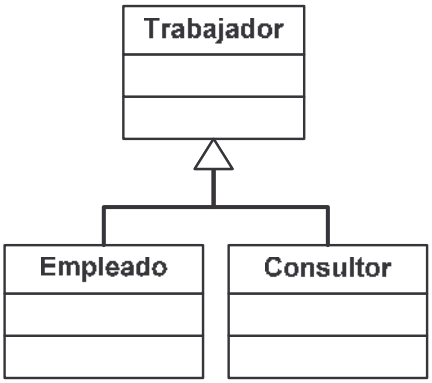
# *Herencia*

Hay clases que comparten gran parte de sus características.

> El mecanismo conocido con el nombre de herencia permite reutilizar clases: Se crea una nueva clase que extiende la funcionalidad de una clase existente sin tener que reescribir el código asociado a esta última.

- La nueva clase, a la que se denomina **subclase**, puede poseer atributos y métodos que no existan en la clase original.

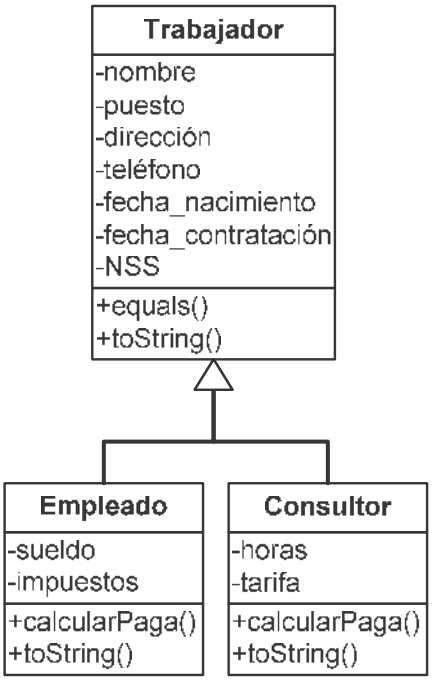
- Los objetos de la nueva clase **heredan** los atributos y los métodos **de la** clase original, que se denomina **superclase**.



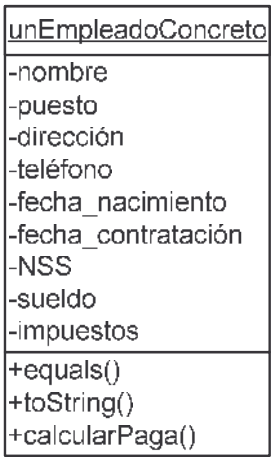

§ Trabajador es una clase genérica que sirve para almacenar datos como el nombre, la dirección, el número de teléfono o el número de la seguridad social de un trabajador.

§ 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).




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




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

En Java:

```
import java.util.Date;        // Para las fechas

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+")";
  }
}
```

NOTA: Siempre es recomendable definir los métodos `equals()` y `toString()`

```
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.

```
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.

- Tanto la clase `Empleado` como la clase `Consultor` redefinen el método `toString()` que convierte un objeto en una cadena de caracteres.

  De hecho, `Trabajador` también redefine este método, que se hereda de la clase `Object`, la clase base de la que heredan todas las clases en Java.

## *Redefinición de métodos*

Como hemos visto en el ejemplo con el método `toString()`, cada subclase hereda las operaciones de su superclase pero tiene la posibilidad de modificar localmente el comportamiento de dichas operaciones (redefiniendo métodos).

```
// 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)
```

```
// Comparación de objetos con equals()

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

```
false
```

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

```
true
```

## *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).

**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
  }
```

En las subclases de `Trabajador`, no obstante, 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;
  }
```

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);
...
```

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.

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**).

La búsqueda del método que se ha de invocar como respuesta a un mensaje dado se inicia con la clase del receptor. Si no se encuentra un método apropiado en esta clase, se busca en su clase padre (de la hereda la clase del receptor). Y así sucesivamente hasta encontrar la implementación adecuada del método que se ha de ejecutar como respuesta a la invocación original.

## *El Principio de Sustitución de Liskov*

*"Debe ser posible utilizar cualquier objeto instancia de una subclase en el lugar de cualquier objeto instancia de su superclase sin que la semántica del programa escrito en los términos de la superclase se vea afectado."*

Barbara H. Liskov & Stephen N. Zilles:
*"Programming with Abstract Data Types"*
Computation Structures Group, Memo No 99, MIT, Project MAC, 1974.
(ACM SIGPLAN Notices, 9, 4, pp. 50-59, April 1974.)

El cumplimiento del Principio de Sustitución de Liskov permite obtener un comportamiento y diseño coherente:

*Ejemplo*

Cuando tengamos trabajadores,
sean del tipo particular que sean,
el método `calcularPaga()` siempre calculará
el importe del pago que hay que efectuar
en compensación por los servicios del trabajador.

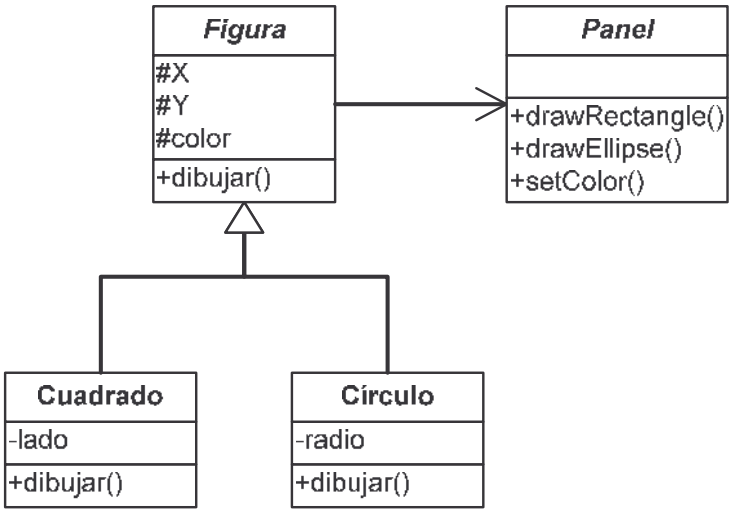
## *Acerca de la sobrecarga de métodos*

No hay que confundir el polimorfismo con la sobrecarga de métodos (distintos métodos con el mismo nombre pero diferentes parámetros).

*Ejemplo*

Podemos definir varios constructores
para crear de distintas formas objetos de una misma clase.

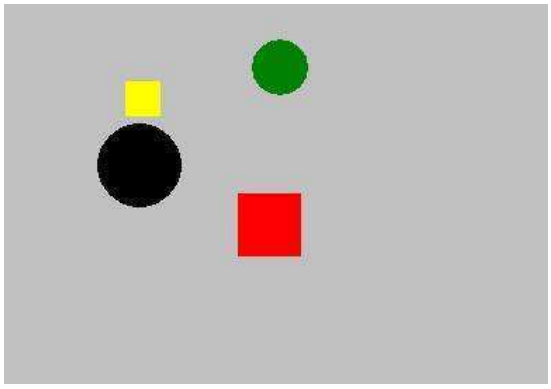
## Un ejemplo clásico: Figuras geométricas

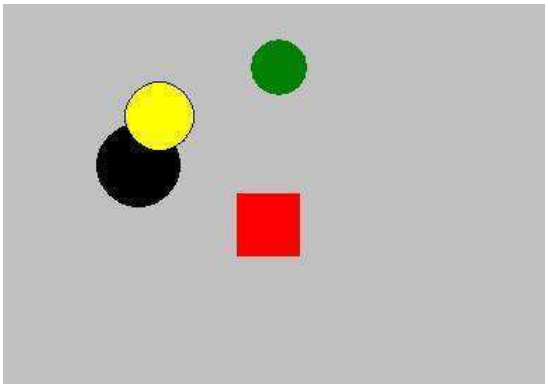



```
public class Figura
{
  protected double x;
  protected double y;
  protected Color  color;
  protected Panel  panel;


  public Figura (Panel panel, double x, double y)
  {
    this.panel = panel;
    this.x     = x;
    this.y     = y;
  }

  public void setColor (Color color)
  {
    this.color = color;
    panel.setColor(color);
  }

  public void dibujar ()
  {
    // No hace nada aquí…
  }
}
```

```
public class Circulo extends Figura
{
  private double radio;

  public Circulo(Panel panel,
                 double x, double y, double radio)
  {
    super(panel,x,y);
    this.radio = radio;
  }

  public void dibujar ()
  {
    panel.drawEllipse(x,y, x+2*radio, y+2*radio);
  }
}
```

```
public class Cuadrado extends Figura
{
  private double lado;

  public Cuadrado(Panel panel,
                  double x, double y, double lado)
  {
    super(panel,x,y);
    this.lado = lado;
  }

  public void dibujar ()
  {
    panel.drawRectangle(x,y, x+lado, y+lado);
  }
}
```





## *La palabra reservada* `final`

En Java, usando la palabra reservada `final`, podemos:

1. 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 podremos cambiarlo aunque queramos).

2. 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 ocasiones, una clase será "final"…
porque no tenga sentido crear subclases o, simplemente, porque deseamos que la clase no se pueda extender.

RECORDATORIO:
En Java, `final` también se usa para definir constantes simbólicas.