

Guía Java

Preguntas

Respuestas correctas

Explicación

1. Which three are bad practices?

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

Es importante gestionar errores que pueden ocurrir durante entrada o salida, como leer o escribir archivos

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

Es una mala práctica pues el error debe prevenirse asegurandose que en el código los índices no queden fuera de rango.

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

No es mala práctica, pues involucra el manejo de entradas

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

Los Error en Java representan problemas graves en el entorno de ejecución, como falta de memoria, y generalmente no deben ser capturados o manejados como excepciones normales. Intentar reiniciar el programa para ocultar estos errores no es una buena estrategia, ya que puede ocultar problemas serios que deben abordarse.

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

La excepción `ArrayIndexOutOfBoundsException` no debe ser utilizada para controlar la iteración sobre un arreglo.

Debes usar un bucle que itere de manera segura dentro del rango del arreglo.

2. Dado

```
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]);  
}
```

¿Cuál es el resultado?

- 3false3
- 3false1
- 2false1
- **3true1**

3 proviene del primer print, pues es la longitud del arreglo que ocupa la posición 0,

`array2D[1].getClass().isArray()` verifica si el segundo subarreglo es de tipo array, por lo que devuelve `true`, y el 1 es el elemento que se encuentra en la posición [0][1].

- 2true3

3. Which two statements are true?

- An interface CANNOT be extended by another interface.

Una interfaz puede extender otra interfaz.

- **An abstract class can be extended by a concrete class.**

Una clase abstracta puede ser extendida por una clase concreta

- An abstract class CANNOT be extended by an abstract class.

Una clase abstracta puede ser extendida por otra clase abstracta.

- An interface can be extended by an abstract class.

Una clase abstracta no puede extender una interfaz, pero puede implementarla.

- **An abstract class can implement an interface.**

Una clase abstracta puede implementar una interfaz

- An abstract class can be extended by an interface.

Una interfaz no puede extender una clase abstracta; una interfaz solo puede extender otra interfaz.

4. Dado

```
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());
    }
}
```

¿Cuál es el resultado?

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

Este código genera un error ClassCastException porque Alpha no se puede convertir a Gamma y Beta no se puede convertir a Gamma ya que Gamma es una subclase de Beta y Alpha, no al revés.

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

```
public class Blip {  
    protected int blipvert(int x) {  
        return 0;  
    }  
}  
  
class Vert extends Blip {  
    // insert code here  
}
```

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

Es una sobrecarga.

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

Es una sobrecarga.

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

Es una sobrecarga.

- **Public int blipvert(int x) { return 0; }**

Es una sobrecarga y usa un modificador más permisivo

- Private int blipvert(int x) { return 0; }

Este método intenta sobrescribir el método pero usa un modificador de acceso más restrictivo.

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

Este método intenta sobrescribir el método pero cambia el retorno.

- **Protected long blipvert(long x) { return 0; }**

Es una sobrecarga.

6. What is the result?

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); }**

Utiliza el método definido en el padre Super, pues como se encuentra es invalido

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

Utiliza el método definido en su propia clase y que a la vez convoca al constructor por default.

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

7. 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.**

No existen problemas

- It does not compile because an abstract class cannot have private methods

Una clase abstracta puede tener métodos privados.

- It does not compile because an abstract class cannot have instance variables.

Una clase abstracta puede tener variables de instancia.

- It does not compile because an abstract class must have at least one abstract method.

En Java, una clase abstracta no necesita tener al menos un método abstracto.

- It does not compile because an abstract class must have a constructor with no arguments.

Una clase abstracta no está obligada a tener un constructor sin argumentos.

8. 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.
- **Atom granite atom granite.**

El orden de ejecución es: constructor de mountain, se llama al constructor de rock, rock llama implicitamente al super de atom y se imprime atom, posterior se imprime granite debido al constructor de rock, enseguida se crea una rock, por lo que se vuelve a llamar al super de atom implicitamente, se imprime atom y finalmente granite.

9. 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**

Es el único método que corresponde con el punto de entrada, el punto de entrada debe ser el método main, debe ser public, static, no debe tener retorno y puede ser final.

- four
- There is no output

10. 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.**

El constructor de Cougar invoca al constructor de Feline, por lo que se imprime feline, posteriormente cougar, finalmente en el método go se llama a la variable de instancia type, el contenido es el mismo.

- Feline cougar c f.
- Compilation fails.

11. 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 objeto Alpha y Beta son superiores al contenedor que se les quiere asignar

12. 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.

13. 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.

14. 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.**

Se instancia un objeto ExtendedA, se invoca al constructor de ClassA y el atributo se coloca como 420.

- An exception is thrown at runtime.
- All constructors must be declared public.

[Los constructores pueden tener diferentes niveles de visibilidad](#)

- Constructors CANNOT use the private modifier.

[Los constructores pueden ser declarados como privados.](#)

- Constructors CANNOT use the protected modifier.

[Los constructores pueden usar el modificador protegido](#)

15. The SINGLETON pattern allows:

- Have a single instance of a class and this instance cannot be used by other classes

[Se debe proporcionar un único acceso global](#)

- **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

[Se debe proporcionar un único acceso global](#)

16. 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

[El redondeo se realiza hacia arriba](#)

- **111.234 222.568**

[El redondeo se realiza hacia arriba](#)

- 111.234 222.5678

[El factorDigits es 3](#)

- An exception is thrown at runtime.

17. 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();`

Utilizar el constructor con parámetros implica que se implica que se imprime Shape constructor y enseguida square: constructor.

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

Utilizar el constructor con parámetros implica que se implica que se imprime Shape constructor y enseguida square: constructor.

- `Square square = new Square (); square.foo (); square.foo(bar);`

No existe un método que soporte

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

- `Square square = new Square (); square.foo (); square.foo ();`

Se repite la linea 2 al ejecutar la tercera instrucción

18. 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 } }`

No se pueden arrojar excepciones de mayor jerarquia a la declarada en la interfaz

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

Lanza un excepción más específica

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

No se puede usar extends para una interfaz

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

19. 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
- 3
- 4

Se estan almacenando 4 elementos con key que son objetos, no los números que se pasan como parámetro a MyKeys

- Compilation fails.

20. 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;**

Ese valor de z garantiza la impresión de los primeros 4 números en el while

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

21. 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.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

Speak no tiene implementado el método, la sobreescritura se realiza sobre la interfaz

- Line 18

El casteo se hace de manera incorrecta

- **Line 19**

El casteo se realiza de manera adecuada y se emplea el método que sobreescribe

- **Line 20**

Se invoca al método dentro de Tell

- Line 21

El casteo se hace de manera incorrecta

- **Line 22**

El casteo se realiza de manera adeuada y se emplea el método que sobreescribe

22. 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 ff**

Se convoca el constructor de Cougar, se llama implicitamente al constructor de felino, se imprime felino, se imprime cougar, se llama al método go, type = "c" corresponde a una variable de método, se imprime el atributo type con valor f.

- No compila

23. ¿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]**

Un arrayList permite elementos repetidos, se tiene 7 1 5 1, se remueve el elemento con índice 1.

- [7, 1, 5, 1]

24. 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);
    }
}
```

- 22

Inicialmente se decrementa a 3, el primer print decrementa a 2 y luego imprime, enseguida únicamente se reimprime 2

- 12
- 32
- 33

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

- Throws throws an exception and throw indicates the type of exception that the method.
throw se usa para lanzar una excepción, pero no indica el tipo de excepción en el método.
- Throws is used in methods and throw in constructors.
throws se usa en la firma del método o constructor y throw en el cuerpo de cualquiera
- **Throws indicates the type of exception that the method does not handle and throw an exception**

26. 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;`

Contenedor para subclase no puede contener una clase más elevada

- `sc = asc;`

Contenedor más general puede contener clases más específicas

- `asc = (Object) sc;`

Contenedor para subclase no puede contener una clase más elevada

- `asc= sc.clone();`

Contenedor para subclase no puede contener una clase más elevada

27. 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**

se accede al arreglo en la posición 4 y posteriormente a su elemento 1 → se imprime 4, se accede al arreglo de la posición 1, se busca el elemento 4 en ese arreglo → el indice excede el tamaño del arreglo

28. Which three are valid? (Choose three)

