Curso de Java a distancia

Clase 31: Clases anidadas en Java

Hemos utilizado las clases anidadas en conceptos anteriores. Decimos que una clase es anidada si está contenida en otra clase.

Veremos que hay varios tipos de clases anidadas en Java:

Clase anidada interna.

Clase anidada estática.

Clase local.

Clase anónima.

Clase anidada interna.

El anidamiento de una clase tiene por objetivo favorecer el encapsulamiento. Una clase anidada se dice que es interna si se la declara dentro de otra clase pero fuera de cualquier método de la clase contenedora.

Puede declararse con cualquiera de los modificadores: private, protected o public.

Una característica fundamental es que una clase interna tiene acceso a todos los atributos de la clase que la contiene, luego para que exista una clase anidada interna es necesario que exista un objeto de la clase contenedora.

Problema:

Confeccionar una clase llamada Coordenadas que almacene la referencia de puntos en el plano mediante x e y. Declarar una clase interna que represente un punto en el plano.
La clase Coordenada debe almacenar en un ArrayList con elementos de tipo Punto. Además la clase Coordenadas debe poder calcular la cantidad de puntos almacenados en cada cuadrante.

Clase: Coordenadas

```
import java.util.ArrayList;

public class Coordenadas {

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

```
    public Punto(int x, int y) {
        fijarX(x);
        fijarY(y);
    }

    public void fijarX(int x) {
        this.x = x;
    }

    public void fijarY(int y) {
        this.y = y;
    }

    public int retornarCuadrante() {
        if (x > 0 && y > 0)
            return 1;
        else if (x < 0 && y > 0)
            return 2;
        else if (x < 0 && y < 0)
            return 3;
        else if (x > 0 && y < 0)
            return 4;
        else
            return -1;
    }
}

private ArrayList<Punto> puntos;

public Coordenadas() {
    puntos = new ArrayList<Punto>();
}

public void agregarPunto(int x, int y) {
    puntos.add(new Punto(x, y));
}

public int cantidadPuntosCuadrante(int cuadrante) {
    int cant = 0;
    for (Punto pun : puntos)
        if (pun.retornarCuadrante() == cuadrante)
            cant++;
    return cant;
}

}
```

Clase: PruebaCoordenadas

```
public class PruebaCoordenadas {
    public static void main(String[] ar) {
        Coordenadas coordenadas = new Coordenadas();
        coordenadas.agregarPunto(30, 30);
        coordenadas.agregarPunto(2, 7);
        coordenadas.agregarPunto(-3, 2);
        coordenadas.agregarPunto(-5, -4);
        coordenadas.agregarPunto(-9, -2);
        System.out.println("Cantidad de puntos en el primer cuadrante:"
                + coordenadas.cantidadPuntosCuadrante(1));
        System.out.println("Cantidad de puntos en el segundo cuadrante:"
                + coordenadas.cantidadPuntosCuadrante(2));
```

```
        System.out.println("Cantidad de puntos en el tercer cuadrante:"
                + coordenadas.cantidadPuntosCuadrante(3));
        System.out.println("Cantidad de puntos en el cuarto cuadrante:"
                + coordenadas.cantidadPuntosCuadrante(4));
    }
}
```

Se dice que la clase privada Punto es una clase interna de la clase Coordenadas:

```
public class Coordenadas {

    private class Punto {
```

La clase Punto define dos atributos privados:

```
        private int x, y;
```

En el constructor de la clase Punto llamamos a los métodos fijarX y fijarY para que se inicialicen los atributos x e y:

```
        public Punto(int x, int y) {
            fijarX(x);
            fijarY(y);
        }

        public void fijarX(int x) {
            this.x = x;
        }

        public void fijarY(int y) {
            this.y = y;
        }
```

Finalmente el método retornarCuadrante nos informa en que cuadrante se encuentra el punto o un -1 si el punto se encuentra sobre uno de los ejes x o y:

```
        public int retornarCuadrante() {
            if (x > 0 && y > 0)
                return 1;
            else if (x < 0 && y > 0)
                return 2;
            else if (x < 0 && y < 0)
                return 3;
            else if (x > 0 && y < 0)
                return 4;
            else
                return -1;
        }
    }
```

Por otro lado la clase Coordenadas define un atributo de tipo ArrayList con componentes de tipo Punto:

```
    private ArrayList<Punto> puntos;
```

En el constructor creamos el ArrayList:

