**Contenido**

[1. Programación Orientada a Objetos (POO). 3](#_heading=h.30j0zll)

[1.1 Introducción. 3](#_heading=h.1fob9te)

[1.2.- Creación de una clase. 4](#_heading=h.2et92p0)

[1.3. Conceptos fundamentales. 4](#_heading=h.tyjcwt)

[3. Clases 5](#_heading=h.3dy6vkm)

[4. Objetos, miembros y referencias 6](#_heading=h.2s8eyo1)

[5. Conceptos básicos. Resumen 8](#_heading=h.3rdcrjn)

[6. Clases - Constructores 9](#_heading=h.26in1rg)

[6.1. Noción de constructor 9](#_heading=h.35nkun2)

[6.2. Constructor no-args. 9](#_heading=h.44sinio)

[6.3. Sobrecarga de constructores. 10](#_heading=h.z337ya)

[6.4. Constructor Primario y secundario 11](#_heading=h.3j2qqm3)

[7. Clases - Miembros estáticos 11](#_heading=h.1y810tw)

[7.1 Datos estáticos 11](#_heading=h.2xcytpi)

[7.2 Métodos estáticos 12](#_heading=h.3whwml4)

[8. Clases - Otros aspectos 13](#_heading=h.2bn6wsx)

[8.1. Inicialización de variables 13](#_heading=h.3as4poj)

[8.2. Ámbito de las variables 13](#_heading=h.49x2ik5)

[8.3. Recogida de basura 14](#_heading=h.147n2zr)

[8.4. Sobrecarga de métodos 14](#_heading=h.3o7alnk)

[8.5. La referencia this 15](#_heading=h.ihv636)

[8.5. El método toString() 15](#_heading=h.32hioqz)

[9. Gestión de Paquetes 16](#_heading=h.1hmsyys)

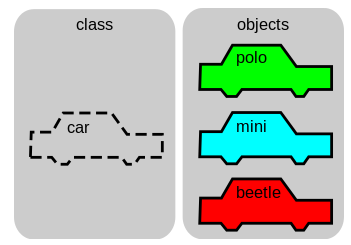
[10. Clase enumerada 16](#_heading=h.41mghml)

# Programación Orientada a Objetos (POO).

## 1.1 Introducción.

El lenguaje Kotlin es lo que se conoce como un Lenguaje Orientado a Objetos y, por lo tanto, sigue dicho paradigma. Este paradigma se basa en la idea de que cualquier programa está formado por objetos y que todo puede ser representado como tal. Así, cualquier elemento que forme parte de una aplicación (un usuario, una factura o pedido, un coche para una aplicación de gestión de un taller, . . .) se considera que es un objeto de la aplicación, con una serie de propiedades y características. A estos objetos también se les suele llamar instancias.

Esta idea de objeto viene acompañada por la definición de su estructura, que es lo que llamamos clase. Habíamos visto otras definiciones de clase. Básicamente una clase es una estructura de código que define los atributos y comportamientos de todos los objetos que pertenecen a un mismo tipo.



A partir del concepto de objeto, el paradigma de POO propone una serie de características que lo definen y que se irán comprendiendo a lo largo de este y siguientes temas, según se vayan exponiendo los diferentes mecanismos de este paradigma para proporcionarlas:

* **Abstracción**: Por el que el programador se abstrae, se despreocupa, de los detalles de implementación de cualquier objeto. Los procesos o métodos que se encuentren definidos funcionan por sí solos y no es necesario saber cómo están implementados si sólo necesitamos hacer que se ejecuten.
* **Encapsulamiento**: Todas las características o propiedades que pertenezcan a un sólo elemento del programa se pueden crear y encapsular dentro de él, aumentando la cohesión de estos componentes.
* **Polimorfismo**: Más adelante veremos cómo invocar al mismo método desde distintos objetos, cada uno de esos objetos puede responder de forma distinta.
* **Herencia**: La herencia entre clases permitirá que se puedan definir nuevas clases basadas en unas ya existentes a fin de reutilizar el código, generando así una jerarquía de clases dentro de una aplicación.

