Programación Orientada a Objetos.

Resumen de conceptos

Índice

[Herencia: 3](#_Toc515464445)

[This y super: 3](#_Toc515464446)

[Clases y métodos abstractos 4](#_Toc515464447)

[Clases y métodos finales 4](#_Toc515464448)

[Métodos de clase: 5](#_Toc515464449)

[Constantes: 5](#_Toc515464450)

[Re definición de métodos: toString, equals 5](#_Toc515464451)

[Casteo 7](#_Toc515464452)

[Polimorfismo 7](#_Toc515464453)

[Sobre carga de métodos y constructores 8](#_Toc515464454)

[Especificaciones de acceso 8](#_Toc515464455)

[Try / Catch 9](#_Toc515464456)

[Relaciones entre objetos 11](#_Toc515464457)

[Interfaces 11](#_Toc515464458)

# Herencia:

El término herencia se refiere al hecho de que una clase hereda los atributos (variables) y el comportamiento (métodos) de otra clase.

En Java se utiliza la palabra clave “extends” para indicar que una clase hereda de otra.

Ejemplo:

**public class Automovil extends Vehiculo {**

**}**

Decimos que Automovil “hereda de” Vehiculo

O que Automovil “es un/una” Vehiculo

Vehiculo: Súper clase, clase padre, clase base

Automovil: Sub clase, clase hija, clase derivada

En Java todas las clases heredan de la clase base Object.

Ejemplo:

**public class MiClase extends Object {**

**}**

Se da implícitamente y no es necesario agregar el “extends Object”

# This y super:

Si quiero hacer referencia a algún atributo o método propio del objeto lo hago utilizando this

* this.propiedad = valor
* this.metodo()
* this(parámetro) // un constructor

Si quiero hacer referencia a algún atributo o método propio de la clase padre lo hago utilizando súper.

* super.setPropiedad(valor)
* super.metodo()
* super(parámetros) // un constructor de la clase padre

# Clases y métodos abstractos

Es posible definir clases que representan un concepto abstracto y como tal no pueden ser instanciadas.

Por ejemplo: en nuestro sistema tenemos Profesores y Alumnos, identificamos una relación de herencia a través de una clase padre Persona, la cual deberíamos evitar ser instanciada porque no sería válido en nuestro dominio.

Ejemplo:

**public abstract class Persona {**

**}**

Este código no será válido:

**Persona juan = new Persona()**

Por el contrario, si la clase no es abstracta decimos que es concreta, es decir se pueden crear instancias

Si la clase es abstracta también tenemos la posibilidad de definir métodos abstractos: Es un método que NO tiene implementación o código

Ejemplo:

**public abstract void prepararClase();**

No lleva {}, se cierra con punto y coma la sentencia luego de los paréntesis.

# Clases y métodos finales

Una clase finales una clase que no puede ser extendida, es inmutable, no cambia. No tiene capacidad de cambio. Es decir nadie puede heredar de esa clase.

Si es posible crear instancias de una clase declarada final.

Por alguna razón, el diseño de la clase nunca necesitará cambios o por razones de seguridad no queremos que la clase sea extendida.

Ejemplo:

**public final class Telefono {**

**}**

Todos los métodos en una clase final son implícitamente final, ya que no es posible sobre escribirlos.

En una clase no final, podemos declarar un método como final es decir si alguna otra clase hereda de esa clase, no podrá sobre escribirse ese método.

**public final void metodo(Object object) {**

**}**

Se llama sobre escritura / re definición de métodos cuando una clase hija sobre escribe un método heredado de su clase padre. Para eso usamos la anotación @Override.

@Override

**public final boolean equals(Object object) {**

**}**

# Métodos de clase:

Un método de clase solo tiene acceso a sus variables locales, parámetros y variables de clase. NO puede acceder a variables ni métodos de instancia. Un método de instancia si puede acceder a métodos y variables de clase.

En los métodos de clase NO está disponible el objeto this, pues no tienen ningún objeto asociado, y tampoco están disponibles las variables de instancia.

Usamos métodos de clase en el caso que no tengamos que operar sobre variables de instancia de objetos. Los métodos que no son de clases se llaman métodos de instancia.

Ejemplo:

**public static metodo() {**

**}**

# Constantes:

Son variables con valores que queremos evitar que sean modificados.

