

## UT4 – PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA EN JAVA

### 1. ATRIBUTOS Y MÉTODOS Static en Clases Java.

Un atributo **static** no tiene un valor específico para cada objeto. Le asignamos un **valor único**, y este valor es compartido por todos los objetos de la clase. Funciona como si fuera una variable global.

**Ejemplo:** En el **ejercicio de Cuentas Bancarias** visto en clase, usar un atributo **static** para el **tipo de interés**

```
private static double tipoInteres=2.0; // Todo objeto de tipo Cuenta tendría un interés asociado del 2.0 %
```

Para acceder al valor del atributo desde fuera de la clase usaremos el habitual método **getTipoInteres()**, con la salvedad de que al ser un atributo **static** el método también debe ser **static** y hemos de usarlo como tal.

```
public static double getTipoInteres() {  
    return tipoInteres;  
}
```

**Cuenta.getTipoInteres()** *// la llamada a un método static siempre se hace a través del nombre de la clase*

Un **método static** tiene acceso sólo a los atributos static de la clase, por lo que está claro que cuando se necesitan, los atributos y métodos static **siempre aparecen y trabajan juntos**.

**Ejemplo:** Implementar un **contador de objetos** para llevar la cuenta de los objetos que se van instanciando.

*//contador para saber cuántos objetos de tipo Cliente se van instanciando (Ejemplo Cuentas bancarias)*

```
public class Cliente  
{  
    private String dni;  
    private String Nombre;  
    private static int contadorCli=0;  
  
    public Cliente(String dni, String nombre)  
    {  
        setDni(dni);  
        setNombre(nombre);  
        contadorCli++; // cada vez que se crea un nuevo objeto, lo contabilizamos - contadorCli++  
    }  
  
    public static int getContadorCli() {  
        return contadorCli;  
    }  
    ...  
}
```

Desde cualquier otra clase, podremos conocer cuántos objetos cliente llevamos creados:

```
System.out.println(Cliente.getContadorCli());
```

**Recordemos:** atributos static requieren SET/GET también static, y se llaman usando el nombre de la clase.

## RELACIONES ENTRE CLASES EN JAVA. ASOCIACIÓN.

Es un tipo de relación **muy frecuente** entre clases. Se da cuando uno o varios atributos de una clase son de un tipo de dato definido por otra. Es una relación **TODO-PARTE** entre clases. Una clase “tiene” a la otra.

### Ejemplos:

- En la aplicación **Cuentas bancarias**, la clase **Cuenta** tiene un atributo llamado **titular**, que es de tipo **Cliente** (la otra clase VO de la aplicación). Una **Cuenta** “tiene un” **Titular** (al menos)
- En la aplicación **Biblioteca**, la clase **Préstamo** tiene un atributo llamado **UsuarioPrest**, que es de tipo **Usuario**, y otro atributo llamado **libroPrest**, que es de tipo **Libro**. Un **Préstamo** “tiene un” **Libro** y “tiene un” **Usuario**.

En una relación de **ASOCIACIÓN**, una clase representa el **TODO** y “tiene a la otra/s”. Se puede hablar fundamentalmente de 2 tipos de asociación, que son la **AGREGACIÓN** (o Asociación “débil”) Y LA **COMPOSICIÓN** (o Asociación “fuerte”).

Las Asociaciones en general se representan en análisis con una línea que une ambas clases y se añade la cardinalidad [ **1**: Uno – **(0,1)** Cero a uno - **(n..m)** de n a m – **(0..\*)** de cero a muchos – **(1..\*)** de 1 a muchos ]

### Tipos de ASOCIACIÓN: AGREGACIÓN Y COMPOSICIÓN

**AGREGACIÓN.** Es la asociación que representa una relación TODO-parte ( “tiene ... ”)

A nivel práctico se suele llamar **AGREGACIÓN** cuando la relación se plasma mediante referencias a objetos, lo que permite que un componente esté referenciado en más de un “TODO” (Un usuario de la biblioteca puede estar referenciado en varios préstamos), y **lo más importante**, el objeto “parte” **tiene vida propia**, se crea de forma independiente y no se destruye cuando el todo que lo contiene desaparece.

