Unidad 2

DIPLOMATURA EN PROGRAMACION JAVA

Capítulo 4

Clases

En este módulo

- Declaraciones básicas
- Visibilidades: control de acceso a la clase
- Crear Paquetes
- Constructores
- Sobrecarga de Métodos
- Uso de this
- Uso de this en Constructores
- Herencia

Declaraciones básicas

Las clases son el mecanismo por el que se pueden crear nuevos Tipos en Java. Las clases son el punto central sobre el que giran la mayoría de los conceptos de la Orientación a Objetos.

Las clases, al igual que las estructuras de datos en lenguajes como C, son tipos de datos definidos por el usuario, por lo tanto, el compilador no sabe nada de ellos hasta que se define la forma que tendrán.

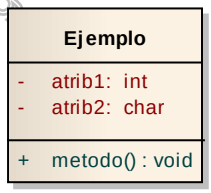
La declaración de la forma de los tipos de datos definidos por el usuario se especifica en una clase y cuando se crea una variable del tipo de la clase declarada, se crea un objeto.

Las clases se componen de métodos y atributos que la definen. Los primeros son los servicios que los objetos de ese tipo brindarán. Los atributos, en cambio, tendrán almacenados los valores que describen a ese objeto en particular y sobre los cuales actuarán los métodos.

Ejemplo

```
package declaraciones;
public class Ejemplo {
	private int atrib1;
	private char atrib2;
	public void metodo(){}
}
```

El lenguaje de modelización UML permite realizar una descripción gráfica de la clase de la siguiente forma:



Visibilidades: control de acceso a la clase

Uno de los beneficios de las clases es que pueden proteger sus variables y métodos miembros frente al acceso de otros objetos. ¿Por qué es esto importante? Bien, consideremos esto. Se ha escrito una clase que representa una petición a una base de datos que contiene toda clase de información secreta, es decir, registros de empleados o proyectos secretos de la compañía. Si se quiere mantener el control de acceso a esta información, sería bueno limitarlo.

Otra forma de ver lo mismo es para aquellos atributos que tiene un objeto que no deben cambiar. Por ejemplo, si existe una clase Persona, no debería existir forma en la clase o en un objeto que se cree a partir de ella, de cambiar el atributo nombre

Los datos que se almacenan dentro de las clases deben tener un acceso controlado ya que son la base de procesamiento de servicios y delimitan el estado de un objeto en un determinado momento del tiempo. Por lo tanto, su acceso debe ser controlado por algún mecanismo capaz de proteger sus valores y permita asegurar el correcto funcionamiento. Dicha herramienta existe en los lenguajes y se implementa a través de la visibilidad

En Java se puede utilizar los modificadores de acceso para proteger tanto las variables como los métodos de la clase cuando se declaran. El lenguaje soporta cuatro niveles de acceso para las variables y métodos miembros: **private**, **protected**, **public**, y acceso de paquete (sin declaración de modificador).

private

El nivel de acceso más restringido es **private**. Un miembro privado es accesible sólo para la clase en la que está definido. Se utiliza este acceso para declarar miembros que sólo deben ser utilizados por la clase. Esto incluye las variables que contienen información que si se accede a ella desde el exterior podría colocar al objeto en un estado de inconsistencia, o los métodos que llamados desde el exterior pueden ponen en peligro el estado del objeto o del programa donde se está ejecutando. Para declarar un miembro privado se utiliza la palabra clave **private** en su declaración.

La clase siguiente contiene una variable miembro y un método privados.

Ejemplo

```
package declaraciones;

public class Alfa {
	private int soyPrivado;
	private void metodoPrivado() {
		System.out.println("metodoPrivado");
	}
}
```

Los objetos del tipo Alfa pueden inspeccionar y modificar la variable soyPrivado y pueden invocar el método metodoPrivado(), pero los objetos de otros tipos no pueden acceder.

Ejemplo

```
package declaraciones;
public class Beta {
	void metodoAccesor() {
		Alfa a = new Alfa();
		a.soyPrivado = 10; // ilegal
		a.metodoPrivado(); // ilegal
	}
}
```

La clase Beta definida aquí no puede acceder a la variable soyPrivado ni al método metodoPrivado() de un objeto del tipo Alfa porque tienen visibilidad **private** y Beta es una clase diferente a Alfa.

Si una clase está intentando acceder a una variable miembro a la que no tiene acceso, el compilador mostrará un mensaje de error similar a este y no compilará su programa:

The field Alfa.soyPrivado is not visible

Y si un programa intenta acceder a un método al que no tiene acceso, generará un error de compilación parecido a este:

The method metodoPrivado() from the type Alfa is not visible

public

El modificador de acceso más sencillo. Todas las clases, en todos los paquetes tienen acceso a los miembros públicos de la clase. Los miembros públicos se declaran sólo si su acceso no produce resultados indeseados si un extraño los utiliza. Por lo tanto, es la visibilidad más fácil de utilizar pero la más difícil de diseñar correctamente. Para declarar un miembro público se utiliza la palabra clave **public**.

Ejemplo

```
package griego;
public class Alfa {
	public int soyPublico;
	public void metodoPublico() {
		System.out.println("metodoPublico");
	}
}
```

Si se vuelve a escribir la clase Beta una vez más y se la coloca en un paquete diferente que la clase Alfa, asegurándose que no están relacionadas (no es una subclase) de Alfa.

Ejemplo

```
package romano;
import griego.*;

public class Beta {
	void metodoAccesor() {
	Alfa a = new Alfa();
	a.soyPublico = 10; // legal
	a.metodoPublico(); // legal
	}
}
```

Como se puede ver en el ejemplo anterior, Beta puede inspeccionar y modificar legalmente la variable soyPublico en la clase Alfa y puede llamar legalmente al método metodoPublico().

Crear Paquetes

Los paquetes son grupos relacionados de clases e interfaces y proporcionan un mecanismo conveniente para manejar un gran juego de clases e interfaces y evitar los conflictos de nombres (porque los paquetes en si mismos definen espacios de nombres). Para crear paquetes de Java se utiliza la sentencia **package**.