Ejemplos:

private int IVA = 21;

private float PI = 3.14f;

# Re definición de métodos: toString, equals

Los métodos toString y equals están definidos en la clase base Object y pueden ser re definidos en cualquier clase:

El método**toString** nos permite mostrar información completa de objeto, es decir, el valor de sus **atributos.**

El método produce una representación textual y legible del contenido de un objeto

@Override

**public String toString() {**

**return this.getPropiedad()**

**}**

El método **equals** es un método incorporado en la clase Object (padre de todas las clases en java) y que por tanto toda clase en java va a heredar. De forma predeterminada este método comparara los objetos por sus ubicaciones en memoria y no por su contenido. Para variar este comportamiento deberemos de sobrescribir este método.

@Override

public final boolean equals(Object object) {

if ((object == null) || !(object instanceof Persona)) {

return false;

}

Persona otraPersona = (Persona) object;

return this.getNumeroDocumento() == otraPersona.getNumeroDocumento();

}

Paso a explicar línea por línea del método: En este caso decimos que dos personas son la misma si tienen el mismo número de documento.

El método recibe un parámetro de tipo Object, por lo tanto lo que primero evaluamos es que si el parámetro es nulo o bien no es del tipo Persona (no es una instancia de la clase Persona, a través de la sentencia instanceOf) devolvemos falso.

Si el parámetro no es nulo y es del tipo Persona lo convertimos o lo “casteamos” a Persona anteponiéndole el nombre de la clase persona entre paréntesis.

Luego nuestra condición de igualdad se reduce a comparar los números de documentos de ambos objetos, tanto de la instancia (this) como del parámetro recibido.

La versión original del método Compara referencias, es decir devuelve true si la referencia del objeto recibido como argumento es igual a la del objeto receptor del mensaje (o sea, las referencias apuntan a la misma instancia de objeto).

NO compara el contenido de los objetos. Es lo mismo que usar ==.

Cuando redefinimos el método, comparamos contenido.

# Casteo

El casting significa convertir tipos de datos. Es posible aplicar casting a variables de tipo primitivo para obtener un valor convertido a otro tipo primitivo (por ej. de un número decimal a un entero) o para convertir objetos de una clase a otra clase.

El operador de casting tiene un solo operando a la derecha que es la expresión que se desea convertir y se representa encerrando entre paréntesis un tipo de datos. Ejemplo: (int), (float), (String).

# Polimorfismo

El término "Polimorfismo" es una palabra de origen griego que significa "muchas formas". Este término se utiliza en la POO para "referirse a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos".

[Múltiples métodos con igual nombre y diferente comportamiento. Facilita la extensibilidad.](https://jarroba.com/wp-content/uploads/2014/04/PolimorfismoFutbol-diag.jpg)

Ejemplo:

**public class Telefono {**

**public void llamar() {**

**}**

**}**

**public class TelefonoFijo extends Telefono {**

**@Override**

**public void llamar() {**

**}**

**}**

**public class TelefonoCelular extends Telefono {**

**@Override**

**public void llamar() {**

**}**

**}**

El polimorfismo se da en que tanto las instancias de Telefono, TelefonoFijo y TelefonoCelular entienden el mismo mensaje llamar(), cada uno con su propia implementación.

# Sobre carga de métodos y constructores

La firma de un método es la combinación del nombre y los tipos de los parámetros o argumentos.   
  
La sobrecarga de métodos es la creación de varios métodos con el mismo nombre pero con diferente lista de tipos de parámetros. Java utiliza el número y tipo de parámetros para seleccionar cuál definición de método ejecutar.   
  
Java diferencia los métodos sobrecargados con base en el número y tipo de parámetros o argumentos que tiene el método y no por el tipo que devuelve.   
  
También existe la sobrecarga de constructores: Cuando en una clase existen constructores múltiples, se dice que hay sobrecarga de constructores.

Ejemplo:

**int sumar(int a, int b) {**

**}**

**int sumar(int a, int b, int c) {**

**}**

**int sumar(float a, float b) {**

**}**

Están sobrecargados porque tienen el mismo nombre, devuelven el mismo tipo de datos, pero varían en los parámetros.

# Especificaciones de acceso

Evitar que los desarrolladores cometan errores llamando métodos que no deben.

En caso de que generemos un error involuntario, podemos reducir el lugar de búsqueda de los errores.

Regular el acceso desde ciertas partes de nuestro proyecto (clases en una herencia, clases del mismo paquete, etc.).

Por ejemplo, queremos limitar el acceso a un método utilitario que modifica atributos internos.

Son public, private y protected. Aplican a variables, métodos y clases.

Public es el más permisivo: cualquier variable, método o clase con el acceso public puede ser accedida desde cualquier clase.

