

# LIBRO ANDROID

1. Introducción.....	3
1.1. Tecnologías de comunicación móvil .....	3
1.2. Androide.....	4
1.3. iOS .....	9
2. Tipos de aplicaciones móviles.....	11
2.1. Aplicaciones nativas.....	11
2.2. Aplicaciones web responsivas (WebApps).....	11
2.3. Aplicaciones híbridas.....	11
2.4. Aplicaciones web progresivas.....	12
2.5. Aplicaciones compiladas.....	13
3. Idioma Kotlin .....	14
3.1. Introducción.....	14
3.1.1. Instalación de Kotlin.....	14
3.1.2. Variables y tipos de datos .....	15
Plantillas de cadenas.....	dieciséis
Tipos que aceptan valores NULL y operador de Elvis .....	dieciséis
3.1.3. Operadores.....	17
Operador de asignación.....	17
Operadores aritméticos .....	17
Operadores relacionales .....	17
Operadores logicos.....	18
Operador condicional ternario.....	18
3.1.4. Estructura de control.....	18
Si-si no.....	18
Cuando.....	18
3.1.5. Excepciones.....	20
3.1.6. Funciones.....	21
Definición e invocación .....	21

Conceptos básicos de las expresiones Lambda.....	21
Funciones anónimas.....	23
3.1.7. POO .....	24
Clases y objetos.....	24
Herencia.....	24
Encapsulación.....	26
Uso de genéricos.....	27
Clases de enumeración.....	27
Objetos únicos.....	28
3.1.8. Tipos de datos complejos .....	29
Clases y clases de datos.....	29
Lista/Lista Mutable .....	30
Conjunto / Conjunto Mutable.....	31
Mapa / Mapa Mutable.....	31
Matrices.....	32
Funciones de orden superior con colecciones.....	32
para cada().....	32
mapa().....	33
filtro() .....	34
agrupar por().....	34
doblar().....	35
ordenado por().....	36

## 1. INTRODUCCIÓN

En esta sección introductoria, echaremos un vistazo rápido a las principales tecnologías a nuestra disposición para el desarrollo de aplicaciones portátiles.

Ahora, desea crear una aplicación móvil:

- ¿Qué es lo primero que se te pasa por la cabeza?
- ¿Qué lenguaje de programación se debe utilizar para el desarrollo?
- ¿En qué entorno de desarrollo?
- ¿Y para qué dispositivos? ¿Teléfonos móviles? ¿Tabletas? ¿Relojes inteligentes? ¿Televisores?

Es una cuestión comparable en complejidad a cuando contemplamos el desarrollo de una aplicación para una computadora personal.

### 1.1. Tecnologías de comunicación móvil

La red de comunicaciones móviles ha ido evolucionando con el tiempo con las diferentes generaciones aportando cambios respecto a la anterior. Las diferentes generaciones y sus principales características son:

- Generación 0: Esta generación puede considerarse las primeras comunicaciones entre dispositivos móviles mediante ondas de radio.
- Primera generación (tecnología 1G): la primera red automatizada de comunicaciones móviles se lanzó en 1979 en Japón y tenía problemas con las comunicaciones de voz y limitaciones de seguridad ya que las llamadas de voz se replicaban en las torres de radio.
- Segunda generación (tecnología 2G): años 90. Fue el primero en ofrecer voz y datos digitales, así como roaming internacional. A partir de la década de 2000, introdujo paquetes en la red para proporcionar transferencia de datos de alta velocidad e Internet de alta velocidad.
- Tercera generación (tecnología 3G): Su principal objetivo era ofrecer un aumento de las tarifas de datos, facilitando el crecimiento y ofreciendo mayor capacidad de voz y datos, además de abaratar el coste de las transmisiones.
- Cuarta generación (tecnología 4G): El objetivo principal era proporcionar alta velocidad, alta calidad, alta capacidad y todo ello a precios bajos tanto para voz como para datos, además de aumentar la seguridad de las comunicaciones.
- Quinta generación (tecnología 5G): Esta tecnología, que inició su lanzamiento comercial en 2019, es 10 veces más rápida que 4G, alcanzando una velocidad de transmisión de 1Gbps.

- Sexta generación (tecnología 6G): Su principal objetivo será reducir la latencia de las conexiones y aumentar la velocidad de transmisión. Se estima que la comercialización será en 2030 y que los primeros casos de uso reales podrían llegar a partir de 2026.

