### Formas de programar (paradigmas)

Existen dos tipos de programación, los antiguos y los modernos:

Los antiguos son los que están orientados a procedimientos, estos surgieron en la década de los 60-70

Los modernos están orientados a programación de objetos

¿Porque surgió la programación moderna(POO)? En aquel entonces la programación antigua (procedimiento) era un montón de código, por ende, era demasiado difícil determinar, cuando ocurría un error, también bine agregado lo de porder reutilizar el código en otro programa parecido. Es por ello que se decide reinventar la programación de procedimiento y sale como resultado la programación orientada a objeto.

El nombre de programación orientada a objeto viene dado, porque se intentó dar al código de programación el mismo comportamiento que tienen los objetos de la vida real.

### Ejemplos de programación orientadas a procedimientos:

* Fortran
* Cobol
* Basic (no los modernos de ahora as visual Basic)

Programación orientada a objetos

### ¿En qué consiste?

* Trasladar la naturaleza de los objetos de la vida real al código de programación.

¿Cuál es la naturaleza de un objeto de la vida real?

* Los objetos tienen un estado, un comportamiento (¿Qué pueden hacer?), unas propiedades.

### Ejemplo: El objeto coche.

* ¿Cuál es el estado de un coche? Un coche puede estar parado, circulando, aparcado etc.
* ¿Qué propiedades tiene un coche? Un coche tiene un color, un peso, un tamaño etc.\
* ¿Qué comportamiento tiene un coche? Un coche puede arrancar, frenar, acelerar, girar etc.

### Algunos ejemplos de programación orientada a objeto:

C++, Java, Visula.net etc.

### Ventajas

* Programas divididos en “trozo”, “Partes”, “módulos”, “clases”. Modularizacion.
* Muy utilizables. Herencia
* Si existe fallo en alguna línea del código, el programa seguirá con su funcionamiento. Tratamiento de exenciones
* Encapsulamiento

### Clase

Una clase es un modelo donde se redactan las características comunes de un grupo de objetos.

Ejemplo

Un coche tiene una plataforma, esta plataforma puede ser la misma para diferentes modelos, la plataforma es la clase

Los objetos son como propiedades de dicha clase

Una clase son características comunes de un grupo de objetos, y luego estarían los objetos que son los que pertenecen a esa clase

Una clase se define como un tipo abstraído de dato que contiene atributos y propiedades. A través de la clase se implementa un concepto abstraído de la realidad.

Un objeto es un ejemplar perteneciente a una clase

### Método constructor

Es un método especial que se encarga de dar un estado inicial a nuestro objeto.

Un método constructor se construye poniendo la palabra reservada public, a continuación, el nombre de dicho método.

Ejemplo:

Public plataforma {}

Característica principal de un método constructor es que debe de tener siempre el mismo nombre de la clase.

### Métodos

Los métodos nos permiten que nuestras clases interactúen entre sí, formando una unidad para que un programa funcionen.

Tipos de métodos que nos permitirán hacer uso de la clase que posee objetos con la nomenclatura **private** llamados métodos de acceso:

### SETTER:

Este está encargado de definir o establecer el valor de una propiedad.

### GETTER

Este está destinado a captar o proporcionar el valor de dicha propiedad.

En conclusión, por un lado, con el método **setter** modificamos el valor de una propiedad y, por otro lado, con el método **getter** obtenemos cual es el valor de esa propiedad.

### Getter

* Función: devolver el valor de la propiedad del objeto
* Sintaxis:

Public [datos a devolver int, strign,doublé etc] [nombre del método](){código + return}

Public string altura (){return “HOLA ”}

### SETTER

* Función: modifica el valor de las propiedades de los objetos
* Sintaxis: public void [nombre del metodo] (){codigo}
* ¿Que indica void? Indica que el método no devuelve ningún valor.

### Paso de parámetros

Es cuando le pasamos un valor a un método sin hacerlo directamente en dicho método

Concepto de ámbito de una variable

Una variable puede ser:

Local

Global

### Otra forma de programar

Cuando se utiliza un único fichero, el cual albergar varias clases, solamente una de esas clases puede ser pública debe de tener el modificador de acceso “Public”

### Palabra reservada final

Final = constant.

