**Object-Oriented Approach - 3, Exception Handling - 1 - Results**

Return to review

Chart

Pie chart with 4 slices.

End of interactive chart.

Attempt 3

All knowledge areas

All questions

Top of Form

Question 1: **Correct**

**Question ID: UKOCP34968**

Consider below method declarations/definitions:

1.

void test();

2.

1. default void test(String name) {
2. System.out.println("Testing " + name);
3. }

3.

1. static void test(int x) {
2. System.out.println(x);
3. }

4.

private default void log1() {}

5.

private void log2() {}

6.

private static void log3() {}

How many of the above declarations/definitions can be used inside an interface?

* 

**One declaration/definition**

* 

**Two declaration(s)/definition(s)**

* 

**Three declaration(s)/definition(s)**

* 

**Four declaration(s)/definition(s)**

* 

**Five declaration(s)/definition(s)**

**(Correct)**

* 

**All six declaration(s)/definition(s)**

**Explanation**

**UKOCP34968:**

Let's check all the options one by one:

void test();

✓  By default test method is public and abstract and it is allowed inside an interface.

1. default void test(String name) {
2. System.out.println("Testing " + name);
3. }

✓  As per Java 8, default methods were added in the interface. It is a valid syntax for the default method to be used inside an interface.

1. static void test(int x) {
2. System.out.println(x);
3. }

✓  As per Java 8, static methods were added in the interface. It is a valid syntax for the static method to be used inside an interface.

Even if all the above 3 methods [test(), test(String) and test(int)] are available in the same interface, there is no issue at all, it is the case of method overloading.

private default void log1() {}

✗  As per Java 9, private methods were added in the interface, these can be static or non-static but not default.

private void log2() {}

✓  As per Java 9, private methods were added in the interface, these can be static or non-static.

private static void log3() {}

✓  As per Java 9, private methods were added in the interface, these can be static or non-static.

Out of 6 declarations/definitions, 5 can be used inside an interface.

Bottom of Form

Top of Form

Question 2: **Correct**

**Question ID: UKOCP29136**

Consider below code fragment:

1. interface Printable {
2. public void setMargin();
3. public void setOrientation();
4. }
6. abstract class Paper implements Printable { //Line 7
7. public void setMargin() {}
8. //Line 9
9. }
11. class NewsPaper extends Paper { //Line 12
12. public void setMargin() {}
13. //Line 14
14. }

Above code currently causes compilation error. Which 2 modifications, done independently, enable the code to compile?

* 

**Replace the code at Line 7 with: class Paper implements Printable {**

* 

**Insert at Line 9: public abstract void setOrientation();**

* 

**Replace the code at Line 12 with: abstract class NewsPaper extends Paper {**

**(Correct)**

* 

**Insert at Line 14: public void setOrientation() {}**

**(Correct)**

**Explanation**

**UKOCP29136:**

First you should find out the reason for compilation error. Methods declared in Printable interface are implicitly abstract, no issues with Printable interface.

class Paper is declared abstract and it implements Printable interface, it overrides setMargin() method but setOrientation() method is still abstract. No issues with class Paper as it is an abstract class and can have 0 or more abstract methods.

class NewsPaper is concrete class and it extends Paper class (which is abstract). So class NewsPaper must override setOrientation() method OR it must be declared abstract.

Hence, there are 2 possible solution:

Insert at Line 14: public void setOrientation() {}

OR

Replace the code at Line 12 with: abstract class NewsPaper extends Paper {

Replacing Line 9 with 'public abstract void setOrientation();' is not necessary and it will not resolve the compilation error in NewsPaper class.

Replacing Line 7 with 'class Paper implements Printable {' will cause compilation failure of Paper class as it inherits abstract method 'setOrientation'.

Bottom of Form

Top of Form

Question 3: **Correct**

**Question ID: UKOCP53245**

Consider below codes of 3 java files:

1. //Profitable1.java
2. package com.udayankhattry.ocp;
4. public interface Profitable1 {
5. default double profit() {
6. return 12.5;
7. }
8. }
9. //Profitable2.java
10. package com.udayankhattry.ocp;
12. public interface Profitable2 {
13. default double profit() {
14. return 25.5;
15. }
16. }
17. //Profit.java
18. package com.udayankhattry.ocp;
20. public abstract class Profit implements Profitable1, Profitable2 {
21. /\*INSERT\*/
22. }

Which of the following needs to be done so that there is no compilation error?

* 

**No need for any modifications, code compiles as is**

* 

**Replace /\*INSERT\*/ with below code:**

* 1. double profit() {
  2. return 50.0;
  3. }
* 

**Replace /\*INSERT\*/ with below code:**

* 1. public default double profit() {
  2. return 50.0;
  3. }
* 

**Replace /\*INSERT\*/ with below code:**

* 1. protected double profit() {
  2. return 50.0;
  3. }
* 

**Replace /\*INSERT\*/ with below code:**

* 1. public double profit() {
  2. return Profitable1.profit();
  3. }
* 

**Replace /\*INSERT\*/ with below code:**

* 1. public double profit() {
  2. return Profitable2.super.profit();
  3. }

**(Correct)**

**Explanation**

**UKOCP53245:**

Profit class causes compilation error as it complains about duplicate default methods: Profitable1.profit() and Profitable2.profit(). To rectify this error abstract class Profit must override the profit() method.

default keyword for method is allowed only inside the interface and default methods are implicitly public. So overriding method should use public modifier and shouldn't use default keyword.

If you want to invoke the default method implementation from the overriding method, then the correct syntax is: [Interface\_name].super.[default\_method\_name].

Hence, `Profitable1.super.profit();` will invoke the default method of Profitable1 interface and `Profitable2.super.profit();` will invoke the default method of Profitable2 interface.

Based on above points, let's check all the options one by one:

No need for any modifications, code compiles as is: ✗

Replace /\*INSERT\*/ with below code:

1. double profit() {
2. return 50.0;
3. }

✗  profit() method must be declared with public access modifier.

Replace /\*INSERT\*/ with below code:

1. public default double profit() {
2. return 50.0;
3. }

✗  default keyword for method is allowed only inside the interface.

Replace /\*INSERT\*/ with below code:

1. protected double profit() {
2. return 50.0;
3. }

✗  profit() method must be declared with public access modifier.

Replace /\*INSERT\*/ with below code:

1. public double profit() {
2. return Profitable1.profit();
3. }

✗  Profitable1.profit(); causes compilation error as correct syntax is: Profitable1.super.profit();

Replace /\*INSERT\*/ with below code:

1. public double profit() {
2. return Profitable2.super.profit();
3. }

✓  It compiles successfully.

Bottom of Form

Top of Form

Question 4: **Correct**

**Question ID: UKOCP51067**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. interface Perishable1 {
4. default int maxDays() {
5. return 1;
6. }
7. }
9. interface Perishable2 extends Perishable1 {
10. default int maxDays() {
11. return 2;
12. }
13. }
15. class Milk implements Perishable2, Perishable1 {}
17. public class Test {
18. public static void main(String[] args) {
19. Perishable1 obj = new Milk();
20. System.out.println(obj.maxDays());
21. }
22. }

Which of the following statements is correct?

* 

**Interface Perishable2 causes compilation error**

* 

**Class Milk causes compilation error**

* 

**Class Test causes compilation error**

* 

**Given code compiles successfully and on execution Test class prints 1 on to the console**

* 

**Given code compiles successfully and on execution Test class prints 2 on to the console**

**(Correct)**

**Explanation**

**UKOCP51067:**

As per Java 8, default methods were added in the interface.

Interface Perishable1 defines default method maxDays(), there is no compilation error in interface Perishable1. Method maxDays() is implicitly public in Perishable1.

Interface Perishable2 extends Perishable1 and it overrides the default method maxDays() of Perishable1, overriding method in Perishable2 is implicitly public. Interface Perishable2 compiles successfully.

Class Milk implements Perishable2 and Perishable1. Although it is redundant for Milk class to implement Preishable1 as Perishable2 already extends Perishable1.

There is no conflict in Milk class as it inherits the default method maxDays() of Perishable2 interface. Milk class compiles successfully.

`Perishable1 obj = new Milk();` It compiles fine as Perishable1 is supertype and Milk is subtype.

`obj.maxDays()` executes the default maxDays() method of Perishable2 interface and it returns 2. `System.out.println(obj.maxDays());` prints 2 on to the console.

Bottom of Form

Top of Form

Question 5: **Correct**

**Question ID: UKOCP78544**

Consider below codes of 4 java files:

1. //Moveable.java
2. package com.udayankhattry.ocp;
4. public interface Moveable {
5. void move();
6. }
7. //Animal.java
8. package com.udayankhattry.ocp;
10. public abstract class Animal {
11. void move() {
12. System.out.println("ANIMAL MOVING");
13. }
14. }
15. //Dog.java
16. package com.udayankhattry.ocp;
18. public class Dog extends Animal implements Moveable {}
19. //Test.java
20. package com.udayankhattry.ocp;
22. public class Test {
23. public static void main(String[] args) {
24. Moveable moveable = new Dog();
25. moveable.move();
26. }
27. }

Which of the following statements is correct?

* 

**There is a compilation error in Animal.java file**

* 

**There is a compilation error in Dog.java file**

**(Correct)**

* 

**There is a compilation error in Test.java file**

* 

**There is no compilation error and on execution, Test class prints ANIMAL MOVING on to the console**

**Explanation**

**UKOCP78544:**

Method move() declared in Moveable interface is implicitly public and abstract.

Abstract class Animal has non-abstract method move() and it is declared with no modifier (package scope). Abstract class in java can have 0 or more abstract methods. Hence Animal class compiles successfully.

class Dog extends Animal and as both the classes Animal and Dog are within the same package 'com.udayankhattry.ocp', Dog inherits the move() method defined in Animal class.

Dog class implements Moveable interface as well, therefore it must implement public move() method as well. But as inherited move() method from Animal class is not public, therefore Dog class fails to compile.

Bottom of Form

Top of Form

Question 6: **Correct**

**Question ID: UKOCP10517**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. interface M {
4. public static void log() {
5. System.out.println("M");
6. }
7. }
9. abstract class A {
10. public static void log() {
11. System.out.println("N");
12. }
13. }
15. class MyClass extends A implements M {}
17. public class Test {
18. public static void main(String[] args) {
19. M obj1 = new MyClass();
20. obj1.log(); //Line n1
22. A obj2 = new MyClass();
23. obj2.log(); //Line n2
25. MyClass obj3 = new MyClass();
26. obj3.log(); //Line n3
27. }
28. }

Which of the following statements is correct?

* 

**There is a compilation error in interface M**

* 

**There is a compilation error in class A**

* 

**Line n1 causes compilation error**

**(Correct)**

* 

**Line n2 causes compilation error**

* 

**Line n3 causes compilation error**

* 

**Given code compiles successfully**

**Explanation**

**UKOCP10517:**

As per Java 8, default and static methods were added in the interface. Interface M defines static method log(), there is no compilation error in interface M.

Also the scope of static log() method of M is limited to interface M and it can be invoked by using Interface name only, M.log().

Abstract class A also defines the static log() method. Abstract class can have 0 or more abstract methods. Hence, no compilation error in class A as well.

Super type reference variable can refer to an instance of Sub type, therefore the statement `M obj1 = new MyClass();` compiles successfully.

obj1 is of M type, hence `obj1.log();` tries to tag the static method of M but static log() method of M can only be invoked by using M.log();.

Therefore, Line n1 causes compilation error.

Scope of static log() method of A is not limited to class A only but MyClass also gets A.log() method in its scope.

There are different ways in which static method of an abstract class can be accessed:

1. By using the name of the abstract class: A.log(); //Preferred way

2. By using the reference variable of abstract class: A o1 = null; o1.log();

3. By using the name of the subclass: MyClass.log();

4. By using the reference variable of the subclass: MyClass o2 = null; o2.log();

Hence, Line n2 and Line n3 compile successfully.

Bottom of Form

Top of Form

Question 7: **Correct**

**Question ID: UKOCP11849**

Consider below code snippet:

1. interface ILog {
2. default void log() {
3. System.out.println("ILog");
4. }
5. }
7. abstract class Log {
8. public static void log() {
9. System.out.println("Log");
10. }
11. }
13. class MyLogger extends Log implements ILog {}

Which of the following statements is correct?

* 

**There is no compilation error in the above code**

* 

**There is a compilation error in interface ILog**

* 

**There is a compilation error in abstract class Log**

* 

**There is a compilation error in MyLogger class**

**(Correct)**

**Explanation**

**UKOCP11849:**

As per Java 8, default and static methods were added in the interface. Interface ILog defines default method log(), there is no compilation error in interface ILog.

Abstract class Log defines the static log() method. Abstract class can have 0 or more abstract methods. Hence, no compilation error in class Log as well.

Default methods of an interface are implicitly public and are inherited by the implementer class. Class MyLogger implements ILog interface and therefore it inherits the default log() method of ILog interface.

Also, the scope of static log() method of abstract class Log is not limited to class Log only but MyLogger also gets Log.log() method in its scope.

So, MyLogger class has instance method log() [inherited from ILog interface] and static method log() [from Log class] and this causes conflict. Static and non-static methods with same signature are not allowed in one scope, therefore class MyLogger fails to compile.

Bottom of Form

Top of Form

Question 8: **Correct**

**Question ID: UKOCP44772**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. interface Rideable {
4. void ride(String name);
5. }
7. class Animal {}
9. class Horse extends Animal implements Rideable {
10. public void ride(String name) {
11. System.out.println(name.toUpperCase() + " IS RIDING THE HORSE");
12. }
13. }
15. public class Test {
16. public static void main(String[] args) {
17. Animal horse = new Horse();
18. /\*INSERT\*/
19. }
20. }

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

Select 4 options.

* 

**horse.ride("EMMA");**

* 

**(Horse)horse.ride("EMMA");**

* 

**((Horse)horse).ride("Emma");**

**(Correct)**

* 

**(Rideable)horse.ride("emma");**

* 

**((Rideable)horse).ride("emma");**

**(Correct)**

* 

**(Rideable)(Horse)horse.ride("EMMA");**

* 

**(Horse)(Rideable)horse.ride("EMMA");**

* 

**((Rideable)(Horse)horse).ride("EMMA");**

**(Correct)**

* 

**((Horse)(Rideable)horse).ride("emma");**

**(Correct)**

**Explanation**

**UKOCP44772:**

Let's check all the options one by one:

horse.ride("EMMA"); ✗

Variable 'horse' is of Animal type and ride(String) method is not defined in Animal class, therefore it causes compilation error.

(Horse)horse.ride("EMMA"); ✗

horse.ride("EMMA") will be evaluated first as dot (.) operator has higher precedence than cast. Variable 'horse' is of Animal type and ride(String) method is not defined in Animal class, therefore it causes compilation error.

((Horse)horse).ride("Emma"); ✓

Variable 'horse' is of Animal type, and it is casted to the Horse type (subtype of Animal). Horse class has ride(String) method, therefore no compilation error.

At runtime, ride(String) method of Horse class will get invoked and will print the expected output. As, name.toUpperCase() method is invoked, it doesn't matter in what case you pass the name, in the output name will always be displayed in the upper case.

(Rideable)horse.ride("emma"); ✗

horse.ride("EMMA") will be evaluated first as dot (.) operator has higher precedence than cast. Variable 'horse' is of Animal type and ride(String) method is not defined in Animal class, therefore it causes compilation error.

