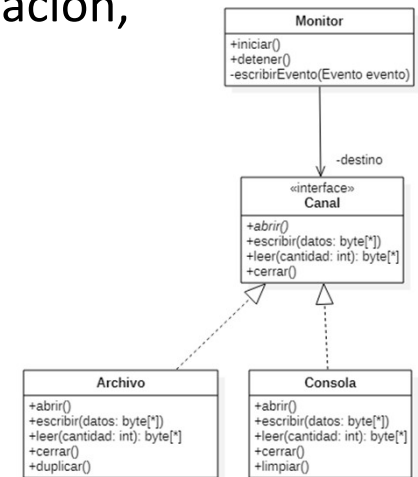


Relaciones interesantes

Objetos que conocen a otros, relaciones uno a muchos, delegación, polimorfismo y el rol de las interfaces



Relaciones entre objetos

- Un objeto conoce a otro porque
 - Es su responsabilidad mantener a ese otro objeto en el sistema (tiene un, conoce a)
 - P.ej., cada cuenta conoce su titular (alguien tiene que acordarse de eso)
 - Necesita delegarle trabajo (enviarle mensajes)
 - P.ej., una cuenta recibe a otra como parámetro (destino) para pedirle que deposite
- Un objeto conoce a otro cuando
 - Tiene una referencia en una variable de instancia (rel. duradera)
 - Le llega una referencia como parámetro (re. temporal)
 - Lo crea (rel. temporal/duradera)
 - Lo obtiene enviando mensajes a otros que conoce (rel. temporal)

Objetos y el chequeo de tipos

- Java es un lenguaje, estáticamente, fuertemente tipado
 - Debemos indicar el tipo de todas las variables (relaciones entre objetos)
 - El compilador chequea la correctitud de nuestro programa respecto a tipos
- Se asegura de que no enviamos mensajes a objetos que no los entienden
- Cuando declaramos el tipo de una variable, el compilador controla que solo “enviemos a esa variable” mensajes acordes a ese tipo
- Cuando asignamos un objeto a una variable, el compilador controla que el tipo del objeto sea compatible con el de la variable

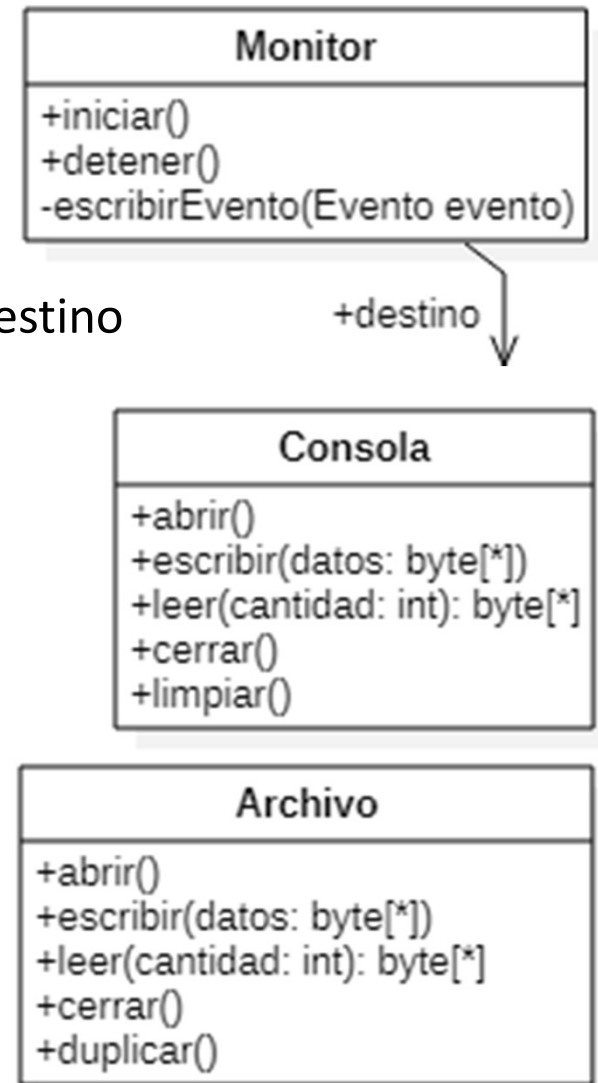
Tipos en lenguajes OO

- Tipo = Conjunto de firmas de operaciones/métodos (nombre, orden y tipos de los argumentos)
- Decimos que un objeto (instancia de una clase) “es de un tipo” si ofrece el conjunto de operaciones definido por el tipo
- Con eso en mente:
 - Cada clase en Java define “explícitamente” un tipo (es un conjunto de firmas de operaciones)
 - Cada instancia de una clase A “es del tipo” definido por esa clase
 - Pero ... las clases no son la única forma de definir tipos