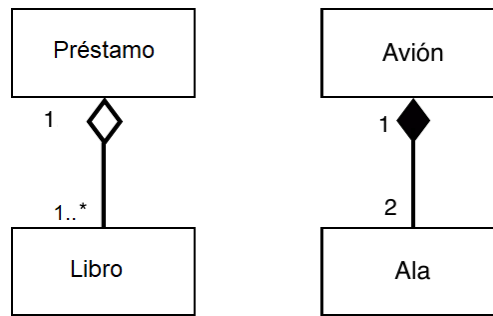
**Los 2 ejemplos vistos en clase son casos claros de AGREGACIÓN:**

- Un objeto de tipo **Cuenta** tiene como **Titular** a un objeto de tipo **Cliente**. Ese **Cliente** puede ser titular en otros objetos de tipo **Cuenta**.
- Lo mismo sucede con el **Libro** y el **Usuario** de un **Préstamo**. Ese **Libro** o ese **Usuario** pueden formar parte de otro **Préstamo**.

En ambos casos, insistir en que los objetos PARTE y TODO tienen vida propia e independiente, lo que significa que, si se **elimina** una **Cuenta**, el objeto asociado **Cliente** tiene sentido que siga “vivo”, ya que puede ser el titular de otra cuenta. De la misma forma si se elimina un **Préstamo**, los objetos asociados **Libro** y **Usuario** tiene sentido que sigan “vivos”.

**COMPOSICIÓN.** Es una asociación **fuerte**, en la que el objeto ‘parte’ está relacionado, como máximo, con una instancia del objeto “TODO”, de forma que cuando un objeto “TODO” es eliminado, también son eliminados sus objetos ‘parte’. **Por ejemplo:** un rectángulo **tiene** cuatro vértices.

Se suele hablar de **COMPOSICIÓN** cuando la relación es una inclusión **por valor** (el/los objeto/s componente/s están “incrustados” en un objeto “TODO” y sólo en ese). **Dentro del constructor** del objeto “TODO” se crean los objetos componentes, y si el TODO **desaparece**, los objetos “parte” **también**.



La **Agregación** se suele representar con un “diamante” blanco. La **Composición** con un “diamante” negro, aunque hay diversas formas y alternativas para la representación de relaciones entre objetos. En este punto lo importante es tener clara la cardinalidad de la asociación, y tener claro que, en el caso de la **Agregación**, que desaparezca un objeto **Préstamo** no obliga a que desaparezca su objeto **Libro** asociado, mientras en la **Composición**, si desaparece un objeto **Avión** deberían desaparecer automáticamente sus 2 objetos **Ala** asociados, porque claramente esos objetos no podían estar asociados a ningún otro Avión.

#### Ejemplo **AGREGACIÓN** clases **Cuenta - Cliente**:

```

public class Cuenta {
    private String iban;
    private Cliente titular;
    private double saldo;
    ...
}
public class Cliente
{
    private String dni;
    private String Nombre;
    ...
}
  
```

El atributo **Titular** para cada **Cuenta** será un objeto **Cliente**. Por ello, cuando instanciamos un objeto **Cuenta**, deberemos instanciar un objeto de tipo **Cliente**. **Están asociados, pero son independientes.**

```
Cuenta c1 = new Cuenta( generaliban() , new Cliente("22222222H","Luisa"), 2000)
```

#### Ejemplo **COMPOSICIÓN** Clases **Cuadrado – Punto**:

```

public class Cuadrado {
    Punto vertice1, vertice2, vertice3, vertice4;
    public Cuadrado(int x1,int y1,int x2,int y2,int x3,int y3,int x4,int y4){
        vertice1=new Punto(x1,y1);
        vertice2=new Punto(x2,y2);
        vertice3=new Punto(x3,y3);
        vertice4=new Punto(x4,y4);
    }
    ...
}
  
```

```
Cuadrado c = new Cuadrado (10,10,50,10,10,40,50,40);
```

Crearía un Cuadrado con 4 vértices (10,10) – (50,10) – (10,40) – (50,40)  
**(Si el objeto c desaparece, los 4 objetos de tipo Punto también)**

```

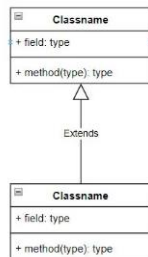
class Punto {
    private int x;
    private int y;
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
  
```

### 3. RELACIONES ENTRE CLASES EN JAVA. HERENCIA.

La **HERENCIA** es una de las características fundamentales de la Programación Orientada a Objetos. Mediante la herencia podemos **definir de forma jerárquica una clase a partir de otra ya existente**. La nueva clase se llama **clase derivada**, subclase o “hija”, y la clase superior **clase base**, superclase o “madre”.

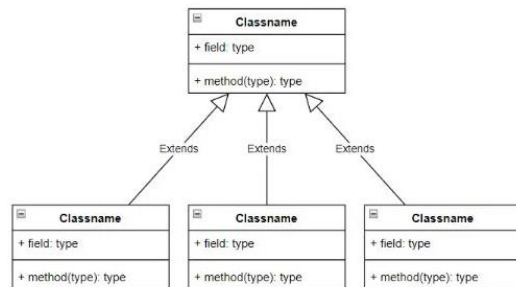
#### Herencia simple o única

Una clase hija hereda de una clase padre.



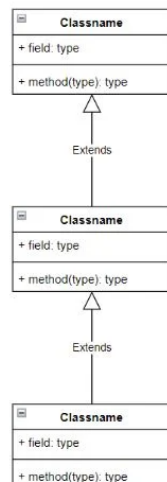
#### Herencia jerárquica

Una clase padre hereda a dos o más clases hijas.



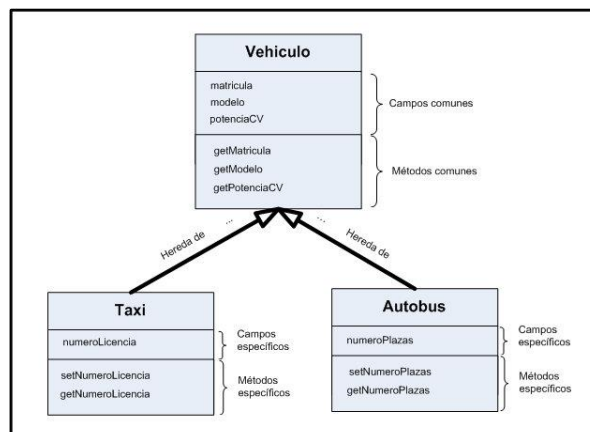
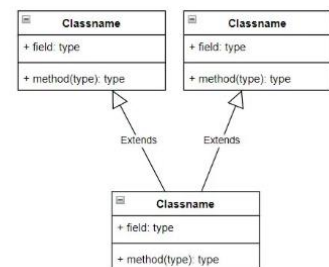
#### Herencia multinivel

Una clase hija hereda de una clase padre, que a su vez es clase hija de otra clase padre.



#### Herencia múltiple

Una clase hija hereda de dos o más clases padre.



La clase derivada **hereda** los componentes (atributos y métodos) de la clase base. Con eso se consigue:

- **Extender** la funcionalidad de la clase base, añadiendo nuevos atributos y métodos en la “hija”.
- **Modificar** el comportamiento de la clase base, **sobrescribiendo (@override)** algunos métodos heredados para adaptarlos a las necesidades de la clase derivada.
- **Reutilización del código** de la clase base.

Este curso ya hemos usado **HERENCIA SIN SABERLO**, pues en los ejemplos de clase hemos hecho **@override** al método **toString()**, para visualizar en formato String los atributos de un objeto. Lo hemos “sobreescrito” para modificar su comportamiento y adaptarlo a cada clase (Cuenta, Cliente, Libro, Usuario...)

El método **toString()** es **heredado** de la clase **Object**, que es la clase base-“madre” de todas las clases de Java. Toda clase en java es “hija” de la clase **Object**, y tiene un comportamiento básico heredado de **Object**. El método **toString()** pertenece a la clase **Object** pero está sin desarrollar, por eso lo que hacemos si queremos usarlo es sobrescribirlo, **y adaptarlo a nuestras necesidades en cada clase propia**.

#### @Override

```
public String toString() {  
    return "(DNI:"+ getDni() + ") " + getNombre();  
}
```

#### CARACTERÍSTICAS DE LA HERENCIA EN JAVA

- En java no existe la herencia múltiple entre clases (si entre **Interfaces**, como veremos). Una clase derivada no puede ser “hija” de varias clases base.
- La pregunta “**es un/una**” referida a la clase “hija” y la clase base es un buen test para probar relaciones de herencia. ¿un Taxi “es un” Vehículo?, ¿un Empleado “es una” Persona? ...
- Una clase derivada **hereda** de la clase base sus **atributos y métodos**, y puede añadir a ellos sus propios atributos y métodos (extender la funcionalidad de la clase base).
- **Los constructores no se heredan**. Las clases derivadas deben implementar su propio constructor.
- Una clase hija puede acceder a los **métodos public/protected** de su base como si fuesen suyos.
- Una clase hija puede ser base a la vez de otra. (**Herencia multinivel**)

#### SINTAXIS DE LA HERENCIA EN JAVA. EJEMPLO.

- La herencia en Java se expresa mediante la palabra **extends**
- Se hace referencia a los métodos de la clase base que no sean **public** con **super.método()**
- Se hace referencia al constructor de la clase base con **super()**
- A los métodos **public** de la clase base se puede acceder directamente

```
public class Figura    //CLASE BASE  
{  
    private int xPos;  
    private int yPos;  
    private String color;  
    public Figura(int xPos, int yPos, String color)  
    {  
        this.xPos = xPos;  
        this.yPos = yPos;  
        this.color = color;  
    }  
    GETTERS/SETTERS y demás métodos de COMPORTAMIENTO  
}
```

```
public class Circulo extends Figura //CLASE DERIVADA – “HIJA”
{
    private int diametro;
    public Circulo(int xPos, int yPos, String color, int diametro)
    {
        super(xPos,yPos,color); // llamada al constructor de la clase Base Figura
        this.diametro = diametro;
    }
    ...
}
```

Para crear un objeto de tipo Círculo: `Circulo sol = new Circulo(100,100,"amarillo",50);`

1. Esto producirá una 1ª llamada al **constructor de Figura** con los valores (100,100,"amarillo"). Como si hiciéramos un `new Figura(100,100,"amarillo")`.
2. A continuación, se completará la construcción del círculo llamando al **constructor de la clase Círculo** con el valor para el atributo diámetro (50)


En este ejemplo vemos como llamamos al constructor de la clase base (Figura) desde el constructor de la clase “hija” (Circulo). Si no lo hacemos, como en este caso, **la JVM lo hará automáticamente y sin parámetros**. Hará una especie de llamada invisible a `super()`, **y esto dará error** sino tenemos un constructor `Figura()` sin parámetros. Es una característica potente pero que hay que saber manejar.

**Podemos modificar el ejemplo para ver cómo funcionaría por defecto:** Simplificamos el constructor para crear las figuras sin parámetros en las posiciones X=0, Y=0 y color “amarillo”.

```
public Figura()
{
    xPos = 0; yPos = 0; color = "amarillo";
}
```

El constructor para **Círculo** quedaría:

```
public Circulo(int diametro) // la JVM ejecutará una 1ª llamada super() al constructor Figura()
{
    this.diametro = diametro;
}
```



`Circulo sol = new Circulo(50);` // crea un círculo con pos(x,y) = (0,0) , color “amarillo” y diámetro 50.

Hemos simplificado la construcción de objetos. El constructor Circulo llama 1º al constructor de **Figura** con un `super()` implícito, y después termina de construir el círculo con el diámetro indicado. Es más sencillo, pero no nos permite crear los objetos desde el principio con la posición y color deseados. Podemos decidir construir los objetos como más convenga, siendo conscientes de lo que hacemos. **Recordemos:**

- **super.** nos permite acceder a los métodos y propiedades ocultos de la clase padre.
- **super()** nos permite ejecutar el constructor de la clase padre.
- **this.** nos permite acceder a los métodos y propiedades de la propia clase.
- **this()** nos permite ejecutar el constructor de la propia clase.

**Como ejemplo de Herencia Se entrega al Alumnado el Proyecto Figuras (3 versiones). ES IMPORTANTE PRACTICAR con él y entenderlo.**

## 4. CLASES ABSTRACTAS.

En herencia, **a menudo la clase base no necesita ser instanciada**. En el ejemplo anterior, no necesitamos crear objetos de tipo **Figura**, sino que vamos a crear objetos de tipo **Circulo, Cuadrado ...Triangulo**.

**Circulo, Cuadrado, Triangulo ...** son figuras geométricas que comparten una parte importante de su estructura y comportamiento, utilizamos la **herencia** para dar forma a esa parte común (clase base **Figura**) y así creamos código limpio y reutilizable. Empleando **HERENCIA** creamos las clases derivadas “hijas” para cada tipo de figura geométrica con sus características específicas: **Circulo, Cuadrado, Triangulo ...**

En realidad, nunca vamos a necesitar instanciar un objeto de la clase **Figura**. La clase **Figura** resulta de utilidad como clase base para la herencia, **pero no como una clase** de la cual vayan a surgir objetos. **A este tipo de clases se les llama clases ABSTRACTAS**, y se diseñan básicamente para que otras clases las hereden.

**Clase Abstracta:** clase que nunca será instanciada, sino que proporciona un marco o modelo a seguir por sus clases derivadas dentro de una jerarquía de herencia. **No se pueden crear objetos de una clase Abstracta.**

### LAS CLASES ABSTRACTAS:

- Incluyen 0 o más métodos abstractos. (que deberán ser reescritos **@override** en las clases “hijas”)
- Pueden incluir atributos y métodos no abstractos completos (serán heredados por las hijas).
- No son instanciables. Aunque tengan constructor, éste sólo puede ser utilizado por las clases “hija”. **No podemos hacer un new() de una clase Abstracta.**

La idea de las **clases abstractas** es que tengamos una clase padre que sirva como plantilla para las **clases hijas**, en las clases hijas se implementaran los métodos abstractos de la **clase padre**. Las clases Abstractas dan soporte a la ABSTRACCIÓN.

Para ver como se construyen y funcionan las clases abstractas **adaptaremos el Ejemplo visto en clase de Figuras Geométricas**, haciendo que la clase base de la Herencia (**Figura**) sea Abstracta. (**Habría sido lo más lógico desde un principio**). Los cambios son muy sencillos. Simplemente añadir **abstract** a la clase **Figura**, y declarar el método **dibujar()** también como **abstract** para que sea sobrescrito en cada clase hija.

```
public abstract class Figura
{
    private int xPos;
    private int yPos;
    private String color;
    public Figura(int xPos, int yPos, String color)
    {
        this.xPos = xPos;
        this.yPos = yPos;
        this.color = color;
    }
    // GETTERS/SETTERS y todo lo demás
}
```

### ¿QUÉ VENTAJAS APORTAN LAS CLASES ABSTRACTAS?

- Se impide la creación no necesaria de objetos de la clase abstracta. **pero si se pueden crear colecciones de objetos de una clase abstracta**. Podemos crear una colección de **Figuras**, y manejar en la colección objetos mezclados de las clases “hijas” **Circulo, Cuadrado, Triangulo ...**
- Se simplifica el **casting entre objetos**, dado que la clase base no es instanciable, y se simplifica la utilización del operador **instanceof()**

## ¿QUÉ HA PASADO CON LAS CLASES ABSTRACTAS A PARTIR DE JAVA 8?

- Desde **Java 8**, existe la posibilidad de introducir métodos **default** en las **Interfaces** (se verá el concepto en el siguiente punto) y esto ha hecho que para muchos programadores las clases **Abstract** hayan perdido mucho interés y funcionalidad. **Casi todo lo que antes se hacía con Herencia y clases Abstractas, a partir de Java 8 se hace con Interfaces y métodos default**, y las clases Abstractas han quedado bastante en el olvido a nivel práctico. Es más, las múltiples posibilidades de la POO en cuando a las relaciones entre objetos, sobre todo **HERENCIA Y COMPOSICIÓN** han generado corrientes de opinión contrapuestas, y hay grupos de desarrolladores que apuestan por descartar la **HERENCIA** porque puede volverse compleja y difícil de gobernar. En su lugar, se apuesta más por análisis orientados a la **COMPOSICIÓN**, que a menudo dan solución a los mismos problemas de relaciones entre objetos con un menor nivel de complejidad.

**HERENCIA Vs COMPOSICIÓN será un tema a discutir/debatir/reflexionar a lo largo del curso.**

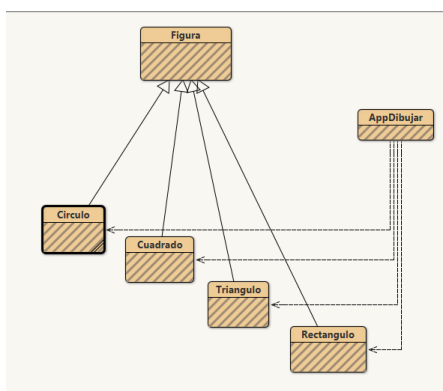
## 5. POLIMORFISMO EN JAVA, ENLAZADO DINÁMICO.

El **polimorfismo** es una de las características típicas de la POO y está ligada a la Herencia. Se define como la cualidad que tienen los objetos para responder de distinto modo a una misma llamada a un método.

Para conseguir un comportamiento polimórfico en un programa Java se debe cumplir lo siguiente:

- Los métodos deben estar implementados o declarados (si son abstractos) en la clase base.
- Los métodos deben estar redefinidos **@override** en las clases derivadas o “hijas”.
- Los objetos deben ser manipulados utilizando referencias a la clase base.

Para probar el polimorfismo utilizaremos de nuevo la aplicación de figuras geométricas con la que estamos trabajando. Tenemos la clase base **Figura**, y sus clases derivadas **Círculo**, **Cuadrado**, **Rectángulo** y **Triángulo**.



La clase base **Figura** contiene un método **dibujar()** que es Abstracto (**abstract**) y que está sobreescrito **@override** en cada una de las clases “hija”. Esto es normal, ya que **Cuadrado**, **Círculo**, **Triángulo** y **Rectángulo** heredan muchas cosas de la clase **Figura**, pero no el método **dibujar()** pues cada tipo de figura se dibuja de una forma diferente, y el método **dibujar()** habrá que reescribirlo para cada una de ellas.



Para probar el **polimorfismo** vamos a crear un **ArrayList** de objetos de tipo **Figura**:

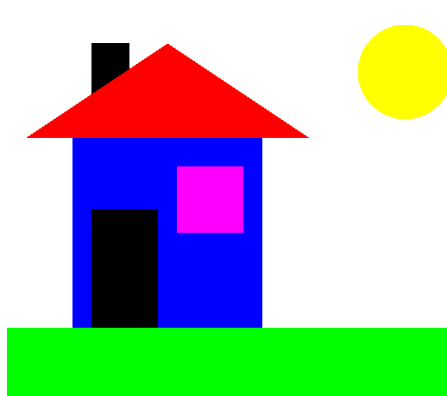
```
ArrayList<Figura> dibujo = new ArrayList();
```

**¿Se puede?** **Figura** es una clase abstracta y **no se pueden crear objetos sueltos** de tipo **Figura**, **pero si una colección**, y esto es interesante, pues en ese **ArrayList** podremos añadir objetos de tipo **Cuadrado**, **Círculo**, **Triángulo** y **Rectángulo**. (Lo mismo nos pasará más adelante con las **interfaces Java**)

```
dibujo.add(new Cuadrado(200,200,"azul",200));
dibujo.add(new Cuadrado(310,230,"magenta",70));
dibujo.add(new Rectangulo(220,275,"negro",125,70));
dibujo.add(new Rectangulo(220,100,"negro",60,40));
dibujo.add(new Triangulo(300,100,"rojo", 100,300));
dibujo.add(new Circulo(500,80,"amarillo",100));
for (Figura f:dibujo){
    f.dibujar();
}
```

El **polimorfismo se produce** cuando para cada uno de los objetos (diferentes) que hemos añadido al **ArrayList**, invocamos un método de la clase base que está **sobreescrito** en cada una de las clases “hija”.

En este caso llamamos a **dibujar()**, que es un método que en la clase base **Figura** está sin definir, pero si en las “hijas” y sorprende el resultado porque ... **funciona!!** ... gracias al polimorfismo.



El **polimorfismo** es posible porque el método que se ejecuta **lo decide la JVM durante la ejecución del programa**. A este proceso de decidir en tiempo de ejecución qué método se ejecuta se le denomina **enlazado dinámico**, que es el mecanismo que hace posible el **polimorfismo**.

Lo que ha pasado, es que **f** llama en cada vuelta del **ForEach** al método **dibujar()**, siendo **f** aparentemente un objeto de tipo **Figura** que ni siquiera tiene definido el método **dibujar()**

Sabemos que las variables Objeto en Java no contienen el objeto en sí mismo, sino la dirección de memoria HEAP dónde ese objeto “vive”. Son una referencia al objeto. De esta forma la **JVM** se da cuenta en tiempo de ejecución de que detrás de **f** (declarado como objeto de tipo **Figura**) hay objetos diferentes de las clases derivadas **Cuadrado**, **Círculo**, **Triángulo** y **Rectángulo**. Java decide entonces ejecutar el método **dibujar()** correspondiente a la clase de cada objeto del **ArrayList**, y no el método **dibujar()** vacío de la clase **Figura**. Java da prioridad al objeto que se encuentra detrás de la variable, antes que al tipo de la variable en sí. De esta forma logra un comportamiento polimórfico ejecutando el método adecuado a cada objeto. **Magia**.

Para entender y ver todo esto en detalle, será necesario revisar el código del ejemplo Figuras Geométricas, tanto el de la clase base **Figura**, como el de las clases derivadas **Cuadrado**, **Círculo**, **Triángulo** y **Rectángulo**.

## 6. INTERFACES EN JAVA.

Las **interfaces** Java permiten llevar aún más allá las posibilidades de la Herencia. Al igual que las clases **Abstractas** son un tipo **especial** de Clase, con la peculiaridad de que **TODOS sus métodos tenían que ser abstractos (no implementados)**. Por tanto, las clases “hija” de la interface estarían obligadas a implementar todos esos métodos. En ese sentido una Interface actúa como un **patrón** o un “**contrato**” a seguir.

Con java **8 y 9** se amplía la gama de métodos que puede contener una interface, y resulta especialmente importante la posibilidad de incluir métodos **default**, que son métodos que pueden implementarse en la propia interface y que hacen desaparecer la limitación original de las interfaces de no poder tener ningún método “propio”. Gracias a esto, desde Java 8 se opta por usar **interfaces en los proyectos con Herencia** en la gran mayoría de casos.

### Las interfaces:

- No se pueden instanciar. No podemos crear objetos a partir de una Interface.
- No tienen constructor.
- Todos sus atributos (si tienen) son por defecto **public static final**.
- Admiten **herencia múltiple**. Es la forma de hacerlo en Java, y de ahí su importancia. Una Interface puede heredar de varias interfaces.

### ¿PARA QUÉ SIRVE UNA INTERFACE?

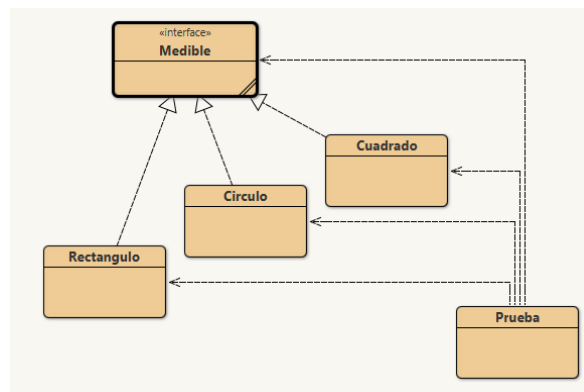
Al principio servían como guion para el diseño de clases, marcando la línea a seguir y obligando a las clases “hijas” a **definir un comportamiento concreto**. Una **interface** no se hereda, se **implementa (implements)** y la clase que lo hace está obligada a desarrollar **TODOS** los métodos abstractos que contiene la interface.

Actualmente, desde java 8, las interfaces se usan en cualquier proyecto POO con herencia, pues gracias a los métodos **default** cubren todas las necesidades:

- Definen un comportamiento o guion de diseño (papel original de las interfaces)
- Pueden proporcionar funcionalidad (métodos default) a las clases “hijas” para que la amplíen.
- Permiten herencia múltiple.

Así pues, podemos concluir que casi todo aquello que hagamos en Java **con relaciones de herencia** entre clases lo haremos con **Interfaces**. En eso no hay discusión. En lo que si hay discusión es, como se ha comentado, en si conviene analizar los proyectos Java en POO hacia **Herencia** o **Composición** de clases.

**Ejemplo:** interface **Medible** que defina el comportamiento matemático de figuras geométricas que nos obligue a implementar los métodos **area()** y **perímetro()** en las Clases para todas las figuras geométricas. (Se recomienda que los nombres de las interfaces terminen en **able**: Dibujable, Comparable, Relacionable ... )



```

public interface Medible {
    float PI = 3.1416f; // Por defecto public static final en una interface
    float area(); // Por defecto public abstract en una interface
    float perimetro();
}

```

```

public class Cuadrado implements Medible {
    private float lado;
    public Cuadrado (float lado){
        this.lado=lado;
    }
    // resto de métodos de la clase Cuadrado

    public float area() {
        return lado*lado;
    }
    public float perimetro() {
        return lado*4;
    }
}

```

```

public class Circulo implements Medible {
    private float radio;
    public Circulo (float radio){
        this.radio=radio;
    }
    // resto de métodos de la clase Circulo

    public float area() {
        return PI* (float) Math.pow(radio,2);
    }
    public float perimetro() {
        return 2*PI*radio;
    }
}

```

```

public class Rectangulo implements Medible {
    private float base;
    private float altura;
    public Rectangulo (float base, float altura) {
        this.base=base;
        this.altura=altura;
    }
    // resto de métodos de la clase Rectángulo

    public float area() {
        return (base*altura);
    }
    public float perimetro() {
        return (2*base+2*altura);
    }
}

```

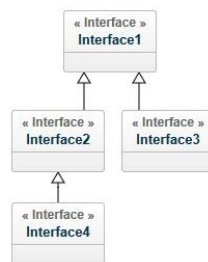
**Ejemplo:** Igual que hicimos para probar el **polimorfismo** con una clase **abstract**, también se puede crear una **colección** de objetos declarando una **interface** como tipo. A esa colección podemos añadirle objetos de cualquier tipo que haya implementado la interface, y Java se encargará de aplicar el **polimorfismo** a la hora de invocar a los métodos **area()** y **perimetro()** según el tipo de objeto del que se trate.

