Tema 7

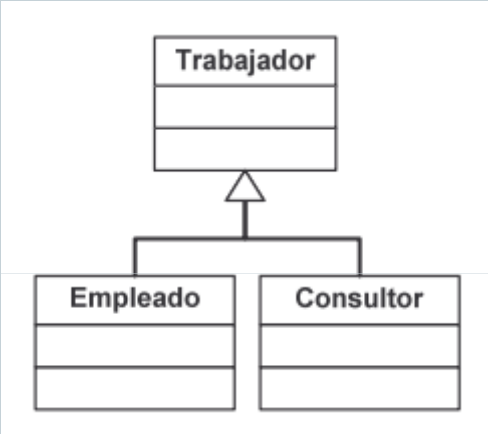
UTILIZACIÓN AVANZADA DE CLASES: HERENCIA Y POLIMORFISMO

1. HERENCIA

- La **herencia** es uno de los tres pilares básicos de la programación orientada a objetos.
- Permite la creación de clasificaciones jerárquicas.
- La herencia es la base de la reutilización del código.
- Cuando una clase deriva de una clase padre, ésta hereda todos los miembros y métodos de su antecesor.
- También es posible redefinir (***override***) los miembros para adaptarlos a la nueva clase o bien ampliarlos.
- En general, todas las subclases no solo adoptan las variables y comportamiento de las superclases sino que los amplían.

1. HERENCIA

- Un ejemplo de herencia sería:



- **Trabajador** es una clase genérica que sirve para almacenar datos como el nombre, la dirección, el número de teléfono o el número de la seguridad social de un trabajador.

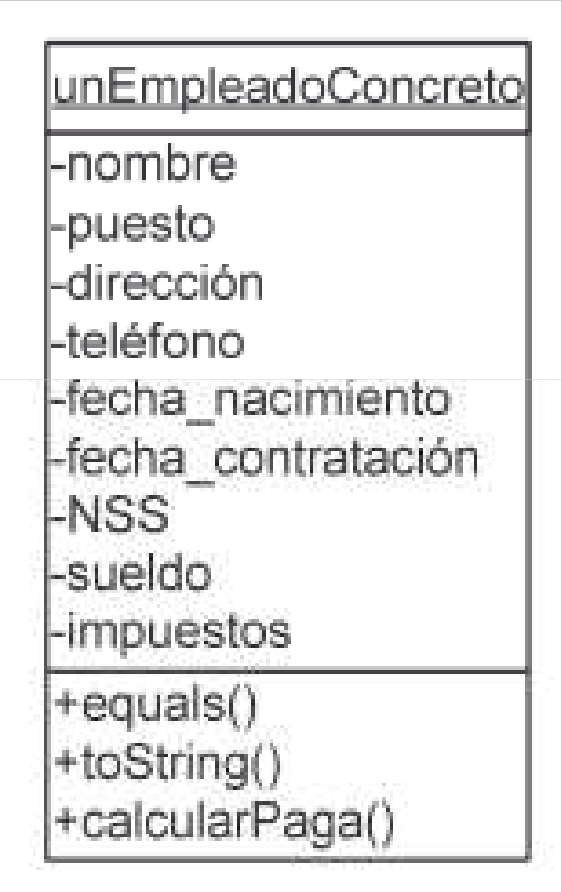
1. HERENCIA

- **Empleado** es una clase especializada para representar los empleados que tienen una nómina mensual (encapsula datos como su salario anual o las retenciones del IRPF).
- **Consultor** es una clase especializada para representar a aquellos trabajadores que cobran por horas (por ejemplo, registra el número de horas que ha trabajado un consultor y su tarifa horaria).

1. HERENCIA

- Las clases **Empleado** y **Consultor** además de los atributos y de las operaciones que definen, heredan de **Trabajador** todos sus atributos y operaciones.

1. HERENCIA



- Un empleado concreto tendrá, además de sus atributos y operaciones como **Empleado**, todos los atributos correspondientes a la superclase **Trabajador**.