En una aplicación posiblemente nos encontremos con algún valor que permanece constante durante la ejecución. Podemos definirla como una variable común, pero perderíamos el control. Por allí, en algún descuido, se cambiaría de valor, pero no nos enteraríamos. Podemos agregar a la definición de la variable el modificador **final**, que indica que a esa variable solo se le puede asignar un valor u objeto una única vez. La sintaxis es la siguiente:

**final** tipo\_variable nombre\_de\_variable = valor;

Por ejemplo :

**final** **int** unaConstante = 10;

### característica de un método **static**

* No actúan sobre un objeto
* No acceden a campo ejemplar (variables / constante declaradas en la clase), a menos que estas sean también **static**
* Para llamarlo se utiliza el nombre de la clase + nombre del método.

Ejemplo math.sqrt();

### Sobre carga de constructor:

Es cuando en una misma clase tenemos más de un método constrictor.

Como hace java para determinar a qué método constructor hacemos referencia, pues por la cantidad de parámetros que le pasemos a dichos constructores.

Como aviamos visto en clases anteriores, un método constructor debe de tener el mismo nombre que la clase.

### constructor que recibe 5 parámetro

class Empleado{

public Empleado(String nom, double sue, int agno, int mes, int dia) {

nombre = nom;

sueldo = sue;

GregorianCalendar calendario = new GregorianCalendar (agno, mes-1, dia);

altaContrato = calendario.getTime() ;

## constructor que recibe dos parámetros

Public Empleado(String nom, int agno)

}

### Constructor por defecto:

El constructor por defecto pues es uno que tiene el mismo nombre que la clase, pero no recibe parámetros.

### Herencia “Mundo de la programación”

Cuando hablamos de herencia en programación podemos tener como paradigma, a cunado los bienes de un abuelo, pasan a los de su hijo, como ya he mencionado el abuelo, pues es evidente, para ser abuelo, el hijo de este debe de tener hijos, a los cuales heredan de su padre, que a la vez heredan de su abuelo.

En programación no es más diferente que esto, por ejemplo

La **clase** **coche** que se trabajó en videos anteriores, puede ceder, sus: propiedades, métodos etc. A una **clase** nueva que se llame **furgoneta**.

Para que una clase herede los métodos o propiedades que esta posee se utiliza la palabra reservada **extends** delante de la nueva clase, luego la clase de la cual va a heredar.

Sintaxis para proceder a como se hace una herencia.

public class Furgoneta extends Coche {

public Furgoneta () {

}

### Super

¿Qué es esta instrucción? La instrucción **super**, es la que nos permitirá llamará al constructor de la clase padre.

### Como saber quién hereda

Para saber que clase será el padre y cual el hijo debemos de hacer uso de una regla llamada “**ES UN…**”, para entender esto debemos de saber o comprender que es una “superclase y una subclase”, esto es más de lo mismo… quien es el padre, quien es el hijo.

Par seguir con la fase de entendimiento, vamos con un ejemplo de todo esto.

Diseñando la herencia. Clase jefatura: para determinar quién será la superclase o subclase. Es aquí donde hacemos uso de la regla “**ES UN…**”

Ejemplo:

¿Un jefe **es un** empleado? (siempre) Respuesta (si, verdadero)

¿Un Empleado **es un** jefe? (siempre ) Respuesta (no, Falso)

De tal modo que la **superclase** es **empleado** y la **subclase** es **jefe**.

Porque esto es así, pues porque, un jefe dentro del marco explicativo puede ser un empleado y jefe a la vez, pero un empleado no. Visto esto desde el marco explicativo, de que un jefe es un jefe y un empleado, y un empleado solo podrá ser un empleado. Sabemos que en la vida real esto varia, pero para el marco explicativo con esto creo que queda claro.

### Polimorfismo **y** enlazado dinámico

El porlimorfismo no es mas que caracteristica que tienen los metodos de harcer mas de una cosa. Por ejemplo un metodo que es utilizado para hacer una suma, se puede utilizar para la multiplicacion, la division, todo esto es posible por el polimorfismo.

### Polimorfismo. Principio de sustitución

¿Qué es el polimorfismo y el principio de sustitución?

Principio de sustitución: se puede utilizar un objeto de la subclase, siempre que el programa espere un objeto de la superclase.