Sintaxis de la declaración de un paquete o el uso de otro en el código Java:

[< **package** "nombre del paquete o subpaquete">]
[< **import** "nombre del paquete o subpaquete">]
< class_declaration>+

Ejemplo

```
package clinica.medicos;

import java.util.HashSet;
import clinica.utilidades.Horario;

public abstract class Medico {

	private String nombre;
	private String apellido;
	private String especialidad;
	protected int turnosPorHora = 0;
	private HashSet<Horario> horarios = new HashSet<>();

	public String getNombre() {
		return nombre;
	}

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

	public String getApellido() {
		return apellido;
	}

	public void setApellido(String apellido) {
		this.apellido = apellido;
	}

	public String getEspecialidad() {
		return especialidad;
	}

	public void setEspecialidad(String especialidad) {
		this.especialidad = especialidad;
	}

	public int getTurnosPorHora() {
```

```
            return turnosPorHora;
      }

      public void setTurnosPorHora(int turnosPorHora) {
            this.turnosPorHora = turnosPorHora;
      }
}
```

Nota: La clase HashSet se explicará posteriormente en el módulo de colecciones

Si se está implementando un grupo de clases que representan una serie de objetos gráficos como círculos, rectángulos, líneas y puntos y se quiere que estas clases estén disponibles para otros programadores. Se las puede poner en un paquete, por ejemplo, `graficos` y entregar el paquete a los programadores (junto con alguna documentación de referencia como qué hacen las clases y los interfaces y qué interfaces de programación son públicas).

De esta forma, otros programadores pueden determinar fácilmente para qué es el grupo de clases, cómo utilizarlos, y cómo relacionarlos unos con otros o, también, con otras clases y paquetes. Los nombres de clases no tienen conflictos con los nombres de las clases de otros paquetes porque las clases y los interfaces dentro de un paquete son referenciados en términos de su paquete (técnicamente un paquete crea un nuevo espacio de nombres).

Ejemplo

```
package graficos;

public class Rectangulo {

}
```

```
package graficos;

public class Circulo {

}
```

La primera línea del código anterior crea un paquete llamado `graficos`. Todas las clases definidas en el archivo que contiene esta sentencia son miembros del paquete. Por lo tanto, `Circulo`, y `Rectangulo` son miembros del paquete `graficos`.

Los archivos .class son generados por el compilador cuando se compila el archivo que contienen el código fuente para `Circulo` y `Rectangulo` y deben situarse en un directorio llamado `graficos` en algún lugar del camino indicado en **CLASSPATH**. **CLASSPATH** es una lista de directorios que indica al sistema (máquina virtual) donde se han instalado varias clases compiladas en Java (se explicará bien posteriormente).

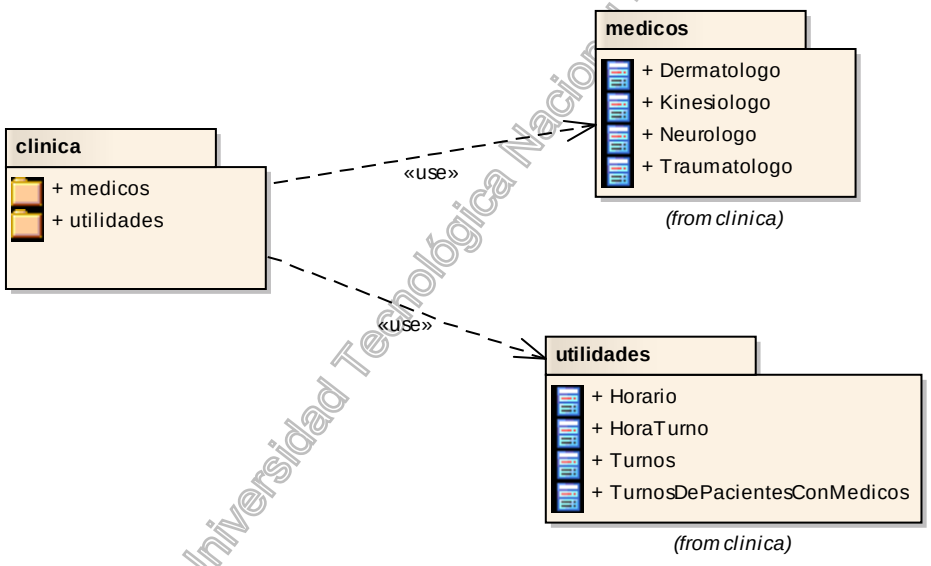
Nota: Dependiendo del sistema operativo, las clases de Java de los paquetes principales para la máquina virtual se pueden encontrar en un directorio prestablecido en un formato especial de archivo llamado JAR (Java ARchive).

Los nombres de paquetes pueden contener varios componentes (separados por puntos). De hecho, los nombres de los paquetes de Java tienen varios componentes: java.util, java.lang, etc... A estos componentes muchas veces se los denomina sub paquetes.

Los paquetes ayudan a manejar grandes sistemas de software porque agrupan clases en su interior que en conjunto pueden representar una funcionalidad específica de un sistema. Por ejemplo, pueden contener clases, subclases e interfaces que definen dicha funcionalidad.

Como se mencionó anteriormente, los paquetes definen espacios de nombres y también una visibilidad para acceder a ellos. Por eso, el último nivel de acceso es el que se obtiene si no se especifica ningún otro nivel de acceso a los miembros. Este nivel de acceso permite que las clases del mismo paquete que la clase tengan acceso a los miembros.

Un ejemplo gráfico en UML del manejo de un paquete, (por creación y uso) se puede ver en la siguiente figura



Otro ejemplo puede ser, una nueva versión de la clase Alfa en la que se declara una variable y un método con acceso de paquete. Alfa reside en el paquete griego:

```
package griego;
public class Alfa {
	public int soyPublico;
	protected int estoyProtegido;

	public void metodoPublico() {
```

```
            System.out.println("metodoPublico");
        }
        protected void metodoProtegido() {
        System.out.println("metodoProtegido");
        }

}
```

La clase Alfa tiene acceso a estoyProtegido y a metodoProtegido().

Además, todas las clases declaradas dentro del mismo paquete que Alfa también tienen acceso a estoyProtegido y metodoProtegido(). Si Alfa y Beta están declaradas como parte del paquete griego podrían Beta acceder a los elementos protegidos de Alfa.

Ejemplo

```
package griego;
public class Beta {
        void metodoAccesor() {
        Alfa a = new Alfa();
        a.estoyProtegido = 10; // legal
        a.metodoProtegido(); // legal
        }
}
```

Entonces Beta puede acceder legalmente a estoyProtegido y metodoProtegido().

Para importar una clase específica o un interfaz al archivo actual se utiliza la sentencia **import**. Esta debe estar al principio del archivo antes que cualquier definición de clase o interfaz, pero después de la declaración **package** que define en donde se guardará la clase compilada en bytecode.

Esto provoca que la clase y/o el interfaz estén disponibles para su uso por las clases y los interfaces definidos en el archivo que se esté elaborando.

Si se quiere importar todas las clases e interfaces de un paquete, por ejemplo, el paquete griego completo, se utiliza la sentencia **import** con un carácter comodín, un asterisco '*':

```
import griego.*;
```

Si intenta utilizar una clase y/o una interfaz desde un paquete que no ha sido importado, el compilador mostrará un error:

Se debe tener en cuenta que sólo las clases y/o interfaces declaradas como públicas pueden ser utilizadas en otras clases fuera del paquete en el fueron definidas. El sistema de ejecución también importa automáticamente el paquete java.lang.

Nota: Es probable que el mismo nombre para una clase se encuentre en más de un paquete. En estos casos se debe colocar el nombre totalmente calificado en al menos una de ellas (la que no se

encuentre mediante la sentencia **import**) para diferenciarlas. El nombre totalmente calificado implica colocar la secuencia de paquetes desde la raíz hasta aquel en que se encuentre la clase en cuestión.

CLASSPATH

Para ejecutar una aplicación Java, se especifica el nombre de la aplicación como el nombre de una clase que en su interior tiene un método **main**, el cual es el que se desea ejecutar en el intérprete Java.

Una aplicación puede utilizar otras clases y objetos que están en las mismas o diferentes localizaciones. Como las clases pueden estar en cualquier lugar, se debe indicar al intérprete Java donde puede encontrarlas. Se puede hacer esto con la variable de entorno CLASSPATH que comprende una lista de directorios que contienen clases Java compiladas.

La construcción de **CLASSPATH** depende de cada sistema. Cuando el intérprete obtiene un nombre de clase, desde la línea de comandos desde una aplicación busca en todos los directorios definidos en **CLASSPATH** hasta que encuentra la clase que está buscando.

Se deberá poner el directorio de nivel más alto que contiene las clases Java en el **CLASSPATH** (se lo puede interpretar como el directorio raíz a partir de donde se encuentran las clases que se desean utilizar). Por convención, muchos programadores tienen un directorio de clases en su directorio raíz donde pone todo su código Java. Si se tiene dicho directorio, se debe poner en el **CLASSPATH**.

Las clases incluidas en el entorno de desarrollo Java están disponibles automáticamente porque este añade el directorio del proyecto al **CLASSPATH** cuando se lo crea.

Observar que el orden es importante. Cuando el intérprete Java está buscando una clase, busca por orden en los directorios indicados en **CLASSPATH** hasta que encuentra la clase con el nombre correcto.

El intérprete Java ejecuta la primera clase con el nombre correcto que encuentre y no busca en el resto de directorios. Normalmente es mejor dar a las clases nombres únicos, pero si no se puede evitar, hay que asegurarse de que el **CLASSPATH** busca las clases en el orden apropiado. Recordar esto cuando se seleccione el **CLASSPATH** y el árbol de paquetes declarados en el código fuente.

Nota: Todas las clases e interfaces (el tema interfaces se verá posteriormente) pertenecen a un paquete. Incluso si no especifica uno con la sentencia **package** (en ese caso, van en el directorio raíz definido en el **CLASSPATH**). Si no se especifica las clases e interfaces se convierten en miembros del paquete por defecto (directorio raíz), que no tiene nombre y que siempre es visible. Esto se debe evitar porque cualquier clase que quiera utilizar una descarada en el paquete por defecto no puede accederla porque no tiene paquete que importar.

Constructores

Todas las clases Java tienen métodos especiales llamados constructores que se utilizan para inicializar un objeto nuevo de ese tipo. Se puede declarar e implementar un constructor como se haría con cualquier otro método en una clase, salvo por un detalle, no se tiene que especificar el valor de retorno del constructor porque esta predefinido (devuelve la referencia al objeto que se crea).

Los constructores tienen el mismo nombre que la clase, por ejemplo, el nombre del constructor de la clase Rectangulo es Rectangulo(), el nombre del constructor de la clase Thread es Thread(), etc...

Ejemplo

```
package constructores;
public class Ejemplo {
	private int atrib1;
	private char atrib2;
	Ejemplo(int a){ atrib1 = a;}
	public void metodo(){
		//[sentencias;]
	}
}
```

La clase Ejemplo no tiene constructor por defecto. Si se quiere diseñar la clase para que lo posea, se debe declarar explícitamente.

Las clases tienen siempre un constructor explícito o implícito. Si no se declara uno explícitamente, el lenguaje lo agrega sin argumentos, lo cual quiere decir que si no se agrega la definición un constructor sería como si la clase tuviera declarado en su interior.

```
Ejemplo();
```

Si el constructor es explícito debe tener el mismo nombre de la clase y no retornar ningún valor, pero cuando se agrega un constructor explícito, el lenguaje no agrega el constructor por defecto, lo que implica para tenerlo que se debe declarar en el código.

Ejemplo

```
package constructores.defecto;

public class Ejemplo {
	private int atrib1;
	private char atrib2;
	Ejemplo(int a){ atrib1 = a;}
	Ejemplo(){ atrib1 = 5;}
	public void metodo(){
		//[sentencias;]
	}
}
```

Cuando se declaren constructores para las clases, se pueden utilizar los modificadores de acceso que determinan si otros objetos pueden crear otros de su tipo:

- **private**
  - Ninguna otra clase puede crear un objeto de su clase. La clase puede contener métodos públicos estáticos y esos métodos pueden construir un objeto y devolverlo, pero nada más.
- **protected**
  - Sólo las subclases de la clase o aquellas que se encuentren en el mismo paquete pueden crear objeto de ella.
- **public**
  - Cualquiera pueda crear un objeto de la clase.
- **Acceso por defecto (a nivel de paquete)**
  - Nadie externo al paquete puede construir un objeto de su clase. Esto es muy útil si se quiere que las clases que tenemos en un paquete puedan crear objetos de la clase pero no se quiere que lo haga nadie más.

Java soporta poner más de un constructor en una clase gracias a la sobrecarga de los nombres de métodos, tema que se explicará posteriormente. Para que una clase pueda tener cualquier número de constructores, todos los cuales tienen el mismo nombre. Si este es el caso, los mismos deben diferenciarse unos de otros cumpliendo al menos una de las siguientes condiciones en sus argumentos:

- El número
- El tipo (para la misma posición de un argumento en la lista)

La clase Ejemplo a continuación, proporciona dos constructores diferentes, ambos llamados Ejemplo(), pero cada uno con número o tipo diferentes de argumentos a partir de los cuales se puede crear un nuevo objeto Ejemplo. Particularmente en este caso, sólo cambia el tipo de argumento

Ejemplo

```
package constructores.sobrecargados;
public class Ejemplo {
	private int atrib1;
	private char atrib2;
	Ejemplo(int a){ atrib1 = a;}
	Ejemplo(char a){ atrib2 = a;}
	public void metodo(){
		//[sentencias;]
	}
```

```
}
```

Un constructor utiliza sus argumentos para inicializar el estado del nuevo objeto. Entonces, cuando se crea un objeto, se debe elegir el constructor cuyos argumentos reflejen mejor cómo se quiere inicializar dicho objeto.