((Rideable)horse).ride("emma"); ✓

Variable 'horse' is of Animal type, and it can be casted to any interface type, though in this case it is casted to Rideable type (super type of Horse). Rideable interface has ride(String) method, therefore no compilation error.

At runtime, ride(String) method of Horse class will get invoked at runtime and will print the expected output.

(Rideable)(Horse)horse.ride("EMMA"); ✗

horse.ride("EMMA") will be evaluated first as dot (.) operator has higher precedence than cast. Variable 'horse' is of Animal type and ride(String) method is not defined in Animal class, therefore it causes compilation error.

(Horse)(Rideable)horse.ride("EMMA"); ✗

horse.ride("EMMA") will be evaluated first as dot (.) operator has higher precedence than cast. Variable 'horse' is of Animal type and ride(String) method is not defined in Animal class, therefore it causes compilation error.

((Rideable)(Horse)horse).ride("EMMA"); ✓

Variable 'horse' is of Animal type, it is first casted to the Horse type (subtype of Animal) and then casted to Rideable type (interface). Rideable interface has ride(String) method, therefore no compilation error.

At runtime, ride(String) method of Horse class will get invoked and will print the expected output.

((Horse)(Rideable)horse).ride("emma"); ✓

Variable 'horse' is of Animal type, and it can be casted to any interface type, though in this case it is casted to Rideable type (super type of Horse). Then it is casted to Horse type (subtype of Rideable).

Horse class has ride(String) method, therefore no compilation error.

At runtime, ride(String) method of Horse class will get invoked and will print the expected output.

Bottom of Form

Top of Form

Question 9: **Correct**

**Question ID: UKOCP44776**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. interface X1 {
4. default void print() {
5. System.out.println("X1");
6. }
7. }
9. interface X2 extends X1 {
10. void print();
11. }
13. interface X3 extends X2 {
14. default void print() {
15. System.out.println("X3");
16. }
17. }
19. class X implements X3 {}
21. public class Test {
22. public static void main(String[] args) {
23. X1 obj = new X();
24. obj.print();
25. }
26. }

Which of the following statements is correct?

* 

**interface X1 fails to compile**

* 

**interface X2 fails to compile**

* 

**interface X3 fails to compile**

* 

**class X fails to compile**

* 

**class Test fails to compile**

* 

**class Test compiles successfully and on execution prints X1 on to the console**

* 

**class Test compiles successfully and on execution prints X3 on to the console**

**(Correct)**

**Explanation**

**UKOCP44776:**

As per Java 8, default methods were added in the interface.

Interface X1 defines default method print(), there is no compilation error in interface X1. Method print() is implicitly public in X1.

Interface X2 extends X1 and it overrides the default method print() of X1, overriding method in X2 is implicitly abstract and public. An interface in java can override the default method of super type with abstract modifier. Interface X2 compiles successfully.

Interface X3 extends X2 and it implements the abstract method print() of X2, overriding method in X3 is default and implicitly public. An interface in java can implement the abstract method of super type with default modifier. Interface X3 compiles successfully.

Class X implements X3 and therefore it inherits the default method print() defined in interface X3.

`X1 obj = new X();` compiles successfully as X1 is of super type (X implements X3, X3 extends X2 and X2 extends X1).

`obj.print();` invokes the default method print() defined in interface X3 and hence X3 is printed on to the console.

Bottom of Form

Top of Form

Question 10: **Correct**

**Question ID: UKOCP10592**

Which of these access modifiers can be used for a top level interface?

* 

**private**

* 

**protected**

* 

**public**

**(Correct)**

* 

**All of the other options**

**Explanation**

**UKOCP10592:**

A top level interface can be declared with either public or default modifiers.

public interface is accessible across all packages but interface declared with default modifier and be accessed in the defining package only.

Bottom of Form

Top of Form

Question 11: **Correct**

**Question ID: UKOCP43018**

Given the following definitions of the class Insect and the interface Flyable, the task is to declare a class Mosquito that inherits from the class Insect and implements the interface Flyable.

1. class Insect {}
2. interface Flyable {}

Select the correct option to accomplish this task:

* 

**class Mosquito implements Insect extends Flyable{}**

* 

**class Mosquito implements Insect, Flyable{}**

* 

**class Mosquito extends Insect, Flyable{}**

* 

**class Mosquito extends Insect implements Flyable{}**

**(Correct)**

**Explanation**

**UKOCP43018:**

A class in Java extends from another class and implements interface(s). Hence correct syntax is:

class Mosquito extends Insect implements Flyable{}

Bottom of Form

Top of Form

Question 12: **Correct**

**Question ID: UKOCP66596**

For the given code of Implementer.java file:

1. package com.udayankhattry.ocp;
3. interface I01 {
4. void m1();
5. }
7. public class Implementer extends Object implements I01 {
8. protected void m1() {
10. }
11. }

Which of the following statements is true?

* 

**Interface I01 causes compilation error as method m1 is not public**

* 

**Implementer class declaration is not correct**

* 

**Method m1() in Implementer class is not implemented correctly**

**(Correct)**

* 

**None of the other options**

**Explanation**

**UKOCP66596:**

void m1(); in interface I01 is equivalent to `public abstract void m1();`. So method m1() is implicitly public and abstract.

In java,  a class can extend from only one class but can implement multiple interfaces. Correct keywords are: extends and implements. So, class declaration is correct.

As method m1() is implicitly public in I01, hence overriding method in Implementer class should also be public. But it is protected and hence compiler complains.

Bottom of Form

Top of Form

Question 13: **Correct**

**Question ID: UKOCP66115**

Below are the possible definitions of Printable interface:

1.

public abstract interface Printable {}

2.

1. public interface Printable {
2. protected void print();
3. }

3.

1. public interface Printable {
2. int i;
3. void print();
4. }

4.

1. public interface Printable {
2. void print();
4. default void log() {
5. }
7. private default void log1() {
8. }
10. private static void log2() {
11. }
13. static int getNumLogs() {
14. return -1;
15. }
16. }

5.

1. @MarkerInterface
2. public interface Printable {}

6.

1. public interface Printable {
2. public static final int x = 10;
3. public final int y = 20;
4. int z = 30;
5. }

How many definitions are valid?

* 

**None of the definitions is valid**

* 

**Only 1 definition is valid**

* 

**2 definitions are valid**

**(Correct)**

* 

**3 definitions are valid**

* 

**4 definitions are valid**

* 

**5 definitions are valid**

* 

**All 6 definitions are valid**

**Explanation**

**UKOCP66115:**

public abstract interface Printable {}

✓  Valid, as interface in java is implicitly abstract, so using abstract keyword doesn't cause any error.

1. public interface Printable {
2. protected void print();
3. }

✗  abstract method of the interface are implicitly public and if you provide access modifier for the abstract method of the interface, then only 'public' is allowed. As 'protected' is used for print() method, hence it causes compilation error.

1. public interface Printable {
2. int i;
3. void print();
4. }

✗  Variables declared inside interface are implicitly public, static and final and therefore compiler complains about un-initialized final variable i.

1. public interface Printable {
2. void print();
4. default void log() {
5. }
7. private default void log1() {
8. }
10. private static void log2() {
11. }
13. static int getNumLogs() {
14. return -1;
15. }
16. }

✗  As per Java 8, default and static methods were added in the interface and as per Java 9, private methods were added in the interface.

default modifier is not allowed with private method of the interface, hence method log1() causes compilation error. Methods print(), log(), log2() and getNumLogs() compile successfully.

1. @MarkerInterface
2. public interface Printable {}

✗  @MarkerInterface annotation is not available in Java and hence it causes compilation error.

1. public interface Printable {
2. public static final int x = 10;
3. public final int y = 20;
4. int z = 30;
5. }

✓  Interfaces can define public, static and final variables and these modifiers are implicit.

Hence, for 'y' compiler adds static modifier and for 'z' compiler adds public, static and final modifiers.

Therefore, only 2 interface definitions are valid.

Bottom of Form

Top of Form

Question 14: **Correct**

**Question ID: UKOCP10117**

Consider below codes of 3 java files:

1. //Sellable.java
2. package com.udayankhattry.ocp;
4. public interface Sellable {
5. double getPrice();
7. default String symbol() {
8. return "$";
9. }
10. }
11. //Chair.java
12. package com.udayankhattry.ocp;
14. public class Chair implements Sellable {
15. public double getPrice() {
16. return 35;
17. }
19. public String symbol() {
20. return "£";
21. }
22. }
23. //Test.java
24. package com.udayankhattry.ocp;
26. public class Test {
27. public static void main(String[] args) {
28. Sellable obj = new Chair(); //Line n1
29. System.out.println(obj.symbol() + obj.getPrice()); //Line n2
30. }
31. }

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

* 

**Compilation error in Chair class**

* 

**Compilation error in Test class**

* 

**It compiles successfully and on execution prints $35 on to the console**

* 

**It compiles successfully and on execution prints $35.0 on to the console**

* 

**It compiles successfully and on execution prints $35.00 on to the console**

* 

**It compiles successfully and on execution prints £35 on to the console**

* 

**It compiles successfully and on execution prints £35.0 on to the console**

**(Correct)**

* 

**It compiles successfully and on execution prints £35.00 on to the console**

**Explanation**

**UKOCP10117:**

default methods were added in Java 8. Class Chair correctly implements Sellable interface and it also overrides the default symbol() method of Sellable interface.

At Line n1, 'obj' refers to an instance of Chair class, so obj.symbol() and obj.getPrice() invoke the overriding methods of Chair class only.

obj.symbol() returns "£" and obj.getPrice() returns 35.0

At Line n2, '+' operator behaves as concatenation operator and Line n2 prints £35.0 on to the console.

Bottom of Form

Top of Form

Question 15: **Correct**

**Question ID: UKOCP79419**

Consider below codes of 2 java files:

1. //Flyable.java
2. package com.udayankhattry.ocp;
4. public interface Flyable {
5. static int horizontalDegree() { //Line n1
6. return 20;
7. }
9. default void fly() {
10. System.out.println("Flying at " + horizontalDegree() + " degrees."); //Line n2
11. }
13. void land();
14. }
15. //Aeroplane.java
16. package com.udayankhattry.ocp;
18. public class Aeroplane implements Flyable {
19. public void land() {
20. System.out.println("Landing at " + -Flyable.horizontalDegree() + " degrees."); //Line n3
21. }
23. public static void main(String[] args) {
24. new Aeroplane().fly();
25. new Aeroplane().land();
26. }
27. }

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

* 

**Compilation error at Line n1**

* 

**Compilation error at Line n2**

* 

**Compilation error at Line n3**

* 

**Given code compiles successfully and on execution prints below in the output: Flying at 20 degrees. Landing at -20 degrees.**

**(Correct)**

**Explanation**

**UKOCP79419:**

As per Java 8, default and static methods were added in the interface and default methods can invoke static method as well. Hence, there is no issue with the Flyable interface.

Class Aeroplane implements Flyable interface, hence it inherits the default method fly() and static method horizontalDegree() can be accessed using Flyable.horizontalDegree(). Class Aeroplane also provides the implementation of land() method. There is no issue with Aeroplane class as well.

On execution below text is printed on to the console:

Flying at 20 degrees.

Landing at -20 degrees.

Bottom of Form

Top of Form

Question 16: **Correct**

**Question ID: UKOCP43491**

Consider below codes of 3 java files:

1. //Super.java
2. package com.udayankhattry.ocp;
4. public interface Super {
5. String name = "SUPER"; //Line n1
6. }
7. //Sub.java
8. package com.udayankhattry.ocp;
10. public interface Sub extends Super { //Line n2
12. }
13. //Test.java
14. package com.udayankhattry.ocp;
16. public class Test {
17. public static void main(String[] args) {
18. Sub sub = null;
19. System.out.println(sub.name); //Line n3
20. }
21. }

Which of the following statements is correct?

* 

**Line n1 causes compilation error**

* 

**Line n2 causes compilation error**

* 

**Line n3 causes compilation error**

* 

**Line n3 throws an exception at runtime**

* 

**Test class compiles successfully and on execution prints SUPER on to the console**

**(Correct)**

**Explanation**

**UKOCP43491:**

Variable 'name' declared inside interface Super is implicitly public, static and final. Line n1 compiles successfully.

In Java, a class can extend from only one class but an interface can extend from multiple interfaces. Line n2 compiles successfully.

Variable 'name' can be accessed in 2 ways: Super.name and Sub.name.

Though correct way to refer static variable is by using the type name, such as Sub.name but it can also be invoked by using Sub reference variable. Hence, sub.name at Line n3 correctly points to the name variable at Line n1.

For invoking static fields, object is not needed, therefore even if sub refers to null, sub.name doesn't throw NullPoionterException.

Test class compiles successfully and on execution prints SUPER on to the console.

Bottom of Form

Top of Form

Question 17: **Correct**

**Question ID: UKOCP32368**

Consider below codes of 4 java files:

1. //I1.java
2. package com.udayankhattry.ocp;
4. public interface I1 {
5. int i = 10;
6. }
7. //I2.java
8. package com.udayankhattry.ocp;
10. public interface I2 {
11. int i = 20;
12. }
13. //I3.java
14. package com.udayankhattry.ocp;
16. public interface I3 extends I1, I2 { //Line n1
18. }
19. //Test.java
20. package com.udayankhattry.ocp;
22. public class Test {
23. public static void main(String[] args) {
24. System.out.println(I1.i); //Line n2
25. System.out.println(I2.i); //Line n3
26. System.out.println(I3.i); //Line n4
27. }
28. }

Which of the following statements is correct?

* 

**Line n1 causes compilation error**

* 

**Line n2 causes compilation error**

* 

**Line n3 causes compilation error**

* 

**Line n4 causes compilation error**

**(Correct)**

* 

**There is no compilation error**

**Explanation**

**UKOCP32368:**

Variable 'i' declared inside interface I1 is implicitly public, static and final. Similarly, variable i declared inside interface I2 is implicitly public, static and final as well.

In Java, a class can extend from only one class but an interface can extend from multiple interfaces. static variables are not inherited and hence there is no issue with Line n1.

I1.i points to variable 'i' of interface I1.

I2.i points to variable 'i' of interface I2.

I3.i is an ambiguous call, as compiler is not sure whether to point to I1.i or I2.i and therefore, Line n4 causes compilation error.

Bottom of Form

Top of Form

Question 18: **Correct**

**Question ID: UKOCP71932**

Consider below codes of 2 java files:

1. //Counter.java
2. package com.udayankhattry.ocp;
4. public interface Counter {
5. int count = 10; //Line n1
6. }
7. //Test.java
8. package com.udayankhattry.ocp;
10. public class Test {
11. public static void main(String[] args) {
12. Counter [] arr = new Counter[2]; //Line n2
13. for(Counter ctr : arr) {
14. System.out.print(ctr.count); //Line n3
15. }
16. }
17. }

Which of the following statements is correct?

* 

**Only Line n1 causes compilation error**

* 

**Only Line n2 causes compilation error**

* 

**Line n1 and Line n2 cause compilation error**

* 

**Only Line n3 causes compilation error**

* 

**Line n3 throws an exception at runtime**

* 

**Test class compiles successfully and on execution prints 1010 on to the console**

**(Correct)**

**Explanation**

**UKOCP71932:**

Variable 'count' declared inside interface Counter is implicitly public, static and final. Line n1 compiles successfully.

