UT4: PROGRAMACIÓN ORIENTADA A OBJETOS

Contenido

1 Introducción

En el tema de Conceptos básicos de la Programación Orientada a Objetos vimos los conceptos: Clase, Atributo (o propiedad), Objeto y Método.

En este tema vamos a presentar y profundizar en otros conceptos que forman parte junto a la Abstracción lo que se ha dado en llamar los 4 pilares de la Orientación a Objetos. Vemos brevemente cuáles son y en qué consisten:

- **Encapsulamiento**: Es la característica de un lenguaje POO que permite que **todo lo referente a un objeto quede aislado dentro de este**. Es decir, que todos los datos referentes a un objeto queden "encerrados" dentro de él y solo se puede acceder a ellos a través de los miembros que la clase proporcione (propiedades y métodos).

- **Abstracción**: La abstracción es la capacidad de **obtener y aislar toda la información y cualidades de un objeto** que nos parezcan relevantes para poder "encapsularlos" en una clase. Es un proceso intelectual que tenemos que hacer para realizar el diseño de nuestras clases.

- **Herencia:** Dado que una clase es un patrón que define cómo es y cómo se comporta una cierta entidad, una clase que hereda de otra obtiene todos los rasgos de la primera y **añade otros nuevos** y además también **puede modificar algunos de los que ha heredado**.

- **Polimorfismo**: Se refiere al hecho de que **varios objetos de diferentes clases, pero con una clase padre/base común, se pueden usar de manera indistinta**, sin tener que saber de qué clase exacta son para poder hacerlo.

2 Encapsulamiento

Al empaquetamiento de las variables de un objeto con la protección de sus métodos se le llama ***encapsulamiento***. Encapsulamiento es el ocultamiento del estado, es decir, de los datos miembro de un objeto, de manera que solo se pueda visualizar/cambiar mediante las operaciones definidas para ese objeto. El encapsulamiento protege a los datos asociados de un objeto contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones.

De esta forma, el usuario de la clase puede obviar la implementación de los métodos y propiedades para concentrarse solo en cómo usarlos. Por otro lado, se evita que el usuario pueda cambiar su estado de manera imprevista e incontrolada.

El encapsulamiento de variables y métodos en un componente de software provee de dos principales ***beneficios*** a los desarrolladores de software:

- **Modularidad**. Esto es, el código fuente de un objeto puede ser escrito, así como darle mantenimiento, independientemente del código fuente de otros objetos.
  La *modularidad* es la capacidad que tiene un sistema de ser estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan para alcanzar un objetivo común, realizando cada una de ellas una tarea necesaria para la consecución de dicho objetivo. Cada una de esas partes en que se encuentre dividido el sistema recibe el nombre de **módulo**.

- **Ocultamiento de la información.** Es decir, un objeto tiene una "interfaz publica" que otros objetos pueden utilizar para comunicarse con él. Pero el objeto puede mantener información y métodos privados que pueden ser cambiados en cualquier momento sin afectar a los demás objetos que dependan de ello.

Los **objetos** tienen la ventaja de la modularidad y el ocultamiento de la información. Las **clases** proveen el beneficio de la **reutilización**. Los programadores de software utilizan la misma clase, y por lo tanto el mismo código, una y otra vez para crear muchos objetos.

En las implementaciones orientadas a objetos se percibe un objeto como un paquete de datos y procedimientos que se pueden llevar a cabo con estos datos. Esto encapsula los datos y los procedimientos.

Los **atributos** se relacionan al **objeto** o instancia y los **métodos** a la **clase**. ¿Por qué se hace así? Los atributos son variables comunes en cada objeto de una clase y cada uno de ellos puede tener un valor para cada variable, diferente al que tienen para esa misma variable los demás objetos.
Los **métodos**, por su parte, pertenecen a la clase y no se almacenan en cada objeto, puesto que sería un desperdicio almacenar el mismo procedimiento varias veces y ello va contra el principio de reutilización de código.

3 Herencia.

La *herencia* es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases, subclases y objetos.

La herencia está fuertemente ligada a la reutilización del código en la POO. Es una de las características fundamentales de la **Programación Orientada a Objetos**.

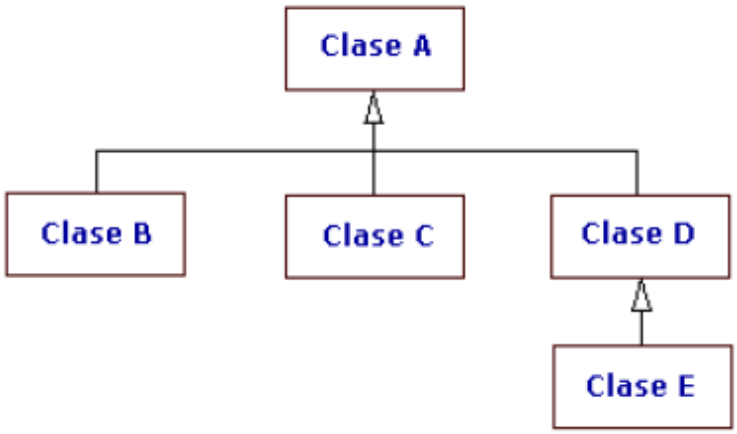
En Java, las clases pueden derivar desde otras clases. La clase derivada (la clase que proviene de otra clase) se llama **subclase**. La clase de la que está derivada se denomina **superclase**.

En Java todas las clases deben derivar de alguna clase. La clase más alta, la clase de la que todas las demás descienden, es la clase **Object**, definida en java.lang. Object es la raíz de la herencia de todas las clases.

Hay dos **tipos de herencia**: Herencia **Simple** y Herencia **Múltiple**. La primera indica que se pueden definir nuevas clases solamente a partir de una clase inicial mientras que la segunda indica que se pueden definir nuevas clases a partir de dos o más clases iniciales. **Java solo permite herencia simple (de forma directa)**.

El concepto de herencia en Java conduce a una estructura jerárquica de clases o estructura de árbol. **En esta estructura jerárquica, cada clase tiene solo una clase padre**. La clase padre de cualquier clase es conocida como su **superclase** (también clase base). La clase hija de una superclase es llamada una **subclase** (también clase derivada).

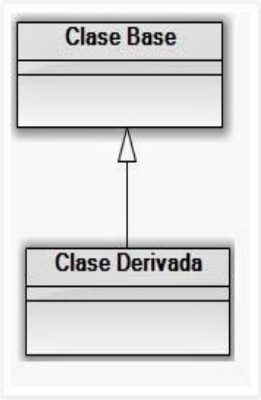
- Una superclase puede tener cualquier número de subclases.
- Una subclase puede tener solo una superclase.



- A es la superclase directa de B, C y D.
- D es la superclase directa de E.
- B, C y D son subclases directas de A.
- E es una subclase directa de D.

Una subclase puede agregar nueva funcionalidad (atributos y métodos) que la superclase no tenía.

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



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

En resumen, la finalidad de la herencia es:

- **Extender** la funcionalidad de la clase base: en la clase derivada se pueden **añadir** atributos y métodos nuevos.

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

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

El código se escribe una vez en la clase base y se utiliza en todas las clases derivadas.
Una clase base puede serlo de tantas derivadas como se desee: Un solo padre, varios hijos.

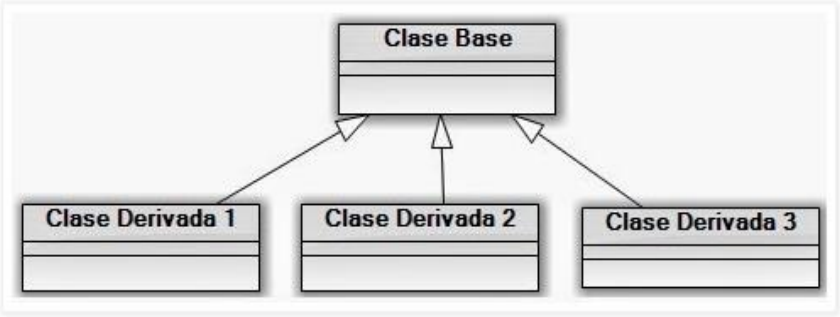
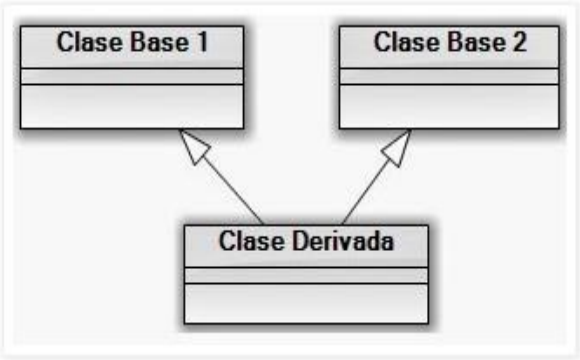


Diagrama UML de herencia múltiple **no permitida** en Java



La herencia expresa una relación **"ES UN/UNA"** entre la clase derivada y la clase base. Esto significa que **un objeto de una clase derivada es también un objeto de su clase base**. **Al contrario, NO es cierto**. Un objeto de la clase base no es un objeto de la clase derivada.

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

Un objeto de una clase derivada es a su vez un objeto de su clase base, por lo tanto, se puede utilizar en cualquier lugar donde aparezca un objeto de la clase base. Si esto no fuese posible entonces la herencia no está bien planteada.

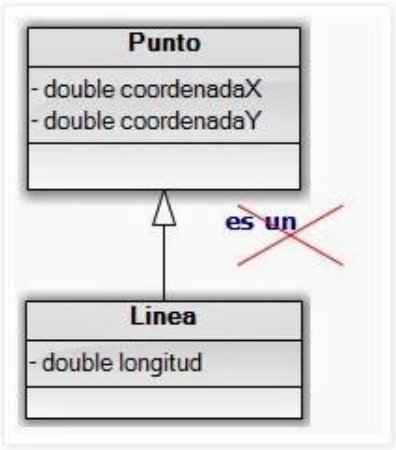
Ejemplo de herencia bien planteada:

A partir de una clase Persona que tiene como atributos el nif y el nombre, podemos obtener una clase derivada Alumno. Un Alumno es una Persona que tendrá como atributos nif, nombre y curso.