Recordemos que por convención nunca una propiedad es public, sino que la declaramos private y accedemos a ella a través de su getter y setter.

Protected es similar a public, pero permite ocultar un método o atributo hacia el exterior de un paquete.

El objetivo es permitir que un “hijo” de nuestra clase pueda llamar o pisar métodos para extender cierta funcionalidad.

Una clase no puede ser protected, si queremos que solamente sea utilizada dentro de su paquete, en su declaración omitimos la especificación de acceso.

Ejemplo:

class Auto {

}

Private solamente puede ser accedida dentro de la misma clase.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Clase | Método | Propiedad |
| public | La clase puede ser instanciada y referenciada en cualquier lugar del programa | El método puede ser invocado en cualquier parte de nuestro programa. | La propiedad puede ser utilizada en cualquier parte de nuestro programa, pero por convención las declaramos siempre private o protected y creamos getters y setters para dar visibilidad. |
| protected | La clase puede ser instanciada y referenciada solamente dentro de ese paquete (no se pone la palabra protected, no va modificador de acceso) | El método puede ser invocado solamente por alguna clase hija o derivada. | La propiedad puede ser usada solamente dentro de alguna clase hija. |
| private | La clase puede ser instanciada y referenciada solamente dentro de la clase dentro de la cual fue declarada. Es decir una clase privada se declara dentro de una clase. | El método solo puede ser invocado dentro de la clase que lo declara. | La propiedad puede ser usada solamente dentro de la clase que la declara. |

# Try / Catch

Una excepción es un error que se produjo durante la ejecución de un programa. Para poder trabajar con excepciones y desarrollar un programa tolerante a errores se utiliza el bloque try-catch-finally.

Podemos capturar el error dentro de la sección catch para ya sea informar sobre el error al usuario de manera elegante o también para loguearlo en algún archivo del tipo Log para luego ver esa información y entender por qué ocurrió la excepción para arreglarla.

int resultado = 0;

try {

/\*

El código dentro de try está vigilado: si se produce un error y se lanza una excepción,

el control pasa al bloque catch. Se pueden incluir tantos catchs como se deseen.

Luego si está presente se ejecuta el bloque finally

\*/

resultado = dividendo / divisor;

}

catch(ArithmeticException arithmeticException) {

/\*

Acabo de capturar una Excepción del tipo Aritmética

\*/}

catch(Exception exception) {

/\*

Acabo de capturar una Excepción genérica. Puedo poner más de un catch pero el primero tiene

que capturar la excepción más específica y el segundo más genérica.

\*/

}

finally {

/\*

Este bloque es opcional.

Este código se va a ejecutar siempre: ya sea porque lo que está dentro de try se ejecutó sin problemas o bien porque ocurrió un error y se capturo el error en el catch

\*/

}

# Relaciones entre objetos

Un conjunto de objetos aislados tiene una escasa capacidad para resolver un problema.

Los objetos colaboran e intercambian información, mantienen distintos tipos de relaciones entre ellos.

Vida real de los objetos:

* Unos están compuestos de otros (Factura compuesto de ItemsFacturas)
* Unos necesitan la colaboración de otros para llevar a cabo ciertas tareas (Usuario y Factura)

Tipos de relaciones

* Agregación/Composición: Al crear A se crea B. Relación de A “el todo” y B “las partes”. Al inicio “todo” construye cada “parte”. Al final “todo” destruye cada “parte”. Relación entre Factura e ItemFactura.
* Asociación: Un atributo de A es una referencia a un objeto B. La creación de B no implica la creación de A. Por ejemplo ItemFacturas y Productos. Si bien cada ItemFactura requiere de un Producto, un Producto no requiere de ItemFacturas. Si la cardinalidad es mayor que uno se usan ArrayLists. Durable.
* Uso: A necesita de B para hacer una funcionalidad. Por ejemplo: en un método de B se crea una instancia de A o en un método de B aparece un objeto de A como argumento. Temporal.
* Herencia: una clase B hereda de A

# Interfaces

En las clases abstractas se pueden definir métodos y comportamiento. La herencia representa el tipo de relaciones “es un” las interfaces nos permiten representar relaciones “se comporta como un”.

Las interfaces definen métodos que luego serán implementados en una o más clases. Una clase puede implementar una o más interfaces.

Se usa la palabra implements.

Ejemplo:

**public interface Trabajador {**

**void trabajar();**

**}**

**public class Persona implements Trabajador {**

**@Override**

**public void trabajar() {**

**}**

**}**