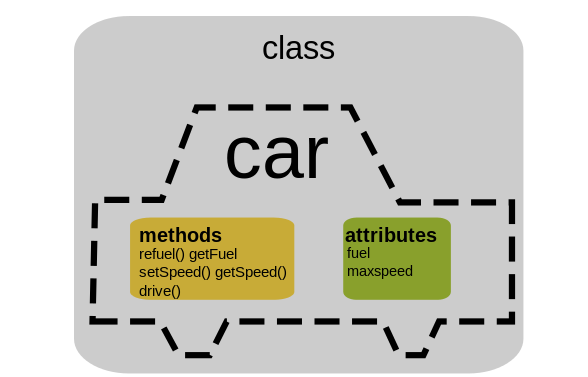
## 1.2.- Creación de una clase.

Una clase es una estructura de código que define las características y operaciones que tiene y puede realizar todos los objetos que se creen a partir de ella.

Ejemplos:

* Si necesito crear para mi capa de negocio un objeto por cada Sala disponible en un Cine, será necesario crear la clase Sala.
* Si en un videojuego de estrategia voy a disponer de múltiples soldados, necesitaré crear la clase Soldado.

A continuación, podemos ver la estructura de una clase:



## 1.3. Conceptos fundamentales.

**Objetos**: unidad que engloba dentro de sí un conjunto de datos y módulos necesarios para el tratamiento de esos datos. Cada objeto contiene datos y funciones.

**Atributos**: Son los datos incluidos en el objeto. Parecidas a las variables de C clásico pero incluidas en los objetos.

**Métodos**: Son módulos o funciones que pertenecen a los objetos y que manejarán los atributos.

**Mensajes**: El modo mediante el que se comunican los objetos. Consiste en llamar a un método de un objeto.

**Interfaz**: Las clases (y, por lo tanto, los objetos) tienen partes públicas y partes privadas. La parte pública es visible para todos los objetos, mientras que la parte privada es sólo visible para el propio objeto. A la parte pública de un objeto se le denomina interfaz.

# 3. Clases

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

Una clase es una agrupación de datos y de código que actúa sobre esos datos, a la que se le da un nombre.

Una clase contiene:

* Datos (se denominan Datos Miembro). Estos pueden ser de tipos primitivos o referencias.
* Métodos (se denominan Métodos Miembro).

La sintaxis general para la declaración de una clase es:

*modificadores* **class** *nombre\_clase* **{**  
       *declaraciones\_de\_miembros* **;**  
**}**

Por ejemplo:

**class** Punto **{**  
     **int** x;  
     **int** y;  
**}**

**Modificador de visibilidad:** Se indicará (o no) la visibilidad, también se le conoce como modificador de acceso (o scope) de la clase con respecto al resto. Por lo general se indicará una visibilidad completa a través de la palabra reservada **public**.

public: Esto se utiliza por defecto en Kotlin para las clases, interfaces, funciones y propiedades, lo que significa que son visibles para todo el proyecto. Puedes usar explícitamente public para mayor claridad, pero no es necesario.

class MiClase {

public val miAtributo = 42

}

protected: En Kotlin, el modificador protected hace que el atributo sea visible para la propia clase y las clases que hereden de ella, pero no es visible para otras clases en el mismo paquete.

protected class MiClaseProtegida {

protected val miAtributo = 42

}

private: El modificador private hace que el atributo sea visible solo para la propia clase. Esto es igual que en Java.

class MiClasePrivada {

private val miAtributo = 42

}

Sin especificar un modificador de acceso: Si no se especifica ningún modificador de acceso, el atributo estará accesible dentro del paquete donde sea definida su clase, pero no fuera de ese paquete.

class MiClaseDefault {

val miAtributo = 42

}

Es importante tener en cuenta que estos niveles de accesibilidad también se aplican a funciones y clases. Puedes utilizar estos modificadores de acceso según tus necesidades para controlar la visibilidad y el acceso a los miembros de tus clases en Kotlin.

# 4. Objetos, miembros y referencias

Un objeto es una instancia (ejemplar) de una clase. La clase es la definición general y el objeto es la materialización concreta (en la memoria del ordenador) de una clase.

El fenómeno de crear objetos de una clase se llama instanciación.

Los objetos se manipulan con referencias. Una referencia es una variable que apunta a un objeto. Las referencias se declaran igual que las variables de Tipos primitivos (tipo nombre).