1.2. ANDROIDE

Android es un sistema operativo desarrollado por Google y basado en el Kernel de Linux, creado específicamente para dispositivos portátiles con pantalla táctil: teléfonos móviles, tabletas, relojes inteligentes, televisores o incluso algunos coches.


El viaje del sistema operativo móvil Android comenzó con el debut público de su versión beta inaugural el 5 de noviembre de 2007. Posteriormente, la versión comercial inicial Android 1.0 vio la luz el 23 de septiembre de 2008. Google ha supervisado consistentemente el desarrollo de este sistema operativo en un anualmente desde 2011. Los principales lanzamientos nuevos se dan a conocer durante Google I/O, acompañados simultáneamente por el lanzamiento de la primera versión beta pública para dispositivos Google Pixel compatibles. Posteriormente, la versión estable se lanzará más adelante este año.



NOMBRE	VERSIÓN	FECHA DE LANZAMIENTO	PRINCIPALES MEJORAS
Androide 1.0	1,0 - 1,1	septiembre 2008	Primera versión estable
Magdalena Androide	1.5	abril de 2009	Diseño refinado Teclado virtual Widgets para aplicaciones Copiar y pegar en el navegador Transiciones animadas Rotación automática de pantalla
Donut Android	1.6	septiembre 2009	Búsqueda rápida Mercado Android renovado Adaptado a más formatos de pantalla sintetizador de voz Mejoras en la cámara y la galería. Soporte CDMA y VPN

Pastel de Android	2.0 - 2.1	octubre 2009	Rutas en mapas Soporte para múltiples cuentas Fondos de pantalla vivos Soporte de flash y zoom Mejoras en aplicaciones preinstaladas como Mapas, navegador o calendario
androide froyo	2.2 - 2.2.3	mayo de 2010	Comandos de voz Puntos de acceso Wi-Fi Rendimiento mejorado del navegador soporte flash Notificaciones push C2DM Mover aplicaciones a SD
Pan de jengibre Android	2.3 - 2.3.7	diciembre 2010	API para juegos NFC Primer huevo de pascua Cambios en el diseño de iconos Soporte para resolución WXGA y superior Seleccionar antes de copiar Soporte para múltiples cámaras Soporte de giroscopio y barómetro. Videollamadas en Hangouts
Panel de Android	3.0 - 3.2.6	febrero 2011	Adaptado para tabletas Barra del sistema Ajustes rápidos Pestañas del navegador Aceleración de hardware Soporte USB OTG
Helado Android Sándwich	4.0 - 4.0.4	octubre 2011	Interfaz holográfica Barra de navegación Carpetas Tipografía robótica Capturas de pantalla Desbloqueo facial Descartar notificaciones una por una
Android Jelly Bean	4.1 - 4.3.1	julio 2012	Google ahora Movimiento más suave Ajustes rápidos Mejor accesibilidad Widgets en la pantalla de bloqueo Soporte de emoji nativo
Android KitKat	4.4 - 4.4.4	octubre 2013	Cambios de diseño Modo inmersivo ARTE Aplicación renovada de reloj, teléfono y descargas
Piruleta Android	5.0 - 5.1.1	noviembre 2014	Diseño de materiales Nueva pantalla de bloqueo (sin widgets) Mejoras de rendimiento Mejoras recientes Buscador de configuraciones