1. HERENCIA

- Para indicar que una clase hereda de otra se etiqueta con la cláusula **extends** detrás del nombre de la clase.

```
public class Trabajador
{
  private String nombre;
  private String puesto;
  private String direccion;
  private String telefono;
  private Date   fecha_nacimiento;
  private Date   fecha_contrato;
  private String NSS;

  // Constructor

  public Trabajador (String nombre, String NSS)
  {
    this.nombre = nombre;
    this.NSS = NSS;
  }

  // Métodos get & set

  // Comparación de objetos

  public boolean equals (Trabajador t)
  {
    return this.NSS.equals(t.NSS);
  }

  // Conversión en una cadena de caracteres

  public String toString ()
  {
    return nombre + " (NSS "+NSS+")";
  }
}
```

1. HERENCIA

```
public class Empleado extends Trabajador
{
  private double sueldo;
  private double impuestos;

  private final int PAGAS = 14;

  // Constructor

  public Empleado
        (String nombre, String NSS, double sueldo)
  {
    super(nombre,NSS);

    this.sueldo    = sueldo;
    this.impuestos = 0.3 * sueldo;
  }

  // Nómina

  public double calcularPaga ()
  {
    return (sueldo-impuestos)/PAGAS;
  }

 // toString

 public String toString ()
 {
   return "Empleado "+super.toString();
 }
}
```

Con la palabra reservada **extends** indicamos que Empleado es una subclase de Trabajador.

Con la palabra reservada **super** accedemos a miembros de la superclase desde la subclase.
Generalmente, en un constructor, lo primero que nos encontramos es una llamada al constructor de la clase padre con **super (...)**. Si no ponemos nada, se llama al constructor por defecto de la superclase antes de ejecutar el constructor de la subclase.

1. HERENCIA

```
class Consultor extends Trabajador
{
  private int    horas;
  private double tarifa;

  // Constructor

  public Consultor (String nombre, String NSS,
                    int horas, double tarifa)
  {
    super(nombre,NSS);

    this.horas  = horas;
    this.tarifa = tarifa;
  }

  // Paga por horas

  public double calcularPaga ()
  {
    return horas*tarifa;
  }

 // toString

 public String toString ()
 {
   return "Consultor "+super.toString();
 }
}
```

La clase **Consultor** también define un método llamado **calcularPaga()**, si bien en este caso el cálculo se hace de una forma diferente por tratarse de un trabajador de un tipo distinto.

1. HERENCIA

- Utilización de las clases:

```
// Declaración de variables

Trabajador trabajador;
Empleado   empleado;
Consultor  consultor;

// Creación de objetos

trabajador = new Trabajador ("Juan", "456");

empleado = new Empleado  ("Jose", "123", 24000.0);

consultor = new Consultor ("Juan", "456", 10, 50.0);


// Salida estándar con toString()

System.out.println(trabajador);
```

```
Juan (NSS 456)
```

```
System.out.println(empleado);
```

```
Empleado Jose (NSS 123)
```

```
System.out.println(consultor);
```

```
Consultor Juan (NSS 456)
```

1. HERENCIA

- Utilización de las clases:

```
// Comparación de objetos con equals()
System.out.println(trabajador.equals(empleado));
```

```
false
```

```
System.out.println(trabajador.equals(consultor));
```

```
true
```

1. HERENCIA

- Otro ejemplo:

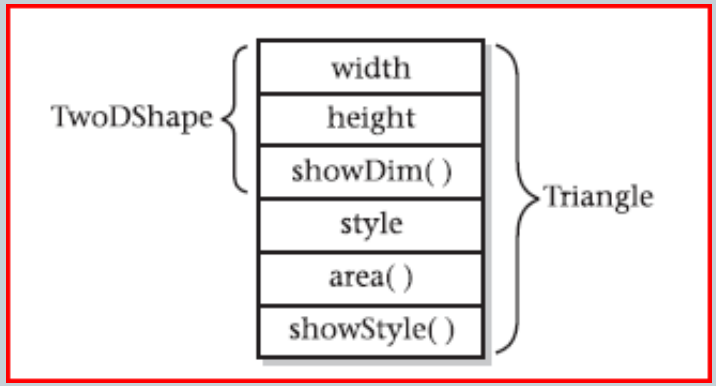
```
public class TwoDShape {

    double width;
    double height;

    void showDim() {
        System.out.println("Width and height are " + width + " and " + height);
    }
}
```

```
class Triangle extends TwoDShape {

    String style;
    double area() {
        return width * height / 2;
    }
    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}
```



1. HERENCIA

- Ejemplo de uso:

```
public class Shapes {
    public static void main(String args[]) {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle();
        t1.width = 4.0;
        t1.height = 4.0;
        t1.style = "filled";
        t2.width = 8.0;
        t2.height = 12.0;
        t2.style = "outlined";
        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());
        System.out.println();
        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());
    }
}
```

1.1 Acceso de miembros y herencia

- Como ya sabemos, es común declarar un atributo de una clase como **private** para evitar usos no autorizados.
- La herencia no anula la restricción de acceso privado.
- Por lo tanto, aunque una subclase incluya todos los miembros de su superclase, no puede acceder a estos miembros de la superclase que han sido declarados **private**.

1.1 Acceso de miembros y herencia

- Ejemplo:

```
public class AccesoTwoDShape {

    private double width; // these are
    private double height; // now private

    void showDim() {
        System.out.println("Width and height are "
                + width + " and " + height);
    }
}

class AccesoTriangle extends AccesoTwoDShape {

    String style;

    double area() {
        return width * height / 2; // Error! can't access
    }

    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}
```

1.1 Acceso de miembros y herencia

- La solución estaría en utilizar métodos para poder acceder a estas variables:

```
class AccesoTwoDShape2 {

    private double width; // these are
    private double height; // now private

    double getWidth() {
        return width;
    }

    double getHeight() {
        return height;
    }
    void setWidth(double w) {
        width = w;
    }

    void setHeight(double h) {
        height = h;
    }

    void showDim() {
        System.out.println("Width and height are " + width + " and " + height);
    }
}
```

1.1 Acceso de miembros y herencia

- La solución estaría en utilizar métodos para poder acceder a estas variables:

```
class AccesoTriangle2 extends AccesoTwoDShape2 {

    String style;

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}
```

1.2 Constructores y Herencia

- En una jerarquía, es posible que tanto la superclase como las subclases tengan sus propios constructores.
- El constructor de la superclase construye la porción de la superclase del objeto y el constructor de la subclase construye la parte de la subclase.

1.2 Constructores y Herencia

- Ejemplo 1: **Superclase sin constructor.**

```
class TwoDShape {
    private double width; // these are
    private double height; // now private
    double getWidth() {
        return width;     }

    double getHeight() {
        return height;     }

    void setWidth(double w) {
        width = w;     }

    void setHeight(double h) {
        height = h;     }

    void showDim() {
        System.out.println("Width and height are " + width + " and " + height);     }

}
```

1.2 Constructores y Herencia

- Ejemplo 1: **Superclase sin constructor.**

```
class Triangle extends TwoDShape {
    private String style;
    Triangle(String s, double w, double h) { //Constructo
        setWidth(w);
        setHeight(h);
        style = s;
    }

    double area() {
        return getWidth() * getHeight() / 2;     }

    void showStyle() {
        System.out.println("Triangle is " + style);     }
}
```

1.2.1 Usando **super**

- Una subclase puede llamar al constructor definido en la superclase usando la palabra reservada **super**, cuya sintaxis es:

  super(parameter-list);

- *parameter-list* especifica los parámetros necesitados por el constructor de la superclase.

- ***super()*** debe ser siempre la primera sentencia ejecutada dentro del constructor de la subclase.

1.2.1 Usando **super**

- Ejemplo:

```
class TwoDShape {
    private double width;
    private double height;

    TwoDShape(double w, double h) {
        width = w;
        height = h;
    }
    double getWidth() {
        return width;     }
    double getHeight() {
        return height;     }
    void setWidth(double w) {
        width = w;     }
    void setHeight(double h) {
        height = h;     }
    void showDim() {
        System.out.println("Width and height are " + width + " and " + height);
    }
}
```

1.2.1 Usando **super**

- Ejemplo:

```
class Triangle extends TwoDShape {

    private String style;

    Triangle(String s, double w, double h) {
        super(w, h); // call superclass constructor
        style = s;
    }
    double area() {
        return getWidth() * getHeight() / 2;     }

    void showStyle() {
        System.out.println("Triangle is " + style);     }
}
```

1.2.2 Añadiendo más constructores

- Como ya sabemos, las clases pueden tener más de un constructor. Esto no cambia cuando se trabaja con la **herencia**.
- Ejemplo:

```
class TwoDShape {

    private double width;
    private double height;

    TwoDShape() {    //Primer constructor
        width = height = 0.0;
    }

    TwoDShape(double w, double h) { //Segundo constructor
        width = w;
        height = h;
    }

    TwoDShape(double x) {    //Tercer constructor
        width = height = x;
    }
}
```

1.2.2 Añadiendo más constructores

- Ejemplo:

```
class Triangle extends TwoDShape {

    private String style;

    Triangle() {
        super();
        style = "none";
    }

    Triangle(String s, double w, double h) {
        super(w, h);    // call superclass constructor
        style = s;
    }

    Triangle(double x) {
        super(x);   // call superclass constructor
        style = "filled";
    }
}
```

1.3 Otro uso de **super**

- Hay un segundo uso de **super** que actúa similar a **this** excepto que siempre se refiere a la superclase de la subclase en la que es usado.
- La sintaxis es:

  super.member

  donde **member** puede ser un método o un atributo.

1.3 Otro uso de **super**

- Ejemplo:

```
class A {
    int i;
}

class B extends A {
    int i;

    B(int a, int b) {
        super.i = a;
        i = b;
    }

    void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
    }
}

class UseSuper {

    public static void main(String args[]) {
        B subOb = new B(1, 2);
        subOb.show();
    }
}
```

1.4 Jerarquía multinivel

- Hasta este punto, hemos estado viendo una jerarquía simple donde una subclase heredaba de una superclase.
- Sin embargo, es posible construir jerarquías que contengan tantos niveles de herencia como quieras.
- Por lo tanto, sería posible usar una subclase como superclase de otra.

```
class ColorTriangle extends Triangle {
    private String color;

    ColorTriangle(String c, String s, double w, double h) {
        super(s, w, h);
        color = c;     }

    String getColor() {
        return color;     }

    void showColor() {
        System.out.println("Color is " + color);     }
}
```

1.5 Métodos Override

- En una jerarquía de clases, cuando un método en una subclase tiene el mismo encabezado y el mismo valor de retorno que un método de su superclase, se dice que el método de la subclase **sobreescribe** (override) el método de la superclase.
- Cuando un método sobreescrito es llamado desde una subclase, siempre estará referenciando al método de la subclase y el método definido en la superclase será obviado.

1.5 Métodos Override

- Ejemplo:

```
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    void show() {
    }
}
class B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    void show() {
        System.out.println("k: " + k);
    }
}
```

```
class B2 extends A {
    int k;
    B2(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    void show() {
        super.show(); // this calls A's show()
        System.out.println("k: " + k);
    }
}
class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show(); // this calls show() in B
    }
}
```

1.6 Métodos Overloaded

- Los métodos con diferentes encabezados son **sobrecargados** y **NO sobreescritos**:

```
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}
class B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    void show(String msg) {
        System.out.println(msg + k);
    }
}
```

```
class Overload {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show("This is k: "); // this calls show() in B
        subOb.show(); // this calls show() in A
    }
}
```

Actividad

- Realiza los ejercicios del 1 al 5 de la Hoja de Ejercicios

2. Polimorfismo

- Al redefinir métodos, objetos de diferentes tipos pueden responder de forma diferente a la misma llamada (y podemos escribir código de forma general sin preocuparnos del método concreto que se ejecutará en cada momento).
- Por ejemplo, podemos añadirle a la clase ***Trabajador*** un método ***calcularPaga()*** genérico (que no haga nada por ahora.)

```
public class Trabajador…

  public double calcularPaga ()
  {
    return 0.0;                    // Nada por defecto
  }
```

2. Polimorfismo

- En las subclases de Trabajador, sí que definimos el método calcularPaga() para que calcule el importe del pago que hay que efectuarle a un trabajador (en función de su tipo).

```
public class Empleado extends Trabajador…
  public double calcularPaga ()        // Nómina
  {
    return (sueldo-impuestos)/PAGAS;
  }


class Consultor extends Trabajador…
  public double calcularPaga ()        // Por horas
  {
    return horas*tarifa;
  }
```

2. Polimorfismo

- Como los consultores y los empleados son trabajadores, podemos crear un array de trabajadores con consultores y empleados:

```
...
Trabajador trabajadores[] = new Trabajador[2];

trabajadores[0] = new Empleado
                  ("Jose", "123", 24000.0);

trabajadores[1] = new Consultor
                  ("Juan", "456", 10, 50.0);

...
```

2. Polimorfismo

- Una vez que tenemos un vector con todos los trabajadores de una empresa, podríamos crear un programa que realizase los pagos correspondientes a cada trabajador de la siguiente forma:

```
…
public void pagar (Trabajador trabajadores[])
{
  int i;

  for (i=0; i<trabajadores.length; i++)
      realizarTransferencia ( trabajadores[i],
                  trabajadores[i].calcularPaga());
}
…
```

Para los trabajadores del vector anterior, se realizaría una transferencia de 1200 € para el empleado Jose y otra transferencia, esta vez de 500€, para el consultor Juan.

2. Polimorfismo

- Cada vez que se invoca el método calcularPaga(), se busca automáticamente el código que en cada momento se ha de ejecutar en función del tipo de trabajador **(enlace dinámico)**.
- No hay que confundir el polimorfismo con la sobrecarga de métodos (distintos métodos con el mismo nombre pero diferentes parámetros).

3. La palabra reservada **final**

- En Java, usando la palabra reservada final, podemos:
  - Evitar que un método se pueda redefinir en una subclase.

```
class Consultor extends Trabajador
{
   …
   public final double calcularPaga ()
   {
      return horas*tarifa;
   }
   …
}
```

Aunque creemos subclases de Consultor, el dinero que se le pague siempre será en función de las horas que trabaje y de su tarifa horaria (y eso no se puede cambiar).

3. La palabra reservada **final**

- En Java, usando la palabra reservada final, podemos:
  - Evitar que se puedan crear subclases de una clase dada:

```
public final class Circulo extends Figura
…

public final class Cuadrado extends Figura
…
```

Al usar ***final***, tanto *Circulo* como *Cuadrado* son ahora clases de las que no se pueden crear subclases.

*** En Java, ***final*** también se usa para definir constantes simbólicas.

Actividad

- Realiza los ejercicios del 6 al 8 de la Hoja de Ejercicios

4. La palabra reservada **abstract**

- Una **clase abstracta** es aquella clase de la que no se tiene intención de crear objetos, sino que únicamente sirve para unificar datos u operaciones de subclases.
- Supongamos un esquema de herencia que consta de la clase Profesor de la que heredan ProfesorInterino y ProfesorTitular.
- Es posible que todo profesor haya de ser o bien ProfesorInterino o bien ProfesorTitular, es decir, que no vayan a existir instancias de la clase Profesor.
- Entonces, ¿qué sentido tendría tener una clase Profesor?

4. La palabra reservada **abstract**

- El sentido está en que una superclase permite unificar campos y métodos de las subclases, evitando la repetición de código y unificando procesos.
- Sin embargo, si no se tiene intención de crear objetos la declararemos en Java de forma especial: como **clase abstracta**.
- La declaración de que una clase es abstracta se hace con la sintaxis

  public abstract class NombreDeLaClase { … }.
- Por ejemplo:

  public abstract class Profesor.

4. La palabra reservada **abstract**

- Sin embargo, sigue funcionando como superclase de forma similar a como lo haría una superclase “normal”.
- La diferencia principal radica en que no se pueden crear objetos de esta clase.
- Declarar una clase abstracta es distinto a tener una clase de la que no se crean objetos.
- En una clase abstracta, no existe la posibilidad. En una clase normal, existe la posibilidad de crearlos aunque no lo hagamos.
- El hecho de que no creemos instancias de una clase no es suficiente para que Java considere que una clase es abstracta.

4. La palabra reservada **abstract**

- Para lograr esto hemos de declarar explícitamente la clase como **abstracta** mediante la sintaxis que hemos indicado.
- Si una clase no se declara usando abstract se cataloga como "clase concreta".
- Una clase abstracta para Java es una clase de la que nunca se van a crear instancias: simplemente va a servir como superclase a otras clases.
- No se puede usar la palabra clave *new* aplicada a clases abstractas.
- A su vez, las clases abstractas suelen contener **métodos abstractos**: la situación es la misma.

4. La palabra reservada **abstract**

- Para que un método se considere abstracto ha de incluir en su signatura la palabra clave **abstract**.
- Además un método abstracto tiene estas peculiaridades:
  - **No tiene cuerpo** (llaves): sólo consta de signatura con paréntesis.
  - Su signatura **termina con un punto y coma.**
  - **Sólo puede existir dentro de una clase abstracta.** De esta forma se evita que haya métodos que no se puedan ejecutar dentro de clases concretas. Visto de otra manera, si una clase incluye un método abstracto, forzosamente la clase será una clase abstracta.
  - Los métodos abstractos **forzosamente habrán de estar sobreescritos en las subclases.** Si una subclase no implementa un método abstracto de la superclase tiene un método no ejecutable, lo que la fuerza a ser una subclase abstracta. Para que la subclase sea concreta habrá de implementar métodos sobreescritos para todos los métodos abstractos de sus superclases.

4. La palabra reservada **abstract**

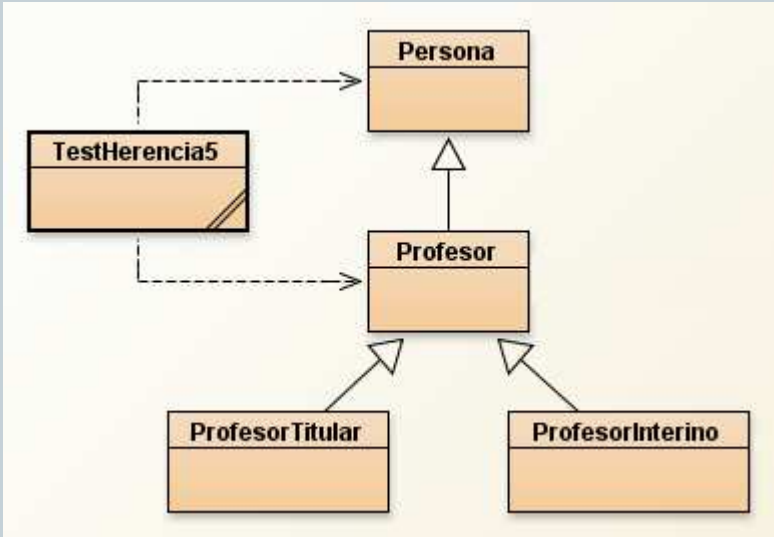
- Un **método abstracto** para Java es un método que nunca va a ser ejecutado porque no tiene cuerpo.
- Simplemente, un método abstracto referencia a otros métodos de las subclases.
- ¿Qué utilidad tiene un método abstracto? Podemos ver un método abstracto como una palanca que fuerza dos cosas:
  - La primera, que no se puedan crear objetos de una clase.
  - La segunda, que todas las subclases sobreescriban el método declarado como abstracto.
- **Sintaxis tipo:**

  abstract public/private/protected TipodeRetorno/void nombreMétodo(parámetros);

4. La palabra reservada **abstract**

- **Por ejemplo:**

  abstract public void generarNomina (int diasCotizados, boolean plusAntiguedad);
- Veamos un ejemplo basado en el siguiente esquema:



4. La palabra reservada **abstract**

- En este diagrama de clases vemos cómo hemos definido una clase abstracta denominada Profesor.
- Sin embargo, hereda de la clase Persona que no es abstracta, lo cual significa que puede haber instancias de Persona pero no de Profesor.
- El programa se basa en lo siguiente: ProfesorTitular y ProfesorInterino son subclases de la clase abstracta Profesor.
- **ListinProfesores** sirve para crear un Array de profesores que pueden ser tanto interinos como titulares.
- El listín se basa en el tipo estático Profesor, pero su contenido dinámico siempre será a base de instancias de ProfesorTitular o de ProfesorInterino ya que Profesor es una clase abstracta, no instanciable.

4. La palabra reservada **abstract**

- En la clase de test creamos profesores interinos y profesores titulares y los vamos añadiendo a un listín.
- Posteriormente, invocamos el método imprimirListin, que se basa en los métodos toString de las subclases y de sus superclases mediante invocaciones sucesivas a super.
- Por otro lado, en la clase ListinProfesores hemos definido el método importeTotalNominaProfesorado() que se basa en un bucle que calcula la nómina de todos los profesores que haya en el listín (sean interinos o titulares) mediante el uso de un método abstracto: importeNomina().
- Este método está definido como abstract public float importeNomina (); dentro de la clase abstracta profesor, e implementado en las clases ProfesorInterino y ProfesorTitular.

4. La palabra reservada **abstract**

- El aspecto central de este ejemplo es comprobar cómo una clase abstracta como Profesor nos permite realizar operaciones conjuntas sobre varias clases, **ahorrando código y ganando en claridad** para nuestros programas.
- El código se muestra en las siguientes diapositivas.

4. La palabra reservada **abstract**

- Clase **Persona**:

```
public class Persona {
    private String nombre;
    private String apellidos;
    private int edad;

    public Persona() {
        nombre = "";
        apellidos = "";
        edad = 0;
    }
    public Persona (String nombre, String apellidos, int edad) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
    }
    public String getNombre() { return nombre; }
    public String getApellidos() { return apellidos;  }
    public int getEdad() { return edad;  }

    public String toString() {
        Integer datoEdad = edad;

        return "-Nombre: ".concat(nombre).concat(" -Apellidos: ")
                .concat(apellidos).concat(" -Edad: ").concat(datoEdad.toString() ); }

} //Cierre de la clase
```

4. La palabra reservada **abstract**

- Clase **Profesor**:

```
public abstract class Profesor extends Persona {
    private String IdProfesor;

    public Profesor () {
        super();
        IdProfesor = "Unknown";
    }
    public Profesor (String nombre, String apellidos, int edad, String id) {
        super(nombre, apellidos, edad);
        IdProfesor = id;
    }
    public void setIdProfesor (String IdProfesor) { this.IdProfesor = IdProfesor;}
    public String getIdProfesor () { return IdProfesor;   }
    public void mostrarDatos() {
        System.out.println ("Datos Profesor. Profesor de nombre: " + getNombre() + " " +
        getApellidos() + " con Id de profesor: " + getIdProfesor() );   }

    public String toString () {
        return super.toString().concat(" -IdProfesor: ").concat(IdProfesor);
    }

    abstract public float importeNomina ();  // Método abstracto

} //Cierre de la clase
```

4. La palabra reservada **abstract**

- Clase **ProfesorTitular**:

```
public class ProfesorTitular extends Profesor {

        public ProfesorTitular(String nombre, String apellidos, int edad, String id) {

            super(nombre, apellidos, edad, id);
        }

        public float importeNomina () {
            return 30f * 43.20f;
        }  //Método abstracto sobreescrito en esta clase

} //Cierre de la clase
```

4. La palabra reservada **abstract**

- Clase **ProfesorInterino**:

```
public class ProfesorInterino extends Profesor {

    private Calendar fechaComienzoInterinidad;

    public ProfesorInterino (Calendar fechaInicioInterinidad) {

        super();
        fechaComienzoInterinidad = fechaInicioInterinidad;
    }
    public ProfesorInterino (String nombre, String apellidos,
            int edad, String id, Calendar fechaInicioInterinidad) {

                super(nombre, apellidos, edad, id);
                fechaComienzoInterinidad = fechaInicioInterinidad;
    }

    public Calendar getFechaComienzoInterinidad () { return fechaComienzoInterinidad; }

    public String toString () {

        return super.toString().concat (" Fecha comienzo interinidad: ")
                .concat (fechaComienzoInterinidad.getTime().toString()); }

    public float importeNomina () {
        return 30f * 35.60f ;
    } //Método abstracto sobreescrito en esta clase

} //Cierre de la clase
```

4. La palabra reservada **abstract**

- Clase **ListinProfesores**:

```
public class ListinProfesores {

    private ArrayList <Profesor> listinProfesores; //Campo de la clase

    public ListinProfesores () {
        listinProfesores = new ArrayList <Profesor> ();
    } //Constructor

    public void addProfesor (Profesor profesor) {
        listinProfesores.add(profesor);
    } //Método

    public void imprimirListin() {  //Método
        String tmpStrl = ""; //String temporal que usamos como auxiliar
        System.out.println ("Se procede a mostrar los datos de los profesores existentes en el listín \n")
        for (Profesor tmp: listinProfesores) {
            System.out.println (tmp.toString () );
            if (tmp instanceof ProfesorInterino) {
                tmpStrl = "Interino";
            }
            else {
                tmpStrl = "Titular";
            }

         System.out.println("-Tipo de este profesor:"+tmpStrl
                 +" -Nómina de este profesor: "+(tmp.importeNomina())+ "\n");}

    } //Cierre método imprimirListin
```

4. La palabra reservada **abstract**

- Clase **ListinProfesores**:

```
    public float importeTotalNominaProfesorado() {

        float importeTotal = 0f; //Variable temporal que usamos como auxiliar

        Iterator<Profesor> it = listinProfesores.iterator();

        while (it.hasNext() ) {
            importeTotal = importeTotal + it.next().importeNomina();
        }

        return importeTotal;

    } //Cierre del método importeTotalNominaProfesorado

}
```

4. La palabra reservada **abstract**

- Clase **test**:

```
public class testAbtract {

    public static void main (String [ ] Args) {

        Calendar fecha1 = Calendar.getInstance();
        fecha1.set(2019,10,22); //Los meses van de 0 a 11, luego 10 representa noviembre
        ProfesorInterino pi1 = new ProfesorInterino("José", "Hernández López", 45, "45221887-K", fecha1);
        ProfesorInterino pi2 = new ProfesorInterino("Andrés", "Moltó Parra", 87, "72332634-L", fecha1);
        ProfesorInterino pi3 = new ProfesorInterino ("José", "Ríos Mesa", 76, "34998128-M", fecha1);
        ProfesorTitular pt1 = new ProfesorTitular ("Juan", "Pérez Pérez", 23, "73-K");
        ProfesorTitular pt2 = new ProfesorTitular ("Alberto", "Centa Mota", 49, "88-L");
        ProfesorTitular pt3 = new ProfesorTitular ("Alberto", "Centa Mota", 49, "81-F");

        ListinProfesores listinProfesorado = new ListinProfesores ();
        listinProfesorado.addProfesor (pi1);
        listinProfesorado.addProfesor(pi2);
        listinProfesorado.addProfesor (pi3);
        listinProfesorado.addProfesor (pt1);
        listinProfesorado.addProfesor(pt2);
        listinProfesorado.addProfesor (pt3);
        listinProfesorado.imprimirListin();

        System.out.println ("El importe de las nóminas del profesorado que consta en el listín es " +
                listinProfesorado.importeTotalNominaProfesorado()+ " euros");

    }
}
```