Sumario

Programación Orientada a objetos

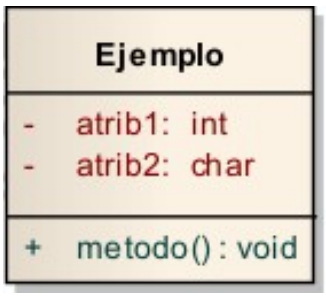
Clases

Declaraciones básicas

Las clases en Java (Java Class) son plantillas para la creación de objetos (Tipos en Java), en lo que se conoce como programación orientada a objetos, la cual es una de los principales paradigmas de desarrollo de software en la actualidad. Representan ideas genéricas del mundo real y se la suele detectar como sustantivos en la realidad de negocio a modelar.
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.

```
//Declaración de una clase en Java
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:



Miembros de una clase

Una clase tiene miembros, estos pueden ser atributos o métodos, se describirán mas adelante.

Qué es un objeto

Definición
Un objeto es una instancia de una clase. Como ejemplo, una instancia de la clase Persona, podria ser una persona llamada Mario. Puede tenerse varias instancias de una misma clase, pero una única clase de Persona, una persona Mario, y una persona Maria, con ambas instancias de la clase Persona.

Ejemplos de cómo codificar diferentes objetos en Java:

```
Persona mario = new Persona();
Persona maria = new Persona();
```

Atributos

Definición
Los atributos determinan el estado que tiene un objeto. Estos pueden ser tipos de datos primitivos u objetos de cualquier clase.

Ejemplo:
La clase Persona tiene como atributos la altura, el color de ojos y la edad.
Ejemplo en Java:

```
public class Persona{
	public int altura;
	public String colorDeOjos;
	public int edad;
}
```

Utilización:

```
Persona mario = new Persona();
mario.altura = 183;
mario.colorDeOjos = "marrones";
mario.edad = 40;
```

Métodos

Los métodos, o como se los llama en otros lenguajes “funciones”, son los elementos de una clase mediante los cuales se define una operación que un objeto de su tipo puede realizar.

La declaración de un método puede adoptar el siguiente formato:

```
[<modificador>] <tipo retornado> <identificador>([lista de argumentos]){
	[declaraciones y / o métodos a utilizar]
	[ return [valor retornado];]
}
```

Si bien el formato de la declaración del método es incompleta (falta la posibilidad de declarar excepciones), se usará así de momento porque la parte faltante es un tema avanzado para este punto.

El formato quiere decir lo siguiente:

Modificador: Opcional. Es el modificador de visibilidad del método (indica como se lo puede utilizar y desde donde). Las posibilidades son:

```
public (pertenece a la interfaz. Se accede con notación de punto)
private (pertenece a la clase. No se accede con notación de punto)
protected (pertenece a la cadena de herencia y se verá
            posteriormente. No se accede con notación de punto)
Sin modificador (pertenece al paquete. Se accede si esta en el mismo
            paquete)
```

El formato quiere decir lo siguiente: Tipo retornado: se debe elegir entre un tipo primitivo o uno referenciado. Si el método no devuelve ningún valor se puede indicar poniendo en este lugar la palabra clave void .

Identificador: Es el nombre que se le asignará al método y mediante el cual se lo invocará.

Lista de argumentos: En este lugar se indican los parámetros que recibirá el método. Los argumentos se declaran igual que las declaraciones de variables con la diferencia que no se pone el “;” final. Si el método posee varios argumentos, se deben separar unos de otros con el operador de continuación de declaración (“,”). Si el método no recibe argumentos, se dejan sólo los paréntesis del mismo.

Comienzo del bloque de sentencias: Obligatoriamente poner la llave de apertura de bloque de sentencias.

Sentencias: Opcionalmente agregar declaraciones e invocaciones a métodos. Existe la posibilidad de no poner nada a lo que se llama método de cuerpo vacío.

Valor retornado: opcionalmente poner una sentencia return para terminar la ejecución del método. Si se indico que el método retorna un valor (recordar que puede ser void), este se debe poner a continuación.

Fin del bloque de sentencias: Poner obligatoriamente la llave de cierre.

Ejemplo

```
public int calculaVolumen(int ancho, int largo, int alto) {
      // bloque de sentencias
      return 0;
}
```

Este método es público, retorna un entero y recibe tres enteros como parámetros

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…
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.
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.

Cuando se declaren constructores para las clases, se pueden utilizar los modificadores de acceso que determinan si otros objetos pueden crear los 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. A esto se lo denomina sobrecarga de constructores (la sobrecargas se explicará a continuación).

```
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 cuál de ellos utilizar.

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.

```
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.

```
public class Ejemplo {
	private int atrib1;
	private char atrib2;
	Ejemplo (char atrib2,int atrib1){
		this.atrib1 = atrib1;
		this.atrib2 = atrib2;
	}
}
```

Varargs o Argumentos variables

El uso de **Java varargs** es poco conocido dentro del mundo Java y a veces nos pueden ser realmente útiles, fue introducido en Java 5. Son métodos que permiten variar el número de parámetros que reciben permitiendo trabajar de una forma más cómoda.

Ejemplo de uso:

```
//Primero lo primero, declaramos las variables que se usarán en el
//ejemplo.
int[] num={4,7,8,2};
int cuatro=4,dos=2;

//Y los métodos también. Nótese la diferencia entre ambos.

//Este método usa varargs
int suma_a (int... numero){
	int resultado = 0;
	for(int i = 0; i < numero.length; i++){
		resultado += numero[i];
	}
	return resultado;
}

//Este método no usa varargs
int suma_b (int[] numero){
	int resultado = 0;
	for(int i = 0; i < numero.length; i++){
		resultado += numero[i];
	}
	return resultado;
}



//Llamadas válidas:

System.out.println(suma_a(num));
System.out.println(suma_a(num[0],num[1],num[2],num[3]));
System.out.println(suma_a(cuatro,7,8,dos));

System.out.println(suma_b(num));
//Llamadas NO válidas:
System.out.println(suma_b(num[0],num[1],num[2],num[3]));
System.out.println(suma_b(cuatro,7,8,dos));
```

```
//Declaraciones válidas:

int suma_a(String cadena, int... numero){...}
int suma_b(String cadena, int[] numero){...}
int suma_b(int[] numero, String cadena){...}
//Declaración NO válida:
int suma_a(int... numero, String cadena){...}
```

Reglas de uso:

1 – Los métodos solo pueden tener un solo parámetro varargs.

2 – El parametro del typo varargs debe ser el ultimo parámetro del método

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.

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 .

Ocultamiento de la información

Para proteger los datos que se almacenan dentro de un objeto, la clase utiliza la declaración de visibilidad private para que no se pueda acceder la variable, ya que esta declaración indica que la variable no pertenece a la interfaz de la clase. Cuando un elemento no pertenece a la interfaz no se puede acceder por notación de punto una vez creado un objeto, sólo lo podrán acceder aquellos elementos que pertenezcan a la clase.

Encapsulado

Como se mostró con anterioridad, en toda clase existe una parte pública y una privada. La primera define los elementos de la clase que son accesibles a través de su interfaz. Muchas veces se hace referencia a este hecho como “accesible por el mundo exterior o el universo”.
Se puede decir que el ocultamiento de la información y los métodos privados son en conjunto conocidos como “encapsulado”, estén presente uno de ellos o ambos.
Un ejemplo claro de esto es cuando en una clase se oculta la información pero se quiere brindar la posibilidad de acceder a los datos almacenados, ya sea para guardar valores como para leerlos. En este caso se deben crear métodos públicos que cumplan ese rol, como se mostró en el ejemplo anterior.

Ejemplo de ocultamiento y getters && setters:

```
public class Persona {
      private String primerNombre;
      private String segundoNombre;
      private String apellido;
      private String documento;

      public Persona() { } //Constructor vacio.
      public String getApellido()                    { return apellido;      }
      public String getDocumento()                   { return documento;     }
      public String getPrimerNombre()                { return primerNombre;  }
      public String getSegundoNombre()               { return segundoNombre; }
      public void setApellido(String string)         { apellido = string;    }
      public void setDocumento(String string)        { documento = string;   }
      public void setPrimerNombre(String string)     { primerNombre = string;}
      public void setSegundoNombre(String string)    { segundoNombre = string}
}
```

Paquetes java

Los paquetes son grupos relacionados de clases, interfaces y enumeraciones que proporcionan un mecanismo conveniente para manejar un gran juego de clases e interfaces y evitar los conflictos de nombres (porque los paquetes en sí 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>+
```

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 las 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).

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.

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 utilidades.*;
```

Cuidado importar todas las clases de un paquete y no usarlas, implica usar memoria RAM innecesaria.
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.

Variable de entorno 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.

Relaciones entre clases

Las relaciones entre clases pueden ser las siguientes:

- Asociaciones simples
- Agregaciones
- Composiciones
- Herencia

Las relaciones entre clases también son llamadas comunicación entre clases o colaboración de clases.

Acoplamiento y cohesión entre clases

En palabras simples, el acoplamiento se refiere al grado de dependencia que tiene una clase con otra y la cohesión indica cuanto dos clases deben mantenerse juntas, en ambos casos, para definir su funcionamiento (servicios y atributos).

Por ejemplo, si la creación de un objeto depende de otro que deba existir como una parte integral de él, se debe crear primero la relación con el objeto del tipo de la otra clase antes que finalice la construcción del actual, lo cual implica un alto grado de acoplamiento y un bajo grado de cohesión. Mientras que si la creación de ambos objetos es independiente pero necesitan estar ambos para cumplir con la funcionalidad requerida, indica un acoplamiento menor, pero un mayor grado de cohesión.

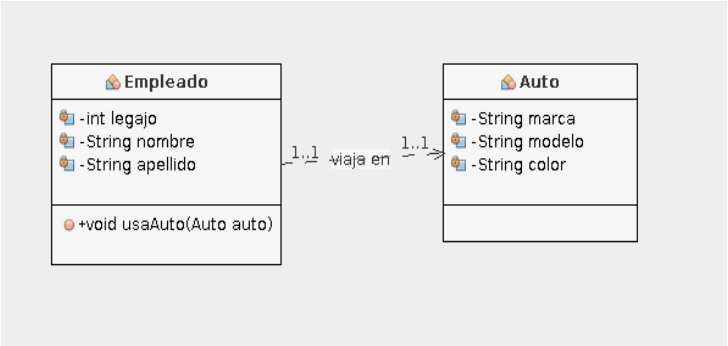
Consideraciones de diseño: es una buena práctica que el software sea altamente cohesivo y poco acoplado sin forzar este hecho en el diseño.

Asociaciones simples

Dentro de las posibles asociaciones existen las más débiles (o mejor expresado aún, las menos acopladas y más cohesivas) llamadas asociaciones simples. Para detectar una asociación simple entre dos clases u objetos se utilizan las palabras “usa un” o “usa una”, a las que se denominan frases de validación. Por ejemplo, “una persona usa un auto”.

Es claro que la relación entre la persona y el auto es temporal e independiente de la vida de los objetos involucrados. Por ejemplo, el hecho que la “persona use el auto” no implica que la persona cree el auto para usarlo.

La asociación es inherentemente bidireccional, lo cual significa que se la puede crear de cualquiera de las clases que intervienen para definir un recorrido “lógico” en ambas direcciones.



No Existe en la clase Empleado ningún atributo que vincule a la otra clase, solo un método que indica que auto usa y la relación dura solo el tiempo en que el método se ejecuta.
Un objeto Auto puede ser usado por otra Persona.

```
public class Auto {
    private String marca;
    private String modelo;
    private String color;
    public Auto(String marca, String modelo, String color) {
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
    }
    @Override public String toString() {
        return "Auto{" + "marca=" + marca + ", modelo=" + modelo + ", color=" +
color + '}';
    }
}

public class Empleado {
    private int legajo;
    private String nombre;
    private String apellido;
    public Empleado(int legajo, String nombre, String apellido) {
        this.legajo = legajo;
        this.nombre = nombre;
        this.apellido = apellido;
    }
```

```
    @Override public String toString() {
        return "Empleado{" + "legajo=" + legajo + ", nombre=" + nombre + ",
apellido=" + apellido + '}';
    }
    public void usaAuto(Auto auto){
        System.out.println("Se esta usando el auto: "+auto);
    }
}
```

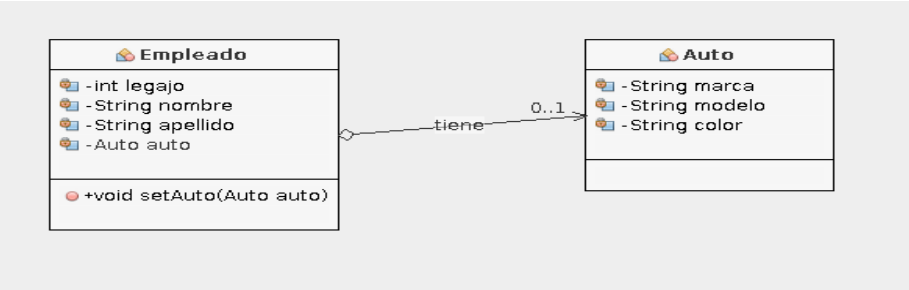
Agregaciones

Es una relación más fuerte entre clases. Por lo general, cuando se enuncia el problema de diseño se puede reconocer fácilmente una agregación porque se caracteriza por las palabras “tiene un” o “tiene una”.
Otra forma de reconocer las asociaciones es la relación “totalidad – parte de”. Por ejemplo, se puede definir como problema de diseño que:

- Un auto tiene una radio
- Un auto puede tener accesorios

El auto es la totalidad y la radio o los accesorios son una parte que se reconoce como un objeto independiente. Sin embargo, es claro que aunque la relación puede interpretarse como “totalidad – parte de”, esto no implica que ambos objetos mantengan su relación durante todo el tiempo que cada uno este activo (por ejemplo, un auto puede cambiar en algún momento la radio que tiene, además de incorporar o quitar accesorios), aunque también, la relación puede durar la vida del objeto (por ejemplo, un auto nunca cambia la radio que tiene, aunque pueda).

Consideraciones de diseño: las agregaciones son las relaciones más frecuentes en los sistemas y son, por lo tanto, las más utilizadas. Otra característica importante es que ofrecen un buen balance entre la cohesión y el acoplamiento ya que la dependencia entre clases es de una fuerza media. La asociación a lo largo del tiempo puede variar pero es más estable que la simple que suele tener una duración corta entre las clases.



Existe un atributo en la clase Empleado que vincula a la clase Auto, no se requiere dicho atributo en el constructor. Por que la existencia de un objeto una clase, no implica la existencia del objeto de la otra clase. El objeto Empleado puede cambiar de auto y el objeto Auto puede pertenecer a otra persona en forma simultanea.

```
public class Auto {
    private String marca;
    private String modelo;
    private String color;
    public Auto(String marca, String modelo, String color) {
```

```
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
        this.auto = null;
    }
    @Override public String toString() {
        return "Auto{" + "marca=" + marca + ", modelo=" + modelo + ", color=" +
color + '}';
    }
}

public class Empleado {
    private int legajo;
    private String nombre;
    private String apellido;
    private Auto auto;
    public Empleado(int legajo, String nombre, String apellido) {
        this.legajo = legajo;
        this.nombre = nombre;
        this.apellido = apellido;
        this.auto = null;
    }
    @Override public String toString() {
        return "Empleado{" + "legajo=" + legajo + ", nombre=" + nombre + ",
apellido=" + apellido + ", auto=" + auto + '}';
    }
    public void setAuto(Auto auto) {
        this.auto = auto;
    }
}
```

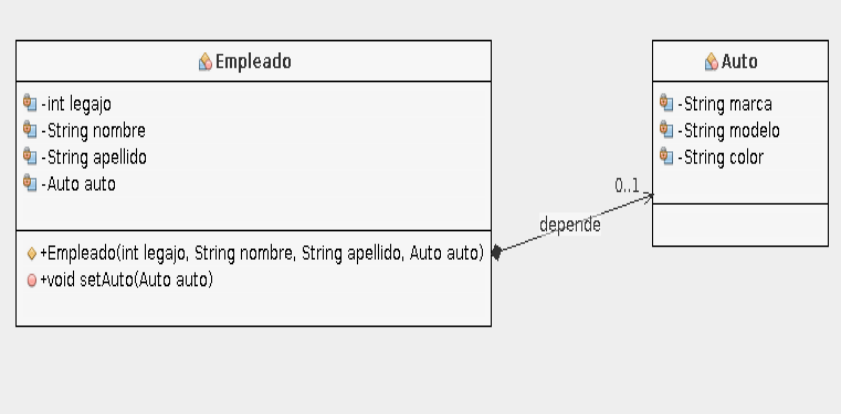
Composiciones

Es la relación más fuerte entre clases, ya que cuando una clase compone un objeto del tipo de otra, el problema de diseño indica que para resolverlo, una clase no tiene sentido sin la instancia del objeto de la clase que compone.
La frase de validación se caracteriza por las palabras “siempre tiene”. Por ejemplo, un problema de diseño donde se puede apreciar lo afirmado es el siguiente:

1. Un auto siempre tiene un motor
2. Una factura de siempre tiene detalles de facturación

Las composiciones tiene como característica que duran el mismo tiempo que la vida del objeto siempre, ya que si faltase el objeto compuesto, el que lo compone no tiene sentido. Por ejemplo, no tiene sentido tratar de diseñar un auto sin su motor o una factura sin detalles de facturación (en este caso la composición también incluye una multiplicidad en el rol. Atención, cuando se realiza una factura los detalles están bien determinados, se agregan para confeccionarla pero cuando se emite está determinado su detalle).

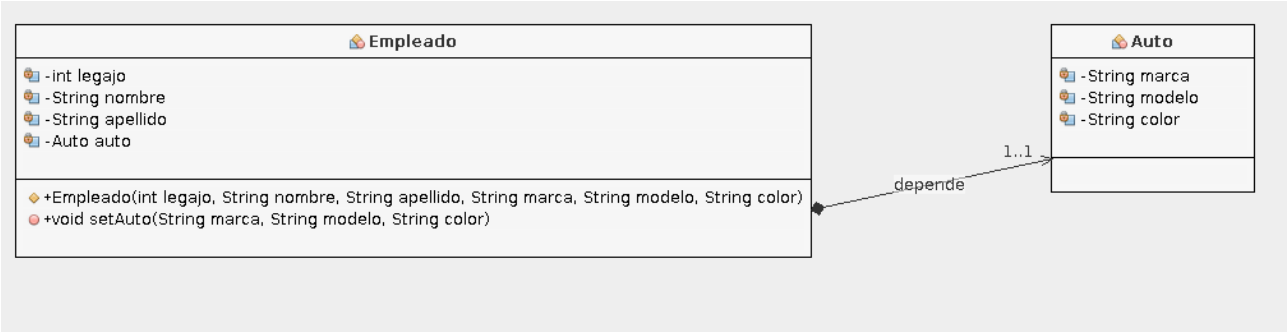
Consideraciones de diseño: las composiciones son relaciones menos frecuentes que las agregaciones en los sistemas. Son las de mayor acoplamiento ya que la dependencia entre clases es de una gran fuerza y la asociación a lo largo del tiempo esta enlazada entre las clases.
En otras palabras, un objeto de un tipo compuesto con otro no puede existir si el que compone no se crea. Por lo tanto, la cohesión en este caso es prácticamente nula (no se suelen usar mucho las interfaces para acceder a los servicios del objeto compuesto).



En el ejemplo de la imagen: Un Empleado si o si tiene un Auto, y depende de ese auto, y lo puede cambiar. Un objeto de auto puede pertenecer a ningún uno o varios Empleados o Personas.

```
public class Auto {
    private String marca;
    private String modelo;
    private String color;
    public Auto(String marca, String modelo, String color) {
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
    }
    @Override public String toString() {
        return "Auto{" + "marca=" + marca + ", modelo=" + modelo + ", color=" +
color + '}';
    }
}

public class Empleado {
    private int legajo;
    private String nombre;
    private String apellido;
    private Auto auto;
    public Empleado(int legajo, String nombre, String apellido, Auto auto) {
        this.legajo = legajo;
        this.nombre = nombre;
        this.apellido = apellido;
        this.auto = auto;
    }
    public void setAuto(Auto auto) {
        this.auto = auto;
    }
    @Override public String toString() {
        return "Empleado{" + "legajo=" + legajo + ", nombre=" + nombre + ",
apellido=" + apellido + ", auto=" + auto + '}';
    }
}
```



En este ejemplo un Auto solo pertenece a un Empleado y no puede pertenecer a otra persona, dado que es creado dentro de la clase Empleado y contenido dentro de ella. El Empleado puede cambiar de Auto pero al hacerlo destruye el auto anterior.

```
public class Auto {
    private String marca;
    private String modelo;
    private String color;
    public Auto(String marca, String modelo, String color) {
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
    }
    @Override public String toString() {
        return "Auto{" + "marca=" + marca + ", modelo=" + modelo + ", color=" +
color + '}';
    }
}

public class Empleado {
    private int legajo;
    private String nombre;
    private String apellido;
    private Auto auto;
    public Empleado(int legajo, String nombre, String apellido, String marca,
String modelo, String color) {
        this.legajo = legajo;
        this.nombre = nombre;
        this.apellido = apellido;
        this.auto = new Auto(marca, modelo, color);
    }
    public void setAuto(String marca, String modelo, String color) {
        this.auto = new Auto(marca, modelo, color);
    }
    @Override public String toString() {
        return "Empleado{" + "legajo=" + legajo + ", nombre=" + nombre + ",
apellido=" + apellido + ", auto=" + auto + '}';
    }
}
```

Nota: Para lograr una composición real, deberíamos declarar la clase Auto dentro de Empleado de esta manera un objeto auto no puede existir por fuera de la clase. Este concepto en la actualidad resulta poco practico.

```
public class Empleado {
    private int legajo;
    private String nombre;
    private String apellido;
    private Auto auto;
    public Empleado(int legajo, String nombre, String apellido, String marca,
String modelo, String color) {
```

```
        this.legajo = legajo;
        this.nombre = nombre;
        this.apellido = apellido;
        this.auto = new Auto(marca, modelo, color);
    }
    public void setAuto(String marca, String modelo, String color) {
        this.auto = new Auto(marca, modelo, color);
    }
    @Override public String toString() {
        return "Empleado{" + "legajo=" + legajo + ", nombre=" + nombre + ",
apellido=" + apellido + ", auto=" + auto + '}';
    }
    private class Auto {
        //La clase auto es interna de Empleado, es un miembro más
        //Al ser privada la clase no puede ser instanciada desde fuera de la
clase
        //El acoplamiento entre clases es muy fuerte.
        private String marca;
        private String modelo;
        private String color;
        public Auto(String marca, String modelo, String color) {
            this.marca = marca;
            this.modelo = modelo;
            this.color = color;
        }
        @Override public String toString() {
            return "Auto{" + "marca=" + marca + ", modelo=" + modelo + ",
color=" + color + '}';
        }
    }//end class Auto
}//end class Persona
```

Cardinalidades

En Relaciones simples, agregación y composición, se representaron con la cardinalidad ‘1 a 1’ o ‘1 a 0’ existen cardinalidades multiples ‘1 a *’ o ‘0 a *’ (el * se lee como muchos).
Las cardinalidades múltiples serán mostradas mas adelante en forma adecuada con el uso de colecciones.

Relación de Herencia

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:

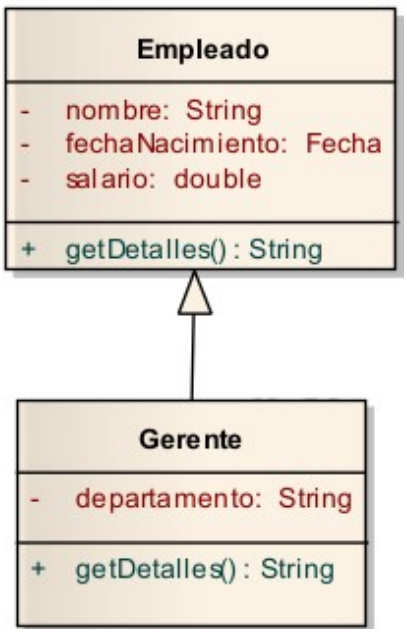
```
class SubClase extends SuperClase {
```

```
        . . .
}
```

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.
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{
        . . .
}
```

Ejemplo de Herencia en UML



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

¿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).

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 .

Super

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

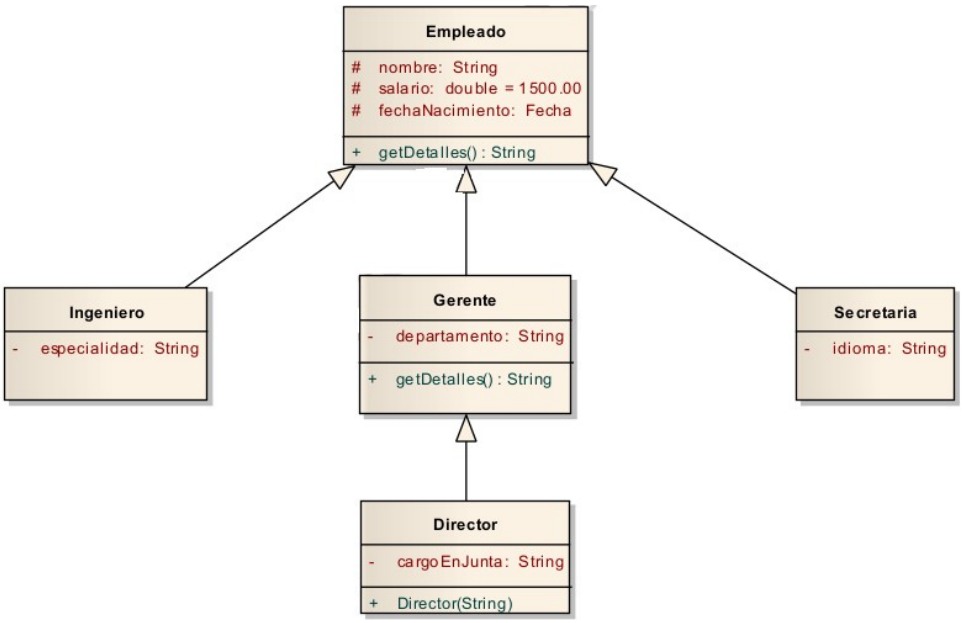
¿Qué métodos hereda una subclase?