Basándose en el número y tipos de los argumentos que se pasan al constructor, el compilador determina cual de ellos utilizar, Así el compilador sabe que cuando se escribe:

```
new Ejemplo (200);
```

Utilizará el constructor que requiere un argumento entero, y cuando se escribe:

```
new  Ejemplo ('a');
```

Utilizará el constructor que requiere como argumento un carácter.

Sobrecarga de Métodos

Java soporta la sobrecarga de métodos, por eso varios métodos pueden compartir el mismo nombre. Por ejemplo, si se ha escrito una clase que puede proporcionar varios tipos de datos (cadenas, enteros, etc...) en un área de dibujo, se podría escribir un método que supiera como tratar a cada tipo de dato. En otros lenguajes no orientados a objetos, se tendría que pensar un nombre distinto para cada uno de los métodos. dibujaCadena(), dibujaEntero, etc...

En Java, se puede utilizar el mismo nombre para todos los métodos pasándole un tipo de parámetro diferente a cada uno de los métodos. Entonces en la clase de dibujo, se podrán declarar tres métodos llamados dibujar() y que cada uno aceptara un tipo de parámetro diferente.

Ejemplo

```
package sobrecarga;
public class DibujodeDatos {
	void dibujar (String s) {
		/*
		 * . . .
		 */
		}
		void dibujar (int i) {
		/*
		 * . . .
		 */
		}
		void dibujar (float f) {
		/*
		 * . . .
		 */
		}
```

```
}
```

Los métodos son diferenciados por el compilador basándose en el número y tipo de sus argumentos. Para ello, redefine internamente los nombres de los métodos valiéndose de las declaraciones de parámetros. De esta manera, los métodos internamente serían:

```
dibujarString
dibujarint
dibujarfloat
```

Por lo tanto internamente son todos distintos. Generalizando la forma de uso, suponiendo las siguientes declaraciones:

```
public void metodo(float f, int i);
public void metodo(float f);
public void metodo(String s);
```

Internamente serían:

```
metodofloatint
metodofloat
metodoString
```

Motivo por el cual se deben poner los nombres con diferente cantidad o tipo de argumentos.

Nota: Los valores devueltos por un método no los diferencian en la sobrecarga. Por lo tanto dos métodos con distinto valor retornado pero el mismo tipo de argumentos, genera un error. Por otra parte, si los argumentos son diferentes, no importa si el valor retornado es igual o distinto.

Uso de `this`

La palabra reservada **`this`** posee la referencia al objeto actualmente en ejecución y se utiliza para encontrar los elementos que pertenecen a dicho objeto. Como se mencionó anteriormente, este valor se almacena en el stack para cada método en ejecución y es por esta referencia que se encuentran las variables de instancia de un objeto.

Sólo los elementos de un objeto, métodos y atributos, poseen asociada una referencia del tipo **`this`**, por lo tanto se lo puede utilizar para resolver ambigüedades.

Normalmente, dentro del cuerpo de un método de un objeto se puede referir directamente a los atributos del mismo. Sin embargo, algunas veces no se querrá tener ambigüedad sobre el nombre de la variable de instancia y uno de los argumentos del método que tengan el mismo nombre.

Por ejemplo, si una clase tiene declarados atributos que tienen el mismo nombre que los argumentos de un método, utilizando **`this`** se resuelve la ambigüedad.

Ejemplo

```
package ambiguedad;
public class Ejemplo {
```

```
	private int atrib1;
	private char atrib2;
	Ejemplo(int a){ atrib1 = a;}
	Ejemplo(char a){ atrib2 = a;}
	Ejemplo (char atrib2,int atrib1){
		this.atrib1 = atrib1;
		this.atrib2 = atrib2;
	}
	public void metodo(){
		//[sentencias;]
		atrib2 = 'a';
	}
}
```

Otro punto importante a tener en cuenta es que todo elemento que pertenece a un objeto tiene asociado una referencia **this** porque esta es la forma de encontrar los miembros que pertenecen a cada objeto en particular. Pero se debe ser cuidadoso al evaluar esta afirmación. Por ejemplo, un método de un objeto utiliza **this** para encontrar las variables de instancia y por lo tanto siempre tiene un **this** asociado. Sin embargo, los parámetros y variables locales del mismo no tienen asociado un **this** ya que se alojan en el stack. Si un método no tiene un **this** asociado, no pertenece al objeto por más que este declarado dentro de la clase que lo define. Este es el caso de los métodos estáticos como se vera posteriormente.

Uso de `this` en Constructores

Se puede utilizar **this** en la primera línea de un constructor para invocar otro constructor de la misma clase.

Como se mencionó anteriormente, **this** almacena la referencia al objeto actualmente en ejecución. Si se está construyendo el objeto, la suma de esta palabra reservada con el operador de llamado a función, Java lo resuelve invocando al constructor que posee los argumentos pasados como parámetros dentro el operador de llamado a función utilizado con **this**

Ejemplo

```
package constructores.conthis;
public class Ejemplo {
	Ejemplo(int a){ atrib1 = a;}
	Ejemplo(int a, float b){
			this(a);
			atrib2 = b;
	}
	private int atrib1;
	private float atrib2;
	public void metodo(){
		//[sentencias;]
	}
}
```

Por lo visto hasta el momento, se puede aprovechar la sobrecarga y las características de **this** en el llamado a constructores de la propia clase para utilizarlos en conjunto.

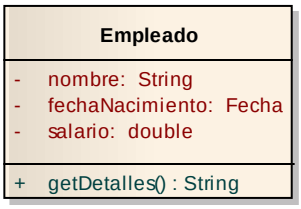
Ejemplo

```
package constructores.conthis;

public class Empleado {
	private String nombre;
	private Fecha fechaNacimiento;
	private double salario;
	private static final double SALARIO_BASE = 15000.00;

	// Constructor
	public Empleado(String nombre, Fecha fDN, double salario) {
		this.nombre = nombre;
		this.fechaNacimiento = fDN;
		this.salario = salario;
	}

	public Empleado(String nombre, double salario) {
		this(nombre, null, salario);
	}

	public Empleado(String nombre, Fecha fDN) {
		this(nombre, fDN, SALARIO_BASE);
	}

	public Empleado(String nombre) {
		this(nombre, SALARIO_BASE);
	}
}
```

Herencia

Para comenzar a desarrollar este tema, primero se plantea la necesidad del mismo desde el punto de vista del código. Para esto se supone una clase simple con atributos y métodos como se describe a continuación en el diagrama UML y el código que se deriva de él.

Ejemplo




```
package herencia;
```

```
import constructores.conthis.Fecha;
public class Empleado {
        private String nombre;
        private Fecha fechaNacimiento;
        private double salario;
        public String getDetalles(){ return "detalles";}
}
```

En Java, como en otros lenguajes de programación orientados a objetos, 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.

Las subclases heredan el estado y el comportamiento en forma de las variables y los métodos de su superclase. La subclase puede utilizar los ítems heredados de su superclase tal y como son, o puede modificarlos o rescribirlos. Por eso, según se recorre la cadena de la herencia, las clases se convierten en más y más especializadas.

> *Una subclase es una clase que desciende de otra clase. Una subclase hereda el manejo del estado y el comportamiento de su superclase.*

Declarar la Superclase de la Clase

Se declara que una clase es una subclase de otra clase en la declaración de la misma clase. Por ejemplo, si se quiere crear una subclase llamada SubClase de otra clase llamada SuperClase, la forma de escribirlo es:

```
class SubClase extends SuperClase {
. . .
}
```

Esto declara que SubClase es una subclase de SuperClase. Y también declara implícitamente que SuperClase es la superclase de SubClase. Una subclase también hereda variables y miembros de las superclases de su superclase, y así a lo largo de la cadena de herencia. Para hacer esta explicación un poco más sencilla, *cuando se haga referencia a la superclase de una clase significa el ancestro más directo* (la clase que figura en la declaración luego del **extend**).

> *Cuando se haga referencia a una clase que antecede a otra se la denomina superclase de la subcadena de herencia. Si la superclase en particular es la primera de la cadena de herencia, se la denomina superclase de la cadena de herencia.*

Para especificar explícitamente la superclase de una clase, se debe poner la palabra clave **extends** más el nombre de la superclase entre el nombre de la clase que se ha creado y la llave de comienzo del cuerpo de la clase, como se mencionó anteriormente y se muestra a continuación.

Ejemplo

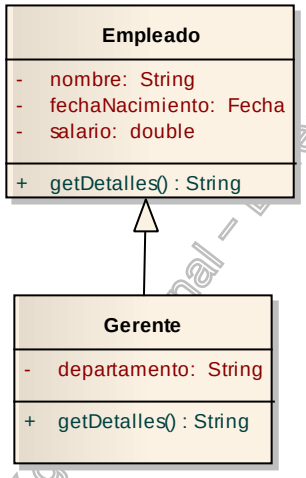
```
class Gerente extends Empleado{
      . . .
}
```

Esto declara explícitamente que la clase Empleado es la superclase de Gerente. Declarar esto implica implícitamente que Gerente es una subclase de Empleado. Una subclase hereda las variables y los métodos de su superclase.

Crear una subclase puede ser tan sencillo como incluir la cláusula **extends** en su declaración de clase. Sin embargo, se tendrán que tener en cuenta otras posibilidades que se derivan de esto en su código cuando se crea una subclase, como rescribir métodos, lo cual se explicará posteriormente.

Basándose en los ejemplos anteriores, el código para las clases Empleado y Gerente se puede plantear como muestra el diagrama UML y el código que se deriva de él.

Ejemplo



Una clase Java sólo puede tener una superclase directa. Java no soporta la herencia múltiple.

Crear una subclase puede ser tan sencillo como incluir la cláusula **extends** en la declaración de la clase. Sin embargo, normalmente se deberá realizar alguna cosa más cuando se crea una subclase, como sobrescribir métodos, etc...

Existe un mecanismo que simula una herencia múltiple a través de declarar interfaces, pero este es un tema que se expondrá posteriormente

Se puede definir la sintaxis apropiada de la herencia, basado en la sintaxis UML, de la siguiente manera:

```
< modificador> class <nombre> [extends < superclase>] {
                  < declaraciones>*
}
```

¿Qué variables miembro hereda una subclase?

*Una subclase hereda todas las variables de instancia de su superclase que puedan ser **visibles** desde la subclase (a menos que el atributo se oculte por una declaración).*

Esto es, las subclases:

- Heredan aquellas variables miembros declaradas como `public` o `protected`
- Heredan aquellas variables miembros declaradas sin modificador de acceso siempre que la subclases esté en el mismo paquete que la clase que se define
- No hereda las variables de instancia de la superclase si la subclase declara una variable que utiliza el mismo nombre. El atributo de la subclase oculta a la variable miembro de la superclase.
- No hereda las variables miembro `private`

Ocultar Variables Miembro

Como se mencionó en la sección anterior, los atributos definidos en la subclase ocultan las variables miembro o de instancia que tienen el mismo nombre en la superclase.

Así como esta característica del lenguaje Java es poderosa y conveniente, también puede ser una fuente de errores: ocultar una variable miembro puede hacerse deliberadamente o por accidente. Entonces, cuando se use nombres para las variables de instancia hay que ser cuidadoso y ocultar sólo aquellas que realmente se desean.

Una característica interesante de los atributos en Java es que una clase puede acceder a aquella que este oculta en su superclase gracias a que se puede resolver la visibilidad. Por ejemplo, observar las declaraciones en esta pareja de superclase y subclase:

Ejemplo

```
package herencia;

public class Super {
	Number unNumero;
}
```

```
package herencia;

public class Sub extends Super {
	float unNumero;

	public Sub() {
		Number ej = super.unNumero;
	}
}
```

La variable unNumero de Sub oculta a la variable unNumero de Super. Pero se puede acceder a la variable de la superclase utilizando:

```
super.unNumero
```

super es una palabra clave del lenguaje Java que permite a un método referirse a las variables ocultas y métodos sobrescritos de una superclase.

¿Qué métodos hereda una Subclase?

La regla que especifica los métodos heredados por una subclase es similar a la de las variables miembro.

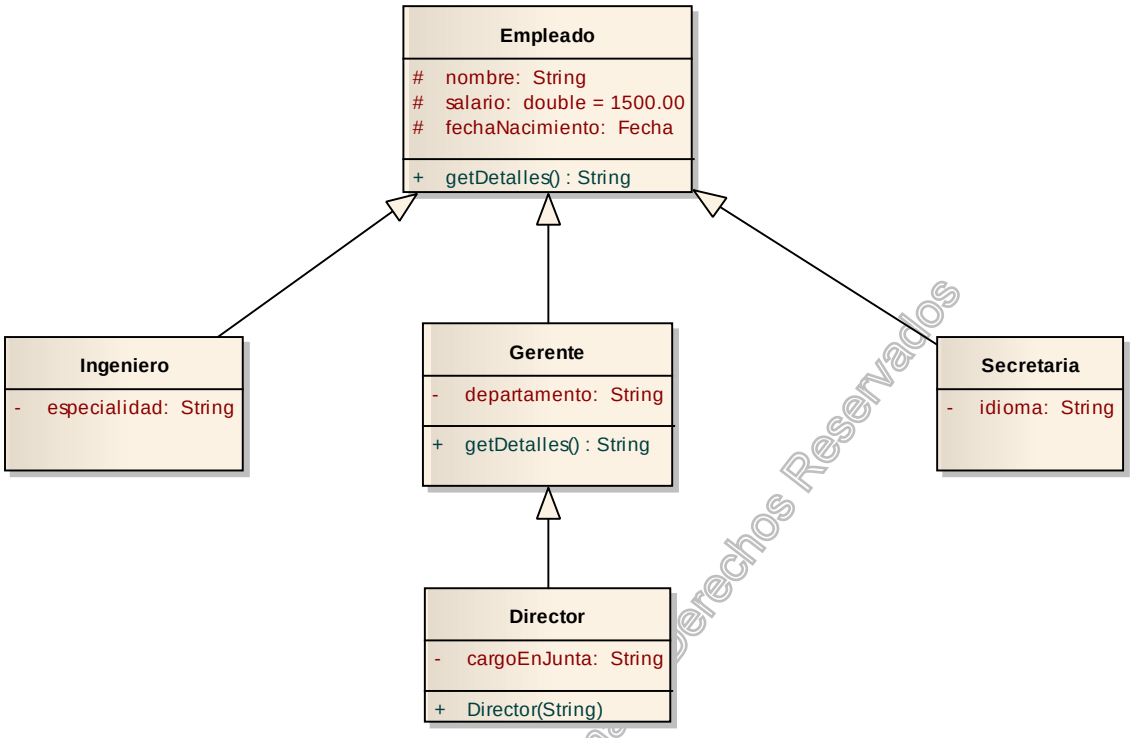
> *Una subclase hereda todos los métodos de su superclase que son **visibles** para la subclase (a menos que el método sea sobrescrito por la subclase).*

Esto es, una Subclase:

- Hereda aquellos métodos declarados como **public** o **protected**
- Hereda aquellos métodos sin modificador de acceso, siempre que la subclase esté en el mismo paquete que la clase
- No hereda un método de la superclase si la subclase declara un método que utiliza el mismo nombre. Se dice que el método de la subclase sobrescribe al método de la superclase.
- No hereda los métodos **private**

En Java, una clase puede heredar de tan sólo una superclase. Sin embargo, una superclase lo puede ser de muchas superclase y formar así distintas cadenas de herencia. Por ejemplo, en el siguiente gráfico UML se muestran tres cadenas de herencia que convergen a la misma superclase.

Ejemplo



El código que representa a este gráfico es el siguiente

```
package herencia;

import constructores.conthis.Fecha;
public class Empleado {
	protected String nombre;
	protected double salario = 1500.00;
	protected Fecha fechaNacimiento;

	public String getDetalles() {
		return "Nombre: " + nombre + "\nSalario: " + salario;
	}
}


package herencia;
class Gerente extends Empleado {
	private String departamento;

	public String getDetalles() {
		return "detalles";
	}
}
```

```
package herencia;
public class Director extends Gerente {
	private String cargoEnJunta;

	public Director(String c) {
		cargoEnJunta = c;
	}
}
```

```
package herencia;
public class Secretaria extends Empleado {
	private String idioma;
}
```

```
package herencia;
public class Ingeniero extends Empleado {
	private String especialidad;
}
```

La siguiente tabla le muestra los niveles de acceso permitidos por cada modificador:

Modificador	Misma Clase	Mismo Paquete	Subclase	Universo
`private`	Si	No	No	No
defecto	Si	Si	No	No
`protected`	Si	Si	Si	No
`public`	Si	Si	Si	Si

La segunda columna indica si la propia clase tiene acceso al miembro definido por el modificador de acceso. La tercera columna indica si las clases del mismo paquete que la clase. La cuarta columna indica si las subclases de la clase (sin importar dentro de que paquete se encuentren estas) tienen acceso a los miembros. La quinta columna indica si todas las clases tienen acceso a los miembros.

El modificador `protected`

Permite a la propia clase, a las clases del mismo paquete y las subclases que accedan a los miembros declarados con este modificador. Este nivel de acceso se utiliza cuando es apropiado para una subclase de la clase tener acceso a los miembros o una que este relacionada con ella a nivel de paquete (recordar que los paquetes definen espacios de nombres que deberían utilizarse para agrupar clases que se utilicen con un mismo fin)

Para declarar un miembro protegido, se utiliza la palabra clave `protected`. Primero se evaluará cómo afecta este modificador de acceso a las clases del mismo paquete. Teniendo en cuenta el siguiente ejemplo, la clase Alfa se declara para estar incluida en el paquete griego. Además, tiene una variable y un método que son miembros protegidos:

Ejemplo

```
package griego;
public class Alfa {
	public int soyPublico;
	protected int estoyProtegido;

	public void metodoPublico() {
		System.out.println("metodoPublico");
	}
	protected void metodoProtegido() {
	System.out.println("metodoProtegido");
	}
}
```

Por otra parte, la clase Gama, también está declarada como miembro del paquete griego (y no es una subclase de Alfa). La clase Gama puede acceder legalmente al miembro estoyProtegido del objeto Alfa y puede llamar legalmente a su método metodoProtegido():

Ejemplo

```
package griego;

public class Gama {
	public void metodoAccesor() {
		Alfa a = new Alfa();
		a.estoyProtegido = 10; // legal
		a.metodoProtegido(); // legal
	}
}
```

Nota: ***Se debe tener en cuenta que para acceder al miembro protegido se crea un nuevo objeto y se utiliza la referencia con notación de punto a este como si fuera un miembro público. Esto sólo es posible porque se encuentra en el mismo paquete.***

Para ver como afecta el modificador **protected** a una subclase de Alfa, se introduce una nueva clase, Delta, que es subclase de Alfa pero reside en un paquete diferente, latin. La clase Delta puede acceder tanto a estoyProtegido como a metodoProtegido(), pero solo en objetos del tipo Delta o sus subclases. La clase Delta no puede acceder a estoyProtegido o metodoProtegido() en objetos del tipo Alpha. metodoAccesor() en el siguiente ejemplo intenta acceder a la variable miembro estoyProtegido de un objeto del tipo Alpha, que es ilegal, y en un objeto del tipo Delta que es legal. Similarmente, metodoAccesor() intenta invocar a metodoProtegido() en un objeto del tipo Alfa, que también es ilegal:

Ejemplo

```
package latin;
import griego.Alfa;
public class Delta extends Alfa {
	void metodoAccesor(Alfa a, Delta d) {
		a.estoyProtegido = 10; // ilegal
		d.estoyProtegido = 10; // legal
```

```
            a.metodoProtegido(); // ilegal
            d.metodoProtegido(); // legal
        }
}
```

Cabe destacar que al invocar al método protegido como d.estoyProtegido, se llama al método del objeto que se recibe como referencia. Si en cambio el llamado hubiese sido tan sólo estoyProtegido, el llamado también seria legal pero se invocaría al método a través de la visibilidad que tiene Delta por heredar de Alfa, y no al método de la referencia d.

