Justificación de Respuestas en APX con Java

1. Which three are bad practices?

* 1/1

a. Checking for an IOException and ensuring that the program can recover if one occurs.

b. Checking for ArrayIndexOutOfBoundsException and ensuring that the program can recover if one occurs.

c. Checking for FileNotFoundException to inform a user that a filename entered is not valid.

d. Checking for Error and, if necessary, restarting the program to ensure that users are unaware problems.

e. Checking for ArrayIndexOutOfBoundsException when iterating through an array to determine when all elements.-have been visited.

Justificación:

Estas opciones se consideran buenas prácticas:

- La opción **a** es una buena práctica cuando se trata de operaciones de entrada y salida, como leer y escribir. Esto permite que el programa gestione los errores ofallos de manera adecuada.

- La opción **c** es útil y es una buena práctica porque permite informar al usuario que el nombre del archivo ingresado no es válido

Estas opciones se consideran malas prácticas:

- La opción **b** es una mala práctica ya que es mejor prevenir este el error **ArrayIndexOutOfBoundsException** que intentar recuperarse de él.

- La opción **d** se considera una mala práctica porque reiniciar el programa al encontrar un error y ocultar los problemas graves en lugar de solucionarlos puedeenmascarar problemas serios y llevar a un comportamiento del sistema inestable.

- La opción **e** implica usar excepciones para controlar iteraciones, lo cual es ineficiente y no es el propósito para el cual están diseñadas las excepciones. Poreso se considera una mala práctica.

2. Indica a JUnit que la propiedad que usa esta anotación es una simulación y, por lo tanto, se inicializa como tal y es susceptible de ser inyectada por @InjectMocks.

* 1/1

a.Mockito
b. Mock
c. Inject
d. InjectMock

Justificación:

La anotación @Mock cumple con la sentencia anteriormente descrita. Además, las opciones **a** y **d** no existen, mientras que la opción **c** se usa en otros marcos de inyección de dependencias, pero no crea mocks.

3. What is the result?

*1/1

```
Given

public static void main(String[] args){
      int[][] array2D = {{0,1,2}, {3,4,5,6}};
      System.out.print(array2D[0].length + "");
      System.out.print(array2D[1].getClass().isArray() + "");
      System.out.print(array2D[0][1]);
}
What is the result?
```

a. 3false3
b. 3false1
c. 2false1
d. 3true1
e. 2true3

Justificación:

El código imprime tres valores consecutivos. Primero, la longitud del primer array que es 3. Luego, verifica si el segundo array es efectivamente un array, lo cual es cierto, por lo que imprime "true". Finalmente, accede al segundo elemento del primer array, que es 1. Así, el resultado completo impreso en la consola es "3true1".

4. Which two statments are true?

*1/1

a. An interface CANNOT be extended by another interface.
b. An abstract class can be extended by a concrete class.

c. An abstract class CANNOT be extended by an abstract class.
d. An interface can be extended by an abstract class.
e.An abstract class can implement an interface.
f. An abstract class can be extended by an interface.

Justificación:

Las opciones b y e son correctas porque una clase abstracta puede ser heredada por una clase concreta, lo que permite que la clase concreta implemente los métodos abstractos. Además, una clase abstracta puede implementar una interfaz, lo que significa que puede declarar y cumplir los métodos definidos en la interfaz.
Las otras afirmaciones son incorrectas. Una interfaz puede ser extendida por otra interfaz, y una clase abstracta puede ser extendida por otra clase abstracta. Sin embargo, una clase abstracta no puede ser extendida por una interfaz y una interfaz no puede extender una clase abstracta.

5. What is the result?

*1/1

```
Given:

class Alpha{ String getType(){ return "alpha";}}
class Beta extends Alpha{String getType(){ return "beta";}}
public class Gamma  extends Beta { String getType(){ return "gamma";}
      public static void main(String[] args) {
        Gamma g1 = (Gamma) new Alpha();
        Gamma g2 = (Gamma) new Beta();
        System.out.print(g1.getType()+ " " +g2.getType());
    }
}
What is the result?
```

a. Gamma gamma
b. Beta beta
c. Alpha beta
d. Compilation fails

Justificación:

El programa intenta convertir objetos de las clases Alpha y Beta a la clase Gamma. Sin embargo, como Alpha no puede ser convertido directamente a Gamma, esto causa un error en la compilación. Lo mismo ocurre con Beta y Gamma.

6. Which five methods, inserted independiently at line 5, will compile? (Choose five)

*1/1

```
1 public class Blip{
2       protected int blipvert(int x){ return 0
3 }
4 class Vert extends Blip{
5       //insert code here
6 }
```

Private int blipvert(long x) { return 0; }
Protected int blipvert(long x) { return 0; }
Protected long blipvert(int x, int y) { return 0; }
Public int blipvert(int x) { return 0; }

Private int blipvert(int x) { return 0; }
Protected long blipvert(int x) { return 0; }
Protected long blipvert(long x) { return 0; }

Justificación:

Para que los métodos en la línea 5 compilen, deben cumplir con reglas de sobrecarga y anulación. Los cinco métodos que siguen estas reglas son:

1. **Public int blipvert(int x)**: Anula el método de la clase base con mayor visibilidad.
2. **Protected int blipvert(long x)**: Sobrecarga con un parámetro de tipo diferente.
3. **Private int blipvert(long x)**: Sobrecarga y cambia la visibilidad a "private".
4. **Protected long blipvert(int x, int y)**: Sobrecarga añadiendo un parámetro adicional.
5. **Protected long blipvert(long x)**: Sobrecarga con un tipo de parámetro y tipo de retorno diferente.

7. Pregunta

*0/1

```
Given:

1. class Super{
2.    private int a;
3.    protected Super(int a){ this.a = a; }
4. }
...
11. class Sub extends Super{
12.   public Sub(int a){ super(a);}
13.   public Sub(){ this.a = 5;}
14. }
Which two independently, will allow Sub to compile? (Choose two)
```

Change line 2 to: public int a;
Change line 13 to: public Sub(){ super(5);}
Change line 2 to: protected int a;

Change line 13 to: public Sub(){ this(5);}

Change line 13 to: public Sub(){ super(a);}

Respuestas correctas

Change line 13 to: public Sub(){ super(5);}
Change line 13 to: public Sub(){ this(5);}

Justificación:

- Cambiar la línea 13 a public Sub() { super(5); }: Esta línea es correcta porque llama al constructor de la clase Super y le pasa el valor 5, lo que inicializa la variable **a** correctamente en la clase padre.
- Cambiar la línea 13 a public Sub() { this(5); }: Esta línea es correcta porque llama al otro constructor de la clase Sub que toma un argumento int a. Ese constructor ya maneja la inicialización de a correctamente al llamar al constructor de la clase Super.

Estas opciones aseguran que la variable **a** se inicialice adecuadamente en la clase Super, permitiendo que la clase Sub compile sin problemas.

8. Whats is true about the class Wow?

*1/1

```
public abstract class Wow {
        private int wow;
        public Wow(int wow) { this.wow = wow; }
        public void wow() { }
        private void wowza() { }
}
```

It compiles without error.
It does not compile because an abstract class cannot have private methods
It does not compile because an abstract class cannot have instance variables.
It does not compile because an abstract class must have at least one abstract method.
It does not compile because an abstract class must have a constructor with no arguments.

Justificación:

La clase abstracta Wow compila sin errores porque está correctamente definida. Una clase abstracta puede tener métodos y variables privadas, así como métodos concretos y constructores con argumentos. No es necesario que tenga métodos abstractos ni constructores sin argumentos.

9. What is the result?

*1/1

```
class Atom {
        Atom() { System.out.print("atom "); }
}
class Rock extends Atom {
        Rock(String type) { System.out.print(type); }
}
public class Mountain extends Rock {
        Mountain() {
                super("granite ");
                new Rock("granite ");
        }
        public static void main(String[] a)   { new Mountain(); }
}
```

Compilation fails.
Atom granite.
Granite granite.
Atom granite granite.
An exception is thrown at runtime.
Atom granite atom granite.

Justificación:

Primero, se crea una instancia de Mountain. Al hacer esto, se llama al constructor de Mountain, que a su vez llama al constructor de Rock con el argumento "granite". Antes de que el constructor de Rock se ejecute, se llama al constructor de Atom, que imprime "atom". Luego, el constructor de Rock imprime "granite". Después, el constructor de Mountain crea una nueva instancia de Rock con "granite", repitiendo el proceso anterior e imprimiendo de nuevo "atom" y "granite".
Por eso, la secuencia final es "atom granite atom granite".

10. What is printed out when the program is excuted?

*1/1

```
public class MainMethod {
        void main() {
                System.out.println("one");
        }
        static void main(String args) {
                System.out.println("two");
        }
        public static final void main(String[] args) {
                System.out.println("three");
        }
        void mina(Object[] args) {
                System.out.println("four");
        }
}
```

a. one
b. two

c. three
d. four
e. There is no output.

Justificación:

Al ejecutar este código, se imprimirá "three". Esto se debe a que el método main que es ejecutado por la JVM tiene la firma public static void main(String[] args). Aunque hay otros métodos main definidos con diferentes firmas, solo este método específico es reconocido como el punto de entrada del programa. Por lo tanto, solo se imprime "three".

11. What is the result?

*0/1

```
class Feline {
        public String type = "f ";
        public Feline() {
                System.out.print("feline ");
        }
}
public class Cougar extends Feline {
        public Cougar() {
                System.out.print("cougar ");
        }
        void go() {
                type = "c ";
                System.out.print(this.type + super.type);
        }
        public static void main(String[] args) {
                new Cougar().go();
        }
}
```

a. Cougar c f.
b. Feline cougar c c.
c. Feline cougar c f.
d. Compilation fails.

Respuesta correcta

Feline cougar c c.

Justificación:

*1/1

El código imprime "feline cougar c c" porque:

1. El constructor de Feline se ejecuta primero e imprime "feline ".
2. Luego, el constructor de Cougar se ejecuta e imprime "cougar ".
3. Finalmente, el método go de Cougar cambia el valor de la variable type a "c " e imprime

esta nueva type seguida de la type de la clase base Feline, que ha sido sobrescrita en la instancia actual. Por lo tanto, ambos son "c ".

