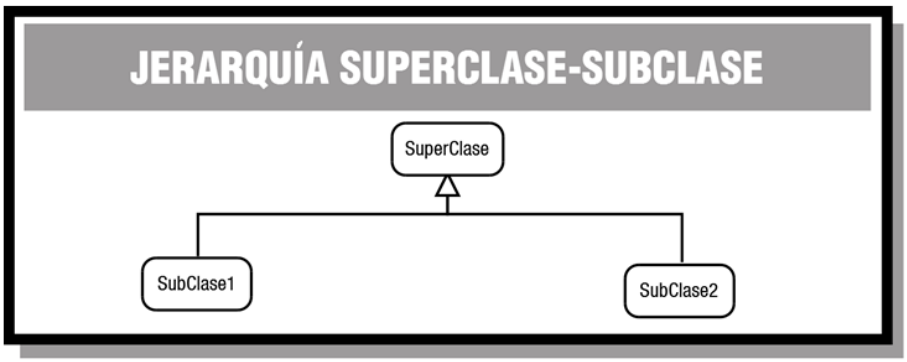
7.C. Herencia.

La **herencia** es el mecanismo que permite definir una nueva clase a partir de otra, pudiendo añadir nuevas características, sin tener que volver a escribir todo el código de la clase base.



La clase de la que se hereda suele ser llamada **clase base**, **clase padre** o **superclase**. A la clase que hereda se le suele llamar **clase hija**, **clase derivada** o **subclase**.

Una clase derivada puede ser a su vez **clase padre** de otra que herede de ella y así sucesivamente dando lugar a una **jerarquía de clases**, excepto aquellas que estén en la parte de arriba de la jerarquía (sólo serán **clases padre**) o en la parte de abajo (sólo serán **clases hijas**).

Una **clase hija** no tiene acceso a los miembros **privados** de su **clase padre**, tan solo a los **públicos** (como cualquier parte del código tendría) y los **protegidos** (a los que sólo tienen acceso las **clases derivadas** y las del mismo **paquete**). Aquellos miembros que sean privados en la clase base también habrán sido heredados, pero el acceso a ellos estará restringido al propio funcionamiento de la **superclase** y sólo se podrá acceder a ellos si la **superclase** ha dejado algún medio indirecto para hacerlo (por ejemplo a través de algún método).

Todos los miembros de la **superclase**, tanto atributos como métodos, son heredados por la subclase. Algunos de estos miembros heredados podrán ser **redefinidos** o **sobrescritos** (**overriden**) y también podrán añadirse nuevos miembros. De alguna manera podría decirse que estás “ampliando” la **clase base** con características adicionales o modificando algunas de ellas (proceso de **especialización**).

Una clase derivada extiende la funcionalidad de la clase base sin tener que volver a escribir el código de la clase base.

Autoevaluación

Una clase derivada hereda todos los miembros de su clase base, pudiendo acceder a cualquiera de ellos en cualquier momento. ¿Verdadero o Falso?

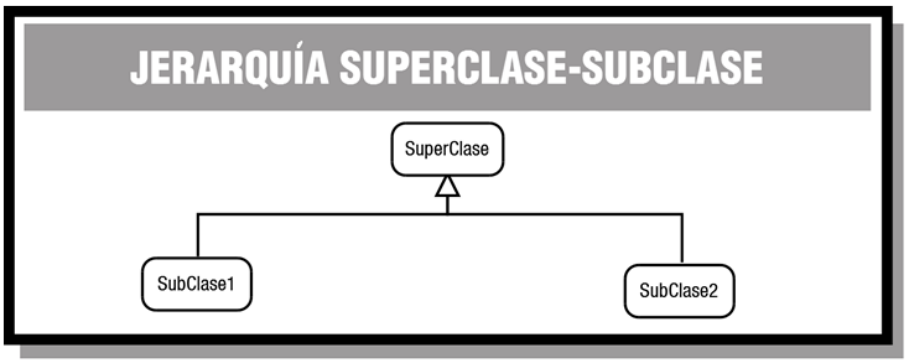
Sitio: Formación Profesional a Distancia
Curso: Programación
Libro: 7.C. Herencia.

Tabla de contenidos

1. Herencia.
1.1. Sintaxis de la herencia.
1.2. Acceso a miembros heredados.
1.3. Utilización de miembros heredados (I). Atributos.
1.4. Utilización de miembros heredados (II). Métodos.
1.5. Redefinición de métodos heredados.
1.6. Ampliación de métodos heredados.
1.7. Sobreescritura vs Sobrecarga
1.8. Constructores y herencia.
1.9. Creación y utilización de clases derivadas.
1.10. La clase Object en Java.
1.11. El método toString() de Object
1.12. Igualdad, Identidad y método EQUALS
1.13. Herencia múltiple.
1.14. Clases y métodos finales.

1. Herencia.

La **herencia** es el mecanismo que permite definir una nueva clase a partir de otra, pudiendo añadir nuevas características, sin tener que volver a escribir todo el código de la clase base.



La clase de la que se hereda suele ser llamada **clase base**, **clase padre** o **superclase**. A la clase que hereda se le suele llamar **clase hija**, **clase derivada** o **subclase**.

Una clase derivada puede ser a su vez **clase padre** de otra que herede de ella y así sucesivamente dando lugar a una **jerarquía de clases**, excepto aquellas que estén en la parte de arriba de la jerarquía (sólo serán **clases padre**) o en la parte de abajo (sólo serán **clases hijas**).

Una **clase hija** no tiene acceso a los miembros **privados** de su **clase padre**, tan solo a los **públicos** (como cualquier parte del código tendría) y los **protegidos** (a los que sólo tienen acceso las **clases derivadas** y las del mismo **paquete**). Aquellos miembros que sean privados en la clase base también habrán sido heredados, pero el acceso a ellos estará restringido al propio funcionamiento de la **superclase** y sólo se podrá acceder a ellos si la **superclase** ha dejado algún medio indirecto para hacerlo (por ejemplo a través de algún método).

Todos los miembros de la **superclase**, tanto atributos como métodos, son heredados por la subclase. Algunos de estos miembros heredados podrán ser **redefinidos** o **sobrescritos** (**overriden**) y también podrán añadirse nuevos miembros. De alguna manera podría decirse que estás “ampliando” la **clase base** con características adicionales o modificando algunas de ellas (proceso de **especialización**).

