

# Examen práctico Java

## 1. Which three are bad practices?

- Checking for an IOException and ensuring that the program can recover if one occurs.
- Checking for ArrayIndexOutOfBoundsException and ensuring that the program can recover if one occurs.
- Checking for FileNotFoundException to inform a user that a filename entered is not valid.
- Checking for Error and, if necessary, restarting the program to ensure that users are aware of problems.
- Checking for ArrayIndexOutOfBoundsException when iterating through an array to determine when all elements have been visited.

Tres malas prácticas de la lista proporcionada son:

1. \*\*Comprobar `Error` y, si es necesario, reiniciar el programa para asegurarse de que los usuarios no sean conscientes de los problemas\*\*:

- `Error` típicamente representa problemas serios que un programa no debería intentar manejar (por ejemplo, `OutOfMemoryError`). Reiniciar el programa sin abordar el problema subyacente puede llevar a fallos repetitivos y un sistema inestable.

2. \*\*Comprobar `ArrayIndexOutOfBoundsException` al iterar a través de un array para determinar cuándo se han visitado todos los elementos\*\*:

- Usar excepciones para el flujo de control, como determinar el final de un array, es ineficiente y se considera una mala práctica. En su lugar, utiliza estructuras de bucles apropiadas (por ejemplo, un bucle `for` con comprobaciones de límites) para iterar a través de un array.

3. \*\*Comprobar `ArrayIndexOutOfBoundsException` y asegurarse de que el programa puede recuperarse si ocurre una\*\*:

- Similar al punto anterior, confiar en excepciones para gestionar los límites del array es ineficiente. Se debe implementar una comprobación adecuada de los límites antes de acceder a los elementos del array para evitar excepciones en primer lugar.

## **2. Excepciones de Java permitida**

- org.springframework.web.client.RestClientException
- No conozco la respuesta java.lang.NumberFormatException
- com.bbva.apx.exception.db.NoResultException
- com.bbva.elara.utility.interbackend.cics.exceptions.BusinessException

Es una excepción común en el marco de trabajo Spring para manejar errores relacionados con llamadas a servicios web (REST). Aquí hay algunas razones por las que se permite y se usa esta excepción:  
Abstracción y Simplificación.

## **3. 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.**

- Mockito
- Mock
- Inject
- InjectMock

Así, la propiedad que usa la anotación @Mock es reconocida como una simulación (mock) y es susceptible de ser inyectada en otra clase mediante @InjectMocks.

## **4.**

**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?**

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

**Explicación:**

1. `array2D[0]. length`: Da la longitud del primer array `{0,1,2}`, que es 3.
2. `array2D[1]. getClass(). isArray()`: Esto comprueba si `array2D[1]` es un array. Puesto que `array2D[1]` es de hecho un array (`{3,4,5,6}`), esto devuelve true.
3. `array2D[0][1]`: Esto accede al elemento en el índice 1 del primer arreglo `{0,1,2}`, que es 1.

Salida:

- El programa se imprimirá: 3true1

Así que la respuesta correcta es la opción que muestra 3true1.

#### 5. Which two statements are true?

- An interface CANNOT be extended by another interface.
- An abstract class can be extended by a concrete class.
- An abstract class CANNOT be extended by an abstract class.
- An interface can be extended by an abstract class.
- An abstract class can implement an interface.
- An abstract class can be extended by an interface

Las dos afirmaciones que son ciertas son:

1. Una clase abstracta puede ser extendida por una clase concreta:
  - Esto es cierto. Una clase concreta puede heredar de una clase abstracta y debe proporcionar implementaciones para todos los métodos abstractos definidos en la clase abstracta.
2. Una clase abstracta puede implementar una interfaz:
  - Esto también es cierto. Una clase abstracta puede implementar una o más interfaces, y debe proporcionar implementaciones para los métodos de la interfaz, aunque puede dejar algunos métodos sin implementar (es decir, como métodos abstractos), obligando a las clases concretas que la extienden a proporcionar esas implementaciones.

## 6. Problem:

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?

- Gamma gamma
- Beta beta
- Alpha beta
- Compilation fails

En resumen, aunque la compilación en sí no falla, el programa no se ejecutará correctamente debido a las excepciones en tiempo de ejecución. Sin embargo, la opción “Compilation fails” es la que parece ajustarse a la situación de fallo del programa, aunque técnicamente sería una “runtime exception”. Respuesta correcta

## 7. Which five methods, inserted independently at line 5, will compile? (Choose five)

```
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; }

Estas opciones reflejan las reglas de sobrecarga y sobrescritura válidas en Java.

## 8. Problem

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);}

Explicación:

- super(5) llama al constructor de la clase Super que acepta un parámetro entero.
- Esto es válido si la clase Super tiene un constructor que toma un entero como argumento.
- Este cambio asegura que el constructor de Super se ejecute correctamente antes de ejecutar cualquier lógica en el constructor de Sub.

## 9. Problem: What's true about the class Wow?

```
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.

No compila porque una clase abstracta debe tener un constructor sin argumentos: Incorrecto, una clase abstracta puede tener cualquier tipo de constructor. La opción correcta es que compila sin errores: "Compila sin error."

10. What is the result?

```
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

11. What is printed out when the program is executed?

```
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");  
    }  
}
```

- one
- two
- three
- four
- There is no output.

El punto de entrada estándar para una aplicación Java es el método `public static void main(String[] args)`. Por lo tanto, cuando se ejecuta esta clase, se llamará al método `public static final void main(String[] args)`, y se imprimirá:

three

Los otros métodos `main` y el método `mina` no se ejecutarán a menos que sean llamados explícitamente dentro del programa o utilizados en un contexto diferente. Así que la salida será `three`.

## 12. What is the result?

```

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();
    }
}

```

- Cougar c f.
- Feline cougar c c.
- Feline cougar c f.
- Compilation fails.

Respuesta correcta

Cuando se ejecuta este programa, el siguiente es el flujo de ejecución:

1. Se crea una nueva instancia de Cougar, lo que primero llama al constructor de Feline, imprimiendo "feline ".
2. Luego, el constructor de Cougar se ejecuta, imprimiendo "cougar ".
3. El método go() se llama, que cambia el valor de type a "c" y luego imprime this.type + super.type.

En este caso, this.type es "c" y super.type es "f" (el valor original de la clase base Feline).

Por lo tanto, la salida del programa será:

feline cougar cf

**13. What is the result?**

```
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 = new Alpha();
        Gamma g2 = new Beta();
        System.out.println(g1.getType() + " " + g2.getType());
    }
}
```

- Alpha beta
- Beta beta.
- Gamma gamma.
- Compilation fails

El código intenta asignar una instancia de Alpha y Beta a variables de tipo Gamma, lo cual es incorrecto porque Alpha y Beta no son subclases de Gamma. Esto generará un error de compilación debido a tipos incompatibles.

**14. What is the result?**

```
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);
    }
}
```

- 11
- 111
- 1111
- An exception is thrown at runtime.  
Dado que `s.nextInt()` encuentra un carácter no numérico  
(a) en la segunda iteración, se lanzará una excepción en  
tiempo de ejecución.

15. What is the result?

```
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.
- Explicación:** El programa espera 5 segundos entre imprimir 1 y 2.

**16. Which statement is true?**

```

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.

**Explicación:** Algun problema en tiempo de ejecución causa que se lance una excepción.

17. The **SINGLETON pattern allows:**

- 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.

"Tener una sola instancia de una clase, permitiendo que todas las clases tengan acceso a esa instancia."

Esto se debe a que el propósito principal del patrón Singleton es asegurarse de que una clase tenga una única instancia y proporcionar un punto global de acceso a esa instancia.

**18. What is the result?**

```
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

Respuesta correcta

111.234 222.5678

**Explicación:** Los valores se formatean correctamente sin lanzar una excepción.

**19. What is the result?**

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");
```

20. Which three implementations are valid?

```
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 } }

La primera implementación es válida si `SampleCloseable` declara `void close() throws java.io.IOException`.

La segunda implementación no es válida porque `FileNotFoundException` no puede ser usada si `SampleCloseable` declara `java.io.IOException` y no hay una relación directa de superclase entre las excepciones.

La tercera implementación es válida porque no declarar ninguna excepción es una relajación permitida de la declaración original.

21. What is the result?

```
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 dos
- 3 tres
- 4 cuatro
- Compilation fails.

Hay un error de compilación en el código proporcionado.

22. What value of x, y, z will produce the following result?

1234,1234,1234 -----, 1234, -----

```
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;`

**Explicación:** Esta combinación de valores produce la salida esperada.

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

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

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

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



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

```

R: Line 19, Line 20, Line 21

**Explicación:** Estas líneas son las que compilan y producen la salida "Right on!".

24. What is the result?

```
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

25. ¿Cuál es el resultado?

```
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]

**Explicación:** El código da como resultado [7, 1] debido a cómo se manipulan las variables.

26. Un \_\_\_\_\_ proporciona una solución a un problema de diseño.

Debe cumplir con diferentes características, como la efectividad al resolver problemas similares en ocasiones anteriores. Por lo tanto, debe de ser re-utilizable, es decir, aplicable a diferentes problemas en diferentes circunstancias.

- Reutilizando código
- Anti-patrón
- Metodología
- Patrón de diseño

Un patrón de diseño proporciona una solución a un problema de diseño. Debe cumplir con diferentes características, como la efectividad al resolver problemas similares en ocasiones anteriores. Por lo

tanto, debe ser re-utilizable, es decir, aplicable a diferentes problemas en diferentes circunstancias.

27. ¿Cuál es el resultado?

```
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]

28. **REPEAT** Which five methods, inserted independently at line 5, will compile? (Choose five)

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

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

Estas opciones reflejan las reglas de sobrecarga y sobrescritura válidas en Java.

## 29. REPEAT

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);}

Respuesta correcta

Explicación:

- super(5) llama al constructor de la clase Super que acepta un parámetro entero.
- Esto es válido si la clase Super tiene un constructor que toma un entero como argumento.
- Este cambio asegura que el constructor de Super se ejecute correctamente antes de ejecutar cualquier lógica en el constructor de Sub.

30. REPEAT

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?

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

31. ¿Qué enunciados son verdaderos? REPEAT

- An interface CANNOT be extended by another interface.
- An abstract class can be extended by a concrete class.
- An abstract class CANNOT be extended by an abstract class.
- An interface can be extended by an abstract class.
- An abstract class can implement an interface.
- An abstract class can be extended by an interface.

32. REPEAT

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?

- Gamma gamma
- Beta beta
- Alpha beta
- Compilation fails

No se compilará debido a que no se puede convertir directamente Alpha o Beta a una Gamma, ya que no es una superclase y no se están casteando.

33. Which three are bad practices? REPEAT

- Checking for an IOException and ensuring that the program can recover if one occurs.
- Checking for ArrayIndexOutOfBoundsException and ensuring that the program can recover if one occurs. Es una mala práctica porque es una excepción no corroborada al no corroborar los índices de los arrays.
- Checking for FileNotFoundException to inform a user that a filename entered is not valid.
- Checking for Error and, if necessary, restarting the program to ensure that users are unaware problems. Intentar capturar Error y reiniciar el programa es generalmente una mala práctica, ya que podría ocultar problemas graves y llevar a un comportamiento impredecible.
- Checking for ArrayIndexOutOfBoundsException when iterating through an array to determine when all elements have been visited. En su lugar se recomienda usar bucles adecuados que eviten estas excepciones.

34. What is the result?\*

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

- 2 2
- 1 2
- 3 2
- 3 3

El b -- es un postdecremento, por lo que el 4 se convierte en un 3, posteriormente se le aplica un predecremento con el -b, entonces la ser 3 pasa a ser 2 y ese es el primer valor impreso. Para el segundo valor b sigue siendo un 2, ya que no vuelve a sufrir modificaciones.

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

- 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.

Throw se utiliza para lanzar una excepción dentro de un métodos, mientras que throws se declara en el método para así lanzar ciertas excepciones.

36. What is the result?

```
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();  
    }  
}
```

- Cougar c f.
- Feline cougar c c.
- Feline cougar c f.
- Compilation fails.

Al crear al Cougar, el constructor de Feline, regresará un feline, después el constructor Cougar imprimirá un cougar, por último en el go(), llama al tipo c de la clase Cougar y una f como el super proveniente de feline.

Respuesta correcta

Feline cougar c c.

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

```
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();

sc = asc; asigna la instancia asc a la variable sc. Y como AnotherSampleClass hereda de SampleClass, la asignación es válida

38. What is the result?

```
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.

El programa imprimirá primero un 4, debido al elemento mencionado en el primer System.out, y después lanzará la excepción de Array fuera de los límites porque el array al que se refiere en el segundo System.out, no tiene esas dimensiones.

39. What is the result?REPEAT

```
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, 1, 5, 1]
- [7, 5, 1]
- [7, 5]
- [ 7, 1]

Dentro del código se añaden 7, 1, 5 y 1 con el p.add. Posterior a esto, con p.remove se quita el elemento del índice 1, por lo que el primer 1 se elimina, dejando así 7, 5, 1.

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

```

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

Las respuestas son las líneas 19 porque speakIT es casteado a Truth y el método tellItLikeItIs es implementado, y como speakIT es el objeto Tell, va a invocar al TellItLikeItIs.

La línea 20 hace referencia al objeto Tell que llama directamente al método TellItLikeItIs.

La línea 22 castea el TellIT al Truth, y llama al método TellItLikeItIs.

41. What is the result?

```
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.

Arrojará 1, porque IllegalMonitorStateException es arrojado cuando el método wait es mal llamado