Ejemplo:

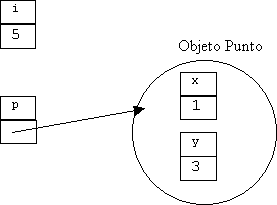
var p: Punto  
var p = Punto()

La primera línea del ejemplo declara una referencia (p) que es de Tipo Punto. La referencia no apunta a ningún sitio. En la segunda línea se crea un objeto de Tipo Punto y se hace que la referencia p apunte a él. Se puede hacer ambas operaciones en la misma expresión:

 var p = Punto ()

A los miembros de un objeto se accede a través de su referencia. La sintaxis es:

*nombre\_referencia****.****miembro*

En el ejemplo, se puede poner:

p.x = 1;  
p.y = 3;

var p = Punto ()

p.x = 1;  
p.y = 3;

var i = 5;

Es importante señalar que, en el ejemplo, p no es el objeto. Es una referencia que apunta al objeto.

Los métodos miembros se declaran dentro de la declaración de la clase, Por ejemplo:

**class** Circulo {  
        var centro:Punto() //dato miembro.Referencia a un objeto punto  
        var radio:Int      // dato miembro. Valor primitivo  
        **fun** superficie() : Float {   // método miembro.   
                **return** 3.14 \* radio \* radio  
        }        // fin del método superficie  
}                   // fin de la clase Circulo

El acceso a métodos miembros es igual que el que ya se ha visto para datos miembro. En el ejemplo:

Var c = Circulo()  
c.centro.x = 2  
c.centro.y = 3  
c.radio = 5  
var s: Float

s = c.superficie()

Es interesante observar en el ejemplo:

* Los datos miembros pueden ser tanto primitivos como referencias. La clase Círculo contiene un dato miembro de tipo Punto (que es el centro del círculo).
* El acceso a los datos miembros del Punto centro se hace encadenando el operador “.” en la expresión c.centro.x que se podría leer como 'el miembro x del objeto (Punto) centro del objeto (Circulo) c'.
* Aunque el método superficie no recibe ningún argumento los paréntesis son obligatorios (Distinguen los datos de los métodos).
* Existe un Objeto Punto para cada instancia de la clase Circulo (que se crea cuando se crea el objeto Circulo).

# 5. Conceptos básicos. Resumen

* Una Clase es una definición de un nuevo Tipo, al que se da un nombre.
* Una Clase contiene Datos Miembro y Métodos Miembro que configuran el estado y las operaciones que puede realizar.
* Un Objeto es la materialización (instanciación) de una clase. Puede haber tantos Objetos de una Clase como resulte necesario.
* Los Objetos se crean (se les asigna memoria).
* Los Objetos se manipulan con Referencias.
* Una Referencia es una Variable que apunta a un Objeto.
* El acceso a los elementos de un Objeto (Datos o métodos) se hace con el operador. (punto): *nombre\_referencia****.****miembro*

# 6. Clases - Constructores

## 6.1. Noción de constructor

Cuando se crea un objeto (se instancia una clase) es posible definir un proceso de inicialización que prepare el objeto para ser usado. Esta inicialización se lleva a cabo invocando un método especial denominado constructor. Los constructores tienen algunas características especiales:

* El nombre del constructor tiene que ser “constructor”.
* Puede recibir cualquier número de argumentos de cualquier tipo, como cualquier otro método.
* No devuelve ningún valor.

**El constructor no es un miembro más de una clase. Sólo es invocado cuando se crea el objeto. No puede invocarse explícitamente en ningún otro momento.**

Continuando con los ejemplos se podría escribir un constructor para la clase Punto, de la siguiente forma:

**class** Punto {  
    var x:Int; var y:Int  
    constructor ( a:Int , b:Int ) {  
        x = a ; y = b ;  
    }  
}

Con este constructor se crearía un objeto de la clase Punto de la siguiente forma:

Var p = Punto (1, 2);

## 6.2. Constructor no-args.

Si una clase no declara ningún constructor, Kotlin incorpora un constructor por defecto (denominado constructor no-args) que no recibe ningún argumento y no hace nada.

Si se declara algún constructor, entonces ya no se puede usar el constructor no-args. Es necesario usar el constructor declarado en la clase.

En el ejemplo el constructor no-args sería:

**class** Punto {  
 var x:Int; var y:Int  
    constructor ( ) { }  
}

## 6.3. Sobrecarga de constructores.

Una clase puede definir varios constructores (un objeto puede inicializarse de varias formas). Para cada instanciación se usa el que coincide en número y tipo de argumentos. Si no hay ninguno coincidente se produce un error en tiempo de compilación.

Por ejemplo:

**class** Punto {  
    var x:Int; var y:Int  
    constructor ( a:Int , b:Int ) {  
        x = a ; y = b ;  
    }  
 constructor ( ) { }  
        x = 0 ; y = 0;  
    }  
}

En el ejemplo se definen dos constructores. El citado en el ejemplo anterior y un segundo que no recibe argumentos e inicializa la variable miembro a 0.

Desde un constructor puede invocarse explícitamente a otro constructor utilizando la palabra reservada **this** , **this** es una referencia al propio objeto. Cuando **this** es seguido por paréntesis se entiende que se está invocando al constructor del objeto en cuestión. Puedes ver el uso más habitual de this. El ejemplo anterior puede reescribirse de la siguiente forma:

**class** Punto {  
    var x:Int; var y:Int  
    constructor ( a:Int , b:Int ) {  
        x = a ; y = b ;  
    }  
    constructor () {  
        **this** (0,0);  
    }  
}

Cuando se declaran varios constructores para una misma clase estos deben distinguirse en la lista de argumentos, bien en el número, bien en el tipo.

Esta característica de definir métodos con el mismo nombre se denomina sobrecarga y es aplicable a cualquier método miembro.

## 6.4. Constructor Primario y secundario

El constructor primario es el constructor principal de una clase y se define dentro de los paréntesis en la declaración de la clase. Puede tener parámetros y estos parámetros se usan para inicializar los atributos de la clase.

Los constructores secundarios son otros constructores que pueden tener diferentes parámetros y se utilizan para proporcionar diferentes maneras de crear una instancia de una clase. Cada constructor secundario debe llamar al constructor primario usando la palabra clave "this".

Ejemplo de una clase con un constructor primario y un constructor secundario:

class **Person(var name: String, var age: Int)** { //primario

constructor(name: String) : this(name, 0) { //secundario

}

}

La clase "Person" tiene un constructor primario con dos parámetros "name" y "age". También tiene un constructor secundario con un solo parámetro "name", que llama al constructor primario y proporciona un valor predeterminado para la edad.

# 7. Clases - Miembros estáticos

## 7.1 Datos estáticos

Las variables estáticas son un tipo especial de variables que como su nombre indica son estáticas ya que no se crean dinámicamente con cada objeto o instancia de la clase a la que pertenecen. Esto quiere decir que no necesitamos una instancia de clase para utilizar este tipo de variable. Además, una vez que esta variable es creada es compartido por todos los objetos o instancias de clase que se hayan creado.

Por ejemplo:

class Punto {

var x: Int

var y: Int

companion object {

//como sería en java🡪   static int numPuntos = 0;

var numPuntos: Int = 0;

}  
  constructor ( int a , int b ) {  
        x = a

y = b  
        numPuntos ++   
    }

fun getNumpuntos():Int{

return numPuntos;

}

}

En el ejemplo numPuntos es un contador que se incrementa cada vez que se crea una instancia de la clase Punto.

El acceso a las variables estáticas desde fuera de la clase donde se definen se realiza a través del nombre de la clase y no del nombre del objeto como sucede con las variables miembro normales (no estática). En el ejemplo anterior puede escribirse:

**var** x: Int = Punto.numPuntos

No obstante, también es posible acceder a las variables estáticas a través de una referencia a un objeto de la clase y un método. Por ejemplo:

var p = Punto()  
**int** x = p.getNumpuntos()

**Las variables estáticas de una clase existen, se inicializan y pueden usarse antes de que se cree ningún objeto de la clase.**

## 7.2 Métodos estáticos

Para los métodos, la idea es la misma que para los datos: los métodos estáticos se asocian a una clase, no a una instancia. Se crean dentro de companion object

# 8. Clases - Otros aspectos

## 8.1. Inicialización de variables

Desde el punto de vista del lugar donde se declaran existen dos tipos de variables:

* Variable miembro: Se declaran en una clase, fuera de cualquier método.
* Variables locales: Se declaran y usan en un bloque de código dentro de un método.

Las variables miembros se recomienda inicializarlas, de la siguiente forma:

* Las numéricas a 0.
* Las booleanas a **false**.
* Las char al carácter nulo (hexadecimal 0).

Los nulos en Kotlin no existen mientras no se diga lo contrario. Es decir, a ningún objeto, por defecto, se le puede asignar null.

Por tanto, esto no compilará:

val x: Int = null

Si quieres que una variable acepte nulos, tienes que marcar el tipo con una “?”:

val x: Int? = null

Las variables locales se aconseja asignarles un valor antes de ser usadas.