Si una clase es una subclase que se encuentra en el mismo paquete de la clase con el miembro protegido, la clase tiene acceso a éste, ya sea por acceso directo como se explicó anteriormente en el caso de una subclase, o por notación de punto si se recibe como parámetro como consecuencia de estar en el mismo paquete.

Sobrescribir métodos

En muchas oportunidades, el comportamiento de una operación definida en la superclase no es adecuado y se debe modificar para el servicio que brindará la subclase. Una subclase puede sobrescribir completamente la implementación de un método heredado o puede mejorar el método añadiéndole funcionalidad y cambiar así el servicio que presta de manera que sea adecuado a su implementación.

> *Cuando se sobrescribe un método en una subclase, se oculta la visibilidad del método que existe en la superclase.*

Remplazar la implementación de un método de una superclase

Algunas veces, una subclase puede necesitar remplazar completamente la implementación de un método de su superclase. De hecho, muchas superclases proporcionan implementaciones de métodos vacías con el propósito que la mayoría, si no todas, sus subclases remplacen completamente la implementación de ese método.

Un ejemplo de esto es el método run() de la clase Thread. La clase Thread proporciona una implementación vacía (el método no hace nada) para el método run(), porque por definición, este método depende de la subclase.

En muchas oportunidades, cuando se define una superclase se sabe que las subclases deberán proveer un determinado servicio, pero los detalles de la implementación del método que lo brinda se tendrán cuando se especialice la clase (se cree la subclase) que brinde ese mencionado servicio. La clase Thread no puede proporcionar una implementación del método run(), ya que cada clase que herede de ella tendrá el conocimiento de cómo implementar el código particular para este método (el cual se ejecutará cuando comience el thread)

Para remplazar completamente la implementación de un método de la superclase, simplemente se llama a un método con el mismo nombre que el del método de la superclase y se sobrescribe el método con la misma firma (prototipo) que la del método sobrescrito. Por ejemplo:

Ejemplo

```
package rescritura;

public class ThreadSegundoPlano extends Thread {
	public void run() {
		// sentencias de esta clase
	}
}
```

La clase ThreadSegundoPlano sobrescribe completamente el método run() de su superclase y remplaza completamente su implementación.

Añadir implementación a un método de la superclase

En otro tipo de situaciones, una subclase debe mantener la implementación del método de su superclase y posteriormente ampliar algún comportamiento específico en la subclase. Por ejemplo, los métodos constructores de una subclase lo hacen normalmente, la subclase debe preservar la inicialización realizada por la superclase ya que sólo cada clase sabe como construir los objetos de su tipo, pero puede proporcionar inicialización adicional específica de la subclase.

Métodos que no se pueden sobrescribir en una subclase

Una subclase no puede sobrescribir métodos que hayan sido declarados como **final** en la superclase (por definición, los métodos finales no pueden ser sobrescritos). Si intentamos sobrescribir un método final, el compilador mostrará un mensaje de error y no compilará el programa:

Una subclase tampoco pude sobrescribir métodos que se hayan declarado como **static** en la superclase. En otras palabras, una subclase no puede sobrescribir un método de clase (forma en la cual se llama a los métodos **static** que se verán posteriormente).

Métodos que una subclase debe sobrescribir

Las subclases deben sobrescribir aquellos métodos que hayan sido declarados como **abstract** en la superclase, o la propia subclase debe ser abstracta. Escribir clases y métodos abstractos se explica con más detalle posteriormente.

Sobrescribir un método abstracto significa darle funcionalidad.

Ejemplo

```
package abstractos;

import constructores.conthis.Fecha;
```

```
public abstract class Empleado {
	protected String nombre;
	protected double salario = 1500.00;
	protected Fecha fechaNacimiento;

	public abstract String getDetalles();

}

package abstractos;
class Gerente extends Empleado {
	private String departamento;

	public String getDetalles() {
		return "Nombre: " + nombre + "\nSalario: " + salario;
	}
}
```

Más sobre super

La palabra reservada **super** se usa para resolver visibilidad. Para entender la mecánica de esto, se debe comprender como se ocultan elementos de una superclase en una subclase, lo cual sucede cuando tienen la superclase o la subclase declarados exactamente el mismo identificador, ya sea un atributo o un método de la clase. Este hecho, como se explicará posteriormente se denomina rescritura y el resultado de ella es ocultar los miembros de la superclase.

Si una declaración oculta alguna variable miembro o método de la superclase, se puede referir a estos utilizando **super** con notación de punto. De esta manera se pueden acceder miembros con visibilidad en la superclase, ya sean atributos o métodos.

Ejemplo

```
package rescritura;

public class MiClase {
	boolean unaVariable;

	void unMetodo() {
		unaVariable = true;
	}
}
```

Y la declaración de la subclase que oculta unaVariable y sobrescribe unMetodo():

```
package rescritura;

public class OtraClase extends MiClase {
	boolean unaVariable;

	void unMetodo() {
		unaVariable = false;
		super.unMetodo();
		System.out.println(unaVariable);
		System.out.println(super.unaVariable);
```

```
	}
}
```

Primero unMetodo()inicializa unaVariable (la que esta declarada en OtraClase que oculta a la declarada en MiClase) a **false**. Luego unMetodo()llama a su método sobrescrito con esta sentencia:

```
super.unMetodo();
```

Esto inicializa la versión oculta de unaVariable (la declarada en MiClase) a **true**. Luego unMetodo() muestra las dos versiones de unaVariable con diferentes valores:

- **False**
- **true**

Para invocar un método de la superclase se puede utilizar llamados como se muestran en el siguiente código.

Ejemplo

```
package rescritura;
import constructores.conthis.Fecha;
public class Empleado {
	protected String nombre;
	protected double salario = 1500.00;
	protected Fecha fechaNacimiento;

	public String getDetalles() {
		return "Nombre: " + nombre + "\nSalario: " + salario;
	}
}

package rescritura;
class Gerente extends Empleado {
	private String departamento;

	public String getDetalles() {
		return super.getDetalles() + "\nDepartamento: " + departamento;
	}
}
```


Constructores en la herencia

Los constructores no se heredan, son propios de cada clase, lo cual implica que la construcción de un objeto es pura responsabilidad del tipo al que pertenece (clase).

Una subclase hereda métodos y atributos de la superclase que se podrán acceder por visibilidad directa o resolviéndola con **super**. Esta última sentencia también se puede invocar en un constructor, donde este se utiliza con paréntesis, o mejor dicho, con el operador de llamado a función que invoca al constructor de la superclase.

Se deberá acceder al constructor de la clase base en la primera línea del constructor de la subclase, salvo que exista un constructor por defecto (sin argumentos) en cuyo caso se accede a él automáticamente.

Cuando se construye un objeto de una subclase, se debe invocar al constructor de la superclase y pasarle los parámetros necesarios para la inicialización que esta requiera. Por ejemplo, el siguiente código invoca en la construcción de un objeto de tipo Gerente al constructor adecuado de tipo Empleado, pasándole el parámetro necesario para la construcción de la superclase.

Ejemplo

```
package herencia.constructores;

import constructores.conthis.Fecha;
public class Empleado {
	private String nombre;
	private double salario = 1500.00;
	private Fecha fechaNacimiento;
	public Empleado(String n, Fecha fDN) {
		// Llamado implícito a super();
	nombre = n;
	fechaNacimiento = fDN;
	}
	public Empleado(String n) {
		this(n, null);
	}
}


package herencia.constructores;
public class Gerente extends Empleado {
	private String departamento;
	public Gerente(String n, String d) {
		super(n);
		departamento = d;
	}
}
```


Existen situaciones en las cuales una superclase puede ser construida de diversas maneras. Es responsabilidad de la subclase utilizar el constructor apropiado en cada oportunidad. Si una superclase tiene una diversidad de constructores, la subclase deberá seleccionar entre ellos el que se ajuste al tipo de construcción que realice para los objetos de su tipo (el de la subclase). Por ejemplo, si se tiene la siguiente superclase:

Ejemplo

```
package herencia.constructores.conthis;

import constructores.conthis.Fecha;
public class Empleado {
	private String nombre;
```

```
	private static final double SALARIO_BASE = 1500.00;
	private double salario = 1500.00;
	private Fecha fechaNacimiento;

	public Empleado(String nombre, double salario, Fecha fDN) {
			this.nombre = nombre;
			this.salario = salario;
			this.fechaNacimiento = fDN;
		}
	public Empleado(String nombre, double salario) {
		this(nombre, salario, null);
	}
	public Empleado(String nombre, Fecha fDN) {
		this(nombre, SALARIO_BASE, fDN);
	}
	public Empleado(String nombre) {
		this(nombre, SALARIO_BASE);
	}
		// más código de Empleado...
}
```

Una subclase para construir un objeto que tenga como superclase a la anterior, puede seleccionar los constructores de la superclase a través de los parámetros con que se invoque a **super**. Si la cantidad de parámetros o el tipo de los mismos no se puede asociar a un constructor de la superclase, derivará en un error en tiempo de compilación, por ejemplo:

Ejemplo

```
package herencia.constructores.conthis;
public class Gerente extends Empleado {
	private String departamento;

	public Gerente(String name, double salario, String dep) {
		super(name, salario);
		departamento = dep;
		}
	public Gerente(String n, String d) {
		super(n);
		departamento = d;
	}
	public Gerente(String dep) { // Error, no existe Empleado()
		departamento = dep;
	}
}
```


En resumen, los objetos se construyen e inicializan según los llamados a los constructores en una cadena de herencia y la manera en que estos construyan cada respectiva superclase. El proceso completo de construcción e inicialización se puede resumir en lo siguiente:

- Se aloja memoria y ocurren las inicializaciones por defecto

- La inicialización de una variable de referencia utiliza estos pasos recursivamente cuando se construye una cadena de herencia:
  1. Enlazar los parámetros del constructor
  2. Si se llama explícitamente a **this**(), realizarlo recursivamente y luego saltar al paso 5
  3. Llamar recursivamente las invocaciones a **super** explícita o implícitas
  4. Ejecutar las inicializaciones explícitas de las variables de instancia
  5. Ejecutar el bloque de sentencias dentro del constructor actual

El operador instanceof

Este operador sólo puede usarse con variables de referencia a un objeto. El objetivo del operador **instanceof** es determinar si un objeto es de un tipo determinado. Por tipo se entiende a clase o interfaz (interface), es decir si responde a las palabras calificadoras "es un" para esa clase o interfaz, especificado a la derecha del operador.

Ejemplo

```
package instancia;

public class DeterminarInstancia {
	public static void main(String[] args) {
		Empleado empleado = new Empleado("Alejandro", "Ignacio",
				"Fursi","17.767.076",
				"Ventas","0","GV50",5500.0F);
		Gerente gerente = new Gerente("Juan", "Pedro",
				"Goyena","17.767.076",
				"Ventas","0","GV50",5500.0F,20);
		Director director = new Director("Daniel", "Federico",
				"Ruiz","17.767.076",
				"Ventas","0","GV50",5500.0F,20, "vocal");
		Secretaria secretaria = new Secretaria("María", "Juana", "Roldán",
				"20.202.020","Ventas", "20","10", 800.0F);

		if (empleado instanceof Empleado)
			System.out.println("Empleado es una instancia de Empleado");
		if (empleado instanceof Persona)
			System.out.println("Empleado es una instancia de Persona");
		if (secretaria instanceof Empleado)
			System.out.println("Secretaria es una instancia de
				Empleado");
		if (gerente instanceof Empleado)
			System.out.println("Gerente es una instancia de Empleado");
		if (director instanceof Empleado)
			System.out.println("Director es una instancia de Empleado");
		if (director instanceof Empleado)
			System.out.println("Director es una instancia de Empleado");
		if (director instanceof Persona)
			System.out.println("Director es una instancia de Persona");

		// Esto es un error de compilación
		// if(ingeniero instanceof Gerente)
		// System.out.println("e es una instancia de Empleado");
	}
```

```
}
```

La salida es:

```
Empleado es una instancia de Empleado
Empleado es una instancia de Persona
Secretaria es una instancia de Empleado
Gerente es una instancia de Empleado
Director es una instancia de Empleado
Director es una instancia de Empleado
Director es una instancia de Persona
```

Notar que si la clase se encuentra en la misma sub cadena de herencia y el objeto es de ese tipo, el operador retorna **`true`**.