Una clase derivada extiende la funcionalidad de la clase base sin tener que volver a escribir el código de la clase base.

Autoevaluación

Una clase derivada hereda todos los miembros de su clase base, pudiendo acceder a cualquiera de ellos en cualquier momento. ¿Verdadero o Falso?

- Verdadero
- Falso

1.1. Sintaxis de la herencia.

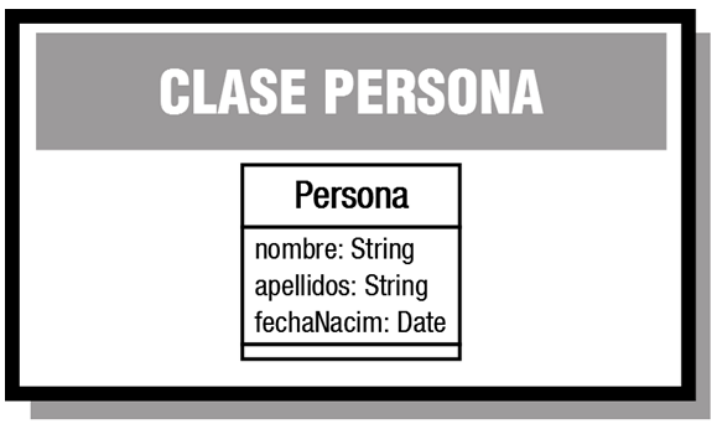
En Java la **herencia** se indica mediante la palabra reservada `extends`:

```
[modificador] class ClasePadre {
    // Cuerpo de la clase
    …
}

[modificador] class ClaseHija extends ClasePadre {
// Cuerpo de la clase
…
}
```

Imagina que tienes una clase **Persona** que contiene atributos como **nombre**, **apellidos** y **fecha de nacimiento**:

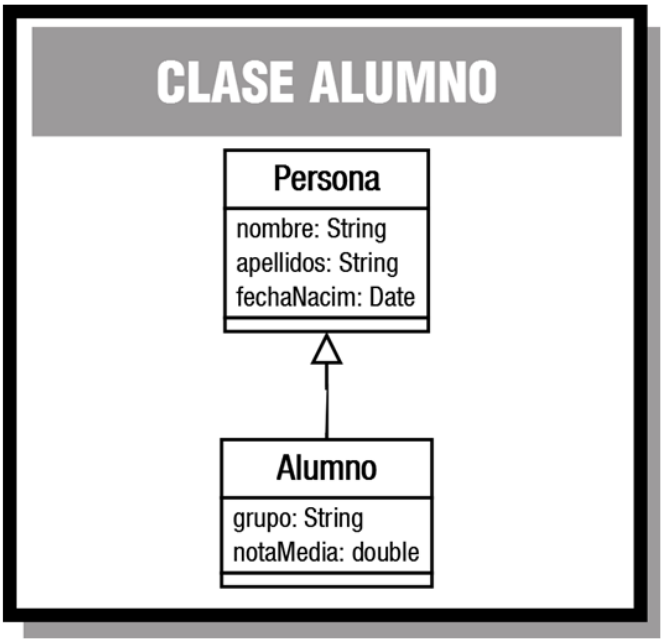
```
public class Persona {
    String nombre;
    String apellidos;
    GregorianCalendar fechaNacim;
    …
}
```



Es posible que, más adelante, necesites una clase **Alumno** que compartirá esos atributos (dado que todo alumno es una persona, pero con algunas características específicas que lo **especializan**). En tal caso tendrías la posibilidad de crear una clase **Alumno** que repitiera todos esos atributos o bien **heredar** de la clase

Persona:

```
public class Alumno extends Persona {
    String grupo;
    double notaMedia;
    …
}
```



A partir de ahora, un objeto de la clase **`Alumno`** contendrá los atributos **`grupo`** y **`notaMedia`** (propios de la clase **`Alumno`**), pero también **`nombre`**, **`apellidos`** y **`fechaNacim`** (propios de su **clase base** **`Persona`** y que por tanto ha heredado).

Autoevaluación

En Java la herencia se indica mediante la palabra reservada inherits**. ¿Verdadero o Falso?**

- Verdadero
- Falso

Ejercicio resuelto

Imagina que también necesitas una clase Profesor, que contará con atributos como nombre, apellidos, fecha de nacimiento, salario y especialidad. ¿Cómo crearías esa nueva clase y qué atributos le añadirías?

Solución:

Está claro que un **Profesor** es otra especialización de **Persona**, al igual que lo era **Alumno**, así que podrías crear otra clase derivada de **Persona** y así aprovechar los atributos genéricos (**nombre**, **apellidos**, **fecha de nacimiento**) que posee todo objeto de tipo **Persona**. Tan solo faltaría añadirle sus atributos específicos (**salario** y **especialidad**):

```
public class Profesor extends Persona {
```

```
    String especialidad;
```

```
    double salario;
```

```
    …
```

```
}
```

1.2. Acceso a miembros heredados.

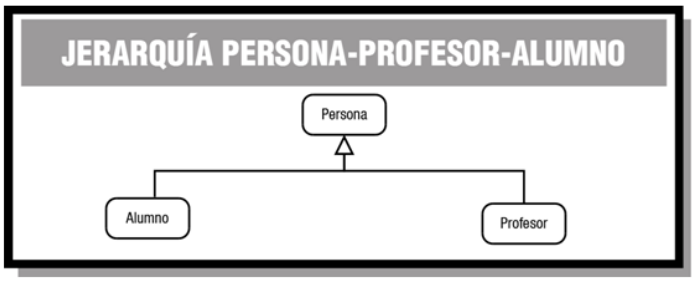
Como ya has visto anteriormente, no es posible acceder a miembros **privados** de una superclase. Para poder acceder a ellos podrías pensar en hacerlos **públicos**, pero entonces estarías dando la opción de acceder a ellos a cualquier objeto externo y es probable que tampoco sea eso lo deseable. Para ello se inventó el modificador `protected` (**protegido**) que permite el **acceso desde clases heredadas**, pero no desde fuera de las clases (estrictamente hablando, desde fuera del **paquete**), que serían como miembros **privados**.