El ejemplo conductor

- Un monitor de eventos que escribe en algún lugar (destino)
- Objetos de varias clases que el monitor podría usar como destino
- ¿Cómo “tipamos” la relación entre el monitor y el destino?

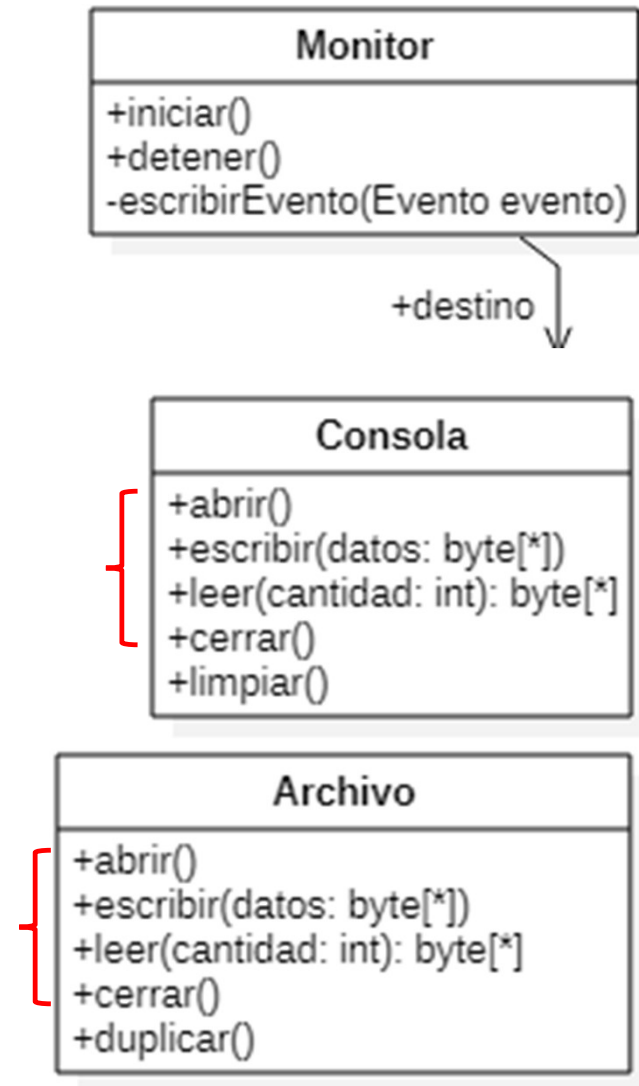
```
public class Monitor {  
    private ??? destino;  
  
    public Monitor( ??? destino) {  
        this.destino = destino;  
    }  
  
    private void escribirEvento(Evento evento) {  
        destino.escribir(evento.asBytes());  
    }  
}
```



El ejemplo conductor

¿Nos interesa decir que de clase es “destino” o que mensajes entiende?

```
public class Monitor {  
    private ??? destino;  
  
    public Monitor( ??? destino) {  
        this.destino = destino;  
    }  
  
    private void escribirEvento(Evento evento) {  
        destino.escribir(evento.asBytes());  
    }  
}
```



```

public class Monitor {
    private Canal destino;

    public Monitor(Canal destino) {
        this.destino = destino;
    }

    private void escribirEvento(Evento evento) {
        destino.escribir(evento.asBytes());
    }
}

```

```

public interface Canal {

    public void abrir();

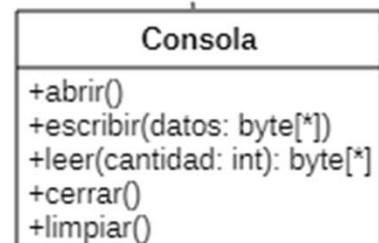
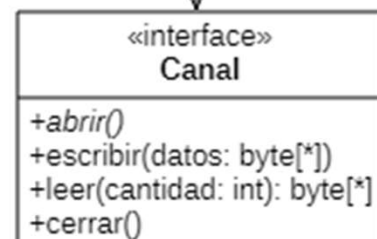
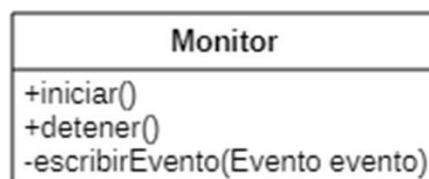
    public void escribir(byte[] datos);

    public byte[] leer(int cantidad);

    public void cerrar();

}

```



```

public class Archivo implements Canal {
    ...
}

```

```

public class Consola implements Canal {
    ...
}

```

-destino

Interfaces

- Una **clase** define un tipo, y también implementa los métodos correspondientes
- Una variable tipada con una **clase** solo “acepta” instancias de esa clase (*)
- Una **interfaz** nos permite declarar tipos sin tener que ofrecer implementación (desacopla tipo e implementación)
- Puedo utilizar Interfaces como tipos de variables
- Las clases deben declarar explícitamente que interfaces implementan
- Una clase puede implementar varias interfaces
- El compilador chequea que la clase implemente las interfaces que declara (salvo que sea una clase abstracta)

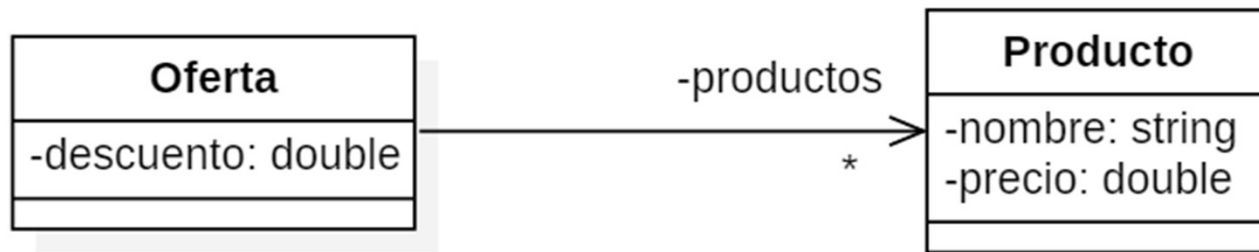
(*) ya hablaremos de herencia ...

Un objeto que conoce a muchos ...

- Las relaciones de un objeto a muchos se implementan con colecciones
- Decimos que un objeto conoce a muchos, pero en realidad conoce a una colección, que tiene referencias a esos muchos
- Para modificar y explorar la relación, envío mensajes a la colección

Un objeto que conoce a muchos ...

- Lo dibujamos así:



Un objeto que conoce a muchos ...

- Lo programamos así:

```
//Declarar una variable que apunta a un Lista de Productos  
private List<Producto> productos;
```

```
// Inicializar una variable que apunta a un Lista de Productos  
productos = new ArrayList<Producto>();
```

```
// Agregar un elemento a un Lista de Productos  
productos.add(producto);
```

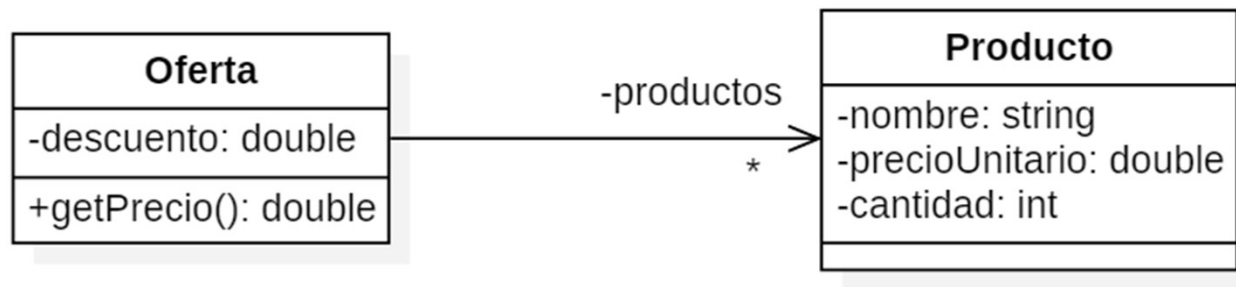
Un objeto que conoce a muchos ...

- Lo programamos así:

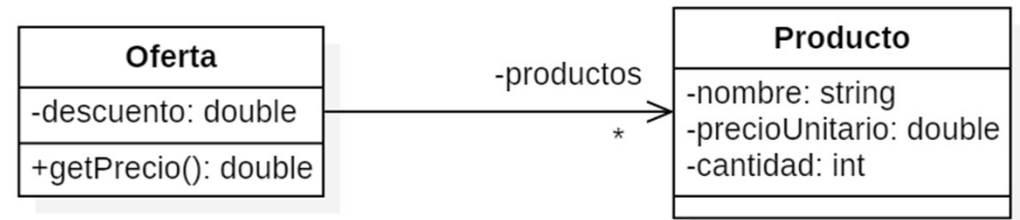
```
//Recorremos una lista de productos  
for (Producto producto : productos) {  
    // hacemos algo con cada producto  
}
```

¿Envidia o delegación?

¿Cómo implementarían `getPrecio()` en la clase oferta?



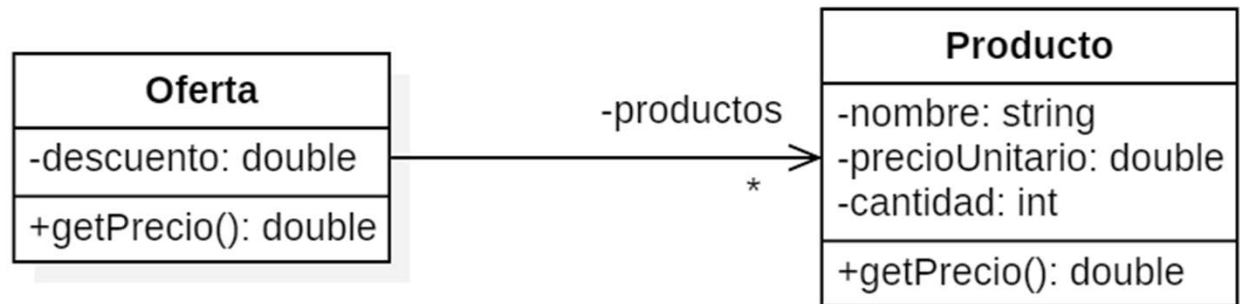
Envidia ...



```
public double getPrecio() {
    double precio = 0;
    for (Producto producto : productos) {
        precio = precio + (producto.getPrecioUnitario() * producto.getCantidad());
    }
    return precio * descuento;
}
```

- Una clase Oferta que envidiosa y egoísta que quiere hacer todo
- Responsabilidades poco repartidas (Producto es solo datos)
- Clases más acopladas y poco cohesivas

Delegación ...



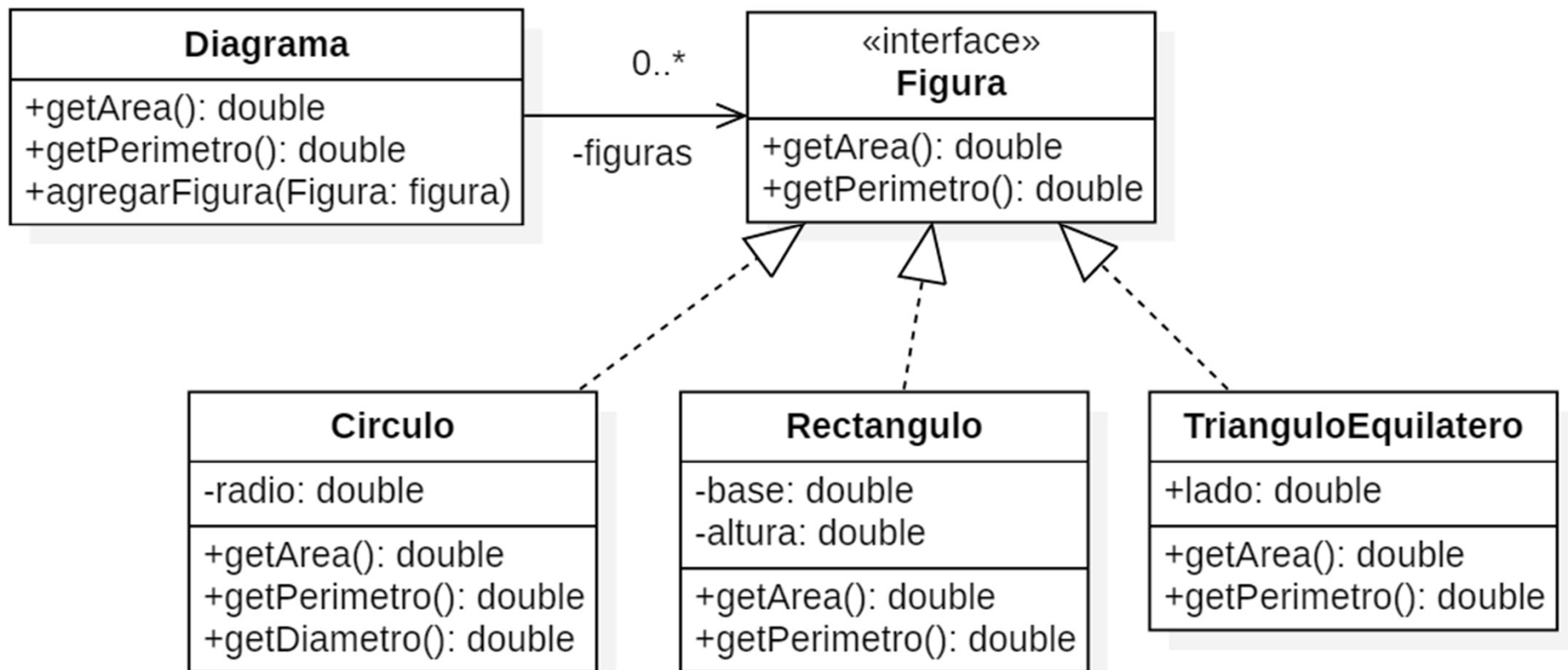
```
public double getPrecio() {
    double precio = 0;
    for (Producto producto : productos) {
        precio = precio + producto.getPrecio();
    }
    return precio * descuento;
}
```

- El cálculo del precio de un producto está con los datos requeridos
- Oferta “delega” y se despreocupa de cómo se hace el cálculo
- Clases más desacopladas y más cohesivas

Polimorfismo

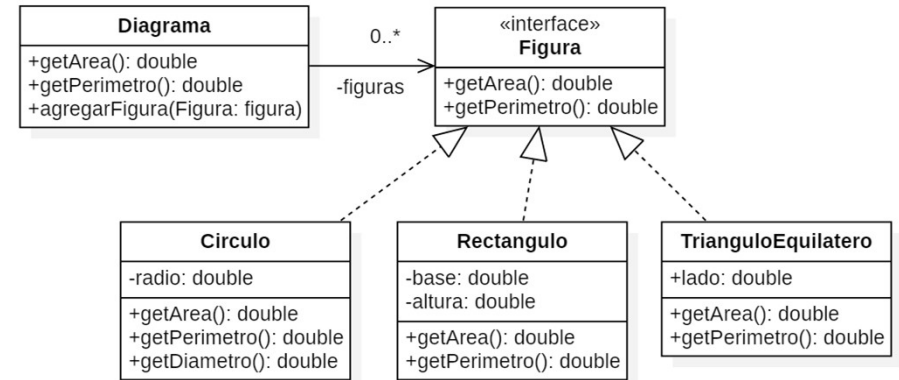
- Objetos de distintas clases son ***polimórficos*** con respecto a un mensaje, si todos lo entienden, aun cuando cada uno lo implemente de un modo diferente
- Polimorfismo implica:
 - Un mismo mensaje se puede enviar a objetos de distinta clase
 - Objetos de distinta clase “podrían” ejecutar métodos diferentes en respuesta a un mismo mensaje
- Cuando dos clases Java implementan una interfaz, se vuelven polimórficas respecto a los métodos de la interfaz

Figuras polimórficas ...



Figuras polimórficas ...

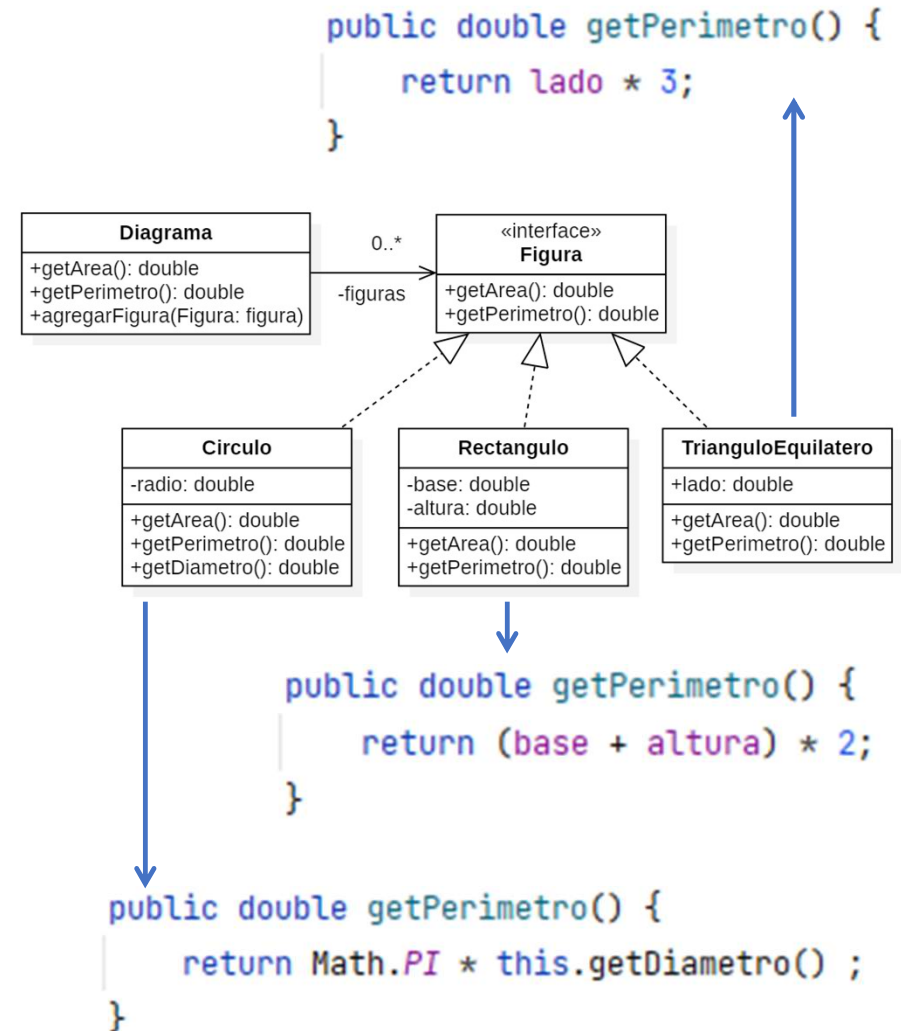
```
public class Diagrama {  
    private List<Figura> figuras;  
  
    public Diagrama() {  
        figuras = new ArrayList<Figura>();  
    }  
  
    public double getPerimetro() {  
        double total = 0;  
        for (Figura figura : figuras) {  
            ??????????  
        }  
        return total;  
    }  
}
```



- ¿Qué hago con cada figura?
- Pueden ser de distintas clases
- ¿Necesito saber de qué clase es?

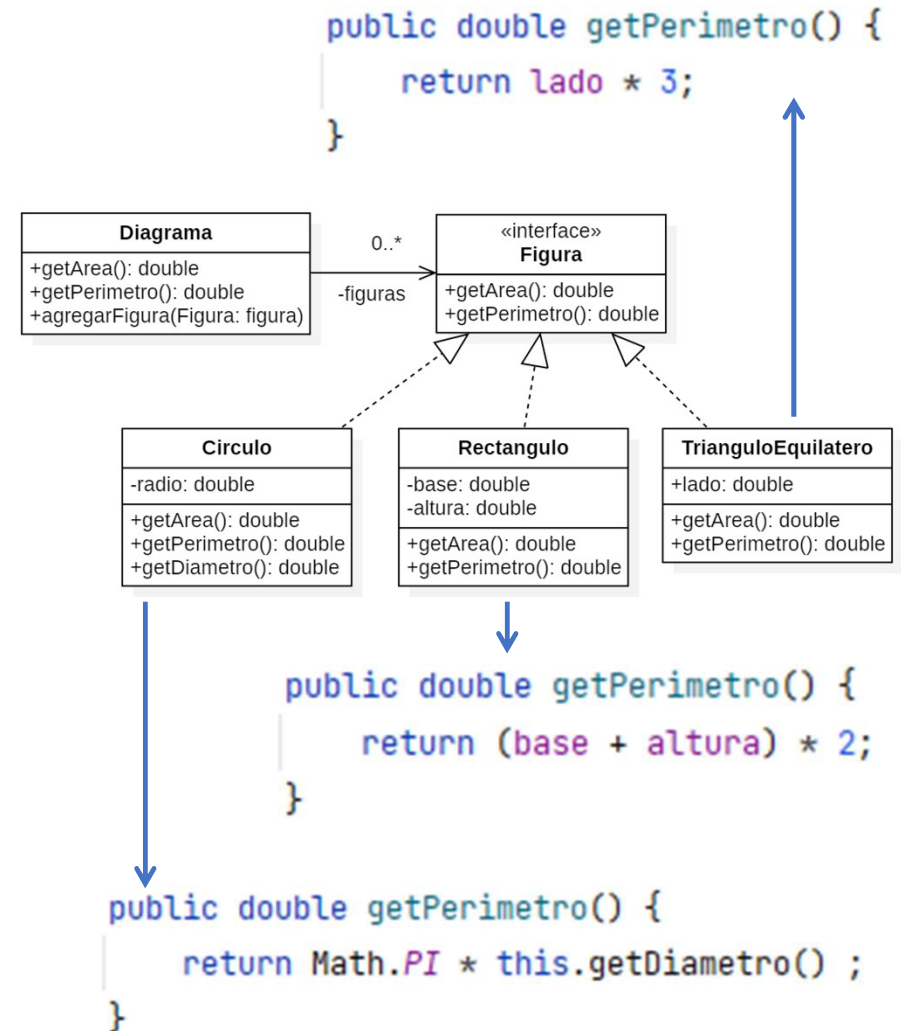
Figuras polimórficas ...

```
public class Diagrama {  
    private List<Figura> figuras;  
  
    public Diagrama() {  
        figuras = new ArrayList<Figura>();  
    }  
  
    public double getPerimetro() {  
        double total = 0;  
        for (Figura figura : figuras) {  
            ??????????  
        }  
        return total;  
    }  
}
```



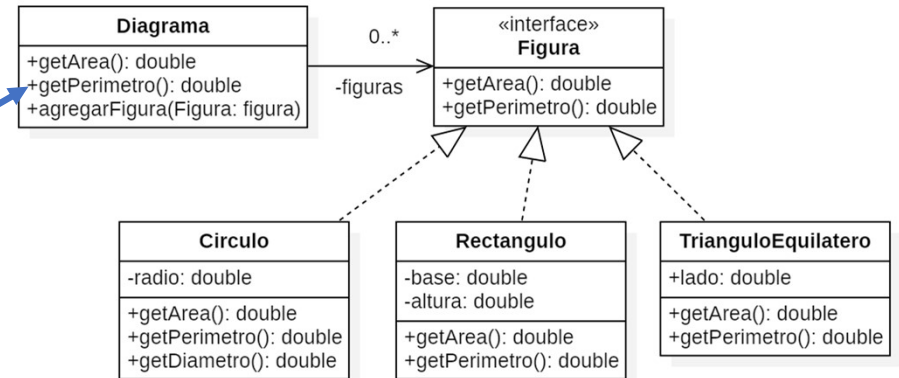
Figuras polimórficas ...

```
public class Diagrama {  
    private List<Figura> figuras;  
  
    public Diagrama() {  
        figuras = new ArrayList<Figura>();  
    }  
  
    public double getPerimetro() {  
        double total = 0;  
        for (Figura figura : figuras) {  
            total = total + figura.getPerimetro();  
        }  
        return total;  
    }  
}
```



Comparemos con ...

```
public double getPerimetro() {  
    double total = 0;  
    for (Figura figura : figuras) {  
        if (figura es un Rectangulo)  
            total = total + (figura.getBase() * figura.getAltura() * 2);  
        if (figura es un TrianguloEquilatero)  
            total = total + (figura.getLado() * 3);  
        if (figura es un Circulo)  
            total = total + (Math.PI * figura.getDiametro());  
    }  
    return total;  
}
```



Polimorfismo bien aplicado

- Permite repartir mejor las responsabilidades (delegar)
- Desacopla objetos y mejora la cohesión (cada cual hace lo suyo)
- Concentra cambios (reduce el impacto de los cambios)
- Permite extender sin modificar (agregando nuevos objetos)
- Lleva a código más genérico y objetos reusables
- Nos permite programar por protocolo, no por implementación