```
    public Coordenadas() {
        puntos = new ArrayList<Punto>();
    }
```

El método 'agregarPunto' crea un objeto de la clase Punto y lo añade al ArrayList:

```
   public void agregarPunto(int x, int y) {
      puntos.add(new Punto(x, y));
   }
```

Para conocer la cantidad de puntos que hay en un determinado cuadrante debemos recorrer el ArrayList y consultar a cada Punto si coincide con el cuadrante que estamos buscando:

```
   public int cantidadPuntosCuadrante(int cuadrante) {
      int cant = 0;
      for (Punto pun : puntos)
         if (pun.retornarCuadrante() == cuadrante)
            cant++;
      return cant;
   }
```

Para probar la clase 'Coordenadas' creamos la clase 'PruebaCoordenadas' donde creamos un objeto, agregamos un conjunto de puntos y finalmente llamamos al método 'cantidadPuntosCuadrante':

```
public class PruebaCoordenadas {
   public static void main(String[] ar) {
      Coordenadas coordenadas = new Coordenadas();
      coordenadas.agregarPunto(30, 30);
      coordenadas.agregarPunto(2, 7);
      coordenadas.agregarPunto(-3, 2);
      coordenadas.agregarPunto(-5, -4);
      coordenadas.agregarPunto(-9, -2);
      System.out.println("Cantidad de puntos en el primer cuadrante:"
            + coordenadas.cantidadPuntosCuadrante(1));
      System.out.println("Cantidad de puntos en el segundo cuadrante:"
            + coordenadas.cantidadPuntosCuadrante(2));
      System.out.println("Cantidad de puntos en el tercer cuadrante:"
            + coordenadas.cantidadPuntosCuadrante(3));
      System.out.println("Cantidad de puntos en el cuarto cuadrante:"
            + coordenadas.cantidadPuntosCuadrante(4));
   }
}
```

La clase interna puede acceder a los atributos de la clase contenedora, veamos un ejercicio donde se muestra su acceso.

Problema:

Confeccionar una clase llamada JuegoDeDados que contenga una clase anidada interna llamada Dado. La clase JuegoDeDados tiene como atributo el nombre del jugador que tirará el dado y un objeto de la clase Dado.

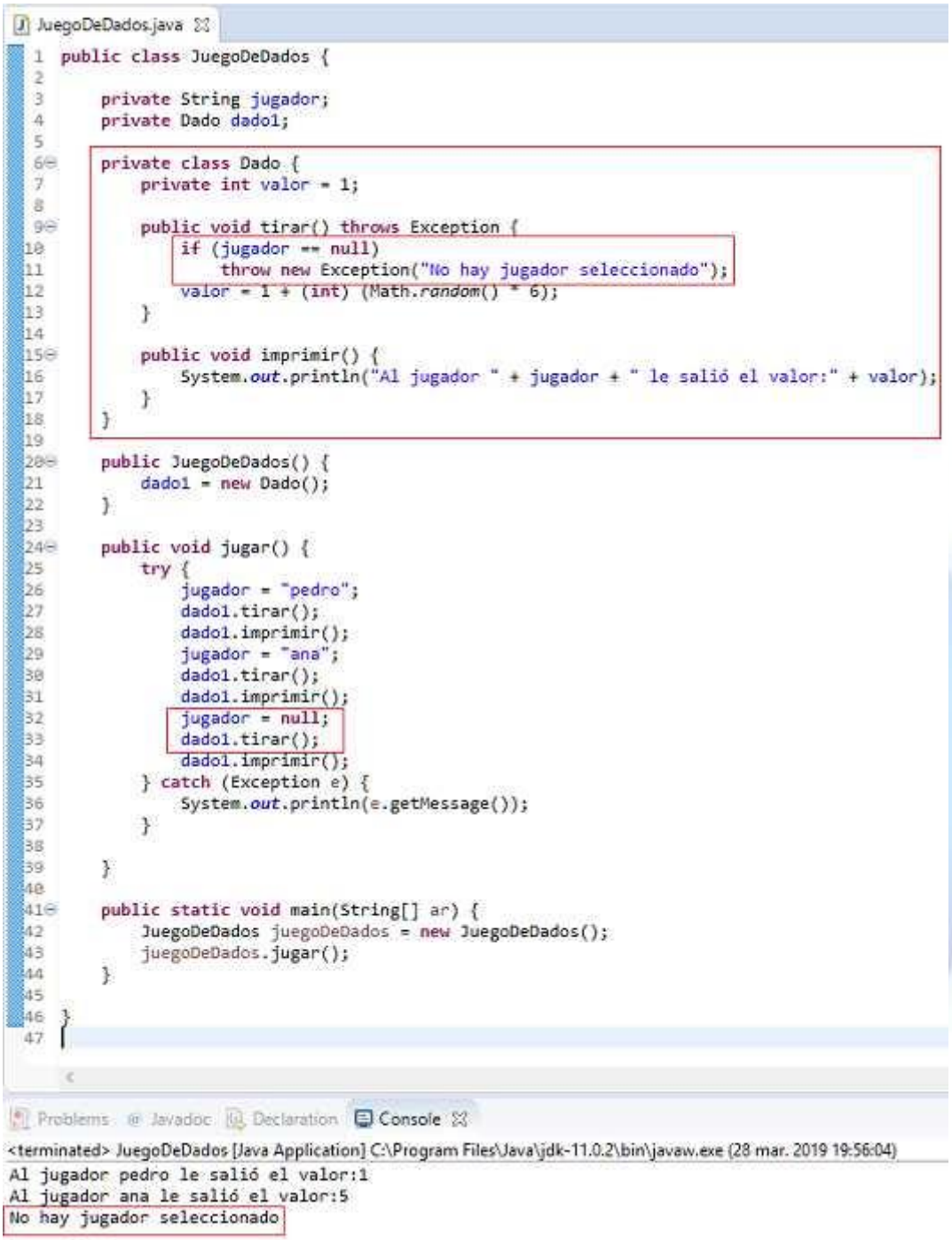
Cada vez que se tire un dado la clase Dado debe verificar que el atributo 'jugador' de la clase externa tenga el nombre de una persona.

Clase: JuegoDeDados

