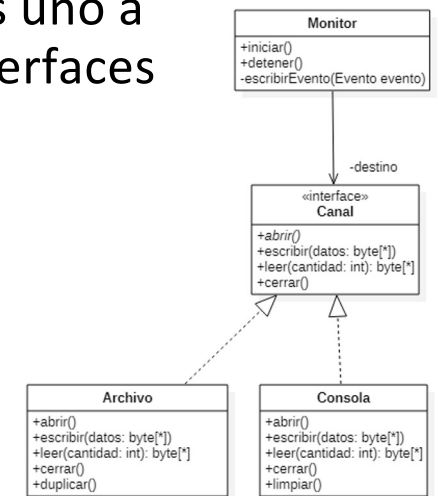


Relaciones objetosas

Objetos que conocen a otros, identidad e igualdad, relaciones uno a muchos, delegación, polimorfismo y el rol de los tipos y 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)

This (un objeto que habla solo)



- **this** (o en algunos lenguajes self) es una “pseudo-variable”
 - no puedo asignarle valor
 - Toma valor automáticamente cuando un objeto comienza a ejecutar un método
- **this** hace referencia al objeto que ejecuta el método (al receptor del mensaje que resultó en la ejecución del método)
- Se utiliza para:
 - Descomponer métodos largos (refinamiento top-down)
 - Reutilizar comportamiento repetido en varios métodos
 - Aprovechar comportamiento heredado (próximamente ...)
 - Pasar una referencia para que otros puedan enviarnos mensajes
- En algunos lenguajes (p.e. Java):
 - Puede obviarse (es implícito), aunque en OO1 preferimos no hacerlo
 - Para desambiguar referencia a las variables de instancia del objeto

Reutilizar comportamiento repetido


```
// Moverse  
public void translateBy(Point2D point) {  
    this.position.translateBy(point);  
    this.disableShields();  
    this.energy -= 1;  
}
```

```
//Disparar  
public void fireUpon(Ship ship) {  
    ship.takeFireFrom(this.position);  
    this.disableShields();  
    this.energy -= 1;  
}
```

Reutilizar comportamiento repetido

```
// Moverse
public void translateBy(Point2D point) {
    this.position.translateBy(point);
    this.disableShields();
    this.energy -= 1;
}

//Disparar
public void fireUpon(Ship ship) {
    ship.takeFireFrom(this.position);
    this.disableShields();
    this.energy -= 1;
}
```



```
private void payThePrice() {
    this.disableShields();
    this.energy -= 1;
}
```

The diagram illustrates code reuse. Two red arrows point from the `this.disableShields();` and `this.energy -= 1;` lines in both `translateBy` and `fireUpon` methods to the corresponding lines in the `payThePrice` method. Red brackets are placed next to the lines being reused in each method.

Reutilizar comportamiento repetido

```
// Move
public void translateBy(Point2D point) {
    this.position.translateBy(point);
    this.payThePrice();
}
```

```
//Disparar
public void fireUpon(Ship ship) {
    ship.takeFireFrom(this.position);
    this.payThePrice();
}
```

```
private void payThePrice() {
    this.disableShields();
    this.energy -= 1;
}
```

Identidad / el operador ==



- Las variables son punteros a objetos
- Mas de una variable pueden apuntar a un mismo objeto
- Para saber si dos variables apuntan al mismo objeto utilizo “==”
- == es un operador, no puede redefinirse

```
// quiero saber si dos autos pertenecen a la misma persona  
if (unAuto.getPropietario() == otroAuto.getPropietario()) {  
    // los autos pertenecen a la misma persona  
}
```

- El ejemplo asume que solo hay un objeto que representa a cada persona (suele ser el caso)

Igualdad / el método equals()

- Dos objetos pueden ser iguales - la igualdad se define en función del dominio

```
// quiero saber si dos autos son iguales  
if (unAuto.equals(otroAuto)) {  
    // los autos son iguales  
}
```

- Implementamos equals en la clase Automovil así (o casi)
 - El ejemplo asume que solo hay un objeto que representa a cada persona (suele ser el caso)

```
public boolean equals(Automovil otroAuto) {  
    return marca.equals(otroAuto.getMarca()) &&  
        modelo.equals(otroAuto.getModelo());  
}
```



Igualdad e identidad (ejemplo con Colores)

// implementación "aproximada" de equals en la clase Color

```
public boolean equals(Color otro) {  
    return this.getRed() == otro.getRed() &&  
           this.getGreen() == otro.getGreen() &&  
           this.getBlue() == otro.getBlue();  
}
```

```
Color blanco = new Color(255, 255, 255);  
Color otroBlanco = new Color(255, 255, 255);  
Color unColor = blanco;
```

```
System.out.println(blanco == blanco); // true  
System.out.println(blanco == unColor); // true  
System.out.println(blanco == otroBlanco); // false  
System.out.println(blanco.equals(unColor)); // true  
System.out.println(blanco.equals(otroBlanco)); // true  
System.out.println(blanco.equals(new Color(0,0,0))); // false
```

Relaciones entre objetos y 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 su clase sea “compatible” con el tipo de la variable

Tipos en lenguajes OO (simplificado)

- Tipo \Leftrightarrow Conjunto de firmas de operaciones/métodos (nombre, orden y tipos de los argumentos)
- Algunos lenguajes diferencian entre tipos primitivos y tipos de referencias (objetos)
 - Nos enfocaremos principalmente en los segundos
- Cada clase en Java define “explícitamente” un tipo (es un conjunto de firmas de operaciones)
 - Puedo utilizar clases, para dar tipo a las variables
- Asignar un objeto a una variable, no afecta al objeto (no cambia su clase)
 - La clase de un objeto se establece cuando se crea, y no cambia más
- 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?

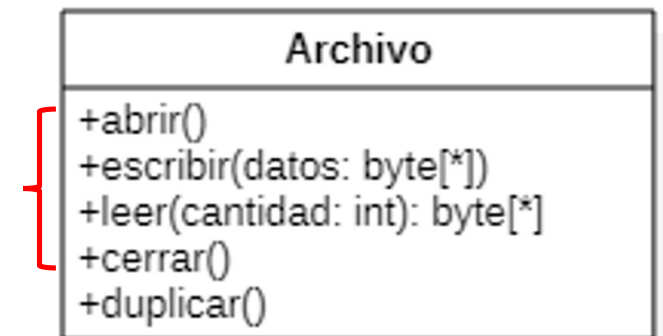
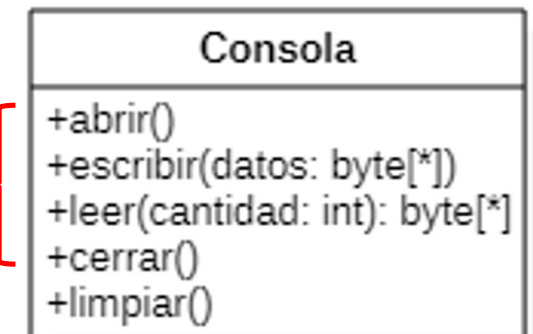
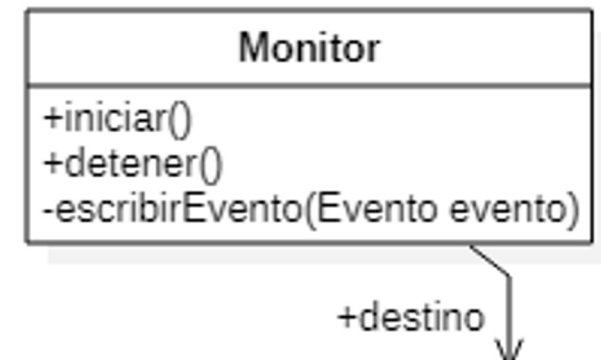
```
public class Monitor {  
    private ??? destino;  
  
    public Monitor( ??? destino) {  
        this.destino = destino;  
    }  
  
    private void escribirEvento(Evento evento) {  
        destino.escribir(evento.asBytes());  
    }  
}
```



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();

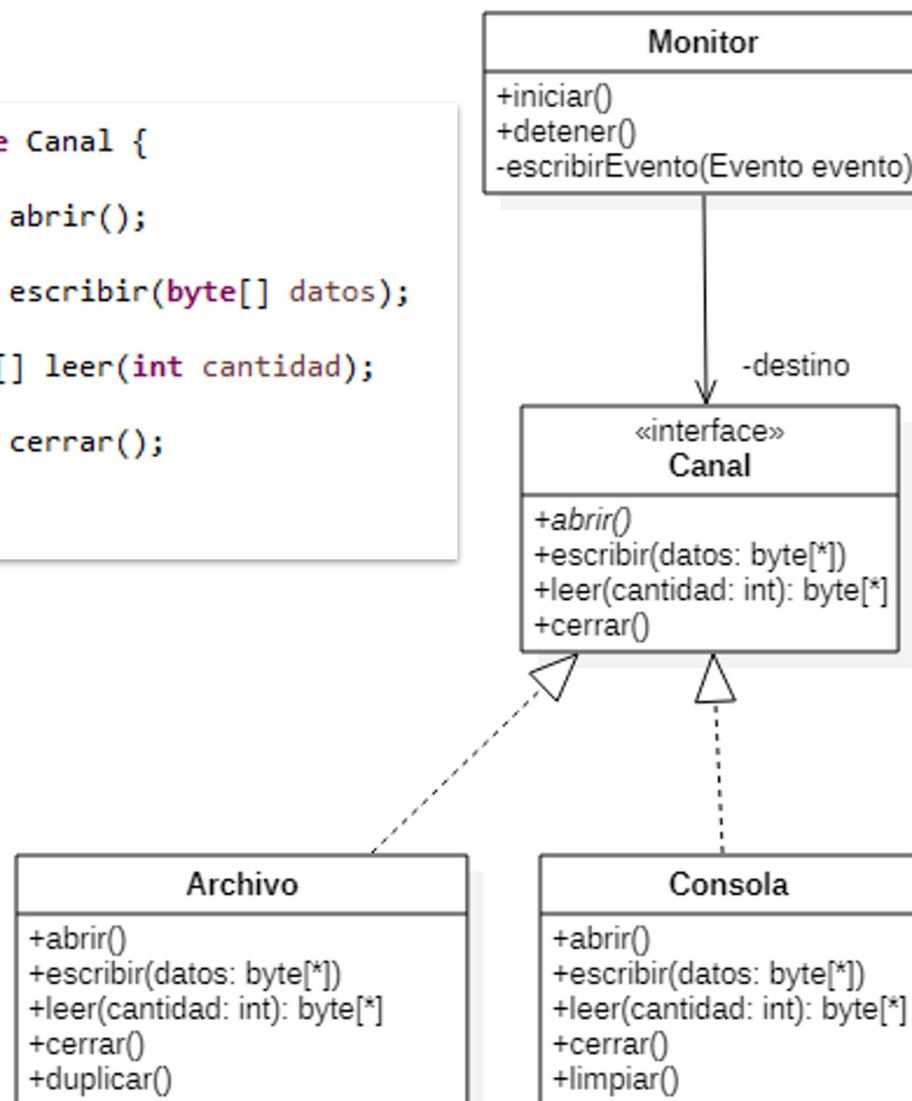
}
```

```
public class Archivo implements Canal {
```

```
    public void abrir() {
```

```
public class Consola implements Canal {
```

```
    public void abrir() {
```



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)

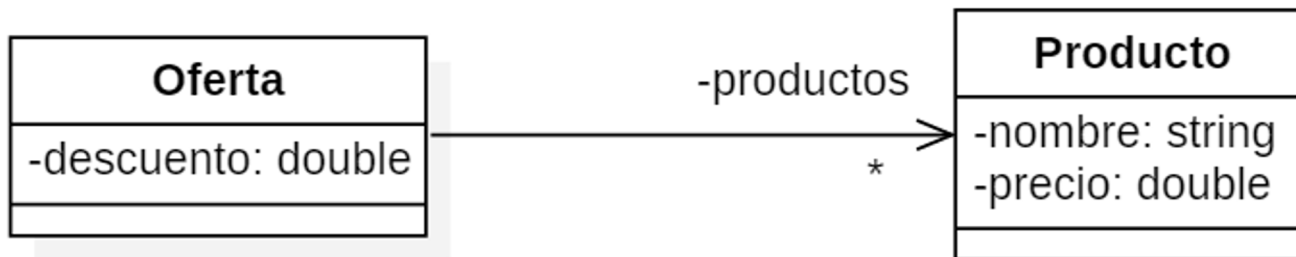
(*) ya hablaremos de herencia ...

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);
```

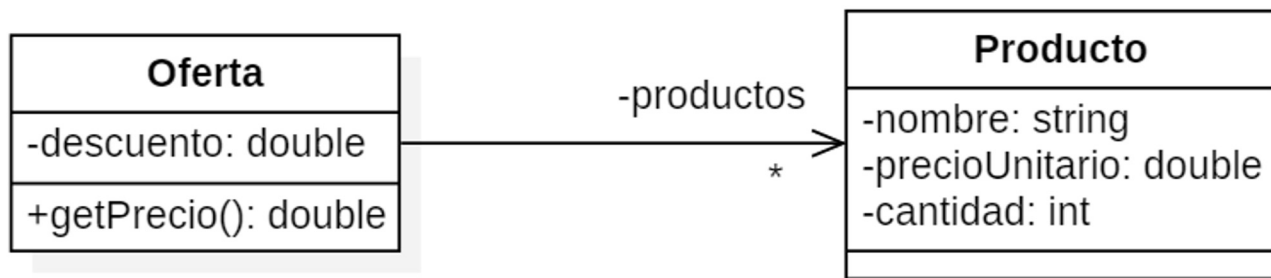
Un objeto que conoce a muchos ...

- Lo programamos así:

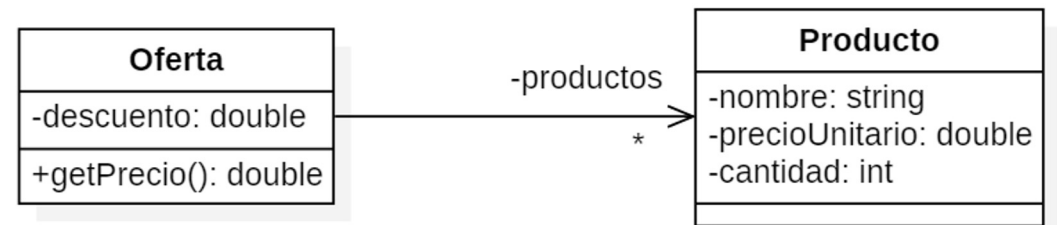
```
//Recorremos una lista de productos  
for (Producto producto : productos) {  
    // hacemos algo con cada producto  
}
```

¿Envidia o delegación?

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



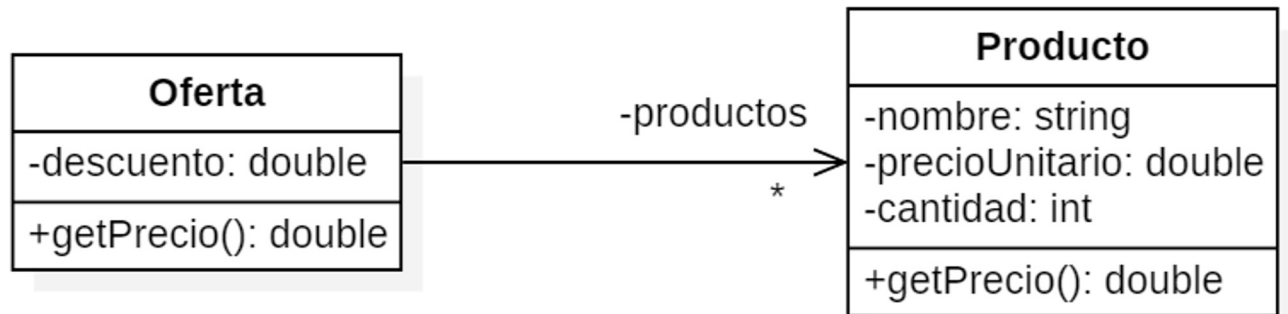
Envidia ...



```
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

Method Lookup (recordamos)

- Cuando un objeto recibe un mensaje, se busca en su clase un método cuya firma se corresponda con el mensaje.
 - En un lenguaje dinámico, podría no encontrarlo (error en tiempo de ejecución)
 - En un lenguaje con tipado estático sabemos que lo entenderá (aunque no sabemos lo que hará)

+ Luz
+encender() +apagar()

+ Calefaccion
+encender() +apagar() +temperatura(t)

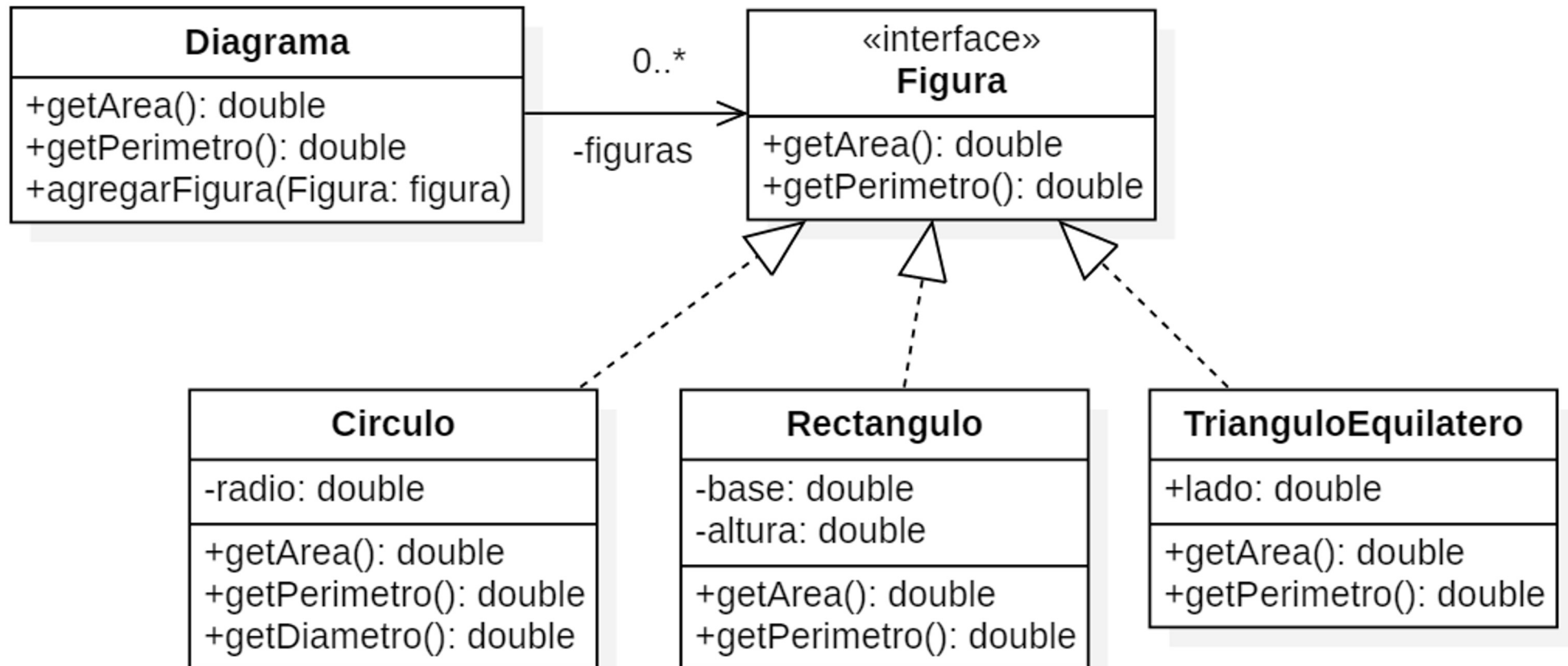
+ Puerta
+abrir() +cerrar()

```
(new Luz()).encender();  
(new Calefaccion()).encender();  
(new Puerta()).encender();
```

Polimorfismo

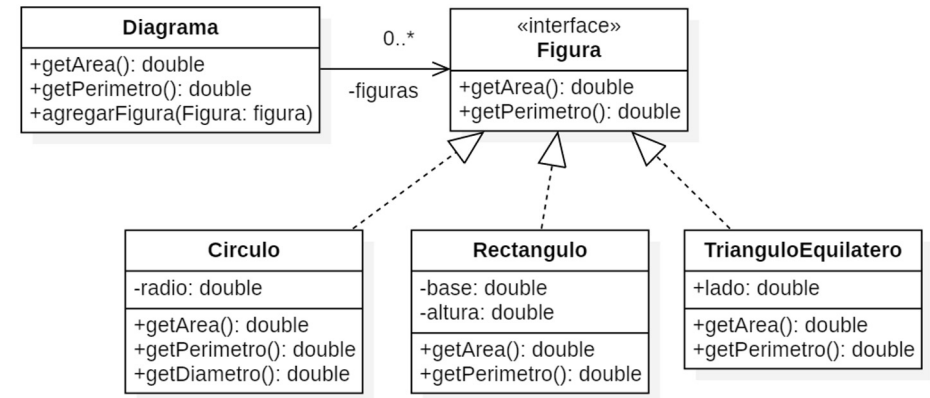
- Objetos de distintas clases son ***polimórficos*** con respecto a un mensaje, si todos lo entienden, aun cuando cada uno lo implemente de un modo diferente
- Polimorfismo implica:
 - Un mismo mensaje se puede enviar a objetos de distinta clase
 - Objetos de distinta clase “podrían” ejecutar métodos 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 ...

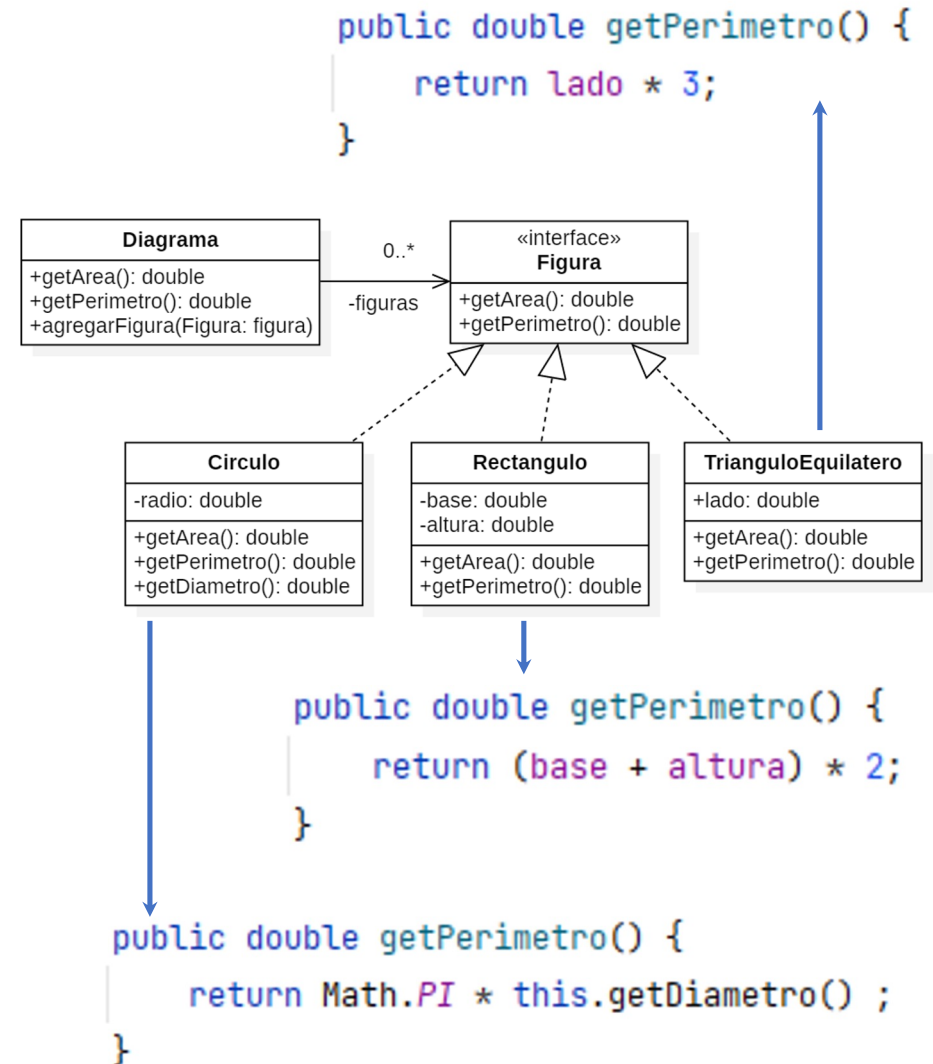
```
public class Diagrama {  
    private List<Figura> figuras;  
  
    public Diagrama() {  
        figuras = new ArrayList<Figura>();  
    }  
  
    public double getPerimetro() {  
        double total = 0;  
        for (Figura figura : figuras) {  
            ??????????  
        }  
     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



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

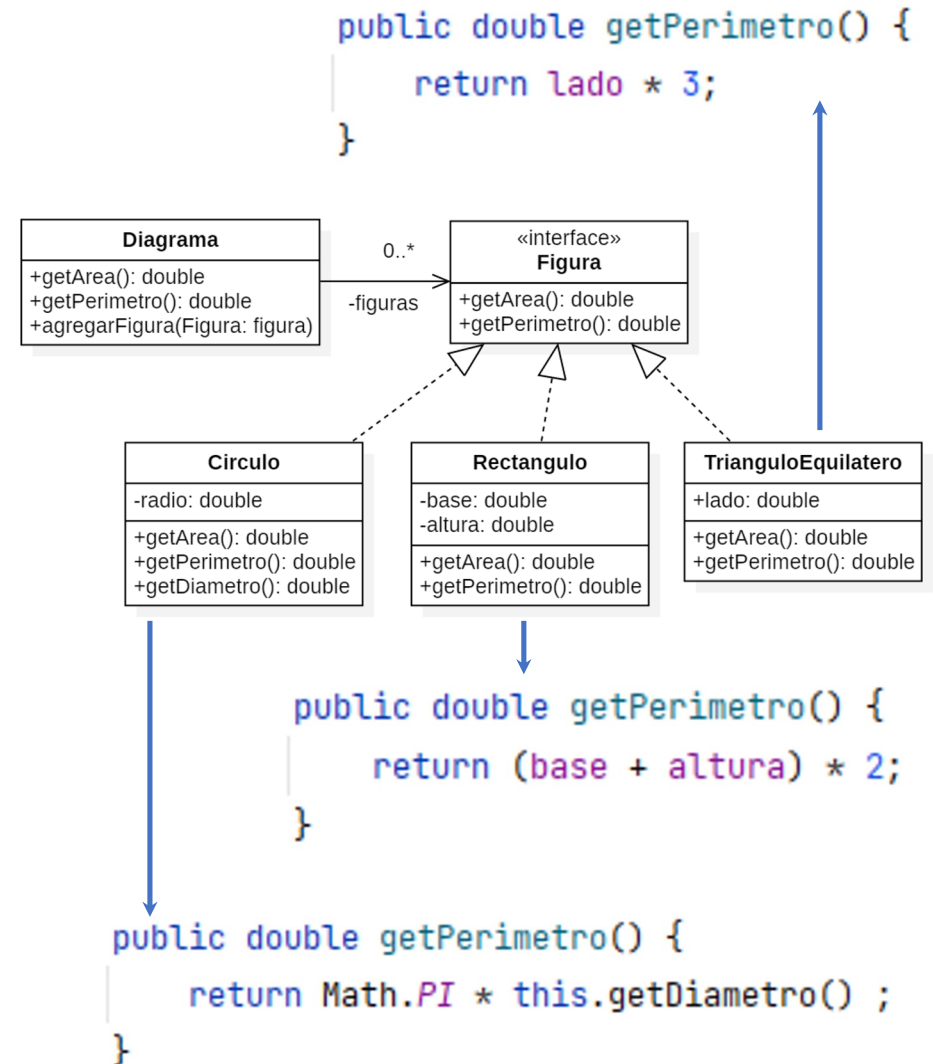
Figuras polimórficas ...

```
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;  
    }  
}
```



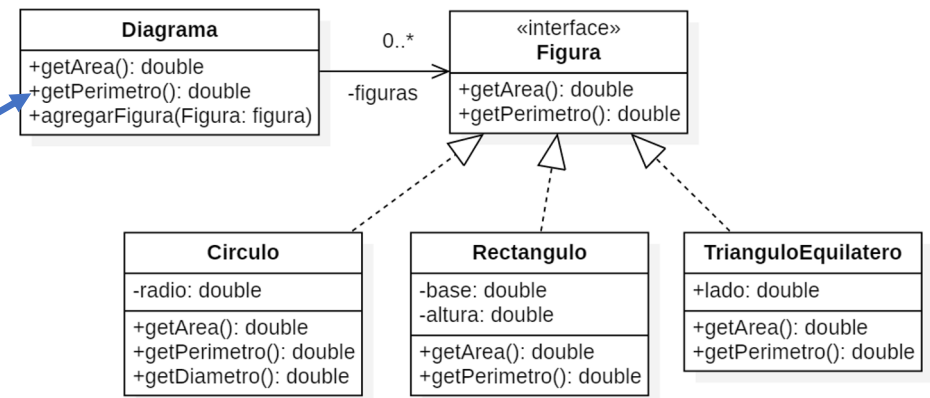
Figuras polimórficas ...

```
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