INSTITUTO POLITÉNICO NACIONAL

Escuela Superior de Cómputo

Unidad de Aprendizaje:

Programación Orientada a Objetos

Grupo:

2CV8

Profesor:

Daniel Cruz García

Práctica 7

Alumnos:

Luciano Espina Melisa

Fecha de entrega:

20/06/2018

Introducción

La programación orientada a objetos tiene varios niveles de dificultad, los cuales se van observando conforme al progreso de cada tema. Uno de los pilares más importantes de la programación orientada a objetos es el polimorfismo, ya que éste implementa herencia y los términos básicos con los cuales se ha ido trabajando.

Marco teórico

Polimorfismo

El concepto de polimorfismo es en realidad algo muy básico. Realmente, cuando estamos aprendiendo Programación Orientada a Objetos (también conocida por sus siglas POO / OOP) muchos estudiantes nos hacemos un embolado tremendo al tratar de entender el concepto, pero en su base es algo extremadamente sencillo.

El polimorfismo es una relajación del sistema de tipos, de tal manera que una referencia a una clase (atributo, parámetro o declaración local o elemento de un vector) acepta direcciones de objetos de dicha clase y de sus clases derivadas (hijas, nietas, …). [1]

Polimorfismo en objetos

Una función cuyo parámetro se haya declarado de una clase, sólo te aceptará recibir objetos de esa clase. Un array que se ha declarado que es de elementos de una clase determinada, solo aceptará que rellenemos sus casillas con objetos de esa clase declarada.

```
Vehiculo[] misVehiculos = new Vehiculo[3];
```

Esa variable misVehiculos es un array y en ella he declarado que el contenido de las casillas serán objetos de la clase "Vehiculo". Como se ha explicado, en lenguajes fuertemente tipados sólo podría contener objetos de la clase Vehiculo. Pues bien, polimorfismo es el mecanismo por el cual podemos "relajar el sistema de tipos", de modo que nos acepte también objetos de las clases hijas o derivadas.

Por tanto, la "relajación" del sistema de tipos no es total, sino que tiene que ver con las clasificaciones de herencia que tengas en tus sistemas de clases. Si defines un array con casillas de una determinada clase, el compilador también te aceptará que metas en esas casillas objetos de una clase hija de la que fue declarada. Si declaras que una función recibe como parámetros objetos de una determinada clase, el compilador también te aceptará que le envíes en la invocación objetos de una clase derivada de aquella que fue declarada.

INTERFACES

Una interface es una colección de declaraciones de métodos (sin definirlos) y también puede incluir constantes.

Runnable es un ejemplo de interface en el cual se declara, pero no se implementa, una función miembro *run*.

```
public interface Runnable {
    public abstract void run();
}
```

Las clases que implementen (**implements**) el interface *Runnable* han de definir obligatoriamente la función *run*.

```
class Animacion implements Runnable{
//..
        public void run(){
                //define la función run
        }
}
```

El papel de la interface es el de describir algunas de las características de una clase. Por ejemplo, el hecho de que una persona sea un futbolista no define su personalidad completa, pero hace que tenga ciertas características que las distinguen de otras.

Clases que no están relacionadas pueden implementar la interface *Runnable*, por ejemplo, una clase que describa una animación, y también puede implementar el interface *Runnable* una clase que realice un cálculo intensivo. [2]

Diferencias entre un interface y una clase abstracta

Un interface es simplemente una lista de métodos no implementados, además puede incluir la declaración de constantes. Una clase abstracta puede incluir métodos implementados y no implementados o abstractos, miembros dato constantes y otros no constantes.

Ahora bien, la diferencia es mucho más profunda. Imaginemos que *Runnable* fuese una clase abstracta. Un applet descrito por la clase *MiApplet* que moviese una figura por su área de trabajo, derivaría a la vez de la clase base *Applet* (que describe la funcionalidad mínima de un applet que se ejecuta en un navegador) y de la clase *Runnable*. Pero el lenguaje Java no tiene herencia múltiple.

