Interfaces.

- Una interfaz es una clase completamente abstracta, sin implementación.
- Las interfaces se declaran con la palabra reservada **interface**
- En la declaración de la interfaz lo único que puede aparecer son declaraciones de métodos y definiciones de constantes.
- Para indicar que una clase impelmenta una interfaz se utiliza la palabra reservada **implements**
- La clase debe implementar **todos** los métodos definidos por la interfaz o declararse a su vez como una clase abstracta

Interfaces. Ejemplo

```java
public interface MatematicaVectorial {

 public static double pi    = 3.1415;
 public static double e     = 2.71828;

 public double[] sumar(int[][] vectores);
 public double[] restar(int[][] vectores);
 public double productoPunto(int[][] vectores);

}


public class VectorR3 implements MatematicaVectorial
{
    public double valorX;
    public double valorY;
    public double valorZ;

    public double[] sumar(int[][] vectores){
        //Implementacion
    }

    public double[] restar(int[][] vectores){
        //Implementacion
    }

    public double productoPunto(int[][] vectores)
    {
        //Implementacion
    }
}
```

Interfaces vs clases abstractas.

- Java no permite la herencia múltiple
- Cuando la herencia jerárquica (tipo árbol) no es suficiente podemos valernos de las interfaces
- Una subclase puede heredar de una sola superclase, pero puede implementar varias interfaces

Polimorfismo

El polimorfismo en Java consiste en dos propiedades:

1. Una referencia a una superclase puede apuntar a un objeto de cualquiera de sus subclases

   Vehículo v1=new Coche(Vehiculo.rojo,12345,2000);

   Vehículo v2=new Barco(Vehiculo.azul,2345);

2. La operación se selecciona en base a la clase del objeto, no a la de la referencia

   v1.toString() usa el método de la clase Coche, puesto que v1 es un coche

   v2.toString() usa el método de la clase Barco, puesto que v2 es un barco

Polimorfismo

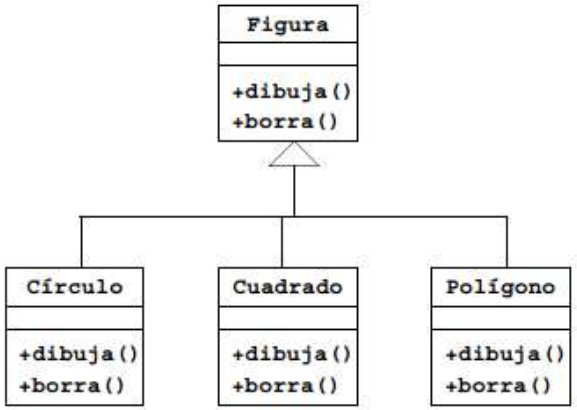
Ejemplo:

Suponer que existe la clase Figura y sus subclases

- Círculo
- Cuadrado
- Polígono

Todas ellas con las operaciones:

- dibuja
- borra



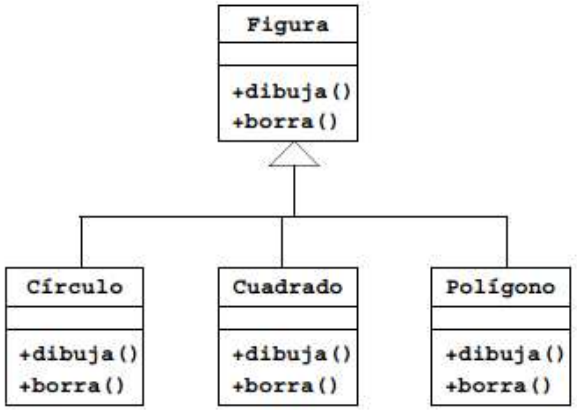
Polimorfismo

Nos gustaría poder hacer la operación polimórfica **mueveFigura** que opere correctamente con cualquier clase de figura:

- mueveFigura
- borra dibuja en la nueva posición

Esta operación polimórfica debería:

- llamar a la operación borra del Círculo cuando la figura sea un círculo
- llamar a la operación borra del Cuadrado cuando la figura sea un cuadrado
- etc.



Polimorfismo

Gracias a esas dos propiedades, el método moverFigura sería:

```
public void mueveFigura(Figura f, Posición pos){
    f.borra();
    f.dibuja(pos);
}
```

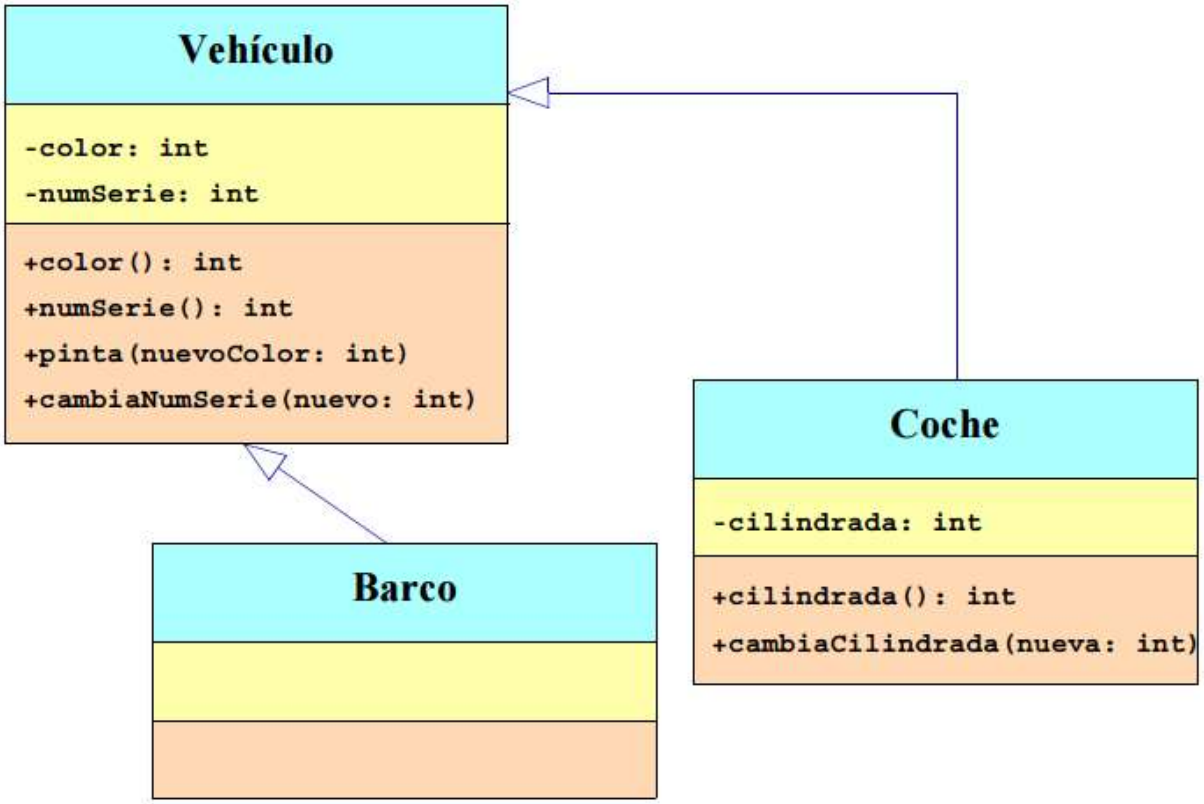
Y podría invocarse de la forma siguiente:

```
Círculo c = new Círculo(...);
Polígono p = new Polígono(...);
mueveFigura(c, pos);
mueveFigura(p, pos);
```

- Gracias a la primera propiedad el parámetro f puede referirse a cualquier subclase de Figura
- Gracias a la segunda propiedad en mueveFigura se llama a las operaciones borra y dibuja apropiadas

Polimorfismo

Supongamos el siguiente diagrama de clases:



Polimorfismo

El lenguaje permite que una referencia a una superclase pueda apuntar a un objeto de cualquiera de sus subclases pero no al revés

Vehículo v = new Coche(...); // permitido

Coche c = new Vehículo(...); // ¡NO permitido!

Justificación:

- un coche es un vehículo - cualquier operación de la clase Vehículo existe (sobrescrita o no) en la clase Coche

  v.cualquierOperación(...); // siempre correcto

- un vehículo no es un coche - sería un error tratar de invocar la operación:

  c.cilindrada(); // ERROR: cilindrada() no // existe para un vehículo, por esa razón el lenguaje lo prohíbe

Polimorfismo. Casting

Es posible convertir referencias

```
Vehículo v=new Coche(...);
Coche c=(Coche)v;
v.cilindrada(); // ¡ERROR!
c.cilindrada(); // correcto
```

El casting cambia el “punto de vista” con el que vemos al objeto

- a través de v le vemos como un Vehículo (y por tanto sólo podemos invocar métodos de esa clase)
- a través de c le vemos como un Coche (y podemos invocar cualquiera de los métodos de esa clase)

Polimorfismo. Casting

Hacer una conversión de tipos incorrecta produce una excepción ClassCastException en tiempo de ejecución

```
Vehículo v=new Vehículo(...);
Coche c=(Coche)v; // lanza ClassCastException en tiempo de ejecución
```

Java proporciona el operador **instanceof** que permite conocer la clase de un objeto

```
if (v instanceof Coche) {
    Coche c=(Coche)v;
    ...
}
```

“v instanceof Coche” retorna true si v apunta a un objeto de la clase Coche o de cualquiera de sus (posibles) subclases

Clases y tipos genéricos o parametrizados.

En Java, cuando definimos una nueva clase, **debemos conocer el tipo de dato** con el que trabajaremos.

Si queremos realizar una operación específica dentro de esta nueva clase, **sea cual sea el tipo de datos** que va a recibir, podemos hacer uso de los **tipos genéricos**.

Este tipo genérico asumirá el tipo de dato que realmente le pasaremos a la clase.

Clases y tipos genéricos o parametrizados.

```java
class ClaseGenerica<T> {
  T obj;

  public ClaseGenerica(T o) {
    obj = o;
  }

  public void classType() {
    System.out.println("El tipo de T es " + obj.getClass().getName());
  }
}

public class MainClass {
  public static void main(String args[]) {
    // Creamos una instancia de ClaseGenerica para Integer.
    ClaseGenerica<Integer> intObj = new ClaseGenerica<Integer>(88);
    intObj.classType();

    // Creamos una instancia de ClaseGenerica para String.
    ClaseGenerica<String> strObj = new ClaseGenerica<String>("Test");
    strObj.classType();

  }
}
```

- T es el tipo genérico que será reemplazado por un tipo real.
- T es el nombre que damos al parámetro genérico.
- Este nombre se sustituirá por el tipo real que se le pasará a la clase.