Taller siete. Clases y métodos abstractos y el operador *instanceOf.*

Programación orientada objetos – 2021-1S

Simón Cuartas Rendón – C.C. 1.037.670.103
Estudiante de estadística

Ejercicio uno. **Declaraciones con clases abstractas.**

Se da el código mostrado en la figura 1.

```
abstract class Prueba {
      abstract void m1(Object o);
}
```

Figura uno. *Código dado para el ejercicio uno.*

Y se deben elegir las declaraciones correctas entre las mostradas en la figura 2.

A.
```
class A extends Prueba {
      void m1 (String o) { }
}
```

B.
```
class B extends Prueba {
      void m1 (Object o);
}
```

C.
```
abstract class C extends Prueba {
      void m1 (String o);
}
```

D.
```
abstract class D extends Prueba {
      void m1 (String o) { }
}
```

Figura dos. *Opciones de respuesta para el ejercicio uno.*

A. Es incorrecta, ya que el método `m1()` que hereda de su padre `Prueba` se está sobrescribiendo este método en el sentido que inicialmente recibe un `Object` pero en `A` se le está dando un `String`, por lo que no se está incorporando realmente el método que obliga su padre. Así, para solucionarlo, hay que cambiar el argumento del método o hacer de `A` una clase abstracta.

B. Hay un error con esta declaración, ya que al ser `B` una subclase de `Prueba`, espera que se le defina el código, y esta no tiene más hijos según lo mostrado, por lo que visto así es incorrecto.

C. Es incorrecto, ya que si bien la clase `C` es abstracta, por lo cual no se vería obligada a incorporar al método `m1()` definido en `Prueba`, el error se genera en la definición de su método

`m1()`, pues no se le está dando un cuerpo, por lo que debería ser un método abstracto y en este caso no se le asignó como tal.

D. Es correcta, pues en la `D` al ser una clase abstracta, no tiene la obligación de definir los métodos de su padre `Prueba`, que es justamente lo que ocurre. Además, para su método `m1()` que recibe como argumento a un objeto de tipo `String`, siendo este un método no abstracto, se le está realizando una definición (en este caso una definición vacía, lo cual sigue siendo válido).

Ejercicio dos. **Preguntas varias.**

a. Explique por qué la clase **`Instrumento`** debe definirse como abstracta y qué la diferencia de una clase normal en la figura 3.

```
public abstract  class Instrumento
{
    private String tipo;
    public Instrumento(String tipo) { this.tipo = tipo; }
    public String getTipo()    { return tipo;}

    public abstract void Tocar();
    public abstract void Afinar();
}

public class Saxofon extends Instrumento
{
            public Saxofon(String tipo) { super(tipo); }
            public void Tocar() { System.out.println("Tocando Saxofon");}
            public void Afinar() {System.out.println("Afinando Saxofon");}
}

public class Guitarra extends Instrumento {
            public Guitarra(String tipo) { super(tipo); }
            public void Tocar() {System.out.println("Tocando Guitarra");}
            public void Afinar() {System.out.println("Afinando Guitarra");}
}
```

Figura 3. Código de referencia para el punto 2A.

En este caso se debe observar que los métodos `Tocar()` y `Afinar()` están siendo definidos únicamente y que carecen de código, lo que los hace **métodos abstractos,** lo cual requiere que la clase que los acoge, es decir, `Instrumentos`, sea definida como **abstracta,** o de lo contrario, se tendría un error de compilación.

Además, esta característica relacionada con los métodos permite justamente diferenciarla de una clase normal, en la cual no es posible dejar a los métodos sin código como sucede en este ejemplo, y esto va a representar una ventaja ya que va a facilitar los procesos de herencia que se hagan con una superclase.

b. Elabore una clase denominada **`Piano`** heredada de **`Instrumento`**, añada un constructor y redefina para ella los métodos **`Tocar()`** y **`Afinar()`**.
El código asociado a este ejercicio es el siguiente;

```
public class Piano extends Instrumento {

    public Piano(String tipo) {
            super(tipo);
    }

    public void Tocar() {
            System.out.println("Tocando piano");
    }

    public void Afinar() {
            System.out.println("Afinando piano");
    }

}
```

c. ¿Cuáles de las líneas mostradas en la figura cuatro no pueden ser ejecutadas?

```
public class Test {
    public static void main(String[] args) {
            Instrumento x;
            x = new Instrumento();
            x = new Saxofon("xxxxx");
            x = new Guitarra();
            x = new Piano("xxxxxxx");
    }
}
```

Figura 4. Código asociado al ejercicio 2C. Las líneas de código señaladas en recuadros rojos no pueden ser ejecutadas.

En este caso no van a poder ser ejecutadas las líneas encerradas en rojo en la figura cuatro. A continuación se explica el porqué de cada caso.

- `x = new Instrumento();`
  De acuerdo con el código dado, existe un constructor para el método instrumento que solicita al usuario que pase como parámetro un objeto de tipo **`String`**, esto es, **`Instrumento(String tipo)`**. Así, el compilador no va a agregar el constructor por defecto y por lo tanto no va a ser posible crear un nuevo objeto de tipo **`Instrumento`** si no se emplea el constructor definido.
- `x = new Guitarra();`
  En este caso sucede una situación análoga a la explicada anteriormente, ya que la clase **`Guitarra`** carece de un constructor que no tenga ningún parámetro, por lo que el compilador no podrá ejecutar el código adecuadamente.

d. ¿Qué imprime el programa mostrado en la figura cinco?
El código dado para este caso es el de la figura cinco:

```
public class Test {
    public static void main(String[] args) {
            Instrumento x;
            x = new Saxofon("xxxxx");  x.Tocar();
            x = new Guitarra("xxxxx");  x.Tocar();
            x = new Piano("xxxxxxx");  x.Tocar();
    }
}
```

Figura 5. Código del ejercicio 2D.

El resultado de la ejecución es el siguiente:

```
Tocando saxofón
Tocando Guitarra
Tocando piano
```

- La primera línea, *Tocando saxofón*, se da porque inicialmente se está llevando al apuntador de tipo `Instrumento x` a un objeto de tipo `Saxofón`, por lo que se ejecuta el método en su propio contexto.
- El proceso con las siguientes dos líneas es análogo.

Ejercicio tres. **Responder a partir del código dado.**

1. ¿Qué sucede si se define el método **`explotar()`** de la clase Estrella como se indica a continuación? Explique su respuesta.
   Se generaría un error, ya que el método `explotar()` está siendo definido como **abstracto,** de modo que no tiene permitido contener código, pero en este caso se le está dando un cuerpo a este código para que imprima por pantalla "*Estrella explotar*". En este caso, se debería asignar el método como no abstracto para que todos sus subclases lo puedan incorporar, o bien, dejar el método sin código y definirlo según corresponda en cada subclase de `Estrella`.

2. ¿Qué significa que los métodos **`tipoCuerpo2()`** y **`getID()`** de la clase **`ObjetoAstronomicoExtraSolar`**, no se definan como *abstract*? ¿Podría considerarse esta situación un error? Explique.
   Esto no es un error, pues lo que se quiere al dejar ese par de métodos como métodos no abstractos es que todas las subclases de `ObjetoAstronomicoExtraSolar` hereden ese método y lo ejecuten de esa misma manera, o bien, que lo puedan sobrescribir u ocultar según corresponda.

3. Si se define como abstracta la clase **`ObjetoAstronomicoExtraSolar`**, como se indica a continuación, ¿puede considerarse un error definir una clase abstracta sin métodos abstractos? Explique.

```
abstract class ObjetoAstronomicoExtraSolar {
private int ID;

 public void tipoCuerpo2() {
 System.out.println("Extrasolar");
 }

 public int getID() {
 return this.ID;
 }
}
```

No es un error, ya que una clase abstracta puede o no tener métodos abstractos. Puede ser que el diseñador de este código simplemente quería que no se pudiesen crear objetos de la clase `ObjetoAstronomicoExtraSolar` y facilitar la herencia de sus métodos y sus atributos a sus subclases.

4. **Explique por qué el arreglo `oa` (línea 19) hace referencia a una clase abstracta y sin embargo, en la línea 25 se invoca el método correspondiente a cada clase derivada.**
   Porque al incorporar los elementos que van a conformar el arreglo dado, estos son de clases que sí pueden tener objetos al no ser abstractos, estos son, `Galaxia`, `Nova` y `Supernova`. A continuación, cuando se lee la descripción de cada uno de ellos, se realiza un proceso de ligadura dinámica, por lo que finalmente se toma el método en el contexto de cada objeto cuando llega su turno en el ciclo *for*.

5. **¿Por qué la línea 29 imprime "Soy una Super Nova" sabiendo que el arreglo `oa` en esa posición fue inicializado con un objeto de tipo `Galaxia`?**
   Porque justo en la línea anterior se hizo la siguiente declaración:

   ```
   oa[0] = oa[2];
   ```

   Con la cual se hizo un cambio un cambio al apuntador hacia el tercer elemento del arreglo oa, que es del tipo `SuperNova`, lo cual explica el resultado obtenido.

6. **¿Por qué en la clase Estrella no se define el método `descripcion()` si la superclase lo está solicitando, ya que en este método descripción en abstracto?**
   Porque `Estrella` es una clase **abstracta**, de manera que no tiene la obligación de implementar los métodos de su superclase.

7. **¿Qué sucede si el método `tipoCuerpo1()` de la clase `Galaxia` se define como privado? ¿Por qué se genera error?**
   Se debe notar que el método en cuestión fue definido en la superclase de `Galaxia`, `ObjetoAstronomicoExtraSolar` con visibilidad **public**, lo cual inhibe que se le cambie la visibilidad a cualquier otra opción ya que un método heredado no puede ser más privado que el método definido en la superclase.

8. **¿Por qué la clase `Nova` no define el método `tipoCuerpo1()`? ¿Se podría definir? Si lo define, ¿qué interpreta de esta situación?**

Porque ese método no es abstracto, y como se puede ver, tiene un código asociado, por lo que al crearse esta subclase, hereda ese mismo método de la misma forma en que fue definido en Estrella.

9. ¿Qué imprime la línea 9? ¿Por qué se puede llamar al método **`toString()`** si la clase Galaxia no lo define y su papá **`ObjetoAstronomicoExtraSolar`** tampoco?
   En este caso se regresa la ubicación en memoria que tiene el objeto g y su tipo antes del signo arroba. Además, se debe recordar que este es un método de la clase `Object`, el cual es super-clase de cualquier método creado, y este incluye el método `toString()`, lo cual permite que pueda ser llamado y ejecutado.

10. ¿Por qué en la línea 11 se puede crear un puntero **`obN`** de tipo **`ObjetoAstronomicoExtraSolar`** si esta es una clase abstracta?
    Porque simplemente se está generando una referencia, mas no se está creando un objeto de esta clase. De hecho, se puede observar que previamente se había definido el objeto al cual se quiere referenciar con un puntero del tipo `ObjetoAstronomicoExtraSolar`, y este objeto es del tipo `Nova`. Como tal no se está creando un objeto de esta clase abstracta por lo que la declaración es lícita.

11. ¿Las siguientes instrucciones (instrucciones en las líneas B y C) son válidas? Explique..

```
A. Nova nova = new Nova();
B. ObjetoAstronomicoExtraSolar ob = new ObjetoAstronomicoExtraSolar();
C. ObjetoAstronomicoExtraSolar oa = nova;
```

    La declaración de la línea B es incorrecta ya que está creando un objeto de la clase `ObjetoAstronomicoExtraSolar`, lo cual no es posible por ser esta una clase abstracta. En cambio, la declaración de la línea C sí es posible porque solo se está generando un puntero a un objeto de la clase `Nova` que, al no ser abstracto, puede darse.