Ejemplo de herencia mal planteada:

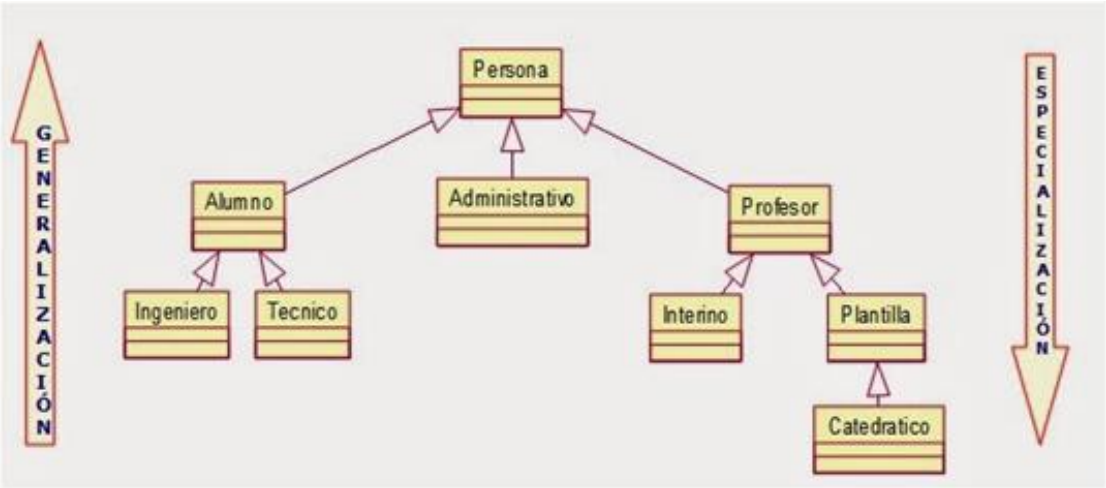
Supongamos una clase **Punto** que tiene como atributos coordenadaX y coordenadaY.

Se puede crear una clase **Linea** a partir de la clase **Punto**. Simplificando mucho para este ejemplo, podemos considerar una línea como un punto de origen y una longitud. En ese caso podríamos crear la Clase Linea como derivada de la clase Punto, pero el planteamiento no es correcto ya que **no se cumple la relación *ES UN*.**



Una Linea NO ES un Punto. En este caso no se debe utilizar la Herencia. Se trataría de una relación de Composición.

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



Las clases más generales se sitúan en lo más alto de la jerarquía. Cuánto más arriba en la jerarquía, menor nivel de detalle.

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

CARACTERÍSTICAS DE LAS CLASES DERIVADAS

Como se ha dicho antes, una clase derivada hereda de la clase base sus componentes (atributos y métodos). Los constructores no se heredan. Las clases derivadas deberán implementar sus propios constructores.

- Una clase derivada **puede acceder a los miembros públicos y protegidos** de la clase base como si fuesen miembros propios.

- Una clase derivada **no tiene acceso a los miembros privados** de la clase base. Deberá acceder a través de métodos públicos heredados de la clase base.

- Si se necesita tener **acceso directo** a los miembros de la clase base se deben declarar **protected** en lugar de private en la clase base.

En resumen, una clase derivada puede **añadir** a los miembros heredados sus propios atributos y métodos (extender la funcionalidad de la clase). También puede **modificar** los métodos heredados (especializar el comportamiento de la clase base). Una clase derivada puede a su vez, ser una clase base, dando lugar a una jerarquía de clases.

Sintaxis de la herencia en Java

La herencia en Java se expresa mediante la palabra **extends.** Para crear una subclase, se incluye la palabra clave **extends** en la declaración de la clase.

```
class nombreSubclase extends nombreSuperclase {
  ..............
}
```

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

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

Ejemplo de herencia Java

Disponemos de una clase **Persona** con los atributos nif y nombre.

```
//Clase Persona. Clase Base

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

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

Creamos un proyecto y añadimos Persona.java, escribiendo el método toString().

Queremos crear ahora una clase **Alumno** con los atributos nif, nombre y curso. Podemos crear la clase Alumno como derivada de la clase Persona. La clase Alumno contendrá solamente el atributo curso. El nombre y el nif son los heredados de Persona:

```
//Clase Alumno. Clase derivada de Persona

public class Alumno extends Persona {
   private String curso;

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

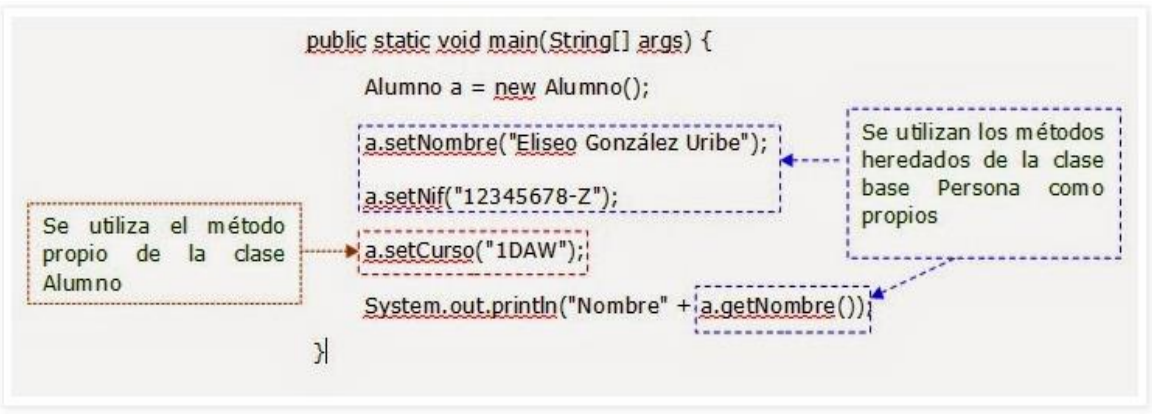
La clase Alumno hereda los atributos nombre y nif de la clase Persona y añade el atributo propio curso. Por lo tanto, a efectos prácticos los atributos de la clase Alumno son nif, nombre y curso.

