

Curso de Java a distancia

Clase 32: Interfaces en el API de Java

En el concepto anterior aprendimos como crear y consumir nuestras propias interfaces. Ahora veremos que el API de Java hace un uso amplio de interfaces.

Cuando vimos la librería de componentes visuales Swing utilizamos interfaces para la captura de eventos.

Ahora conociendo como se crea una interface repasaremos algunos problemas donde el API de Java las utiliza.

Problema:

Implementar un formulario que muestre un menú de opciones y un botón. Permitir finalizar el programa cuando se selecciona el MenuItem o el JButton.

Clase: Formulario

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;

public class Formulario extends JFrame implements ActionListener{
    JButton boton1;
    JMenuBar mb1;
    JMenu menu1;
    JMenuItem menuItem1;

    public Formulario() {
        setLayout(null);
        boton1=new JButton("Salir");
        boton1.setBounds(650, 480, 100, 30);
        add(boton1);
        boton1.addActionListener(this);

        mb1=new JMenuBar();
        setJMenuBar(mb1);
        menu1=new JMenu("Opciones");
        mb1.add(menu1);
```

```

        menuItem1=new JMenuItem("Salir");
        menu1.add(menuItem1);
        menuItem1.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        if (boton1==e.getSource())
            System.exit(0);
        if (menuItem1==e.getSource())
            System.exit(0);
    }

    public static void main(String[] args) {
        Formulario formulario1=new Formulario();
        formulario1.setBounds(0, 0, 800, 600);
        formulario1.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        formulario1.setVisible(true);
    }
}

```

En el problema implementamos la interface 'ActionListener':

```
public class Formulario extends JFrame implements ActionListener{
```

La interface 'ActionListener' se encuentra almacenada en el paquete:

```
java.awt.event:
import java.awt.event.ActionListener;
```

Luego la clase 'Formulario' al implementar la interface 'ActionListener' debe codificar el método 'actionPerformed':

```

    public void actionPerformed(ActionEvent e) {
        if (boton1==e.getSource())
            System.exit(0);
        if (menuItem1==e.getSource())
            System.exit(0);
    }

```

Para que los objetos 'boton1' y 'menuItem1' puedan llamar al método 'actionPerformed' debemos pasar la referencia del objeto formulario1 llamando al método 'addActionListener':

```

        boton1.addActionListener(this);
        menuItem1.addActionListener(this);

```

Recordemos que el mismo problema lo podemos resolver creando objetos de clases anónimas que implementan la interfaz 'ActionListener' cuando llamamos a los métodos 'addActionListener':

Clase: Formulario

```

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;

public class Formulario extends JFrame {
    JButton boton1;

```

```

JMenuBar mb1;
JMenu menu1;
JMenuItem menuItem1;

public Formulario() {
    setLayout(null);
    boton1 = new JButton("Salir");
    boton1.setBounds(650, 480, 100, 30);
    add(boton1);
    boton1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    });

    mb1 = new JMenuBar();
    setJMenuBar(mb1);
    menu1 = new JMenu("Opciones");
    mb1.add(menu1);
    menuItem1 = new JMenuItem("Salir");
    menu1.add(menuItem1);
    menuItem1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    });
}

public static void main(String[] args) {
    Formulario formulario1 = new Formulario();
    formulario1.setBounds(0, 0, 800, 600);
    formulario1.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    formulario1.setVisible(true);
}
}

```

De esta forma la clase 'Formulario' no implementa la interfaz 'ActionListener', sino las clases anónimas que declaramos cuando llamamos a los métodos 'addActionListener' del JButton y el JMenuItem:

```

    boton1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    });

    menuItem1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    });

```

Podemos analizar el código fuente de las API de Java en el sitio github:

ActionListener

KeyListener

WindowListener

MouseListener

MouseListener

ItemListener

FocusListener

Una interfaz puede heredar de otra interfaz:

```
public interface ActionListener extends EventListener {  
  
    /**  
     * Invoked when an action occurs.  
     */  
    public void actionPerformed(ActionEvent e);  
  