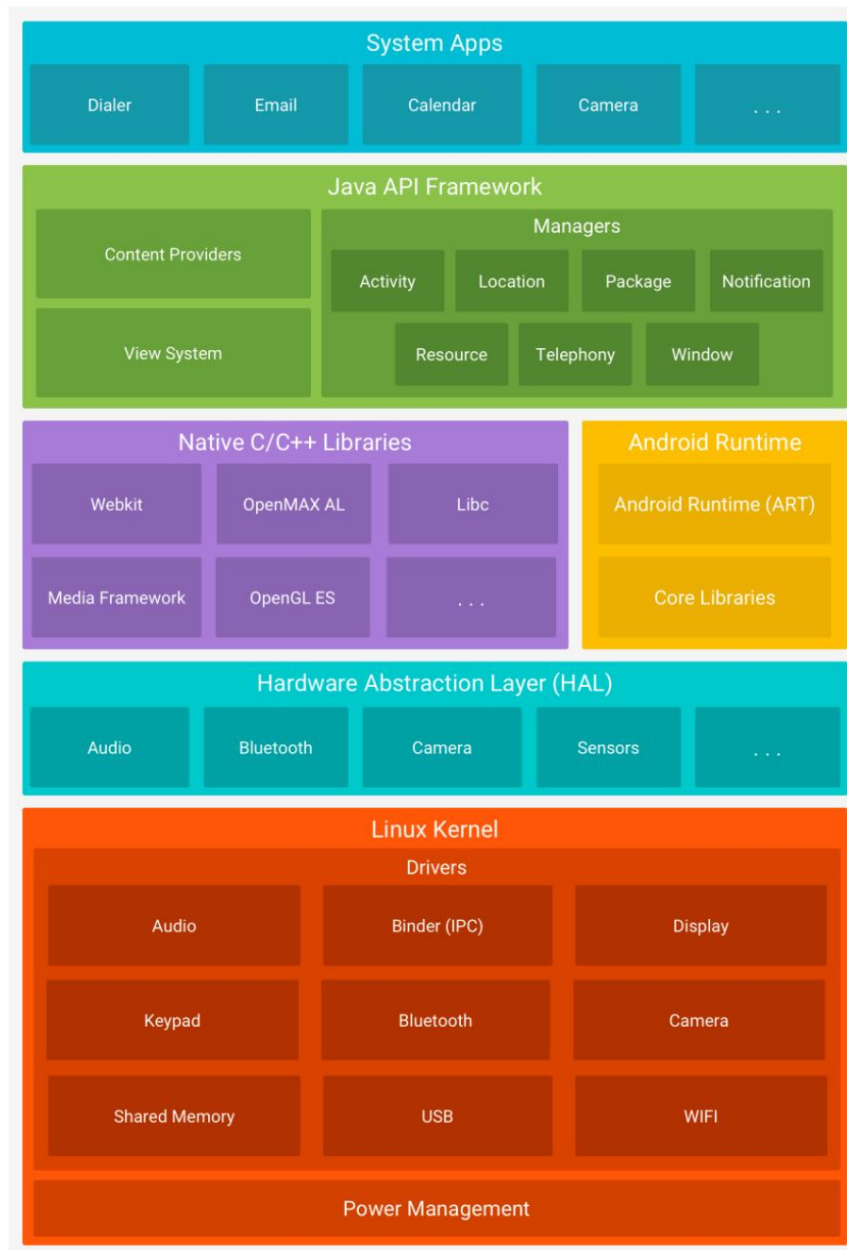
Androide malvavisco	6.0 - 6.0.1	octubre 2015	Permisos de tiempo de ejecución Modo dormido Compatibilidad con USB-C y 4K Soporte para lector de huellas dactilares Ventana múltiple experimental Compartir directo Ahora al alcance de la mano
Turrón Android	7.0 - 7.1.2	agosto 2016	Mejoras en el sueño Mejoras en el compilador JIT Ensueño de realidad virtual Modo de ventanas múltiples PIP en AndroidTV Vulcano 3D Configuración rápida de la aplicación Atajos del iniciador
Android Oreo	8,0 - 8,1	agosto 2017	Proyecto agudos Modo PIP móvil Iconos adaptativos Cambios de notificación API de autocompletar Optimizaciones de rendimiento
Pastel de Android	9.0	agosto 2018	Mejoras de privacidad Brillo y batería inteligente Acciones de la aplicación Porciones de aplicaciones Bienestar digital Navegación por gestos
androide 10 (Tarta de membrillo)	10.0	septiembre 2019	modo oscuro Subtítulos en tiempo real Respuestas inteligentes Nueva navegación por gestos Optimizaciones plegables Mejoras de privacidad Actualizaciones del sistema Google Play
androide 11 (Pastel de terciopelo rojo)	11.0	septiembre 2020	Cambios en las notificaciones. Burbujas de chat Grabador de pantalla nativo Domótica en el menú de apagado Permiso por única vez Android Auto inalámbrico para todos
androide 12 (Cono de nieve)	12.0	octubre 2021	material usted Mejoras de privacidad Permiso de ubicación aproximada Indicadores de uso de micrófono, cámara y ubicación La domótica desaparece del menú de apagado Mejoras de rendimiento Capturas de pantalla con desplazamiento
Android 12L (Cono de nieve V2)	12.1	marzo 2022	Optimizado para tabletas, dispositivos plegables y computadoras barra de tareas inferior Mejoras en multitarea y múltiples ventanas Diseño de dos columnas

androide 13 (Tiramisu)	13.0	agosto 2022	Más Material Tu personalización Cambios de permiso Nuevos permisos de notificación Elige el idioma para cada aplicación Lector de QR Nuevo selector de fotos Aplicaciones activas Nuevo menú del portapapeles
Androide 14 (Pastel al revés)	14.0	octubre de 2023 en Pixel dispositivos	
Androide 15 (Helado de vainilla)	15.0	esperado 2024	