ArrayList <Medible> figuras = new ArrayList (); // figuras puede contener Cuadrados, Círculos y Rectángulos

```
import java.util.ArrayList;
public class Prueba {
    public static void main (String [ ] Args) {
        ArrayList <Medible> figuras = new ArrayList ();
        figuras.add (new Cuadrado (3.5));
        figuras.add (new Cuadrado (5));
        figuras.add (new Circulo (9));
        figuras.add (new Cuadrado (7.2));
        figuras.add (new Rectangulo (4 , 8.2));
        figuras.add (new Rectangulo (6.3 , 3.3));
        for (Medible m : figuras){
            System.out.println ("Area: " + m.area() + "\t\tPerímetro: " +m.perimetro());
        }
    }
}
```

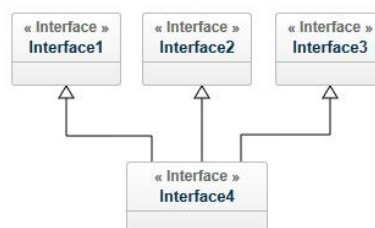
Area:	12.25	Perímetro:	14.0
Area:	25.0	Perímetro:	20.0
Area:	50.2656	Perímetro:	25.1328
Area:	49.0	Perímetro:	28.0
Area:	32.8	Perímetro:	24.4
Area:	20.79	Perímetro:	19.2

## HERENCIA ENTRE INTERFACES



```
public interface Interface2 extends Interface1 { } //Las interfaces se heredan entre si igual que las clases (extends)
public interface Interface3 extends Interface1 { }
public interface Interface4 extends Interface2 { }
```

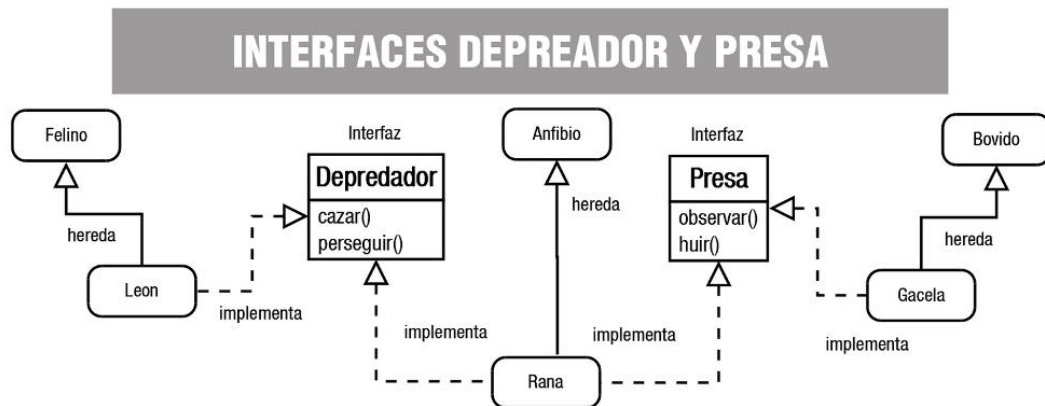
## HERENCIA MÚLTIPLE:



```
public interface Interface4 extends Interface1, Interface2, Interface3{
}
public class Prueba implements Interface4 {
    //la clase prueba debería implementar todos los métodos de las interfaces 1,2,3,4
}
```

## RELACIONES CLASES/INTERFACES.

Las relaciones de herencia entre clases e interfaces pueden ser muy variadas. En la siguiente **figura** vemos un sencillo **ejemplo** de cómo una clase puede heredar de otra, y a la vez implementar un interface ... o varias.



```
public class Leon extends Felino implements Depredador {
    //hereda atributos + métodos de Felino y debe implementar los métodos cazar() - perseguir() de Depredador
}
public class Rana extends Anfibio implements Depredador, Presa {
    //hereda atributos + métodos de Anfibio y debe implementar los métodos:
    cazar() - perseguir() de Depredador + observar() - huir() de Presa
}
```