O lo que es lo mismo: un objeto se puede comportar de diferente forma dependiendo del contexto. Las variables objetos son polimórficas.

### Refundición de objetos (casting) clases y métodos final

### clases y métodos final

Puntos a tener en cuenta, al igual que como habíamos visto la palabra **final** nos permite dejar entendido que no queremos que se modifique el valor de una variable de clase.

Del mismo modo podemos hacerlo con las clases, ¿por qué con las clases? porque de esta forma podemos evitar que otra clase herede de esta.

Otro punto es que del mismo modo que con las **variables** y las **clases**, podemos hacerlo con los métodos. De esta forma evitamos utilizar un método que se hereda, con el mismo nombre.

Ejemplos

Variable de clase **static**

**Int static** valor1= 100;

Clase **static** nada podrá heredar de esta clase

**Static class sexo {**

**Int static** valor1= 100;

**}**

Método **static** para evitar que este método sea utilizado con el mismo nombre

**class sexo {**

**Int static** valor1 = 100;

**Static Public String** DevolverValor1(){

**Return** “El valor es” + valor1;

}

**}**

Debes volver a repasar desde el video 44 en adelante.

### Refundición de objetos (casting)

La refundición de objeto no es más o tan diferente que cuando hacemos un casting de variables.

Ejemplo de tipo de datos primitivos con casting

double var = 45.4;

int var1;

var1 = (int) var;

ejemplo de casting de objeto

Jefatura [método Jefatura] [nombre] = (el tipo a que lo queremos convertir) luego donde se tiene almacenado el objeto de tipo empleado. Entoces quedaría

Jefatura jefe\_finanzas = Jefatura misEmpleados[5];

Esto es posible por la regla **es un** con lo cual podemos saber cuándo tiene sentido hacer un casting de métodos, al yo saber que un jefe es siempre un empleado, y un empleado no es un jefe

Léeme Lo de arriba esta inconcluso debo de entender mejor como se hacen los castings de métodos

### Clases abstractas

Una clase abstracta es la que marca el diseño en la jerarquía de la herencia

Una clase es abstracta desde el momento en que se tiene un método abstracto.

Una clase abstracta no se puede instanciar (es decir, no se puede volver objeto) peros si se pueden heredar y las clases hijas serán las encargadas de agregar funcionalidad.

### Modificadores de acceso

* Public
* Protected
* Private
* Por defecto

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Modificador** | **Clase** | **Package** | **Subclase** | **Todo** |
| **Public** | Si | Si | Si | Si |
| **protected** | Si | Si | Si | No |
| **Private** | Si | No | No | No |
| **Por defecto** | Si | Si | No | No |

### Modificador public

Que implica que nosotros pongamos el modificador de acceso **public** delante de una variable o en la construcción de un método, esto implica que esta variable o método sea visible para:

* implica que al poner la palabra **public** delante de nuestra **variable** o de la construcción un **método**, será visible desde la propia clase.
* También es visible desde el propio paquete (Package) podemos acceder desde otra clase, siempre y cuando este dentro del mismo paquete.
* También es visible desde una subclase, es decir que si heredamos desde una clase que tenga un método o una variable con el modificado **public** será visible dentro de la subclase
* En definitiva, podremos acceder desde cualquier otra clase que este incluso en un paquete diferente, da igual que se una variable o un método, son accesible desde cualquier otra clase.

### Modificador private

También tenemos el modificador **private** el cual será visible solo para:

* La propia clase, en videos anteriores podíamos ver que al poner **private** en una variable o método solo se podía acceder a ella desde la mima **calse**, lo cual se conoce como encapsulamiento, es poco frecuente ver un **private** delante de un método pero se da el caso.