Los métodos que se pueden usar desde la clase Alumno son: getNif(), setNif(String nif), getNombre(), setNombre(String nombre), getCurso(), setCurso(String curso).

La clase **Alumno,** aunque hereda los atributos nif y nombre, no puede acceder a ellos de forma directa ya que son privados a la clase **Persona**. Se accede a través de los métodos heredados de la clase base.

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

Ejemplo de uso de la clase Alumno (añadimos los componentes que faltan en las clases de ejemplo)**:**



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

1. Se busca en su clase el método correspondiente.
2. Si no se encuentra, se busca en su clase base.
3. Si no se encuentra se sigue buscando hacia arriba en la jerarquía de clases hasta que el método se encuentra.
4. Si al llegar a la clase base raíz y el método no se ha encontrado se producirá un error.

CONSTRUCTORES Y HERENCIA EN JAVA. CONSTRUCTORES EN CLASES DERIVADAS:

Cada clase derivada tendrá sus propios constructores.

- La clase **base** es la encargada de inicializar **sus** atributos.
- La clase **derivada** se encarga de inicializar **solo** los suyos.

Cuando se crea un objeto de una clase derivada se ejecutan los constructores en este orden:

1. Primero se ejecuta el constructor de la clase base
2. Después se ejecuta el constructor de la clase derivada.

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

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

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

Si creamos un objeto Alumno:

Alumno a = new Alumno();

Se muestra por pantalla:

Ejecutando el constructor de Persona
Ejecutando el constructor de Alumno

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

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

La instrucción para invocar al constructor por defecto de la clase base explícitamente es:

super();

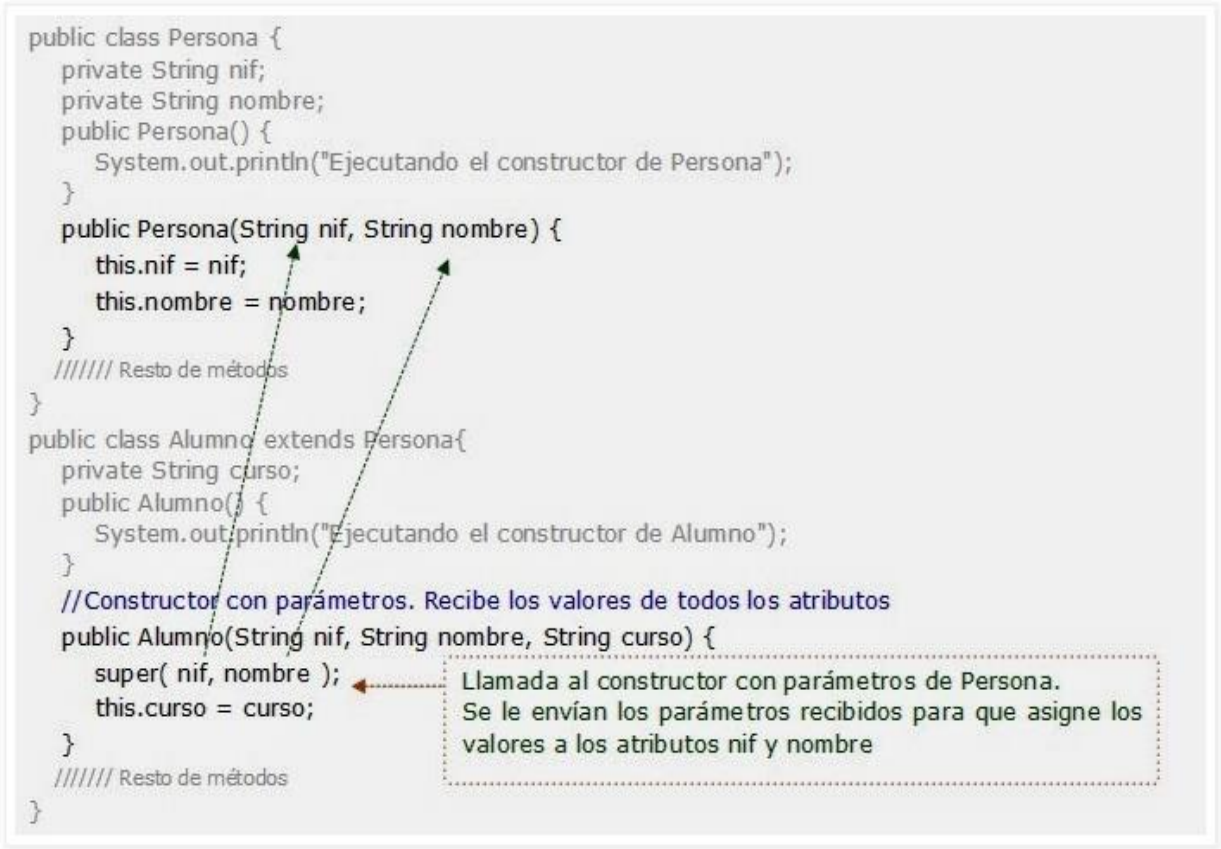
Todos los constructores por defecto en las clases derivadas contienen de forma implícita la instrucción super() como primera instrucción.

```
public Alumno() {
    super(); //esta instrucción se ejecuta siempre. No es necesario escribirla
    System.out.println("Ejecutando el constructor de Alumno");
}
```

Cuando se crea un objeto de la clase derivada y queremos asignar valores a los atributos heredados de la clase base con un constructor, debemos realizar los siguientes pasos:

1. La clase derivada debe tener un constructor con parámetros adecuado que reciba los valores a asignar a los atributos de la clase base.
2. La clase base debe tener un constructor con parámetros adecuado.
3. El constructor de la clase derivada invoca al constructor con parámetros de la clase base y le envía los valores iniciales de los atributos. **Debe ser la primera instrucción.**
4. La clase base es la encargada de asignar valores iniciales a sus atributos.
5. A continuación, el constructor de la clase derivada asigna valores a los atributos de su clase.

Ejemplo: en las clases Persona y Alumno anteriores añadimos constructores con parámetros:



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

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

4 Sobreescritura de métodos

Una subclase hereda todos los métodos de su superclase que son accesibles a dicha subclase a menos que la subclase **sobrescriba los métodos**.

Una **subclase sobrescribe un método de su superclase**, cuando define un método con las mismas características (nombre, número y tipo de argumentos y retorno) que el método de la superclase.

Las subclases emplean la sobreescritura de métodos la mayoría de las veces para agregar o modificar la funcionalidad del método heredado de la clase padre.

Ej:

```
class ClaseA {
        void miMetodo(int var1, int var2){ … }
        String miOtroMetodo(){ … }
}
class ClaseB extends ClaseA {
        //Estos métodos sobreescriben a los métodos de la clase padre
        void miMetodo(int var1 ,int var2){ … }
        String miOtroMetodo(){ … }
}
```

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

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

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

Cuando en una clase derivada se sobrescribe un método de una clase base, se oculta el método de la clase base y todas las sobrecargas del mismo en las clases anteriores.

Si queremos acceder al método de la clase base que ha quedado oculto en la clase derivada utilizamos:

super.metodo();

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

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

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

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

Esto la convierte en clase final. Una clase final no se puede heredar.

Si intentamos crear una clase derivada de la clase final A, se producirá un error de compilación:

```
public class B extends A { //extends A producirá un error de compilación
        ........
}
```

Ejemplo:

Vamos a añadir a la clase Persona que vimos en un apartado anterior un método leer() para introducir por teclado los valores a los atributos de la clase. La clase Persona quedará así:

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

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

La clase Alumno que es derivada de Persona, heredará este método leer() y lo puede usar como propio.

Podemos crear un objeto Alumno:

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

Y utilizar este método:

```
a.leer(sc);
```

Pero utilizando este método solo se leen por teclado el nif y el nombre. En la clase Alumno se debe **sobreescribir** o modificar este método heredado para que también se lea el curso. El método leer() de Alumno invocará al método leer() de Persona para leer el nif y nombre y a continuación se leerá el curso.

La clase Alumno con el método leer() modificado queda así:

```
//Clase Alumno. Clase derivada de Persona

public class Alumno extends Persona{
   private String curso;

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

   @Override  //indica que se modifica un método heredado
   public void leer(Scanner sc){
      super.leer(sc); //se llama al método leer de Persona para leer nif y
nombre
      System.out.print("Curso: ");
      curso = sc.nextLine(); //se lee el curso
   }
}
```

Como se ha dicho antes, cuando en una clase derivada se rescribe un método de una clase base, se oculta el método de la clase base y todas las sobrecargas del mismo en la clase base. Por eso para ejecutar el método leer() de Persona se debe escribir **super.leer(sc);**

Nota: Lo visto es un ejemplo de cómo sobrescribir un método, pero en un caso real sería más adecuado que el método de lectura de datos se hiciera en un sitio diferente de la misma clase, ya que así, el usuario de la clase podría hacer la lectura de los datos de la manera que considere más adecuada (JOptionPane, Scanner, base de datos, fichero…)

Una clase derivada puede volver a declarar un **atributo** heredado (atributo public o protected en la clase base). En este caso, el atributo de la clase base queda oculto por el de la clase derivada.

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

super.atributo;

5 Sobrecarga de métodos y constructores

Existe una característica para tener dos métodos (o constructores) con el mismo nombre. Esta característica se denomina **sobrecarga**.

La **cabecera de un método** es la combinación del tipo de dato que devuelve, su nombre y su lista de argumentos.

> Java diferencia los métodos sobrecargados en base al **número y tipo de argumentos** que tiene el método y **no por el tipo que devuelve**.

También existe la **sobrecarga de constructores**. Cuando en una clase existen varios constructores, se dice que hay sobrecarga de constructores.

Veamos un ejemplo en el caso de una clase llamada Publicacion y cómo los constructores nos dan un ejemplo de sobrecarga de métodos:

```
public class Publicacion {
        public long idPublicacion;
        public String titulo;
        public String autor;
        public static long siguienteId = 0;
        public static String propietario = "Carmen Perez";

        public Publicacion() {
                idPublicacion = siguienteId++;
        }

        public Publicacion(String tituloPublicacion, String autorPublicacion) {
                this();
                titulo = tituloPublicacion;
                autor = autorPublicacion;
        }
//..
}
```

El compilador resolverá qué constructor debe ejecutar en cada momento en función del número de parámetros y su tipo. Si se llama al constructor sin parámetros se ejecutará el primer constructor y en caso de hacerlo con dos parámetros String, se ejecutará el segundo (que, en este caso, llama al constructor sin parámetros).

> **Nota:** El concepto de **sobrecarga de métodos** se puede aplicar siempre que los parámetros sean diferentes, bien por su tipo, bien porque el número de parámetros de un método u otro es diferente.

Ej:

```
int calculaSuma(int x, int y, int z) {
...
}
double calculaSuma(double x, double y, double z) {
...
}
/* Error: los métodos siguientes no están sobrecargados */
int calculaSuma(int x, int y, int z) {
...
}
double calculaSuma(int x, int y, int z) {
...
}
```

Ej:

```
package usuario;
publicclass Usuario {
	String nombre;
	int edad;
	String direccion;
	//El constructor está sobrecargado
	public Usuario() { // no hacen falta esas sentencias
		nombre = null;
		edad = 0;
		direccion = null;
	}
```

```
	public Usuario(String nombre, int edad, String direccion) {
		this.nombre = nombre;
		this.edad = edad;
		this.direccion = direccion;
	}

	public Usuario(Usuario usr) {
		nombre = usr.getNombre();
		edad = usr.getEdad();
		direccion = usr.getDireccion();
	}

	public void setNombre(String n) {
		nombre = n;
	}
	public String getNombre() {
		return nombre;
	}
	//El metodo setEdad() está sobrecargado
	public void setEdad(int e) {
		edad = e;
	}
	public void setEdad(double e) {
		edad = (int)e;
	}
	public int getEdad() {
		return edad;
	}
	public void setDireccion(String d) {
		direccion = d;
	}
	public String getDireccion() {
		return direccion;
	}
	// Añadir un método para imprimir un usuario para eliminarlo del
main
}
```

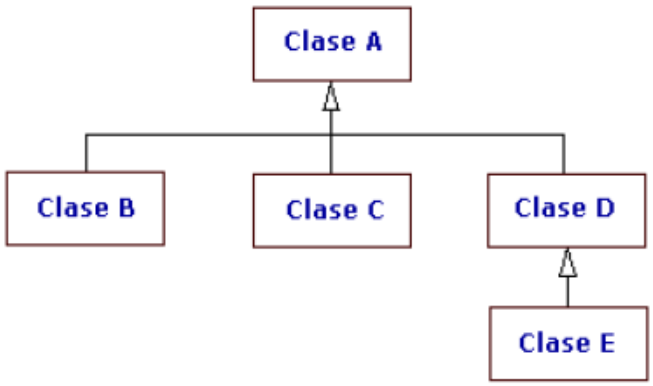
```
package prueba;
import usuario.Usuario;

public class Main{
	void imprimeUsuario(Usuario usr){
		System.out.println("\nNombre: " + usr.getNombre());
		System.out.println("Edad: " + usr.getEdad());
		System.out.println("Direccion: " + usr.getDireccion());
	}
	public static void main(String args[]){
		Main prog = new Main();
		//La otra opción sería que imprimeUsuario fuera static
		Usuario usr1, usr2;

		usr1 = new Usuario();
		prog.imprimeUsuario(usr1);
		usr2 = new Usuario ("Eduardo",24,"Mi direccion");
		prog.imprimeUsuario(usr2);

		usr1 = new Usuario(usr2);
		usr1.setEdad(50);
		usr2.setEdad(30.45);

		prog.imprimeUsuario(usr1);
		prog.imprimeUsuario(usr2);
	}
}
```

6 Polimorfismo

Otro concepto de la POO es el polimorfismo. Un objeto solamente tiene una forma (la que se le asigna cuando se construye ese objeto) pero la referencia al objeto es polimórfica porque puede referirse a objetos de diferentes clases (es decir, la referencia toma múltiples formas).

Para que esto sea posible **debe haber una relación de herencia entre esas clases**. Por ejemplo, considerando la figura que vimos al principio del tema sobre herencia se tiene que:

- Una referencia a un objeto de la clase B también puede ser una referencia a un objeto de la clase A.
- Una referencia a un objeto de la clase C también puede ser una referencia a un objeto de la clase A.
- Una referencia a un objeto de la clase D también puede ser una referencia a un objeto de la clase A.
- Una referencia a un objeto de la clase E también puede ser una referencia a un objeto de la clase D.
- Una referencia a un objeto de la clase E también puede ser una referencia a un objeto de la clase A.



Vemos ejemplo de Futbol como ejemplo de polimorfismo después de abstractas. Está hecho con ArrayList.

7 Clases abstractas

Volviendo a la figura anterior de relación de herencia entre clases, se puede pensar en una jerarquía de clases como la definición de conceptos **más abstractos en lo alto de la jerarquía** y esas ideas se convierten en algo más concreto conforme se desciende por la cadena de la superclase.

Sin embargo, las clases hijas no están limitadas al estado y conducta provistos por sus superclases; **pueden agregar variables y métodos** además de los que ya heredan de sus clases padres. Las clases hijas pueden también, **sobrescribir los métodos** que heredan por implementaciones especializadas para esos métodos. De igual manera, no hay limitación a un solo nivel de herencia por lo que se tiene un árbol de herencia en el que se puede heredar varios niveles hacia abajo y cuantos más niveles se desciende en una clase, más especializada será su conducta.

Las subclases proveen conductas especializadas sobre la base de elementos comunes provistos por la superclase. A través del uso de herencia, los programadores pueden *reutilizar el código de la superclase* muchas veces.

Clases abstractas

Los programadores pueden implementar superclases llamadas **clases abstractas** que definen conductas "genéricas". Las superclases abstractas definen, y pueden implementar parcialmente la conducta, pero parte de la clase no está definida ni implementada. Otros programadores podrán concluir esos detalles con subclases especializadas.

Una clase que declara la existencia de métodos, pero no la implementación de dichos métodos, se considera una clase **abstracta**. Una clase abstracta puede contener métodos no abstractos.

Para declarar una clase o un método como abstractos, se utiliza la palabra reservada **abstract**.

```
abstract class Dibujar {
        abstract void miMetodo(int var1, int var2);
        String miOtroMetodo(){ … }
}
```

Una clase abstracta no se puede instanciar, pero sí se puede heredar de ella y las clases hijas serán las encargadas de agregar la funcionalidad a los métodos abstractos. Si no lo hacen así, las clases hijas deben ser también abstractas.

Hacemos Música con clases abstractas

Ej:

```
Public abstract class FiguraGeometrica {
        . . .
        abstract double Area();
        . . .
}
class Circulo extends FiguraGeometrica {
        double radio;
        double Area() {
                return 3.14*radio*radio;
        }
}
```

Se pueden crear referencias a clases abstractas como cualquier otra. No hay ningún problema en poner:

```
FiguraGeometrica figura;
```

Una clase abstracta es una clase que NO se puede instanciar, es decir, no se pueden crear objetos de esa clase. El compilador producirá un error si se intenta:

```
FiguraGeometrica figura = new FiguraGeometrica();
```

Esto es coherente dado que una clase abstracta no tiene completa su implementación y encaja bien con la idea de que algo abstracto no puede materializarse.

Sin embargo, utilizando el **up-casting*** se puede escribir:

```
FiguraGeometrica figura = new Circulo(. . .);
figura.Area();
```

(*) **Es la operación en que un objeto de una clase derivada se asigna a una referencia cuyo tipo es alguna de las superclases**. Cuando se realiza este tipo de operaciones, hay que tener cuidado porque la referencia es a los miembros de la clase hija, aunque la referencia sea de tipo la clase padre.

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

Las clases abstractas:

- Pueden contener cero o más métodos abstractos.
- Pueden contener métodos no abstractos.
- Pueden contener atributos.

Las clases abstractas se diseñan solo para que otras clases hereden de ellas. Todas las clases no abstractas que hereden de una clase abstracta deben implementar todos los métodos abstractos heredados.

Si una clase derivada de una clase abstracta no implementa algún método abstracto se convierte en abstracta y tendrá que declararse como tal (tanto la clase como los métodos que siguen siendo abstractos).

Aunque no se pueden crear objetos de una clase abstracta, **sí pueden tener constructores** para inicializar sus atributos. Estos constructores serán invocados cuando se creen objetos de clases derivadas.

La forma general de declarar una clase abstracta en Java es:

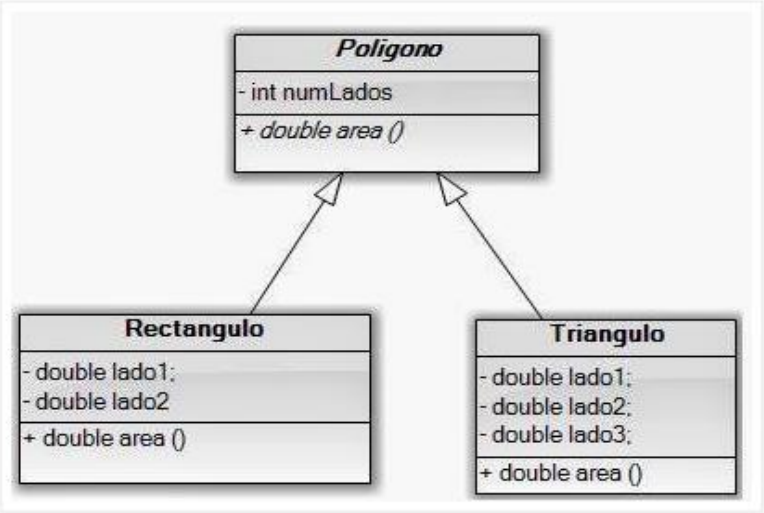
```
[modificador] abstract class nombreClase {
}
```

Ejemplo de clase Abstracta en Java: Clase Polígono.

```
//Clase abstracta Poligono

public abstract class Poligono {

   private int numLados;

   public Poligono() {}

   public Poligono(int numLados) {
      this.numLados = numLados;
   }

   public int getNumLados() {
      return numLados;
   }

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

 //Declaración del método abstracto area()
   public abstract double area();
}
```

La clase Poligono contiene un único atributo *numLados*. Es una clase abstracta porque contiene el método abstracto area().
A partir de la clase Poligono vamos a crear dos clases derivadas Rectangulo y Triangulo. Ambas deberán implementar el método area(), de lo contrario también serían clases abstractas.

El diagrama UML es el siguiente:



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

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

   private double lado1;
   private double lado2;

   public Rectangulo() {
   }

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

   public double getLado1() {
      return lado1;
   }

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

```
  public double getLado2() {
      return lado2;
   }

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


   //Implementación del método abstracto area()
   //heredado de la clase Polígono
   @Override
   public double area(){
      return lado1 * lado2;
   }
}
```

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

   private double lado1;
   private double lado2;
   private double lado3;

   public Triangulo() {
   }

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


   public double getLado1() {
      return lado1;
   }

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

   public double getLado2() {
      return lado2;
   }

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

```
  public double getLado3() {
      return lado3;
   }

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


 //Implementación del método abstracto area()
   //heredado de la clase Polígono
   @Override
   public double area(){
       double p = (lado1+lado2+lado3)/2;

       return Math.sqrt(p * (p-lado1) * (p-lado2) * (p-lado3));
   }
}
```

Ejemplo de uso de las clases:

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

      Triangulo t = new Triangulo(3.25, 4.55, 2.71);
      System.out.printf("Área del triángulo: %.2f %n", t.area());

      Rectangulo r = new Rectangulo(5.70,2.29);
      System.out.printf("Área del rectángulo: %.2f %n", r.area());

}
```

8 Interfaces (se ampliará en otro documento aparte)

Una interface es una variante de una clase abstracta con la condición de que todos sus métodos deben ser abstractos (lo serán, aunque no se especifique) y públicos. Si la interface va a tener atributos, estos serán **public, static** y **final** (por defecto) y con un valor inicial ya que funcionan como constantes.

La clase que implementa una interface tiene dos opciones:

1) Implementar todos los métodos de la interface.
2) Implementar solo algunos de los métodos de la interface, pero esa clase entonces debe ser una clase abstracta.

9 Control de acceso a miembros de una clase

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

Ver ejemplo de Empleados y ejercicio Operaciones con paquetes.

El significado concreto del modificador de acceso protected varía ligeramente entre Java y C++:

- En C++, un atributo que sea declarado como "protected" en una clase será visible únicamente en dicha clase y en las subclases de la misma.
- En Java, un atributo que sea declarado como "protected" será visible en dicha clase, en el "package" que se encuentre la misma, y en las subclases de la misma (aunque no estén en el mismo paquete).

10 Casting: conversiones entre clases

UpCasting: Conversiones implícitas

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

Por esta razón:

Se puede asignar de forma implícita una referencia a un objeto de una clase derivada a una referencia de la clase base. Son ***tipos compatibles****.*
También se llaman conversiones ascendentes o ***upcasting****.*

En el ejemplo anterior un triángulo es un Poligono y un cuadrado es un Poligono.

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

Como un triángulo es un polígono, se puede hacer esta asignación:

p=t;

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

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

En el ejemplo, la instrucción:

p.getLado1();

Provocará un error, ya que p es de tipo Poligono y el método getLado1() no es un método de esa clase.

Cuando manejamos un objeto a **través de una referencia a una superclase** (directa o indirecta) y **se invoca a un método que está redefinido en las subclases se ejecuta el método de la clase a la que pertenece el objeto no el de la referencia (por ejemplo, el caso del Milpies que vimos en herencia)**.

En el ejemplo, la instrucción (después de haber hecho lo anterior):

p.area();

Ejecutará el método area() de Triángulo.

DownCasting: Conversiones explícitas

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

También se llaman conversiones descendentes o **downcasting**.
Esta conversión debe hacerse mediante un **casting explícito**.

Siguiendo con el ejemplo anterior:

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

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

Esta asignación se puede hacer porque p contiene la referencia de un objeto triángulo.

Las siguientes instrucciones provocarán un error de ejecución del tipo **ClassCastException**:

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

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

EL OPERADOR instanceof

Las operaciones entre clases y en particular el downcasting requieren que las clases sean de tipos compatibles. Para asegurarnos de ello podemos utilizar el operador **instanceof**.

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

La sintaxis es:

```
Objeto instanceof Clase
```

Ejemplo:

```
Triangulo t;
Poligono p1 = new Rectangulo(3,2);

If (p1 instanceof Triangulo)
        t = (Triangulo)p1;
else
        System.out.println("Objetos incompatibles");
```