```

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;**
p0 es contenedor superior, p1 puede ser referido desde p0
- **p1 = p2;**
p1 es un contenedor hijo de A pero hermano de C, p2 es tipo C, no se puede
- **p2 = p4;**
p2 es un contenedor de C, p4 es un tipo C pero en un contenedor tipo A, no se esta realizando el downcast
- **p2 = (ClassC)p1;**
p2 es un contenedor de C, p1 contiene un tipo B, no se puede hacer el cast
- **p1 = (ClassB)p3;**
p1 es un contenedor de B, p3 es un tipo B pero en contenedor A, es permitido haciendo el cast
- **p2 = (ClassC)p4;**
p2 es un contenedor de C, p4 es un C en contenedor A, es permitido haciendo el cast

29. Which three options correctly describe the relationship between the classes?

```

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

```

- Class2 has-a v3.

No existe esa relación

- Class1 has-a v2.

No existe esa relación

- **Class2 has-a v2.**

La clase 2 si tiene ese atributo

- **Class3 has-a v1.**

La clase 3 tiene ese atributo debido a que contiene una clase 2, lo que le entrega un clase 1 y a su vez el v1

- Class2 has-a Class3.

No existe esa relación

- **Class2 has-a Class1.**

Si tiene ese atributo

30. What is the result?

```

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.

31. Which two possible outputs?

```

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");
    }
}

```

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

- 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.

No hay manejo de excepción, el programa se interrumpe

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

Sí existe al menos una impresión

- Before if clause After if clause.

32. What is the result?

```
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.

El break se encuentra en case White

- Found Red Found Blue Found White Found Default.

33. What is the result?

```
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

i se define en main, posteriormente se crea una copia para el método m, al finalizar m la copia de i es eliminada, el flujo en main conserva el valor de i en 12.

- 19
- Compilation fails

34. Which is true?

```
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.    }
```

- If line 10 is removed, the compilation succeeds.

Barn2 es un contenedor Barn, build 1 es un building, con el downcasting se puede almacenar

- If line 11 is removed, the compilation succeeds.

Obj es un contenedor Object, build1 es subclase, se puede almacenar

- **If line 12 is removed, the compilation succeeds.**

str1 es un Contenedor String, build1 no puede ser casteado, no hay relación

- If line 13 is removed, the compilation succeeds.

build2 es contenedor Building, barn es un hijo, puede ser almacenado

- More than one line must be removed for compilation to succeed.

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

```

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

Al ejecutarse fox ya no se ejecuta la siguiente sentencia relacionada a ese grupo de condicionales

- The fox lazy**
- Quick fox over lazy

33 es diferente de 0

36. What is the result?

```

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 =
19.         id; }
20. }
21. public class EmployeeTest {
22.     public static void main(String[] args)
23.     {
24.         Employee e = new Employee("4321");
25.         System.out.println(e.empID);
26.     }

```

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

El constructor en 18 se encuentra incompleto, no se pasa el nombre de la persona y existen comillas sin uso?

37. Which code fragment is illegal?

- Class Base1 { abstract class Abs1 { } }
- La clase abstracta puede estar anidada dentro de otra clase
- Abstract class Abs2 { void doit() { } }
- Una clase abstracta puede tener métodos concretos
- class Base2 { abstract class Abs3 extends Base2 { } }
- **class Base3 { abstract int var1 = 89; }**
- No se puede declarar una variable de instancia como abstract.

38. What is the result?

```

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**

En el primer print se realiza la concatenación total debido a la presencia del string, en el segundo print se hace la concatenación debido al string pero antes se opera el producto.

- Result: 215 Result: 215
- Compilation fails.

39. What is the result?

```
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.**

Retorna true pues m1 no tiene un nombre null, y no cumple con lo esperado

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

40. Which one is valid as a replacement for foo?

```
public static void main(String[] args) {
    Boolean b1 = true;
    Boolean b2 = false;
    int i = 0;
    while (foo) { }
}
```

- b1.compareTo(b2)
No retorna un booleano, retorna int
- i = 1
No retorna un booleano
- i == 2? -1:0
No retorna un booleano
- **foo.equals("bar")**