```
public class JuegoDeDados {

   private String jugador;
   private Dado dado1;

   private class Dado {
      private int valor = 1;

      public void tirar() throws Exception {
```

```
        if (jugador == null)
            throw new Exception("No hay jugador seleccionado");
        valor = 1 + (int) (Math.random() * 6);
    }

    public void imprimir() {
        System.out.println("Al jugador " + jugador + " le salió el valor:" + valor);
    }
}

public JuegoDeDados() {
    dado1 = new Dado();
}

public void jugar() {
    try {
        jugador = "pedro";
        dado1.tirar();
        dado1.imprimir();
        jugador = "ana";
        dado1.tirar();
        dado1.imprimir();
        jugador = null;
        dado1.tirar();
        dado1.imprimir();
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }

}

public static void main(String[] ar) {
    JuegoDeDados juegoDeDados = new JuegoDeDados();
    juegoDeDados.jugar();
}

}
```

Si ejecutamos la aplicación podemos observar que se genera la excepción cuando no hay seleccionado un nombre de jugador:



La clase Dado puede consultar el valor del atributo 'jugador' de la clase 'JuegoDeDados':

```
public void tirar() throws Exception {
   if (jugador == null)
      throw new Exception("No hay jugador seleccionado");
```

Es importante tener en cuenta que la clase JuegoDeDados no puede acceder directamente a los atributos de la clase interna, sino a través de un objeto de la misma.

Creación de un objeto de la clase interna desde fuera de la clase externa.

Podemos crear un objeto de la clase interna desde fuera de la clase externa siempre y cuando la clase interna sea visible a nivel de paquete (no se especifica modificador de acceso), public o protected:

Clases: Externa e Interna

```
public class Externa {

   public class Interna {
      public void imprimir() {
         System.out.println("Clase interna");
      }
   }

   public void imprimir() {
      System.out.println("Clase externa");
   }
}
```

Clase: PruebaClaseInterna

```
public class PruebaClaseInterna {

   public static void main(String[] args) {
      Externa externa=new Externa();
      Externa.Interna interna=externa.new Interna();
      interna.imprimir();
      externa.imprimir();
   }

}
```

Para crear un objeto de la clase 'Interna' primero debemos crear un objeto de la clase 'Externa':

```
      Externa externa=new Externa();
```

Ahora podemos definir un objeto de la clase 'Interna' mediante la sintaxis:

```
      Externa.Interna interna=externa.new Interna();
```

Clase anidada estática.

En Java podemos definir clases internas con el modificador 'static'. Luego la clase interna se comporta como una clase normal de Java con la salvedad que se encuentra dentro de otra.

Para crear un objeto de la clase interna tenemos que utilizar la siguiente sintaxis:

Clases: Externa e Interna

```
public class Externa {

   public static class Interna {
      public void imprimir() {
         System.out.println("Clase interna estática");
      }
   }

   public void imprimir() {
      System.out.println("Clase externa");
   }
}
```

Clase: PruebaClaseInterna

```
public class PruebaClaseInterna {
```

```
   public static void main(String[] args) {
      Externa.Interna interna = new Externa.Interna();
      interna.imprimir();
   }

}
```

Cuando declaramos una clase interna 'static' luego podemos crear objetos de la misma en forma independiente a la clase externa:

```
      Externa.Interna interna = new Externa.Interna();
      interna.imprimir();
```

Es importante tener en cuenta que una clase interna estática solo puede acceder a los atributos y métodos estáticos de la clase que la contiene y no a los atributos de instancia como vimos en las clases no estáticas.

Clase local.

El lenguaje Java permite declarar una clase local a un método o inclusive a un bloque dentro de un método.

Clases: Externa y Local

```
public class Externa {
   public void imprimir() {
      System.out.println("Comienzo del método imprimir de la clase Externa.");
      class Local {
         public void imprimir() {
            System.out.println("Método imprimir de la clase Local.");
         }
      }
      Local local1 = new Local();
      local1.imprimir();
      System.out.println("Fin del método imprimir de la clase Externa.");
   }

   public static void main(String[] ar) {
      Externa externa1 = new Externa();
      externa1.imprimir();
   }
}
```

La clase 'Local' se encuentra declarada dentro del método 'imprimir' de la clase 'Externa'. Luego solo en dicho método podemos crear objetos de dicha clase local.

La clase local tiene acceso a los métodos y atributos de la clase externa, variables locales y parámetros del método donde se la declara:

Clases: Externa y Local

```
public class Externa {
   int atributo1 = 10;

   public void imprimir(String parametro) {
      System.out.println("Comienzo del método imprimir de la clase Externa.");
      int variablelocal = 4;
      class Local {
         public void imprimir() {
            System.out.println("Método imprimir de la clase Local.");
            System.out.println(atributo1);
            System.out.println(parametro);
```

```
            System.out.println(variablelocal);
        }
    }
    Local local1 = new Local();
    local1.imprimir();
    System.out.println("Fin del método imprimir de la clase Externa.");
  }

  public static void main(String[] ar) {
    Externa externa1 = new Externa();
    externa1.imprimir("Prueba");
  }
}
```

Clase anónima.

Las clases anónimas en Java son clases anidadas sin un nombre de clase. Normalmente se declaran como una subclase de una clase existente o como la implementación de una interfaz:

Ejemplo

```
public class PruebaClaseAnonima {

  abstract class A {
    public abstract void imprimir();
  }

  interface B {
    void imprimir();
  }

  public void probar() {
    (new A() {
      public void imprimir() {
        System.out.println("Clase");
      }
    }).imprimir();

    (new B() {
      public void imprimir() {
        System.out.println("Interface");
      }
    }).imprimir();
  }

  public static void main(String[] ar) {
    PruebaClaseAnonima p = new PruebaClaseAnonima();
    p.probar();
  }

}
```

En el método probar creamos dos clases anónimas. Primero creamos una clase anónima que hereda de la clase A e implementa su método abstracto:

```
    (new A() {
      public void imprimir() {
        System.out.println("Clase");
      }
    }).imprimir();
```

A partir del objeto que se crea de la clase anónima llamamos al método imprimir.

De forma similar podemos crear una clase anónima implementando una interfaz:

```
(new B() {
   public void imprimir() {
      System.out.println("Interface");
   }
}).imprimir();
```

En conceptos anteriores utilizamos clases anónimas para la captura del click de botones. Veamos un ejemplo donde cada vez que se presiona un botón se incrementa en uno la etiqueta del mismo:

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

import javax.swing.JButton;
import javax.swing.JFrame;

public class FormularioBoton extends JFrame {
   private JButton boton1;

   public FormularioBoton() {
      setLayout(null);
      boton1 = new JButton("0");
      boton1.setBounds(40, 20, 100, 50);
      add(boton1);
      boton1.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            int valor = Integer.parseInt(boton1.getText());
            valor++;
            boton1.setText(String.valueOf(valor));
         }
      });
   }

   public static void main(String[] args) {
      FormularioBoton fb = new FormularioBoton();
      fb.setBounds(0, 0, 200, 120);
      fb.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
      fb.setVisible(true);

   }

}
```