### Modificador por defecto

También hemos visto o mejor dicho no hemos puesto nada delante de una variable, lo cual se conoce como modificador por defecto, también hay que ser distintivos para saber desde qué lugar tiene o no visibilidad, cuando hacemos uso, o mejor dicho cuando no hacemos uso de un modificador, puesto que:

* Cuando no ponemos nada estamos haciendo uso del modificador por defecto
* Este será visible desde la propia clase
* También será visible dentro del propio paquete(package)
* No será visible dentro de una subclase, ojo, aquí es donde hay que ser distintivo, no será visible desde una subclase que este dentro de otro paquete(package), pero si será visible para una subclase que este dentro del mismo paquete(package).
* Por lo cual no es accesible desde todos los sitios

### Modificador protected

El modificador protected y el modificador por defecto son muy parecidos pero tiene un diferencia sutil, pero importante. Una **variable o método,**  que tenga el modificador protected es:

* Accesible desde su propia clase
* También es accesible desde otra clase, siempre que esté en el mismo paquete.
* La diferencia con respecto al modificador de acceso por defecto viene cuando hablamos de una subclase que este dentro de otro paquete, mejor dicho, lo que quiere decir que, aunque este en otro paquete, será visible para una subclase X
* No será accesible desde una clase que no sea subclase y que este en un paquete diferente

### Tipos de enumerados

### Clase ENUM

Que es esto de los tipos enumerados, pues muchas veces surge la necesidad de almacenar en una variable, una serie de valores, pero, no queremos que en esa variable se almacenen, en un futuro más valores, o que también se puedan modificar los valores existentes.

## Interfaces

### ¿Que son la interface?

Son conjunto de directrices que deben cumplir las clases. Cuando hablamos de directrices, hacemos referencia a métodos.

Las interfaces establecen los comportamientos (directrices) a cumplir por las clases.

En definitivas las interfaces lo que hacen es: establecer los comportamientos a cumplir por las clases que implementan esas interfaces. Dichos comportamientos vienen determinados por los métodos que nos obligan a incluir en las clases que implementan las interfaces de turno.

Podemos tener interfaces:

* Predefinidas

Son todas la que vienen predefinida en la API de java

* Propias

Estas son las que construimos nosotros

Característica de las interfaces

* Solo contienen métodos abstractos y constante

Hay que tener en cuenta de las interfaces, que:

* Al igual que las clases, se almacena en un fichero .class, es decir, que a la hora de compilar se genera un .class por cada interfaz.
* Que no se pueden instanciar (no uso de new)
* Todos los métodos son públicos y abstracto, esto quiere decir que solamente se definen al igual que con las clases abstractas.
* No tienen variables, lo que tienen es constante.

Si es tan parecido a una clase abstracta, ¿por qué la necesidad de tener las interfaces? Existen dos razones, una es: la comprobación de tipos estrictos y la otra es el problema de la herencia simple

### Clases internas (inner calss)

¿Que son las clases internas?

Como su nombre lo indica, es una clase dentro de otra

Public class Clase1{

class Clase2{

código clase2

}

Código clase1

}

¿Y por qué serían necesarias?

* Para acceder a los campos privados de una clase desde otra clase.
* Para ocultar una clase de otras pertenecientes al mismo paquete.
* Para crear clases internas “anónimas”, muy útiles para gestionar eventos y retro-llamadas.
* Cuando solo una clase debe acceder a los campos de ejemplar (instanciar,objeto)de otra clase.

Como hemos visto en el curso cuando implementamos una clase que está dentro de otra clase (inner class) esta puede acceder a las variables que están en la clase que la engloban.

Tips de las clases locales:

* No tiene modificación de acceso public

### Clases internas locales ¿Qué son?

* Una clase dentro de un método.
* ¿Cuándo se utilizan estos tipos de clases y por qué?
* Son utilices cuando solo se van a utilizar (instanciar) la clase interna una vez. El objetivo es simplificar aún más el código.
* Su ámbito queda restringido al método donde son declaradas. ¿Ventajas?
* Están muy “encapsuladas”. Ni siquiera la clase al que pertenecen puede acceder a ellas. Tan solo puede acceder a ella el método donde está declarada.
* El código es aún más simple.

