

20231003_01 _ LISTO

What changes will make this code compile?

```
1 package com.curso.martes3;
2
3 class X {
4
5     X() {
6     }
7
8     private void one() {
9     }
10 }
11
12
13 public class Y extends X {
14
15     Y() {
16     }
17
18     private void two() {
19
20         one();
21     }
22
23
24     public static void main(String[] args) {
25
26         new Y().two();
27     }
28
29 }
30 }
```

Notas:

- Cambio en la línea 8 de **private** por **protected**
- Ejercicio con error de sintaxis
- Opcion correct : Changing the **private** modifier on the declaration of the one() to **protected**

```
1 package com.curso.martes3;
2
3 public class One{
4     void foo(){}
5 }
6 class Two extends One {
7     // int foo(){return 0;} Es valido
8     void foo(){}
9     // public void foo(){} Es valido
10    // protected void foo(){} Es valido p
11 }
12
```

Which three methods, inserted individually at line 14, will correctly complete class Two? (Choose three.)

- `int foo() { /* more code here */ }`
- `void foo() { /* more code here */ }`
- `public void foo() { /* more code here */ }`
- `private void foo() { /* more code here */ }`
- `protected void foo() { /* more code here */ }`

Respuestas válidas:

- `void foo(){}`
- `public void foo(){}`

- protected void foo(){}

What is the result? *

1 punto

```
public class SampleClass{
    public static void main(String[] args){
        AnotherSampleClass asc = new AnotherSampleClass();
        SampleClass sc = new SampleClass();
        sc = asc;
        System.out.println("sc:" + sc.getClass());
        System.out.println("asc:" + asc.getClass());
    }
}
class AnotherSampleClass extends SampleClass{}
```

- ☐ sc: class.AnotherSampleClass asc: class.AnotherSampleClass
- ☐ sc: class.Object asc: class.AnotherSampleClass
- ☐ sc: class.AnotherSampleClass asc: class.SampleClass
- ☐ sc: class.SampleClass asc: class.AnotherSampleClass

Respuesta válida:

sc:class.AnotherSampleClass asc: class.AnotherSampleClass

Given: * (1 Punto)

```
public class Test2 {
    public static void doChange(int[] arr) {
        for(int pos = 0; pos < arr.length; pos++) {
            arr[pos] = arr[pos] + 1;
        }
    }
    public static void main (String[] args) {
        int[] arr = (10, 20, 30);
        doChange(arr);
        for(int x : arr) {
            System.out.print(x + ",");
        }
        doChange(arr[0], arr[1], arr[2]);
        System.out.print(arr[0] + ", " + arr[1] + ", " + arr[2]);
    }
}
```

Notas:

- Error de compilación: no hay respuesta correcta.

int[] arr = (10, 20, 30);
por los signos ()

```
doChange(arr[0], arr[1], arr[2]);
System.out.print(arr[0] + ", " + arr[1] + ", " + arr[2]);
```

por los 3 arrays

-Correcta: `doChange(new int[]{arr[0],arr[1],arr[2]});|`

- Versiones de la versión correcta del código:

```
int[] arr = {10,20,30};  
doChange(arr);
```

versión 1:

```
for(int x:arr) {  
    System.out.print(x+", "); //11,21,31
```

versión 2:

```
int[] arr = new int[3];  
arr[0] = 10;  
arr[1] = 20;  
arr[2] = 30;
```

`doChange(new int[]{arr[0],arr[1],arr[2]});|`

Revisión de problema que tiene la posibilidad de no estar completamente correcto.

Problema con modificadores de acceso: Estos controlan la visibilidad y el alcance de clases, métodos y variables. Los principales modificadores de acceso son:

1. **Public (public):**
2. **Protected (protected):**
3. **Default (paquete privado, sin modificador explícito)**
4. **Private (private)::**

Problema herencia:

Se presenta un error de sintaxis afectando la herencia, pero la herencia en programación orientada a objetos surge como una forma de reutilización de código y organización lógica.

Problema con inicialización de Arrays,

1. **Declaración e Inicialización:**
 - Los arrays deben ser declarados y luego inicializados antes de usarlos.
 - Declaración: `int[] array;`
 - Inicialización: `array = new int[10];` o `int[] array = {1, 2, 3};`
2. **Error Común:**
 - Acceder a un array sin inicializarlo resulta en un error de compilación.
 - Ejemplo: `int[] array; System.out.println(array[0]); // Error`

Uso de `.length` de los Arrays

1. **Propiedad `.length`:**
 - Proporciona el número de elementos en el array.
 - Se utiliza para iterar sobre los elementos del array.
 - Ejemplo: `int[] array = {1, 2, 3};`
`System.out.println(array.length); // Output: 3`
2. **Error Común:**
 - Intentar acceder a un índice fuera del rango del array lanza `ArrayIndexOutOfBoundsException`.
 - Ejemplo: `int[] array = {1, 2, 3}; System.out.println(array[3]);`
`// Error`

Uso del `.length` en un `for-each`

1. `for-each` Loop:

- Itera automáticamente sobre todos los elementos del array sin necesidad de un índice.
- Ejemplo: `int[] array = {1, 2, 3}; for (int num : array) { System.out.println(num); }`

2. Ventajas:

- Evita errores de índice fuera de límites.
- Simplifica el código al eliminar la necesidad de manejar índices manualmente.

Estos puntos resumen los aspectos clave sobre la inicialización de arrays, el uso de `.length`, y cómo aprovechar `for-each` para iterar de manera segura sobre arrays en Java.

Explicación objetos inmutables: Es que su estado no puede cambiar después de haber sido creado

20231009_02

```
1 package p3;  
2 import p1.DoInterface;
```

Recordatorio: **package** siempre va antes de **import**

Which one of these is a proper definition of a class TestClass that cannot be sub-classed?

Please select 1 option

- ☒ `final class TestClass { }`
- ☐ `abstract class TestClass { }`
- ☐ `native class TestClass { }`
- ☐ `static class TestClass { }`
- ☐ `private class TestClass { }`

Respuesta: `final class TestClass{ }`

Consider the following classes:
class A implements Runnable{ ...}
class B extends A implements Observer { ...}
(Assume that Observer has no relation to Runnable.)

and the declarations :

```
A a = new A() ;  
B b = new B();
```

Which of the following Java code fragments will compile and execute without throwing exceptions?

Please select 2 options

- ☐ Object o = a; Runnable r = o;
- ☒ Object o = a; Runnable r = (Runnable) o;
- ☐ Object o = a; Observer ob = (Observer) o ;
- ☐ Object o = b; Observer o2 = o;
- ☒ Object o = b; Runnable r = (Runnable) b;

Respuesta seleccionadas en verde.

Problema donde se utiliza método static o de instancia con la misma firma, elección de respuestas correctas según el código.

Which of the following method definitions will prevent overriding of that method?

Please select 4 options

- ☒ public final void m1()
- ☒ public static void m1()
- ☒ public static final void m1()
- ☐ public abstract void m1()
- ☒ private void m1()

Consider the following classes :

```
class A{  
    public static void sM1() { System.out.println("In base static"); }  
}  
  
class B extends A{  
Line 1 -> // public static void sM1() { System.out.println("In sub static"); }  
Line 2 -> // public void sM1() { System.out.println("In sub non-static"); }  
}
```

Which of the following statements are true?

Please select 2 options

- ☐ class B will not compile if line 1 is uncommented.
- ☒ class B will not compile if line 2 is uncommented.
- ☒ class B will not compile if line 1 and 2 are both uncommented.
- ☐ Only the second option is correct.
- ☐ Only the third option is correct.

What, if anything, is wrong with the following code?

```
//Filename: TestClass.java
class TestClass implements T1, T2{
    public void m1(){
    }
    interface T1{
        int VALUE = 1;
        void m1();
    }
    interface T2{
        int VALUE = 2;
        void m1();
    }
}
```

Please select 1 option

- ☐ TestClass cannot implement them both because it leads to ambiguity.
- ☒ There is nothing wrong with the code.
- ☐ The code will work fine only if VALUE is removed from one of the interfaces
- ☐ The code will work fine only if m1() is removed from one of the interfaces.
- ☐ None of the above.

```
12 interface T1 {
13     //public static final
14     int VALUE = 1;
15     //public abstract
16     void m1();
17 }
18
19 interface T2 {
20     int VALUE = 2;
21     void m1();
}
```

Problema de conceptos

Problema con interfaces: Comportamiento Inconsistente: Los elementos de la interfaz no siempre responden de la misma manera a las acciones del usuario, lo que puede generar frustración y errores.

Problema conceptual, opciones que previenen el overriding

1. Palabra Clave **final** (Java) / **sealed** (C#):

- En Java, puedes declarar un método como **final**. Esto evitará que las subclases puedan sobrescribir ese método.
- En C#, la palabra clave **sealed** puede utilizarse en combinación con **override** para prevenir más sobrescrituras.

2. Métodos Estáticos:

- Los métodos estáticos pertenecen a la clase en lugar de a instancias de la clase. Por lo tanto, no pueden ser sobrescritos por subclases.

3. Clases No Heredables:

- Puedes hacer que una clase completa no pueda ser heredada utilizando la palabra clave **final** en Java o **sealed** en C#.

4. Privatización de Métodos:

- Declarar métodos como **private** los hace inaccesibles y no sobrescribibles desde subclases.

5. Uso de Interfaces:

- Las interfaces solo definen contratos sin implementación. Puedes diseñar tu sistema de tal manera que las implementaciones específicas no puedan ser sobrescritas.

Estas opciones permiten diseñar sistemas más seguros y controlados, donde el comportamiento de las clases y sus métodos puede ser protegido contra modificaciones no deseadas.