Line n2 creates one dimensional array of 2 elements of Counter type and both the elements are initialized to null. Line n2 compiles successfully.

Though correct way to refer static variable is by using the type name, such as Counter.count but it can also be invoked by using reference variable of Counter type. Hence ctr.count at Line n3 correctly points to the count variable at Line n1.

For invoking static fields, object is not needed, therefore even if 'ctr' refers to null, ctr.count doesn't throw NullPoionterException. Given loop executes twice and therefore output is: 1010

Bottom of Form

Top of Form

Question 19: **Correct**

**Question ID: UKOCP38900**

Consider below codes of 2 java files:

1. //GetSetGo.java
2. package com.udayankhattry.ocp;
4. public interface GetSetGo {
5. int count = 1; //Line n1
6. }
7. //Test.java
8. package com.udayankhattry.ocp;
10. public class Test {
11. public static void main(String[] args) {
12. GetSetGo [] arr = new GetSetGo[5]; //Line n2
13. for(GetSetGo obj : arr) {
14. obj.count++; //Line n3
15. }
16. System.out.println(GetSetGo.count); //Line n4
17. }
18. }

Which of the following statements is correct?

* 

**Line n1 causes compilation error**

* 

**Line n2 causes compilation error**

* 

**Line n3 causes compilation error**

**(Correct)**

* 

**Line n4 causes compilation error**

* 

**Test class compiles successfully and on execution prints 5 on to the console**

* 

**Test class compiles successfully and on execution prints 6 on to the console**

**Explanation**

**UKOCP38900:**

Variable 'count' declared inside interface GetSetGo is implicitly public, static and final. Line n1 compiles successfully.

Line n2 creates one dimensional array of 5 elements of GetSetGo type and all 5 elements are initialized to null. Line n2 compiles successfully.

Though correct way to refer static variable is by using the type name, such as GetSetGo.count but it can also be invoked by using reference variable of GetSetGo type. Hence, obj.count at Line n3 correctly points to the count variable at Line n1. But as variable 'count' is implicitly final, therefore obj.count++ causes compilation error. Line n3 fails to compile.

Line n4 compiles successfully as variable 'count' is implicitly static and GetSetGo.count is the correct syntax to point to 'count' variable of interface GetSetGo.

Bottom of Form

Top of Form

Question 20: **Correct**

**Question ID: UKOCP55426**

Consider below codes of 3 java files:

1. //Shrinkable.java
2. package com.udayankhattry.ocp;
4. public interface Shrinkable {
5. public static void shrinkPercentage() {
6. System.out.println("80%");
7. }
8. }
9. //AntMan.java
10. package com.udayankhattry.ocp;
12. public class AntMan implements Shrinkable { }
13. //Test.java
14. package com.udayankhattry.ocp;
16. public class Test {
17. public static void main(String[] args) {
18. AntMan.shrinkPercentage();
19. }
20. }

Which of the following statements is correct?

* 

**There is a compilation error in Shrinkable.java file**

* 

**There is a compilation error in AntMan.java file**

* 

**There is a compilation error in Test.java file**

**(Correct)**

* 

**There is no compilation error and on execution, Test class prints 80% on to the console**

**Explanation**

**UKOCP55426:**

As per Java 8, default and static methods were added in the interface and as per Java 9, private methods were added in the interface. There is no issue in Shrinkable.java file.

class AntMan implements Shrinkable interface but as there is no abstract method in Shrinkable interface, hence AntMan class is not needed to implement any method. AntMan.java file compiles successfully.

static method of Shrinkable interface can only be accessed by using Shrinkable.shrinkPercentage(). `AntMan.shrinkPercentage();` causes compilation error.

Bottom of Form

Top of Form

Question 21: **Correct**

**Question ID: UKOCP35818**

Consider below codes of 3 java files:

1. //Buyable.java
2. package com.udayankhattry.ocp;
4. public interface Buyable {
5. int salePercentage = 85;
7. public static String salePercentage() {
8. return salePercentage + "%";
9. }
10. }
11. //Book.java
12. package com.udayankhattry.ocp;
14. public class Book implements Buyable {}
15. //Test.java
16. package com.udayankhattry.ocp;
18. public class Test {
19. public static void main(String[] args) {
20. Buyable [] arr = new Buyable[2];
21. for(Buyable b : arr) {
22. System.out.println(b.salePercentage); //Line n1
23. System.out.println(b.salePercentage()); //Line n2
24. }
26. Book [] books = new Book[2];
27. for(Book b : books) {
28. System.out.println(b.salePercentage); //Line n3
29. System.out.println(b.salePercentage()); //Line n4
30. }
31. }
32. }

Which of the following statements are correct?

Select 2 options.

* 

**There is a compilation error in Buyable.java file**

* 

**There is a compilation error in Book.java file**

* 

**There is a compilation error at Line n1**

* 

**There is a compilation error at Line n2**

**(Correct)**

* 

**There is a compilation error at Line n3**

* 

**There is a compilation error at Line n4**

**(Correct)**

**Explanation**

**UKOCP35818:**

Variable 'salePercentage' declared inside interface Buyable is implicitly public, static and final. As per Java 8, default and static methods were added in the interface and as per Java 9, private methods were added in the interface. There is no compilation error in Buyable.java file.

class Book implements Buyable interface but as there is no abstract method in Buyable interface, hence Book class is not needed to implement any method. Book.java file compiles successfully.

`Buyable [] arr = new Buyable[2];` creates one dimensional array of 2 elements of Buyable type and both the elements are initialized to null.

There are some difference in which static variables and static methods of the interface are accessed.

Correct and only way to access static method of an Interface is by using the name of the interface, such as Buyable.salePercentage(). Line n2 and Line n4 cause compilation error.

As far as public static final variable of interface is concerned, even through the correct way to access static variable is by using the name of the interface, such as Buyable.salePercentage but it can also be accessed by using following:

Reference variable of the interface: Buyable obj1 = null; System.out.println(obj1.salePercentage);

Name of the implementer class: System.out.println(Book.salePercentage);

Reference variable of the implementer class: Book obj2 = null; System.out.println(obj2.salePercentage);

Hence, Line n1 and Line n3 compile successfully.

Bottom of Form

Top of Form

Question 22: **Correct**

**Question ID: UKOCP53249**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. interface Profitable {
4. double profitPercentage = 42.0;
5. }
7. class Business implements Profitable {
8. double profitPercentage = 50.0; //Line n1
9. }
11. public class Test {
12. public static void main(String[] args) {
13. Profitable obj = new Business(); //Line n2
14. System.out.println(obj.profitPercentage); //Line n3
15. }
16. }

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

* 

**Line n1 causes compilation error**

* 

**Line n2 causes compilation error**

* 

**Line n3 causes compilation error**

* 

**Test class compiles successfully and on execution prints 42.0 on to the console**

**(Correct)**

* 

**Test class compiles successfully and on execution prints 50.0 on to the console**

**Explanation**

**UKOCP53249:**

'profitPercentage' variable of Profitable interface is implicitly public, static and final.

Line n1 defines the instance variable 'profitPercentage' of Business class. There is no error at Line n1.

Super type reference variable can refer to an instance of Sub type, therefore no issues at Line n2 as well.

Even though correct syntax for accessing interface variable is by using Interface name, such as Profitable.profitPercentage but reference variable also works. obj.profitPercentage doesn't cause any compilation error.

As, obj is of Profitable type, hence obj.profitPercentage points to the 'profitPercentage' variable of Profitable type. Given code compiles successfully and on execution prints 42.0 on to the console.

Bottom of Form

Top of Form

Question 23: **Correct**

**Question ID: UKOCP51068**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. interface I1 {
4. public static void print(String str) {
5. System.out.println("I1:" + str.toUpperCase());
6. }
7. }
9. class C1 implements I1 {
10. void print(String str) {
11. System.out.println("C1:" + str.toLowerCase());
12. }
13. }
15. public class Test {
16. public static void main(String[] args) {
17. I1 obj = new C1();
18. obj.print("Java");
19. }
20. }

Which of the following statements is correct?

* 

**Class C1 causes compilation error**

* 

**Class Test causes compilation error**

**(Correct)**

* 

**Interface I1 causes compilation error**

* 

**Given code compiles successfully and on execution prints I1:JAVA on to the console**

* 

**Given code compiles successfully and on execution prints C1:java on to the console**

**Explanation**

**UKOCP51068:**

As per Java 8, default and static methods were added in the interface. Interface I1 defines static method print(String), there is no compilation error in interface I1.

Also the scope of print(String) method of I1 is limited to interface I1 and it can be invoked by using Interface name only, I1.print("").

class C1 implements I1 and it also defines print(String) instance method. Even though class C1 implements I1, it doesn't have static print(String) method in its scope, therefore class C1 compiles successfully.

Super type reference variable can refer to an instance of Sub type, therefore the statement `I1 obj = new C1();` compiles successfully.

obj is of I1 type, hence `obj.print("Java");` tries to tag the static method of I1 but static print(String) method of I1 can only be invoked by using I1.print("Java");.

Therefore, `obj.print("Java");` causes compilation error.

Bottom of Form

Top of Form

Question 24: **Correct**

**Question ID: UKOCP38092**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. interface Document {
4. default String getType() {
5. return "TEXT";
6. }
7. }
9. interface WordDocument extends Document {
10. String getType();
11. }
13. class Word implements WordDocument {}
15. public class Test {
16. public static void main(String[] args) {
17. Document doc = new Word(); //Line n1
18. System.out.println(doc.getType()); //Line n2
19. }
20. }

Which of the following statements is correct?

* 

**Interface Document causes compilation error**

* 

**Interface WordDocument causes compilation error**

* 

**Class Word causes compilation error**

**(Correct)**

* 

**Test class compiles successfully and on execution prints TEXT on to the console**

**Explanation**

**UKOCP38092:**

As per Java 8, default methods were added in the interface.

Interface Document defines default method getType(), there is no compilation error in interface Document. Method getType() is implicitly public in Document.

Interface WordDocument extends Document and it overrides the default method getType() of Document, overriding method in WordDocument is implicitly abstract and public. An interface in java can override the default method of super type with abstract modifier. Interface WordDocument compiles successfully.

Class Word implements WordDocument and as WordDocument interface has abstract method getType(), and as class Word doesn't implement the getType() method hence it causes compilation failure.

Bottom of Form

Top of Form

Question 25: **Correct**

**Question ID: UKOCP44779**

Consider below code snippet:

1. interface Workable {
2. void work();
3. }
5. /\*INSERT\*/ {
6. public void work() {} //Line n1
7. }

And the statements:

1. abstract class Work implements Workable

2. class Work implements Workable

3. interface Work extends Workable

4. abstract interface Work extends Workable

5. abstract class Work

How many statements can replace /\*INSERT\*/ such that there is no compilation error?

* 

**One statement**

* 

**Two statements**

* 

**Three statements**

**(Correct)**

* 

**Four statements**

* 

**Five statements**

**Explanation**

**UKOCP44779:**

/\*INSERT\*/ cannot be replaced with interface as work() method at Line n1 is neither abstract nor default. Hence, statements 3 and 4 will not work.

Let's check other statements:

1. abstract class Work implements Workable: abstract class in java can have 0 or more abstract methods. It compiles successfully.

2. class Work implements Workable: It correctly implements the work() method of Workable interface, hence it compiles successfully.

5. abstract class Work: abstract class in java can have 0 or more abstract methods. It compiles successfully.

Hence, out of 5 statements, 3 will compile successfully.

Bottom of Form

Top of Form

Question 26: **Correct**

**Question ID: UKOCP43448**

Which of the annotation is used for Functional Interface?

* 

**@Functional**

* 

**@FI**

* 

**@FunctionalInterface**

**(Correct)**

* 

**@Functional Interface**

**Explanation**

**UKOCP43448:**

@FunctionalInterface annotation is used to tag a functional interface.

Bottom of Form

Top of Form

Question 27: **Incorrect**

**Question ID: UKOCP21649**

Does below code compile successfully?

1. package com.udayankhattry.ocp;
3. @FunctionalInterface
4. interface I1 {
5. void print();
6. boolean equals();
7. }

* 

**Yes**

**(Incorrect)**

* 

**No**

**(Correct)**

**Explanation**

**UKOCP21649:**

@FunctionalInterface annotation cannot be used here as interface I1 specifies two non-overriding abstract methods. This code causes compilation error.

Bottom of Form

Top of Form

Question 28: **Correct**

**Question ID: UKOCP30130**

Is below functional interface correctly defined?

1. @FunctionalInterface
2. interface Tester {
3. boolean equals(Object obj);
4. }

* 

**Yes**

* 

**No**

**(Correct)**

**Explanation**

**UKOCP30130:**

Functional interface must have one and only one non-overriding abstract method.

boolean equals(Object) is declared and defined in Object class, hence it is not non-overriding abstract method.

@FunctionalInterface annotation causes compilation error.

Bottom of Form

Top of Form

Question 29: **Correct**

**Question ID: UKOCP67892**

Which of the following are Functional Interfaces?

Select 3 options.

* 

**java.util.Comparator**

**(Correct)**

* 

**java.lang.Runnable**

**(Correct)**

* 

**java.awt.event.ActionListener**

**(Correct)**

* 

**java.io.Serializable**

* 

**java.lang.Cloneable**

**Explanation**

**UKOCP67892:**

Comparator has only one non-overriding abstract method, compare. Runnable has only one non-overriding abstract method, run.

ActionListener has only one non-overriding abstract method, actionPerformed. Serializable and Cloneable are marker interfaces.

Bottom of Form

Top of Form

Question 30: **Correct**

**Question ID: UKOCP63051**

Consider below code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. enum Directions {
5. NORTH("N"), SOUTH("S"), EAST("E"), WEST("W")
7. private String notation;
9. Directions(String notation) {
10. this.notation = notation;
11. }
13. public String getNotation() {
14. return notation;
15. }
16. }
18. public static void main(String[] args) {
19. System.out.println(Test.Directions.NORTH.getNotation());
20. }
21. }

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

* 

**N**

* 

**NORTH**

* 

**Compilation error**

**(Correct)**

* 

**Exception is thrown at runtime**

**Explanation**

**UKOCP63051:**

As enum Directions contains more code after constant declarations, hence last constant declaration must be followed by a semicolon.

Correct constant declaration is:

NORTH("N"), SOUTH("S"), EAST("E"), WEST("W");

Bottom of Form

Top of Form

Question 31: **Correct**

**Question ID: UKOCP67893**

Consider the code of Test.java file:

1. package com.udayankhattry.ocp;
3. enum Flags {
4. TRUE, FALSE;
6. Flags() {
7. System.out.println("HELLO");
8. }
9. }
11. public class Test {
12. public static void main(String[] args) {
13. Flags flags = new Flags();
14. }
15. }

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

* 

**HELLO is printed twice**

* 

**HELLO is printed once**

* 

**Exception is thrown at runtime**

* 

**None of the other options**

**(Correct)**

**Explanation**

**UKOCP67893:**

`new Flags();` tries to instantiate enum type Flags, but enum types may not be instantiated.

Bottom of Form

Top of Form

Question 32: **Incorrect**

**Question ID: UKOCP11854**

Consider the code of Test.java file:

1. package com.udayankhattry.ocp;
3. enum Flags {
4. TRUE, FALSE;
6. Flags() {
7. System.out.println("HELLO");
8. }
9. }
11. public class Test {
12. public static void main(String[] args) {
13. Flags flags = Flags.TRUE;
14. }
15. }

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

* 

**HELLO is printed twice**

**(Correct)**

* 

**HELLO is printed once**

**(Incorrect)**

* 

**Exception is thrown at runtime**

* 

**None of the other options**

**Explanation**

**UKOCP11854:**

Enum constructor is invoked once for every constant.

For 'Flags.TRUE', enum constructor is invoked for TRUE as well as FALSE.

Bottom of Form

Top of Form

Question 33: **Correct**

**Question ID: UKOCP34974**

Consider the code of Test.java file:

1. package com.udayankhattry.ocp;
3. enum Flags {
4. TRUE, FALSE;
6. public Flags() {
7. System.out.println("HELLO");
8. }
9. }
11. public class Test {
12. public static void main(String[] args) {
13. Flags flags = Flags.TRUE;
14. }
15. }

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

* 

**HELLO is printed twice**

* 

**HELLO is printed once**

* 

**Exception is thrown at runtime**

* 

**Compilation error**

**(Correct)**

**Explanation**

**UKOCP34974:**

Enum constructors are implicitly private, even though you can provide private access modifier but it will be redundant.

Using 'public' or 'protected' for enum constructors is not allowed.

Bottom of Form

Top of Form

Question 34: **Correct**

**Question ID: UKOCP19334**

Consider the code of Test.java file:

1. package com.udayankhattry.ocp;
3. enum Flags {
4. TRUE;
6. Flags() {
7. System.out.println("HELLO");
8. }
9. }
11. public class Test {
12. public static void main(String[] args) {
13. Flags f1 = Flags.TRUE;
14. Flags f2 = Flags.TRUE;
15. Flags f3 = Flags.TRUE;
16. }
17. }

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

* 

**HELLO is printed once**

**(Correct)**

* 

**HELLO is printed twice**

* 

**HELLO is printed thrice**

* 

**HELLO Is not printed on to the console**

**Explanation**

**UKOCP19334:**

Enum constructor is invoked once for every constant. There is only one constant, hence constructor is invoked only once.

For first 'Flags.TRUE', enum constructor is invoked but for later statements, enum constructor is not invoked.

Bottom of Form

Top of Form

Question 35: **Correct**

**Question ID: UKOCP35807**

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

1. package com.udayankhattry.ocp;
3. enum Status {
4. PASS, FAIL, PASS;
5. }
7. public class Test {
8. public static void main(String[] args) {
9. System.out.println(Status.FAIL);
10. }
11. }

* 

**FAIL**

* 

**fail**

* 

**Fail**

* 

**None of the other options**

**(Correct)**

**Explanation**

**UKOCP35807:**

enum Status causes compilation error as constant name should be unique, but PASS is declared twice.

Bottom of Form

Top of Form

Question 36: **Correct**

**Question ID: UKOCP30118**

Below is the code of Test.java file:

1. public class Test {
2. enum TrafficLight {
3. RED, YELLOW, GREEN;
4. }
6. public static void main(String[] args) {
7. TrafficLight tl = TrafficLight.valueOf(args[0]);
8. switch(tl) {
9. case RED:
10. System.out.println("STOP");
11. break;
12. case YELLOW:
13. System.out.println("SLOW");
14. break;
15. case GREEN:
16. System.out.println("GO");
17. break;
18. }
19. }
20. }

What will be the output if Test class is executed by the commands:

javac Test.java

java Test GREEN AMBER

* 

**GO**

**(Correct)**

* 

**IllegalArgumentException is thrown at runtime**

* 

**NullPointerException is thrown at runtime**

* 

**Compilation error**

**Explanation**

**UKOCP30118:**

args[0] refers to "GREEN" and args[1] refers to "AMBER".

TrafficLight.valueOf(args[0]); -> TrafficLight.valueOf("GREEN");

GREEN is a valid enum constant, hence case label for GREEN is executed and "GO" is printed to the console.

Bottom of Form

Top of Form

Question 37: **Incorrect**

**Question ID: UKOCP30119**

Below is the code of Test.java file:

1. public class Test {
2. enum TrafficLight {
3. RED, YELLOW, GREEN;
4. }
6. public static void main(String[] args) {
7. TrafficLight tl = TrafficLight.valueOf(args[1]);
8. switch(tl) {
9. case TrafficLight.RED:
10. System.out.println("STOP");
11. break;
12. case TrafficLight.YELLOW:
13. System.out.println("SLOW");
14. break;
15. case TrafficLight.GREEN:
16. System.out.println("GO");
17. break;
18. }
19. }
20. }

What will be the result of compiling and executing Test class by using the commands:

javac Test.java

java Test RED AMBER

* 

**STOP**

* 

**No output**

**(Incorrect)**

* 

**IllegalArgumentException is thrown**

* 

**None of the other options**

**(Correct)**

**Explanation**

**UKOCP30119:**

case labels accept unqualified names of enum constants.

case TrafficLight.RED:, case TrafficLight.YELLOW: and case TrafficLight.GREEN: cause  compilation error.

Correct case labels should be:

case RED:, case YELLOW: and case GREEN:

Bottom of Form

Top of Form

Question 38: **Correct**

**Question ID: UKOCP74198**

Which of the following statement is correct about java enums?

* 

**An enum can extend another class**

* 

**An enum can extend another enum**

* 

**An enum can implement interfaces**

**(Correct)**

* 

**All java enums implicitly extend from java.util.Enum class**

**Explanation**

**UKOCP74198:**

Java enums cannot extend from another class or enum but an enum can implement interfaces.

All java enums implicitly extend from java.lang.Enum class and not from java.util.Enum class.

Bottom of Form

Top of Form

Question 39: **Correct**

**Question ID: UKOCP81700**

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

1. package com.udayankhattry.ocp;
3. public class Test {
4. enum TrafficLight {
5. private String message;
6. GREEN("go"), AMBER("slow"), RED("stop");
8. TrafficLight(String message) {
9. this.message = message;
10. }
12. public String getMessage() {
13. return message;
14. }
15. }
17. public static void main(String[] args) {
18. System.out.println(TrafficLight.AMBER.getMessage().toUpperCase());
19. }
20. }

* 

**slow**

* 

**SLOW**

* 

**NullPointerException is thrown at runtime**

* 

**Compilation error**

**(Correct)**

**Explanation**

**UKOCP81700:**

Enum constant list must be the first item in an enum.

GREEN("go"), AMBER("slow"), RED("stop"); should be the first line inside TrafficLight enum.

Bottom of Form

Top of Form

Question 40: **Correct**

**Question ID: UKOCP55895**

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

1. package com.udayankhattry.ocp;
3. public class Test {
4. enum JobStatus {
5. SUCCESS, FAIL; //Line n1
6. }
8. enum TestResult {
9. PASS, FAIL; //Line n2
10. }
12. public static void main(String[] args) {
13. JobStatus js = JobStatus.FAIL;
14. TestResult tr = TestResult.FAIL;
16. System.out.println(js.equals(tr)); //Line n3
17. System.out.println(js == tr); //Line n4
18. }
19. }

* 

**Compilation error at Line n3**

* 

**Compilation error at Line n4**

**(Correct)**

* 

**true**

**true**

* 

**false**

**false**

**Explanation**

**UKOCP55895:**

enums JobStatus and TestResult are siblings as both implicitly extend from java.lang.Enum class.

Siblings cannot be compared using double equals operator (==).

equals(Object) method accepts any instance which Object can refer to, which means all the instances.

Bottom of Form

Top of Form

Question 41: **Correct**

**Question ID: UKOCP74191**

Given:

1. package com.udayankhattry.ocp;
3. enum TrafficLight {
4. RED, YELLOW, GREEN;
5. }
7. public class Test {
8. public static void main(String[] args) {
9. TrafficLight tl1 = TrafficLight.GREEN;
10. TrafficLight tl2 = tl1.clone(); //Line n1
11. System.out.println(tl2); //Line n2
12. }
13. }

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

* 

**GREEN**

* 

**Compilation error at Line n1**

**(Correct)**

* 

**Line n2 throws CloneNotSupportedException at runtime**

**Explanation**

**UKOCP74191:**

Every enum extends from java.lang.Enum class and it contains following definition of clone method:

1. protected final Object clone() throws CloneNotSupportedException {
2. throw new CloneNotSupportedException();
3. }

Every enum constant (RED, YELLOW, GREEN) is an instance of TrafficLight enum and as clone method is protected in Enum class so it cannot be accessed in com.udayankhattry.ocp package using reference variable.

Bottom of Form

Top of Form

Question 42: **Correct**

**Question ID: UKOCP55892**

Given code of Test.java file:

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

Does above code compile successfully?

* 

**Yes**

**(Correct)**

* 

**No**

**Explanation**

**UKOCP55892:**

Throwable is the root class of the exception hierarchy and it contains some useful constructors:

1. public Throwable() {...} : No-argument constructor

2. public Throwable(String message) {...} : Pass the detail message

3. public Throwable(String message, Throwable cause) {...} : Pass the detail message and the cause

4. public Throwable(Throwable cause) {...} : Pass the cause

Exception and RuntimeException classes also provide similar constructors.

Hence all 3 statements Line n1, Line n2 and Line n3 compile successfully.

Throwable class also contains methods, which are inherited by all the subclasses (Exception, RuntimeException etc.)

1. public String getMessage() {...} : Returns the detail message (E.g. detail message set by 2nd and 3rd constructor)

2. public String toString() {} :

Returns a short description of this throwable. The result is the concatenation of:

the name of the class of this object

": " (a colon and a space)

the result of invoking this object's getLocalizedMessage() method

If getLocalizedMessage returns null, then just the class name is returned.

Because of the toString() method,

Line n1 prints "java.lang.RuntimeException".

Line n2 prints "java.lang.RuntimeException: HELLO"

Line n3 prints "java.lang.RuntimeException: java.lang.RuntimeException: HELLO"

Bottom of Form

Top of Form

Question 43: **Correct**

**Question ID: UKOCP41258**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. Error obj = new Error();
6. boolean flag1 = obj instanceof RuntimeException; //Line n1
7. boolean flag2 = obj instanceof Exception; //Line n2
8. boolean flag3 = obj instanceof Error; //Line n3
9. boolean flag4 = obj instanceof Throwable; //Line n4
10. System.out.println(flag1 + ":" + flag2 + ":" + flag3 + ":" + flag4);
11. }
12. }

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

* 

**Compilation error**

**(Correct)**

* 

**false:false:true:true**

* 

**false:true:true:true**

* 

**true:true:true:true**

* 

**false:false:true:false**

**Explanation**

**UKOCP41258:**

class Error extends Throwable, so `obj instanceof Error;` and `obj instanceof Throwable;` return true.

But Error class is not related to Exception and RuntimeException classes in multilevel inheritance and that is why Line n1 and Line n2 cause compilation error.

Bottom of Form

Top of Form

Question 44: **Correct**

**Question ID: UKOCP43435**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. interface ILogger {
4. void log();
5. }
7. public class Test {
8. public static void main(String[] args) {
9. ILogger [] loggers = new ILogger[2]; //Line n1
10. for(ILogger logger : loggers)
11. logger.log(); //Line n2
12. }
13. }

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

* 

**Line n1 causes compilation error**

* 

**Line n2 causes compilation error**

* 

**An exception is thrown at runtime**

**(Correct)**

* 

**No output is displayed but program terminates successfully**

**Explanation**

**UKOCP43435:**

Line n1 creates an array instance of ILogger containing 2 elements. null is assigned to both the array elements. Line n1 compiles successfully.

As, log() method is declared in ILogger interface, hence statement at Line n2: logger.log(); doesn't cause any compilation error. Compiler is happy to see that log() method is invoked on the reference variable of ILogger type.

1st iteration:

logger --> null, logger.log(); throws NullPointerException as method log() is invoked on null reference.

Bottom of Form

Top of Form

Question 45: **Correct**

**Question ID: UKOCP21659**

Consider codes of 3 java files:

1. //Class1.java
2. package com.udayankhattry.ocp;
4. import java.io.FileNotFoundException;
6. public class Class1 {
7. public void read() throws FileNotFoundException {}
8. }
9. //Class2.java
10. public class Class2 {
11. String Class2;
12. public void Class2() {}
13. }
14. //Class3.java
15. public class Class3 {
16. private void print() {
17. private String msg = "HELLO";
18. System.out.println(msg);
19. }
20. }

Which of the following statements is true?

* 

**Only Class1.java compiles successfully**

* 

**Only Class2.java compiles successfully**

* 

**Only Class3.java compiles successfully**

* 

**Class1.java and Class2.java compile successfully**

**(Correct)**

* 

**Class1.java and Class3.java compile successfully**

* 

**Class2.java and Class3.java compile successfully**

**Explanation**

**UKOCP21659:**

java.io.FileNotFoundException exception is a checked exception.

Method declaring checked exception in its throws clause doesn't mean that it should have code to actually throw that type of Exceptions. So even though read() method of Class1 declares to throw FileNotFoundException but its body doesn't actually throw an instance of FileNotFoundException.

Variable and method name can be same as class name, so code of Class2 is also valid. Remember: Though you don't get any compilation error but it is not recommended to use the Class name for variable and method names.

LOCAL variable can be declared with final modifier only. msg variable inside print() method of Class3 is declared private and this causes compilation error.

Bottom of Form

Top of Form

Question 46: **Incorrect**

**Question ID: UKOCP42586**

1. Given code of Test.java file:
2. package com.udayankhattry.ocp;
4. import java.io.FileNotFoundException;
6. public class Test {
7. public static void main(String[] args) {
8. try {
9. System.out.println(args[1].length());
10. } catch (RuntimeException ex) {
11. System.out.println("ONE");
12. } catch (FileNotFoundException ex) {
13. System.out.println("TWO");
14. }
15. System.out.println("THREE");
16. }
17. }

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

* 

**ONE**

**THREE**

**(Incorrect)**

* 

**TWO**

**THREE**

* 

**THREE**

* 

**None of the System.out.println statements is executed**

* 

**Compilation error**

**(Correct)**

**Explanation**

**UKOCP42586:**

java.io.FileNotFoundException exception is a checked exception.

Java doesn't allow to catch specific checked exceptions if these are not thrown by the statements inside try block. catch(FileNotFoundException ex) {} causes compilation error in this case as System.out.println(1); will never throw FileNotFoundException.

NOTE: Java allows to catch Exception type. catch(Exception ex) {} will never cause compilation error.

Bottom of Form

Top of Form

Question 47: **Correct**

**Question ID: UKOCP41253**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. StringBuilder sb = new StringBuilder();
6. try {
7. for(;;) {
8. sb.append("1Z0-819");
9. }
10. } catch(Exception e) {
11. System.out.println("Exception!!!");
12. }
13. System.out.println("Main ends!!!");
14. }
15. }

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

* 

**"Main ends!!!" is printed on to the console and program terminates successfully**

* 

**"Exception!!!" and "Main ends!!!" are printed on to the console and program terminates successfully**

* 

**"Exception!!!" is printed on to the console and program terminates successfully**

* 

**"Exception!!!" is printed on to the console and program terminates abruptly**

* 

**Program terminates abruptly**

**(Correct)**

**Explanation**

**UKOCP41253:**

for(;;) is an infinite loop and hence `sb.append("1Z0-819");` causes OutOfMemoryError which is a subclass of Error class.

main(String []) method throws OutOfMemoryError and program terminates abruptly.

Bottom of Form

Top of Form

Question 48: **Correct**

**Question ID: UKOCP31451**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. try {
6. main(args);
7. } catch (Exception ex) {
8. System.out.println("CATCH-"); //Line n1
9. }
10. System.out.println("OUT"); //Line n2
11. }
12. }

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

* 

**CATCH-OUT is printed and program terminates successfully**

* 

**OUT is printed and program terminates successfully**

* 

**System.out.println statements at Line n1 and Line n2 are not executed and program ends abruptly**

**(Correct)**

* 

**Compilation error**

**Explanation**

**UKOCP31451:**

main(args) method is invoked recursively without specifying any exit condition, so this code ultimately throws java.lang.StackOverflowError. StackOverflowError is a subclass of Error type and not Exception type, hence it is not handled. Stack trace is printed to the console and program ends abruptly. Statements at Line n1 and Line n2 are not executed.

Java doesn't allow to catch specific checked exceptions if these are not thrown by the statements inside try block.

catch(java.io.FileNotFoundException ex) {} will cause compilation error in this case as main(args); will never throw FileNotFoundException. But Java allows to catch Exception type, hence catch (Exception ex) {} doesn't cause any compilation error.

Bottom of Form

Top of Form

Question 49: **Correct**

**Question ID: UKOCP42581**

Which of the following are Java Exception classes?

Select 3 options.

* 

**ClassCastException**

**(Correct)**

* 

**NullException**

* 

**NumberFormatException**

**(Correct)**

* 

**IllegalArgumentException**

**(Correct)**

* 

**ArrayIndexException**

**Explanation**

**UKOCP42581:**

ClassCastException, NumberFormatException and IllegalArgumentException are Runtime exceptions. There are no exception classes in java with the names: NullException and ArrayIndexException.

Bottom of Form

Top of Form

Question 50: **Correct**

**Question ID: UKOCP19344**

Which of the following is a checked Exception?

* 

**ClassCastException**

* 

**FileNotFoundException**

**(Correct)**

* 

**ExceptionInInitializerError**

* 

**RuntimeException**

**Explanation**

**UKOCP19344:**

ClassCastException extends RuntimeException (unchecked exception),

FileNotFoundException extends IOException, IOException extends Exception (checked exception),

ExceptionInInitializerError is from Error family and is thrown by an static initializer block,

RuntimeException and all its sub classes are unchecked exceptions.

Bottom of Form

Top of Form

Question 51: **Correct**

**Question ID: UKOCP30122**

Given code of Test.java file:

1. public class Test {
2. private static int [] arr;
3. public static void main(String [] args) {
4. if(arr.length > 0 && arr != null) {
5. System.out.println(arr[0]);
6. }
7. }
8. }

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

* 

**No Output**

* 

**NullPointerException is thrown at runtime**

**(Correct)**

* 

**ArrayIndexOutOfBoundsException is thrown at runtime**

* 

**Error is thrown at runtime**

**Explanation**

**UKOCP30122:**

Variable 'arr' is a class variable of int [] type, so by default it is initialized to null.

In if block, arr.length > 0 is checked first. Accessing length property on null reference throws NullPointerException.

Correct logical if block declaration should be:

if(arr != null && arr.length > 0)

First check for null and then access properties/methods.

Bottom of Form

Top of Form

Question 52: **Correct**

**Question ID: UKOCP53229**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. private static void div() {
5. System.out.println(1/0);
6. }
8. public static void main(String[] args) {
9. try {
10. div();
11. } finally {
12. System.out.println("FINALLY");
13. }
14. }
15. }

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

* 

**FINALLY is printed to the console and program ends normally**

* 

**FINALLY is printed to the console, stack trace is printed and then program ends normally**

* 

**FINALLY is printed to the console, stack trace is printed and then program ends abruptly**

**(Correct)**

* 

**Compilation error**

**Explanation**

**UKOCP53229:**

As method div() doesn't declare to throw any Checked Exception, hence main(String []) method is not suppose to handle it, try-finally without catch is valid here. There is no compilation error in the code.

Method div() throws an instance of ArithmeticException and method div() doesn't handle it, so it forwards the exception to calling method main(String []).

Method main(String []) doesn't handle ArithmeticException so it forwards it to JVM, but just before that, finally block is executed. This prints FINALLY on to the console.

After that JVM prints the stack trace and terminates the program abruptly.

Bottom of Form

Top of Form

Question 53: **Incorrect**

**Question ID: UKOCP56776**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. private static void test() throws Exception {
5. throw new Exception();
6. }
8. public static void main(String [] args) {
9. try {
10. test();
11. } finally {
12. System.out.println("GAME ON");
13. }
14. }
15. }

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

* 

**GAME ON is printed to the console and program ends normally**

* 

**GAME ON is printed to the console, stack trace is printed and then program ends normally**

* 

**GAME ON is printed to the console, stack trace is printed and then program ends abruptly**

**(Incorrect)**

* 

**Compilation error**

**(Correct)**

**Explanation**

**UKOCP56776:**

Method test() throws Exception (checked) and it declares to throw it, so no issues with method test().

But main(String []) method neither provides catch handler nor throws clause and hence main(String []) method causes compilation error.

Handle or Declare rule should be followed for checked exception if you are not re-throwing it.

Bottom of Form

Top of Form

Question 54: **Incorrect**

**Question ID: UKOCP12715**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. private static String s;
5. public static void main(String[] args) {
6. try {
7. System.out.println(s.length());
8. } catch(NullPointerException | RuntimeException ex) {
9. System.out.println("DONE");
10. }
11. }
12. }

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

* 

**DONE**

* 

**Executes successfully but no output**

* 

**Compilation error**

**(Correct)**

* 

**None of the other options**

**(Incorrect)**

**Explanation**

**UKOCP12715:**

NullPointerException extends RuntimeException and in multi-catch syntax we can't specify multiple Exceptions related to each other in multilevel inheritance.

Bottom of Form

Top of Form

Question 55: **Correct**

**Question ID: UKOCP46971**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.io.IOException;
4. import java.sql.SQLException;
6. public class Test {
7. public static void main(String[] args) {
8. /\*INSERT\*/
9. }
11. private static void save() throws IOException {}
13. private static void log() throws SQLException {}
14. }

Which of the block of codes can be used to replace /\*INSERT\*/ such that there is no compilation error?

Select 3 options.

* 
  1. try {
  2. save();
  3. log();
  4. } catch(IOException | SQLException ex) {}

**(Correct)**

* 
  1. try {
  2. save();
  3. log();
  4. } catch(SQLException | IOException ex) {}

**(Correct)**

* 
  1. try {
  2. save();
  3. log();
  4. } catch(IOException | Exception ex) {}
* 
  1. try {
  2. save();
  3. log();
  4. } catch(SQLException | Exception ex) {}
* 
  1. try {
  2. save();
  3. log();
  4. } catch(Exception | RuntimeException ex) {}
* 
  1. try {
  2. save();
  3. log();
  4. } catch(Exception ex) {}

**(Correct)**

**Explanation**

**UKOCP46971:**

save() method throws IOException (which is a Checked Exception) and log() method throws SQLException (which is also a Checked Exception).

Let's check all the options one by one (I am just using the catch-block as try-block of all the options are same):

catch(IOException | SQLException ex) {}: ✓ As IOException and SQLException are not related to each other in multi-level inheritance, hence this multi-catch syntax is valid.

catch(SQLException | IOException ex) {}: ✓ Same as above, order of exceptions in multi-catch syntax doesn't matter.

catch(IOException | Exception ex) {}: ✗ Causes compilation error as IOException extends Exception.

catch(SQLException | Exception ex) {}: ✗ Causes compilation error as SQLException extends Exception.

catch(Exception | RuntimeException ex) {}: ✗ Causes compilation error as RuntimeException extends Exception.

catch(Exception ex) {}: ✓ As Exception is the super class of both IOException and SQLException, hence it can handle both the exceptions.

Bottom of Form

Top of Form

Question 56: **Correct**

**Question ID: UKOCP46973**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. try { //outer
6. try { //inner
7. System.out.println(1/0);
8. } catch(ArithmeticException e) {
9. System.out.println("INNER");
10. } finally {
11. System.out.println("FINALLY 1");
12. }
13. } catch(ArithmeticException e) {
14. System.out.println("OUTER");
15. } finally {
16. System.out.println("FINALLY 2");
17. }
18. }
19. }

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

* 

**INNER**

**FINALLY 1**

* 

**OUTER**

**FINALLY 2**

* 

**INNER**

**FINALLY 2**

* 

**INNER**

**FINALLY 1**

**FINALLY 2**

**(Correct)**

**Explanation**

**UKOCP46973:**

`System.out.println(1/0);` throws ArithmeticException, handler is available in inner catch-block, it executes and prints "INNER" to the console.

Once an exception is handled, no other catch block will get executed unless the exception is re-thrown.

Inner finally-block gets executed and prints "FINALLY 1" to the console.

Rule is finally-block always gets executed, so outer finally-block gets executed and prints "FINALLY 2" to the console.

Bottom of Form

Top of Form

Question 57: **Correct**

**Question ID: UKOCP17174**

java.lang.IllegalArgumentException extends java.lang.RuntimeException

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void convert(String s)
5. throws IllegalArgumentException, RuntimeException, Exception {
6. if(s.length() == 0) {
7. throw new RuntimeException("LENGTH SHOULD BE GREATER THAN 0");
8. }
9. }
10. public static void main(String [] args) {
11. try {
12. convert("");
13. }
14. catch(IllegalArgumentException | RuntimeException | Exception e) { //Line n1
15. System.out.println(e.getMessage()); //Line n2
16. } //Line n3
17. catch(Exception e) {
18. e.printStackTrace();
19. }
20. }
21. }

Line n1 causes compilation error. Which of the following changes enables to code to print LENGTH SHOULD BE GREATER THAN 0?

* 

**Replace Line n1 with 'catch(RuntimeException | Exception e) {'**

* 

**Replace Line n1 with 'catch(IllegalArgumentException | Exception e) {'**

* 

**Replace Line n1 with 'catch(IllegalArgumentException | RuntimeException e) {'**

* 

**Replace Line n1 with 'catch(RuntimeException e) {'**

**(Correct)**

* 

**Comment out Line n1, Line n2 and Line n3**

**Explanation**

**UKOCP17174:**

Throwable is the root class of the exception hierarchy and it contains some useful constructors:

1. public Throwable() {...} : No-argument constructor

2. public Throwable(String message) {...} : Pass the detail message

3. public Throwable(String message, Throwable cause) {...} : Pass the detail message and the cause

4. public Throwable(Throwable cause) {...} : Pass the cause

Exception and RuntimeException classes also provide similar constructors.

Throwable class also contains methods, which are inherited by all the subclasses (Exception, RuntimeException etc.)

1. public String getMessage() {...} : Returns the detail message (E.g. detail message set by 2nd and 3rd constructor)

2. public String toString() {} :

Returns a short description of this throwable. The result is the concatenation of:

the name of the class of this object

": " (a colon and a space)

the result of invoking this object's getLocalizedMessage() method

If getLocalizedMessage() returns null, then just the class name is returned.

In multi-catch statement, classes with multi-level hierarchical relationship can't be used.

RuntimeException is subclass of Exception, IllegalArgumentException is indirect subclass of Exception and IllegalArgumentException is subclass of RuntimeException, hence these pairs can't be used in multi-catch statement.

Only one option is left to replace Line n1 with 'catch(RuntimeException e) {'.

Commenting out Line n1, Line n2 and Line n3 will resolve the compilation error but it will print the whole stack trace rather than just printing the message.

Bottom of Form

Top of Form

Question 58: **Correct**

**Question ID: UKOCP53263**

Given code of Test.java file:

1. public class Test {
2. public static void main(String[] args) {
3. try {
4. try {
5. System.out.println(args[1]); //Line n1
6. } catch(RuntimeException e) {
7. System.out.print("INHALE-"); //Line n2
8. throw e; //Line n3
9. } finally {
10. System.out.print("EXHALE-"); //Line n4
11. }
12. } catch(RuntimeException e) {
13. System.out.print("INHALE-"); //Line n5
14. } finally {
15. System.out.print("EXHALE"); //Line n6
16. }
17. }
18. }

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

* 

**INHALE-EXHALE**

* 

**INHALE-EXHALE-**

* 

**INHALE-EXHALE-INHALE-**

* 

**INHALE-EXHALE-EXHALE**

* 

**INHALE-EXHALE-INHALE-EXHALE**

**(Correct)**

**Explanation**

**UKOCP53263:**

As command-line argument is not passed, hence Line n1 throws ArrayIndexOutOfBoundsException (subclass of RuntimeException), handler is available in inner catch block, it executes Line n2 and prints INHALE- on to the console.

throw e; re-throws the exception.

But before exception instance is forwarded to outer catch-block, inner finally-block gets executed and prints EXHALE- on to the console.

In outer try-catch block, handler for RuntimeException is available, so outer catch-block gets executed and prints INHALE- on to the console.

After that outer finally-block gets executed and prints EXHALE- on to the console.

Hence, the output is: INHALE-EXHALE-INHALE-EXHALE

Bottom of Form

Top of Form

Question 59: **Correct**

**Question ID: UKOCP44311**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. private static void div(int i, int j) {
5. try {
6. System.out.println(i / j);
7. } catch(ArithmeticException e) {
8. Exception ex = new Exception(e);
9. throw ex;
10. }
11. }
12. public static void main(String[] args) {
13. try {
14. div(5, 0);
15. } catch(Exception e) {
16. System.out.println("END");
17. }
18. }
19. }

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

* 

**Compilation error**

**(Correct)**

* 

**END is printed and program terminates successfully**

* 

**END is printed and program terminates abruptly**

* 

**END is not printed and program terminates abruptly**

**Explanation**

**UKOCP44311:**

throw ex; causes compilation error as div method doesn't declare to throw Exception (checked) type.

Bottom of Form

Top of Form

Question 60: **Correct**

**Question ID: UKOCP75492**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. private static void div(int i, int j) {
5. try {
6. System.out.println(i / j);
7. } catch(ArithmeticException e) {
8. throw (RuntimeException)e;
9. }
10. }
12. public static void main(String[] args) {
13. try {
14. div(5, 0);
15. } catch(ArithmeticException e) {
16. System.out.println("AE");
17. } catch(RuntimeException e) {
18. System.out.println("RE");
19. }
20. }
21. }

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

* 

**Compilation error**

* 

**Program ends abruptly**

* 

**AE is printed on to the console and program terminates successfully**

**(Correct)**

* 

**RE is printed on to the console and program terminates successfully**

**Explanation**

**UKOCP75492:**

Any RuntimeException can be thrown without any need it to be declared in throws clause of surrounding method.

`throw (RuntimeException)e;` doesn't cause any compilation error.

Even though variable 'e' is type casted to RuntimeException but exception object is still of ArithmeticException, which is caught in main method and 'AE' is printed to the console.

Bottom of Form

Top of Form

Question 61: **Correct**

**Question ID: UKOCP51080**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.sql.SQLException;
5. public class Test {
6. private static void m() throws SQLException {
7. try {
8. throw new SQLException();
9. } catch (Exception e) {
10. throw e;
11. }
12. }
14. public static void main(String[] args) {
15. try {
16. m();
17. } catch(SQLException e) {
18. System.out.println("CAUGHT SUCCESSFULLY");
19. }
20. }
21. }

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

* 

**Method m() causes compilation error**

* 

**Method main(String []) causes compilation error**

* 

**CAUGHT SUCCESSFULLY is printed on to the console and program terminates successfully**

**(Correct)**

* 

**Program ends abruptly**

**Explanation**

**UKOCP51080:**

Even though it seems like method m() will not compile successfully, but starting with JDK 7, it is allowed to use super class reference variable in throw statement referring to sub class Exception object.

In this case, method m() throws SQLException and compiler knows that variable e (Exception type) refers to an instance of SQLException only and hence allows it.

Program executes successfully and prints CAUGHT SUCCESSFULLY on to the console.

Bottom of Form

Top of Form

Question 62: **Correct**

**Question ID: UKOCP65699**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.sql.SQLException;
5. public class Test {
6. private static void getData() throws SQLException {
7. try {
8. throw new SQLException();
9. } catch (Exception e) {
10. e = new SQLException();
11. throw e;
12. }
13. }
15. public static void main(String[] args) {
16. try {
17. getData();
18. } catch(SQLException e) {
19. System.out.println("SQL");
20. }
21. }
22. }

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

* 

**Method getData() causes compilation error**

**(Correct)**

* 

**Method main(String []) causes compilation error**

* 

**SQL is printed on to the console and program terminates successfully**

* 

**Program ends abruptly**

**Explanation**

**UKOCP65699:**

If you don't initialize variable e inside catch block using `e = new SQLException();` and simply throw e, then code would compile successfully as compiler is certain that 'e' would refer to an instance of SQLException only.

But the moment compiler finds `e = new SQLException();`, `throw e;` causes compilation error as at runtime 'e' may refer to any Exception type.

Bottom of Form

Top of Form

Question 63: **Correct**

**Question ID: UKOCP22518**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.sql.SQLException;
5. public class Test {
6. private static void checkData() throws SQLException {
7. try {
8. throw new SQLException();
9. } catch (Exception e) {
10. e = null; //Line n1
11. throw e; //Line n2
12. }
13. }
15. public static void main(String[] args) {
16. try {
17. checkData(); //Line n3
18. } catch(SQLException e) {
19. System.out.println("NOT AVAILABLE");
20. }
21. }
22. }

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

* 

**NOT AVAILABLE is printed on to the console and program terminates successfully**

* 

**Program ends abruptly**

* 

**Line n1 causes compilation failure**

* 

**Line n2 causes compilation failure**

**(Correct)**

* 

**Line n3 causes compilation failure**

**Explanation**

**UKOCP22518:**

Exception is a java class, so `e = null;` is a valid statement and compiles successfully.

If you comment Line n1, and simply throw e, then code would compile successfully as compiler is certain that 'e' would refer to an instance of SQLException only.

But the moment compiler finds `e = null;`, `throw e;` (Line n2) causes compilation error as at runtime 'e' may refer to any Exception type.

NOTE: No issues with Line n3 as method checkData() declares to throw SQLException and main(String []) method code correctly handles it.

Bottom of Form

Top of Form

Question 64: **Correct**

**Question ID: UKOCP31480**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.sql.SQLException;
5. public class Test {
6. private static void getReport() throws SQLException {
7. try {
8. throw new SQLException();
9. } catch (Exception e) {
10. throw null; //Line n1
11. }
12. }
14. public static void main(String[] args) {
15. try {
16. getReport(); //Line n2
17. } catch(SQLException e) {
18. System.out.println("REPORT ERROR");
19. }
20. }
21. }

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

* 

**REPORT ERROR is printed on to the console and program terminates successfully**

* 

**Program ends abruptly**

**(Correct)**

* 

**Line n1 causes compilation failure**

* 

**Line n2 causes compilation failure**

**Explanation**

**UKOCP31480:**

Classes in Exception framework are normal java classes, hence null can be used wherever instances of Exception classes are used, so Line n1 compiles successfully.

No issues with Line n2 as method getReport() declares to throw SQLException and main(String []) method code correctly handles it.

Program compiles successfully but on execution, NullPointerException is thrown, stack trace is printed on to the console and program ends abruptly.

If you debug the code, you would find that internal routine for throwing null exception causes NullPointerException.

Bottom of Form

Top of Form

Question 65: **Correct**

**Question ID: UKOCP67875**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.sql.SQLException;
5. public class Test {
6. private static void availableSeats() throws SQLException {
7. throw null; //Line n1
8. }
10. public static void main(String[] args) {
11. try {
12. availableSeats(); //Line n2
13. } catch(SQLException e) {
14. System.out.println("SEATS NOT AVAILABLE");
15. }
16. }
17. }

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

* 

**SEATS NOT AVAILABLE is printed on to the console and program terminates successfully**

* 

**Program ends abruptly**

**(Correct)**

* 

**Line n1 causes compilation failure**

* 

**Line n2 causes compilation failure**

**Explanation**

**UKOCP67875:**

Classes in Exception framework are normal java classes, hence null can be used wherever instances of Exception classes are used, so Line n1 compiles successfully.

No issues with Line n2 as method availableSeats() declares to throw SQLException and main(String []) method code correctly handles it.

Program compiles successfully but on execution, NullPointerException is thrown, stack trace is printed on to the console and program ends abruptly.

If you debug the code, you would find that internal routine for throwing null exception causes NullPointerException.

Bottom of Form

Top of Form

Question 66: **Correct**

**Question ID: UKOCP43463**

Which of the following keywords is used to manually throw an exception?

* 

**throw**

**(Correct)**

* 

**throws**

* 

**thrown**

* 

**catch**

**Explanation**

**UKOCP43463:**

catch is for catching the exception and not throwing it.

thrown is not a java keyword.

throws is used to declare the exceptions, a method can throw.

To manually throw an exception, throw keyword is used. e.g., throw new Exception();

Bottom of Form

Top of Form

Question 67: **Correct**

**Question ID: UKOCP64360**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. check(); //Line n1
6. }
8. private static void check() throws Exception { //Line n2
9. System.out.println("NOT THROWING ANY EXCEPTION"); //Line n3
10. }
11. }

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

* 

**Compilation error at Line n1**

**(Correct)**

* 

**Compilation error at Line n2**

* 

**Compilation error at Line n3**

* 

**NOT THROWING ANY EXCEPTION**

**Explanation**

**UKOCP64360:**

If a method declares to throw Exception or its sub-type other than RuntimeException types, then calling method should follow handle or declare rule.

In this case, as method check() declares to throw Exception, so main method should either declare the same exception or its super type in its throws clause OR check(); should be surrounded by try-catch block.

Line n1 in this case causes compilation error.

Bottom of Form

Top of Form

Question 68: **Correct**

**Question ID: UKOCP19357**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.io.FileNotFoundException;
4. import java.io.IOException;
6. abstract class Super {
7. public abstract void m1() throws IOException;
8. }
10. class Sub extends Super {
11. @Override
12. public void m1() throws IOException {
13. throw new FileNotFoundException();
14. }
15. }
17. public class Test {
18. public static void main(String[] args) {
19. Super s = new Sub();
20. try {
21. s.m1();
22. } catch (IOException e) {
23. System.out.print("A");
24. } catch(FileNotFoundException e) {
25. System.out.print("B");
26. } finally {
27. System.out.print("C");
28. }
29. }
30. }

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

* 

**AC**

* 

**BC**

* 

**class Sub causes compilation error**

* 

**class Test causes compilation error**

**(Correct)**

**Explanation**

**UKOCP19357:**

Method m1() of Sub class correctly overrides the method m1() of Super class, so there is no compilation error in Sub class.

FileNotFoundException extends IOException and hence catch block of FileNotFoundException should appear before the catch block of IOException.

Therefore, class Test causes compilation error.

Bottom of Form

Top of Form

Question 69: **Correct**

**Question ID: UKOCP89662**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.io.FileNotFoundException;
4. import java.io.IOException;
6. abstract class Parent {
7. public abstract void find() throws IOException;
8. }
10. class Child extends Parent {
11. @Override
12. public void find() throws IOException {
13. throw new FileNotFoundException();
14. }
15. }
17. public class Test {
18. public static void main(String[] args) {
19. Parent p = new Child();
20. try {
21. p.find();
22. } catch (FileNotFoundException e) {
23. System.out.print("X");
24. } catch (IOException e) {
25. System.out.print("Y");
26. } finally {
27. System.out.print("Z");
28. }
29. }
30. }

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

* 

**XZ**

**(Correct)**

* 

**YZ**

* 

**XYZ**

* 

**Compilation Error**

**Explanation**

**UKOCP89662:**

Method find() of Child class correctly overrides the method find() of Parent class, so there is no compilation error in Child class.

Code in Test class doesn't cause any error as well.

java.io.FileNotFoundException extends java.io.IOException

and

java.io.IOException extends java.lang.Exception

Even though method find() declares to throw IOException but at runtime an instance of FileNotFoundException is thrown.

A catch handler for FileNotFoundException is available and hence X is printed on to the console.

After that finally block is executed, which prints Z to the console.

Bottom of Form

Top of Form

Question 70: **Incorrect**

**Question ID: UKOCP55437**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.io.FileNotFoundException;
4. import java.io.IOException;
6. abstract class Base {
7. public abstract void print() throws IOException;
8. }
10. class Derived extends Base {
11. @Override
12. public void print() throws IOException {
13. throw new FileNotFoundException();
14. }
15. }
17. public class Test {
18. public static void main(String[] args) {
19. Base b = new Derived();
20. try {
21. b.print();
22. } catch (FileNotFoundException e) {
23. System.out.print("AWE");
24. } finally {
25. System.out.print("SOME");
26. }
27. }
28. }

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

* 

**AWESOME**

**(Incorrect)**

* 

**SOME**

* 

**AWE**

* 

**Compilation error**

**(Correct)**

* 

**Program ends abruptly**

**Explanation**

**UKOCP55437:**

Method print() of Derived class correctly overrides the method print() of Base class, so there is no compilation error in Derived class.

java.io.FileNotFoundException extends java.io.IOException

and

java.io.IOException extends java.lang.Exception

Even though an instance of FileNotFoundException is thrown by method print() at runtime, but method print() declares to throw IOException.

Reference variable 'b' is of Base type and hence for compiler, call to b.print(); is to method print() of Base, which throws IOException.

And as IOException is checked exception hence calling code should handle it.

As calling code doesn't handle IOException or its supertype, so b.print(); causes compilation error.

Bottom of Form

Top of Form

Question 71: **Correct**

**Question ID: UKOCP58927**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. class Base {
4. public void log() throws NullPointerException {
5. System.out.println("Base: log()");
6. }
7. }
9. class Derived extends Base {
10. public void log() throws RuntimeException {
11. System.out.println("Derived: log()");
12. }
13. }
15. public class Test {
16. public static void main(String[] args) {
17. Base obj = new Derived();
18. obj.log();
19. }
20. }

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

* 

**Base: log()**

* 

**Derived: log()**

**(Correct)**

* 

**Compilation error in Derived class**

* 

**Compilation error in Test class**

**Explanation**

**UKOCP58927:**

NullPointerException extends RuntimeException. Overriding method may or may not throw any RuntimeException. Only thing to remember is that if overridden method throws any unchecked exception or Error, then overriding method must not throw any checked exceptions.

So, method log() in Derived class correctly overrides Base class's method.

Rest is simple polymorphism. 'obj' refers to an instance of Derived class and hence obj.log(); invokes method log() of Derived class, which prints "Derived: log()" on to the console.

Bottom of Form

Top of Form

Question 72: **Correct**

**Question ID: UKOCP51079**

Consider the following interface declaration:

1. public interface I1 {
2. void m1() throws java.io.IOException;
3. }

Which of the following incorrectly implements interface I1?

* 
  1. public class C1 implements I1 {
  2. public void m1() {}
  3. }
* 
  1. public class C2 implements I1 {
  2. public void m1() throws java.io.FileNotFoundException{}
  3. }
* 
  1. public class C3 implements I1 {
  2. public void m1() throws java.io.IOException{}
  3. }
* 
  1. public class C4 implements I1 {
  2. public void m1() throws Exception{}
  3. }

**(Correct)**

**Explanation**

**UKOCP51079:**

NOTE: Question is asking for "incorrect" implementation and not "correct" implementation.

java.io.FileNotFoundException extends java.io.IOException

and

java.io.IOException extends java.lang.Exception

According to overriding rules, if super class / interface method declares to throw a checked exception, then overriding method of sub class / implementer class has following options:

1. May not declare to throw any checked exception.

2. May declare to throw the same checked exception thrown by super class / interface method.

3. May declare to throw the sub class of the exception thrown by super class / interface method.

4. Cannot declare to throw the super class of the exception thrown by super class / interface method.

5. Cannot declare to throw unrelated checked exception.

6. May declare to throw any RuntimeException or Error.

Based on above rules, class C4 incorrectly implements method m1() of interface I1 as it declares to throw Exception, which is super class of java.io.IOException.

Bottom of Form

Top of Form

Question 73: **Correct**

**Question ID: UKOCP54104**

Given Code:

1. import java.io.\*;
3. class ReadTheFile {
4. static void print() { //Line n1
5. throw new IOException(); //Line n2
6. }
7. }
9. public class Test {
10. public static void main(String[] args) { //Line n3
11. ReadTheFile.print(); //Line n4
12. }
13. }

Which 2 changes are necessary so that code compiles successfully?

* 

**Replace Line n1 with static void print() throws Exception {**

**(Correct)**

* 

**Replace Line n1 with static void print() throws Throwable {**

* 

**Replace Line n3 with public static void main(String[] args) throws IOException {**

* 

**Surround Line n4 with below try-catch block:**

* 1. try {
  2. ReadTheFile.print();
  3. } catch(IOException e) {
  4. e.printStackTrace();
  5. }
* 

**Surround Line n4 with below try-catch block:**

* 1. try {
  2. ReadTheFile.print();
  3. } catch(IOException | Exception e) {
  4. e.printStackTrace();
  5. }
* 

**Surround Line n4 with below try-catch block:**

* 1. try {
  2. ReadTheFile.print();
  3. } catch(Exception e) {
  4. e.printStackTrace();
  5. }

**(Correct)**

**Explanation**

**UKOCP54104:**

This question is tricky as 2 changes are related and not independent. Let's first check the reason for compilation error. Line n2 throws a checked exception, IOException but it is not declared in the throws clause. So, print method should have throws clause for IOException or the classes in top hierarchy such as Exception or Throwable.

Based on this deduction, Line n1 can be replaced with either "static void print() throws Exception {" or "static void print() throws Throwable" but we will have to select one out of these as after replacing Line n1, Line n4 will start giving error as we are not handling the checked exception at Line n4.

This part is easy, do we have other options, which mention "Throwable"? NO. Then mark the first option as "Replace Line n1 with static void print() throws Exception {".

As, print() method throws Exception, so main method should handle Exception or its super type and not it's subtype. Two options working only with IOException can be ruled out.

Multi-catch statement "catch(IOException | Exception e)" causes compilation error as IOException and Exception are related to each other in multilevel inheritance. So you are left with only one option to pair with the 1st choice:

Surround Line n4 with below try-catch block:

1. try {
2. ReadTheFile.print();
3. } catch(Exception e) {
4. e.printStackTrace();
5. }

Bottom of Form

Top of Form

Question 74: **Correct**

**Question ID: UKOCP77207**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.io.FileNotFoundException;
5. public class Test {
6. static String [] names = {"Williamson.pdf", "Finch.pdf", "Kohli.pdf", "Morgan.pdf"};
7. public static void main(String[] args) {
8. try {
9. if (search("virat.pdf"))
10. System.out.println("FOUND");
12. } catch(FileNotFoundException ex) {
13. System.out.println("NOT FOUND");
14. }
15. }
17. private static boolean search(String name) throws FileNotFoundException {
18. for(int i = 0; i <= 4; i++) {
19. if (names[i].equalsIgnoreCase(name)) {
20. return true;
21. }
22. }
23. throw new FileNotFoundException();
24. }
25. }

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

* 

**FOUND**

* 

**NOT FOUND**

* 

**Compilation error**

* 

**None of the other options**

**(Correct)**

**Explanation**

**UKOCP77207:**

search(String) method declares to throw FileNotFoundException, which is a checked exception. It returns true if match is found otherwise it throws an instance of FileNotFoundException.

main(String[]) provides try-catch block around `search("virat.pdf")` and catch handler checks for FileNotFoundException. Given code compiles successfully.

There are 4 elements in 'names' array, so starting index is 0 and end index is 3, but given for loop goes till index number 4.

As search string is "virat.pdf" (not present in names array), hence for loop will execute for i = 0, 1, 2, 3, 4.

For i = 4, `names[i].equalsIgnoreCase(name)` throws ArrayIndexOutOfBoundsException (it is a RuntimeException). main(String []) method doesn't provide handler for ArrayIndexOutOfBoundsException and therefore stack trace is printed on to the console and program terminates abruptly.

Bottom of Form

Top of Form

Question 75: **Correct**

**Question ID: UKOCP75022**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. try {
6. play();
7. return;
8. } catch(Exception ex) {
9. System.out.println(ex.getMessage());
10. return;
11. } finally {
12. System.out.println("MATCH ABANDONED");
13. }
14. System.out.println("DONE");
15. }
17. static void play() throws Exception {
18. throw new Exception("INJURED");
19. }
20. }

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

* 

**INJURED**

**MATCH ABANDONED**

* 

**INJURED**

**MATCH ABANDONED**

**DONE**

* 

**MATCH ABANDONED**

* 

**INJURED**

* 

**INJURED**

**DONE**

* 

**MATCH ABANDONED**

**DONE**

* 

**Compilation error**

**(Correct)**

**Explanation**

**UKOCP75022:**

Both try and catch blocks have return; statement, which means either of the return statements will definitely get executed. Hence, compiler tags `System.out.println("DONE");` as unreachable and this causes compilation error.

Bottom of Form

Top of Form

Question 76: **Incorrect**

**Question ID: UKOCP29169**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.io.FileNotFoundException;
5. public class Test {
6. public static void main(String[] args) {
7. try {
8. findSecretFile();
9. } catch(Throwable ex) {
10. System.out.println(ex.getMessage());
11. return;
12. } finally {
13. System.out.println("LEVEL 1");
14. }
15. System.out.println("DONE");
16. }
18. static void findSecretFile() throws FileNotFoundException {
19. throw new FileNotFoundException("ACCESS REQUIRED");
20. }
21. }

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

* 

**ACCESS REQUIRED**

**LEVEL 1**

**DONE**

**(Incorrect)**

* 

**ACCESS REQUIRED**

**LEVEL 1**

**(Correct)**

* 

**ACCESS REQUIRED**

* 

**ACCESS REQUIRED**

**DONE**

* 

**Compilation error**

**Explanation**

**UKOCP29169:**

findSecretFile() method declares to throw FileNotFoundException (checked exception) and main(String []) method invokes findSecretFile(); method.

catch-handler in this case can specify FileNotFoundException or IOException or Exception or Throwable. As catch-handler for Throwable is available, hence exception is handled correctly inside main(String []) method.

As, return; statement is available inside catch-block only, therefore compiler doesn't tag `System.out.println("DONE");` as unreachable.

If return; statement was present in both try and catch block or finally-block, then unreachable code compilation error would have caused by the last statement in main(String []) method.

Throwable is the root class of the exception hierarchy and it contains some useful constructors:

1. public Throwable() {...} : No-argument constructor

2. public Throwable(String message) {...} : Pass the detail message

3. public Throwable(String message, Throwable cause) {...} : Pass the detail message and the cause

4. public Throwable(Throwable cause) {...} : Pass the cause

Exception and RuntimeException classes also provide similar constructors.

Throwable class also contains methods, which are inherited by all the subclasses (Exception, RuntimeException etc.)

1. public String getMessage() {...} : Returns the detail message (E.g. detail message set by 2nd and 3rd constructor)

2. public String toString() {} :

Returns a short description of this throwable. The result is the concatenation of:

the name of the class of this object

": " (a colon and a space)

the result of invoking this object's getLocalizedMessage() method

If getLocalizedMessage returns null, then just the class name is returned.

On execution, main(String []) method invokes findSecretFile(), which throws an instance of FileNotFoundException.

There is a matching catch-handler available in main(String []) method, therefore `System.out.println(ex.getMessage());` is executed and this prints ACCESS REQUIRED on to the console.

As there is finally-block available, so just before the return; statement takes control out of main(String []) method, code inside finally-block gets executed and this prints LEVEL 1 on to the console.

Control exits the main(String []) method and program terminates successfully.

Bottom of Form

Top of Form

Question 77: **Correct**

**Question ID: UKOCP18033**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.io.FileNotFoundException;
4. import java.io.IOException;
6. public class Test {
7. public static void main(String[] args) {
8. try {
9. find();
10. } catch(Exception ex) {
11. System.out.println(ex.getMessage());
12. }
13. }
15. static void find() throws Exception {
16. try {
17. System.out.print(1);
18. throw new FileNotFoundException("FNF");
19. } catch(FileNotFoundException ex) {
20. System.out.print(2);
21. throw new IOException("IO");
22. } catch(IOException ex) {
23. System.out.print(3);
24. throw new Exception("EXP");
25. } finally {
26. System.out.print(4);
27. throw new Exception("FINALLY");
28. }
29. }
30. }

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

* 

**Compilation error**

* 

**1234FINALLY**

* 

**124FINALLY**

**(Correct)**

* 

**14FINALLY**

* 

**12IO**

* 

**124IO**

* 

**14FNF**

**Explanation**

**UKOCP18033:**

Method find() declares to throw Exception and the throw statements inside this method is throwing the Subclasses of Exception, hence no issues in find() method.

main(String []) method also provide legal try-catch block, hence the given code compiles successfully.

Throwable is the root class of the exception hierarchy and it contains some useful constructors:

1. public Throwable() {...} : No-argument constructor

2. public Throwable(String message) {...} : Pass the detail message

3. public Throwable(String message, Throwable cause) {...} : Pass the detail message and the cause

4. public Throwable(Throwable cause) {...} : Pass the cause

Exception and RuntimeException classes also provide similar constructors.

Throwable class also contains methods, which are inherited by all the subclasses (Exception, RuntimeException etc.)

1. public String getMessage() {...} : Returns the detail message (E.g. detail message set by 2nd and 3rd constructor)

2. public String toString() {} :

Returns a short description of this throwable. The result is the concatenation of:

the name of the class of this object

": " (a colon and a space)

the result of invoking this object's getLocalizedMessage() method

If getLocalizedMessage returns null, then just the class name is returned.

Let's check the execution:

main(String []) method invokes find() and first statement inside try-block gets executed, 1 is printed on to the console.

Next statement is executed and an instance of FileNotFoundException is thrown.

Matching catch-handler is available, so control goes inside the first catch-handler and 2 is printed on to the console. An instance of IOException is thrown by the catch-block but another catch-block at the same level are not executed.

As finally-block always executes, hence, 4 is printed on to the console and an Exception instance is thrown by the finally-block.

Control goes back to the calling method main(String []), catch-handler is executed and it prints FINALLY on to the console.

Hence output is: 124FINALLY

Bottom of Form

Top of Form

Question 78: **Correct**

**Question ID: UKOCP23863**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. public class Test {
4. public static void main(String[] args) {
5. try {
6. check();
7. } catch(RuntimeException e) {
8. System.out.println(e.getClass().getName()); //Line n1
9. }
10. }
12. private static void check() {
13. try {
14. RuntimeException re = new RuntimeException(); //Line n2
15. throw re; //Line n3
16. } catch(RuntimeException e) {
17. System.out.println(1);
18. ArithmeticException ex = (ArithmeticException)e; //Line n4
19. System.out.println(2);
20. throw ex;
21. }
22. }
23. }

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

* 

**1**

**2**

**java.lang.RuntimeException**

* 

**1**

**2**

**java.lang.ArithmeticException**

* 

**1**

**java.lang.ArithmeticException**

* 

**1**

**java.lang.RuntimeException**

* 

**1**

**java.lang.ClassCastException**

**(Correct)**

**Explanation**

**UKOCP23863:**

Line n3 throws an instance of RuntimeException. As catch(RuntimeException e) is available, hence control starts executing catch-block inside check() method.

1 is printed on to the console.

At Line n4, instance of super-class (RuntimeException) is type-casted to sub-class (ArithmeticException), hence Line n4 throws an instance of ClassCastException.

ClassCastException is a sub-class of RuntimeException, so catch-block of main method is executed and Line n1 prints the fully qualified name of ClassCastException. java.lang.ClassCastException is printed on to the console.

Bottom of Form

Top of Form

Question 79: **Correct**

**Question ID: UKOCP79860**

java.sql.SQLException extends java.lang.Exception

and

java.sql.SQLWarning extends java.sql.SQLException

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.sql.\*;
5. interface Multiplier {
6. void multiply(int... x) throws SQLException;
7. }
9. class Calculator implements Multiplier {
10. public void multiply(int... x) throws /\*INSERT\*/ {
12. }
13. }
14. public class Test {
15. public static void main(String[] args) {
16. try {
17. Multiplier obj = new Calculator(); //Line n1
18. obj.multiply(1, 2, 3);
19. } catch(SQLException e) {
20. System.out.println(e);
21. }
22. }
23. }

Which of the options can be used to replace /\*INSERT\*/ such that there is no compilation error?

Select 5 options.

* 

**java.io.IOException**

* 

**SQLException**

**(Correct)**

* 

**SQLWarning**

**(Correct)**

* 

**Throwable**

* 

**RuntimeException**

**(Correct)**

* 

**Error**

**(Correct)**

* 

**Exception**

* 

**NullPointerException**

**(Correct)**

**Explanation**

**UKOCP79860:**

At Line n1, reference variable 'obj' is of Multiplier type (supertype) and it refers to an instance of Calculator class (subtype). This is polymorphism and allowed in Java.

multiply(int...) method declared in Multiplier interface declares to throw SQLException, hence the catch handler for Line n1 should provide handler for SQLException or its supertype. As catch-handler for SQLException is available, therefore Test class compiles successfully.

According to overriding rules, if super class / interface method declares to throw a checked exception, then overriding method of sub class / implementer class has following options:

1. May not declare to throw any checked exception

2. May declare to throw the same checked exception thrown by super class / interface method: SQLException is a valid option.

3. May declare to throw the sub class of the exception thrown by super class / interface method: SQLWarning is a valid option.

4. Cannot declare to throw the super class of the exception thrown by super class / interface method: Exception, Throwable are not valid options.

5. Cannot declare to throw unrelated checked exception: java.io.IOException is not a valid option as it is not related java.sql.SQLException in multi-level inheritance.

6. May declare to throw any RuntimeException or Error: RuntimeException, NullPointerException and Error are valid options.

Therefore 5 options can successfully replace /\*INSERT\*/: SQLException, SQLWarning, RuntimeException, Error and NullPointerException

Bottom of Form

Top of Form

Question 80: **Incorrect**

**Question ID: UKOCP45634**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.io.FileNotFoundException;
5. interface Printer {
6. default void print() throws FileNotFoundException {
7. System.out.println("PRINTER");
8. }
9. }
11. class FilePrinter implements Printer {
12. public void print() { //Line n1
13. System.out.println("FILE PRINTER");
14. }
15. }
16. public class Test {
17. public static void main(String[] args) {
18. Printer p = new FilePrinter(); //Line n2
19. p.print(); //Line n3
20. var fp = new FilePrinter(); //Line n4
21. fp.print(); //Line n5
22. }
23. }

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

* 

**Compilation error at Line n1**

* 

**Compilation error at Line n2**

**(Incorrect)**

* 

**Compilation error at Line n3**

**(Correct)**

* 

**Compilation error at Line n4**

* 

**Compilation error at Line n5**

* 

**1122 is printed on to the console**

* 

**1111 is printed on to the console**

* 

**2211 is printed on to the console**

* 

**2222 is printed on to the console**

**Explanation**

**UKOCP45634:**

According to overriding rules, if super class / interface method declares to throw a checked exception, then overriding method of sub class / implementer class has following options:

1. May not declare to throw any checked exception.

2. May declare to throw the same checked exception thrown by super class / interface method.

3. May declare to throw the sub class of the exception thrown by super class / interface method.

4. Cannot declare to throw the super class of the exception thrown by super class / interface method.

5. Cannot declare to throw unrelated checked exception.

6. May declare to throw any RuntimeException or Error.

default methods were added in Java 8 and FilePrinter class correctly overrides the default method print() of Printer interface. Line n1 compiles successfully.

At Line n2, 'p' is of Printer type (supertype) and it refers to an instance of FilePrinter class (subtype), this is polymorphism and allowed in Java. Line n2 compiles successfully.

At Line n3, method print() is invoked on 'p' reference (Printer type) and as print() method defined in Printer interface declares to throw FileNotFoundException (checked exception). As main(String []) method doesn't declare to throw FileNotFoundException and also there is not try-catch block available, therefore Line n3 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

At Line n4, 'fp' infers to FilePrinter type.

At Line n5, method print() is invoked on 'fp' reference (FilePrinter type) and as print() method defined in FilePrinter class doesn't declare to throw any checked exception, hence Line n5 compiles successfully.

Only Line n3 causes compilation failure.

Bottom of Form

Top of Form

Question 81: **Correct**

**Question ID: UKOCP21203**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. interface Blogger {
4. default void blog() throws Exception {
5. System.out.println("GENERIC");
6. }
7. }
9. class TravelBlogger implements Blogger {
10. public void blog() {
11. System.out.println("TRAVEL");
12. }
13. }
15. public class Test {
16. public static void main(String[] args) {
17. Blogger blogger = new TravelBlogger(); //Line n1
18. ((TravelBlogger)blogger).blog(); //Line n2
19. }
20. }

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

* 

**Compilation error in TravelBlogger class**

* 

**Compilation error in Test class**

* 

**GENERIC is printed on to the console and program terminates successfully**

* 

**TRAVEL is printed on to the console and program terminates successfully**

**(Correct)**

* 

**An exception is thrown at runtime**

**Explanation**

**UKOCP21203:**

According to overriding rules, if super class / interface method declares to throw a checked exception, then overriding method of sub class / implementer class has following options:

1. May not declare to throw any checked exception.

2. May declare to throw the same checked exception thrown by super class / interface method.

3. May declare to throw the sub class of the exception thrown by super class / interface method.

4. Cannot declare to throw the super class of the exception thrown by super class / interface method.

5. Cannot declare to throw unrelated checked exception.

6. May declare to throw any RuntimeException or Error.

default methods were added in Java 8 and TravelBlogger class correctly overrides the default method blog() of Blogger interface. Blogger interface compiles successfully.

At Line n1, 'blogger' is of Blogger type (supertype) and it refers to an instance of TravelBlogger class (subtype), this is polymorphism and allowed in Java. Line n1 compiles successfully.

At Line n2, blog() method is being invoked on typecasting 'blogger' to TravelBlogger and as TravelBlogger class doesn't declare to throw any checked exception, hence Line n2 compiles successfully.

As instance is of TravelBlogger type, therefore on execution, Line n2 invokes blog() method of TravelBlogger instance, which prints TRAVEL on to the console.

Bottom of Form

Top of Form

Question 82: **Correct**

**Question ID: UKOCP87476**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. abstract class Animal {
4. abstract void jump() throws RuntimeException;
5. }
7. class Deer extends Animal {
8. void jump() { //Line n1
9. System.out.println("DEER JUMPS");
10. }
12. void jump(int i) {
13. System.out.println("DEER JUMPS TO " + i + " FEET");
14. }
15. }
17. public class Test {
18. public static void main(String[] args) {
19. Animal animal = new Deer();
20. ((Deer)animal).jump(); //Line n2
21. ((Deer)animal).jump(5); //Line n3
22. }
23. }

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

* 

**Line n1 causes compilation error**

* 

**Line n2 causes compilation error**

* 

**Line n3 causes compilation error**

* 

**An exception is thrown at runtime**

* 

**Test class executes successfully and prints:**

**DEER JUMPS**

**DEER JUMPS TO 5 FEET**

**(Correct)**

**Explanation**

**UKOCP87476:**

Method jump() in Animal class declares to throw RuntimeException.

Overriding method may or may not throw any RuntimeException. Only thing to remember is that if overridden method throws any unchecked exception or Error, then overriding method must not throw any checked exceptions. Line n1 compiles successfully as it correctly overrides the jump() method of Animal class.

Class Deer also provides overloaded jump(int) method.

Inside main(String []) method, reference variable 'animal' is of Animal class (supertype) and it refers to an instance of Deer class (subtype), this is polymorphism and allowed in Java.

As instance is of Deer class, hence 'animal' reference can easily be casted to Deer type. Line n2 and Line n3 compiles successfully and on execution prints below on to the console:

DEER JUMPS

DEER JUMPS TO 5 FEET

Bottom of Form

Top of Form

Question 83: **Correct**

**Question ID: UKOCP88324**

Consider below code fragment:

1. import java.util.\*;
3. class A{}
4. class B extends A{}
6. abstract class Super {
7. abstract List<A> get() throws IndexOutOfBoundsException;
8. }
10. abstract class Sub extends Super {
11. /\*INSERT\*/
12. }

Which of the following options replaces /\*INSERT\*/ such that there is no compilation error?

* 

**abstract List<A> get() throws ArrayIndexOutOfBoundsException;**

**(Correct)**

* 

**abstract List<B> get();**

* 

**abstract ArrayList<A> get() throws Exception;**

* 

**abstract ArrayList<B> get();**

**Explanation**

**UKOCP88324:**

Few things to keep in mind:

1.

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

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

B. 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).

2.

In case of overriding, if overridden method declares to throw any RuntimeException or Error, overriding method may or may not throw any RuntimeException but overriding method must not throw any checked exceptions.

3.

In generics syntax, Parameterized types are not polymorphic, this means even if B is subtype of A, List<B> is not subtype of List<A>. Remember this point. So below syntaxes are NOT allowed:

List<A> list = new ArrayList<B>(); OR ArrayList<A> list = new ArrayList<B>();

Let's check all the options one by one:

abstract List<A> get() throws ArrayIndexOutOfBoundsException; => ✓ It returns the same return type 'List<A>' and it is allowed to throw any RuntimeException (ArrayIndexOutOfBoundsException is RuntimeException)

abstract List<B> get(); => ✗ List<B> is not subtype of List<A>, it is not covariant return type.

abstract ArrayList<A> get() throws Exception; => ✗ As overridden method declares to throw IndexOutOfBoundsException, which is a Runtime Exception, overriding method is not allowed to declare to throw any checked Exception. Class Exception and its subclasses are checked exceptions.

abstract ArrayList<B> get(); => ✗ ArrayList<B> is not subtype of List<A>, it is not covariant return type.

Bottom of Form

Top of Form

Question 84: **Correct**

**Question ID: UKOCP64379**

Consider below code snippets available in the same package:

1. abstract class Traveller {
2. void travel(String place){}
3. }
5. abstract class BeachTraveller extends Traveller {
6. /\*INSERT\*/
7. }

Which of the following declarations/definitions can replace /\*INSERT\*/ such that there is no compilation error?

Select 5 options.

* 

**abstract void travel();**

**(Correct)**

* 

**abstract void travel(String beach);**

**(Correct)**

* 

**public abstract void travel();**

**(Correct)**

* 

**public void travel() throws RuntimeException {}**

**(Correct)**

* 

**public void travel(String beach) throws Exception {}**

* 

**void travel(String beach) throws java.io.IOException {}**

* 

**public void travel(Object obj) {}**

**(Correct)**

**Explanation**

**UKOCP64379:**

Both Traveller and BeachTraveller are abstract classes and BeachTraveller extends Traveller. It is possible to have abstract class without any abstract method. Code as is compiles successfully as BeachTraveller inherits travel(String) method of Traveller class.

But as per the question, /\*INSERT\*/ must be replaced such that there is no compilation error.

Let's check all the options one by one:

abstract void travel();

✓ This is method overloading. BeachTraveller has 2 methods: `void travel(String){}` and `abstract void travel()`.

abstract void travel(String beach);

✓ As BeachTraveller is abstract, hence travel(String) method can be declared abstract.

public abstract void travel();

✓ This is method overloading. BeachTraveller has 2 methods: `void travel(String){}` and `abstract void travel()`.

public void travel() throws RuntimeException {}

✓ This is method overloading. BeachTraveller has 2 methods: `void travel(String){}` and `public void travel() throws RuntimeException {}`.

public void travel(String beach) throws Exception {}

✗ As overridden method doesn't declare to throw any checked Exception hence overriding method is not allowed to declare to throw Exception.

void travel(String beach) throws java.io.IOException {}

✗ As overridden method doesn't declare to throw any checked Exception hence overriding method is not allowed to declare to throw java.io.IOException.

public void travel(Object obj) {}

✓ This is method overloading. BeachTraveller has 2 methods: `void travel(String){}` and `public void travel(Object){}`.

Bottom of Form

Top of Form

Question 85: **Incorrect**

**Question ID: UKOCP47808**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.io.FileNotFoundException;
4. import java.io.IOException;
6. class Base {
7. Base() throws IOException {
8. System.out.print(1);
9. }
10. }
12. class Derived extends Base {
13. Derived() throws FileNotFoundException {
14. System.out.print(2);
15. }
16. }
18. public class Test {
19. public static void main(String[] args) throws Exception {
20. new Derived();
21. }
22. }

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

* 

**Compilation error in both Base and Derived classes**

**(Incorrect)**

* 

**Compilation error only in Base class**

* 

**Compilation error only in Derived class**

**(Correct)**

* 

**Test class executes successfully and prints 12 on to the console**

* 

**Test class executes successfully and prints 21 on to the console**

**Explanation**

**UKOCP47808:**

It is legal for the constructors to have throws clause.

Constructors are not inherited by the Derived class so there is no method overriding rules related to the constructors but as one constructor invokes other constructors implicitly or explicitly by using this(...) or super(...), hence exception handling becomes interesting.

Java compiler adds super(); as the first statement inside Derived class's constructor:

1. Derived() throws FileNotFoundException {
2. super(); //added by the compiler
3. System.out.print(2);
4. }

As super(); invokes the constructor of Base class (which declares to throw IOException), compiler complains as Derived class no-argument constructor doesn't declare to throw IOException. It declares to throw FileNotFoundException (subclass of IOException), which is not enough for the instances of IOException.

Bottom of Form

Top of Form

Question 86: **Correct**

**Question ID: UKOCP18038**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.io.IOException;
5. class Parent {
6. Parent() throws IOException {
7. System.out.print("HAKUNA");
8. }
9. }
11. class Child extends Parent {
12. Child() throws Exception {
13. System.out.println("MATATA");
14. }
15. }
17. public class Test {
18. public static void main(String[] args) throws Exception {
19. new Child();
20. }
21. }

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

* 

**Compilation error in both Parent and Child classes**

* 

**Compilation error only in Parent class**

* 

**Compilation error only in Child class**

* 

**Test class executes successfully and prints HAKUNAMATATA on to the console**

**(Correct)**

* 

**Test class executes successfully and prints MATATAHAKUNA on to the console**

**Explanation**

**UKOCP18038:**

It is legal for the constructors to have throws clause.

Constructors are not inherited by the Child class so there is no method overriding rules related to the constructors but as one constructor invokes other constructors implicitly or explicitly by using this(...) or super(...), hence exception handling becomes interesting.

Java compiler adds super(); as the first statement inside Child class's constructor:

1. Child() throws Exception {
2. super(); //added by the compiler
3. System.out.println("MATATA");
4. }

super(); invokes the constructor of Parent class (which declares to throw IOException), but as no-argument constructor of Child class declares to throw Exception (super class of IOException), hence IOException is also handled. There is no compilation error and output is: HAKUNAMATATA

Bottom of Form

Top of Form

Question 87: **Correct**

**Question ID: UKOCP42123**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. import java.io.IOException;
5. class Super {
6. Super() throws RuntimeException {
7. System.out.print("CARPE ");
8. }
9. }
11. class Sub extends Super {
12. Sub() throws IOException {
13. System.out.print("DIEM ");
14. }
15. }
17. public class Test {
18. public static void main(String[] args) throws Exception {
19. new Sub();
20. }
21. }

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

* 

**Compilation error in both Super and Sub classes**

* 

**Compilation error only in Super class**

* 

**Compilation error only in Sub class**

* 

**Test class executes successfully and prints CARPE DIEM on to the console**

**(Correct)**

* 

**Test class executes successfully and prints DIEM CARPE on to the console**

**Explanation**

**UKOCP42123:**

It is legal for the constructors to have throws clause.

Constructors are not inherited by the Sub class so there is no method overriding rules related to the constructors but as one constructor invokes other constructors implicitly or explicitly by using this(...) or super(...), hence exception handling becomes interesting.

Java compiler adds super(); as the first statement inside Sub class's constructor:

1. Sub() throws IOException {
2. super(); //added by the compiler
3. System.out.println("DIEM");
4. }

super(); invokes the constructor of Super class (which declares to throw RuntimeException), as RuntimeException is unchecked exception, therefore no handling is necessary in the constructor of Sub class.

Sub class's constructor declares to throw IOException but main(String []) method handles it.

There is no compilation error and output is: CARPE DIEM

Bottom of Form

Top of Form

Question 88: **Correct**

**Question ID: UKOCP88328**

Given:

1. class TestException extends Exception {
2. public TestException() {
3. super();
4. }
6. public TestException(String s) {
7. super(s);
8. }
9. }
11. public class Test {
12. public void m1() throws \_\_\_\_\_\_\_\_\_\_ {
13. throw new TestException();
14. }
15. }

For the above code, fill in the blank with one option.

* 

**Exception**

**(Correct)**

* 

**Object**

* 

**RuntimeException**

* 

**Error**

**Explanation**

**UKOCP88328:**

Method m1() throws an instance of TestException, which is a checked exception as it extends Exception class.

So in throws clause we must provide:

1. Checked exception.

2. Exception of TestException type or it's super types (Exception, Throwable), Object cannot be used in throws clause.

Out of the given options only Exception can be filled in the blank space.

Bottom of Form

Top of Form

Question 89: **Incorrect**

**Question ID: UKOCP30141**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. class MyException1 extends RuntimeException {}
5. class MyException2 extends RuntimeException {}
7. public class Test {
8. private static void m() {
9. try {
10. throw new RuntimeException();
11. } catch(RuntimeException ex) {
12. throw new MyException1();
13. } finally {
14. throw new MyException2();
15. }
16. }
18. public static void main(String[] args) {
19. try {
20. m();
21. } catch(MyException1 e) {
22. System.out.println("MyException1");
23. } catch(MyException2 e) {
24. System.out.println("MyException2");
25. } catch (RuntimeException e) {
26. System.out.println("RuntimeException");
27. }
28. }
29. }

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

* 

**MyException1**

**(Incorrect)**

* 

**MyException2**

**(Correct)**

* 

**RuntimeException**

**Explanation**

**UKOCP30141:**

If finally block throws exception, then exception thrown by try or catch block is ignored.

In this case, method m() throws an instance of MyException2 class.

Bottom of Form

Top of Form

Question 90: **Correct**

**Question ID: UKOCP17187**

Given code:

1. package com.udayankhattry.ocp;
3. class MyException extends RuntimeException {
4. public void log() {
5. System.out.println("Logging MyException");
6. }
7. }
9. class YourException extends RuntimeException {
10. public void log() {
11. System.out.println("Logging YourException");
12. }
13. }
15. public class Test {
16. public static void main(String[] args) {
17. try {
18. throw new MyException();
19. } catch(MyException | YourException ex){
20. ex.log();
21. }
22. }
23. }

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

* 

**Logging MyException**

* 

**Logging YourException**

* 

**Compilation error**

**(Correct)**

* 

**Runtime Exception**

**Explanation**

**UKOCP17187:**

In a multi-catch block, type of reference variable (ex in this case) is the common base class of all the Exception types mentioned, 'MyException' and 'YourException'.

This means ex is of RuntimeException. Method log() is not available in RuntimeException or its super classes.

ex.log(); causes compilation failure.

Bottom of Form

Top of Form

Question 91: **Incorrect**

**Question ID: UKOCP41274**

1. package com.udayankhattry.ocp;
3. class MyException extends RuntimeException {}
5. class YourException extends RuntimeException {}
7. public class Test {
8. public static void main(String[] args) {
9. try {
10. throw new YourException();
11. } catch(MyException e1 | YourException e2){
12. System.out.println("Caught");
13. }
14. }
15. }

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

* 

**Compilation error**

**(Correct)**

* 

**Caught**

**(Incorrect)**

* 

**Runtime Exception**

**Explanation**

**UKOCP41274:**

Wrong syntax for multi-catch block, only one reference variable is allowed.

Correct syntax of multi-catch statement is: 'catch(MyException | YourException e)'

Bottom of Form

Top of Form

Question 92: **Correct**

**Question ID: UKOCP33652**

Given code of Test.java file:

1. package com.udayankhattry.ocp;
3. class MyException extends RuntimeException {}
5. class YourException extends RuntimeException {}
7. public class Test {
8. public static void main(String[] args) {
9. try {
10. throw new YourException();
11. } catch(MyException | YourException e){
12. e = null;
13. }
14. }
15. }

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

* 

**Compilation error**

**(Correct)**

* 

**Runtime Exception**

* 

**Nothing is printed on to the console and program terminates successfully**

**Explanation**

**UKOCP33652:**

Variable 'e' used in multi-catch block is implicitly final and can't be re-initialized.

e = null; causes compilation failure.

Bottom of Form