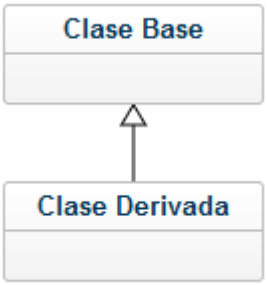
Tema 7 Herencia y Polimorfismo

1. HERENCIA

La herencia es una de las características fundamentales de la POO.

Mediante la herencia podemos **definir una clase a partir de otra ya existente**.
La clase nueva se llama **clase derivada** o subclase y la clase existente se llama **clase base** o superclase.



En UML la herencia se representa con una flecha apuntando a la clase base.

La clase derivada hereda los componentes (atributos y métodos) de la clase base.

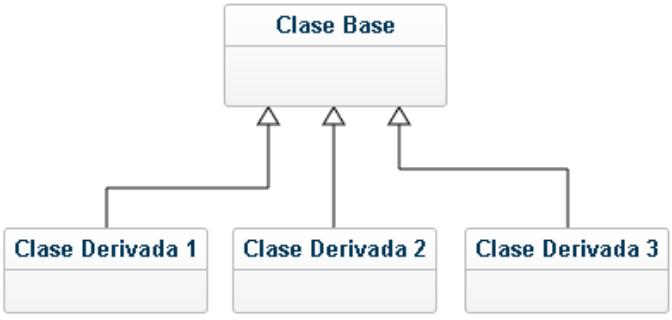
La finalidad de la herencia es:

- **Extender** la funcionalidad de la clase base: en la clase derivada se pueden **añadir** atributos y métodos nuevos.
- **Especializar** el comportamiento de la clase base: en la clase derivada se pueden **modificar** (sobrescribir, override) los métodos heredados para adaptarlos a sus necesidades.

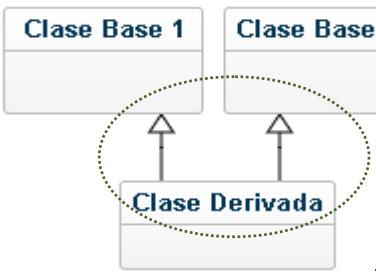
La herencia permite la **reutilización del código**, ya que evita tener que reescribir de nuevo una clase existente cuando necesitamos ampliarla en cualquier sentido. Todas las clases derivadas pueden utilizar el código de la clase base sin tener que volver a definirlo en cada una de ellas.

- **Reutilización de código:** El código se escribe una vez en la clase base y se utiliza en todas las subclases.

Una clase base puede serlo de tantas derivadas como se desee: Un solo padre, varios hijos.



Java no permite la herencia múltiple. En java una clase derivada solo puede tener una clase base.



Herencia múltiple

NO PERMITIDA EN JAVA

La herencia expresa una **relación "ES UN/UNA"** entre la clase derivada y la clase base.

Esto significa que **un objeto de una clase derivada es también un objeto de su clase base**.

Al contrario NO es cierto. Un objeto de la clase base no es un objeto de la clase derivada.

Por ejemplo, supongamos una clase Vehículo como clase base y una clase Coche derivada de Vehículo. Podemos decir que un Coche es un Vehículo pero un Vehículo no siempre es un Coche, puede ser una moto, un camión, etc.

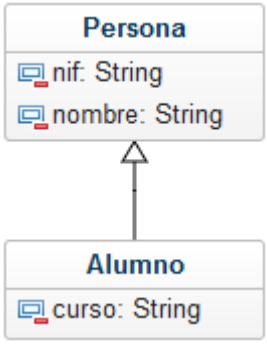
Un objeto de la clase derivada se puede utilizar en cualquier lugar donde aparezca un objeto de la clase base.

Si esto no es posible entonces la herencia no está bien planteada.

Ejemplo de herencia: Tenemos una clase Persona que tiene como atributos el nif y el nombre. Queremos crear una clase Alumno que tenga los atributos nif, nombre y curso.

En este caso podemos crear la clase Alumno como derivada de la clase Persona.

Un Alumno es una Persona que tendrá como atributos nif, nombre y curso.



Ejemplo de herencia: Un Alumno ES UNA Persona

En UML los componentes privados de una clase se representan con el signo –

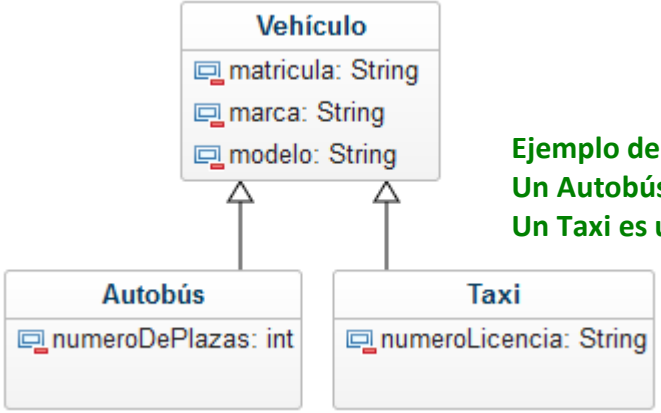
Los componentes protected se representan con el signo #

Los componentes públicos se representan con el signo +

Los componentes package se representan con el signo ~

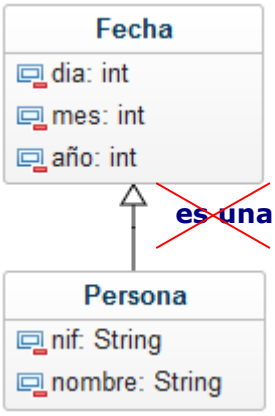
Los componentes static se representan subrayados.

Ejemplo de herencia: Tenemos una clase Vehículo que tiene como atributos la matrícula, marca y modelo. Queremos crear las clases Autobús que contenga los atributos matrícula, marca, modelo y número de plazas y la clase Taxi que contenga los atributos matrícula, marca, modelo y número de licencia. Podemos derivar las clases Autobús y Taxi de la clase Vehículo.



