Utilización avanzada de clases (I): herencia

Contenido

Herencia

La **herencia** es uno de los principios fundamentales de la programación orientada a objetos.

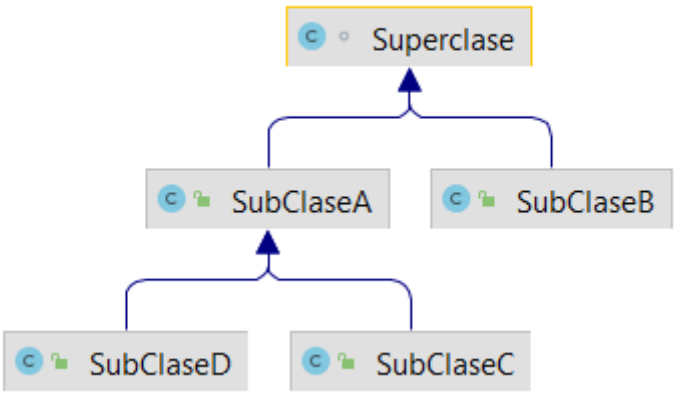
Es el mecanismo por el cual se derivan o extienden clases (clases derivadas, clases hijas o subclases) a partir de una clase (la clase base, clase padre o superclase), de forma que las clases derivadas heredan atributos y métodos de la superclase.

De esta forma podemos desarrollar **jerarquías** de clases y **reutilizar** gran cantidad de código.

Para derivar una clase a partir de otra se utiliza la **palabra clave `extends`**, con la siguiente sintaxis:

```
class SuperClase { }

class SubClaseA extends SuperClase { }

class SubClaseB extends SuperClase { }

class SubClaseC extends SubClaseA { }

class SubClaseD extends SubClaseA { }
```

En el ejemplo anterior tenemos una clase padre de la cual heredan tres clases hijas. Esto se puede representar así:



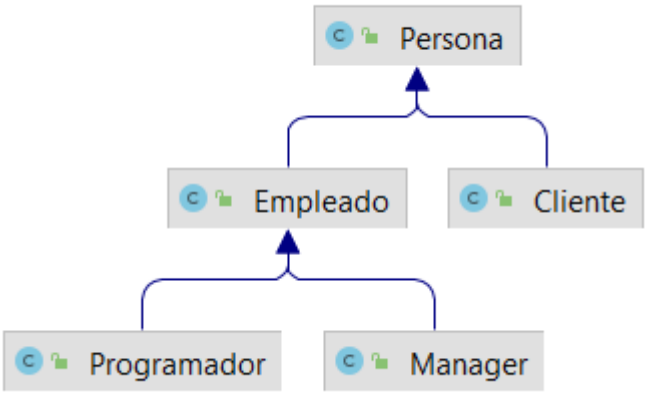
Algunas características particulares de la herencia en Java son las siguientes:

- Java no soporta herencia múltiple: una clase hija sólo puede heredar de una única superclase.
- Una jerarquía de clases puede tener múltiples niveles (en el ejemplo anterior, las subclases C y D heredan de la subclase A, que a su vez hereda de la clase SuperClase.
- Una superclase puede tener más de una subclase.

- Una subclase hereda todos los atributos y métodos protected y public de la superclase, y, aparte, la superclase puede tener atributos y métodos propios (private y default package).
    - Una subclase no hereda los atributos y métodos privados de la superclase.
    - Si queremos hacer que los miembros de la superclase sean accesibles desde todas las subclases pero no desde otras clases (excepto las que se encuentren en el mismo paquete) utilizaremos el modificador de acceso protected.
- Los constructores no se heredan, pero el constructor de la superclase puede ser invocado desde la subclase utilizando la palabra super().

La herencia representa una relación ES-UN, en la que la superclase es el caso general y cada subclase es un caso específico.

Veamos un ejemplo más desarrollado: una empresa de telecomunicaciones proporciona servicios a sus clientes. Tiene una plantilla pequeña compuesta sólo por managers y programadores.