En la unidad dedicada a la utilización de clases ya estudiaste los posibles modificadores de acceso que podía tener un miembro: **sin modificador** (acceso **de paquete**), **público**, **privado** o **protegido**. Aquí tienes de nuevo el resumen:

Cuadro de niveles accesibilidad a los atributos de una clase				
	Misma clase	**Subclase**	**Mismo paquete**	**Otro paquete**
Sin modificador (paquete)	X		X	X
public	X	X	X	X
Private	X			
Protected	X	X	X	

Si en el ejemplo anterior de la clase `Persona` se hubieran definido sus atributos como `private`:

```
public class Persona {
    private String nombre;
    private String apellidos;
    …
}
```



Al definir la clase `Alumno` como heredera de `Persona`, no habrías tenido acceso a esos atributos, pudiendo ocasionar un grave problema de operatividad al intentar manipular esa información. Por tanto, en estos casos lo más recomendable habría sido declarar esos atributos como `protected` o bien sin modificador (para que también tengan acceso a ellos otras clases del mismo paquete, si es que se considera oportuno):

```
public class Persona {
    protected String nombre;
```

```
    protected String apellidos;
    …
}
```

Sólo en aquellos casos en los que se desea explícitamente que un miembro de una clase no pueda ser accesible desde una clase derivada debería utilizarse el modificador **`private`**. En el resto de casos es recomendable utilizar **`protected`**, o bien no indicar modificador (acceso a nivel de **paquete**).

Ejercicio resuelto

Rescribe las clases `Alumno` y `Profesor` utilizando el modificador `protected` para sus atributos del mismo modo que se ha hecho para su superclase `Persona`

Solución:

1. **Clase `Alumno`.**

Se trata simplemente de añadir el modificador de acceso **`protected`** a los nuevos atributos que añade la clase.

```
public class Alumno extends Persona {
    protected String grupo;
    protected double notaMedia;
    …
}
```

2. **Clase `Profesor`.**

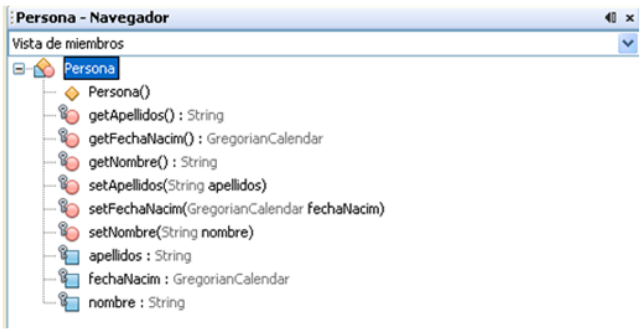
Exactamente igual que en la clase **`Alumno`**.

```
public class Profesor extends Persona {
    protected String especialidad;
    protected double salario;
    …
}
```

1.3. Utilización de miembros heredados (I). Atributos.

Los **atributos heredados** por una clase son, a efectos prácticos, iguales que aquellos que sean definidos específicamente en la nueva **clase derivada**.

En el ejemplo anterior la clase `Persona` disponía de tres atributos y la clase `Alumno`, que heredaba de ella, añadía dos atributos más. Desde un punto de vista funcional podrías considerar que la clase `Alumno` tiene cinco atributos: tres por ser `Persona` (**nombre**, **apellidos**, **fecha de nacimiento**) y otros dos más por ser `Alumno` (**grupo** y **nota media**).



Ejercicio resuelto

Dadas las clases `Alumno` y `Profesor` que has utilizado anteriormente, implementa métodos get y set en las clases `Alumno` y `Profesor` para trabajar con sus cinco atributos (tres heredados más dos específicos).

Solución:

Una posible solución sería:

1. **Clase `Alumno`.**

Se trata de heredar de la clase `Persona` y por tanto utilizar con normalidad sus atributos heredados como si pertenecieran a la propia clase (de hecho se puede considerar que le pertenecen, dado que los ha heredado).

```
public class Alumno extends Persona {
	protected String grupo;
	protected double notaMedia;

	// Método getNombre
	public String getNombre (){
	    return nombre;
	}

	// Método getApellidos
```

```
        public String getApellidos (){
            return apellidos;
        }

        // Método getFechaNacim
        public GregorianCalendar getFechaNacim (){
            return this.fechaNacim;
        }

        // Método getGrupo
        public String getGrupo (){
            return grupo;
        }

        // Método getNotaMedia
        public double getNotaMedia (){
            return notaMedia;
        }

        // Método setNombre
        public void setNombre (String nombre){
            this.nombre= nombre;
        }

        // Método setApellidos
        public void setApellidos (String apellidos){
            this.apellidos= apellidos;
        }

        // Método setFechaNacim
        public void setFechaNacim (GregorianCalendar fechaNacim){
            this.fechaNacim= fechaNacim;
```

```
        }

        // Método setGrupo
        public void setGrupo (String grupo){
            this.grupo= grupo;
        }

        // Método setNotaMedia
        public void setNotaMedia (double notaMedia){
            this.notaMedia= notaMedia;
        }

}
```

Si te fijas, puedes utilizar sin problema la referencia **`this`** a la propia clase con esos atributos heredados, pues pertenecen a la clase: **`this.nombre`**, **`this.apellidos`**, etc.

2. **Clase `Profesor`.**

Seguimos exactamente el mismo procedimiento que con la clase **`Alumno`**.

```
public class Profesor extends Profesor {
        String especialidad;
        double salario;

        // Método getNombre
        public String getNombre (){
            return nombre;
        }

        // Método getApellidos
        public String getApellidos (){
            return apellidos;
        }

        // Método getFechaNacim
```

```
        public GregorianCalendar getFechaNacim (){
            return this.fechaNacim;
        }

        // Método getEspecialidad
        public String getEspecialidad (){
            return especialidad;
        }

        // Método getSalario
        public double getSalario (){
            return salario;
        }

        // Método setNombre
        public void setNombre (String nombre){
            this.nombre= nombre;
        }

        // Método setApellidos
        public void setApellidos (String apellidos){
            this.apellidos= apellidos;
        }

        // Método setFechaNacim
        public void setFechaNacim (GregorianCalendar fechaNacim){
            this.fechaNacim= fechaNacim;
        }

        // Método setSalario
        public void setSalario (double salario){
            this.salario= salario;
```

```
        }

        // Método setESpecialidad
        public void setESpecialidad (String especialidad){
            this.especialidad= especialidad;
        }
}
```

Una conclusión que puedes extraer de este código es que has tenido que escribir los métodos **get** y **set** para los tres atributos heredados, pero ¿no habría sido posible definir esos seis métodos en la clase base y así estas dos clases derivadas hubieran también heredado esos métodos? La respuesta es afirmativa y de hecho es como lo vas a hacer a partir de ahora. De esa manera te habrías evitado tener que escribir seis métodos en la clase **Alumno** y otros seis en la clase **Profesor**. Así que recuerda: **se pueden heredar tanto los atributos como los métodos**.

Aquí tienes un ejemplo de cómo podrías haber definido la clase **Persona** para que luego se hubieran podido heredar de ella sus métodos (y no sólo sus atributos):

```
public class Persona {
        protected String nombre;
        protected String apellidos;
        protected GregorianCalendar fechaNacim;

        // Método getNombre
        public String getNombre (){
            return nombre;
        }

        // Método getApellidos
        public String getApellidos (){
            return apellidos;
        }

        // Método getFechaNacim
        public GregorianCalendar getFechaNacim (){
            return this.fechaNacim;
```

```
        }

        // Método setNombre
        public void setNombre (String nombre){
            this.nombre= nombre;
        }

        // Método setApellidos
        public void setApellidos (String apellidos){
            this.apellidos= apellidos;
        }

        // Método setFechaNacim
        public void setFechaNacim (GregorianCalendar fechaNacim){
            this.fechaNacim= fechaNacim;
        }

}
```

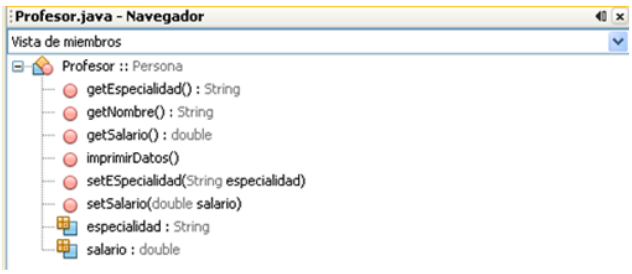
1.4. Utilización de miembros heredados (II). Métodos.

Del mismo modo que se heredan los **atributos**, también se heredan los **métodos**, convirtiéndose a partir de ese momento en otros **métodos** más de la **clase derivada**, junto a los que hayan sido definidos específicamente.

En el ejemplo de la clase `Persona`, si dispusiéramos de métodos **get** y **set** para cada uno de sus tres atributos (`nombre`, `apellidos`, `fechaNacim`), tendrías seis métodos que podrían ser heredados por sus **clases derivadas**. Podrías decir entonces que la clase `Alumno`, derivada de `Persona`, tiene diez métodos:

- Seis por ser `Persona` (`getNombre`, `getApellidos`, `getFechaNacim`, `setNombre`, `setApellidos`, `setFechaNacim`).
- Oros cuatro más por ser `Alumno` (`getGrupo`, `setGrupo`, `getNotaMedia`, `setNotaMedia`).

Sin embargo, sólo tendrías que definir esos cuatro últimos (los **específicos**) pues los **genéricos** ya los has heredado de la **superclase**.



Autoevaluación

En Java los métodos heredados de una superclase deben volver a ser definidos en las subclases. ¿Verdadero o Falso?

- Verdadero
- Falso

Ejercicio resuelto

Dadas las clases `Persona`, `Alumno` y `Profesor` que has utilizado anteriormente, implementa métodos get y set en la clase `Persona` para trabajar con sus tres atributos y en las clases `Alumno` y `Profesor` para manipular sus cinco atributos (tres heredados más dos específicos), teniendo en cuenta que los métodos que ya hayas definido para `Persona` van a ser heredados en `Alumno` y en `Profesor`.

Solución:

Una posible solución:

1. **Clase `Persona`.**

```
public class Persona {
        protected String nombre;
        protected String apellidos;
        protected GregorianCalendar fechaNacim;

        // Método getNombre
        public String getNombre (){
            return nombre;
        }

        // Método getApellidos
        public String getApellidos (){
            return apellidos;
        }

        // Método getFechaNacim
        public GregorianCalendar getFechaNacim (){
            return this.fechaNacim;
        }

        // Método setNombre
        public void setNombre (String nombre){
            this.nombre= nombre;
        }

        // Método setApellidos
        public void setApellidos (String apellidos){
            this.apellidos= apellidos;
        }

        // Método setFechaNacim
        public void setFechaNacim (GregorianCalendar fechaNacim){
```

```
            this.fechaNacim= fechaNacim;
        }
```

2. **Clase `Alumno`.**

Al heredar de la clase **`Persona`** tan solo es necesario escribir métodos para los nuevos atributos (**métodos especializados** de acceso a los **atributos especializados**), pues los **métodos genéricos** (de acceso a los **atributos genéricos**) ya forman parte de la clase al haberlos heredado.

```
public class Alumno extends Persona {
        protected String grupo;
        protected double notaMedia;

        // Método getGrupo
        public String getGrupo (){
            return grupo;
        }

        // Método getNotaMedia
        public double getNotaMedia (){
            return notaMedia;
        }

        // Método setGrupo
        public void setGrupo (String grupo){
            this.grupo= grupo;
        }

        // Método setNotaMedia
        public void setNotaMedia (double notaMedia){
            this.notaMedia= notaMedia;
        }

}
```

Aquí tienes una demostración práctica de cómo la herencia permite una reutilización eficiente del código, evitando tener que repetir atributos y métodos. Sólo has tenido que escribir cuatro métodos en lugar de diez.

3. **Clase `Profesor`.**

Seguimos exactamente el mismo procedimiento que con la clase **`Alumno`**.

```
public class Profesor extends Profesor {
        String especialidad;
        double salario;

        // Método getEspecialidad
        public String getEspecialidad (){
            return especialidad;
        }

        // Método getSalario
        public double getSalario (){
            return salario;
        }

        // Método setSalario
        public void setSalario (double salario){
            this.salario= salario;
        }

        // Método setEspecialidad
        public void setESpecialidad (String especialidad){
            this.especialidad= especialidad;
        }
}
```

1.5. Redefinición de métodos heredados.

Una clase puede **redefinir** algunos de los métodos que ha heredado de su **clase base**. En tal caso, el nuevo método (**especializado**) sustituye al **heredado**. Este procedimiento también es conocido como de **sobrescritura de métodos**.

En cualquier caso, aunque un método sea **sobrescrito** o **redefinido**, aún es posible acceder a él a través de la referencia `super`, aunque sólo se podrá acceder a métodos de la **clase padre** y no a métodos de clases superiores en la **jerarquía de herencia**.

Los **métodos redefinidos** pueden **ampliar su accesibilidad** con respecto a la que ofrezca el método original de la **superclase**, pero **nunca restringirla**. Por ejemplo, si un método es declarado como `protected` o **de paquete** en la clase base, podría ser redefinido como `public` en una clase derivada.

Los **métodos estáticos** o de clase no pueden ser sobrescritos. Los originales de la clase base permanecen inalterables a través de toda la **jerarquía de herencia**.

En el ejemplo de la clase `Alumno`, podrían redefinirse algunos de los métodos heredados. Por ejemplo, imagina que el método `getApellidos` devuelva la cadena “Alumno:” junto con los apellidos del alumno. En tal caso habría que rescribir ese método para realizara esa modificación:

```
public String getApellidos () {
```

```
    return “Alumno: “ + apellidos;
```

```
}
```

Cuando sobrescribas un método heredado en Java puedes incluir la **anotación** `@Override`. Esto indicará al compilador que tu intención es **sobrescribir el método de la clase padre**. De este modo, si te equivocas (por ejemplo, al escribir el nombre del método) y no lo estás realmente sobrescribiendo, el compilador producirá un error y así podrás darte cuenta del fallo. En cualquier caso, no es necesario indicar `@Override`, pero puede resultar de ayuda a la hora de localizar este tipo de errores (crees que has sobrescrito un **método heredado** y al confundirte en una letra estás realmente creando un nuevo método diferente). En el caso del ejemplo anterior quedaría:

```
@Override
```

```
public String getApellidos ()
```

Autoevaluación

Dado que el método `finalize()` de la clase `Object` es `protected`, el método `finalize()` de cualquier clase que tú escribas podrá ser `public`, `private` o `protected`. ¿Verdadero o Falso?

- Verdadero
- Falso

Ejercicio resuelto

Dadas las clases `Persona`, `Alumno` y `Profesor` que has utilizado anteriormente, redefine el método `getNombre` para que devuelva la cadena "`Alumno:` ", junto con el nombre del alumno, si se trata de un objeto de la clase `Alumno` o bien "`Profesor` ", junto con el nombre del profesor, si se trata de un objeto de la clase `Profesor`.

Solución:

1. **Clase `Alumno`.**

Al heredar de la clase `Persona` tan solo es necesario escribir métodos para los nuevos atributos (**métodos especializados** de acceso a los **atributos especializados**), pues los **métodos genéricos** (de acceso a los **atributos genéricos**) ya forman parte de la clase al haberlos heredado. Esos son los métodos que se implementaron en el ejercicio anterior (`getGrupo`, `setGrupo`, etc.).

Ahora bien, hay que escribir otro método más, pues tienes que redefinir el método `getNombre` para que tenga un comportamiento un poco diferente al `getNombre` que se hereda de la clase base `Persona`:

```
// Método getNombre
@Override
public String getNombre (){
            return "Alumno: " + this.nombre;
}
```

En este caso podría decirse que se "renuncia" al método heredado para redefinirlo con un comportamiento más especializado y acorde con la clase derivada.

2. **Clase `Profesor`.**

Seguimos exactamente el mismo procedimiento que con la clase `Alumno` (redefinición del método `getNombre`).

```
// Método getNombre
@Override
public String getNombre (){
            return "Profesor: " + this.nombre;
}
```

1.6. Ampliación de métodos heredados.

Hasta ahora, has visto que para **redefinir** o **sustituir** un **método** de una **superclase** es suficiente con crear otro método en la **subclase** que tenga el mismo nombre que el método que se desea **sobrescribir**. Pero, en otras ocasiones, puede que lo que necesites no sea sustituir completamente el comportamiento del método de la superclase, sino simplemente **ampliarlo**.

Para poder hacer esto necesitas poder **preservar el comportamiento antiguo** (el de la **superclase**) y **añadir el nuevo** (el de la **subclase**). Para ello, puedes invocar desde el método "**ampliador**" de la **clase derivada** al método "**ampliado**" de la clase superior (teniendo ambos métodos el mismo nombre). ¿Cómo se puede conseguir eso? Puedes hacerlo mediante el uso de la referencia `super`.

La palabra reservada `super` es una referencia a la **clase padre** de la clase en la que te encuentres en cada momento (es algo similar a `this`, que representaba una referencia a la **clase actual**). De esta manera, podrías invocar a cualquier método de tu **superclase** (si es que se tiene acceso a él).

Por ejemplo, imagina que la clase `Persona` dispone de un método que permite mostrar el contenido de algunos datos personales de los objetos de este tipo (**nombre**, **apellidos**, etc.). Por otro lado, la clase `Alumno` también necesita un método similar, pero que muestre también su información especializada (**grupo**, **nota media**, etc.). ¿Cómo podrías aprovechar el método de la **superclase** para no tener que volver a escribir su contenido en la subclase?

Podría hacerse de una manera tan sencilla como la siguiente:

```
public void mostrar () {
    super.mostrar ();     // Llamada al método "mostrar" de la superclase
    // A continuación mostramos la información "especializada" de esta subclase
System.out.printf ("Grupo: %s\n", this.grupo);
           System.out.printf ("Nota media: %5.2f\n", this.notaMedia);
}
```

Este tipo de **ampliaciones de métodos** resultan especialmente útiles por ejemplo en el caso de los **constructores**, donde se podría ir llamando a los **constructores** de cada **superclase** encadenadamente hasta el **constructor** de la clase en la **cúspide de la jerarquía** (el **constructor** de la clase `Object`).

Ejercicio resuelto

Dadas las clases `Persona`, `Alumno` y `Profesor`, define un método mostrar para la clase `Persona`, que muestre el contenido de los atributos (datos personales) de un objeto de la clase `Persona`. A continuación, define sendos métodos mostrar especializados para las clases `Alumno` y `Profesor` que "amplíen" la funcionalidad del método mostrar original de la clase `Persona`.