Ejemplo de herencia:
Un Autobús es un Vehículo
Un Taxi es un Vehículo

Ejemplo de herencia mal planteada: Tenemos una clase Fecha que tiene como atributos día, mes y año. Queremos crear una clase Persona que tenga como atributos el nif, nombre y la fecha de nacimiento. Podríamos plantearnos crear la clase Persona como derivada de la clase Fecha ya que esta clase contiene los atributos día, mes y año que nos pueden servir como fecha de nacimiento y así no tenemos que incluirlos en la clase Persona, pero este es un caso de herencia mal planteada:



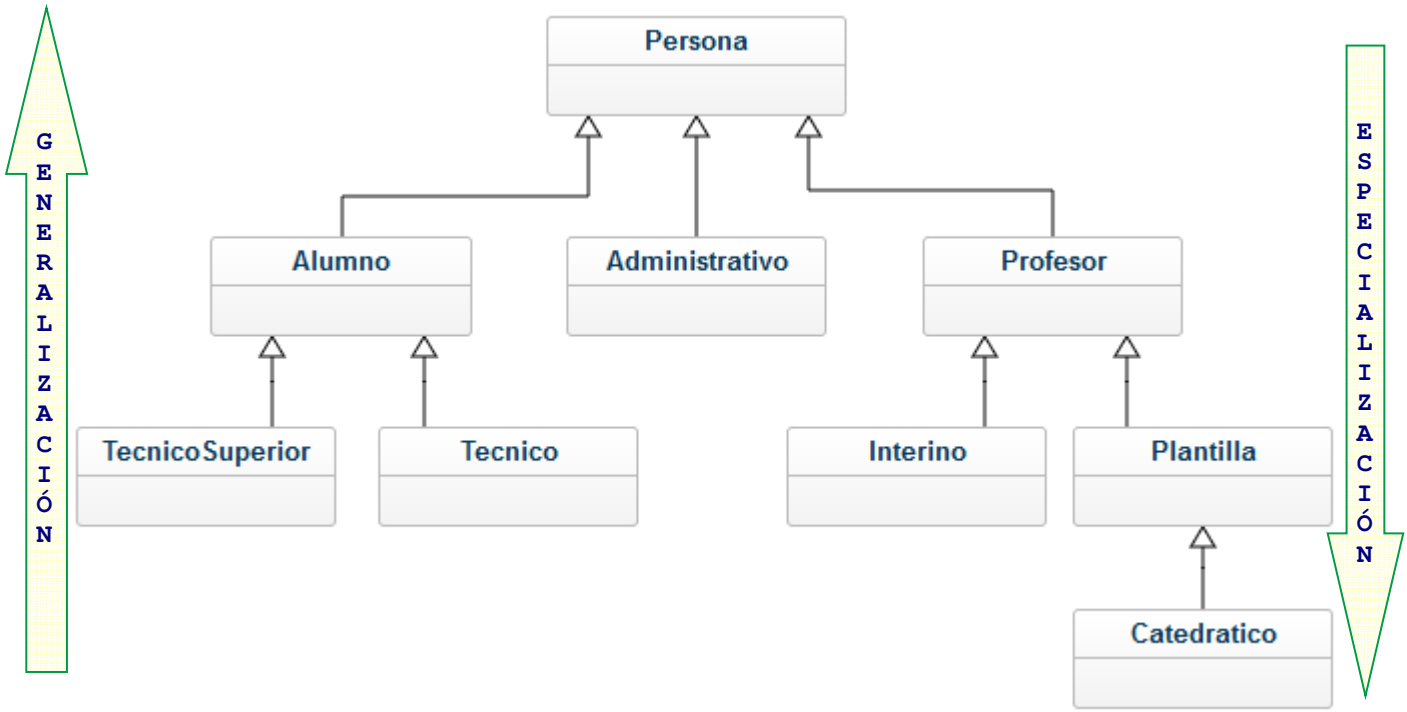
Ejemplo de herencia mal planteada:
Una persona NO ES una Fecha

Una Persona CONTIENE una fecha de nacimiento

EN ESTE CASO NO SE DEBE UTILIZAR LA HERENCIA

Jerarquía de Clases

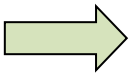
Una clase derivada a su vez puede ser la clase base de otra clase en un nuevo proceso de derivación, formando de esta manera una Jerarquía de Clases.



Las clases más generales se sitúan en lo más alto de la jerarquía.

Cuánto más arriba en la jerarquía, menor nivel de detalle.

Cada clase derivada debe implementar únicamente lo que la distingue de su clase base.



En java todas las clases derivan directa o indirectamente de la **clase Object**.

Todas las clases Java, tanto las que forman el lenguaje como las que crean los usuarios derivan de la clase Object y por lo tanto heredan los métodos que contiene la clase Object entre ellos el método toString() y el método equals().

Cuando creamos una clase no hay que indicar que hereda de Object porque esto ya va implícito en el lenguaje.

Object es la clase base de toda la jerarquía de clases Java.

Todos los objetos en un programa Java son Object.

2. CLASES DERIVADAS

CARACTERÍSTICAS

- Una clase derivada hereda de la clase base los componentes (atributos y métodos) de la clase base.
- Los constructores no se heredan. Las clases derivadas deberán implementar sus propios constructores.
- Una clase derivada puede acceder a los miembros públicos y protegidos de la clase base como si fuesen miembros propios.
- Una clase derivada no tiene acceso a los miembros privados de la clase base. Deberá acceder a través de métodos heredados de la clase base.
- Si se necesita tener acceso directo a los miembros privados de la clase base se deben declarar protected en lugar de private en la clase base.
- Una clase derivada puede añadir a los miembros heredados, sus propios atributos y métodos (extender la funcionalidad de la clase).
- También puede modificar los métodos heredados (override, especializar el comportamiento de la clase base).
- Una clase derivada puede, a su vez, ser una clase base, dando lugar a una jerarquía de clases.