En el lenguaje Java la clase *MiApplet* deriva de la clase base *Applet* e implementa el interface *Runnable*

```
class MiApplet extends Applet implements Runnable{
//...
//define la función run del interface
        public void run(){
                //...
```

```
        }
//redefine paint de la clase base Applet
        public void paint(Graphics g){
                //...
        }
//define otras funciones miembro
}
```

Una clase solamente puede derivar **extends** de una clase base, pero puede implementar varios interfaces. Los nombres de los interfaces se colocan separados por una coma después de la palabra reservada **implements**.

El lenguaje Java no fuerza por tanto, una relación jerárquica, simplemente permite que clases no relacionadas puedan tener algunas características de su comportamiento similares. [2]

Interfaces gráficas

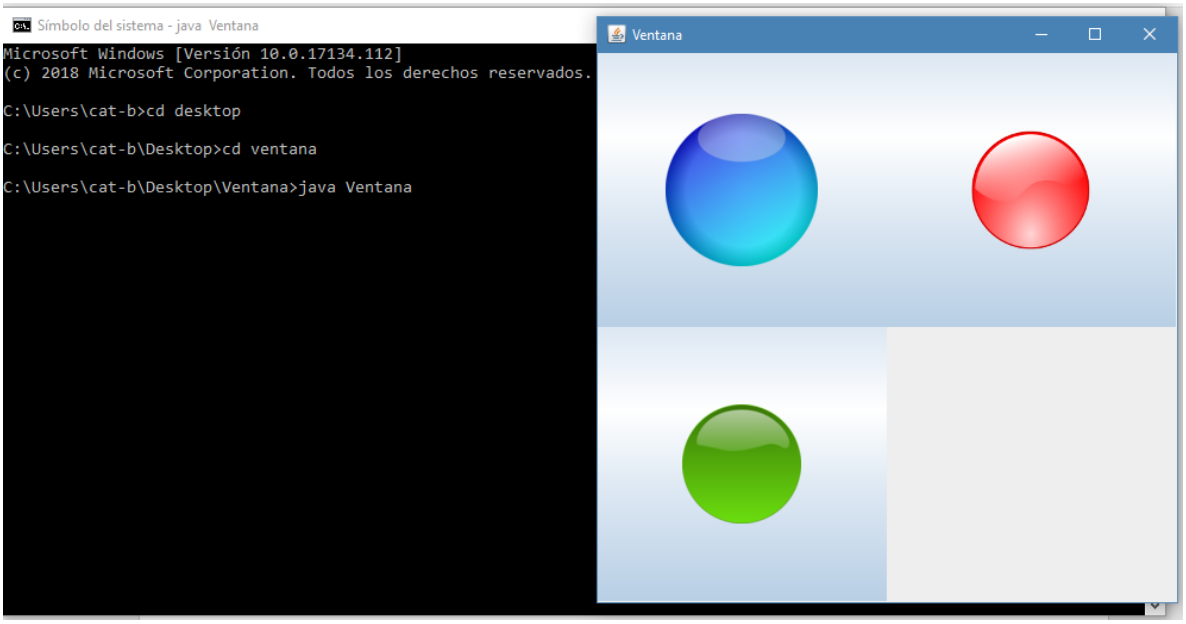
Para empezar la discusión de cómo realizar interfaces graficas en Java, debemos primero realizar algunas aclaraciones:

- El concepto de "Interfaz de usuario" se refiere a los mecanismos para construir ventanas, botones, menús, etc. que permiten crear una interfaz amigable para nuestros programas y no al concepto de "interface" que existe en Java y que se refiere a una especie de clase abstracta pura.
- Los mecanismos para crear "interfaces de usuario" en Java están pensados para favorecer la creación de la lógica de negocio separada de la creación de la interfaz de usuario. Sin embargo, no hay garantía de que esto suceda. Es responsabilidad del programador cristalizar este objetivo. Para poder lograrlo le sugerimos al lector estudiar el patrón de diseño conocido como MVC. Ligas a MVC: Spring Tutorial Patrón
- Para el manejo de eventos Java utiliza un modelo en el cual existe un "ente" generador de eventos y otros "entes" interesados en dar respuesta a los eventos. Cuando un evento se produce, el generador del evento avisa a todos los "entes" que hayan manifestado su interés en el evento. Y cada uno de los interesados responderá al evento utilizando su propia lógica.