## 8.2. Ámbito de las variables

El ámbito de una variable es el área del programa donde la variable existe y puede ser utilizada. Fuera de ese ámbito la variable, o bien no existe o no puede ser usada (que viene a ser lo mismo).

El ámbito de una variable miembro (que pertenece a un objeto) es el de la usabilidad de un objeto. Un objeto es utilizable desde el momento en que se crea y mientras existe una referencia que apunte a él. Cuando la última referencia que lo apunta sale de su ámbito el objeto queda 'perdido' y el espacio de memoria ocupado por el objeto puede ser recuperado por la JVM cuando lo considere oportuno. Esta recuperación de espacio en memoria se denomina 'recogida de basura'.

El ámbito de las variables locales es el bloque de código donde se declaran. Fuera de ese bloque la variable es desconocida.

Ejemplo:

{  
    var x:Int=0     // empieza el ámbito de x. (x es conocida y utilizable)  
    {  
        **var** q:Int=0    // empieza el ámbito de q. x sigue siendo conocida.  
        . . .  
    }            // finaliza el ámbito de q (termina el bloque de código)  
     . . .        // q ya no es utilizable   
}                // finaliza el ámbito de x

## 8.3. Recogida de basura

Cuando ya no se necesita un objeto simplemente puede dejar de referenciarse. No existe una operación explícita para 'destruir' un objeto o liberar el área de memoria usada por él.

La liberación de memoria la realiza el recolector de basura (*garbage collector*) que es una función de la JVM. El recolector revisa toda el área de memoria del programa y determina que objetos pueden ser borrados porque ya no tienen referencias activas que los apunten. El recolector de basura actúa cuando la JVM lo determina.

## 8.4. Sobrecarga de métodos

Una misma clase puede tener varios métodos con el mismo nombre siempre que se diferencien en el tipo o número de los argumentos. Cuando esto sucede se dice que el método está sobrecargado. Por ejemplo, una misma clase podría tener los métodos:

**fun** metodoSobrecargado() { . . .}  
**fun** metodoSobrecargado(x:Int) { . . .}

Sin embargo, no se puede sobrecargar cambiando sólo el tipo del valor devuelto. Por ejemplo:

**fun** metodoSobrecargado():Int { . . .}  
**fun** metodoSobrecargado():Boolen { . . .}  // error en compilación

con esta definición, en la expresión y .*metodoSobrecargado*() la JVM no sabría que método invocar.

Se puede sobrecargar cualquier método miembro de una clase, así como el constructor.

## 8.5. La referencia this

En ocasiones es conveniente disponer de una referencia que apunte al propio objeto que se está manipulando. Esto se consigue con la palabra reservada **this**, **this** es una referencia implícita que tienen todos los objetos y que apunta a sí mismo. Por ejemplo:

**class** Circulo {  
    var centro: Punto  
    **var** radio:Int  
    . . .

constructor(centro:Punto, radio:Int){

this.centro = centro

this.radio = radio

}  
    fun elMayor(c:Circulo):Circulo {  
        **if** (radio > c.radio) **return** **this**;  
        **else** **return** c;  
    }  
}

El método *elMayor* devuelve una referencia al círculo que tiene mayor radio, comparando los radios del Círculo c que se recibe como argumento y el propio. En caso de que el propio resulte mayor el método debe devolver una referencia a sí mismo. Esto se consigue con la expresión **return** **this.**

## 8.5. El método toString()

El método toString() es un método de la clase base en Kotlin y en muchos otros lenguajes de programación orientados a objetos. Su objetivo principal es proporcionar una representación de cadena legible de un objeto. Cuando se llama al método toString() en un objeto, devuelve una cadena que describe ese objeto en un formato que sea comprensible para los humanos.

En Kotlin, cada clase hereda el método toString() de la clase base Any. Si deseas proporcionar tu propia implementación de toString() para una clase personalizada, puedes hacerlo anulando el método.

class Persona(val nombre: String, val edad: Int) {

// Anulación del método toString para proporcionar una representación personalizada

override fun toString(): String {

return "Nombre: $nombre, Edad: $edad"

}

}

fun main() {

val persona = Persona("Juan", 30)

println(persona.toString()) // Esto imprime "Nombre: Juan, Edad: 30"

println(persona) // otra forma de llamar al método "Nombre: Juan, Edad: 30"

}

# 9. Gestión de Paquetes

Kotlin gestiona los paquetes de una manera muy similar a Java, ya que comparte la máquina virtual de Java (JVM). Los paquetes en Kotlin son utilizados para organizar y estructurar el código en módulos lógicos, y facilitan la gestión de la visibilidad y el acceso a clases, funciones y propiedades.