RESUMEN DE LOS MODIFICADORES DE ACCESO

	ACCESO EN			
MODIFICADOR	**PROPIA CLASE**	**PACKAGE**	**CLASE DERIVADA**	**RESTO**
private	**SI**	**NO**	**NO**	**NO**
<Sin modificador>	**SI**	**SI**	**NO**	**NO**
protected	**SI**	**SI**	**SI**	**NO**
public	**SI**	**SI**	**SI**	**SI**

SINTAXIS

La herencia en Java se expresa mediante la palabra **extends**

Por ejemplo, para declarar una clase B que hereda de una clase A:

```
public class B extends A{
    . . . . .
}
```

Ejemplo: vamos a utilizar una clase Persona con los atributos *nif* y *nombre* y de ella vamos a obtener una clase derivada Alumno.

```
public class Persona {
    private String nif;
    private String nombre;

    public String getNif() {
        return nif;
    }
    public void setNif(String nif) {
        this.nif = nif;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

Un alumno **ES UNA** persona. Suponiendo que un alumno tiene como atributos nif, nombre y curso podemos derivar la clase Alumno de la clase Persona y añadir el atributo curso:

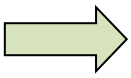
```
public class Alumno extends Persona{

    private String curso;

    public String getCurso() {
        return curso;
    }
    public void setCurso(String curso) {
        this.curso = curso;
    }
}
```

La clase alumno hereda los atributos *nombre* y *nif* de la clase Persona y añade el atributo propio *curso*. Por lo tanto:

- Los atributos de la clase Alumno son nif, nombre y curso.
- Los métodos de la clase Alumno son: getNif(), setNif(String nif), getNombre(), setNombre(String nombre), getCurso(), setCurso(String curso).



La clase Alumno aunque hereda los atributos nif y nombre, no puede acceder a ellos de forma directa ya que son privados a la clase Persona.

Para acceder desde la clase Alumno a los atributos privados de la clase Persona se utilizarán los métodos get/set que se han heredado de la clase Persona.

La clase Alumno puede utilizar los componentes public y protected de la clase Persona como si fueran propios.

Un ejemplo de uso de la clase Alumno desde la clase principal:

```
public static void main(String[] args) {

        Alumno a = new Alumno();
        a.setNombre("Eliseo Gonzáles Manzano");
        a.setNif("12345678-Z");
        a.setCurso("1DAW");
        System.out.println("Nombre" + a.getNombre());
 }
```

Se utilizan los métodos heredados de la clase base Persona

En una jerarquía de clases, cuando un objeto invoca a un método:

1. Se busca en su clase el método correspondiente.

2. Si no se encuentra, se busca en su clase base.

3. Si no se encuentra se sigue buscando hacia arriba en la jerarquía de clases.

4. Si al llegar a la clase base raíz (Object) el método no se ha encontrado se producirá un error.

En el ejemplo anterior al ejecutar por ejemplo la instrucción:

```
a.setNombre("Eliseo Gonzáles Manzano");
```

Se busca el método *setNombre* en la clase Alumno que es a la que pertenece el objeto *a*.

Como el método no se encuentra en la clase Alumno se busca en su clase base que es Persona. El método se encuentra en la clase Persona y se ejecuta.

REDEFINIR (MODIFICAR) MIEMBROS HEREDADOS DE LA CLASE BASE.

Redefinir métodos (Override)

Los métodos heredados de la clase base se pueden redefinir o modificar en las clases derivadas.

El método que queremos modificar en la clase derivada se debe escribir con el mismo nombre, el mismo número y tipo de parámetros y el mismo tipo de retorno que en la clase base. Si no fuera así estaríamos sobrecargando el método, no redefiniéndolo.

El método redefinido puede tener un modificador de acceso menos restrictivo que el de la clase base. Si por ejemplo el método heredado es *protected* se puede redefinir como *public* pero no como *private* porque sería un acceso más restrictivo que el que tiene en la clase base.

Cuando en una clase derivada se redefine un método de una clase base, se oculta el método de la clase base.

Si queremos acceder al método de la clase base que ha quedado oculto en la clase derivada utilizamos: **super.metodo();**

Mediante *super* estamos indicando que el método que queremos ejecutar no es el que se encuentra en esta clase sino que es el que se encuentra en la clase base ya que ambos se llaman igual.

Si se quiere evitar que un método sea redefinido en la clase derivada se debe declarar como método **final**. Por ejemplo:

```
public final void metodo(){
        ........
}
```

Ejemplo de método heredado y modificado en la clase derivada:

Vamos a añadir a la clase Persona un método *leer* para introducir por teclado los valores a los atributos:

```
public class Persona {
    private String nif;
    private String nombre;
    public String getNif() {
        return nif;
    }
    public void setNif(String nif) {
        this.nif = nif;
    }
    public String getNombre() {
        return nombre;
    }
```

```
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public void leer(){
        Scanner sc = new Scanner(System.in);
        System.out.print("Nif: ");
        nif = sc.nextLine();
        System.out.print("Nombre: ");
        nombre = sc.nextLine();
    }
}
```

La clase Alumno hereda de la clase persona y por lo tanto hereda este método.

Un objeto Alumno puede utilizar el método leer():

```
Alumno a = new Alumno();
a.leer();
```

Pero este método leer() que se encuentra en la clase Persona solo asignará valores al nif y al nombre.

Para que asigne también el valor al atributo curso necesitamos sobrescribir el método leer() en la clase Alumno. De esta forma podremos introducir por teclado los atributos nif, nombre y curso.

```
//método leer en la clase Alumno.
//Se modifica el método leer heredado de Persona
@Override
public void leer(){
        Scanner sc = new Scanner(System.in);
        String nifAlumno, nombreAlumno;
        System.out.print("Nif: ");
        nifAlumno = sc.nextLine();
        setNif(nifAlumno);
        System.out.print("Nombre: ");
        nombreAlumno = sc.nextLine();
        setNombre(nombreAlumno);
        System.out.print("Curso: ");
        curso = sc.nextLine();
}
```

nif y nombre son atributos privados en Persona. Para asignarles los valores introducidos por teclado en el método leer de la clase Alumno se deben utilizar los métodos set heredados de Persona.

Pero **este método NO es del todo correcto ya que solo debería pedir el *curso* que el atributo de la clase Alumno y hacer uso del método leer de Persona donde se introduce el nombre y el nif**.

No olvidemos que uno de los objetivos al utilizar la herencia es reutilizar código.

Por lo tanto el **método leer en la clase Alumno escrito de forma correcta** sería así:

```
@Override
public void leer(){
        Scanner sc = new Scanner(System.in);
        super.leer();
        System.out.print("Curso: ");
        curso = sc.nextLine();
}
```

Llamada al método leer de la clase Persona para leer los atributos nif y nombre

Lectura del atributo curso

Este método leer() que hemos escrito en la clase Alumno oculta al método leer() de la clase Persona.

Esto quiere decir que cuando se ejecuta la instrucción `a.leer()` se está invocando al método leer() de Alumno y no al método leer() de Persona.

Dentro del método leer(), la instrucción:

```
super.leer();
```

está invocando al método leer de la clase Persona.

Ejemplo de método heredado y modificado en la clase derivada:

Otro ejemplo de sobreescritura de un método heredado es el método toString.

Si tenemos un método toString en Persona que devuelva un String con el nif y el nombre, podemos modificarlo en la clase Alumno para que devuelva un String con el nif, el nombre y el curso del alumno. Para ello el método toString de Alumno invoca al toString de Persona:

```
//Clase Persona
public class Persona {

    ..............