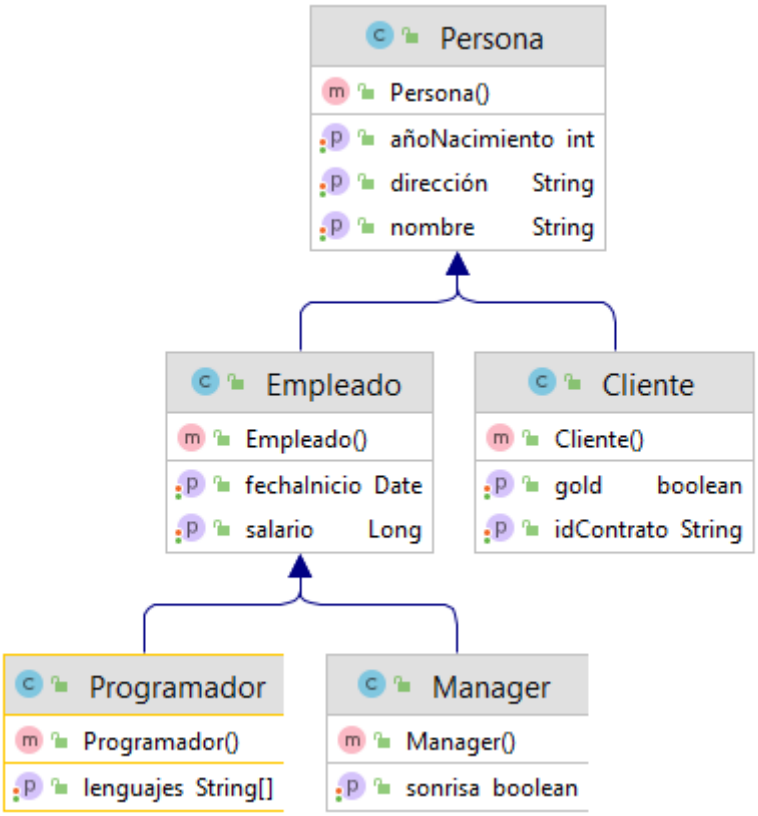
- La superclase Persona tiene atributos para almacenar datos comunes a todas las personas: nombre, año de nacimiento y dirección.
- La clase Cliente tiene atributos adicionales para almacenar número de contrato y status (oro o no),
- La clase Empleado tiene atributos adicionales para almacenar la fecha en que empezó a trabajar en la compañía y el salario.
- La clase Programador tiene un array con los lenguajes de programación que utiliza.
- La clase Manager puede tener (o no) una sonrisa deslumbrante.

Veamos el código:

```
class Persona {
    protected String nombre;
    protected int añoNacimiento;
    protected String dirección;
```

```
    // aquí irían los getters y setters public para todos los atributos
}

class Cliente extends Persona {
    protected String idContrato;
    protected boolean gold;

    // aquí irían los getters y setters public para todos los atributos
}

class Empleado extends Persona {
    protected Date fechaInicio;
    protected Long salario;

    // aquí irían los getters y setters public para todos los atributos
}



class Programador extends Empleado {
    protected String[] lenguajes;

    public String[] getLenguajes() {
        return lenguajes;
    }

    public void setLenguajes(String[] lenguajes) {
        this. lenguajes = lenguajes;
    }
}

class Manager extends Empleado {
    protected boolean sonrisa;

    public boolean isSonrisa() {
        return sonrisa;
    }

    public void setSonrisa(boolean sonrisa) {
        this.sonrisa = sonrisa;
    }
}
```

Esta jerarquía tiene dos niveles y cinco clases en total. Todos los atributos son protected, lo que significa que son visibles para las subclases.



Vamos a crear un objeto de tipo Programador y a rellenar los atributos heredados utilizando los setters heredados. Para leer los valores de los atributos podemos utilizar getters heredados también.

```
public class Main {
    public static void main(String[] args) {
        Programador p = new Programador();

        p.setNombre("A. Arenal");
        p.setAñoNacimiento(1985);
        p.setDirección("Pobladura del Valle, 15");
        p.setFechaInicio(new Date());
        p.setSalario(50_000L);
        p.setLenguajes(new String[] { "Java", "JavaScript", "PHP" });

        System.out.println(p.getNombre()); // A. Arenal
        System.out.println(p.getSalario()); // 50000
        System.out.println(Arrays.toString(p.getLenguajes())); // [Java,
JavaScript, PHP]

    }
}
```

La palabra clave super

Sirve para acceder a miembros (atributos o métodos) o constructores de la superclase desde una subclase. Vamos a verlo con ejemplos.

Cómo acceder a los atributos y métodos de la superclase

En este sentido, la palabra clave **`super`** es análoga a `this`. Si **`this`** sirve para referirnos a un elemento de "esta" clase, **`super`** sirve para referirnos a un elemento de la clase inmediatamente superior.

La palabra clave `super` es opcional si los miembros de una subclase tienen nombres distintos de los de la superclase, pero, si los miembros se llaman igual en la subclase y en la superclase, la única forma de distinguirlos es utilizando super y this.

```
class SuperClase {

    protected int atributo;

    protected int getAtributo() {
        return atributo;
    }

    protected void printValorSuper() {
        System.out.println(atributo);
    }
}

class SubClase extends SuperClase {

    protected int atributo;

    public SubClase() {
        this.atributo = 30;  // Inicializa la variable en la SubClase
        atributo = 30;   // También inicializa la variable en la SubClase
        super.atributo = 20; // Inicializa la variable en la SuperClase
    }

    /**
     * Imprime el valor de SuperClase y luego el valor de SubClase
     */
    public void printValorSub() {
        super.printValorSuper (); /* Invoca al método de la
SuperClase, super aquí es opcional */
        System.out.println(atributo);
    }
}
```

Cómo invocar al constructor de la superclase

Las subclases no heredan los constructores de su superclase, pero…

El constructor por defecto de una subclase automáticamente invoca al constructor por defecto de la superclase.