12. Explique por qué (ver código a continuación) la siguiente instrucción en la línea B es correcta y la instrucción en la línea C es incorrecta. Omitiendo la instrucción en la línea C, ¿qué se imprime por pantalla? Explique su respuesta.

```
A. Nova nova = new Nova();
B. ObjetoAstronomicoExtraSolar oa = nova;
C. oa.explotar();
D. ((Nova) oa).explotar();
```

    Porque con el puntero `ObjetoAstronomicoExtraSolar` no se conoce el método `explotar()` que se está tratando de procesar en la línea C, dando lugar a un error. En cambio, al hacer un cast explícito al tipo del objeto de la clase `oa`, en un proceso de especialización, se va a poder ejecutar el método, pues este sí está definido para la clase `Nova`.

13. ¿Por qué la línea 15 imprime true? ¿Para cualquier objeto que se cree siempre imprimirá lo mismo? ¿Qué imprimen las siguientes líneas? ¿Por qué?

```
obN = null;
System.out.println(obN instanceof Object);
System.out.println("" + obN instanceof Object);
```

La línea quince imprime ***true*** ya que por polimorfismo, todo objeto es de clase que fue inicialmente declarado y de la clase **Object**, ya que esta es superclase para cualquier otra clase en Java, lo que quiere decir que con cualquier otro objeto que se le pase a la línea quince en lugar de **obN** se va a obtener un ***true***. Respecto a las líneas dadas anteriormente, se va a obtener lo siguiente:

```
true
true
```

Lo cual ocurre dado que, si bien un **null** no es objeto de nadie, en este caso declarado previamente un objeto, el cual ya fue definido como un objeto de la clase **Nova**, por lo que el resultado va a ser ***true***. En la siguiente línea es lo mismo, pues antes de la evaluación lógica se agrega un String sin elementos.

14. Agregue el siguiente constructor, ¿Se genera error? ¿Se puede colocar constructores a clases abstractas? ¿Qué sentido tiene?

```
public ObjetoAstronomicoExtraSolar () {
this.ID = 4;
 this.tipoCuerpo2();
}
```

No se va a generar un error, ya que una clase abstracta sí va a poder tener constructores en el sentido de que le va a permitir a sus subclases definir en sus propios constructores de manera ágil los atributos que heredan de la superclase abstracta.

15. Suponga que agrega la clase **EnanaBlanca** como se indica a continuación. ¿Qué se puede concluir de esta nueva situación? ¿Qué errores se presentan y cómo podría corregirlos?

```
class EnanaBlanca extends Estrella
{ void agotarCombustible() {
 System.out.println("Enana blanca muere");
 }
}
```

El problema se da porque esta nueva clase no está incorporando los métodos abstractos de sus dos padres: **Estrella** y **ObjectoAstronomicoExtaSolar**, los cuales son **explotar()** y **descripción()** respectivamente, de modo que se tienen que agregar para que el código pueda compilar. Así, este debería quedar de la siguiente manera:

```
class EnanaBlanca extends Estrella {
    void agotarCombustible() {
            System.out.println("Enana blanca muere");
    }

    void explotar() {
            System.out.println("¡Baaaam!");
```

```
    }

    void descripcion() {
            System.out.println("Soy una enana blanca");
    }
}
```

16. ¿Por qué las líneas 16 y 17 imprimen false y false respectivamente?
    En ambos sucede que, a diferencia de la situación similar discutida previamente, el apuntador `obN` no se dirige a ningún objeto de las clases `SuperNova` y `Galaxia`, lo cual hace que se dé el *`false`* en ambas situaciones.