Al método addActionListener le pasamos la referencia de un objeto de una clase anónima que implementa la interfaz ActionListener:

```
boton1.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      int valor = Integer.parseInt(boton1.getText());
      valor++;
      boton1.setText(String.valueOf(valor));
   }
});
```

El código que se requiere con una clase anónima es mucho más compacto que declarar una clase concreta:

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

```
import javax.swing.JButton;
import javax.swing.JFrame;

public class FormularioBoton extends JFrame {
   private JButton boton1;

   public FormularioBoton() {
      setLayout(null);
      boton1 = new JButton("0");
      boton1.setBounds(40, 20, 100, 50);
      add(boton1);
      boton1.addActionListener(new EscuchaPresionBoton());
   }

   class EscuchaPresionBoton implements ActionListener {

      public void actionPerformed(ActionEvent e) {
         int valor = Integer.parseInt(boton1.getText());
         valor++;
         boton1.setText(String.valueOf(valor));
      }

   }

   public static void main(String[] args) {
      FormularioBoton fb = new FormularioBoton();
      fb.setBounds(0, 0, 200, 120);
      fb.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
      fb.setVisible(true);

   }

}
```

Polimorfismo en Java

El polimorfismo es una característica de la programación orientada a objetos que permite llamar a métodos con igual nombre pero que pertenecen a clases distintas.

En Java es necesario que las clases compartan una superclase común para implementar el polimorfismo, luego veremos que también se puede implementar el polimorfismo en Java mediante interfaces.

Con el polimorfismo podemos implementar programas que luego son fácilmente extensibles.

Problema:

Confeccionar una clase abstracta llamada 'Operacion', en la misma definir los atributos: valor1, valor2 y resultado.

Definir un método abstracto 'operar'.

Luego definir 2 subclases llamadas 'Suma' y 'Resta' que hereden de la clase 'Operacion'.

Crear un ArrayList de tipo 'Operacion' y almacenar objetos de tipo 'Suma' y 'Resta'.

Clases: Operacion

```
public abstract class Operacion {
   protected int valor1, valor2, resultado;

   public Operacion(int valor1, int valor2) {
      this.valor1 = valor1;
      this.valor2 = valor2;
   }

   public void imprimir() {
      System.out.println(resultado);
   }

   public abstract void operar();
}
```

La clase 'Operacion' es abstracta y define un método abstracto llamado 'operar'. Luego significa que cualquier clase que herede de 'Operacion' está obligado a implementarlo.

La implementación del método operar depende de la subclase y el concepto que represente la misma.

Clases: Suma

```
public class Suma extends Operacion {

   public Suma(int valor1, int valor2) {
      super(valor1, valor2);
   }

   public void operar() {
      resultado = valor1 + valor2;
   }

}
```

Clases: Resta

```
public class Resta extends Operacion {

   public Resta(int valor1, int valor2) {
      super(valor1, valor2);
   }

   public void operar() {
      resultado = valor1 - valor2;
   }

}
```

Las dos subclases de la clase 'Operacion' cumplen con el contrato de implementar el método 'operar', en caso de no hacerlo se genera un error sintáctico.

Veamos ahora como se genera el comportamiento polimórfico:

Clases:

```
import java.util.ArrayList;

public class Prueba {
```

```
    public static void main(String[] ar) {
        ArrayList<Operacion> lista1 = new ArrayList<Operacion>();
        lista1.add(new Suma(2, 34));
        lista1.add(new Resta(3, 2));
        lista1.add(new Suma(100, 1));
        for (Operacion op : lista1) {
            op.operar();
            op.imprimir();
        }
    }
}
```

Definimos un 'ArrayList' de tipo 'Operacion'. Esto no significa que crearemos objetos de tipo 'Operacion', ya que no se pueden crear objetos de una clase abstracta, sino almacenaremos objetos de clases que hereden de 'Operacion':

```
        ArrayList<Operacion> lista1 = new ArrayList<Operacion>();
        lista1.add(new Suma(2, 34));
        lista1.add(new Resta(3, 2));
        lista1.add(new Suma(100, 1));
```

Creamos objetos de las clases 'Suma' y 'Resta' y los agregamos al 'ArrayList'.