Ref: 1 - <https://www.xatakandroid.com/sistema-operativo/todas-versiones-android-historia> y Wikipedia

La pila de software de Android se compone de aplicaciones que se ejecutan en un marco Java de aplicaciones orientadas a objetos además de las bibliotecas centrales de Java. La máquina virtual Java sobre la que se ejecutan estas aplicaciones fue Dalvik hasta la versión 5.0, para cambiar en versiones posteriores al entorno Android Runtime (ART). La principal diferencia entre estas máquinas virtuales fue que Dalvik realizó la compilación en tiempo de ejecución, mientras que ART compila el código de bytes de Java durante la instalación de la aplicación.

Las bibliotecas utilizadas están escritas en lenguajes C/C++ e incluyen (entre otros): un administrador de interfaz gráfica (administrador de superficie), un marco OpenCore, una base de datos relacional SQLite, una interfaz de programación gráfica 3D OpenGL ES 2.0, un motor de renderizado WebKit, un Motor gráfico SGL, SSL y una biblioteca C estándar.



Ref: 2 - <https://developer.android.com/guide/platform>

El lenguaje para el desarrollo de Android ha sido tradicionalmente Java. Sin embargo, Google ha adoptado Kotlin como lenguaje de programación oficial de Android, que es un lenguaje más potente que genera código ejecutable directamente en la JVM.



Kotlin	Java
<pre> data class Person(var name: String, var age: Int)  /* var: read and write    val: read-only i.e. no setters */ </pre>	<pre> public class Person {     private String name;     private String age;      public Person(String name, String age) {         this.name = name;         this.age = age;     }      public String getName() {         return name;     }      public void setName(String name) {         this.name = name;     }      public String getAge() {         return age;     }      public void setAge(String age) {         this.age = age;     }      @Override     public boolean equals(Object o) {         /*         Code for equals function         */     }      @Override     public int hashCode() {         /*         Code for hashCode function         */     } } </pre>

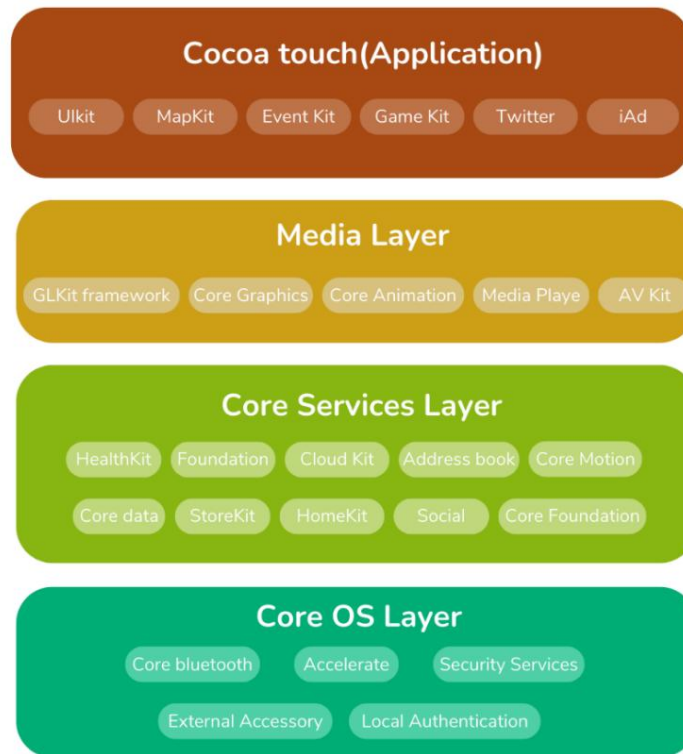
### 1.3. iOS

Android es el sistema operativo más vendido y después viene iOS, propiedad de Apple©. Fue creado para el iPhone y luego adoptado en el iPod touch y el iPad. iOS no permite la instalación en hardware de otras empresas.

iOS proporciona varios elementos de control, como controles deslizantes, palancas y botones, que brindan una respuesta inmediata a los comandos del usuario y una interfaz fluida. El usuario puede interactuar con el sistema operativo mediante gestos, deslizamientos, toques, pellizcos...

iOS se deriva de macOS y macOS se deriva de Darwin BSD, que es un sistema operativo similar a Unix.