Declaración de un paquete: Para declarar un paquete en Kotlin, simplemente coloca la declaración de paquete al comienzo del archivo fuente antes de cualquier otra declaración.

Ejemplo: package com.ejemplo.miproyecto

Esta declaración indica que todas las clases, funciones y propiedades definidas en ese archivo pertenecen al paquete "com.ejemplo.miproyecto".

Ubicación en el sistema de archivos: En general, la ubicación del archivo fuente en el sistema de archivos debe reflejar la jerarquía de paquetes. Por ejemplo, si estás usando el paquete "com.ejemplo.miproyecto", el archivo fuente debería estar en un directorio con la estructura com/ejemplo/miproyecto.

Importación de paquetes: Para acceder a clases, funciones y propiedades de otros paquetes en Kotlin, debes importar esos paquetes en tus archivos fuente. Puedes hacerlo usando la palabra clave import.

Ejemplo : import com.ejemplo.ClaseDeOtroPaquete

# 10. Clase enumerada

Mediante enum se puede definir un tipo enumerado, de esta forma un atributo solo podrá tener uno de los posibles valores que se dan como opción. Los valores que se especifican en el tipo enumerado se suelen escribir con todas las letras en mayúscula.

Ejemplo de uso: El sexo de un animal solo puede ser macho, hembra o hermafrodita. Una forma de delimitar los valores que puede tomar un atributo es definir un tipo enumerado.

enum class Sexo {

MACHO,

HEMBRA,

HERMAFRODITA

}

fun main() {

val animal1 = Animal("Perro", Sexo.MACHO)

val animal2 = Animal("Gato", Sexo.HEMBRA)

val animal3 = Animal("Caracol", Sexo.HERMAFRODITA)

println("Animal 1: ${animal1.nombre}, Sexo: ${animal1.sexo}")

println("Animal 2: ${animal2.nombre}, Sexo: ${animal2.sexo}")

println("Animal 3: ${animal3.nombre}, Sexo: ${animal3.sexo}")

}

Otro Ejemplo:

enum class DiaSemana {

LUNES,

MARTES,

MIÉRCOLES,

JUEVES,

VIERNES,

SÁBADO,

DOMINGO

}

fun main() {

val diaActual = DiaSemana.MIÉRCOLES

val diasSemana = DiaSemana.values() // Obtiene un array de todos los valores de la clase enumerada deprecated

for (dia in diasSemana) {

println("Día: $dia")

}

// Ejemplo de uso de la clase enumerada

when (diaActual) {

DiaSemana.LUNES -> println("Es lunes, ánimo.")

DiaSemana.VIERNES -> println("¡Viernes! Fin de semana a la vista.")

else -> println("Es un día cualquiera.")

}

}

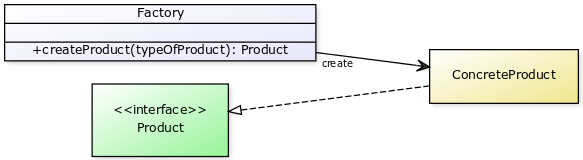
# 11.- Patrones de diseño. Factoría de clases.

Los patrones de diseño son, en definitiva, soluciones a problemas recurrentes y que se ha documentado que funcionan y resuelven el problema de manera eficiente.

Tenemos diferentes patrones de diseño, puedes consultarlos en <https://refactoring.guru/es/design-patterns/java>

#### En esta sección vamos a estudiar un patrón muy sencillo la factoría de clases (Factory Method). Es un patrón de diseño que pertenece a la categoría de los patrones creacionales y nos ayuda a crear distintos objetos sin exponer la lógica de su creación gracias a la implementación de una interfaz común.

## Estructura del patrón Factory



La estructura del patrón Factory comprende las siguientes clases:

**Clase Factory:** Nos crea el producto que necesitamos.

**Clase Producto Concreto:** Implementa la interfaz Producto y desarrolla su método con la lógica correspondiente.

**Interfaz del Producto:** Contiene los métodos comunes que van a reunir los productos concretos. Se verá en otro tema.

# Tenemos un ejemplo sencillo en factoriaEjemplo2