Solución:

1. Método **mostrar** de la clase **`Persona`**.

```
public void mostrar () {
    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");
    String Stringfecha= formatoFecha.format(this.fechaNacim.getTime());

    System.out.printf ("Nombre: %s\n", this.nombre);
    System.out.printf ("Apellidos: %s\n", this.apellidos);
    System.out.printf ("Fecha de nacimiento: %s\n", Stringfecha);
}
```

2. Método **mostrar** de la clase **`Profesor`**.

Llamamos al método **`mostrar`** de su **clase padre** (**`Persona`**) y luego añadimos la **funcionalidad específica** para la **subclase** **`Profesor`**:

```
public void mostrar () {
                super.mostrar ();     // Llamada al método "mostrar" de la superclase

// A continuación mostramos la información "especializada" de esta subclase
    System.out.printf ("Especialidad: %s\n", this.especialidad);
    System.out.printf ("Salario: %7.2f euros\n", this.salario);
}
```

3. Método **mostrar** de la clase **`Alumno`**.

Llamamos al método **`mostrar`** de su **clase padre** (**`Persona`**) y luego añadimos la **funcionalidad específica** para la **subclase** **`Alumno`**:

```
public void mostrar () {
                super.mostrar ();

// A continuación mostramos la información "especializada" de esta subclase
    System.out.printf ("Grupo: %s\n", this.grupo);
    System.out.printf ("Nota media: %5.2f\n", this.notaMedia);
}
```

1.7. Sobreescritura vs Sobrecarga

Sobreescritura vs Sobrecarga

- *No hay que confundir la sobreescritura (Overriding) con la sobrecarga (Overloading) de métodos.*
- **Sobrecarga de Métodos**
  - Se produce dentro de la misma clase.
  - Sobrecargar un método es permitir tener dos o mas métodos con el mismo nombre **dentro de la misma clase** pero con diferente combinación de parámetros o firma.
  - Deben presentar diferencia en alguna de éstas características de los parámetros del método :
    - Número
    - Tipo
    - Orden
  - La firma de un método es la combinación del nombre y los tipos de los parámetros o argumentos.
- **Sobreescritura de Métodos**
  - Se produce entre clases heredadas.
  - Sobreescribir un método es replicar un método en una subclase declarado en una clase padre.
    - Al invocar al método dentro de la clase derivada se accede al de la clase derivada y no al de la clase padre.
  - Una subclase esta sobreescribiendo un método de su clase padre cuando la subclase defina un método con el mismo prototipo que el método de la superclase.

Sobreescritura vs Sobrecarga. Ejemplo.

- Supongamos la Jerarquía de Clase
  - Tenemos una Clase Base de la que hacemos heredar una Clase Derivada : **Derivada extend Base**.
- Supongamos los siguientes métodos de cada clase :
  1. Base
     1. int f(int)
     2. int f(Stirng)
     3. int f(boolean)
  2. Derivada
     1. int f (int)
     2. int f (String)
     3. int f (boolean , int)
- Las funciones 01.01, 01.02 y 01.03 están sobrecargadas.
- Las funciones 01.01 y 02.01 están sobreescritas.
- Las funciones 02.01, 02.02 y 02.03 están sobrecargadas.
- Las funciones 01.02 y 02.02 están sobreescritas.
- Pero las funciones 01.03 y 02.03 **NO** están sobreescritas por diferenciarse en el número de parámetros que reciben.

1.8. Constructores y herencia.

Recuerda que cuando estudiaste los **constructores** viste que un **constructor** de una clase puede llamar a otro **constructor** de la misma clase, previamente definido, a través de la referencia `this`. En estos casos, la utilización de `this` sólo podía hacerse en la primera línea de código del **constructor**.

Como ya has visto, un **constructor** de una **clase derivada** puede hacer algo parecido para llamar al **constructor** de su **clase base** mediante el uso de la palabra `super`. De esta manera, el **constructor** de una **clase derivada** puede llamar primero al **constructor** de su **superclase** para que inicialice los **atributos heredados** y posteriormente se inicializarán los **atributos específicos** de la clase: los no heredados. Nuevamente, esta llamada también **debe ser la primera sentencia de un constructor** (con la única excepción de que exista una llamada a otro constructor de la clase mediante `this`).

Si no se incluye una llamada a `super()` dentro del **constructor**, el compilador incluye automáticamente una llamada al constructor por defecto de **clase base** (llamada a `super()`). Esto da lugar a una **llamada en cadena de constructores de superclase** hasta llegar a la clase más alta de la jerarquía (que en Java es la clase `Object`).

En el caso del **constructor por defecto** (el que crea el compilador si el programador no ha escrito ninguno), el compilador añade lo primero de todo, antes de la inicialización de los atributos a sus valores por defecto, una llamada al constructor de la **clase base** mediante la referencia `super`.

A la hora de destruir un objeto (método `finalize`) es importante llamar a los finalizadores en el **orden inverso** a como fueron llamados los constructores (p**rimero se liberan los recursos de la clase derivada y después los de la clase base** mediante la llamada `super.finalize()`).

Si la clase `Persona` tuviera un constructor de este tipo:

```
public Persona (String nombre, String apellidos, GregorianCalendar fechaNacim) {
    this.nombe= nombre;
    this.apellidos= apellidos;
    this.fechaNacim= new GregorianCalendar (fechaNacim);
}
```

Podrías llamarlo desde un constructor de una clase derivada (por ejemplo `Alumno`) de la siguiente forma:

```
public Alumno (String nombre, String apellidos, GregorianCalendar fechaNacim, String
grupo, double notaMedia) {
    super (nombre, apellidos, fechaNacim);
    this.grupo= grupo;
    this.notaMedia= notaMedia;
}
```

En realidad se trata de otro recurso más para optimizar la **reutilización de código**, en este caso el del

constructor, que aunque no es heredado, sí puedes invocarlo para no tener que rescribirlo.

Autoevaluación

Puede invocarse al constructor de una superclase mediante el uso de la referencia `this`. ¿Verdadero o Falso?