    //método toString en la clase Persona
    public String toString() {
        return "Nif: " + nif + "\nNombre: " + nombre;
    }
}
```

```
public class Alumno extends Persona{

    ..............

    //método toString en la clase Alumno
    @Override
    public String toString() {
        return super.toString() + "\nCurso: " + curso;
    }
}
```

Se llama al método toString de la clase Persona

Redefinir atributos

Una clase derivada puede volver a declarar un atributo heredado de la clase base. Incluso puede declararlo con un tipo diferente.

Cuando se redefine un atributo en una clase derivada el atributo de la clase base queda oculto por el de la clase derivada.

Para acceder a un atributo de la clase base que ha quedado oculto en la clase derivada se escribe: **super.atributo;**

Solo se pueden redefinir atributos que sean *public* o *protected* en la clase base.

Generalmente no se recomienda redefinir atributos en las clases derivadas porque produce un código difícil de leer.

3. CONSTRUCTORES EN CLASES DERIVADAS

Los constructores no se heredan. Cada clase derivada tendrá sus propios constructores.

La clase base es la encargada de inicializar sus atributos.

La clase derivada se encarga de inicializar solamente los suyos.

Cuando se crea un objeto de una clase derivada:

1º –> Se ejecuta el constructor de la clase base

2º –> Se ejecuta el constructor de la clase derivada.

Esto lo podemos comprobar si añadimos los constructores por defecto a las clases Persona y Alumno y hacemos que cada constructor muestre un mensaje:

```
public class Persona {
    private String nif;
    private String nombre;
    public Persona() {
        System.out.println("Ejecutando el constructor de Persona");
    }
    /////// Resto de métodos
}
```

```
public class Alumno extends Persona{
    private String curso;
    public Alumno() {
        System.out.println("Ejecutando el constructor de Alumno");
    }
    /////// Resto de métodos
}
```

Si creamos un objeto Alumno:

```
Alumno a = new Alumno();
```

Se muestra por pantalla:

```
Ejecutando el constructor de Persona
Ejecutando el constructor de Alumno
```

Cuando se invoca al constructor de la clase Alumno se invoca automáticamente al constructor de la clase Persona y después continúa la ejecución del constructor de la clase Alumno.

El constructor por defecto de la clase derivada llama al constructor por defecto de la clase base.

La instrucción para invocar al constructor por defecto de la clase base es: **super();**

Todos los constructores en las clases derivadas contienen de forma implícita la instrucción super() como primera instrucción.

```
public Alumno() {
        super();
        System.out.println("Ejecutando el constructor de Alumno");
}
```

Llamada al constructor por defecto de Persona.
No es necesario escribirlo ya que se hace de forma automática

Cuando se crea un objeto de la clase derivada y queremos asignarle al mismo tiempo valores a sus atributos, tanto a los propios como a los heredados de la clase base:

- La clase derivada debe tener un constructor con parámetros adecuado que reciba los valores a asignar a los atributos de la clase base.
- La clase base debe tener un constructor con parámetros adecuado.
- En el constructor con parámetros de la clase derivada se debe invocar al constructor con parámetros de la clase base y se le envían los valores iniciales de los atributos. Debe ser la primera instrucción que aparezca en el constructor de la clase derivada.
- La clase base es la encargada de asignar valores iniciales a sus atributos.
- A continuación el constructor de la clase derivada asigna valores a los atributos de su clase.

Por ejemplo en las clases Persona y Alumno anteriores añadimos los constructores con parámetros:

```
public class Persona {
    private String nif;
    private String nombre;
    public Persona() {
        System.out.println("Ejecutando el constructor de Persona");
    }
    //Constructor con parámetros de Persona.
    //Recibe los valores de los atributos de la clase Persona
    public Persona(String nif, String nombre) {
        this.nif = nif;
        this.nombre = nombre;
    }
    /////// Resto de métodos
}
public class Alumno extends Persona{
    private String curso;
    public Alumno() {
        System.out.println("Ejecutando el constructor de Alumno");
    }
    //Constructor con parámetros de Alumno.
    //Recibe los valores de todos los atributos de la clase Alumno
    //Los propios y los heredados de Persona
    public Alumno(String nif, String nombre, String curso) {
        super( nif, nombre );
        this.curso = curso;
    }
    /////// Resto de métodos
}
```

Llamada al constructor con parámetros de Pesona. Se le envían los parámetros recibidos para que asigne los valores a los atributos nif y nombre

Ahora se pueden crear objetos de tipo Alumno y asignarles valores iniciales. Por ejemplo:

```
Alumno a = new Alumno("12345678-Z","Eliseo Gonzáles Manzano","1DAW");
```

4. CLASES FINALES

Si queremos evitar que una clase tenga clases derivadas debe declararse con el modificador **final** delante de class:

```
public final class A{
        ........
}
```

Esto la convierte en clase final. Una clase final no se puede heredar. Si intentamos crear una clase derivada de A se producirá un error de compilación.

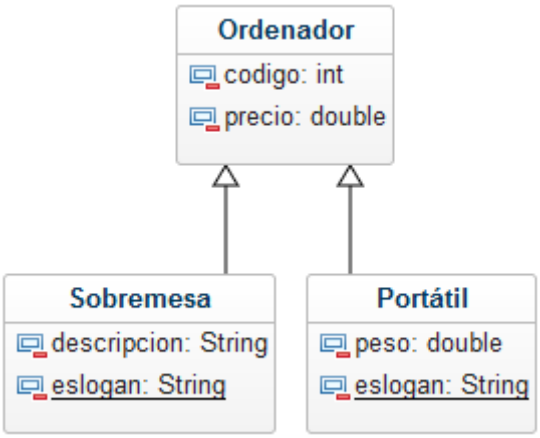
5. EJEMPLO DE HERENCIA

En una tienda se venden 2 tipos de ordenadores: portátiles y de sobremesa. Ambos tipos de ordenadores tienen los atributos *código* de tipo entero y *precio* de tipo double.

Cada tipo de ordenador tiene un *eslogan* que es: "Ideal para sus viajes" en el caso de los portátiles y "Es el que más pesa, pero el que menos cuesta" para el caso de los ordenadores de sobremesa. Este eslogan se guarda en un atributo static constante de tipo String. Al ser constante no se podrá modificar durante la ejecución del programa.

Además los ordenadores portátiles tienen un atributo *peso* de tipo double, y los de sobremesa un atributo *descripción* de tipo String que contiene la descripción del equipo (CPU, RAM, etc).

El diagrama de clases es:



Clase Ordenador:

```
public class Ordenador {
    private int codigo;
    private double precio;

    //Constructor por defecto
    public Ordenador() {
    }

    //Constructor con parámetros
    public Ordenador(int codigo, double precio) {
        this.codigo = codigo;
        this.precio = precio;
    }

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }
```

```
    public double getPrecio() {
        return precio;
    }
    public void setPrecio(double precio) {
        this.precio = precio;
    }

    public void leer() {
        Scanner sc = new Scanner(System.in);
        System.out.print("Código: ");
        codigo = sc.nextInt();
        System.out.print("Precio: ");
        precio = sc.nextDouble();
    }

    @Override
    public String toString() {
        return "codigo: " + codigo + "\nPrecio: " + precio;
    }
}
```

Clase Sobremesa:

```
public class Sobremesa extends Ordenador {

    //atributo estático constante. Se inicializa en la declaración
    //no confundir static con constante. Son cosas distintas
    //Al declararlo como constante (final) el valor inicial que se le asigna
    //no se puede modificar.
    private final static String eslogan = "Es el que más pesa, pero el que menos cuesta";
    private String descripcion;

    //Constructor por defecto
    public Sobremesa() {
    }

    //Constructor con parámetros
    public Sobremesa(int codigo, double precio, String descripcion) {
        super(codigo, precio);
        this.descripcion = descripcion;
    }

    public String getDescripcion() {
        return descripcion;
    }
    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

    //sobreescritura del método leer heredado de Ordenador
    @Override
    public void leer() {
        Scanner sc = new Scanner(System.in);
        super.leer();
        System.out.print("Descripción: ");
        descripcion = sc.nextLine();
    }
```

```
    //sobreescritura del método toString heredado de Ordenador
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Ordenador de Sobremesa\n");
        sb.append(super.toString());
        sb.append("\nDescripción: ");
        sb.append(descripcion);
        return sb.toString();
    }
}
```

Clase Portátil:

```
public class Portatil extends Ordenador {
    private double peso;
    //atributo estático constante. Se inicializa en la declaración
    private final static String eslogan = "Ideal para sus viajes";

    //Constructor por defecto
    public Portatil() {
    }
    //Constructor con parámetros
    public Portatil(int codigo, double precio, double peso) {
        super(codigo, precio);
        this.peso = peso;
    }

    public double getPeso() {
        return peso;
    }

    public void setPeso(double peso) {
        this.peso = peso;
    }
    //sobreescritura del método leer heredado de Ordenador
    @Override
    public void leer() {
        Scanner sc = new Scanner(System.in);
        super.leer();
        System.out.print("Peso: ");
        peso = sc.nextDouble();
    }
    //sobreescritura del método toString heredado de Ordenador
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Ordenador portátil\n");
        sb.append(super.toString());
        sb.append("\nPeso: ");
        sb.append(peso);
        return sb.toString();
    }
}
```

Clase principal:

En la clase principal vamos a escribir un programa que introduzca por teclado datos de portátiles y ordenadores de sobremesa y los guarde en dos ArrayList llamados *ordenadoresPortatiles* y *ordenadoresSobremesa*.

A continuación se mostrarán todos los ordenadores leídos, el ordenador de sobremesa de mayor precio y el portátil que menos pesa.

```
public class Ejemplo {
    static ArrayList<Sobremesa> ordenadoresSobremesa = new ArrayList<>();
    static ArrayList<Portatil> ordenadoresPortatiles = new ArrayList<>();
    static Scanner sc = new Scanner(System.in);

    public static void main(String[] args) {
        leer();
        mostrar();
        mayorPrecio();
        menorPeso();
    }

    public static void leer() {
        int tipo;
        do {
            do {
             System.out.print("Tipo de ordenador 1->Sobremesa 2->Portátil 0->FIN: ");
             tipo = sc.nextInt();
            } while (tipo < 0 || tipo > 2);
            switch (tipo) {
                case 1:
                    leerSobremesa();
                    break;
                case 2:
                    leerPortatil();
                    break;
            }
        } while (tipo != 0);
    }


    public static void leerSobremesa() {
        Sobremesa s = new Sobremesa();
        System.out.println("Introduzca los datos del ordenador de sobremesa");
        s.leer();
        ordenadoresSobremesa.add(s);
    }


    public static void leerPortatil() {
        Portatil p = new Portatil();
        System.out.println("Introduzca los datos del ordenador portátil");
        p.leer();
        ordenadoresPortatiles.add(p);
    }
```

```
    public static void mostrar() {
        if (!ordenadoresSobremesa.isEmpty()) {
            for (Sobremesa s : ordenadoresSobremesa) {
                System.out.println(s);
            }
        } else {
            System.out.println("No hay ordenadores de sobremesa");
        }
        if (!ordenadoresPortatiles.isEmpty()) {
            for (Portatil p : ordenadoresPortatiles) {
                System.out.println(p);
            }
        } else {
            System.out.println("No hay ordenadores portátiles");
        }
    }

    public static void mayorPrecio() {
        if (!ordenadoresSobremesa.isEmpty() ) {
            System.out.println("Ordenador de sobremesa con mayor precio");
            Sobremesa mayor = ordenadoresSobremesa.get(0);
            for (int i = 1; i < ordenadoresSobremesa.size(); i++) {
                if (ordenadoresSobremesa.get(i).getPrecio() > mayor.getPrecio()) {
                    mayor = ordenadoresSobremesa.get(i);
                }
            }
            System.out.println(mayor);
        }
    }


    public static void menorPeso() {
        if (!ordenadoresPortatiles.isEmpty()) {
            System.out.println("Ordenador portátil de menor peso");
            Portatil menor = ordenadoresPortatiles.get(0);
            for (int i = 1; i < ordenadoresPortatiles.size(); i++) {
                if (ordenadoresPortatiles.get(i).getPeso() < menor.getPeso()) {
                    menor = ordenadoresPortatiles.get(i);
                }
            }
            System.out.println(menor);
        }
    }
}
```

6. MÉTODOS ABSTRACTOS

Un método abstracto es un método declarado pero no implementado, es decir, es un método del que solo se escribe su nombre, parámetros y tipo devuelto pero no su código.

La forma general de declarar un método abstracto es:

```
[modificador] abstract tipoDevuelto nombreMetodo([parámetros]);
```

Los métodos abstractos se escriben sin llaves {} y con ; al final de la declaración.

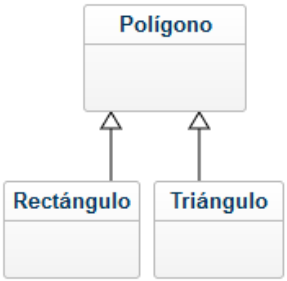
Por ejemplo:

```
public abstract double area();
```

En este ejemplo se ha declarado como abstracto un método llamado área. Se indica que es abstracto mediante la palabra *abstract*. Además el método no tiene bloque de instrucciones.

Un método se declara como abstracto porque en ese momento (en esa clase donde se declara) no se conoce aún cómo va a ser su implementación.

Por ejemplo: Tenemos una clase Polígono de la que se han derivado las clases Rectángulo y Triángulo.



Ambas clases derivadas van a usar un método para calcular el área del rectángulo y del triángulo. Podemos declarar el método *área* en la clase Polígono como abstracto y dejar que cada clase lo herede y lo implemente según sus necesidades, es decir, en la clase Rectángulo el método área se escribirá usando la fórmula del área del rectángulo y en la clase Triángulo el método área usará el área del triángulo.

Al incluir el método abstracto en la clase base se obliga a que todas las clases derivadas lo sobrescriban con el mismo formato utilizado en la declaración.

Si una clase contiene un método abstracto se convierte en clase abstracta y debe ser declarada como tal.

7. CLASES ABSTRACTAS

Es una clase que NO se puede instanciar (no se pueden crear objetos de la clase).

Las clases abstractas se diseñan solo para que **otras clases hereden de ella**.

La clase abstracta normalmente es la raíz de una jerarquía de clases y contendrá el comportamiento general que deben tener todas las subclases. En las clases derivadas se detalla la implementación.

Pueden contener cero o más métodos abstractos.

Pueden contener métodos no abstractos.

Pueden contener atributos.

Todas las subclases que hereden de una clase abstracta deben implementar todos los métodos abstractos heredados.

Si una clase derivada de una clase abstracta no implementa algún método abstracto se convierte a su vez en abstracta y tendrá que declararse como tal.

Aunque no se pueden crear objetos, las clases abstractas pueden tener constructores para inicializar sus atributos que serán invocados cuando se creen objetos de clases derivadas.

La forma general de declarar una clase abstracta es:

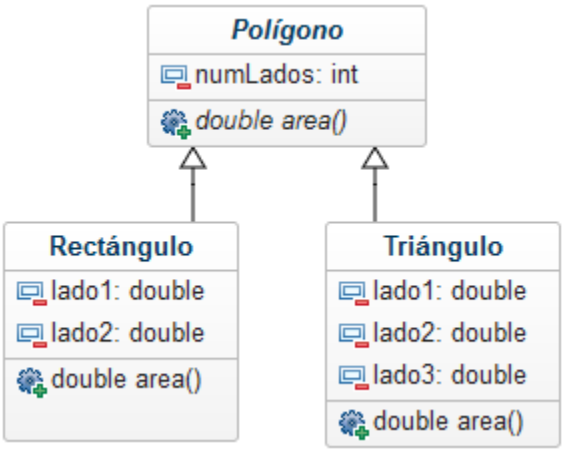
```
[modificador] abstract class nombreClase{
    . . . .
}
```

Ejemplo: Creamos una clase llamada Polígono que contiene el atributo numLados de tipo int. Además la clase contiene un método abstracto area().

Polígono es una clase abstracta y debe declararse como tal ya que contiene un método abstracto.

De la clase Polígono heredan las clases Rectángulo y Triángulo.

Ambas heredan el método abstracto area() y deberán implementarlo, de lo contrario deberán declararse como abstractas.



En UML las clases abstractas y métodos abstractos se escriben con su nombre en cursiva.

```
//Clase Poligono
public abstract class Poligono {

    private int numLados;

    //Constructor por defecto
    public Poligono() {
    }

    //Constructor con paránmetros
    public Poligono(int numLados) {
        this.numLados = numLados;
    }

    public int getNumLados() {
        return numLados;
    }

    public void setNumLados(int numLados) {
        this.numLados = numLados;
    }

    public abstract double area();
}
```

Declaración del método abstracto area()

```
//Clase Rectangulo
public class Rectangulo extends Poligono{

    private double lado1;

    private double lado2;

    //Constructor por defecto
    public Rectangulo() {
    }

    //Constructor con parámetros
    public Rectangulo(double lado1, double lado2) {
        super(2);
        this.lado1 = lado1;
        this.lado2 = lado2;
    }

    public double getLado1() {
        return lado1;
    }

    public void setLado1(double lado1) {
        this.lado1 = lado1;
    }

    public double getLado2() {
        return lado2;
    }

    public void setLado2(double lado2) {
        this.lado2 = lado2;
    }

    @Override
    public double area(){
        return lado1 * lado2;
    }
}
```

Implementación del método abstracto area() heredado de Poligono

```
//Clase Triangulo
public class Triangulo extends Poligono{


    private double lado1;

    private double lado2;

    private double lado3;

    //Constructor por defecto
    public Triangulo() {
    }

    //Constructor con parámetros
    public Triangulo(double lado1, double lado2, double lado3) {
        super(3);
        this.lado1 = lado1;
        this.lado2 = lado2;
        this.lado3 = lado3;
    }

    public double getLado1() {
        return lado1;
    }

    public void setLado1(double lado1) {
        this.lado1 = lado1;
    }
```

```
    public double getLado2() {
        return lado2;
    }
    public void setLado2(double lado2) {
        this.lado2 = lado2;
    }

    public double getLado3() {
        return lado3;
    }

    public void setLado3(double lado3) {
        this.lado3 = lado3;
    }


    @Override
    public double area(){
        double p = (lado1+lado2+lado3)/2;
        return Math.sqrt(p * (p-lado1) * (p-lado2) * (p-lado3));
    }
}
```

Implementación del método abstracto area() heredado de Poligono

Ejemplo básico para probar las clases:

```
public static void main(String[] args) {

        Triangulo t = new Triangulo(3.25,4.55,2.71);

        System.out.printf("Área del triángulo: %.2f %n" , t.area());