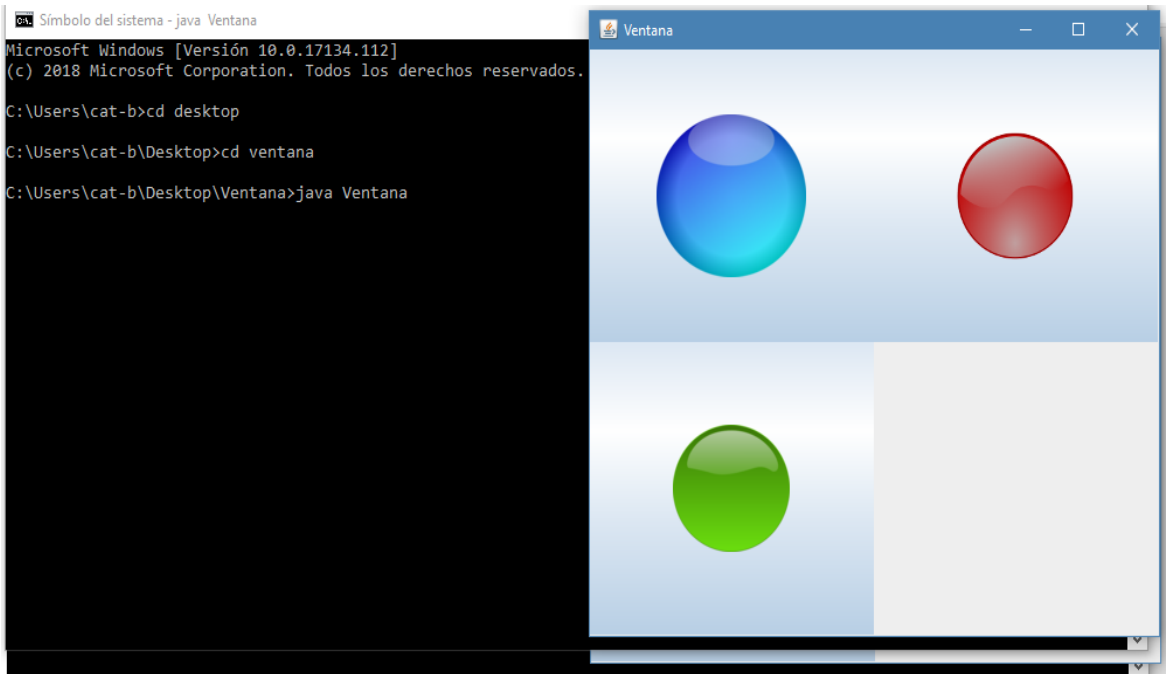
La ventana (JFrame) sólo puede contener dos cosas: un menu, y un panel. Cada uno de estos elementos con su espacio ya determinado. Por el momento no utilzaremos el menú. Los botones y demás elementos que agregar no se colocan directamente en la ventana, sino en un objeto interno, el "Panel". Este objeto interno a su vez utiliza otro objeto interno conocido como LayoutManager, que se encarga de controlar la disposción de los elementos dentro del panel. [3]