- Verdadero
- Falso

Ejercicio resuelto

Escribe un constructor para la clase `Profesor` que realice una llamada al constructor de su clase base para inicializar sus atributos heredados. Los atributos específicos (no heredados) sí deberán ser inicializados en el propio constructor de la clase `Profesor`.

Solución:

```
public Profesor (String nombre, String apellidos, GregorianCalendar fechaNacim,
String especialidad, double salario) {
    super (nombre, apellidos, fechaNacim);
    this.especialidad= especialidad;
    this.salario= salario;
}
```

1.9. Creación y utilización de clases derivadas.

Ya has visto cómo crear una **clase derivada**, cómo acceder a los **miembros heredados** de las **clases superiores**, cómo redefinir algunos de ellos e incluso cómo invocar a un **constructor** de la **superclase**. Ahora se trata de poner en práctica todo lo que has aprendido para que puedas crear tus propias **jerarquías de clases,** o basarte en clases que ya existan en Java para heredar de ellas, y las utilices de manera adecuada para que tus aplicaciones sean más fáciles de escribir y mantener.

La idea de la **herencia** no es complicar los programas, sino todo lo contrario: **simplificarlos al máximo**. Procurar que haya que escribir la menor cantidad posible de código repetitivo e intentar facilitar en lo posible la realización de cambios (bien para corregir errores bien para incrementar la funcionalidad).

1.10. La clase Object en Java.

Todas las clases en Java son descendentes (directos o indirectos) de la clase `Object`. Esta clase define los **estados y comportamientos básicos que deben tener todos los objetos**. Entre estos comportamientos, se encuentran:

- **La posibilidad de compararse.**
- **La capacidad de convertirse a cadenas.**
- **La habilidad de devolver la clase del objeto.**

Entre los métodos que incorpora la clase `Object` y que por tanto hereda cualquier clase en Java tienes:

Principales métodos de la clase Object	
Método	**Descripción**
`Object ()`	Constructor.
`clone ()`	Método clonador: crea y devuelve una copia del objeto ("clona" el objeto).
`boolean equals (Object obj)`	Indica si el objeto pasado como parámetro es igual a este objeto.
`void finalize ()`	Método llamado por el recolector de basura cuando éste considera que no queda ninguna referencia a este objeto en el entorno de ejecución.
`int hashCode ()`	Devuelve un código hash para el objeto.
`toString ()`	Devuelve una representación del objeto en forma de String.

La clase `Object` representa la **superclase** que se encuentra en la cúspide de la **jerarquía de herencia** en Java. Cualquier clase (incluso las que tú implementes) acaban heredando de ella.

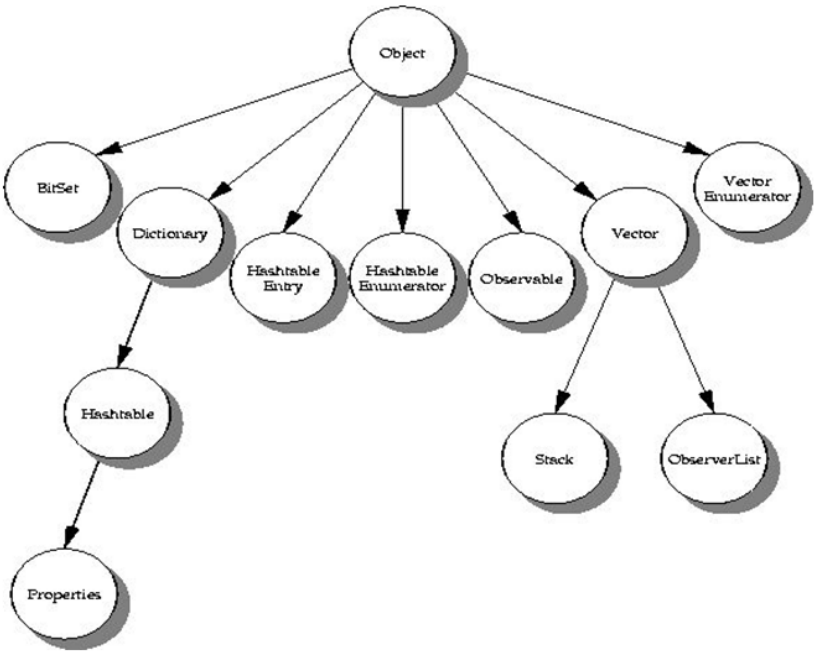


Imagen extraída de curso Programación del MECD.

Para saber más

Para obtener más información sobre la clase `Object`, sus métodos y propiedades, puedes consultar la documentación de la API de **Java** en el sitio web de Oracle.

Documentación de la clase `Object`.

Autoevaluación

Toda clase Java tiene un método `toString` y un método `finalize`. ¿Verdadero o Falso?

- Verdadero
- Falso

1.11. El método toString() de Object

El método toString() de Object

- El método toString nos devuelve la información indicada de un objeto en un String.
- Permite mostrar la información completa de un objeto, es decir, el valor de sus atributos.
- Este método lo tienen todos los objetos ya que se hereda de java.lang.Object
  - Deberemos sobrescribir este método en nuestras clases para poder ser utilizado:

```
//Ejemplo
public class Punto(){
    double x; double y;
    ...
    ...
    public String toString(){
        String retorno = "( " + x + " , " + y + ")";
        return retorno;
    }
}
```

  - Una vez definido en la clase podríamos usa un println con nuestros objetos en una simple concatenación.
    - println("Las coordenadas del punto son" + P)

1.12. Igualdad, Identidad y método EQUALS

Igualdad, Identidad y método EQUALS

- Sabemos que un objeto ocupa un espacio de memoria.
- Supongamos la clase Persona y dos objetos creados como:

```
//Ejemplo
Persona p1 = new Persona(“José Ramírez Mota”, 32);
Persona p2 = new Persona(“José Ramírez Mota”, 32);
```

- Ahora nos podemos plantear lo siguiente: ¿Que quiere decir? **¿p1 == p2?**
- Esta pregunta en Java se traduce así: ¿Es el objeto al que apunta p1 igual al objeto al que apunta p2?
- Es decir, ¿son el mismo objeto los objetos apuntados por p1 y p2, y por tanto existe una relación de identidad entre ambos?
  - La respuesta es que **no**, *no son el mismo objeto*, pues cada vez que usamos new, creamos un nuevo objeto con su zona de memoria, aunque **su contenido** si es el mismo.
- Seguramente no queríamos preguntarnos si las variables apuntadoras apuntaban al mismo objeto, sino si los objetos eran “iguales”.
  - La igualdad entre tipos primitivos sí es evaluable con relativa facilidad porque no distinguimos entre dirección y contenido,
    - En tipos primitivos ambas cosas podemos decir que son lo mismo.
- La igualdad entre objetos en Java se basa en que exista una definición previa de cuál es el criterio de igualdad, y esta definición es la que dé el método **equals** aplicable al objeto (todo objeto en principio tendrá un método equals).
- Peor aún, ¿qué ocurre si hacemos p1 = p2?
  - En este caso un objeto deja de estar apuntado por variable alguna.
  - Las dos variables pasan a apuntar al mismo objeto y por tanto p1 == p2 devuelve true.
  - Hay que recordar que los String son objetos y que por tanto no se pueden comparar usando ==.