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).

Entonces, 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 subclases 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.




```
public class Empleado {
	protected String nombre;
	protected double salario = 1500.00;
	protected Fecha fechaNacimiento;
	public String getDetalles() {
		return "Nombre: " + nombre + "\nSalario: " + salario;
	}
}
```

```
class Gerente extends Empleado {
	private String departamento;
	public String getDetalles() {
		return "detalles";
	}
}

public class Director extends Gerente {
	private String cargoEnJunta;
	public Director(String c) {
		cargoEnJunta = c;
	}
}

public class Secretaria extends Empleado {
	private String idioma;
}

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

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.

Reemplazar 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.

```
	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 cómo construir los objetos de su tipo, pero puede proporcionar inicialización adicional específica de la subclase.

- Para ampliar el método constructor se utiliza super();
- Para llamar a un método declarado en la superclase se usa super.método(), ej
  super.toString();

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.

Métodos que una subclase debe sobrescribir o rescribir

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, completarlo porque está incompleto y ocultarlo para que se puedan declarar objetos de la subclase que lo rescribe.

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

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.

```
public class OtraClase extends MiClase {
      boolean unaVariable;
      void unMetodo() {
            unaVariable = false;
            super.unMetodo();
            System. out .println(unaVariable);
            System. out .println(super.unaVariable);
      }
}
```

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.

```
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);
      }
}

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).

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.

```
if (empleado instanceof Empleado)
```

Método toString()

El método **toString()** nos permite mostrar la información completa de un objeto, es decir, el valor de sus **atributos.**
Este método también se hereda de **java.lang.Object**, por lo que deberemos sobrescribir este **método**.
Esta es su definición:

```
public String toString (){
        String mensaje="El empleado se llama "+nombre+" "+apellido+" con "+edad+"
            años " + "y un salario de "+salario;
        return mensaje;
    }
```

Este método tiene la característica que se puede invocar nombrando solamente el objeto ej:

```
System.out.println(persona.toString());
System.out.pritnln(persona);              //se invoca toString()
```

¿Qué es una simulación (Mock)?

Las simulaciones se utilizan para efectuar operaciones que imitan el comportamiento de un sistema en ejecución. La idea en general es evitar cualquier dependencia externa que pueda tener el sistema, incluyendo los ingresos de datos por parte de un usuario, que pueda tener mediante datos fijos pre establecidos para emular el funcionamiento de las distintas partes que lo componen. De esta manera, se logra una primera evaluación de la correcta funcionalidad de un sistema.
Existen muchas maneras de realizar simulaciones y muchas soluciones de software que se pueden incorporar a un desarrollo en forma de API (Application Program Interface) para llevarlas a cabo. Sin embargo, se puede lograr el mismo fin con clases sencillas diseñadas por el usuario y esta es la solución que se utiliza en este desarrollo. El motivo es incorporar la menor cantidad de conceptos nuevos fuera de los que se centran en las herramientas de la programación orientada a objetos pero sin dejar de conocer la forma moderna de trabajar.
Ejemplo de clase Mock

```
public class EmpleadoApp {

  public static void main(String[] args) {

    //Creamos dos objetos distintos
    Empleado empleado1=new Empleado("Fernando", "Ureña", 23, 600);
    Empleado empleado2=new Empleado("Antonio", "Lopez", 28, 900);
    Empleado empleado3=new Empleado("Alvaro", "Perez", 19, 800);

    //Mostramos la informacion del objeto
    System.out.println(empleado1.toString());
    System.out.println(empleado2.toString());
    System.out.println(empleado3.toString());
```

```
    }
}
```