        Rectangulo r = new Rectangulo(5.70,2.29);

        System.out.printf("Área del rectángulo: %.2f %n" , r.area());

}
```

8. CONVERSIONES ENTRE CLASES

CONVERSIONES IMPLÍCITAS: UPCASTING

La herencia establece una relación es-un entre clases. Esto quiere decir que un objeto de una clase derivada es también un objeto de la clase base.

Por esta razón:

Se puede asignar de forma implícita la referencia de un objeto de una clase derivada a una referencia de la clase base. Son **tipos compatibles**.

También se llaman conversiones ascendentes o **upcasting**.

En el ejemplo anterior un Triángulo en un Poligono y un Cuadrado también es un Poligono.

Si declaramos una variable *p* de tipo Poligono y creamos un objeto Triángulo:

```
Poligono p;
Triangulo t = new Triangulo(3,5,2);
```

Podemos hacer esta asignación:

```
p = t;
```

La variable p de tipo Poligono puede contener la referencia de un objeto Triangulo ya que son tipos compatibles.

Cuando se realizan este tipo de asignaciones y manejamos un objeto a través de una referencia a una superclase (directa o indirecta) solo se pueden ejecutar métodos disponibles en la superclase.

Siguiendo con el ejemplo, después de la asignación

```
p = t
```

la variable p contiene la referencia de un triángulo. Si intentamos acceder al valor de uno de sus lados:

```
p.getLado1();
```

esta instrucción provocará un error ya que p es de tipo Poligono y el método getLado1() no es un método de la clase Poligono.

Cuando manejamos un objeto a través de una referencia a una superclase (directa o indirecta) y se invoca a un método de la superclase que está redefinido en las subclases se ejecuta el método de la clase a la que pertenece el objeto y no el de la superclase.

En el ejemplo, la instrucción

```
p.area();
```

ejecutará el método area() de Triángulo porque p contiene la referencia a un objeto de la clase Triangulo y el método area() está redefinido en la clase Triangulo.

CONVERSIONES EXPLÍCITAS: DOWNCASTING

Se puede asignar una referencia de la clase base a una referencia de la clase derivada, siempre que la referencia de la clase base pertenezca a un objeto de la misma clase derivada a la que se va a asignar o de una clase derivada de ésta.

También se llaman conversiones descendentes o **downcasting**.

Esta conversión debe hacerse de forma explícita mediante un **casting**.

Siguiendo con el ejemplo anterior:

Ya sabemos que podemos asignar de forma implícita un objeto de una clase derivada a una referencia de la clase base, por ejemplo:

```
Poligono p = new Triangulo(1,3,2);
```

Pero si queremos hacer la asignación contraria:

```
Triangulo t = (Triangulo) p;
```

Debemos hacerlo de forma implícita indicándolo mediante un casting: `(Triangulo) p` Con esto estamos indicando que el tipo de la variable p para esta asignación lo estamos convirtiendo (casting) en un Triangulo.

Este casting y asignación se puede hacer porque p contiene la referencia de un objeto Triángulo.

Si p no contiene la referencia de un triángulo provocará un error de ejecución del tipo **ClassCastException**.

Las siguientes instrucciones provocan ese error:

```
Poligono p1 = new Rectangulo(3,2);
Triangulo t1 = (Triangulo)p1;    //----> Error de ejecución ClassCastException
```

p1 contiene la referencia a un objeto Rectangulo y no se puede convertir en una referencia a un objeto Triangulo. No son tipos compatibles.

EL OPERADOR instanceof

Las conversiones entre clases requieren que ambas sean de tipos compatibles.

Para asegurarnos de que podemos realizar la conversión (casting) de un objeto de una clase en otra podemos utilizar el operador *instanceof*.

instanceof devuelve true si el objeto es instancia de la clase y false en caso contrario.

La sintaxis es:

```
Objeto instanceof Clase
```

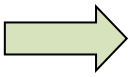
Ejemplo:

```
Triangulo t;
Poligono p1 = new Rectangulo(3,2);
if(p1 instanceof Triangulo){
   t = (Triangulo) p1; //si p1 es una referencia a un Triángulo se puede realizar la conversión
}else{
   System.out.println("Objetos incompatibles");
}
```

En este caso como p1 no es un Triangulo porque contiene la referencia de un Rectángulo se mostrará el mensaje *Objetos incompatibles*

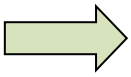
9. POLIMORFISMO

El polimorfismo es otra de las características fundamentales de la POO y está estrechamente relacionado con la herencia.



La herencia implica polimorfismo.

Una jerarquía de clases, los métodos y clases abstractas, la sobrescritura de métodos y las conversiones entre clases de la jerarquía sientan las bases para el polimorfismo.



Polimorfismo es la cualidad que tienen los objetos para responder de distinto modo a un mismo mensaje.

Para conseguir un comportamiento polimórfico en un programa Java:

- Los métodos deben estar declarados (y a veces también pueden estar implementados) en la clase base.
- Los métodos debes estar redefinidos en las clases derivadas.
- Los objetos deben ser manipulados utilizando referencias a la clase base.

De esta forma, cuando se invoque a un método se ejecutará la versión del método correspondiente a la clase del objeto referenciado y no al de la clase de la variable que lo referencia.

Ejemplo: A partir de las clases Polígono, Rectángulo y Triángulo que hemos creado anteriormente, a las que les vamos a añadir el método toString(), vamos a escribir un programa para crear objetos de tipo Triángulo y Rectángulo y guardarlos en un ArrayList. En lugar de tener dos ArrayList para almacenarlos, los guardaremos todos juntos utilizando un ArrayList de Polígonos. Después de leerlos se mostrarán los datos de los objetos y su área.

En este ejemplo se utiliza el polimorfismo ya que:

- Los métodos toString() y area() están declarados en la clase base Polígono. El método toString está además implementado en la clase Polígono.
- Estos métodos están redefinidos en las dos clases derivadas.
- Los métodos se invocan mediante referencias a la clase base Polígono ya que el ArrayList que vamos a crear es de este tipo.

```
//Clase Poligono
public abstract class Poligono {
    private int numLados;
    public Poligono() {
    }
    public Poligono(int numLados) {
        this.numLados = numLados;
    }
    public int getNumLados() {
        return numLados;
    }
    public void setNumLados(int numLados) {
        this.numLados = numLados;
    }
    @Override
    public String toString(){
        return "Numero de lados: " + numLados;
    }
    public abstract double area();
}
//Clase Rectangulo
public class Rectangulo extends Poligono{
    private double lado1;
    private double lado2;
    public Rectangulo() {
    }
    public Rectangulo(double lado1, double lado2) {
        super(2);
        this.lado1 = lado1;
        this.lado2 = lado2;
    }
    public double getLado1() {
        return lado1;
    }
    public void setLado1(double lado1) {
        this.lado1 = lado1;
    }
    public double getLado2() {
        return lado2;
    }
    public void setLado2(double lado2) {
        this.lado2 = lado2;
    }
```

Sobreescritura del método toString() heredado de Object

Declaración del método abstracto area()

```
    @Override
    public String toString() {
        return "Rectangulo{ " + "lado 1 = " + lado1 + ", lado 2 = " + lado2 + '}';
    }
    @Override
    public double area(){
        return lado1 * lado2;
    }
}
```

Sobreescritura del método toString() heredado de Poligono

Implementación del método abstracto area() heredado de Poligono

```
//Clase Triangulo
public class Triangulo extends Poligono{
    private double lado1;
    private double lado2;
    private double lado3;

    public Triangulo() {
    }

    public Triangulo(double lado1, double lado2, double lado3) {
        super(3);
        this.lado1 = lado1;
        this.lado2 = lado2;
        this.lado3 = lado3;
    }

    public double getLado1() {
        return lado1;
    }

    public void setLado1(double lado1) {
        this.lado1 = lado1;
    }

    public double getLado2() {
        return lado2;
    }

    public void setLado2(double lado2) {
        this.lado2 = lado2;
    }

    public double getLado3() {
        return lado3;
    }

    public void setLado3(double lado3) {
        this.lado3 = lado3;
    }

    @Override
    public String toString() {
        return "Triangulo{ " + "lado 1 = " + lado1 +
                             ", lado 2 = " + lado2 + ", lado 3 = " + lado3 + '}';
    }

    @Override
    public double area(){
        double p = (lado1+lado2+lado3)/2;
        return Math.sqrt(p * (p-lado1) * (p-lado2) * (p-lado3));
    }
}
```

Sobreescritura del método toString() heredado de Poligono

Implementación del método abstracto area() heredado de Poligono

```
//Clase Principal
public class Polimorfismo1 {

    static ArrayList<Poligono> poligonos = new ArrayList<>();
    static Scanner sc = new Scanner(System.in);

    public static void main(String[] args) {
        leerPoligonos();
        mostrarPoligonos();
    }

    public static void leerPoligonos() {
        int tipo;
        do {
            do {
              System.out.print("Tipo de poligono 1->Rectangulo 2->Triangulo 0-> FIN >>> ");
               tipo = sc.nextInt();
            } while (tipo < 0 || tipo > 2);
            if (tipo != 0) {
                switch (tipo) {
                    case 1:
                        leerRectangulo();
                        break;
                    case 2:
                        leerTriangulo();
                        break;
                }
            }
        } while (tipo != 0);
    }

    public static void leerRectangulo() {
        double l1, l2;
        System.out.println("Introduzca datos del Rectángulo");
        do {
            System.out.print("Longitud del lado 1: ");
            l1 = sc.nextDouble();
        } while (l1 <= 0);
        do {
            System.out.print("Longitud del lado 2: ");
            l2 = sc.nextDouble();
        } while (l2 <= 0);
        Rectangulo r = new Rectangulo(l1, l2);
        poligonos.add(r);
    }
```

ArrayList de referencias a objetos de la clase base Poligono

Conversión implícita.
Se asigna una referencia de una clase derivada (Rectangulo) a una referencia de la clase base (Poligono), es decir, estamos asignando a un elemento del ArrayList que es del tipo Poligono una variable que es de tipo Rectángulo (clase derivada de Polígono)

```
    public static void leerTriangulo() {
        double l1, l2, l3;
        System.out.println("Introduzca datos del Triangulo");
        do {
            System.out.print("Longitud del lado 1: ");
            l1 = sc.nextDouble();
        } while (l1 <= 0);
        do {
            System.out.print("Longitud del lado 2: ");
            l2 = sc.nextDouble();
        } while (l2 <= 0);
        do {
            System.out.print("Longitud del lado 3: ");
            l3 = sc.nextDouble();
        } while (l3 <= 0);
        Triangulo t = new Triangulo(l1, l2, l3);
        poligonos.add(t);
    }
```

Conversión implícita.
Se asigna una referencia de una clase derivada (Triangulo) a una referencia de la clase base (Poligono), es decir, estamos asignando a un elemento del ArrayList que es del tipo Poligono una variable que es de tipo Triangulo (clase derivada de Polígono)

Se recorre el array *poligonos* que contiene referencias a Triangulos y Rectangulos.
A p se le asignarán mediante upcasting referencias a objetos de tipo Triangulo o Rectangulo

```
    public static void mostrarPoligonos() {
        for(Poligono p: poligonos){
            System.out.print(p.toString());
            System.out.printf(" area: %.2f %n", p.area());
        }
    }
} //fin de la clase principal
```

Polimorfismo
Mediante una referencia (p) a la clase base, se ejecutarán los métodos de la clase derivada a la que pertenece el objeto referenciado

ENLAZADO DINÁMICO

Hemos visto que cuando se invoca un método mediante una referencia a la clase base de la jerarquía, se ejecuta la versión del método correspondiente a la clase del objeto referenciado y no al de la clase de la variable que lo referencia.

El método que se ejecuta se decide por tanto **durante la ejecución del programa.**

A este proceso de decidir en tiempo de ejecución qué método se ejecuta se le denomina **vinculación dinámica o enlazado dinámico.**

El enlazado dinámico es el mecanismo que **hace posible el polimorfismo**.

El enlazado dinámico es lo opuesto a la habitual vinculación estática o enlazado estático que consiste en decidir en tiempo de compilación qué método se ejecuta en cada caso.

En el programa anterior:

```
public void mostrarPoligonos() {
        for(Poligono p: poligonos){
            System.out.print(p.toString());
            System.out.printf(" area: %.2f %n", p.area());
        }
}
```

Para que el polimorfismo se lleve a cabo se está aplicando el enlazado dinámico.
Durante la ejecución se decide qué método se ejecuta.

```
public static void main(String[] args) {
        leerPoligonos();
        mostrarPoligonos();
}
```

Enlazado estático. Cuando se compila el programa ya se sabe que serán esos métodos los que se van a ejecutar.