Uso de equals correcto.

- Consideremos este caso:

```
//Ejemplo
Persona p1 = new Persona(“Andrés Hernández Suárez”, 32, “54221893-D”,
“Economista);
Persona p2 = new Persona(“Andrés Hernández Suárez”, 32, “54221893-D”, “Abogado”);
```

- ¿Cómo establecemos si dos personas son iguales?
  - El concepto de igualdad entre objetos, tiene que ser definido para que el compilador sepa a qué atenerse si se comparan dos objetos, en este caso a dos personas.
  - En el Ejemplo :
    - ¿Son dos personas diferentes? -¿Es la misma persona con dos Profesiones?
  - La comparación usará el metodo **equals()** (**lo define el Programador**) :
    - boolean objeto1.equals(objeto2)
- Java permite el uso del método equals para todo objeto dado que es parte de **object**
- El criterio por defecto de Object no nos va a ser útil para clases creadas por nosotros :
  - **Tendremos que definir nuestro criterio de igualdad dentro de nuestra clase** :
- Hay que sobrecargar **equals()** ya que existe en cualquier clase dado que es parte de **object**.
- **¡Ojo!!** : Ampliamos los métodos necesarios al crear una clase propia completa.

Ejemplo equals para la Clase Persona

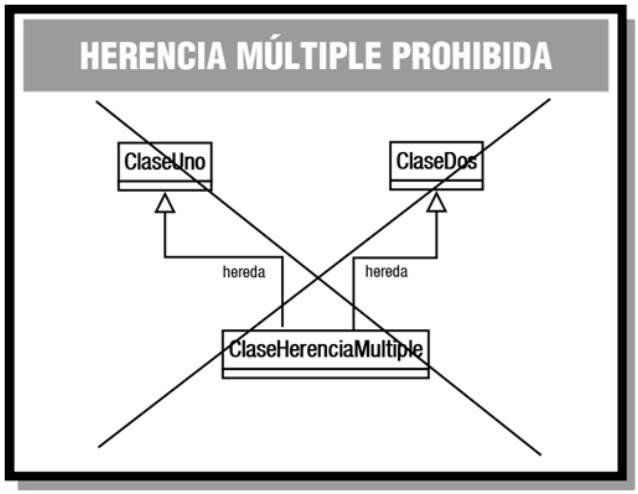
```
//Ejemplo equals en Persona
    @Override
    // Código que añadimos a la clase Persona. Sobreescritura del método equals.
    public boolean equals( Object obj ){
        if (obj instanceof Persona) {
            Persona tmpPersona = (Persona) obj;
                //Se Obtiene una referencia de obj como Persona con un casting.
            if( this.nombre.equals(tmpPersona.nombre) &&
                    this.apellidos.equals(tmpPersona.apellidos) &&
                    this.fechaNacim.equals( tmpPersona.fechaNacim ) ){
                return true;
            }
            else{ return false; }
        }else{ return false; }
    } //Cierre del método equals
```

1.13. Herencia múltiple.

En determinados casos podrías considerar la posibilidad de que se necesite **heredar de más de una clase**, para así disponer de los miembros de dos (o más) clases disjuntas (que no derivan una de la otra). La **herencia múltiple** permite hacer eso: recoger las distintas características (atributos y métodos) de clases diferentes formando una nueva clase derivada de varias clases base.

El problema en estos casos es la posibilidad que existe de que se produzcan ambigüedades, así, si tuviéramos miembros con el mismo identificador en clases base diferentes, en tal caso, ¿qué miembro se hereda? Para evitar esto, los compiladores suelen solicitar que ante casos de ambigüedad, se especifique de manera explícita la clase de la cual se quiere utilizar un determinado miembro que pueda ser ambiguo.

Ahora bien, la posibilidad de **herencia múltiple** no está disponible en todos los lenguajes orientados a objetos, ¿lo estará en Java? La respuesta es negativa.



En Java no existe la herencia múltiple de clases.

1.14. Clases y métodos finales.

En unidades anteriores has visto el modificador **final**, aunque sólo lo has utilizado por ahora para **atributos** y **variables** (por ejemplo para declarar **atributos constantes**, que una vez que toman un valor ya no pueden ser modificados). Pero este modificador también puede ser utilizado con clases y con métodos (con un comportamiento que no es exactamente igual, aunque puede encontrarse cierta analogía: no se permite heredar o no se permite redefinir).

Una clase declarada como final no puede ser heredada, es decir, **no puede tener clases derivadas**. La jerarquía de clases a la que pertenece acaba en ella (no tendrá clases hijas):

```
[modificador_acceso] final class nombreClase  [herencia] [interfaces]
```

Un **método** también puede ser declarado como **final**, en tal caso, ese método no podrá ser redefinido en una **clase derivada**:

```
[modificador_acceso] final <tipo> <nombreMetodo> ([parámetros]) [excepciones]
```

Si intentas redefinir un método **final** en una subclase se producirá un **error de compilación**.

Autoevaluación

Los modificadores **final** y **abstract** son excluyentes en la declaración de un **método**. ¿Verdadero o Falso?

- Verdadero
- Falso

Además de en la declaración de atributos, clases y métodos, el modificador **final** también podría aparecer acompañando a un método de un parámetro. En tal caso no se podrá modificar el valor del parámetro dentro del código del método. Por ejemplo: **`public final metodoEscribir (int par1, final int par2)`**.