Ahora gracias al polimorfismo podemos llamar al método 'operar' y se ejecutará el que corresponde:

```
        for (Operacion op : lista1) {
            op.operar();
            op.imprimir();
        }
```

Nuestra aplicación el fácilmente extensible, si necesitamos implementar las operaciones 'Multiplicacion', 'Division', elevar a una cierta potencia etc. solo debemos heredar de 'Operacion' y definir la lógica para el método 'operar'.

Creación de Interfaces en Java

El objetivo de una interfaz es declarar una serie de métodos sin su implementación. Luego una o varias clases pueden implementar dicha interfaz.

Una interfaz es similar a una clase abstracta con todos sus métodos abstractos. En Java no existe la herencia múltiple por lo que las interfaces son ampliamente utilizadas.

Mediante una interfaz indicamos que debe hacerse pero no como se debe implementar. Veremos con un ejemplo la creación de una interfaz y la implementación de la misma en dos clases distintas.

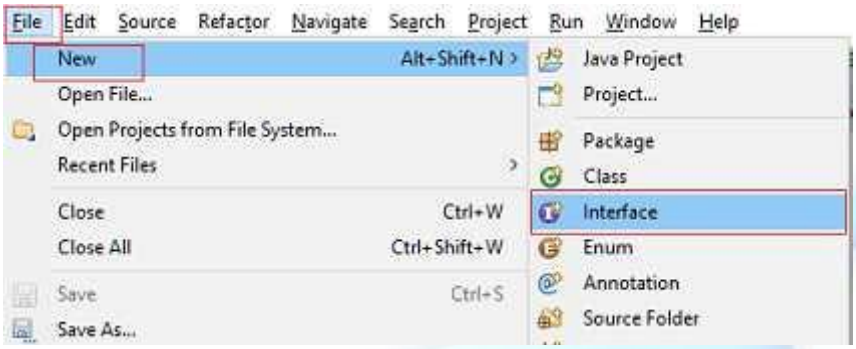
Problema:

Se desea implementar dos juegos distintos.
El primero que permita ingresar el nombre de dos jugadores, tirar dos dados y mostrar cual de los dos ganó.
El segundo juego permita a un jugador adivinar un número entre 1 y 100 que elige la computadora al azar.

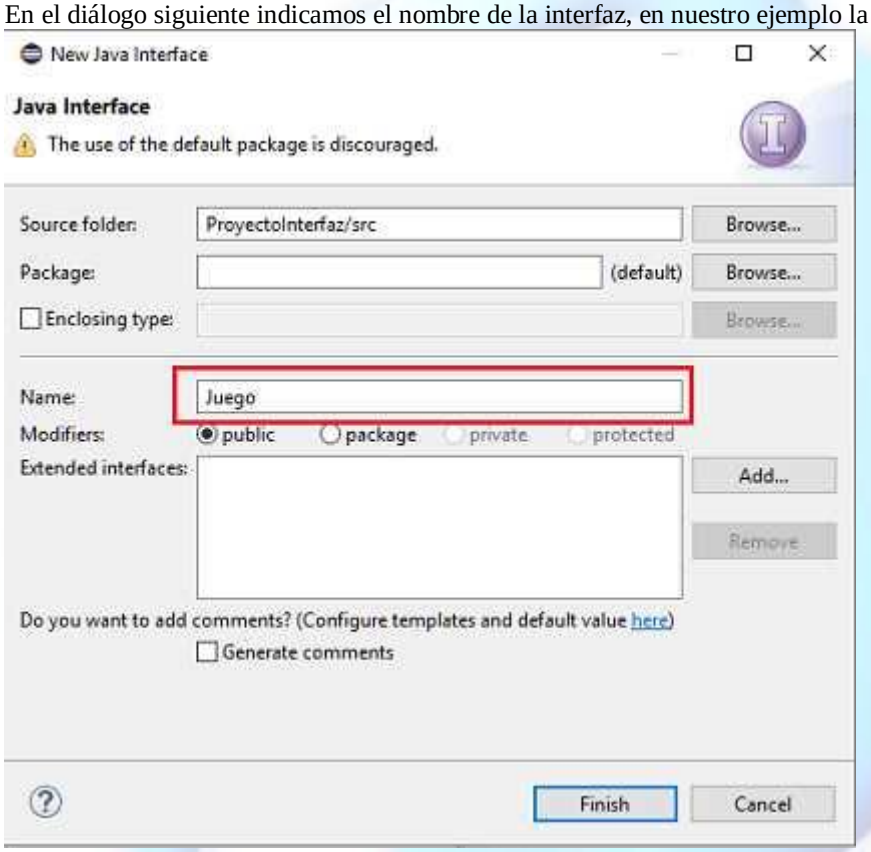
Podemos decir que todo juego tiene un inicio, estado de juego propiamente dicho y un fin. Si queremos estandarizar estos dos juegos y posibles nuevos juegos podemos declarar una interface llamada 'Juego' que declare los métodos: iniciar, jugar y finalizar.

Luego cada vez que creemos un juego debe respetar esta interface implementándola.

Para crear una interfaz en Eclipse elegimos la opción File -> New -> Interface:



En el diálogo siguiente indicamos el nombre de la interfaz, en nuestro ejemplo la llamaremos 'Juego':



Interface: Juego

```
public interface Juego {
   void iniciar();
   void jugar();
   void finalizar();
}
```

Se utiliza la palabra clave 'interface' y seguidamente el nombre de la interface.

Entre llaves declaramos los métodos, pero no su implementación. No se requiere que definamos que son public y abstract, ya que lo son por defecto.

No podemos crear un objeto de la interfaz 'Juego'. Solo se declaran para que otras clases los implementen.

Veamos ahora las clases que implementan la interfaz en nuestro problema:

Clase: JuegoDeDados

```
import java.util.Scanner;

public class JuegoDeDados implements Juego {
   private int dado1, dado2;
   private String jugador1;
   private String jugador2;
   private Scanner teclado;

   public JuegoDeDados() {
      teclado = new Scanner(System.in);
   }

   public void iniciar() {
      System.out.print("Ingrese el nombre del primer jugador:");
      jugador1 = teclado.nextLine();
      System.out.print("Ingrese el nombre del segundo jugador:");
      jugador2 = teclado.nextLine();
   }

   public void jugar() {
      dado1 = 1 + (int) (Math.random() * 6);
      dado2 = 1 + (int) (Math.random() * 6);
      System.out.println(jugador1 + " le salió el valor " + dado1);
      System.out.println(jugador2 + " le salió el valor " + dado2);
   }

   public void finalizar() {
      if (dado1 > dado2)
         System.out.println("Gano " + jugador1 + " con un valor de " + dado1);
      else if (dado2 > dado1)
         System.out.println("Gano " + jugador2 + " con un valor de " + dado2);
      else
         System.out.println("Empataron los dos jugadores con el valor " + dado1);
   }

}
```

Para indicar que la clase 'JuegoDeDados' implementará la interfaz 'Juego' se especifica luego de la palabra clave 'implements':

```
public class JuegoDeDados implements Juego {
```

Una clase puede implementar más de una interface, para ello indicamos los nombre de las interfaces a implementar separadas por coma, por ejemplo:

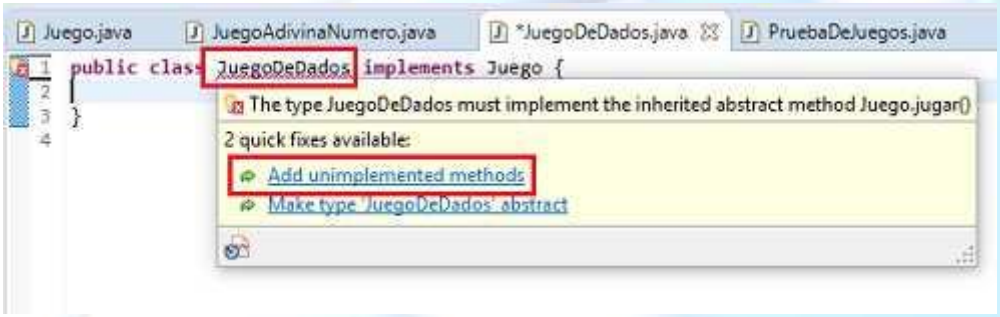
```
public class JuegoDeDados implements Juego, ActionListener, WindowListener {
```

Luego de indicar que la clase 'JuegoDeDados' implementará la interfaz 'Juego' estamos obligados a codificar los algoritmos de los tres métodos contenidos en la interfaz:

```
   public void iniciar() {
      System.out.print("Ingrese el nombre del primer jugador:");
      jugador1 = teclado.nextLine();
      System.out.print("Ingrese el nombre del segundo jugador:");
      jugador2 = teclado.nextLine();
   }
```

```
    public void jugar() {
        dado1 = 1 + (int) (Math.random() * 6);
        dado2 = 1 + (int) (Math.random() * 6);
        System.out.println(jugador1 + " le salió el valor " + dado1);
        System.out.println(jugador2 + " le salió el valor " + dado2);
    }

    public void finalizar() {
        if (dado1 > dado2)
            System.out.println("Gano " + jugador1 + " con un valor de " + dado1);
        else if (dado2 > dado1)
            System.out.println("Gano " + jugador2 + " con un valor de " + dado2);
        else
            System.out.println("Empataron los dos jugadores con el valor " + dado1);
    }
```

Con el entorno de Eclipse podemos fácilmente escribir los tres métodos eligiendo la opción 'Add unimplements methods':



De forma similar creamos la clase 'JuegoAdivinaNumero'

Clase: JuegoAdivinaNumero

```
import java.util.Scanner;

public class JuegoAdivinaNumero implements Juego {
    private int numAdivina;
    private Scanner teclado;
    private int intentos;

    public JuegoAdivinaNumero() {
        teclado = new Scanner(System.in);
    }

    public void iniciar() {
        numAdivina = 1 + (int) (Math.random() * 100);
    }

    public void jugar() {
        int numero;
        do {
            System.out.print("Adivina un número entre 1 y 100:");
            numero = teclado.nextInt();
            if (numAdivina < numero)
                System.out.println("El número a adivinar es menor");
            else if (numAdivina > numero)
                System.out.println("El número a adivinar es mayor");
            intentos++;
        } while (numero != numAdivina);
    }
```

```
    public void finalizar() {
        System.out.println("Ganaste luego de " + intentos + " intentos");
    }

}
```

Si bien son juegos muy distintos, la implementación de la interfaz 'Juego' nos estandariza los métodos mínimos que debe cumplir todo juego.

Par probar el funcionamiento codificaremos una tercer clase donde crearemos un objeto para cada uno de los juegos.

Clase: PruebaDeJuegos

```
public class PruebaDeJuegos {

    public static void main(String[] args) {
        JuegoDeDados juego1 = new JuegoDeDados();
        juego1.iniciar();
        juego1.jugar();
        juego1.finalizar();

        JuegoAdivinaNumero juego2 = new JuegoAdivinaNumero();
        juego2.iniciar();
        juego2.jugar();
        juego2.finalizar();
    }

}
```

Creamos un objeto de la clase 'JuegoDeDados' y llamamos a los métodos: iniciar, jugar y finalizar:

```
        JuegoDeDados juego1 = new JuegoDeDados();
        juego1.iniciar();
        juego1.jugar();
        juego1.finalizar();
```

De forma similar creamos el objeto de la clase 'JuegoAdivinaNumero'.

Herencia en interfaces

Una interfaz puede heredar de otra, luego la clase que la implementa debe codificar todos los métodos.

Interfaz: Interface1

```
public interface Interface1 {
    void metido1();
}
```

Interfaz: Interface2

```
public interface Interface2 extends Interface1 {
    void metodo2();
    void metodo3();
}
```

Clase: Interface2

```
public class PruebaInterfaces implements Interface2 {
```

```
    public void metido1() {

    }

    public void metodo2() {

    }

    public void metodo3() {

    }

}
```

Como podemos comprobar la clase 'PruebaInterfaces' al implementar la interfaz 'Interface2' debe codificar los tres métodos.

Herencia múltiple en interfaces

A diferencia de las clases que pueden heredar solo de una única clase, las interfaces pueden heredar de múltiples interfaces.

Interfaz: A

```
public interface A {
    void metodo1();
}
```

Interfaz: B

```
public interface B {
    void metodo2();
}
```

Interfaz: C

```
public interface C extends A, B {
    void metodo3();
}
```

Clase: PruebaInterfaces

```
public class PruebaInterfaces implements C{

    public void metodo1() {

    }

    public void metodo2() {

    }

    public void metodo3() {

    }

}
```

Como podemos comprobar la interface 'C' hereda de las interfaces 'A' y 'B':

```
public interface C extends A, B {
   void metodo3();
}
```

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