Si no se utiliza el constructor por defecto, un constructor de la superclase puede ser invocados desde la subclase utilizando la palabra clave `super` seguida de paréntesis, con una particularidad: si se utiliza, **`super(…)`** debe ser la primera instrucción en la subclase; si no, no compila.

Por ejemplo, tenemos dos clases, una hereda de la otra y ambas tienen un constructor para inicializar los atributos.

```
class Persona {

    protected String nombre;
    protected int añoNacimiento;
    protected String dirección;

    public Persona(String nombre, int añoNacimiento, String dirección{
        this.nombre = nombre;
        this.añoNacimiento = añoNacimiento;
        this.dirección = dirección;
    }

    // getters y setters
}

class Empleado extends Persona {

    protected Date fechaInicio;
    protected Long salario;

    public Empleado(String nombre, int añoNacimiento, String dirección, Date f
echaInicio, Long salario) {
        super(nombre, añoNacimiento, dirección); /* invoca al constructor de l
a superclase */

        this.fechaInicio = fechaInicio;
        this.salario = salario;
    }

    // getters y setters
}
```

En este ejemplo, el constructor de la clase Empleado invoca al constructor de la clase padre para asignar valores a los atributos pasados como parámetro.

Sobreescritura (override)

Hablamos de sobreescritura cuando en una subclase declaramos un método con el mismo nombre que otro en la superclase, pero cambiamos su desarrollo.

Veamos un ejemplo de sobreescritura:

```
class Mamífero {

    public String saludar() {
        return "el mamífero te saluda";
    }
}

class Gato extends Mamífero {

    @Override
    public String saludar() {
        return "miau";
    }
}

class Humano extends Mamífero {

    @Override
    public String saludar() {
        return "hola";
    }
}
```

Esta jerarquía incluye tres clases. La clase Mamífero tiene el método saludar(). Cada subclase lo sobreescribe a su propia manera.

La anotación `@Override` es opcional, indica que se trata de un método sobreescrito y comprueba si el método de la superclase puede ser sobreescrito (si cumple las reglas); en caso de que no se pueda, generará un error.

Vamos a crear instancias y a invocar al método:

```
Mamífero mamífero = new Mamífero();
System.out.println(mamífero.saludar());//imprime "el mamífero te saluda"

Gato gato = new Gato();
System.out.println(gato.saludar()); // imprime "miau"

Humano humano = new Humano();
System.out.println(humano.saludar()); // imprime "hola"
```

De todas formas, aunque sobreescribamos un método en una clase hija, siempre podremos invocar al método sobreescrito de la superclase utilizando la palabra clave super.

Reglas de la sobreescritura

- El método debe tener el mismo nombre que el de la superclase.
- Los argumentos deben ser exactamente iguales que en la superclase.

- El tipo de retorno debe ser del mismo tipo o de un subtipo que el del método de la superclase.
- El nivel de acceso debe ser el mismo o más abierto que el nivel de acceso en el método sobreescrito.
    - Si superclase y subclase están en el mismo paquete, se pueden sobreescribir métodos con el modificador de acceso por defecto.
- Los métodos estáticos no pueden sobreescribirse, sólo los de instancia.

Cómo prohibir la sobreescritura

Declarando el método con la palabra clave **`final`**, si tratamos de sobreescribirlo en una clase hija dará error de compilación.

```
public final void método() {
    // lo que sea
}
```

Sobreescribir y sobrecargar métodos en una subclase

Recordemos que sobrecarga significa que en una clase haya más de un método con el mismo nombre, siempre que los parámetros sean diferentes.

Para que haya sobreescritura debe tratarse de métodos en distintas clases: una subclase y una superclase, y los parámetros y tipo de retorno deben ser iguales.

Podemos sobreescribir y sobrecargar un método en una subclase, pero en este caso los métodos sobrecargados no sobreescriben los métodos de la superclase, son métodos nuevos, únicos para la subclase.

Mostrémoslo con un ejemplo:

```
class SuperClase {

    public void método() {
        System.out.println("método en la SuperClase");
    }
}

class SubClase extends SuperClase {

    @Override
    public void método() {
        System.out.println("SubClase: el método se sobreescribe");
    }

    // @Override – no sobreescribimos nada
    public void método(String s) {
        System.out.println("SubClase: el método se sobrecarga");
    }
}
```

El siguiente código crea una instancia e invoca ambos métodos:

```
SubClase clase = new SubClase();
```

```
clase.método();    // SubClase: el método se sobreescribe
clase.método("s"); // SubClase: el método se sobrecarga
```

Ocultar métodos estáticos

Los metodos estáticos, como hemos dicho, no pueden sobreescribirse. Si una subclase tiene un método estático con la misma signatura (nombre y parámetros) que un método estático de la superclase, lo que sucede es que el método de la subclase oculta al de la superclase y no habrá forma de acceder a él, lo cual es completamente diferente de la sobreescritura.

Si una subclase tiene un método estático con la misma signatura que un método de instancia de la superclase o viceversa, saldrá un error de compilación. Pero si los métodos tienen el mismo nombre pero diferentes parámetros no debería haber problema.

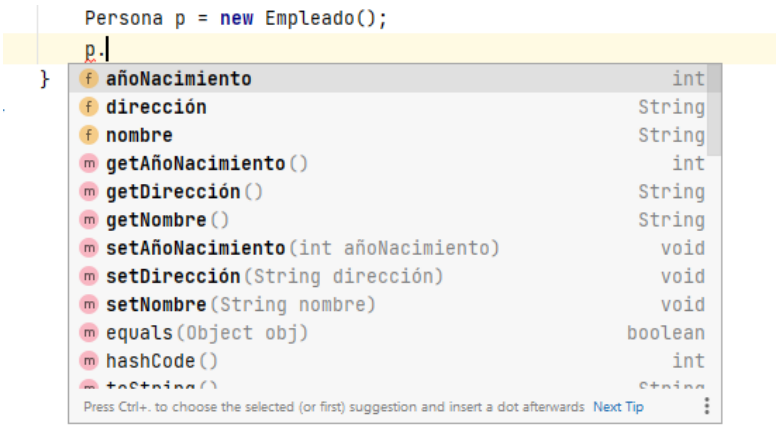
Cómo crear objetos de subclases

Cuando definimos una clase como subclase de otra**, los objetos de la subclase son también objetos de la superclase**, ya que posee todos los miembros de ésta. Por eso podemos declarar un objeto de la subclase como una variable del tipo de la superclase.

Por ejemplo:

```
Persona cliente = new Cliente(); // referencia tipo Persona, objeto tipo Cliente
Persona p = new Empleado(); // referencia tipo Persona, objeto tipo Empleado
```

¿Y da igual crear un Empleado declarándolo como Empleado que como Persona? No exactamente. Cuando declaramos un objeto con la superclase sólo serán visible los miembros (atributos y métodos) definidos en la superclase.




```
Persona p = new Empleado();

p.setNombre("Z. Zambrano"); // Ok
p.setAñoNacimiento(1980); // Ok
p.setSalario(30000); /* Error de compilación, la superclase no conoce
este método */
```

Ahora bien, ¿qué pasaría si en la subclase tuviéramos declarados atributos o métodos iguales que en la superclase? En nuestro caso, esto seria así:

```
public class Persona {
    protected String nombre;
    protected int añoNacimiento;
    protected String dirección;

    public void mostrarDatos() {
        System.out.println("Datos de la persona");
    }
...

public class Empleado extends Persona{
    protected Date fechaInicio;
    protected Long salario;
    protected String nombre; // también está en Persona
    protected String añoNacimiento; // también está en Persona, pero
como int
    protected String dirección; // también está en Persona

    public void mostrarDatos() {
        System.out.println("Datos del empleado"); // sobreescribimos
el método
    }
...
```

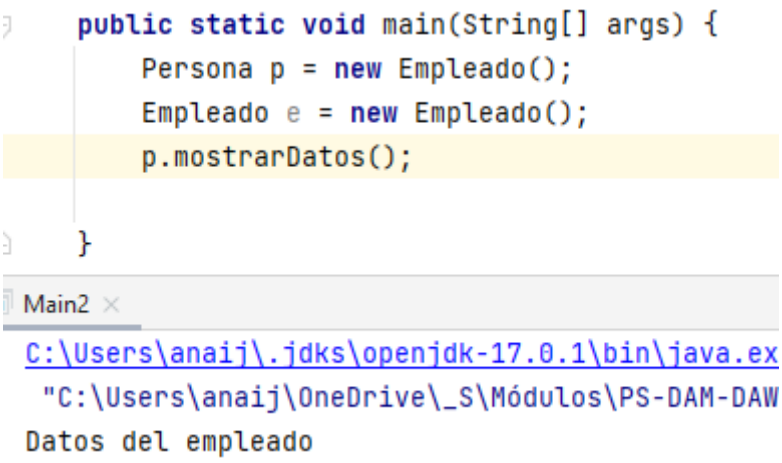
En este caso, cuando declaramos

```
Persona p = new Empleado();
```

Al invocar después atributos y métodos del objeto p, ¿a qué versión accederemos, a la especificada en la subclase o a la de la superclase? Pues depende.

- Atributos: accederemos a los de la superclase.

- Métodos: aquí sucede al contrario; se ejecuta la versión de la subclase. Por tanto, aquí funciona la sobreescritura.



Algunas cosas a recordar:

- No se puede asignar un objeto de una subclase a la referencia de otra subclase porque no hay herencia entre ellos:

```
Cliente quiénEsEste = new Empleado(); // esto no se puede hacer
```

- Tampoco se puede asignar un objeto de la clase padre a la referencia de una subclase suya:

```
Cliente cliente = new Persona(); // esto tampoco se puede hacer
```

Conversión de tipos (casting)

Lo que sí se puede hacer es una **conversión de tipos (casting)** de una superclase a una de sus subclases, pero sólo si el objeto es realmente una instancia de esta subclase. En caso contrario saldrá una excepción ClassCastException.

```
Persona persona = new Cliente();

Cliente cliente2 = (Cliente) persona; // ok

Empleado empleado = (Empleado) persona; // ClassCastException
```

Siempre se puede hacer un casting de un objeto de una subclase a su superclase.

```
Empleado e = new Empleado();
Persona ep = (Persona) e;
```

¿Cuándo utilizar la referencia a la superclase?

Hay dos casos habituales:

- Al procesar un array (u otro tipo de colección) de objetos que tienen diferentes tipos de la misma jerarquía;
- Un método que que acepta un objeto de la superclase, pero también puede operar con objetos de sus subclases.

Vamos a ver los dos casos en un único ejemplo. Tenemos un método imprimirNombres que recibe un array de tipo Persona y muestra sus nombres:

```
public static void imprimirNombres(Persona[] personas) {
    for (Persona persona : personas) {
        System.out.println(persona.getNombre());
    }
}
```

Este método funcionará con arrays que contengan objetos Persona, Cliente y Empleado.

```
Persona persona = new Empleado();
persona.setNombre("G. Gutiérrez");

Cliente cliente = new Cliente();
cliente.setNombre("P. Paredes");

Empleado empleado = new Empleado();
empleado.setNombre("J. Jiménez");

Persona[] personas = {persona, cliente, empleado};

imprimirNombres(personas);
```

El resultado es exactamente el esperado:

```
G. Gutiérrez
P. Paredes
J. Jiménez
```

Polimorfismo

Etimológicamente polimorfismo quiere decir "muchas formas". Java proporciona dos tipos de polimorfismo: estático (en tiempo de compilación) y dinámico (en tiempo de ejecución). El primero se consigue mediante la sobrecarga de métodos; el segundo se basa en la herencia y en la sobreescritura de métodos.

En este tema vamos a considerar solamente el polimorfismo en tiempo de ejecución, ampliamente utilizando en la programación orientada a objetos.

Polimorfismo en tiempo de ejecución

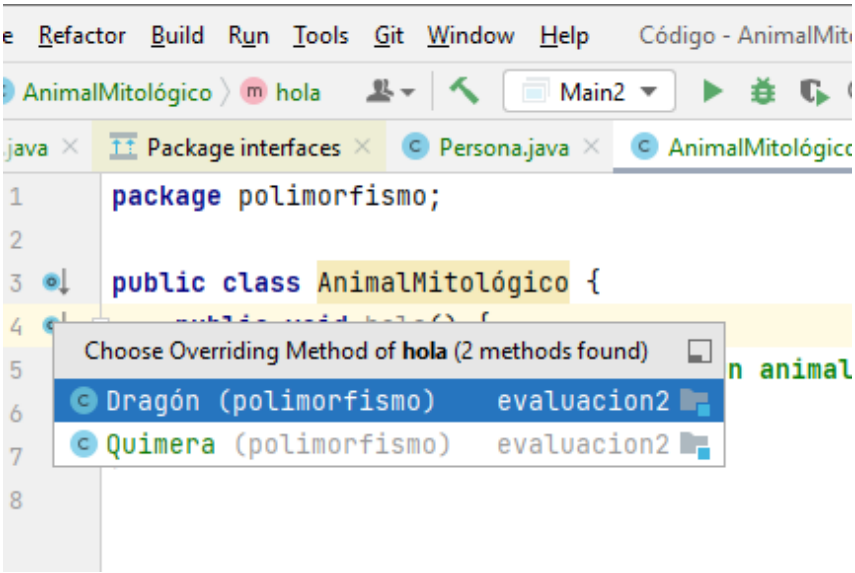
Recordemos que **sobreescritura de métodos** es lo que sucede cuando una subclase redefine un método que está en su superclase.

El polimorfismo en tiempo de ejecución opera cuando **se invoca un método sobreescrito utilizando la variable referencia de una superclase**. Java determina en tiempo de ejecución qué versión del método (superclase o subclase) tiene que ejecutarse en función de tipo del objeto referenciado, no del tipo de la referencia. Utiliza un mecanismo conocido como **resolución dinámica de métodos**.

Por ejemplo, tenemos una jerarquía de clases. La superclase AnimalMitológico tiene dos subclases: Quimera y Dragón. La superclase tiene un método hola. Ambas subclases lo sobreescriben.

```
class AnimalMitológico {

    public void hola() {
        System.out.println("Hola, soy un animal desconocido");
    }
}

class Quimera extends AnimalMitológico {
    @Override
    public void hola() {
        System.out.println("Hola, soy una Quimera");
    }
}

class Dragón extends AnimalMitológico {
    @Override
    public void hola() {
        System.out.println("Grrr...");
    }
}
```

NOTA: IntelliJ IDEA nos da muy buenas pistas cuando tenemos polimorfismo:



Podemos crear una referencia a la clase AnimalMitológico y asignarla al objeto de la subclase, y después invocar a los métodos sobreescritos a través de las referecias de la superclase:

```
AnimalMitológico quimera = new Quimera();
AnimalMitológico dragón = new Dragón();
AnimalMitológico animal = new AnimalMitológico();

quimera.hola(); // Hola, soy una Quimera
dragón.hola(); // Grrr...
animal.hola(); // Hola, soy un animal desconocido
```

De esta forma, el resultado de la invocación de un método depende del tipo real de instancia, no del tipo referencia. Es una característica del polimorfismo en Java. La máquina virtual (JVM) llama al método apropiado para el objeto que es referenciado en cada variable.

Polimorfismo dentro de una jerarquía de clases

Lo mismo sucede con métodos que se utilizan sólo dentro de una jerarquía y no son accesibles desde el exterior.

Por ejemplo, tenemos una jerarquía de ficheros. La clase padre Fichero representa una descripción de un único fichero en el sistema. Tiene una subclase llamada Imagen que sobreescribe el método getInfoFichero de la clase padre.

```
class Fichero {

    protected String nombreCompleto;

    // constructor con un parámetro

    // getters y setters

    public void imprimeInfoFichero() {
        String info = this.getInfoFichero(); // comportamiento polimórfico
        System.out.println(info);
    }

    protected String getInfoFichero() {
        return "Fichero: " + nombreCompleto;
    }
}

class Imagen extends Fichero {

    protected int width;
    protected int height;
    protected byte[] contenido;

    // constructor

    // getters y setters

    @Override
    protected String getInfoFichero() {
        return String.format("Imagen: %s, width: %d, height: %d",
nombreCompleto, width, height);
    }
}
```

La clase padre tiene un método público imprimeInfoFichero y un método protected getInfoFichero. El segundo método se sobreescribe en la sublcase, pero la subclase no sobreescribe el primer método.

Vamos a crear una instancia de Imagen y asignarlo a la variable de Fichero.

```
Fichero img = new Imagen("\\ruta\\al\\fichero\\img.png", 480, 640, new byte[]{4, 32});// aquí asignamos un objeto
```

Si ahora llamamos al método imprimeInfoFichero, lo que se invoca es la versión sobreescrita del método getInfoFichero.

```
img.imprimeInfoFichero();

// Imprime "Imagen: \ruta\al\fichero\img.png, width: 480, height: 640"
```

De esta forma, el polimorfismo en tiempo de ejecución permite invocar un método sobreescrito de una subclase teniendo una referencia a la superclase.

La clase Object

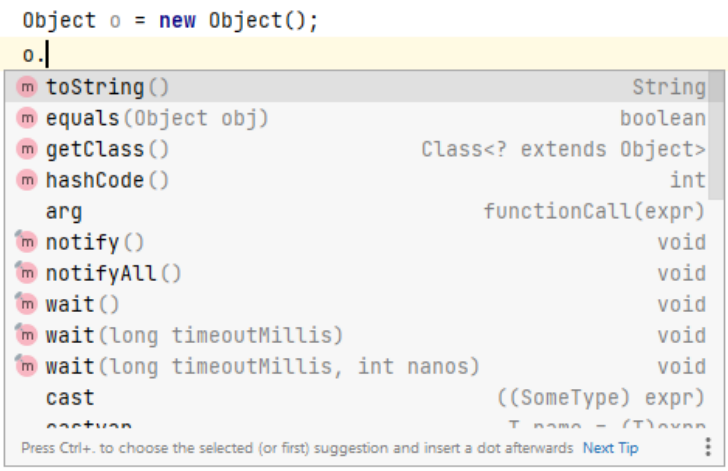
La clase Object del paquete java.lang (este paquete se importa siempre por defecto en los progamas de Java, sin necesidad de escribir import) es una clase de la que heredan todas las clases de Java, incluidas las que creamos nosotros. Es la clase raíz en la cúsplide de la jerarquía de herencia en los programas de Java.

Cuando declaramos una clase en realidad lo que estamos haciendo es extender implícitamente la clase Object.

```
class A extends Object { }
```

De esta forma cualquier clase dispone de una serie de métodos propocionados por Object que son útiles en ocasiones.

Métodos que proporciona la clase Object



toString()

Devuelve un String que representa al objeto que lo invoca con su información.

Su implementación por defecto no suele ser útil porque devuelve el nombre cualificado de la clase, una arroba y la referencia del objeto. Por ejemplo:

```
package teleco;

public class Main2 {
    public static void main(String[] args) {

        Empleado e = new Empleado();
        System.out.println(e.toString());

    }
```

```
Main2
C:\Users\anaij\.jdks\openjdk-17.0.1\bin\java
 "C:\Users\anaij\OneDrive\_S\Módulos\PS-DAM-
teleco.Empleado@1b28cdfa
```

Por eso lo que se suele hacer es sobreescribirlo en cada clase para que muestre la información que queremos representar. Así, en la clase Empleado añadiríamos:

```
@Override
public String toString() {
    return "Empleado{" +
            "fechaInicio=" + fechaInicio +
            ", salario=" + salario +
            ", nombre='" + nombre + '\'' +
            ", añoNacimiento='" + añoNacimiento + '\'' +
            ", dirección='" + dirección + '\'' +
            '}';
}
```

Y entonces al invocarlo tendríamos este resultado:

```
package teleco;

public class Main2 {
    public static void main(String[] args) {

        Empleado e = new Empleado();
        System.out.println(e.toString());
    }
}
```

```
Main2
C:\Users\anaij\.jdks\openjdk-17.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ
 "C:\Users\anaij\OneDrive\_S\Módulos\PS-DAM-DAW-1°-MP03 - Programación\PROG - Curso 21-22\Código\
Empleado{fechaInicio=null, salario=null, nombre='null', añoNacimiento='null', dirección='null'}
```

En realidad, System.out.println() invoca por defecto el método toString(). Por tanto, sólo será necesario escribir:

```java
package teleco;

public class Main2 {
    public static void main(String[] args) {

        Empleado e = new Empleado();
        System.out.println(e);
    }
}
```

```
Main2
C:\Users\anaij\.jdks\openjdk-17.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ
 "C:\Users\anaij\OneDrive\_S\Módulos\PS-DAM-DAW-1°-MP03 - Programación\PROG - Curso 21-22\Código\
Empleado{fechaInicio=null, salario=null, nombre='null', añoNacimiento='null', dirección='null'}
```

hashCode():

Devuelve un valor de código hash para el objeto; una referencia que se mantiene igual para el mismo objeto a lo largo de cada ejecución.

```java
        Empleado e1 = new Empleado();
        Empleado e2 = e1;
        System.out.println(e1.hashCode());
        System.out.println(e2.hashCode());
        System.out.println(e1 == e2);
    }
}
```

```
Main2
C:\Users\anaij\.jdks\openjdk-17.0.1\bin\jav
 "C:\Users\anaij\OneDrive\_S\Módulos\PS-DAM
455659002
455659002
true
```

equals(Object obj):

Devuelve un boolean indicando si un objeto es igual al objeto que lo invoca; ya vimos que este es el método (y no ==) que hay que utilizar para comparar tipos referencias.

También puede sobreescribirse para especificar las condiciones en que queramos que dos objetos sean iguales.

[HACER EJERCICIO AL FINAL EN EL QUE SOBREESCRIBIMOS EQUALS PARA COMPARAR POR DNI – VER EJEMPLO EN EL LIBRO]

getClass():

Devuelve un String con el nombre cualificado de la clase del objeto que lo llama.

Por ejemplo:

```java
        String cadena = "";
        System.out.println(cadena.getClass());
```

```
Main2
C:\Users\anaij\.jdks\openjdk-17.0.1\bin\java.e
 "C:\Users\anaij\OneDrive\_S\Módulos\PS-DAM-DA
class java.lang.String
```

Clases abstractas

En ocasiones tenemos un conjunto de atributos y métodos que necesitamos reutlizar en todas las clases dentro de una jerarquía. **Las clases abstractas permiten juntar todos esos miembros (atributos y métodos) comunes en una clase base y después declarar subclases que puedan acceder a ellos**. Es decir, las clases abstractas solamente existen para que otras hereden de ellas.

Una clase abstracta se declara con la palabra clave **`abstract`**.

Tienen unas características especiales:

- No se pueden crear instancias de clases abstractas.
- Pueden contener **métodos abstractos**, que deben ser implementados en las subclases no abstractas.
- Pueden contener atributos y métodos no abstractos (incluyendo estáticos).
- Pueden extender otra clase, incluso si es abstracta.
- Pueden tener un constructor.

En definitiva, **las clases abstractas** tienen dos diferencias principales respecto a las demás clases: **no pueden instanciarse y tienen métodos abstractos.**

Los métodos abstractos se declaran con la palabra clave abstract (y los demás elementos: modificadores, tipo de retorno y signatura**), pero no se implementan**. **Cada subclase no abstracta debe implementarlos.** (Nota: los métodos abstractos no pueden ser estáticos).

Ejemplo:

```
public abstract class Mascota {

    protected String nombre;
    protected int edad;

    protected Mascota(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public abstract void dice(); // método abstracto
}
```

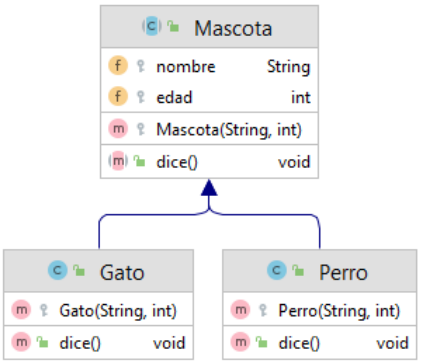
La clase abstracta Mascota tiene dos atributos, un constructor y un método abstracto. Como Mascota es abstracta, no podemos instanciarla.

```
Mascota mascota = new Mascota("Sin nombre", 5); // error de compilación
```

El método dice() se declara abstracto porque en este nivel de abstracción su implementación se desconoce. Deberán implementarlo las subclases.

A continuación declaramos dos subclases concretas (no abstractas) de Mascota.

```
class Gato extends Mascota {

    // Puede tener atributos adicionales

    public Gato(String nombre, int edad) {
        super(nombre, edad);
    }

    @Override
    public void dice() {
        System.out.println("¡Miau!");
    }
}

class Perro extends Mascota {

    // Puede tener atributos
adicionales

    public Perro(String nombre, int edad) {
        super(nombre, edad);
    }

    @Override
    public void dice() {
        System.out.println("¡Guau!");
    }
}
```



Como puede verse, las subclases sobreescriben el método abstracto.

Podemos instanciar estas clases e invocar al método dice()

```
Perro perro = new Perro("Jefe", 5);
Gato gato = new Gato("Tigre", 2);

perro.dice(); // imprime "¡Guau!"
gato.dice(); // imprime “¡Miau!"
```

No olvidemos que Java no soporta herencia múltiple. Por tanto, una subclase solamente puede heredar de una única clase abstracta.

Clases finales

Si no queremos que se puedan crear subclases a partir de una determinada, la declararemos con la palabra clave final.

```
final class ClaseSinHijas { }
```

Si intentamos extender esta clase, dará un error de compilación.

Algunas clases estándar estan declaradas así: Integer, Long, String, Math…

Preguntas cortas

1. Tenemos dos clases: Bicicleta y MountainBike. ¿Qué miembros de Bicicleta hereda MountainBike?

```
class Bicicleta {
    protected int cadencia;
    private int pistones;
    private int velocidad;

    protected int getVelocidad() {
        return velocidad;
    }

    public void acelerar(int incremento) {
        velocidad += incremento;
    }
}

class MountainBike extends Bicicleta { }
```

a) getVelocidad()
b) acelerar(int incremento)
c) cadencia
d) velocidad
e) pistones

2. Tenemos una jerarquía de clases que incluye tres clases:

```
class Animal {
    protected int edad;
    public int getEdad() {      return edad; }
    public void setEdad(int edad) {    this.edad = edad;   }
}

class Mascota extends Animal {
    protected String nombre;
    public String getNombre() { return nombre;      }
    public void setNombre(String nombre) {   this.nombre = nombre;     }
}



class Gato extends Mascota {

    protected String color;
    public String getColor() {        return color;    }

    public void setColor(String color) {        this.color = color;    }
}
```

Dado el siguiente objeto:

```
Mascota gato = new Gato();
```

Indica qué invocaciones de métodos son correctas y cuáles incorrectas:

```
gato.setEdad(5);
gato.getNombre();
gato.getColor();
gato.getEdad();
gato.setColor("Gris");
gato.setNombre("Faraón");
```

Ejercicios

3. Crea un paquete empresa; dentro de él las clases siguientes:
    a. Clase Persona, con los atributos nombre y edad (con los modificadores de acceso adecuados teniendo en cuenta que va a ser la clase padre de una jerarquía de clases), y un método mostrar() que muestre sus valores al ser invocado. Su constructor recibe todos los atributos como parámetros.
    b. Clase Empleado, que
        i. hereda de Persona y tiene el atributo propio double sueldoBruto;
        ii. sobreescribe el método mostrar para que muestre los valores del atributo propio y de los heredados;
        iii. tiene el método `calcularSalarioNeto():double`, que descuenta un 15% al salario bruto y devuelve el valor resultante.
        iv. Su constructor recibe todos los atributos como parámetros
    c. Clase Cliente, que
        i. hereda también de Persona y tiene el atributo propio `String telefonoDeContacto`. De esta clase no hereda ninguna otra.
        ii. También sobreescribe el método mostrar de la misma forma que la clase anterior.
        iii. Su constructor recibe todos los atributos como parámetros.
    d. Clase Directivo, que
        i. hereda de Empleado.
        ii. Tiene el atributo categoría, que es un tipo enumerado cuyos valores obtendremos de la siguiente URL: https://economipedia.com/definiciones/directivo.html
        iii. También sobreescribe el método mostrar con los valores de todos los atributos heredados más el atributo categoría.
        iv. Su constructor recibe todos los atributos como parámetros
    e. Clase Empresa;
        i. tiene el atributo nombre y además empleados y clientes, que representaremos con sendos ArrayList que añadiremos también como atributos.

ii. Su constructor recibe todos los atributos como parámetros.
iii. Añade un método mostrar(), que mostrará el resultado del método toString() */

f. Clase Main; contiene el método main, crea tres empleados (uno de ellos es un directivo de la categoría que quieras) y dos clientes y con todos ellos crea una empresa.

NOTA: solo llevarán getters y setters aquellas clases que tengan atributos declarados como privados