Declaración de clase interna local

Class Clase\_externa{

Public void metodo(){

Class clase\_interna\_local{

Código de la clase interna local;

}

Código del método;

}

Código de la clase externa

}

Tips de las clases internas locales:

* No tiene modificación de acceso public

### Entramos a las interfaces

### Interfaces de usuario (swing y AWT)

Java 1.0

Java.awt

Cuando hablamos de interfaces graficas en la versión de **Java 1.0 existe** un paquete llamado “AWT”, que quiere decir esto, pues (abstract Windows toolkit), este paquete es el que delega la creación y el comportamiento que tendrán las interfaces en el S.O. huésped. En otras palabras, dependiendo del sistema operativo, nuestra interface se vería de una forma u otra (en Windows apariencia a, en MAC apariencia b, Linux apariencia c)

Java 1.2

Javax.swing

En la versión de java 1.2 existe un paquete llamado “SWING”, en este paquete la dinámica es completamente lo contrario, ya que los elementos de una interfaz se van pintando sobre una venta, ya no dependen del sistema operativo huésped.

En la versión de java 1.2 el paquete “SWING” es mucho más potente que el “AWT”, puesto que, el paquete SWING hereda todos los métodos, clases etc. Que posee AWT.

La (X) de javax viene de que es un paquete extendidos ya que la versión de java 1.0 se quedaba corto y en la versión 1.1 se decidió colocarle la (X) al paquete, pero para vercion 1.2 se le quita se le quita la (X) a este paquete, y se proporcionó la etiqueta de clases base.

Característica de los frame

Cuando creamos un Frame esto se crean invisible

* Nacen invisible. Se necesita el método **setVisible** para hacerlo visibles.

Tienen un tamaño de 0 pixeles de alto por 0 pixeles de ancho

* Nacen con un tamaño inútil. Se necesita el método **setSize**

Debemos decirle cuando se debe cerrar un frame imaginemos que tenemos tres frame y cerramos uno, Qué debe de hacer el programa, Dejar de ejecutarse, o cerrar solo la ventana que queremos cerrar.

* Conviene decir que debe de hacer el programa si se cierra un frame,

Para crear un marco se puede hacer de formas diferentes, se pueden hacer todos en una única clase, puedes dividir tu aplicación grafica para crear las ventanas en dos clases, puedes crearlas en dos ficheros fuentes o en uno.

Como dar color a un frame

Para ello vamos a utilizar una clase llamada

Retomando el curso

Los eventos

Un evento es una desencadénate de una acción, por ejemplo, cuándo damos click a un botón ese desencadena un evento que puede ser calcular, abrir un marco, etc.

# Eventos: oyentes y fuentes

## Definicion: desencadenante de la acción.

Entendiendo los eventos

Para entender los eventos existen **tres Factores**:

* ¿Qué desencadena la acción?

En java, esto se conoce como: el objeto evento

* ¿Quién desencadena la acción?

En java esto se conoce como el objeto fuente

* ¿Quién recibe la acción?

En java esto se conoce como el objeto Listener (quien lo escucha)

Para entender cómo funcionan los eventos, vámonos con un ejemplo.

Imaginemos que tenemos un botón y una imagen, al hacer clic en el botón, el clic se convierte en el objeto evento o, mejor dicho, lo que desencadena un evento. este botón se convierte en el **objeto fuente** o, mejor dicho, quien desencadena una acción. Quien recibe la acción es el **objeto Listener** en este caso la imagen.

en resume:

¿Qué desencadena el la acción?

al hacer clic

¿Quién desencadena la acción? el botón

¿Quién recibe la acción? imagen

Hola a todos como esta

Layouts (Disposiciones)

FlowLayout

Este contenedor es el contenedor por defecto, es por ello que cuando creábamos botones, estos estaban alineado o mejor dicho posicionado en el centro. Ejemplo:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  |  |  |  | |

BorderLayout

este contenedor consiste en dividir el contenedor de modo que puedan ser ubicados los elementos de acuerdo a los 4 puntos cardinales, por ejemplo:

|  |  |  |
| --- | --- | --- |
| Zona superior o la zona (north) norte | | |
| Zona izquierda o la zona (west) oeste | Zona superior o la zona (central) norte | Zona derecha o la zona (east) este |
| Zona inferio o la zona (south) sur | | |

GgrdLayout

consiste en tomar tu contenedor y dividirlo en varias filas y columnas de tal forma que quedan como celdas.

Esto es un contenedor, en el cual pueden ponerse botones, menús, casilla de verificación en la seda que así consideres

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| botones |  |  |  |  |  |  |  |  |  |
|  |  |  | casillas de verifica r |  |  |  |  |  |  |
|  |  |  |  |  |  |  | menús |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |