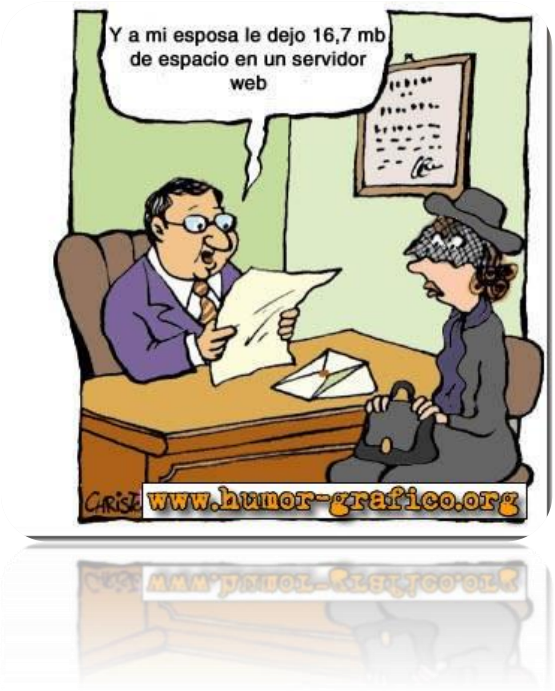
UD 4: PROGRAMACIÓN ORIENTADA A OBJETOS 2

HERENCIA

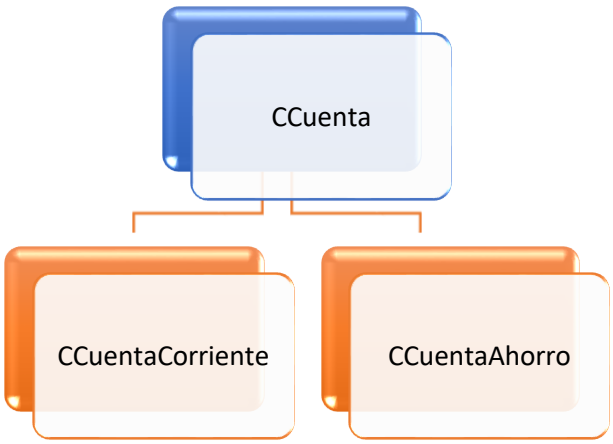
Índice



4.1 INTRODUCCIÓN

La herencia es una de las características fundamentales de la POO. Esta, provee el mecanismo más simple para especificar una forma alternativa de acceso a una clase existente, o bien para definir una nueva clase que añada, nuevas características a una ya existente, además de reutilizar código. Esta nueva clase se denomina **subclase, clase derivada o clase hija y la clase existente superclase, clase base o clase madre**.

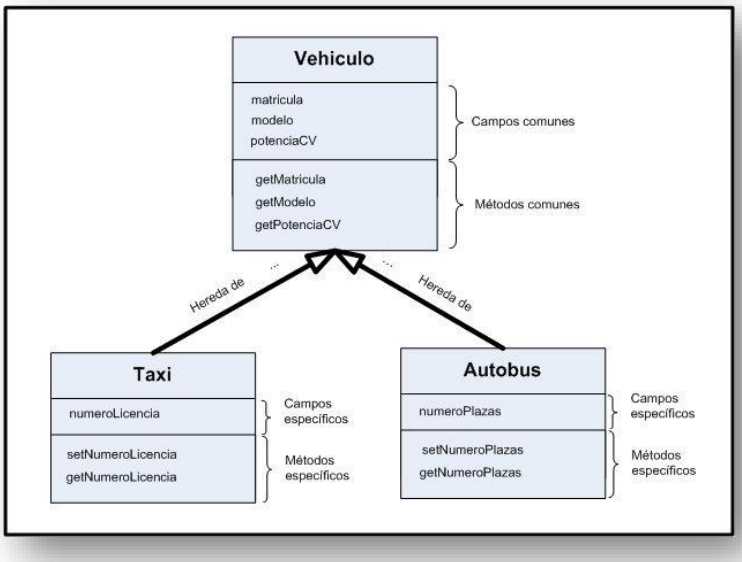
Con la herencia, todas las clases están clasificadas en una jerarquía estricta. Cada clase tiene su superclase (la superior en la jerarquía) y cada clase puede tener una o varias subclases. Se dice que las clases de la parte inferior “heredan” de las clases superiores. Por ejemplo, en la siguiente figura, las clases “CCuentaCorriente” y “CCuentaAhorro” heredan de la clase CCuenta.



4.2 SUBCLASES Y HERENCIA

a) Ejemplo

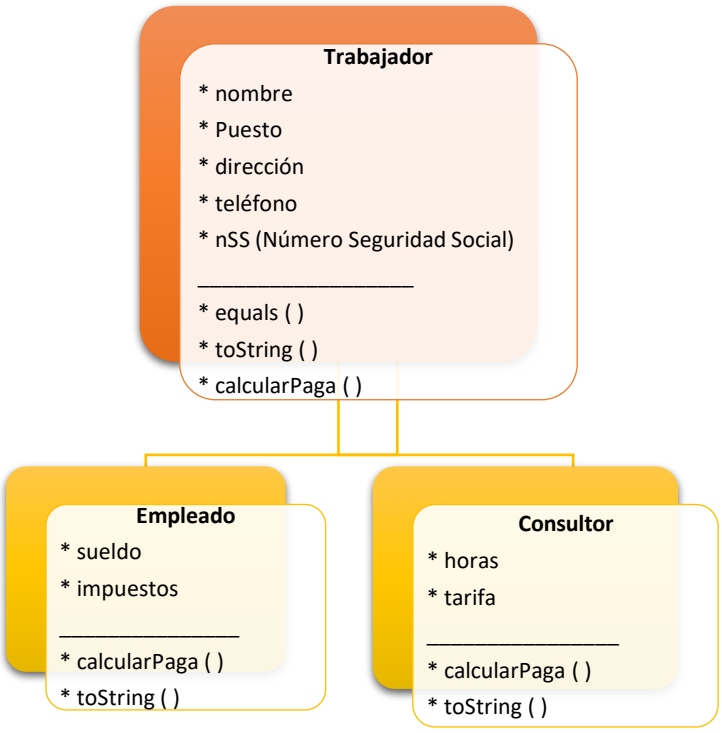
Un ejemplo típico de jerarquía de clases es el siguiente:



En general, este mecanismo permite reutilizar clases sin tener que reescribir código.

- La subclase, puede poseer atributos y métodos que no existan en la original.
- Los objetos de la nueva clase, heredan los atributos y los métodos de la clase madre o superclase. **OJO, LOS CONSTRUCTORES NO SE HEREDAN.**

Veamos un ejemplo concreto:



Los objetos de las clases Empleado y Consultor, tienen sus propios atributos y métodos más los que heredan de la clase Trabajador, es decir, que un empleado tiene nombre, puesto, etc.

Veamos el código:

```
public class Trabajador {

        private String nombre;

        private String Puesto;

        private String direccion;

        private String telefono;

        private String nSS; //Número Seguridad Social


        public Trabajador (String nombre, String puesto, String direccion,String telefono, String nSS)
{
                this.nombre = nombre;

                this.Puesto = puesto;

                this.direccion = direccion;

                this.telefono = telefono;

                this.nSS = nSS;

        }
```

```
public Trabajador (String nombre, String nSS) {
        this.nombre = nombre;
        this.nSS = nSS;
}
public String getNombre() {
        return nombre;
}
public void setNombre(String nombre) {
        this.nombre = nombre;
}
public String getPuesto() {
        return Puesto;
}
public void setPuesto(String puesto) {
        Puesto = puesto;
}
public String getDireccion() {
        return direccion;
}
public void setDireccion(String direccion) {
        this.direccion = direccion;
}
public String getTelefono() {
        return telefono;
}
public void setTelefono(String telefono) {
        this.telefono = telefono;
}
public String getnSS() {
        return nSS;
}
public void setNSS(String nSS) {
        nSS = nSS;
}
```

```
//Siempre es recomendable definir los métodos equals ( ) y toString ( ), veremos más adelante por qué motivos.

//Método para comparar objetos (trabajadores) por el número de la seguridad social, es decir, en este programa un trabajador es igual a otro cuando tienen el mismo número de la seguridad social


	public boolean equals (Trabajador t){
		return this.nSS.equals(t.nSS);
	}
	public String toString (){
		return nombre+" (NSS "+nSS+")";
	}
//No sabemos cómo se calcula la paga para un trabajador genérico, por lo que ponemos el método pero sin hacer ningún cálculo
	public double calcularPaga (){
		return 0.0;
	}
}
```

Veamos la clase hija:

```
public class Empleado extends Trabajador{
	private double sueldo;
	private double impuestos;
	private final int PAGAS= 14;//Para el ejemplo académico

	//Constructor
	public Empleado (String nombre, String nSS, double sueldo){
		super (nombre, nSS);
		this.sueldo= sueldo;
		this.impuestos= 0.3*sueldo;/*NO es buena idea hacer cálculos en el constructor, solo está puesto aquí para hacer el ejemplo más sencillo*/
	}
	public double getSueldo() {
		return sueldo;
	}
	public void setSueldo(double sueldo) {
		this.sueldo = sueldo;
	}
```

```
        public double getImpuestos() {
                return impuestos;
        }
        public void setImpuestos(double impuestos) {
                this.impuestos = impuestos;
        }
        public int getPAGAS() {
                return PAGAS;
        }
        //Para la nómina de un Empleado
        public double calcularPaga (){
                return (sueldo-impuestos)/PAGAS;
        }
        public String toString (){
                return "Empleado "+super.toString();
        }
}
```

OJO: 

- Para indicar herencia se usa la palabra reservada **extends** entre la clase hija y la superclase, esto indica que la clase Empleado extiende (hereda) de la clase Trabajador.
- Con super (), estamos haciendo una llamada al constructor de la clase madre, es decir, llamando al constructor de la clase Trabajador que se encarga de inicializar los atributos de esa clase.
- OJO: Esta llamada debe ser la primera línea del código del constructor de la clase hija. Si no se pone nada, se llama al constructor por defecto de la clase madre.

Clase Consultor:

```
public 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;
```

```
	}
	public int getHoras() {
		return horas;
	}
	public void setHoras(int horas) {
		this.horas = horas;
	}
	public double getTarifa() {
		return tarifa;
	}
	public void setTarifa(double tarifa) {
		this.tarifa = tarifa;
	}
//Ahora la forma de calcular lo que hay que pagar es por horas
	public double calcularPaga (){
		return horas*tarifa;
	}
	public String toString (){
		return "Consultor "+super.toString();
	}
}
```

OJO:

- Esta clase también define un método calcularPaga ( ) pero ahora, la paga se calcula de forma diferente, a esto se le llama “redefinir o reescribir métodos (@overwrite)”.
- Empleado y Constructor redefinen el método toString, que convierte un objeto en una cadena de texto. De hecho, la clase Trabajador también lo hace, puesto que todas las clases heredan de una superclase llamada Object **(en Java, cualquier cosa es un objeto, en serio, cualquier cosa).**

Clase de prueba:

```
public class PruebaHerencia1 {
	public static void main(String[] args) {
		//Declaración de variables
		Trabajador miTrabajador;
```

```
        Empleado miEmpleado;
        Consultor miConsultor;

        //Creación de objetos
        miTrabajador= new Trabajador ("Bill Gates", "456");
        miEmpleado= new Empleado ("Steve Jobs", "123",24000.0);
        miConsultor= new Consultor("Bill Gates", "456", 10, 50.0);

        //Mostrar datos
        System.out.println(miTrabajador);
        System.out.println(miEmpleado);
        System.out.println(miConsultor);

        //Comparación de objetos
        System.out.println(miTrabajador.equals(miEmpleado));//Devolverá false
        System.out.println(miTrabajador.equals (miConsultor));//Devolverá true ya que tenemos definido el método equals en la clase Trabajador que devuelve true si los dos objetos tienen el mismo número de la seguridad social (456)
        //Puede ver qué pasaría si llamas al método calcularPaga con cada uno de los objetos instanciados
    }
}
```

Una subclase solo puede ser "hija" de una superclase, esto se denomina herencia simple. Java no permite la herencia múltiple, es decir, heredar de varias superclases.

Una subclase puede ser a su vez, superclase de otras, dando lugar a una jerarquía de clases

b) Control de acceso a los miembros de las clases

Las subclases heredan todos los miembros de su superclase (OJO, REPETIMOS: LOS CONSTRUCOTRES NO SE HEREDAN), aunque no todos los miembros tienen porque ser accesibles. En particular, los miembros privados de una superclase no son directamente accesibles en sus subclases.

La siguiente tabla muestra las posibilidades de acceso a una clase existente:

	Un miembro declarado en una clase como:			
Puede ser accedido desde:	public	protected	private	Sin especificar (acceso de paquete)
Su misma clase	SÍ	SÍ	SÍ	SÍ
Cualquier clase o subclase del mismo paquete	SÍ	SÍ	NO	SÍ
Cualquier clase de otro paquete	SÍ	NO	NO	NO
Cualquier subclase de otro paquete	SÍ	SÍ	NO	NO

En general:

• <ninguno>: accesible desde el paquete

• public: accesible desde todo el programa

• private: accesible sólo desde esa clase

• protected: accesible desde el paquete y desde sus subclases en cualquier paquete

Definir atributos protected NO suele ser una buena práctica de programación:

• Ese campo sería accesible desde cualquier subclase y puede haber muchas y eso complicaría enormemente la tarea de mantenimiento.

• Además el campo es accesible desde todas las clases del paquete (subclases o no).

• Regla general: todos los campos de una clase son privados.

• Se proporcionan métodos públicos para leer y/o cambiar los campos (pero sólo cuando sea necesario).

• En el caso de que se desee que un campo sólo pueda ser leído y/o cambiado por las subclases se hacen métodos protected.

c) ¿Qué miembros hereda una subclase?

- ✓ Todos los miembros de su superclase excepto los constructores, lo que no significa que tenga acceso directo a todos los miembros.
  Una subclase no tiene acceso directo a aquellos miembros privados (private) de su superclase pero sí puede acceder directamente a los miembros públicos (public) y

protegidos (protected). Si además pertenece al mismo paquete que su superclase también puede acceder a los miembros predeterminados (sin escribir nada).

- ✓ Los miembros heredados por una subclase pueden, a su vez, ser heredados por más subclases de ella. A esto se llama *propagación de herencia",* ("heredar de abuelos").

- ✓ Una subclase puede añadir sus propios atributos y métodos. Si el nombre de alguno de estos miembros coincide con el de un miembro heredado, este último queda oculto para la subclase, es decir, la subclase ya no puede acceder directamente a ese miembro (si podía acceder por permisos). Veremos, en el siguiente apartado, que es posible acceder a un miembro oculto usando la palabra clase **super** (super.miembroOculto).

4.3 ATRIBUTOS CON EL MISMO NOMBRE

Como sabemos, una subclase puede acceder directamente a un atributo público, protegido o predeterminado se su superclase. ¿Qué sucede si definimos en la subclase uno de estos atributos, con el mismo nombre que tiene en la superclase? Por ejemplo:

```
public class Madre{
  public int atributo_x = 1;

  public int método_x(){
    return atributo_x * 10;
  }
  public int método_y(){
    return atributo_x + 100;
  }
}

public class Hija extends Madre
{
  protected int atributo_x = 2;

  public int método_x(){//Redefinido, se ve más adelante
    return atributo_x * (-10);
  }
}
```

Cuando se redefine un método de una superclase en una subclase se oculta el método de la superclase pero NO LAS SOBRECARGAS del mismo.

Ahora, la definición del atributo **atributo_x** en la subclase, oculta la definición del atributo con el mismo nombre en la superclase. Por tanto, la última línea del código siguiente devolverá el valor del atributo_x de la Hija. Si este atributo no hubiera sido definido en la subclase, entonces el valor devuelto sería el valor del atributo_x de la superclase (Madre).

```
public class Test
{
  public static void main(String[] args)
  {
    Hija objHija = new Hija();
    System.out.println(objHija.atributo_x); // escribe 2
    System.out.println(objHija.método_y()); // escribe 101
    System.out.println(objHija.método_x()); // escribe -20
  }
}
```

Para acceder al atributo_x de la superclase debemos utilizar para ese atributo nombres diferentes en la superclase y en la subclase.

De todas formas, si se ha usado el mismo nombre también se puede acceder usando la palabra **super**, por ejemplo:

```
  public int método_x()
  {
      return super.atributo_x * (-10);
  }
```

También podemos referirnos al dato *atributo_x de la subclase con la expresión*:

```
this.atributo_x;
```

4.4 REDEFINIR MÉTODOS DE LA SUPERCLASE

Cuando se invoca a un método en respuesta a un mensaje recibido por un objeto, Java busca su definición en la clase del objeto. La definición del método que allí se encuentra puede pertenecer a la propia clase o puede haber sido heredada. Esto quiere decir que, si no la encuentra, Java sigue buscando en la jerarquía de clases hacia arriba hasta que la localice.

En el caso de que el método heredado por la subclase sea abstracto, es OBLIGATORIO redefinirlo, de lo contrario la subclase también debería ser

En ocasiones, puede que deseemos que un objeto de una subclase responda al mismo método heredado de su superclase pero con un comportamiento diferente. Esto implica REDEFINIR en la subclase el método heredado de su superclase.

Redefinir o rescribir significa volverlo a escribir en la subclase con:

- ✓ **El mismo nombre.**
- ✓ **La misma lista de parámetros.**
- ✓ **Mismo tipo de valor de retorno.**

Su cuerpo (lo que hace el método) será adaptado a las necesidades de la subclase. Esto es lo que se ha hecho en el ejemplo anterior con el método x.

En el main del ejemplo anterior, se creó un objHija y se invocó a su método_y. Como la clase del objeto, Hija, no define este método, Java ejecuta el heredado. También se invocó a su método_x y en este caso, como existe una definición para este método, es la que se ejecuta.

Para acceder a un método de la superclase que ha sido redefinido en la subclase (igual que para los atributos), tendremos que usar la palabra **super**.

Por ejemplo:

```
public int método_z()
{
        atributo_x = super.atributo_x + 3;
        return super.método_x() + atributo_x;
}
```

No se puede redefinir un método en una subclase y hacer que el acceso sea MÁS RESTRICTIVO que el original. El orden de más a menos restrictivo es: ***private, predeterminado, protected y public.***

OJO: Super solo puede ser utilizado desde dentro de la clase que proporciona los miembros redefinidos.

Igualmente, podemos referirnos al método_x de la subclase con **this**:

```
this.metodo_x ( );
```

4.5 CONSTRUCTORES

Al crear un objeto de la subclase, primero se llama a su constructor, que a su vez invoca al constructor de la superclase y así hacia arriba con todas. Por tanto, primero se invocan a los constructores de las superclases de arriba debajo de la jerarquía y finalmente el de la subclase. La llamada al constructor de la superclase será implícita o explícita.

```
nombre_subclase (lista de parámetros) {
        super (lista de parámetros);
        //Cuerpo del constructor de la subclase
    }
```

La palabra super en este caso, invoca al constructor de la superclase. Esta línea TIENE QUE SER LA PRIMERA DEL CUERPO DEL CONSTRUCTOR.

Si la superclase no tiene constructor de forma explícita o tiene uno que no requiere parámetros, no es necesario invocarlo explícitamente ya que Java lo invocará mediante super ( ) sin argumentos.

Ejemplo:

```
public Consultor (String nombre, String nSS, int horas, double tarifa){
        super (nombre, nSS);
        this.horas=horas;
        this.tarifa=tarifa;
}
```

El constructor debe tener tantos parámetros como atributos heredados y propios tenga la clase. Se pueden por tanto crear cuantos constructores sean necesarios.

NOTA: Si nos encontramos con varios niveles de jerarquía, las subclases solo tienen acceso directo a los miembros que puedan de su superclase inmediata y no a niveles superiores, es decir, no es posible usar:

super.super.metodo_x ( );

4.6 EL MODIFICADOR FINAL

La palabra *final* se puede usar para:

a) Evitar que un método se pueda redefinir en una subclase:

```
public 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 lo podremos cambiar, aunque queramos).

b) Evitar que se puedan crear subclases de una clase dada.

Puede que nos encontremos en la situación de que no tenga sentido crear una subclase o que no lo deseemos.

```
public final class Empleado extends Trabajador{
        //…
}
```

Ahora, no se podrían crear subclases de la clase Empleado

c) Como ya hemos visto, también se usa para definir constantes.

4.7 CLASES Y MÉTODOS ABSTRACTOS

En una jerarquía de clases, una clase es tanto más especializada cuanto más alejada está de la raíz, siendo esta última la clase de la que heredan directa o indirectamente el resto de las clases de la jerarquía. Es más genérica cuanto más cerca esté de la raíz. La clase más genérica de Java es la clase **Object** de la que heredan todas las clases del API de Java (“En java cualquier cosa es un objeto, en serio, cualquier cosa”).

Cuando una clase se diseña para ser “genérica”, es casi seguro que no necesitaremos crear objetos de ella. La razón de su existencia es proporcionar atributos y comportamientos que serán “compartidos” por todas las subclases. Una clase que se comporte así, se llama **abstracta** y se define con la palabra reservada ***abstract***.

```
public abstract class CCuenta {
        …
}
```

Una clase abstracta puede contener el mismo tipo de miembros que una que no lo sea y ADEMÁS, puede contener **métodos abstractos** (que las clases no abstractas no pueden tener), es decir, cuando una clase contiene, al menos, un método abstracto, debe ser abstracta y declararse como tal.

a) ¿Qué es un método abstracto?

Es un método calificado con la palabra **abstract** y con la particularidad de que **no tiene cuerpo**, por ejemplo:

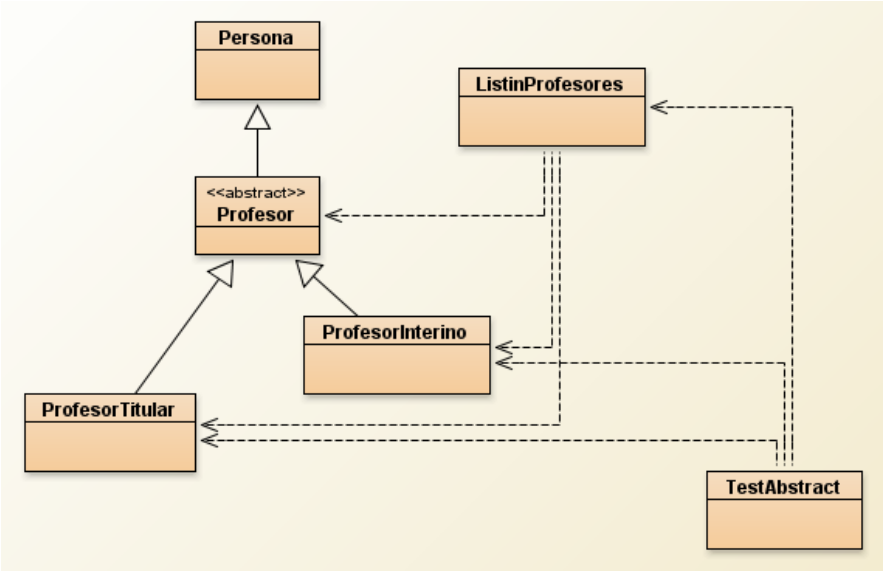
public abstract void comisiones ( );

b) ¿Por qué no tiene cuerpo?

Porque la idea es proporcionar métodos que deban ser "redefinidos" en las subclases (clases hijas) de la clase abstracta, con la intención de adaptarlos a las necesidades particulares de estas.

Veamos todo esto con un ejemplo:

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?



Podríamos definir una public abstract class Profesor. Cuando utilizamos esta sintaxis, no resulta posible instanciar la clase, es decir, no resulta posible crear objetos de ese tipo. 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. 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”.

NOTA: En inglés abstract significa “resumen”, por eso en algunos textos en castellano a las clases abstractas se les llama resúmenes. Nosotros nunca usaremos la palabra resumen.

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. Eclipse muestra un error.

A su vez, las clases abstractas suelen contener métodos abstractos: la situación es la misma. Para que un método se considere abstracto ha de incluir en su signatura la palabra clave abstract. Además, un método abstracto tiene estas peculiaridades:

a) **No tiene cuerpo** (llaves): sólo consta de signatura con paréntesis.

b) Su signatura **termina con un punto y coma.**

c) **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.

d) Los métodos abstractos **forzosamente habrán de estar sobrescritos 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 sobrescritos para todos los métodos abstractos de sus superclases.

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 sobrescriban el método declarado como abstracto.

Sintaxis tipo:

abstract public/private/protected Tipo de Retorno/void nombreMétodo (parámetros …);

Por ejemplo:

abstract public void generarNomina (int diasCotizados, boolean plusAntiguedad);

public abstract void generarNomina (int diasCotizados, boolean plusAntiguedad);

Valen las dos formas.

Que un método sea abstracto tiene otra implicación adicional: que podamos invocar el método abstracto sobre una variable de la superclase que apunta a un objeto de una subclase de modo que el método que se ejecute sea el correspondiente al tipo dinámico de la variable. En cierta manera, podríamos verlo como un método sobrescrito para que Java comprenda que debe buscar dinámicamente el método adecuado según la subclase a la que apunte la variable.

¿Es necesario que una clase que tiene uno o más métodos abstractos se defina como abstracta? Sí, si declaramos un método abstracto el compilador nos obliga a declarar la clase como abstracta porque si no lo hiciéramos así tendríamos un método de una clase concreta no ejecutable, y eso no es admitido por Java.

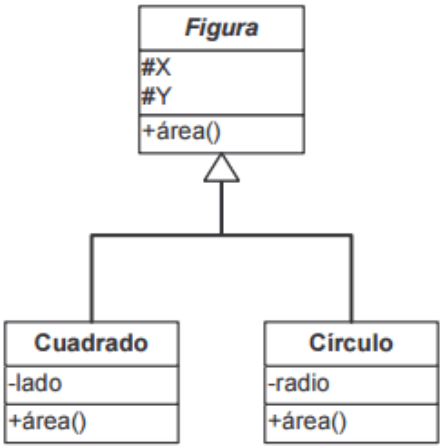
¿Una clase se puede declarar como abstracta y no contener métodos abstractos? Sí, una clase puede ser declarada como abstracta y no contener métodos abstractos. En algunos casos la clase abstracta simplemente sirve para efectuar operaciones comunes a subclases sin necesidad de métodos abstractos. En otros casos sí se usarán los métodos abstractos para referenciar operaciones en la clase abstracta al contenido de la sobrescritura en las subclases.

¿Una clase que hereda de una clase abstracta puede ser no abstracta? Sí, de hecho, esta es una de las razones de ser de las clases abstractas. Una clase abstracta no puede ser instanciada, pero pueden crearse subclases concretas sobre la base de una clase abstracta, y crear instancias de estas subclases. Para ello hay que heredar de la clase abstracta y anular los métodos abstractos, es decir, implementarlos.

Ejemplo completo sencillo, pero bastante clarificador:

¿Tiene sentido crear un objeto tipo Figura? ¿Podrías dibujarlo?

No tiene sentido porque no sabemos qué tipo de figura es, de hecho, no se sabe hasta que no extendemos la clase con las hijas que especifican o concretan la figura a Cuadrado o Círculo.




```java
public abstract class  Figura {
    //PUNTOS PARA GRAFICAR
    private double x;

    private double y;

    //EL CONSTRUCTOR NO LO USAMOS PARA INSTANCIAR SINO PARA DEFINIRLA EN LA SUBCLASE
    public Figura(double x, double y){
        this.x=x;
        this.y=y;

    }
public abstract double calcularArea();
}
```

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

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

    }

    @Override
    public double calcularArea() {
        //IMPLEMENTACION DEL METODO
        return lado*lado;
    }

}
```

```java
public class Rectangulo extends Figura {
private double altura,base;
    public Rectangulo(double x, double y,double b,double h) {
        super(x, y);
        this.altura=h;
        this.base=b;
    }

    @Override
    public double calcularArea() {
        // TODO Auto-generated method stub
        return base*altura;
    }
}
```

```
public class main {
    public static void main(String[] args) {
        Figura cuadrado = new Cuadrado(1,2,2);//NO DA ERROR

        System.out.println(cuadrado.calcularArea());
    }

}
```

Veremos con más profundidad la creación de este tipo de objetos en el polimorfismo. De momento, os dejo a vosotros la creación en el main de varios objetos (si se puede), tipo Figura, tipo Cuadrado, tipo Rectángulo...

4.8 MÉTODOS equals ( ) Y toString ( )

Con lo expuesto en anteriores apartados, ya sabemos que, mientras no se hayan sobreescrito, los métodos de la superclase universal Object estarán disponibles para todos los objetos. Dentro de los métodos de la clase Object hay varios importantes, entre los que podemos citar los métodos toString ( ) que devuelve un tipo String y ya hemos usado mucho y el método equals (Object obj) que devuelve un tipo boolean.

Veamos primero el método toString ( ) con más detenimiento. El propósito de este método es **asociar a todo objeto un texto representativo.** Llamar a toString ( ) sobre un objeto Integer producirá un resultado que es más o menos obvio: nos devuelve el entero asociado, solo que en forma de String. Pero ¿qué ocurre cuando invocamos el método sobre un objeto definido por nosotros? Este sería el caso de una invocación como:

System.out.println (“Obtenemos “ + profesor1.toString());

En este caso, el resultado que obtenemos es del tipo:

“Obtenemos Profesor@1de9ac4”.

El método efectivamente nos ha devuelto un String, pero ¿qué sentido tiene lo que nos ha devuelto? El resultado obtenido consta del nombre de la clase seguido de una cadena “extraña” que representa la dirección de memoria en que se encuentra el objeto. Este resultado en general es poco útil por lo que el método toString() es un método que habitualmente se sobreescribe al crear una clase. De hecho, la mayoría de las clases de la biblioteca estándar de Java sobreescriben el

método toString(). Por otro lado, es frecuente que cuando un programador incluye métodos como imprimir…, mostrar…, listar…, etc. de alguna manera dentro de ellos se realicen llamadas al método toString() para evitar la repetición de código. También es frecuente que toString() aparezca en tareas de depuración ya que nos permite interpretar de forma "humana" los objetos.

Supongamos que en una clase Persona redefinimos toString() de la siguiente manera:

```
public String toString() {
        return nombre.concat(" "). concat(apellidos);
}
```

En la clase profesor, que hereda de Persona, podríamos tener (el método concat hace lo mismo que +, concatenar cadenas):

```
public String toString() {
        return super.toString().concat(" con Id de profesor: ").concat (      getIdProfesor() );
}

public void mostrarDatos() {
        System.out.println ("Los datos disponibles son: " + this.toString() );
}
```

En este ejemplo vemos, aparte del uso del método concat para concatenar Strings, una llamada a la superclase para recuperar el método toString de la superclase y cómo otro método (mostrarDatos) hace uso del método toString(). Usar toString es ventajoso porque no siempre nos interesa mostrar cadenas de texto por consolas en pantalla: tener los datos en un String nos permite p.ej. grabarlos en una base de datos, enviarlos en un correo electrónico, etc., además de poder mostrarlos por pantalla si queremos.

Aunque ya venimos usándolo, conviene remarcar que cuando usamos los métodos println y print del objeto System.out, cuando se incluye un término que no es un String, se invoca automáticamente el método toString() del objeto sin necesidad de escribirlo de forma explícita. Así:

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

es equivalente a

```
System.out.println (Profesor.toString() );
```

Ya hemos utilizado el método equals en diferentes ocasiones y sabemos que es la forma en que debemos comparar objetos.

Los objetos no se pueden comparar utilizando el operador = =

El método equals está implementado en el API de Java para la mayoría de las clases. Por ello, podemos usarlo directamente para comparar Strings por ejemplo. Ahora bien, ¿qué ocurre en una clase creada por nosotros? Si escribimos algo como if (profesor1.equals(profesor2)), al no estar sobrescrito el método equals para la clase Profesor, el resultado es impredecible o incorrecto. Por tanto, para comparar objetos de tipo Profesor hemos de sobrescribir el método en la clase. Veamos esto con un ejemplo:

Supongamos la clase Persona y la clase Profesor:

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

   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 boolean equals (Object obj) {

      if (obj instanceof Persona) {
```

```
            Persona tmpPersona = (Persona) obj;//Casting a tipo persona. Esta forma es algo antigua (a equals se le debe pasar un objeto del tipo que estemos usando, no un Object general)y veremos en el tema siguiente una forma para no tener que castear.
            if (this.nombre.equals(tmpPersona.nombre) &&
            this.apellidos.equals(tmpPersona.apellidos) &&
            this.edad == tmpPersona.edad) {
              return true;
            }else {
              return false;
            }
        } else {
            return false;
        }
    }
}

package ejemploequals;
public class Profesor extends Persona {
    private String IdProfesor;

    public Profesor () {
            super();
        IdProfesor = "Unknown";
    }
    public Profesor (String nombre, String apellidos, int edad) {
        super(nombre, apellidos, edad);
        IdProfesor = "Unknown";
    }
    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 boolean equals (Object obj) {
    if (obj instanceof Profesor) {
      Profesor tmpProfesor = (Profesor) obj;//Usamos el equals de la clase madre en el siguiente if y agregamos la comparación del IdProfesor
      if (super.equals(tmpProfesor) &&
            this.IdProfesor.equals(tmpProfesor.IdProfesor))                    {
        return true;
      } else {
       return false;
      }
    }else {
       return false;
    }
      }
 }//Cierre de la clase

package ejemploequals;
public class Test1 {
      public static void main(String[] args) {
          Profesor profesor1 = new Profesor ("Juan", "Hernández García", 33);
          profesor1.setIdProfesor("Prof 22-387-11");
          Profesor profesor2 = new Profesor ("Juan", "Hernández García", 33);
          profesor2.setIdProfesor("Prof 22-387-11");
          Profesor profesor3 = new Profesor ("Juan", "Hernández García", 33);
          profesor3.setIdProfesor("Prof 11-285-22");

          Persona persona1 = new Persona ("José", "Hernández López", 28);
          Persona persona2 = new Persona ("José", "Hernández López", 28);
          Persona persona3 = new Persona ("Ramiro", "Suárez Rodríguez", 19);

      System.out.println ("¿Son iguales la persona1 y la persona2? " +      persona1.equals(persona2));
      System.out.println ("¿Son el mismo objeto la persona1 y la persona2? " + (persona1      == persona2));
      System.out.println ("¿Son iguales la persona1 y la persona3?  " +      persona1.equals(persona3));
```

```
        System.out.println ("¿Son iguales el profesor1 y el profesor2? " +      profesor1.equals(profesor2));
        System.out.println ("¿Son iguales el profesor1 y el profesor3? " +      profesor1.equals(profesor3));
    }
}
```

Los resultados serán:

```
¿Son iguales la persona1 y la persona2? true
¿Son el mismo objeto la persona1 y la persona2? false
¿Son iguales la persona1 y la persona3? false
¿Son iguales el profesor1 y el profesor2? true
¿Son iguales el profesor1 y el profesor3? false
```

Analicemos ahora lo que hemos hecho. En la clase Persona, mediante la redefinición del método equals, hemos definido que para nosotros dos personas van a ser iguales si coinciden su nombre, apellidos y edad. El criterio lo hemos fijado nosotros.

Otra opción hubiera sido establecer que dos personas son iguales si coinciden nombre y apellidos. Fíjate que los Strings los comparamos usando el método equals del API de Java para los objetos de tipo String, mientras que la edad la comparamos con el operador == por ser un tipo primitivo.

Otra cuestión relevante es el tratamiento de tipos: **el método equals requiere como parámetro un tipo Object** y no un tipo Persona. Así hemos de escribirlo para que realmente sea una redefinición del método de la clase Object. Si usáramos otra signatura, no sería una redefinición del método, sino un nuevo método.

En primer lugar, comprobamos si el objeto pasado como parámetro es un tipo Persona. Si no lo es, devolvemos como resultado del método false; los objetos no son iguales (no pueden serlo si ni siquiera coinciden sus tipos).

En segundo lugar, una vez verificado que el objeto es portador de un tipo Persona, creamos una variable de tipo Persona a la que asignamos el objeto pasado como parámetro valiéndonos de casting (enmascaramiento) que estudiaremos más adelante. Esta variable la creamos para poder invocar campos y métodos de la clase Persona, ya que esto no podemos hacerlo sobre un objeto de tipo Object. Con esta variable, realizamos las comparaciones oportunas y devolvemos un resultado.

En la clase Profesor hemos sobrescrito también el método equals con un tratamiento de tipos y uso de casting similar. En este caso invocamos el método equals de la superclase Persona, con lo que decimos que para que dos profesores sean iguales han de coincidir nombre, apellidos y edad. Además establecemos otro requisito para considerar a dos profesores iguales: que coincida su IdProfesor. Esto lo hacemos a nuestra conveniencia.

Finalmente, en el Test realizamos pruebas de los métodos implementados, comprobando por ejemplo que una persona1 puede ser igual a otra persona2 (de acuerdo con la definición dada al método equals) pero ambas ser diferentes objetos.

4.9 CONVERSIÓN DE TIPOS O CASTING

En Java es posible transformar el tipo de una variable u objeto en otro diferente al original con el que fue declarado. Este proceso se denomina "conversión", "moldeado" o "tipado" y es algo que debemos manejar con cuidado pues un mal uso de la conversión de tipos es frecuente que dé lugar a errores.

¡¡ES MEJOR EVITARLO Y DISEÑAR BIEN LOS TIPOS!!

Una forma de realizar conversiones consiste en colocar el tipo destino entre paréntesis, a la izquierda del valor que queremos convertir de la forma siguiente:

Tipo VariableNueva = (NuevoTipo) VariableAntigua;

Por ejemplo:

```
int b=0;
float a =(float)b;

char c = (char) Leer.datoInt ();
```

En el primer ejemplo, asignamos un valor int a un float. En el segundo caso, leemos por teclado un valor entero que se convierte en un char debido a la conversión (char), y el valor resultante se almacena en la variable de tipo carácter c.

El tamaño de los tipos que queremos convertir es muy importante. No todos los tipos se convertirán de forma segura. Por ejemplo, al convertir un long en un int, el compilador corta los 32 bits superiores del long (de 64 bits), de forma que encajen en los 32 bits del int, con lo que si contienen información útil, ésta se perderá. Este tipo de conversiones que suponen pérdida de información se denominan “conversiones no seguras” y en general se tratan de evitar, aunque de forma controlada pueden usarse puntualmente.

De forma general trataremos de atenernos a la norma de que "en las conversiones debe evitarse la pérdida de información". En la siguiente tabla vemos conversiones que son seguras por no suponer pérdida de información.

TIPO ORIGEN	TIPO DESTINO
byte	*double, float, long, int, char, short*
short	*double, float, long, int*
char	*double, float, long, int*
int	*double, float, long*
long	*double, float*
float	*Double*

No todos los tipos se pueden convertir de esta manera. Como alternativa, existen otras formas para realizar conversiones.

MÉTODO VALUEOF PARA CONVERSIÓN DE TIPOS

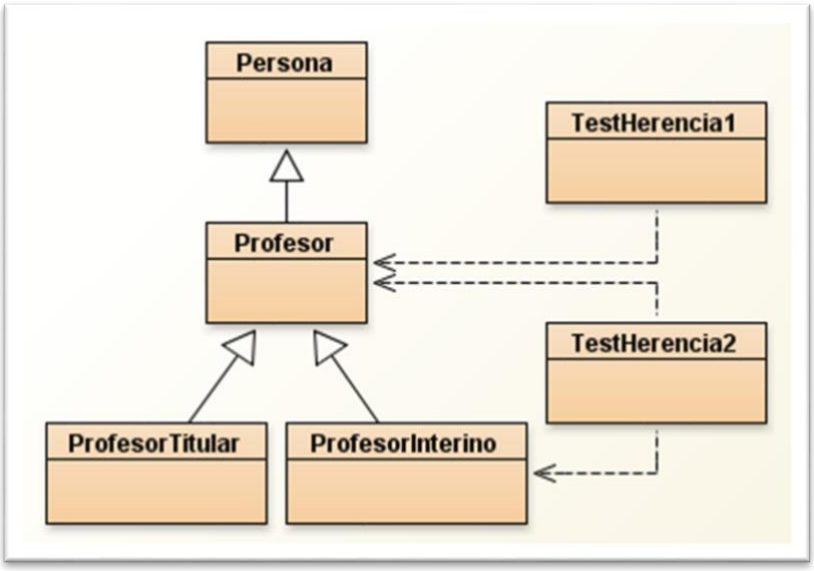
El método valueOf es un método sobrecargado aplicable a numerosas clases de Java y que permite realizar conversiones de tipos. Veamos algunos ejemplos de uso.

EXPRESIÓN	INTERPRETACIÓN
Integer miInteger = miInteger.valueOf (i)	Con i entero primitivo que se transforma en Integer
Integer miInteger = miInteger.valueOf (miString)	El valor del String se transforma en Integer
String miString = miString.valueOf (miBooleano)	El booleano se transforma en String “true” o “false”
String miString = miString.valueOf (miChar)	El carácter (char) se transforma en String
String miString = miString.valueOf (miDouble)	El double se transforma en String. Igualmente aplicable a float, int, long.

No todas las conversiones son posibles. Muchas veces por despiste los programadores escriben instrucciones de conversión incoherentes como miInteger = (int) miString; El resultado en este caso es un error de tipo “Inconvertible types”. Un uso típico de valueOf es para convertir tipos primitivos en objetos.

MÁS SOBRE CASTING

Java admite la conversión de tipos con ciertas limitaciones. Consideremos una jerarquía de herencia como la que vemos en el siguiente diagrama de clases, sobre el que vamos a analizar las posibilidades de conversión de tipos de distintas formas.



a) Conversión hacia arriba (UpCasting)

Se trataría por ejemplo de poner lo que está a un nivel inferior en un nivel superior, por ejemplo, poner un profesor interino como profesor. Posible código:

```
profesor43 = profesorinterino67;
```

Asignamos lo que está abajo (el profesor interino) arriba (como profesor). Esto es posible, pero dado que lo que está abajo generalmente contiene más campos y métodos que lo que está arriba, perderemos parte de la información. **Sobre el objeto profesor43 ya no podremos invocar los métodos propios de los profesores interinos.**

b) Conversión hacia abajo (DownCasting)

Se trataría de poner lo que está arriba abajo, por ejemplo, poner un Profesor como ProfesorInterino. Esto no siempre es posible. El supertipo admite cualquier forma (es polimórfico, ya lo veremos más adelante) de los subtipos: si el supertipo almacena el subtipo al que queremos realizar la conversión, será posible usando lo que se denomina “Enmascaramiento de tipos” o “Hacer Casting” (cast significa “moldear”). Si el supertipo no almacena el subtipo al que queremos convertirlo, la operación no es posible y saltará un error.

Ejemplo:

```
Profesor p1; //p1 es tipo Profesor. Admite ser Profesor, ProfesorTitular o ProfesorInterino.
```

```
ProfesorInterino p44 = new ProfesorInterino(); //p44 es ProfesorInterino.

p1 = p44; // Conversión hacia arriba: sin problema. Ahora p1 que es tipo profesor, almacena un profesor interino

p44 = p1 // ERROR en la conversión hacia abajo.
```

El compilador no llega tan lejos como para saber si p1 almacena un profesor interino u otro tipo de profesor y ante la incertidumbre salta un error. La forma de forzar al compilador a que "comprenda" que p1 está almacenando un profesor interino y por tanto puede asignarse a una variable que apunta a un profesor interino se llama "hacer casting". Escribiríamos lo siguiente:

```
p44 = (ProfesorInterino) p1;
```

El nombre de un tipo entre paréntesis se llama "operador de enmascaramiento de tipos" y a la operación en sí la llamamos enmascaramiento o hacer casting. El casting es posible si el objeto padre contiene a un objeto hijo, **pero si no es así aunque la compilación sea correcta en tiempo de ejecución saltará un error "ClassCastException",** o error al hacer cast con las clases. No siempre es fácil determinar a qué tipo de objeto apunta **una** variable. Por ello el casting o enmascaramiento debiera evitarse en la medida de lo posible ya que el que un programa compile, pero contenga potenciales errores en tiempo de ejecución es algo no deseable.

> Un programa bien estructurado normalmente no requerirá hacer casting reiteradamente, o lo hará de forma muy controlada. Si durante la escritura de un programa nos vemos en la necesidad de realizar casting, debemos plantearnos la posibilidad de reestructurar código para evitarlo.

Cuando incluyamos conversiones de tipos usando casting en nuestro código, para evitar errores conviene filtrar las operaciones de enmascaramiento asegurándonos antes de hacerlas de que el objeto padre contiene un hijo del subtipo al que queremos hacer la conversión. Esta verificación se hace usando la palabra clave instanceof como explicaremos a continuación.

c) Conversión de lado a lado

Se trataría de poner lo que está a un lado al otro lado, por ejemplo, convertir un ProfesorInterino en ProfesorTitular o al revés.

¡¡¡Esto no es posible en ningún caso!!!

d) Determinación del tipo de variables con instanceof

La palabra clave instanceof (es instancia de), todo en minúsculas, sirve para verificar el tipo de una variable. La sintaxis que emplearemos para instanceof y sus normas de uso serán las siguientes:

```
if (profesor43 instanceof ProfesorInterino) {…} else { …}
```

a) Sólo se pueden comparar instancias que relacionen dentro de la jerarquía de tipos (en cualquier dirección) pero no objetos que no relacionen en una jerarquía. Es decir, no podemos comparar profesores con taxis, por ejemplo, porque no relacionarán dentro de una jerarquía.

b) Solo se puede usar instanceof asociado a un condicional. No se puede usar por ejemplo directamente en una impresión por pantalla con System.out.println(…).

Ejemplos de sintaxis:

if (profesor73 instanceof ProfesorInterino) --> la sintaxis es válida.

if (interino1 instanceof ProfesorInterino) --> la sintaxis es válida.

if (interino1 instanceof Profesor) --> la sintaxis es válida.

if (fecha1 instanceof Profesor) --> Error: las instancias no están en una jerarquía de tipos

Ejemplo:

Primero la superclase:

```
package herencia;
public class Empleado {
        private String nombre;
        private int numEmpleado , sueldo;

        private static int contador = 0;
         public Empleado(String nombre, int sueldo) {
            this.nombre = nombre;
            this.sueldo = sueldo;
            numEmpleado = ++contador;//Forma de asignar un número de empleado automático cada vez que se crea un nuevo empleado
   }
   public void aumentarSueldo(int porcentaje) {
      sueldo += (int)(sueldo * porcentaje / 100);
   }
   public String toString() {
      return "Num. empleado " + numEmpleado + " Nombre: " + nombre +
            " Sueldo: " + sueldo;
   }
}
```

Ahora la subclase:

```
package herencia;
public class Ejecutivo extends Empleado {

   private int presupuesto;
   public Ejecutivo(String nombre, int sueldo, int presupuesto) {
            super(nombre, sueldo);
            this.presupuesto = presupuesto;
   }
   void asignarPresupuesto(int p) {
      this.presupuesto = p;
   }

   public String toString() {
      String s = super.toString();
      s = s + " Presupuesto: " + presupuesto;
      return s;
   }
}
```

Veamos el main:

```
package herencia;
public class PruebaHerencia {
         public static void main(String[] args) {

                  Empleado emp1= new Empleado ("Ángel Naranjo", 2000);
                  Ejecutivo eje1= new Ejecutivo ("Miguel Campos", 2500, 10000);

                  //Ejemplo reescritura de métodos
                  System.out.println(emp1);
                  System.out.println(eje1);
                  System.out.println("****************************************");
                  //Ejemplo Up-casting

                  Empleado emp = new Ejecutivo("Máximo Dueño" , 3000, 50000);
                  emp.aumentarSueldo(3);
                  // 1. ok. aumentarSueldo es de Empleado emp.asignarPresupuesto(1500);
                  // 2. error de compilación, porque Asignarpresupuesto es de la clase Ejecutivo y emp
```

```
                //es un empleado
                /*Al llamar a toString, el método que resultará llamado es el de la clase Ejecutivo.
                 * toString existe tanto para Empleado como para Ejecutivo,
                 * por lo que el compilador Java no determina en el momento de la compilación que método va a usarse.
                 * Sintácticamente la expresión es correcta. El compilador retrasa la decisión de invocar a un método o a otro
                 * en el momento de la ejecución. Esta técnica se conoce con el nombre de dinamic binding o late binding.
                 * En el momento de la ejecución la JVM comprueba el contenido de la referencia emp.
                 * Si apunta a un objeto de la clase Empleado invocará al método toString de esta clase.
                 * Si apunta a un objeto Ejecutivo invocará por el contrario al método toString de Ejecutivo.*/

                System.out.println(emp.toString());  // se invoca el metodo toString de Ejecutivo
                System.out.println("*********************************************");

                //Operador casting

                /*Si se desea acceder a los métodos de la clase derivada teniendo una referencia de una clase base,
                 * como en el ejemplo del apartado anterior hay que convertir explicitamente la referencia de un tipo a otro.
                 * Esto se hace con el operador de cast de la siguiente forma:
                 */
            Empleado empCasteado = new Ejecutivo("Máximo Dueño" , 2000, 30000);
            Ejecutivo ej = (Ejecutivo)empCasteado; // se convierte la referencia de tipo
            ej.asignarPresupuesto(1500);

/*La expresión de la segunda línea convierte la referencia de tipo Empleado asignándola a una referencia de tipo Ejecutivo.
 * Para el compilador es correcto porque Ejecutivo es una clase derivada de Empleado.
 * En tiempo de ejecución la JVM convertirá la referencia si efectivamente empCasteado apunta a un objeto de la clase Ejecutivo.
 * Si se intenta:
 */
            Empleado emp2 = new Empleado("Luis Miguel López" , 2000);
            Ejecutivo ej2 = (Ejecutivo)emp2;
        /*no dará problemas al compilar, pero al ejecutar se producirá un error porque la referencia
         * emp2 apunta a un objeto de clase Empleado y no a uno de clase Ejecutivo.
         */
            System.out.println("**************************");

            System.out.println(ej2.toString());  //Da error al ejecutar

        }
}
```

NOTA: En la próxima parte de la unidad veremos el polimorfismo con más detenimiento

4.10 EJEMPLO COMPLETO

- En este apartado se muestra un ejemplo sencillo de herencia con comentarios a modo de resumen.

```
public class Animal {
    public int diaNacimiento;
    public int mesNacimiento;
    public int anioNacimiento;

    public Animal() {
    }

    public Animal(int diaNacimiento,
            int mesNacimiento, int anioNacimiento) {
        this.diaNacimiento = diaNacimiento;
        this.mesNacimiento = mesNacimiento;
        this.anioNacimiento = anioNacimiento;
    }

    public void morir() {
        System.out.println("Ay, muero");
    }

    public void imprimeNacimiento() {
        System.out.println("Naci el dia " + diaNacimiento
            + " del " + mesNacimiento + " de " + anioNacimiento);
    }
}
```

- Ahora, declaramos la clase Perro que extiende de la clase Animal. Se usa la palabra **extends.**
- Una clase derivada hereda las variables y métodos de la clase padre, además de añadir sus variables y métodos propios.

```
public class Perro extends Animal {

    public String raza;

    public Perro(String raza) {
        this.raza = raza;
    }

    public void ladrar() {
        System.out.println("GUAU");
    }
}
```

- Probamos las clases en una principal:

```
public class PruebaAnimales {
    public static void main(String[] args) {
        Animal a1 = new Animal();
        Animal a2 = new Animal(5, 7, 1978);
        Perro p = new Perro("Caniche");

        a1.imprimeNacimiento();
        a2.imprimeNacimiento();
        p.imprimeNacimiento();
        p.ladrar();
        p.morir();
        a1.morir();
        a2.morir();
    }
}
```

- Por defecto desde un constructor de una clase hija se llama al constructor sin argumentos de la clase padre.

```
public class Animal {

    public Animal() {
        System.out.println(
        "Soy un animal y estoy naciendo");
    }
}
```

```
public class Perro extends Animal {

    public String raza;

    public Perro(String raza) {
        System.out.println(
            "Soy un perro y estoy naciendo");
        this.raza = raza;
    }
}
```

- De nuevo probamos:

```
public class PruebaAnimales {
    public static void main(String[] args) {
        System.out.println("Paso 1");
        Perro p = new Perro("Caniche");
        System.out.println("Paso 2");
        Animal a1 = new Animal();
        System.out.println("Paso 3");
    }
}
```

- Por defecto desde un constructor de una clase hija se llama al constructor sin argumentos de la clase padre. Si se desea llamar a otro constructor de la clase padre se utiliza la palabra clave **super**. Para mantener el encapsulamiento, una clase derivada debe inicializar sus variables específicas en el constructor, y dejar al constructor del padre inicializar las suyas.

```
public class Animal {
    public int diaNacimiento;
    public int mesNacimiento;
    public int anioNacimiento;

    public Animal() {
        System.out.println(
            "Soy un animal y nazco no se cuando");
    }

    public Animal(int diaNacimiento,
            int mesNacimiento, int anioNacimiento) {
        System.out.println("Soy un animal y nazco el " +
                diaNacimiento + " del " + mesNacimiento
                + " de " + anioNacimiento);
        this.diaNacimiento = diaNacimiento;
        this.mesNacimiento = mesNacimiento;
        this.anioNacimiento = anioNacimiento;
    }
}
```

```
public class Perro extends Animal {

    public String raza;

    public Perro(String raza) {
        System.out.println(
            "Soy un perro y nazco sin fecha");
        this.raza = raza;
    }

    public Perro(int diaNacimiento, int mesNacimiento,
                 int anioNacimiento, String raza) {
        super(diaNacimiento, mesNacimiento, anioNacimiento);
        System.out.println("Soy un perro y nazco el " +
                diaNacimiento + " del " + mesNacimiento
                + " de " + anioNacimiento);
        this.raza = raza;
    }
}
```

```
public class PruebaAnimales {
    public static void main(String[] args) {
        System.out.println("Paso 1");
        Perro p = new Perro("Caniche");
        System.out.println("Paso 2");
        Perro p2 = new Perro(5, 7, 1978, "Fosterrier");
        System.out.println("Paso 3");
    }
}
```

- Una clase hija puede sobrescribir un método de la clase padre para modificar su implementación.

```
public class Animal {
    public Animal() {
    }

    public void nace() {
        System.out.println("Un animal nace");
    }

    public void crece() {
        System.out.println("Un animal crece");
    }
}
```

```
public class Perro extends Animal {

    public Perro() {
    }

    public void nace() {
        System.out.println("Un perro nace");
    }
}

public class PruebaAnimales {
    public static void main(String[] args) {
        Perro p = new Perro();
        p.nace();
        p.crece();
    }
}
```

- No se pueden sobrescribir los métodos de clase (static).
- Si se declara un método de tipo final, no puede ser sobrescrito por clases derivadas.

```java
public class Animal {
    public Animal() {
    }

    public final void nace() {
        System.out.println("Un animal nace");
    }

    public void crece() {
        System.out.println("Un animal crece");
    }
}

public class Perro extends Animal {

    public Perro() {
    }

    // INCORRECTO!!!!!!
    // No se puede sobreescribir un método final
    public void nace() {
        System.out.println("Un perro nace");
    }
}
```