La arquitectura iOS tiene cuatro capas de abstracción: la capa Core OS, la capa Core Services, la capa Media y la capa Cocoa Touch. El desarrollo de aplicaciones nativas para iOS implica lenguajes como Objective-C y Swift.



Ref: 3 - <https://redfoxsec.com/blog/ios-architecture/>

## 2. TIPOS DE APLICACIONES MÓVILES

### 2.1. APLICACIONES NATIVAS

Son aplicaciones que se desarrollan específicamente para el sistema operativo en el que se ejecutarán y, por tanto, aprovecharán mejor todos sus recursos. Además, permiten acceder a todas las funcionalidades de las plataformas y los usuarios tendrán las nuevas funciones disponibles desde el principio.

Para el desarrollo nativo en Android, se utilizan Java y Kotlin, mientras que Android también admite Objective-C y Swift (iOS), y no se necesitan pasos adicionales para llevar el código al código nativo (código de bytes).

Todo ello da como resultado aplicaciones fluidas que ofrecen la mejor experiencia de usuario. Como punto negativo tendremos un aumento en el coste de producción y mantenimiento.

### 2.2. APLICACIONES WEB RESPONSIVE (WebApps)

Se trata de aplicaciones basadas en tecnología web: HTML + CSS + JavaScript. Para ejecutarse sólo necesitan un navegador web. Ser responsivo significa que su interfaz se adapta a cualquier dispositivo.

Para este tipo de aplicaciones no es necesario desarrollar en código nativo. Son totalmente multiplataforma ya que se ejecutan en el navegador web del sistema operativo. La misma aplicación se puede ejecutar en Android, iOS, Windows, Linux o Mac.

Así, se dispone de un único código para ejecutar en todas las plataformas, con el inconveniente de que no ofrecerá tan buena experiencia al usuario como las apps nativas, sobre todo si se trata de una aplicación compleja.

### 2.3. APLICACIONES HÍBRIDAS

Las aplicaciones híbridas se refieren a aquellas que utilizan tecnología web para construir un sitio web (HTML + CSS + JS) y cargarlo en un WebView. Un WebView no es más que un navegador web sin barra de navegación ni otras opciones, por lo que parece una aplicación nativa en el dispositivo.

Además, las aplicaciones híbridas pueden acceder a algunas funciones del dispositivo, como ubicación, acelerómetros, etc.

Los frameworks más utilizados para este tipo de aplicaciones son Ionic y PhoneGap. Estos son dos marcos ampliamente utilizados por los desarrolladores de aplicaciones web para crear aplicaciones para dispositivos móviles. La idea detrás de estos marcos es simple: la aplicación web está empaquetada en código HTML + CSS + JS junto con WebView. WebView estará en código nativo para cada sistema, pero la aplicación web es la misma. La desventaja de estas tecnologías es que

reaccionan más lentamente ya que deben comunicarse con el sitio web para obtener gran parte de la contenido.

Estos marcos son comparables a Electron o NW.js (anteriormente conocido como node-webkit) en entornos de escritorio.

## 2.4. APLICACIONES WEB PROGRESIVAS

Un poco más cercanas a las aplicaciones nativas son las aplicaciones web progresivas (PWA), que son revolucionando el panorama actual. Estas aplicaciones aumentan las funcionalidades según el dispositivo móvil en el que se ejecutan, para extraerle más potencial (se adaptan progresivamente al dispositivo).

Estas aplicaciones aportan al usuario muchas de las ventajas de las aplicaciones nativas, pero con un desarrollo basado en tecnologías web.

A diferencia de las aplicaciones híbridas, permiten el funcionamiento con una mala conexión al servidor. Utilizando tecnologías como Service Workers, Cache API y Web Storage API, las PWA pueden almacenar información de aplicaciones a las que se accedió recientemente. De esa manera, cuando regresa a una página anterior, el usuario puede ver lo que ya estaba cargado en lugar de obtener la temida página "actualmente fuera de línea". