NavigableMap<K,V> tailMap(K fromKey, boolean inclusive); => Returns a view of the portion of this map whose keys are greater than (or equal to, if 'inclusive' is true) fromKey.

System.out.println(map.tailMap(25, true)); //{25=Pune, 32=Mumbai, 39=Chennai}

//Map.Entry<K,V> firstEntry(); => Returns a key-value mapping associated with the least key in this map.

System.out.println(map.firstEntry()); //11=Sri Nagar

//Map.Entry<K,V> lastEntry(); => Returns a key-value mapping associated with the greatest key in this map.

System.out.println(map.lastEntry()); //39=Chennai

//NavigableMap<K,V> descendingMap(); => Returns a reverse order view of the mappings contained in this map.

System.out.println(map.descendingMap()); //{39=Chennai, 32=Mumbai, 25=Pune, 11=Sri Nagar}

//K floorKey(K key); => Returns the greatest key less than or equal to the given key.

System.out.println(map.floorKey(30)); //25

//K ceilingKey(K key); => Returns the least key greater than or equal to the given key.

System.out.println(map.ceilingKey(30)); //32

Bottom of Form

Top of Form

Question 73: **Incorrect**

**Question ID: UKOCP45111**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. public class Test {
6. public static void main(String[] args) {
7. NavigableMap<Integer, String> map = new TreeMap<>();
8. map.put(25, "Pune");
9. map.put(32, "Mumbai");
10. map.put(11, "Sri Nagar");
11. map.put(39, "Chennai");
13. System.out.println(map.headMap(25));
14. System.out.println(map.tailMap(25));
15. }
16. }

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

* 

**{11=Sri Nagar, 25=Pune}**

**{25=Pune, 32=Mumbai, 39=Chennai}**

* 

**{11=Sri Nagar}**

**{32=Mumbai, 39=Chennai}**

**(Incorrect)**

* 

**{11=Sri Nagar}**

**{25=Pune, 32=Mumbai, 39=Chennai}**

**(Correct)**

* 

**{11=Sri Nagar, 25=Pune}**

**{32=Mumbai, 39=Chennai}**

**Explanation**

**UKOCP45111:**

If you don't use 2nd parameter for headMap() and tailMap() methods to indicate whether keys are inclusive or exclusive,

then by default 'toKey' used in headMap() method is exclusive and 'fromKey' used in tailMap() method is inclusive.

You can confirm this by checking the definition of headMap() and tailMap() methods in TreeMap class.

Methods defined in TreeMap class:

1. public SortedMap<K,V> headMap(K toKey) {
2. return headMap(toKey, false);
3. }
5. public SortedMap<K,V> tailMap(K fromKey) {
6. return tailMap(fromKey, true);
7. }

Bottom of Form

Top of Form

Question 74: **Correct**

**Question ID: UKOCP56246**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Map;
4. import java.util.TreeMap;
6. public class Test {
7. public static void main(String[] args) throws Exception {
8. Map<Integer, String> map = new TreeMap<>();
9. map.put(1, "one");
10. map.put(2, "two");
11. map.put(3, "three");
12. map.put(null, "null");
13. map.forEach((key, value) -> System.out.println("{" + key + ": " + value + "}"));
14. }
15. }

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

* 

**{null: null}**

**{1: one}**

**{2: two}**

**{3: three}**

* 

**{1: one}**

**{2: two}**

**{3: three}**

**{null: null}**

* 

**{1: one}**

**{2: two}**

**{3: three}**

* 

**NullPointerException is thrown at runtime**

**(Correct)**

**Explanation**

**UKOCP56246:**

TreeMap cannot contain null keys. Hence, 'map.put(null, "null");' throws NullPointerException.

Bottom of Form

Top of Form

Question 75: **Incorrect**

**Question ID: UKOCP55399**

Given code:

1. package com.udayankhattry.ocp;
3. import java.util.Map;
5. public class Test {
6. public static void main(String[] args) {
7. var map = Map.of("ONE", 1, "TWO", 2, "THREE", 4, "THREE", 3); //Line n1
8. var res = 0;
9. for(Integer num : map.values()) { //Line n2
10. res += num;
11. }
12. System.out.println(res);
13. }
14. }

What is the result?

* 

**6**

* 

**7**

* 

**10**

* 

**IllegalArgumentException is thrown at Line n1**

**(Correct)**

* 

**Line n2 causes compilation error**

**(Incorrect)**

**Explanation**

**UKOCP55399:**

There are 11 overloaded of methods in Map interface:

Map.of()

Map.of(K k1, V v1)

Map.of(K k1, V v1, K k2, V v2)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9, K k10, V v10)

These create an unmodifiable map containing specified keys and values.

These methods throw NullPointerException - if any key or value is null or IllegalArgumentException - if the keys are duplicates

Please note interfaces List and Set contain 12 overloaded of methods each, whereas interface Map contains 11 overloaded of methods. Varargs version is not available in Map interface.

To provide more than 10 key-value pairs use, Map.ofEntries​(Map.Entry<? extends K,​? extends V>... entries) method.

For the given code, Line n1 throws IllegalArgumentException as key "THREE" is duplicated.

Bottom of Form

Top of Form

Question 76: **Incorrect**

**Question ID: UKOCP45597**

Given code:

1. package com.udayankhattry.ocp;
3. import java.util.Map;
5. public class Test {
6. public static void main(String[] args) {
7. var map = Map.of(
8. 1, "A",
9. 2, "B",
10. 3, "C",
11. 4, "D",
12. 5, "E",
13. 6, "F",
14. 7, "G",
15. 8, "H",
16. 9, "I",
17. 10, "J",
18. 11, "K"
19. );
20. var res = "";
21. for(String str : map.values()) {
22. res += str;
23. }
24. System.out.println(res);
25. }
26. }

What is the result?

* 

**Compilation error**

**(Correct)**

* 

**An exception is thrown at runtime**

**(Incorrect)**

* 

**It prints numbers 1 to 11 in any order**

* 

**It prints alphabets A to K in any order**

**Explanation**

**UKOCP45597:**

There are 11 overloaded of methods in Map interface:

Map.of()

Map.of(K k1, V v1)

Map.of(K k1, V v1, K k2, V v2)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9, K k10, V v10)

These create an unmodifiable map containing specified keys and values.

These methods throw NullPointerException - if any key or value is null or IllegalArgumentException - if the keys are duplicates

Please note interfaces List and Set contain 12 overloaded of methods each, whereas interface Map contains 11 overloaded of methods. Varargs version is not available in Map interface.

To provide more than 10 key-value pairs use, Map.ofEntries​(Map.Entry<? extends K,​? extends V>... entries) method.

For the given code, as we have provided 11 key-value pairs, therefore it causes compilation error.

Bottom of Form

Top of Form

Question 77: **Correct**

**Question ID: UKOCP83821**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.List;
4. import java.util.Map;
6. public class Test {
7. public static void main(String[] args) {
8. var map1 = Map.of("1", "2", "3", "4");
9. var map2 = Map.ofEntries(Map.entry("3", "4"), Map.entry("1", "2"));
10. var list1 = List.of("1", "2", "3", "4");
11. var list2 = List.of("4", "3", "2", "1");
12. System.out.println(map1.equals(map2) + ":" + list1.equals(list2));
13. }
14. }

What is the result?

* 

**true:true**

* 

**false:false**

* 

**true:false**

**(Correct)**

* 

**false:true**

**Explanation**

**UKOCP83821:**

There are 11 overloaded of methods in Map interface:

Map.of()

Map.of(K k1, V v1)

Map.of(K k1, V v1, K k2, V v2)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9)

Map.of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9, K k10, V v10)

These create an unmodifiable map containing specified keys and values.

These methods throw NullPointerException - if any key or value is null or IllegalArgumentException - if the keys are duplicates

Please note interfaces List and Set contain 12 overloaded of methods each, whereas interface Map contains 11 overloaded of methods. Varargs version is not available in Map interface.

To provide more than 10 key-value pairs use, Map.ofEntries​(Map.Entry<? extends K,​? extends V>... entries) method.

Variable 'map1' refers to Map object containing 2 key-value pairs. Variable 'map2' refers to different Map object containing same 2 key-value pairs.

According to the Javadoc of equals(Object) method of Map interface, it compares the specified object with this map for equality. Returns true if the given object is also a map and the two maps represent the same mappings. More formally, two maps m1 and m2 represent the same mappings if m1.entrySet().equals(m2.entrySet()).

Hence, `map1.equals(map2)` evaluates to true.

There are 12 overloaded of methods in List interface:

List.of()

List.of(E e1)

List.of​(E e1, E e2)

List.of(E e1, E e2, E e3)

List.of(E e1, E e2, E e3, E e4)

List.of(E e1, E e2, E e3, E e4, E e5)

List.of(E e1, E e2, E e3, E e4, E e5, E e6)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)

List.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)

List.of(E... elements)

These create an unmodifiable list containing specified elements.

These methods throw NullPointerException - if an element is null

Variable 'list1' refers to List object containing elements "1", "2" , "3" & "4" in order. Variable 'list2' refers to List object containing elements "4", "3", "2" & "1" in order.

According to the Javadoc of equals(Object) method of Map interface, it compares the specified object with this list for equality. Returns true if and only if the specified object is also a list, both lists have the same size, and all corresponding pairs of elements in the two lists are equal. (Two elements e1 and e2 are equal if Objects.equals(e1, e2).) In other words, two lists are defined to be equal if they contain the same elements in the same order.

Hence, `list1.equals(list2)` evaluates to false.

Bottom of Form

Top of Form

Question 78: **Correct**

**Question ID: UKOCP21167**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.\*;
5. public class Test {
6. public static void main(String[] args) {
7. Deque<Integer> deque = new ArrayDeque<>();
8. deque.add(100);
9. deque.add(200);
10. deque.addFirst(300);
11. deque.addLast(400);
12. deque.remove(200);
14. System.out.println(deque.getFirst());
15. }
16. }

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

* 

**100**

* 

**200**

* 

**300**

**(Correct)**

* 

**400**

**Explanation**

**UKOCP21167:**

deque.add(100); => {\*100}.  \* represents HEAD of the deque.

deque.add(200); => {\*100, 200}.  add(E e) invokes addLast(e) method.

deque.addFirst(300); => {\*300, 100, 200}.

deque.addLast(400); => {\*300, 100, 200, 400}.

deque.remove(200); => {\*300, 100, 400}.  Deque interface doesn't have remove(int index) method.

System.out.println(deque.getFirst()); => Prints 300 on to the console.

You should be aware of other methods from Deque interface as well, such as:

removeFirst(); => Removes the first element from the Deque.

removeLast(); => Removes the last element from the Deque.

Bottom of Form

Top of Form

Question 79: **Incorrect**

**Question ID: UKOCP46446**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayDeque;
4. import java.util.Deque;
6. public class Test {
7. public static void main(String[] args) {
8. Deque<Boolean> deque = new ArrayDeque<>();
9. deque.push(Boolean.valueOf("abc"));
10. deque.push(Boolean.valueOf("tRuE"));
11. deque.push(Boolean.valueOf("FALSE"));
12. deque.push(true);
13. System.out.println(deque.pop() + ":" + deque.peek() + ":" + deque.size());
14. }
15. }

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

* 

**true:true:3**

* 

**false:false:3**

* 

**false:true:3**

**(Incorrect)**

* 

**true:false:3**

**(Correct)**

**Explanation**

**UKOCP46446:**

push, pop and peek are Stack's terminology.

push(E) calls addFirst(E), pop() calls removeFirst() and peek() invokes peekFirst(), it just retrieves the first element (HEAD) but doesn't remove it.

deque.push(Boolean.valueOf("abc")); => [\*false]. \* represents HEAD of the deque.

deque.push(Boolean.valueOf("tRuE")); => [\*true, false].

deque.push(Boolean.valueOf("FALSE")); => [\*false, true, false].

deque.push(true); => [\*true, false, true, false].

deque.pop() => removes and returns the HEAD element, true in this case. deque => [\*false, true, false].

deque.peek() => retrieves but doesn't remove the HEAD element, false in this case. deque => [\*false, true, false].

deque.size() => 3.  Hence output is 'true:false:3'.

Bottom of Form

Top of Form

Question 80: **Correct**

**Question ID: UKOCP54064**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.LinkedList;
4. import java.util.List;
5. import java.util.Queue;
7. public class Test {
8. public static void main(String[] args) {
9. List<String> list = new LinkedList<>();
10. list.add("ONE");
11. list.add("TWO");
12. list.remove(1);
13. System.out.println(list);
15. Queue<String> queue = new LinkedList<>();
16. queue.add("ONE");
17. queue.add("TWO");
18. queue.remove();
19. System.out.println(queue);
20. }
21. }

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

* 

**[ONE]**

**[TWO]**

**(Correct)**

* 

**[TWO]**

**[ONE]**

* 

**[TWO]**

**[TWO]**

* 

**[ONE]**

**[ONE]**

**Explanation**

**UKOCP54064:**

LinkedList implements both List and Queue. In this example reference type controls the LinkedList behavior.

list.add("ONE"); => [ONE].

list.add("TWO"); => [ONE,TWO]. Adds to the last.

list.remove(1); => [ONE]. Removes from the specified index.

System.out.println(list); => [ONE].

queue.add("ONE"); => [\*ONE].  \* represents HEAD of the queue.

queue.add("TWO"); => [\*ONE,TWO]. Adds to the end of the queue.

queue.remove(); => [\*TWO]. Removes from the HEAD of the queue.

System.out.println(queue); => [TWO].

Bottom of Form

Top of Form

Question 81: **Incorrect**

**Question ID: UKOCP26854**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayDeque;
4. import java.util.Deque;
6. public class Test {
7. public static void main(String[] args) {
8. Deque<Character> chars = new ArrayDeque<>();
9. chars.add('A');
10. chars.add('B');
11. chars.remove();
12. chars.add('C');
13. chars.remove();
15. System.out.println(chars);
16. }
17. }

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

* 

**[A]**

**(Incorrect)**

* 

**[B]**

* 

**[C]**

**(Correct)**

**Explanation**

**UKOCP26854:**

Deque's add() method invokes addLast(E) method and remove() method invokes removeFirst() method.

chars.add('A'); => [\*A], {\* represents HEAD element}

chars.add('B'); => [\*A,B],

chars.remove(); => [\*B],

chars.add('C'); => [\*B,C],

chars.remove(); => [\*C],

System.out.println(chars); => Prints [C] on to the console.

Bottom of Form

Top of Form

Question 82: **Correct**

**Question ID: UKOCP60726**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayDeque;
4. import java.util.Deque;
6. public class Test {
7. public static void main(String[] args) {
8. Deque<Character> chars = new ArrayDeque<>();
9. chars.add('A');
10. chars.remove();
11. chars.remove();
13. System.out.println(chars);
14. }
15. }

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

* 

**[]**

* 

**[A]**

* 

**Runtime Exception**

**(Correct)**

**Explanation**

**UKOCP60726:**

Deque's add() method invokes addLast(E) method and remove() method invokes removeFirst() method.

chars.add('A'); => [A],

chars.remove(); => [],

chars.remove(); => No elements left to remove() and hence java.util.NoSuchElementException is thrown at runtime.

Bottom of Form

Top of Form

Question 83: **Correct**

**Question ID: UKOCP55390**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.ArrayDeque;
4. import java.util.Arrays;
5. import java.util.Deque;
6. import java.util.List;
8. public class Test {
9. public static void main(String[] args) throws Exception {
10. List<String> list = Arrays.asList("oca", null, "ocp", "java", "null"); //Line n1
11. Deque<String> deque = new ArrayDeque<String>(list); //Line n2
12. System.out.println(deque.size()); //Line n3
13. }
14. }

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

* 

**3**

* 

**4**

* 

**5**

* 

**NullPointerException is thrown at runtime**

**(Correct)**

**Explanation**

**UKOCP55390:**

ArrayDeque cannot store null, hence Line n2 throws NullPointerException exception.

ArrayDeque doesn't store null because its poll() method returns null in case ArrayDeque is empty. If null element was allowed, then it would be not possible to find out whether poll() method is returning null element or ArrayDeque is empty.

Bottom of Form

Top of Form

Question 84: **Correct**

**Question ID: UKOCP77161**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Set;
4. import java.util.TreeSet;
6. class Employee implements Comparable<Employee> {
7. private String name;
8. private int age;
10. Employee(String name, int age) {
11. this.name = name;
12. this.age = age;
13. }
15. @Override
16. public String toString() {
17. return "{" + name + ", " + age + "}";
18. }
20. @Override
21. public int compareTo(Employee o) {
22. return o.age - this.age;
23. }
24. }
26. public class Test {
27. public static void main(String[] args) {
28. Set<Employee> employees = new TreeSet<>();
29. employees.add(new Employee("Udayan", 31));
30. employees.add(new Employee("Neha", 23));
31. employees.add(new Employee("Hou Jian", 42));
32. employees.add(new Employee("Smita", 29));
34. System.out.println(employees);
35. }
36. }

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

* 

**[{Neha, 23}, {Smita, 29}, {Udayan, 31}, {Hou Jian, 42}]**

* 

**[{Hou Jian, 42}, {Udayan, 31}, {Smita, 29}, {Neha, 23}]**

**(Correct)**

* 

**[{Udayan, 31}, {Neha, 23}, {Hou Jian, 42}, {Smita, 29}]**

* 

**Compilation error**

**Explanation**

**UKOCP77161:**

Comparable interface has compareTo(...) method and Comparator interface has compare(...) method.

In this case, class Employee correctly implements Comparable interface.

return o.age - this.age; => This will help to sort the Employee objects in descending order of age and not in ascending order.

As no Comparator is passed in TreeSet, hence it sorts on the basis of implementation of Comparable interface, which means Employee objects will be sorted in descending order of their age.

Bottom of Form

Top of Form

Question 85: **Correct**

**Question ID: UKOCP14873**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.Collections;
5. import java.util.List;
7. class Name {
8. String first;
9. String last;
11. public Name(String first, String last) {
12. this.first = first;
13. this.last = last;
14. }
16. public String getFirst() {
17. return first;
18. }
20. public String getLast() {
21. return last;
22. }
24. public String toString() {
25. return first + " " + last;
26. }
28. }
30. public class Test {
31. public static void main(String[] args) {
32. List<Name> names = Arrays.asList(new Name("Peter", "Lee"), new Name("John", "Smith"),
33. new Name("bonita", "smith"));
35. /\*INSERT\*/
37. System.out.println(names);
38. }
39. }

Currently on executing Test class, [Peter Lee, John Smith, bonita smith] is displayed in the output.

Which of the following options can replace /\*INSERT\*/ such that on executing Test class, [bonita smith, John Smith, Peter Lee] is displayed in the output?

The names list must be sorted in ascending order of first name in case-insensitive manner.

Select 3 options.

* 

**Collections.sort(names, (o1, o2) -> o1.getFirst().compareTo(o2.getFirst()));**

* 

**Collections.sort(names, (o1, o2) -> o1.getFirst().toLowerCase().compareTo(o2.getFirst().toLowerCase()));**

**(Correct)**

* 

**Collections.sort(names, (o1, o2) -> o1.getFirst().toUpperCase().compareTo(o2.getFirst().toUpperCase()));**

**(Correct)**

* 

**Collections.sort(names, (o1, o2) -> o1.getFirst().compareToIgnoreCase(o2.getFirst()));**

**(Correct)**

**Explanation**

**UKOCP14873:**

Collections.sort(names, (o1, o2) -> o1.getFirst().compareTo(o2.getFirst())); => It sorts in the ascending order of first name in case-sensitive manner and displays [John Smith, Peter Lee, bonita smith] in the output.

Collections.sort(names, (o1, o2) -> o1.getFirst().toLowerCase().compareTo(o2.getFirst().toLowerCase())); => At the time of comparison, first names in lower case are considered, this doesn't change the case of displayed output. Output is: [bonita smith, John Smith, Peter Lee].

Collections.sort(names, (o1, o2) -> o1.getFirst().toUpperCase().compareTo(o2.getFirst().toUpperCase())); => At the time of comparison, first names in upper case are considered, this doesn't change the case of displayed output. Output is: [bonita smith, John Smith, Peter Lee].

Collections.sort(names, (o1, o2) -> o1.getFirst().compareToIgnoreCase(o2.getFirst())); => compareToIgnoreCase method compares the first names in case-insensitive manner and displays

[bonita smith, John Smith, Peter Lee] in the output.

Bottom of Form

Top of Form

Question 86: **Correct**

**Question ID: UKOCP12694**

For the code below:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
5. public class Test {
6. public static void main(String[] args) {
7. String [] arr = {"\*\*", "\*\*\*", "\*", "\*\*\*\*\*", "\*\*\*\*"};
8. Arrays.sort(arr, (s1, s2) -> s1.length()-s2.length());
9. for(String str : arr) {
10. System.out.println(str);
11. }
12. }
13. }

What do you need to do so that above code gives following output?

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

* 

**Add the import statement for the Comparator interface: import java.util.Comparator;**

* 

**Existing code without any changes displays above output**

**(Correct)**

* 

**Change the lambda expression to (s1, s2) -> s2.length()-s1.length()**

* 

**Change the lambda expression to (s2, s1) -> s1.length()-s2.length()**

**Explanation**

**UKOCP12694:**

Even though lambda expression is for the compare method of Comparator interface, but in the code name "Comparator" is not used hence import statement is not needed here. Expressions

(s1, s2) -> s2.length()-s1.length() and (s2, s1) -> s1.length()-s2.length() displays the output in reversed order.

Bottom of Form

Top of Form

Question 87: **Incorrect**

**Question ID: UKOCP32286**

Consider below code:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.Collections;
5. import java.util.Comparator;
6. import java.util.List;
8. public class Test {
9. public static void main(String [] args) {
10. List<String> names = Arrays.asList("James", "diana", "Anna");
12. /\*INSERT\*/
14. System.out.println(names);
15. }
16. }

Currently on executing Test class, [James, diana, Anna] is printed in the output.

Which of the following options can replace /\*INSERT\*/ such that on executing Test class, [Anna, diana, James]  is printed in the output?

* 
  1. Collections.sort(names, new Comparator<String>() {
  2. public int compare(String o1, String o2) {
  3. return o1.compareTo(o2);
  4. }
  5. });
* 
  1. Collections.sort(names, new Comparator<String>() {
  2. public int compare(String o1, String o2) {
  3. return o1.compareToIgnoreCase(o2);
  4. }
  5. });

**(Correct)**

* 
  1. Collections.sort(names, new Comparator<String>() {
  2. public int compare(String o1, String o2) {
  3. return o2.compareTo(o1);
  4. }
  5. });

**(Incorrect)**

* 

**Collections.sort(names);**

**Explanation**

**UKOCP32286:**

If you sort String in ascending order, then upper case letters appear before the lower case letters.

So in this case if I sort the list in ascending order then the output will be [Anna, James, diana] and this is what Collections.sort(names); and o1.compareTo(o2); method calls do.

o2.compareTo(o1); sorts the same list in descending order: [diana, James, Anna] but you have to sort the list such that [Anna, diana, James] is printed in the output,

which means sort the names in ascending order but in case-insensitive manner. String class has compareToIgnoreCase() method for such purpose.

Bottom of Form

Top of Form

Question 88: **Correct**

**Question ID: UKOCP42088**

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

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.Collections;
5. import java.util.Comparator;
6. import java.util.List;
8. class Point {
9. private int x;
10. private int y;
12. public Point(int x, int y) {
13. this.x = x;
14. this.y = y;
15. }
17. @Override
18. public String toString() {
19. return "Point(" + x + ", " + y + ")";
20. }
21. }
23. public class TestPoint {
24. public static void main(String [] args) {
25. List<Point> points = new ArrayList<>();
26. points.add(new Point(4, 5));
27. points.add(new Point(6, 7));
28. points.add(new Point(2, 2));
30. Collections.sort(points, new Comparator<Point>() {
31. @Override
32. public int compare(Point o1, Point o2) {
33. return o1.x - o2.x;
34. }
35. });
36. System.out.println(points);
37. }
38. }

* 

**[Point(2, 2), Point(4, 5), Point(6, 7)]**

* 

**[Point(6, 7), Point(4, 5), Point(2, 2)]**

* 

**[Point(4, 5), Point(6, 7), Point(2, 2)]**

* 

**Compilation error**

**(Correct)**

**Explanation**

**UKOCP42088:**

x and y are private variables and are accessible within the boundary of Point class.

TestPoint class is outside the boundary of Point class and hence o1.x and o2.x give compilation error.

Make sure to check the accessibility before working with the logic.

Bottom of Form

Top of Form

Question 89: **Incorrect**

**Question ID: UKOCP68695**

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

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.Collections;
5. import java.util.Comparator;
6. import java.util.List;
8. class Point {
9. private int x;
10. private int y;
12. public Point(int x, int y) {
13. this.x = x;
14. this.y = y;
15. }
17. public int getX() {
18. return x;
19. }
21. public int getY() {
22. return y;
23. }
25. @Override
26. public String toString() {
27. return "Point(" + x + ", " + y + ")";
28. }
29. }
31. public class Test {
32. public static void main(String [] args) {
33. List<Point> points = new ArrayList<>();
34. points.add(new Point(4, 5));
35. points.add(new Point(6, 7));
36. points.add(new Point(2, 2));
38. Collections.sort(points, new Comparator<Point>() {
39. public int compareTo(Point o1, Point o2) {
40. return o1.getX() - o2.getX();
41. }
42. });
44. System.out.println(points);
45. }
46. }

* 

**[Point(2, 2), Point(4, 5), Point(6, 7)]**

**(Incorrect)**

* 

**[Point(6, 7), Point(4, 5), Point(2, 2)]**

* 

**[Point(4, 5), Point(6, 7), Point(2, 2)]**

* 

**Compilation error**

**(Correct)**

**Explanation**

**UKOCP68695:**

Comparator interface has compare(...) method and not compareTo(...) method.

Anonymous inner class's syntax doesn't implement compare(...) method and thus compilation error.

Make sure to check the accessibility and interface method details before working with the logic.

Bottom of Form

Top of Form

Question 90: **Correct**

**Question ID: UKOCP88283**

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

1. package com.udayankhattry.ocp;
3. import java.util.ArrayList;
4. import java.util.Collections;
5. import java.util.Comparator;
6. import java.util.List;
8. class Point {
9. private int x;
10. private int y;
12. public Point(int x, int y) {
13. this.x = x;
14. this.y = y;
15. }
17. public int getX() {
18. return x;
19. }
21. public int getY() {
22. return y;
23. }
25. @Override
26. public String toString() {
27. return "Point(" + x + ", " + y + ")";
28. }
29. }
31. public class Test {
32. public static void main(String [] args) {
33. List<Point> points = new ArrayList<>();
34. points.add(new Point(4, 5));
35. points.add(new Point(6, 7));
36. points.add(new Point(2, 2));
38. Collections.sort(points, new Comparator<Point>() {
39. public int compare(Point o1, Point o2) {
40. return o2.getX() - o1.getX();
41. }
42. });
44. System.out.println(points);
45. }
46. }

* 

**[Point(2, 2), Point(4, 5), Point(6, 7)]**

* 

**[Point(6, 7), Point(4, 5), Point(2, 2)]**

**(Correct)**

* 

**[Point(4, 5), Point(6, 7), Point(2, 2)]**

* 

**Compilation error**

**Explanation**

**UKOCP88283:**

return o2.getX() - o1.getX(); means the Comparator is sorting the Point objects on descending value of x of Point objects.

To sort the Point objects in ascending order of x, use: return o1.getX() - o2.getX();

To sort the Point objects in ascending order of y, use: return o1.getY() - o2.getY();

To sort the Point objects in descending order of y, use: return o2.getY() - o1.getY();

Bottom of Form

Top of Form

Question 91: **Correct**

**Question ID: UKOCP71854**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.Collections;
5. import java.util.List;
7. public class Test {
8. public static void main(String[] args) {
9. List<String> list = Arrays.asList("M", "R", "A", "P");
10. Collections.sort(list, null);
11. System.out.println(list);
12. }
13. }

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

* 

**[M, R, A, P]**

* 

**[A, M, P, R]**

**(Correct)**

* 

**[R, P, M, A]**

* 

**Runtime Exception**

**Explanation**

**UKOCP71854:**

If null Comparator is passed to sort method, then elements are sorted in natural order (based on Comparable interface implementation).

As list is of String type and String implements Comparable, hence list elements are sorted in ascending order.

Bottom of Form

Top of Form

Question 92: **Correct**

**Question ID: UKOCP88281**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.Comparator;
6. public class Test {
7. public static void main(String[] args) {
8. String [] arr = {"A5", "B4", "C3", "D2", "E1"};
9. Arrays.sort(arr, Comparator.comparing(s -> s.substring(1)));
10. for(String str : arr) {
11. System.out.print(str + " ");
12. }
13. }
14. }

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

* 

**E1 D2 C3 B4 A5**

**(Correct)**

* 

**A5 B4 C3 D2 E1**

* 

**A1 B2 C3 D4 E5**

* 

**E5 D4 C3 B2 A1**

**Explanation**

**UKOCP88281:**

Sorting is working on 2nd letter of the array elements, which means 5, 4, 3, 2, 1. Sorting is in ascending order (1, 2, 3, 4, 5) hence the output is:  E1 D2 C3 B4 A5

Bottom of Form

Top of Form

Question 93: **Incorrect**

**Question ID: UKOCP66033**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.Collections;
5. import java.util.Comparator;
6. import java.util.List;
8. public class Test {
9. public static void main(String[] args) {
10. List<String> emails = Arrays.asList("udayan@outlook.com", "sachin@outlook.com", "sachin@gmail.com",
11. "udayan@gmail.com");
12. Collections.sort(emails, Comparator.comparing(str -> str.substring(str.indexOf("@") + 1)));
13. for(String email : emails) {
14. System.out.println(email);
15. }
16. }
17. }

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

* 

**sachin@gmail.com**

**udayan@gmail.com**

**sachin@outlook.com**

**udayan@outlook.com**

**(Incorrect)**

* 

**sachin@gmail.com**

**udayan@gmail.com**

**udayan@outlook.com**

**sachin@outlook.com**

**(Correct)**

* 

**sachin@outlook.com**

**udayan@outlook.com**

**sachin@gmail.com**

**udayan@gmail.com**

* 

**sachin@outlook.com**

**udayan@outlook.com**

**udayan@gmail.com**

**sachin@gmail.com**

**Explanation**

**UKOCP66033:**

Comparator is comparing on the basis of email domain: gmail.com and outlook.com.

Insertion order is:

udayan@outlook.com

sachin@outlook.com

sachin@gmail.com

udayan@gmail.com

gmail records should appear before outlook records. So sorting order is:

sachin@gmail.com

udayan@gmail.com

udayan@outlook.com

sachin@outlook.com

NOTE: It is not specified, what to do in case email domain is matching. So, for matching email domain, records are left at insertion order.

Bottom of Form

Top of Form

Question 94: **Correct**

**Question ID: UKOCP38822**

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

An example: 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

Given code of SortSwiftCode.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.Collections;
5. import java.util.Comparator;
6. import java.util.List;
8. public class SortSwiftCode {
9. public static void main(String[] args) {
10. List<String> swiftCodes = Arrays.asList("ICICINDD016", "ICICINBBRT4", "BOTKINDD075", "BARBINBB011",
11. "SBBJINDD062", "ABNATHBK865", "BKCHTHBK012");
13. Comparator<String> countryLocationBank = Comparator.comparing(SortSwiftCode::extractCountry)
14. .thenComparing(SortSwiftCode::extractLocation).thenComparing(SortSwiftCode::extractBank);
16. Collections.sort(swiftCodes, countryLocationBank);
17. printCodes(swiftCodes);
19. }
21. private static String extractCountry(String swiftCode) {
22. return swiftCode.substring(4, 6);
23. }
25. private static String extractLocation(String swiftCode) {
26. return swiftCode.substring(6, 8);
27. }
29. private static String extractBank(String swiftCode) {
30. return swiftCode.substring(0, 4);
31. }
33. private static void printCodes(List<String> list) {
34. for (String str : list) {
35. System.out.println(str);
36. }
37. }
38. }

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

* 

**ABNATHBK865**

**BKCHTHBK012**

**BARBINBB011**

**ICICINBBRT4**

**BOTKINDD075**

**ICICINDD016**

**SBBJINDD062**

* 

**BARBINBB011**

**ICICINBBRT4**

**BOTKINDD075**

**ICICINDD016**

**SBBJINDD062**

**ABNATHBK865**

**BKCHTHBK012**

**(Correct)**

* 

**BARBINBB011**

**BOTKINDD075**

**ICICINBBRT4**

**ICICINDD016**

**SBBJINDD062**

**ABNATHBK865**

**BKCHTHBK012**

* 

**None of the other options**

**Explanation**

**UKOCP38822:**

Default thenComparing method helps to chain the Comparators. First the list is sorted on the basis of country code, if matching country code is found then sorted on the basis of location code and if location code matches then list is sorted on bank code.

Bottom of Form

Top of Form

Question 95: **Correct**

**Question ID: UKOCP10036**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.Collections;
5. import java.util.Comparator;
6. import java.util.List;
8. public class Test {
9. public static void main(String[] args) {
10. List<String> list = Arrays.asList("#####", "#", "##", "####", "###");
11. Comparator<String> comp = Comparator.comparing(s -> s);
12. Collections.sort(list, comp.reversed());
13. printCodes(list);
15. }
17. private static void printCodes(List<String> list) {
18. for (String str : list) {
19. System.out.println(str);
20. }
21. }
22. }

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

* 

**#####**

**#**

**##**

**####**

**###**

* 

**###**

**####**

**##**

**#**

**#####**

* 

**#####**

**####**

**###**

**##**

**#**

**(Correct)**

* 

**#**

**##**

**###**

**####**

**#####**

**Explanation**

**UKOCP10036:**

Comparator.comparing(s -> s); compares the passed Strings only. As all the characters in the String are '#', this means strings are sorted on the basis of their lengths. Comparator referred by comp sorts on the basis of strings' lengths. Default reversed() method just reverses the ordering of the Comparator referred by comp, which means sorts the strings in descending order of their lengths.

Bottom of Form

Top of Form

Question 96: **Correct**

**Question ID: UKOCP69193**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.Comparator;
5. import java.util.List;
7. class Person {
8. private String firstName;
9. private String lastName;
11. public Person(String firstName, String lastName) {
12. this.firstName = firstName;
13. this.lastName = lastName;
14. }
16. public String getFirstName() {
17. return firstName;
18. }
20. public String getLastName() {
21. return lastName;
22. }
24. public String toString() {
25. return "{" + firstName + ", " + lastName + "}";
26. }
27. }
29. public class Test {
30. public static void main(String[] args) {
31. List<Person> list = Arrays.asList(
32. new Person("Tom", "Riddle"),
33. new Person("Tom", "Hanks"),
34. new Person("Yusuf", "Pathan"));
35. list.stream().sorted(Comparator.comparing(Person::getFirstName).reversed()
36. .thenComparing(Person::getLastName)).forEach(System.out::println);
37. }
38. }

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

* 

**{Tom, Riddle}**

**{Tom, Hanks}**

**{Yusuf, Pathan}**

* 

**{Tom, Hanks}**

**{Tom, Riddle}**

**{Yusuf, Pathan}**

* 

**{Yusuf, Pathan}**

**{Tom, Riddle}**

**{Tom, Hanks}**

* 

**{Yusuf, Pathan}**

**{Tom, Hanks}**

**{Tom, Riddle}**

**(Correct)**

**Explanation**

**UKOCP69193:**

In this case, sorted method accepts an instance of Comparator<Person> type.

Comparator.comparing(Person::getFirstName) => Returns a Comparator for sorting the records in ascending order of first name.

Comparator.comparing(Person::getFirstName).reversed() => Returns a Comparator for sorting the records in descending order of first name.

Comparator.comparing(Person::getFirstName).reversed().thenComparing(Person::getLastName) => Returns a Comparator for sorting the records in descending order of first name and in case first name matches, then ascending order of last name.

So correct answer is:

{Yusuf, Pathan}

{Tom, Hanks}

{Tom, Riddle}

Bottom of Form

Top of Form

Question 97: **Incorrect**

**Question ID: UKOCP26465**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.util.Arrays;
4. import java.util.Comparator;
5. import java.util.List;
7. class Student implements Comparator<Student> {
8. private String name;
9. private String exam;
11. public Student() {
12. super();
13. }
15. public Student(String name, String exam) {
16. this.name = name;
17. this.exam = exam;
18. }
20. public int compare(Student s1, Student s2) {
21. return s2.name.compareToIgnoreCase(s1.name);
22. }
24. public String toString() {
25. return '{' + name + ", " + exam + '}';
26. }
27. }
29. public class Test {
30. public static void main(String[] args) {
31. Student stud1 = new Student("John", "OCA");
32. Student stud2 = new Student("Jack", "OCP");
33. Student stud3 = new Student("Rob", "OCP");
34. List<Student> list = Arrays.asList(stud1, stud2, stud3);
35. list.sort(new Student());
36. list.forEach(System.out::println);
37. }
38. }

What is the result?

* 

**Runtime exception**

* 

**Compilation error**

**(Incorrect)**

* 

**{Rob, OCP}**

**{John, OCA}**

**{Jack, OCP}**

**(Correct)**

* 

**{Jack, OCP}**

**{John, OCA}**

**{Rob, OCP}**

**Explanation**

**UKOCP26465:**

In real-world programming, you will hardly find a bean class implementing Comparator, but it is a legal code. A bean class generally implements a Comparable interface to define natural ordering.

Student class in this case correctly implements Comparator<Student> interface by overriding compare(Student, Student) method. Note, this compare method will sort in descending order of the Student's name.

list.sort(...) accepts an argument of Comparator<Student> type.  new Student() provides the instance of Comparator<Student> type. It sorts the list in descending order of Students' names.

Output is:

{Rob, OCP}

{John, OCA}

{Jack, OCP}

Bottom of Form

s