Pruebas

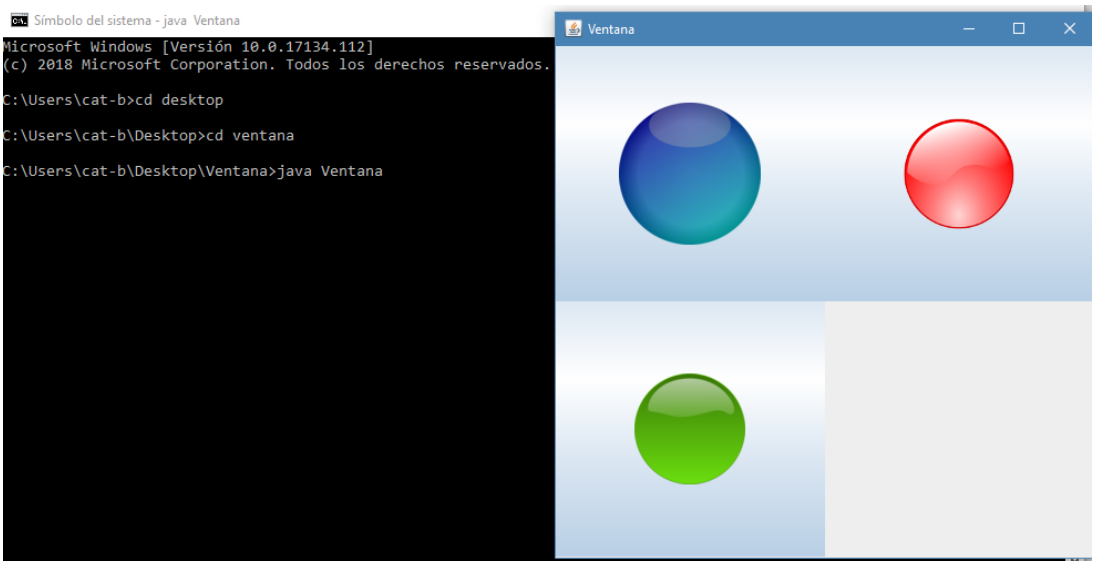
1.- Primero se abre la aplicación por medio de la consola y se muestra lo siguiente:

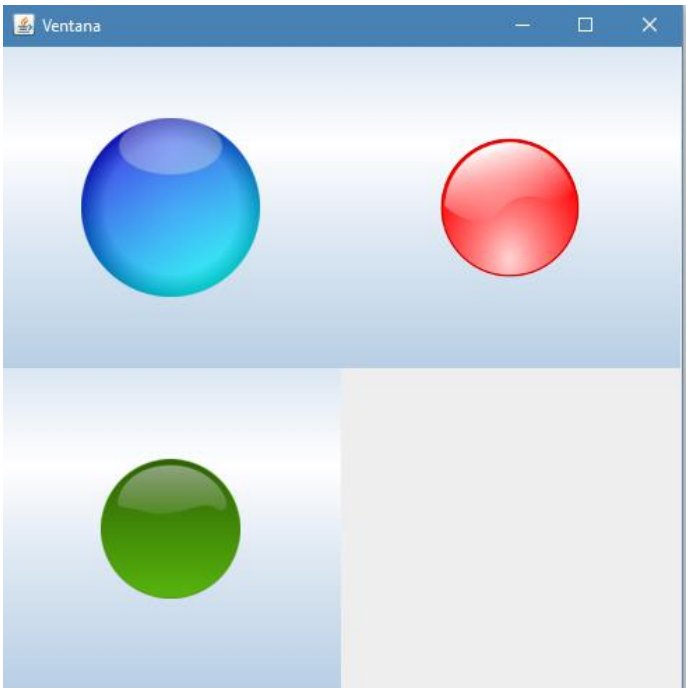


2.- Los botones de la parte de arriba son más brillantes que los que se muestran en la parte de abajo, eso es porque se pasa el cursor por los botones como se muestra en la siguiente imagen

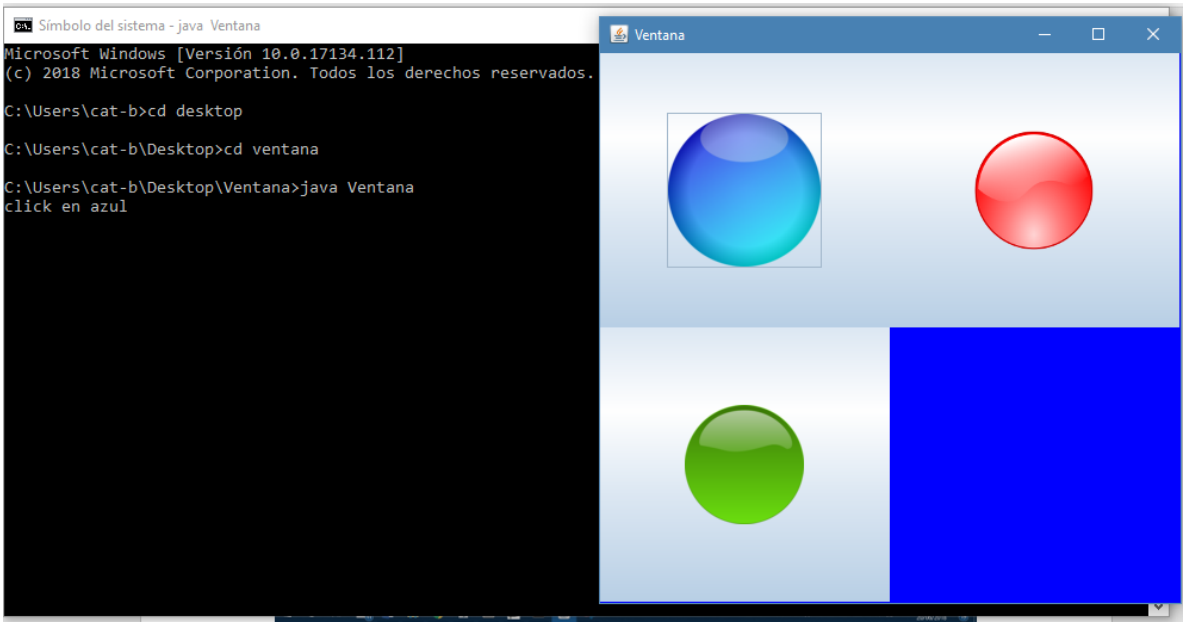


3.-Y así pasa en los botones que están dentro de la ventana.

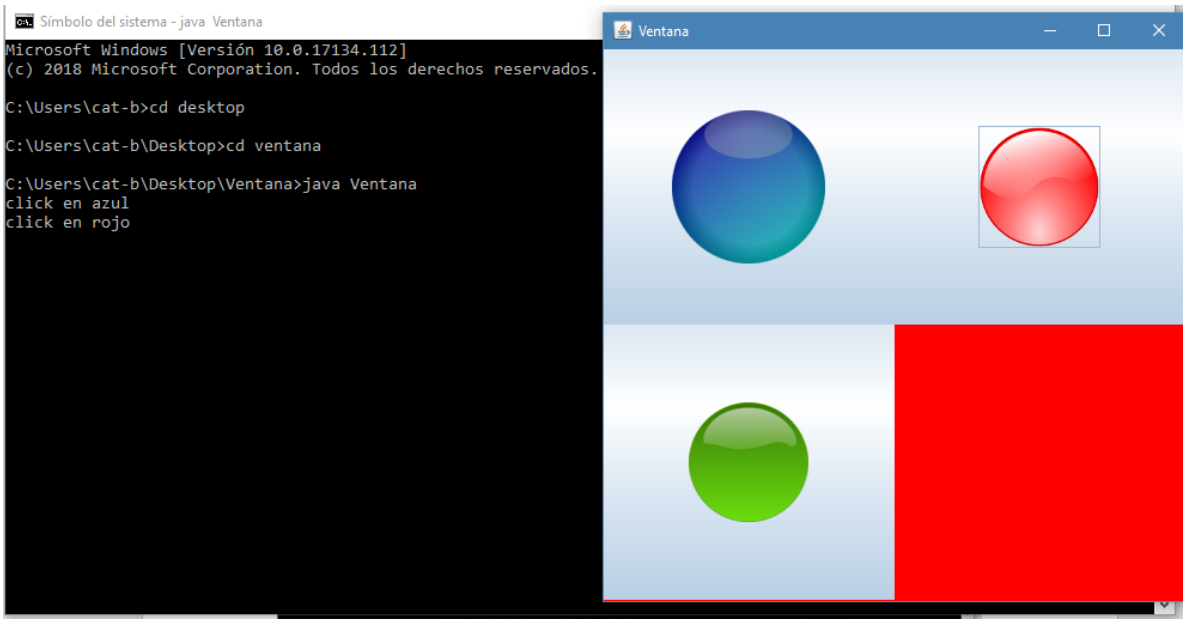


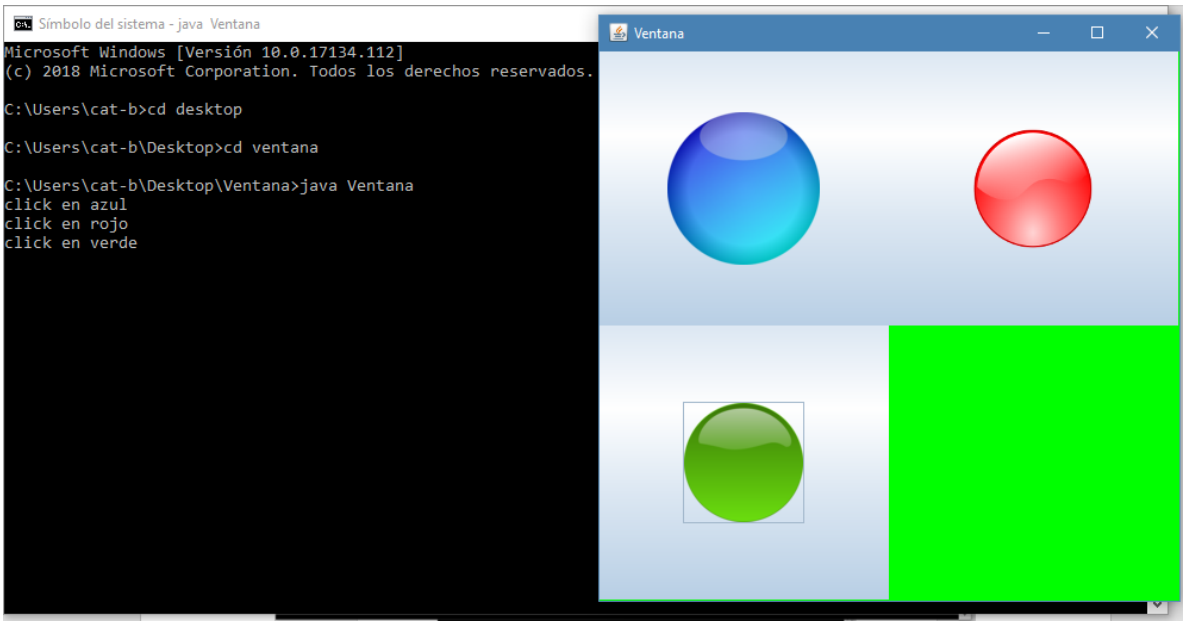


4.-Cuando se da “Click” en alguno de los botones, en la parte inferior derecha de la ventana se muestra el color que se ha seleccionado.

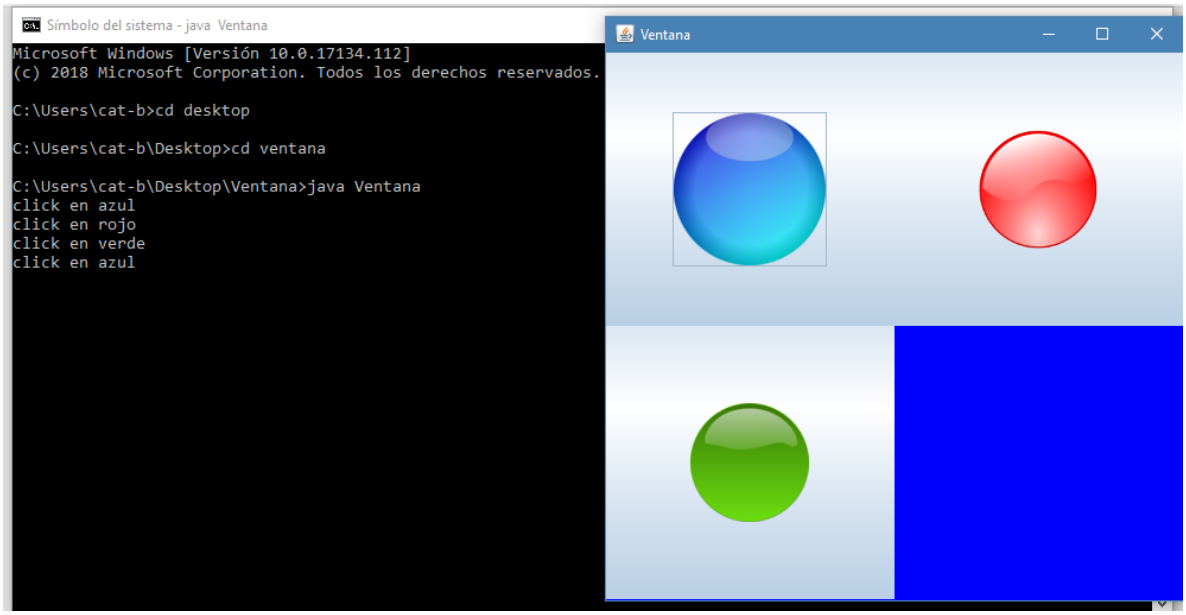


5.- Y así pasa con los otros dos botones, en la parte inferior de la captura se muestra la consola, en la cual se observa qué botones se han seleccionado.




Símbolo del sistema - java Ventana
Microsoft Windows [Versión 10.0.17134.112]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.
C:\Users\cat-b>cd desktop
C:\Users\cat-b\Desktop>cd ventana
C:\Users\cat-b\Desktop\Ventana>java Ventana
click en azul
click en rojo
click en verde
Ventana

6.- Fin


Símbolo del sistema - java Ventana
Microsoft Windows [Versión 10.0.17134.112]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.
C:\Users\cat-b>cd desktop
C:\Users\cat-b\Desktop>cd ventana
C:\Users\cat-b\Desktop\Ventana>java Ventana
click en azul
click en rojo
click en verde
click en azul
Ventana

Conclusiones

En conclusión, no todos los objetos siempre son de un mismo tipo, ya que se pueden manejar diferentes varios objetos pueden venir de una misma clase o de un mismo origen y pueden implementarse de diferente forma. Hace cosas diferentes viniendo de la misma clase interface, es un poco difícil al momento de implementarlo porque uno puede identificar cómo quiere que se hagan las cosas, pero al momento de codificar las ideas más con el polimorfismo es complejo, para que funcionen las cosas de tal forma que hagan las acciones que se soliciten.

Bibliografía

[1] Miguel Alvarez |Polimorfismo en Programación Orientada a Objetos | DesarrolloWeb.com [Online] Available: https://desarrolloweb.com/articulos/polimorfismo-programacion-orientada-objetos-concepto.html

[2] S/A Interfaces | Sc.ehu.es [Online] Avaible: http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/herencia/interfaces.htm

[3] S/A Construcción de interfaces gráficas con java | Peixe.org.mx[Online] Available: http://www.peixe.org.mx/sos/tutVent/