}
```

Hay que tener en cuenta que una interfaz en Java puede heredar inclusive de múltiples interfaces, a diferencia de una clase que puede heredar solo de una.

Problema:

Declarar una clase 'Persona' y sus atributos 'nombre' y 'edad'. Implementar la interface 'Comparable' con tipo genérico.

Crear un ArrayList de tipo 'Persona' y ordenar las personas en forma alfabética.

Clase: Persona

```
public class Persona implements Comparable<Persona> {  
    private String nombre;  
    private int edad;  
  
    public Persona(String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
  
    String retornarNombre() {  
        return nombre;  
    }  
  
    int retornarEdad() {  
        return edad;  
    }  
  
    public int compareTo(Persona o) {  
        return nombre.compareTo(o.nombre);  
    }  
}
```

Existen interfaces genéricas que debemos indicar el tipo de dato que procesa, esto sucede con la interface 'Comparable':

```
public class Persona implements Comparable<Persona> {
```

La interface 'Comparable' debe implementar el método 'compareTo':

```
    public int compareTo(Persona o) {  
        return nombre.compareTo(o.nombre);
```

```
}
```

Para probar la clase 'Persona':

Clase: PruebaPersona

```
import java.util.ArrayList;
import java.util.Collections;

public class PruebaPersona {

    public static void main(String[] args) {
        ArrayList<Persona> lista1 = new ArrayList<Persona>();
        lista1.add(new Persona("Juan", 33));
        lista1.add(new Persona("Ana", 22));
        lista1.add(new Persona("Pedro", 50));

        for (Persona per : lista1)
            System.out.println(per.retornarNombre() + "-" + per.retornarEdad());

        Collections.sort(lista1);

        System.out.println("Lista luego de ordenar en forma alfabética.");
        for (Persona per : lista1)
            System.out.println(per.retornarNombre() + "-" + per.retornarEdad());
    }
}
```

Para ordenar la lista de personas llamamos al método estático 'sort' de la clase 'Collections'. Dicho método requiere que la lista implemente la interface genérica 'Comparable'.

Clase Object y sobrescritura del método toString

En Java por defecto toda clase hereda de 'Object' si no especificamos un 'extends'.

La clase Object tiene entre otros un método llamado 'toString', por defecto retorna el nombre de la clase que se ha instanciado y la dirección de memoria del mismo. Veamos con un ejemplo dicha información.

Clase: Formulario

```
public class Persona {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String retornarNombre() {
        return nombre;
    }

    public void fijarNombre(String nombre) {
```

```
        this.nombre = nombre;
    }

    public int retornarEdad() {
        return edad;
    }

    public void fijarEdad(int edad) {
        this.edad = edad;
    }

    public static void main(String[] ar) {
        Persona persona1 = new Persona("Juan", 33);
        System.out.println(persona1.toString());
    }
}
```

No indicamos el nombre de la clase que hereda 'Persona':

```
public class Persona {
```

Pero el compilador de Java reformula la clase con la siguiente sintaxis:

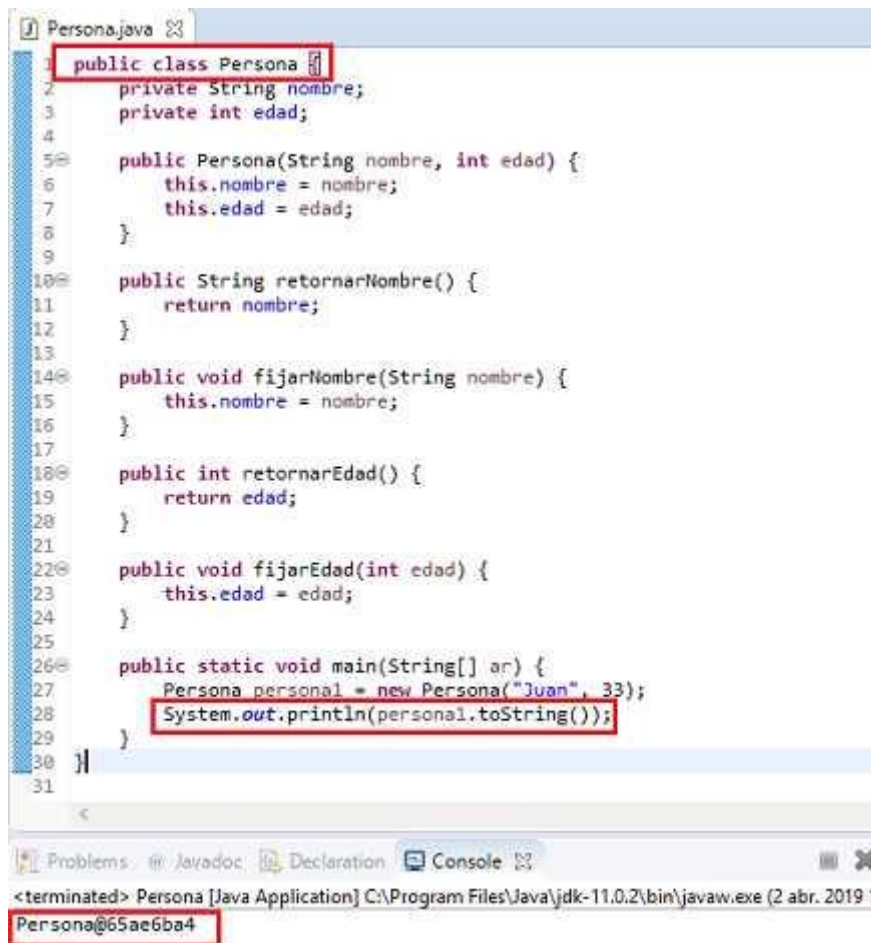
```
public class Persona extends Object {
```

Luego significa que la clase 'Persona' hereda todos los métodos de la clase 'Object' y uno de ellos se llama 'toString'.

Cuando creamos un objeto de la clase 'Persona' luego podemos llamar al método 'toString' que es un método heredado:

```
Persona persona1 = new Persona("Juan", 33);
System.out.println(persona1.toString());
```

El resultado es:



```
1 public class Persona {
2     private String nombre;
3     private int edad;
4
5     public Persona(String nombre, int edad) {
6         this.nombre = nombre;
7         this.edad = edad;
8     }
9
10    public String retornarNombre() {
11        return nombre;
12    }
13
14    public void fijarNombre(String nombre) {
15        this.nombre = nombre;
16    }
17
18    public int retornarEdad() {
19        return edad;
20    }
21
22    public void fijarEdad(int edad) {
23        this.edad = edad;
24    }
25
26    public static void main(String[] ar) {
27        Persona personal = new Persona("Juan", 33);
28        System.out.println(personal.toString());
29    }
30 }
31
```

Problems Javadoc Declaration Console

<terminated> Persona [Java Application] C:\Program Files\Java\jdk-11.0.2\bin\javaw.exe (2 abr. 2019 1

Persona@65ae6ba4

Es muy común sobrescribir el método 'toString' en nuestras clases para que nos provea una información más significativa. Modifiquemos la clase Persona sobrescribiendo el método toString:

```
public class Persona {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String retornarNombre() {
        return nombre;
    }

    public void fijarNombre(String nombre) {
        this.nombre = nombre;
    }

    public int retornarEdad() {
        return edad;
    }

    public void fijarEdad(int edad) {
        this.edad = edad;
    }

    public String toString() {
        return "Nombre:" + nombre + "- Edad:" + edad;
    }
}
```

```

    }

    public static void main(String[] ar) {
        Persona persona1 = new Persona("Juan", 33);
        System.out.println(persona1.toString());
    }
}

```

Hemos sobrescrito el método 'toString' para la clase 'Persona', el String a retornar dependerá de las necesidades de nuestra aplicación:

```

public String toString() {
    return "Nombre:" + nombre + "- Edad:" + edad;
}

```

Ahora cuando llamamos al método 'toString' a partir de una instancia de la clase 'Persona' se ejecutará el método implementado por nosotros:

```
System.out.println(persona1.toString());
```

Si queremos por ejemplo que el método 'toString' retorne también el String que genera la clase 'Object' debemos modificar nuestro código con la siguiente sintaxis:

```

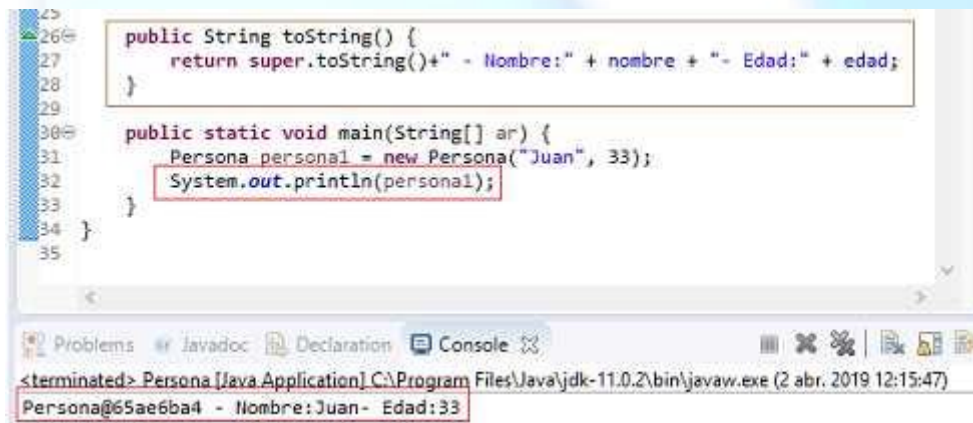
public String toString() {
    return super.toString()+" - Nombre:" + nombre + "- Edad:" + edad;
}

```

Cuando pasamos un objeto al método 'println' y no indicamos el nombre del método, éste accede directamente al método 'toString'. Es común utilizar la sintaxis:

```
System.out.println(persona1);
```

Tenemos como resultado:



```

25
26 public String toString() {
27     return super.toString()+" - Nombre:" + nombre + "- Edad:" + edad;
28 }
29
30 public static void main(String[] ar) {
31     Persona persona1 = new Persona("Juan", 33);
32     System.out.println(persona1);
33 }
34 }
35

```

Problems Javadoc Declaration Console
 <terminated> Persona [Java Application] C:\Program Files\Java\jdk-11.0.2\bin\javaw.exe (2 abr. 2019 12:15:47)
 Persona@65ae6ba4 - Nombre:Juan - Edad:33

API de Java y sobrescritura del método toString

En la librería estándar de Java gran cantidad de clases sobrescriben el método 'toString' brindando una información acorde a su objetivo. Veamos un ejemplo del String que retorna la clase ArrayList.

```
import java.util.ArrayList;
```

```

public class PruebaArrayList {

    public static void main(String[] ar) {
        ArrayList<Integer> lista1 = new ArrayList<Integer>();
    }
}

```

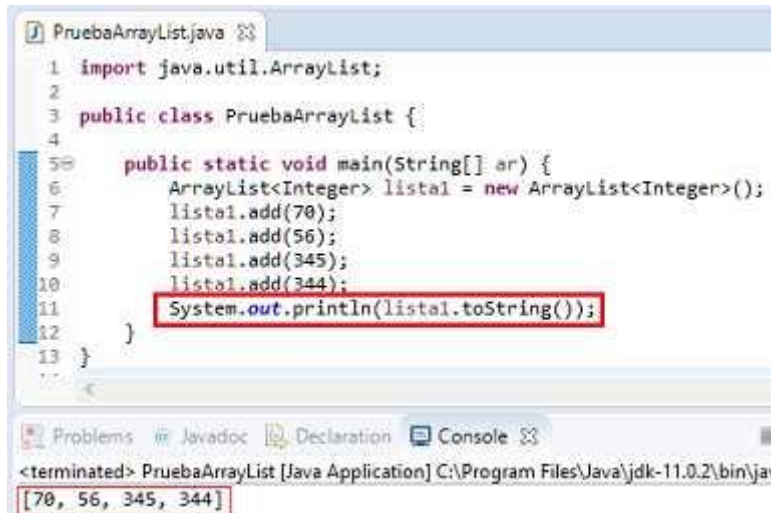


```

        lista1.add(70);
        lista1.add(56);
        lista1.add(345);
        lista1.add(344);
        System.out.println(lista1.toString());
    }
}

```

Nos retorna un String con los elementos del arreglo separados por coma y los caracteres de corchete al principio y al final:



```

1  import java.util.ArrayList;
2
3  public class PruebaArrayList {
4
5      public static void main(String[] ar) {
6          ArrayList<Integer> lista1 = new ArrayList<Integer>();
7          lista1.add(70);
8          lista1.add(56);
9          lista1.add(345);
10         lista1.add(344);
11         System.out.println(lista1.toString());
12     }
13 }

```

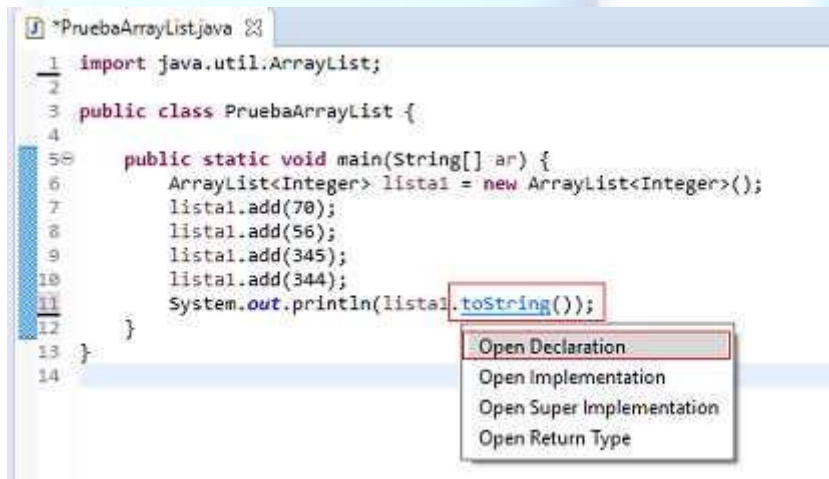
Problems Javadoc Declaration Console

<terminated> PruebaArrayList [Java Application] C:\Program Files\Java\jdk-11.0.2\bin\jav...
[70, 56, 345, 344]

No olvidar que obtenemos el mismo resultado escribiendo:

```
System.out.println(lista1);
```

Si queremos ver como está codificado el método toString de cualquier clase en Eclipse podemos presionar la tecla 'Control' y hacer clic con la flecha del mouse sobre el nombre del método:



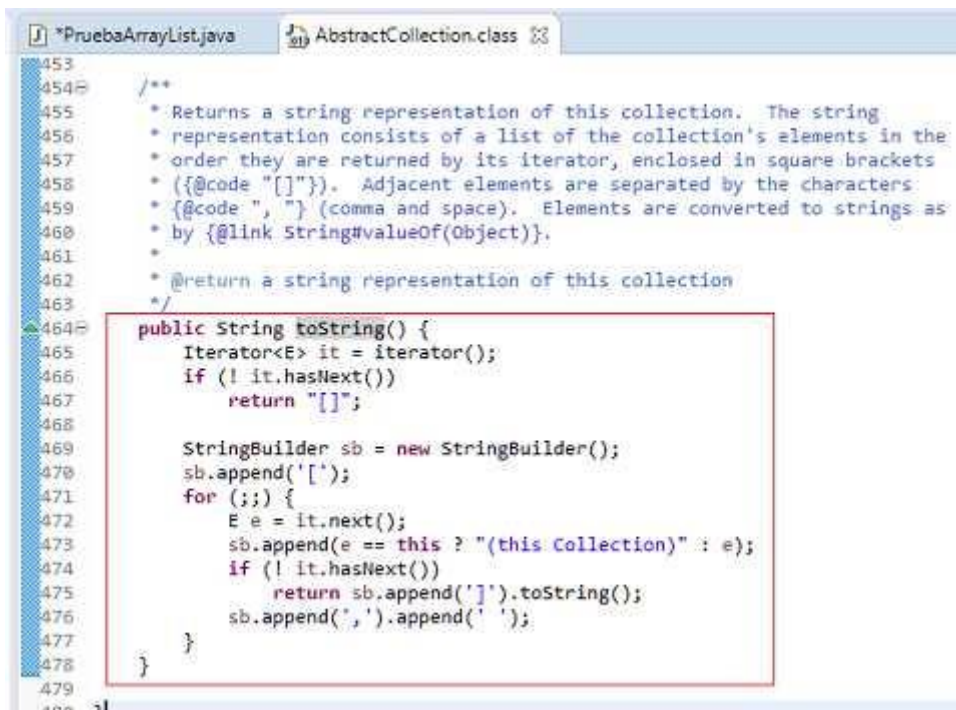
```

1  import java.util.ArrayList;
2
3  public class PruebaArrayList {
4
5      public static void main(String[] ar) {
6          ArrayList<Integer> lista1 = new ArrayList<Integer>();
7          lista1.add(70);
8          lista1.add(56);
9          lista1.add(345);
10         lista1.add(344);
11         System.out.println(lista1.toString());
12     }
13 }
14

```

Open Declaration
Open Implementation
Open Super Implementation
Open Return Type

Se nos abre el código fuente de su implementación:



```
453
454 /**
455  * Returns a string representation of this collection. The string
456  * representation consists of a list of the collection's elements in the
457  * order they are returned by its iterator, enclosed in square brackets
458  * ({@code "["}). Adjacent elements are separated by the characters
459  * {@code ", " (comma and space). Elements are converted to strings as
460  * by {@link String#valueOf(Object)}.
461  *
462  * @return a string representation of this collection
463  */
464 public String toString() {
465     Iterator<E> it = iterator();
466     if (! it.hasNext())
467         return "[]";
468
469     StringBuilder sb = new StringBuilder();
470     sb.append('[');
471     for (;;) {
472         E e = it.next();
473         sb.append(e == this ? "(this Collection)" : e);
474         if (! it.hasNext())
475             return sb.append(']').toString();
476         sb.append(',').append(' ');
477     }
478 }
479
```

Problema propuesto

1. Crear una clase llamada 'Bolillero' que permita generar un vector de 50 enteros comprendidos entre 1 y 50, todos distintos.
Definir el método 'toString' que retorne los 50 valores generados.

Solución

```
public class Bolillero {
    private int[] vec;

    public Bolillero() {
        vec = new int[50];
        generar();
    }

    public void generar() {
        for (int f = 0; f < vec.length; f++) {
            vec[f] = 1 + (int) (Math.random() * 50);
            for (int k = 0; k < f; k++)
                if (vec[f] == vec[k]) {
                    f--;
                    break;
                }
        }
    }

    public String toString() {
        String cad = "[";
        for (int f = 0; f < vec.length; f++)
            cad += vec[f] + " ";
        cad += "]";
        return cad;
    }

    public static void main(String[] ar) {
        Bolillero bolillero1 = new Bolillero();
        System.out.println(bolillero1);
    }
}
```

Clase Object y sobrescritura del método equals

La clase Object tiene un método llamado 'equals' y normalmente las subclases proceden a sobrescribirlo y definir un algoritmo acorde a su objetivo.

La clase Object declara el método 'equals' con la siguiente sintaxis:

```
public boolean equals(Object obj)
```

Veamos un ejemplo que sobrescriba el método equals para una clase cualquiera.

Problema:

Plantear una clase llamada 'Punto' que represente un punto en el plano. Definir los atributos x e y.

Sobrescribir el método 'equals' y definir como lógica que retorne true si los atributos x e y son iguales.

Clase: Punto

```
public class Punto {
    private int x, y;
```

```

public Punto(int x, int y) {
    this.x = x;
    this.y = y;
}

public boolean equals(Object obj) {
    Punto punto = (Punto) obj;
    if (x == punto.x && y == punto.y)
        return true;
    else
        return false;
}

public static void main(String[] ar) {
    Punto punto1 = new Punto(10, 2);
    Punto punto2 = new Punto(10, 2);
    Punto punto3 = new Punto(20, 4);
    if (punto1.equals(punto2))
        System.out.println("punto1 y punto2 pertenecen al mismo punto en el plano.");
    if (punto1.equals(punto3))
        System.out.println("punto1 y punto3 pertenecen al mismo punto en el plano.");
    else
        System.out.println("punto1 y punto3 no pertenecen al mismo punto en el plano.");
}
}

```

La firma del método equals debe ser igual que el declarado en la clase Object:

```

public boolean equals(Object obj) {

```

Lo primero que hacemos es mediante casting convertir el parámetro 'Object' a tipo 'Punto':

```

    Punto punto = (Punto) obj;

```

Luego ya podemos verificar si los atributos x e y coinciden con las referencias x e y del parámetro 'obj':

```

    if (x == punto.x && y == punto.y)
        return true;
    else
        return false;
}

```

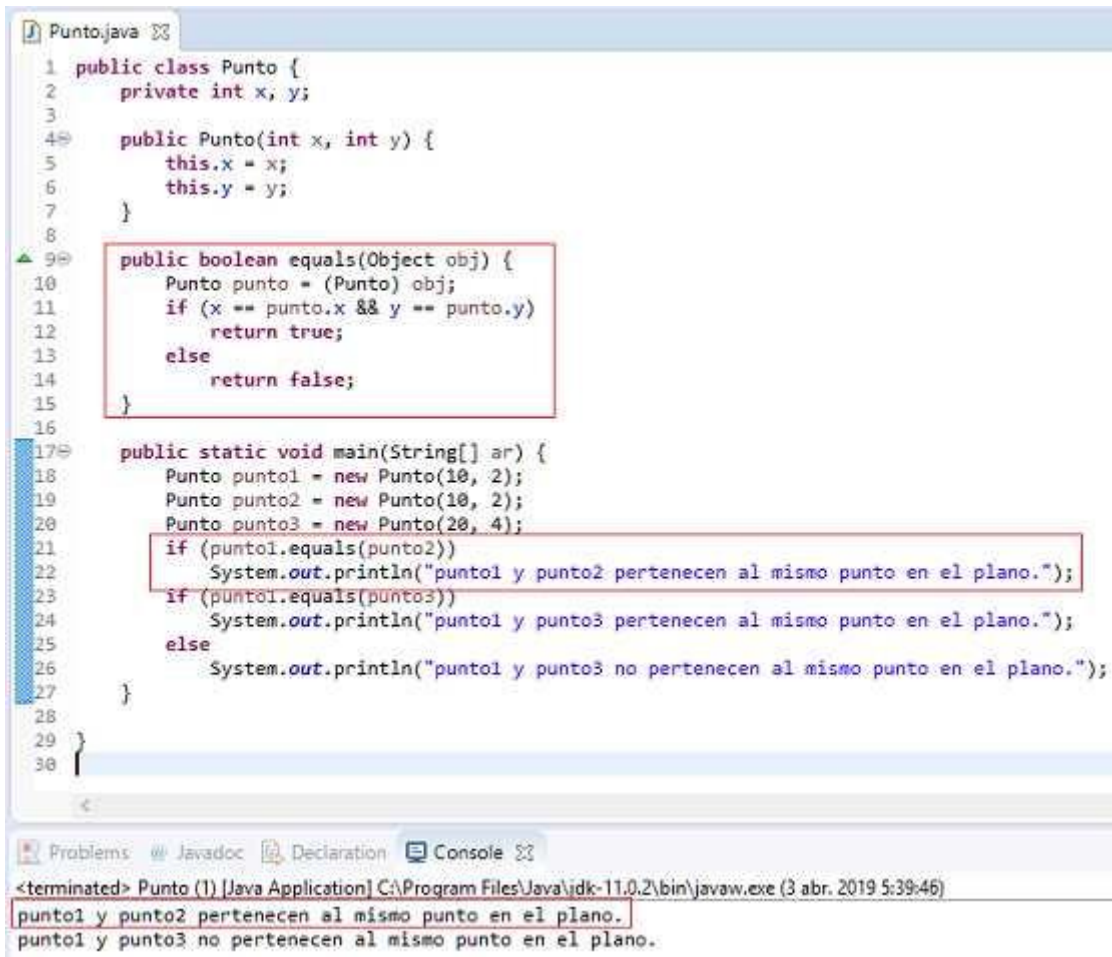
En la main creamos tres objetos de la clase Punto y llamamos al método equals que hemos implementado:

```

public static void main(String[] ar) {
    Punto punto1 = new Punto(10, 2);
    Punto punto2 = new Punto(10, 2);
    Punto punto3 = new Punto(20, 4);
    if (punto1.equals(punto2))
        System.out.println("punto1 y punto2 pertenecen al mismo punto en el plano.");
    if (punto1.equals(punto3))
        System.out.println("punto1 y punto3 pertenecen al mismo punto en el plano.");
    else
        System.out.println("punto1 y punto3 no pertenecen al mismo punto en el plano.");
}
}

```

Luego tenemos como resultado:



```
1 public class Punto {
2     private int x, y;
3
4     public Punto(int x, int y) {
5         this.x = x;
6         this.y = y;
7     }
8
9     public boolean equals(Object obj) {
10        Punto punto = (Punto) obj;
11        if (x == punto.x && y == punto.y)
12            return true;
13        else
14            return false;
15    }
16
17    public static void main(String[] ar) {
18        Punto punto1 = new Punto(10, 2);
19        Punto punto2 = new Punto(10, 2);
20        Punto punto3 = new Punto(20, 4);
21        if (punto1.equals(punto2))
22            System.out.println("punto1 y punto2 pertenecen al mismo punto en el plano.");
23        if (punto1.equals(punto3))
24            System.out.println("punto1 y punto3 pertenecen al mismo punto en el plano.");
25        else
26            System.out.println("punto1 y punto3 no pertenecen al mismo punto en el plano.");
27    }
28 }
29
30 }
```

Problems @ Javadoc Declaration Console

<terminated> Punto (1) [Java Application] C:\Program Files\Java\jdk-11.0.2\bin\javaw.exe (3 abr. 2019 5:39:46)

punto1 y punto2 pertenecen al mismo punto en el plano.

punto1 y punto3 no pertenecen al mismo punto en el plano.

Tener en cuenta que en Java no es lo mismo utilizar el operador `==` ya que en ese caso estamos verificando si las variables apuntan al mismo objeto:

```
Punto punto1 = new Punto(10, 2);
Punto punto2 = new Punto(10, 2);
if (punto1==punto2)
    System.out.println("Apuntan al mismo objeto");
else
    System.out.println("No apuntan al mismo objeto");
```

El if se verifica falso ya que punto1 y punto2 tienen direcciones de objetos distintos.

En cambio si tenemos el código siguiente:

```
Punto punto1 = new Punto(10, 2);
Punto referencia = punto1;
if (punto1==referencia)
    System.out.println("Apuntan al mismo objeto");
else
    System.out.println("No apuntan al mismo objeto");
```

El if se verifica verdadero ya que las variables 'punto1' y 'referencia' almacenan la dirección del mismo objeto que creamos con el operador 'new'.

Hay que tener en cuenta que al método 'equals' nosotros le definimos la semántica de igualdad para dos objetos.

Problema propuesto

1. Crear una clase Persona y definir los atributos nombre, edad y dni.
Sobreescribir el método 'equals' de tal forma que retorne true si los dni son iguales.

Solución

```
public class Persona {  
    private String nombre;  
    private int edad;  
    private String dni;  
  
    public Persona(String nombre, int edad, String dni) {  
        this.nombre = nombre;  
        this.edad = edad;  
        this.dni = dni;  
    }  
  
    public boolean equals(Object obj) {
```



```
Persona persona = (Persona) obj;
if (dni.equals(persona.dni))
    return true;
else
    return false;
}

public static void main(String[] ar) {
    Persona persona1 = new Persona("juan", 22, "20456123");
    Persona persona2 = new Persona("ana", 15, "21100200");
    Persona persona3 = new Persona("juan", 22, "20456123");
    if (persona1.equals(persona2))
        System.out.println("persona1 y persona2 si son la misma persona");
    else
        System.out.println("persona1 y persona2 no son la misma persona");
    if (persona1.equals(persona3))
        System.out.println("persona1 y persona3 si son la misma persona");
}
}
```

Muchas gracias hasta la próxima clase.

Alsina 16 [B1642FNB] San Isidro | Pcia. De Buenos Aires |Argentina |

TEL.: [011] 4742-1532 o [011] 4742-1665 |

www.institutosanisidro.com.ar info@institutosanisidro.com.ar