12. What is the result?

* 1/1

```
import java.util.*;
public class MyScan {
        public static void main(String[] args) {
                String in = "1 a 10 . 100 1000";
                Scanner s = new Scanner(in);
                int accum = 0;
                for (int x = 0; x < 4; x++) {
                        accum += s.nextInt();
                }
                System.out.println(accum);
        }
}
```

a. 11
b. 111
c. 1111
d. An exception is thrown at runtime.

Justificación:

El resultado es "An exception is thrown at runtime" porque el método nextInt() del objeto Scanner intenta leer enteros desde la cadena de entrada. Sin embargo, la cadena contiene caracteres que no son enteros, como la letra "a". Cuando el Scanner encuentra esta letra, lanza una excepción InputMismatchException porque no puede convertir "a" en un número entero.

13. What is the result?

*1/1

```
public class Bees {
	public static void main(String[] args) {
		try {
			new Bees().go();
		} catch (Exception e) {
			System.out.println("thrown to main");
		}
	}
	synchronized void go() throws InterruptedException {
		Thread t1 = new Thread();
		t1.start();
		System.out.print("1 ");
		t1.wait(5000);
		System.out.print("2 ");
	}
}
```

The program prints 1 then 2 after 5 seconds.
The program prints: 1 thrown to main.
The program prints: 1 2 thrown to main.
The program prints:1 then t1 waits for its notification.

Justificación:

El programa imprime "1 thrown to main" debido a un problema con el método wait llamado sobre **t1**. En el método go, después de iniciar el hilo t1, el programa intenta hacer que ese hilo espere 5000 milisegundos. Sin embargo, el método wait debe ser llamado desde un contexto sincronizado en el que el objeto en el que se llama al método wait (en este caso, el hilo t1) debe tener el bloqueo del monitor. Como este no es el caso aquí, se lanza una excepción de IllegalMonitorStateException.

Cuando se lanza esta excepción, el flujo del programa pasa al bloque catch en el método main, donde se imprime "thrown to main".

14. Which statement is true?

*1/1

```
class ClassA {
        public int numberOfInstances;
        protected ClassA(int numberOfInstances) {
                this.numberOfInstances = numberOfInstances;
        }
}
public class ExtendedA extends ClassA {
        private ExtendedA(int numberOfInstances) {
                super(numberOfInstances);
        }
        public static void main(String[] args) {
                ExtendedA ext = new ExtendedA(420);
                System.out.print(ext.numberOfInstances);
        }
}
```

420 is the output.
An exception is thrown at runtime.
All constructors must be declared public.
Constructors CANNOT use the private modifier.
Constructors CANNOT use the protected modifier.

Justificación:

Este código imprime "420" porque superclase ClassA hay una variable llamada numberOfInstances que se establece cuando se crea una nueva instancia de la clase. La clase ExtendedA hereda de esta superclase y su constructor también recibe un número, que pasa al constructor de la clase superclase para inicializar numberOfInstances.
En el método principal (main), se crea una nueva instancia de ExtendedA con el valor 420. Este valor se asigna a numberOfInstances y luego se imprime.

15. The SINGLETON pattern allows:

*0/1

Have a single instance of a class and this instance cannot be used by other classes
Having a single instance of a class, while allowing all classes have access to that instance.
Having a single instance of a class that can only be accessed by the first method that calls it.

Respuesta correcta

Having a single instance of a class, while allowing all classes have access to that instance.

Justificación:

La razón es que el patrón Singleton asegura que una clase tenga solo una instancia y proporciona un punto de acceso global a dicha instancia. Esto significa que cualquier clase dentro de la aplicación puede acceder y usar esa única instancia.

16. What is the result?

*0/1

```
import java.text.*;
public class Align {
        public static void main(String[] args) throws ParseException {
                String[] sa = {"111.234", "222.5678"};
                NumberFormat nf = NumberFormat.getInstance();
                nf.setMaximumFractionDigits(3);
                for (String s : sa) { System.out.println(nf.parse(s)); }
        }
}
```

111.234 222.567
111.234 222.568
111.234 222.5678
An exception is thrown at runtime.

Respuesta correcta
111.234 222.5678

Justificación:

El código realiza los siguientes pasos:

1. Se define un arreglo de cadenas con dos elementos: "111.234" y "222.5678".
2. Se crea una instancia de NumberFormat usando NumberFormat.getInstance().
3. Se establece el número máximo de dígitos fraccionarios en 3 con el método setMaximumFractionDigits(3).
4. Para cada cadena en el arreglo, se utiliza el método parse del NumberFormat para convertir la cadena en un número de 3 dígitos, y se imprimen en consola. Dado que el número máximo de dígitos fraccionarios se establece en 3, entonces los valores se redondean, obteniendo como resultado 111.234 222.568.

17. What is the result?

*0/1

Given

```
public class SuperTest {
        public static void main(String[] args) {
                //statement1
                //statement2
                //statement3
        }
}
class Shape {
        public Shape() {
                System.out.println("Shape: constructor");
        }
        public void foo() {
                System.out.println("Shape: foo");
        }
}
class Square extends Shape {
        public Square() {
                super();
        }
        public Square(String label) {
                System.out.println("Square: constructor");
        }
        public void foo() {
                super.foo();
        }
        public void foo(String label) {
                System.out.println("Square: foo");
        }
}
```

What should statement1, statement2, and statement3, be respectively, in order to produce the result?

Shape: constructor
Shape: foo
Square: foo

Square square = new Square ("bar"); square.foo ("bar"); square.foo();
Square square = new Square ("bar"); square.foo ("bar"); square.foo ("bar");
Square square = new Square (); square.foo (); square.foo(bar);
Square square = new Square (); square.foo (); square.foo("bar");
Square square = new Square (); square.foo (); square.foo ();

Respuesta correcta

Square square = new Square (); square.foo (); square.foo("bar");

Justificación:

Esta opción es la correcta porque sigue la sieguente secuencia:

- Creación del objeto Square:
  - Square square = new Square(); Esta línea crea una instancia de Square y llama al constructor sin parámetros de la clase Square.
  - Dentro de este constructor, se llama a super(), que invoca el constructor sin parámetros de la superclase Shape, lo que imprime "Shape: constructor".
- Primera llamada a foo():

- square.foo(); Esta llamada invoca el método foo() en la clase Square, el cual llama a super.foo().
- super.foo() llama al método foo() de la superclase Shape, imprimiendo "Shape: foo".

- Segunda llamada a foo("bar"):
  - square.foo("bar"); Esta llamada invoca el método foo(String label) de la clase Square, el cual imprime "Square: foo".

18. Which three implementations are valid?

*0/1

```
interface SampleCloseable {
        public void close() throws java.io.IOException;
}
```

class Test implements SampleCloseable { public void close() throws java.io.IOException { // do something } }

class Test implements SampleCloseable { public void close() throws Exception { // do something } }

class Test implements SampleCloseable { public void close() throws FileNotFoundException { // do something } }

class Test extends SampleCloseable { public void close() throws java.io.IOException { // do something } }

class Test implements SampleCloseable { public void close() { // do something }}

Respuesta correcta

class Test implements SampleCloseable { public void close() throws java.io.IOException { // do something } }
class Test implements SampleCloseable { public void close() throws FileNotFoundException { // do something } }
class Test implements SampleCloseable { public void close() { // do something }}

Justificación:

1. La primera implementación es válida porque el método close() lanza java.io.IOException, que coincide con la excepción declarada en la interfaz.
2. La segunda implementación no es válida porque lanza Exception, que es más general y amplía el rango de excepciones permitidas, lo cual no está permitido.
3. La tercera implementación es válida porque lanza FileNotFoundException, que es una subclase de java.io.IOException, por lo tanto, cumple con la excepción especificada en la interfaz.
4. La cuarta implementación no es válida porque intenta extender SampleCloseable en lugar de implementarla. Las interfaces deben ser implementadas, no extendidas.
5. La quinta implementación es válida porque no lanza ninguna excepción, lo cual es permitido y compatible con lanzar java.io.IOException.

19. What is the result?

*1/1

```
class MyKeys {
        Integer key;
        MyKeys(Integer k) { key = k; }
        public boolean equals(Object o) {
                return ((MyKeys) o).key == this.key;
        }
}

And this code snippet:

Map m = new HashMap();
MyKeys m1 = new MyKeys(1);
MyKeys m2 = new MyKeys(2);
MyKeys m3 = new MyKeys(1);
MyKeys m4 = new MyKeys(new Integer(2));
m.put(m1, "car");
m.put(m2, "boat");
m.put(m3, "plane");
m.put(m4, "bus");
System.out.print(m.size());
```

2
3
4
Compilation fails.

Justificación:

El resultado del código es 4 debido a que:

1. Se crean cuatro objetos MyKeys con valores de claves distintos o iguales.
2. Aunque m1 y m3 tienen el mismo valor de clave 1, y m2 y m4 tienen el mismo valor de clave 2, el método hashCode no se ha sobrescrito en la clase MyKeys.
3. La falta de sobrescritura del método hashCode hace que cada instancia se trate como única en el HashMap, ya que se compara la igualdad utilizando el método equals y no por el valor de la clave.
4. Por tanto, el HashMap mantiene las cuatro entradas separadas, resultando en un tamaño de 4.

20. What value of x, y, z will produce the following result? 1234,1234,1234 ------, 1234, ------

* 0/1

```
public static void main(String[] args) {
        // insert code here
        int j = 0, k = 0;
        for (int i = 0; i < x; i ++) {
                do {
                        k = 0;
                        while (k < z) {
                                k++;
                                System.out.print(k + " ");
                        }
                        System.out.println(" ");
                        j++;
                } while (j < y);
                System.out.println("---");
        }
}
```

int x = 4, y = 3, z = 2;
int x = 3, y = 2, z = 3;
int x = 2, y = 3, z = 3;
int x = 2, y = 3, z = 4;
int x = 4, y = 2, z= 3;

Respuesta correcta

int x = 2, y = 3, z = 4;

Justificación:

Con los valores x = 2, y = 3, z = 4 pasa lo siguiente:

- **z=4** hace que el bucle while imprima los números del 1 al 4 en cada iteración del bucle do-while, dando el resultado de 1234.
- **y=3** hace que el bucle do-while interno se ejecute tres veces por cada iteración del bucle externo, y por eso se imprime 1234 1234 1234
- **x=2** hace que el bucle externo for se ejecute dos veces, por eso se imprime ------

21. Which three lines will compile and output "Right on!"?

*0/1

```
13.  public class Speak {
14.      public static void main(String[] args) {
15.          Speak speakIT = new Tell();
16.          Tell tellIt = new Tell();
17.          speakIT.tellItLikeItIs();
18.          (Truth) speakIT.tellItLikeItIs();
19.          ((Truth) speakIT).tellItLikeItIs();
20.          tellIt.tellItLikeItIs();
21.          (Truth) tellIt.tellItLikeItIs();
22.          ((Truth) tellIt).tellItLikeItIs();
23.      }
24.  }

class Tell extends Speak implements Truth {
    @Override
    public void tellItLikeItIs() {
        System.out.println("Right on!");
    }
}

interface Truth {
    public void tellItLikeItIs();
}
```

Line 17
Line 18
Line 19
Line 20
Line 21
Line 22

Respuestas correctas

Line 19
Line 20
Line 22

Justificación:

Para que se compile y se imprima "Right on!" en la salida, es necesario que el método tellItLikeItIs se llame desde una instancia de la clase Tell, que implementa la interfaz Truth. Aquí están las líneas que cumplen con esta condición:

- Línea 19: ((Truth) speakIT).tellItLikeItIs();
  Aquí, speakIT es una instancia de Tell y se hace un casting a Truth. Esto permite llamar al método tellItLikeItIs() de la instancia Tell.

- Línea 20: tellIt.tellItLikeItIs();
  tellIt es directamente una instancia de Tell, por lo que llama al método tellItLikeItIs() y produce "Right on!".

- Línea 22: ((Truth) tellIt).tellItLikeItIs();
  Similar a la línea 19, se hace un casting de tellIt (que es una instancia de Tell) a Truth,

permitiendo la llamada al método tellItLikeItIs().

22. What is the result?

*1/1

```
class Feline {
    public String type = "f";
    public Feline() {
        System.out.print(s: "feline ");
    }
}
public class Cougar extends Feline{
    public Cougar() {
        System.out.print(s: "cougar ");
    }
    void go(){
        String type = "c";
        System.out.print(this.type + super.type);
    }

    Run | Debug
    public static void main(String[] args) {
        new Cougar().go();
    }
}
```

Feline cougar c f
Feline cougar c c
Feline cougar f f
No compila

Justificación:

El código imprime "feline cougar f f" porque:

1. El constructor de Feline se ejecuta primero e imprime "feline ".
2. Luego, el constructor de Cougar se ejecuta e imprime "cougar ".
3. Finalmente, en el método go de Cougar se declara una variable local type con el valor "c". Esta variable es heredada de Feline, por eso this.type tiene el valor "f" al igual que la variable de instancia type de la superclase Feline.

23. ¿Cuál es el resultado?

*1/1

```
import java.util.*;
public class App {
        public static void main(String[] args) {
                List p = new ArrayList();
                p.add(7);
                p.add(1);
                p.add(5);
                p.add(1);
                p.remove(1);
                System.out.println(p);
        }
}
```

[7, 5]
[7, 1]
[7, 5, 1]
[7, 1, 5, 1]

Justificación:

El código crea una lista de enteros y añade los valores 7, 1, 5 y 1, en ese orden. Luego, elimina el primer elemento que coincide con el valor 1.
La lista inicial contiene los elementos [7, 1, 5, 1]. Al eliminar el primer 1, la lista se convierte en [7, 5, 1].
Así que el resultado final que se imprimirá es [7, 5, 1].

24. What is the result?
*1/1

```
public class Test {
        public static void main(String[] args) {
                int b = 4;
                b--;
                System.out.print(--b);
                System.out.println(b);
        }
}
```

22
12
32
33

Justificación:

En el código, la variable b comienza con el valor 4. Primero, se aplica el operador b--, que reduce b a 3 pe. Luego, se utiliza --b, que decrementa b nuevamente a 2 y este valor se imprime. Finalmente, se imprime el valor actual de b, que es 2. Por lo tanto, el resultado del programa será 2 y 2.

25. In Java the difference between throws and throw Is:

*1/1

Throws throws an exception and throw indicates the type of exception that the method.
Throws is used in methods and throw in constructors.
Throws indicates the type of exception that the method does not handle and throw an exception.

Justificación:

"throws" se coloca en la firma del método y se usa para declarar posibles excepciones pero no son manejadas dentro del mismo, , obligando a quien llame al método a manejarlas. Por otro lado, "throw" se usa para lanzar una excepción específica en un momento dado.

26. Which statement, when inserted into line '' // TODO code application logic here", is valid in compilation time change?

*1/1

```
public class SampleClass {
        public static void main(String[] args) {
                AnotherSampleClass asc = new AnotherSampleClass();
                SampleClass sc = new SampleClass();
                // TODO code application logic here
        }
}
class AnotherSampleClass extends SampleClass { }
```

asc = sc;
sc = asc;
asc = (Object) sc;
asc= sc.clone();

Justificación:

sc = asc; es la respuesta correcta porque asc es de tipo AnotherSampleClass, que extiende SampleClass, lo que permite asignar asc a sc sin problemas de compilación.

27. What is the result?

*1/1

```
public class Test {
        public static void main(String[] args) {
                int[][] array = { {0}, {0,1}, {0,2,4}, {0,3,6,9}, {0,4,8,12,16} };
                System.out.println(array[4][1]);
                System.out.println(array[1][4]);
        }
}
```

4 Null.
Null 4.
An IllegalArgumentException is thrown at run time.
4 An ArrayIndexOutOfBoundsException is thrown at run time.

Justificación:

En este código se define un arreglo bidimensional llamado con varias submatrices. Luego, se intenta imprimir dos elementos específicos de este arreglo:

- Elemento en la posición [4][1]: La submatriz en la posición 4 es {0, 4, 8, 12, 16}, y el elemento en la posición 1 de esta submatriz es 4.
- Elemento en la posición [1][4]. La submatriz en la posición 1 es {0, 1}, que solo tiene dos elementos (índices 0 y 1), pero al intentar acceder al índice 4 de esta submatriz causa una excepción de índice fuera de rango.

Por lo tanto, el resultado es que se imprimirá 4 seguido de una excepción ArrayIndexOutOfBoundsException.

28. Which three are valid? (Choose three)
*0/1

```
class ClassA {}
class ClassB extends ClassA {}
class ClassC extends ClassA {}
And:
ClassA p0 = new ClassA();
ClassB p1 = new ClassB();
ClassC p2 = new ClassC();
ClassA p3 = new ClassB();
ClassA p4 = new ClassC();
```

p0 = p1;
p1 = p2;
p2 = p4;
p2 = (ClassC)p1;

p1 = (ClassB)p3;
p2 = (ClassC)p4;

Respuesta correcta

p0 = p1;
p1 = (ClassB)p3;
p2 = (ClassC)p4;

Justificación:

- p0 = p1; Esto es válido porque ClassB es una subclase de ClassA, por lo que un objeto de tipo ClassB puede ser asignado a una variable de tipo ClassA.
- p1 = (ClassB)p3; Esta asignación es válida porque p3 es de tipo ClassA, pero en realidad es un objeto ClassB (polimorfismo). Al hacer un cast a ClassB se puede asignar a p1.
- p2 = (ClassC)p4; Similar al caso anterior, p4 es de tipo ClassA, pero en realidad es un objeto ClassC. Al hacer un cast a ClassC se puede asignar a p2.

29. Which three options correctly describe the relationship between the classes?
*0/1

```
class Class1 { String v1; }
class Class2 {
        Class1 c1;
        String v2;
}
class Class3 { Class2 c1; String v3; }
```

Class2 has-a v3.

Class1 has-a v2.

Class2 has-a v2.

Class3 has-a v1.

Class2 has-a Class3.

Class2 has-a Class1.

Respuesta correcta

Class2 has-a v2.
Class3 has-a v1.
Class2 has-a Class1.

Justificación:

El concepto de "**has-a**" indica que una clase contiene una instancia de otra clase como atributo. Es decir, un objeto de la primera clase "tiene" un objeto de la segunda clase. Analizando el código, se obtiene lo siguiente:

- Class1: Tiene un atributo de tipo cadena llamado v1.
- Class2: Tiene un atributo de tipo Class1 llamado c1 y otro de tipo cadena llamado v2.
- Class3: Tiene un atributo de tipo Class2 llamado c1 y otro de tipo cadena llamado v3.

Por lo tanto, las respuestas correctas son Class2 has-a v2, Class3 has-a v1 y Class2 has-a Class1.

30. What is the result?

*0/1

```
class MySort implements Comparator<Integer> {
        public int compare(Integer x, Integer y) {
                return y.compareTo(x);
        }
}
```

And the code fragment:

```
Integer[] primes = {2, 7, 5, 3};
MySort ms = new MySort();
Arrays.sort(primes, ms);
for (Integer p2 : primes) { System.out.print(p2 + " "); }
```

2 3 5 7
2 7 5 3
7 5 3 2
Compilation fails.

Respuesta correcta

7 5 3 2

Justificación:

En este código, se define una clase llamada MySort que implementa la interfaz Comparator<Integer>. El método compare de esta clase compara dos números enteros, x y y, en orden inverso al habitual (de mayor a menor) utilizando y.compareTo(x). A continuación, se crea un arreglo con los valores {2, 7, 5, 3} y se ordena utilizando una instancia de MySort como comparador. Debido a esta lógica de comparación, el arreglo se imprimirá en la siguiente secuencia de números: 7 5 3 2.

31. Which two possible outputs?
*0/1

```
public class Main {
        public static void main(String[] args) throws Exception {
                doSomething();
        }
        private static void doSomething() throws Exception {
                System.out.println("Before if clause");
                if (Math.random() > 0.5) { throw new Exception();}
                System.out.println("After if clause");
        }
}
```

a. Before if clause Exception in thread "main" java.lang.Exception at Main.doSomething (Main.java:21) at Main.main (Main.java:15).

b. Before if clause Exception in thread "main" java.lang.Exception at Main.doSomething (Main.java:21) at Main.main (Main.java:15) After if clause.

c. Exception in thread "main" java.lang.Exception at Main.doSomething (Main.java:21) at Main.main (Main.java:15).

d. Before if clause After if clause.

Respuesta correcta

a. Before if clause Exception in thread "main" java.lang.Exception at Main.doSomething (Main.java:21) at Main.main (Main.java:15).
d. Before if clause After if clause.

Justificación:

En este código, hay dos posibles salidas debido al uso del método Math.random():
Si el número generado es mayor a 0.5, se lanza una excepción y la salida será:

- "Before if clause"
- "Exception in thread 'main' java.lang.Exception at Main.doSomething (Main.java:21) at Main.main (Main.java:15)"

Si el número es menor o igual a 0.5, no se lanza ninguna excepción y la salida será:

- "Before if clause"
- "After if clause"

32. What is the result?
*0/1

```
public static void main(String[] args) {
        String color = "Red";
        switch (color) {
        case "Red":
                System.out.println("Found Red");
        case "Blue":
                System.out.println("Found Blue");
        case "White":
                System.out.println("Found White");
                break;
        Default:
                System.out.println("Found Default");
        }
}
```

Found Red.
Found Red Found Blue.
Found Red Found Blue Found White.
Found Red Found Blue Found White Found Default.

Respuesta correcta

Found Red Found Blue Found White.

Justificación:

El programa evalúa el valor de una variable llamada "color", que contiene la palabra "Red". Utiliza una estructura switch para tomar decisiones basadas en este valor. Sin embargo, debido a la falta de la palabra reservada break (que se encarga de interrumpir la ejecución de un bloque de código) en algunos casos, el programa continúa ejecutando el código de los casos siguientes. Por esta razón, el programa imprime "Found Red", "Found Blue" y "Found White" en la consola, aunque solo el valor "Red" coincida con un caso.

33. What is the result?

*0/1

```
class X {
        static void m(int i) {
                i += 7;
        }
        public static void main(String[] args) {
                int i = 12;
                m(i);
                System.out.println(i);
        }
}
```

7
12
19
Compilation fails.
An exception is thrown at run time

Respuesta correcta

12

Justificación:

El resultado del código es 12. Aquí está la explicación:

- En el método main, se declara una variable i con el valor 12.
- Se llama al método estático m pasando i como argumento.
- Dentro del método m, la variable i local se incrementa en 7, pero esto no afecta a la variable i del método main.
- Después de la llamada al método m, se imprime el valor de i en el main, que sigue siendo 12 porque la variable local del método m no tiene efecto sobre la variable i del main.

34. Which is true?

*1/1

```
5.  class Building { }
6.      public class Barn extends Building {
7.          public static void main(String[] args) {
8.              Building build1 = new Building();
9.              Barn barn1 = new Barn();
10.             Barn barn2 = (Barn) build1;
11.             Object obj1 = (Object) build1;
12.             String str1 = (String) build1;
13.             Building build2 = (Building) barn1;
14.         }
15.     }
```

Which is true?

If line 10 is removed, the compilation succeeds.
If line 11 is removed, the compilation succeeds.
If line 12 is removed, the compilation succeeds.
If line 13 is removed, the compilation succeeds.
More than one line must be removed for compilation to succeed.

Justificación:

Esto se debe a que la línea 12 intenta convertir un objeto de tipo Building a String, lo cual no es válido y lanzará una excepción ClassCastException. Al eliminar esta línea, el resto del código compilará sin problemas.

35. What is the result?
*0/1

```
public static void main(String[] args) {
        int [][] array2D = { {0, 1, 2}, {3, 4, 5, 6} };
        System.out.print(array2D[0].length + "" );
        System.out.print(array2D[1].getClass().isArray() + "" );
        System.out.println(array2D[0][1]);
}
```

3false1
2true3
2false3
3true1
3false3
2true1
2false1

Respuesta correcta

3true1

Justificación:

El resultado del código proporcionado es:

1. El primer print muestra la longitud de la primera fila del arreglo bidimensional, que es 3, porque la fila {0, 1, 2} tiene tres elementos.
2. El segundo print verifica si la segunda fila del arreglo bidimensional es un arreglo, lo cual es cierto y se imprime true.
3. El tercer print muestra el valor del segundo elemento de la primera fila, que es 1. Por eso la salida es 3true1.

36. What is the result if the integer value is 33?

*1/1

```
public static void main(String[] args) {
        if (value >= 0) {
                if (value != 0) {
                        System.out.print("the ");
                } else {
                        System.out.print("quick ");
                }
                if (value < 10) {
                        System.out.print("brown ");
                }
                if (value > 30) {
                        System.out.print("fox ");
                } else if (value < 50) {
                        System.out.print("jumps ");
                } else if (value < 10) {
                        System.out.print("over ");
                } else {
                        System.out.print("the ");
                }
                if (value > 10) {
                        System.out.print("lazy ");
                } else {
                        System.out.print("dog ");
                }
                System.out.print("... ");
        }
}
```

The fox jump lazy ?

The fox lazy ?

Quick fox over lazy ?

Justificación:

Cuando el valor es 33, el programa sigue una serie de condiciones anidadas para determinar qué

cadenas imprimir. Primero, verifica si el valor es mayor o igual a 0 y, dado que 33 lo es, ingresa al primer bloque. Dentro de este bloque, se confirma que el valor no es 0, por lo que imprime "the ". Luego, verifica si el valor es menor que 10, lo cual no se cumple, por lo que no imprime "brown ". Posteriormente, al verificar si el valor es mayor que 30, esta condición sí se cumple, por lo que imprime "fox ". En el siguiente conjunto de condiciones, se verifica si el valor es mayor que 10, lo cual también se cumple, imprimiendo "lazy ". Finalmente, imprime "...". Por lo tanto, la salida es "the fox lazy ...".

37. What is the result?

*0/1

```
11. class Person {
12. String name  = "No name";
13. public Person (String nm) { name = nm}
14. }
15.
16. class Employee extends Person {
17.  String empID = "0000";
18.  public Employee (String id) { empID "
id; }
19. }
20.
21. public class EmployeeTest {
22.  public static void main(String[] args)
{
23.   Employee e = new Employee("4321");
24.   System.out.println(e.empID);
25.  }
26. }
```

4321.
0000.
An exception is thrown at runtime.
Compilation fails because of an error in line 18.

Respuesta correcta

Compilation fails because of an error in line 18.

Justificación:

El código presenta un error de compilación porque la clase Employee no llama al constructor de la clase base Person, que requiere un argumento de tipo String. Si una subclase tiene un constructor, debe llamar explícitamente al constructor de la superclase usando super(), y en este caso, debe pasar un argumento de tipo String que es requerido por el constructor de Person. Al no hacerlo, el compilador no puede encontrar un constructor sin argumentos en Person, lo que provoca el error.

What is the result?

*0/1

Given

```
public class SuperTest {
        public static void main(String[] args) {
                //statement1
                //statement2
                //statement3
        }
}
class Shape {
        public Shape() {
                System.out.println("Shape: constructor");
        }
        public void foo() {
                System.out.println("Shape: foo");
        }
}
class Square extends Shape {
        public Square() {
                super();
        }
        public Square(String label) {
                System.out.println("Square: constructor");
         }
        public void foo() {
                super.foo();
        }
        public void foo(String label) {
                System.out.println("Square: foo");
        }
}
```

What should statement1, statement2, and statement3, be respectively, in order to produce the result?

Shape: constructor
Shape: foo
Square: foo

Square square = new Square ("bar"); square.foo ("bar"); square.foo();
Square square = new Square ("bar"); square.foo ("bar"); square.foo ("bar");
Square square = new Square (); square.foo (); square.foo(bar);

Square square = new Square (); square.foo (); square.foo("bar");
Square square = new Square (); square.foo (); square.foo ();

Respuesta correcta

Square square = new Square (); square.foo (); square.foo("bar");

38. Which code fragment is illegal?

*0/1

Class Base1 { abstract class Abs1 { } }
Abstract class Abs2 { void doit() { } }

class Base2 { abstract class Abs3 extends Base2 { } }
class Base3 { abstract int var1 = 89; }

Respuesta correcta

class Base3 { abstract int var1 = 89; }

Justificación:

En el fragmento class Base3 { abstract int var1 = 89; }, se está intentando declarar una variable abstracta var1 con un valor inicial de 89, lo cual no está permitido en Java. Las variables en una clase abstracta no pueden ser abstractas. En lugar de esto, las variables deben ser declaradas normalmente y pueden ser inicializadas si es necesario.

39. What is the result?
*1/1

```
public static void main(String[] args) {
        System.out.println("Result: " + 2 + 3 + 5);
        System.out.println("Result: " + 2 + 3 * 5);
}
```

Result: 10 Result: 30
Result: 25 Result: 10
Result: 235 Result: 215
Result: 215 Result: 215
Compilation fails.

Justificación:

Cuando se concatenan cadenas con números en Java, los números se convierten en cadenas antes de la concatenación. En el primer println, se concatenan los números 2, 3 y 5 como cadenas después de "Result: ", resultando en "Result: 235".

En el segundo println, primero se realiza la multiplicación debido a la precedencia de operadores, multiplicando 3 por 5 para obtener 15. Luego, se suman los números y se convierten en cadenas, resultando en "Result: 215".

40. What is the result?
*0/1

```
public class MyStuff {
        String name;
        MyStuff (String n) { name = n; }
        public static void main (String[] args) {
                MyStuff m1 = new MyStuff ("guitar");
                MyStuff m2 = new MyStuff ("tv");
                System.out.println (m2.equals(m1));
        }
        public boolean equals (Object o) {
                MyStuff m = (MyStuff) o;
                if (m.name != null) { return true; }
                return false;
        }
}
```

The output is true and MyStuff fulfills the Object.equals() contract
The output is false and MyStuff fulfills the Object.equals() contract
The output is true and MyStuff does NOT fulfill the Object.equals() contract.
The output is false and MyStuff does NOT fulfill the Object.equals() contract

Respuesta correcta

The output is true and MyStuff does NOT fulfill the Object.equals() contract.

Justificación:

El código define una clase llamada MyStuff con un atributo name de tipo String y un método equals sobrescrito. El método main crea dos objetos de la clase MyStuff y compara su igualdad utilizando el método equals personalizado.

El método equals siempre devuelve true si el atributo name del objeto pasado no es null, sin importar los valores de los atributos de los objetos que se comparan. Esto viola el contrato de equals, que establece que dos objetos deben considerarse iguales solo si sus contenidos son iguales.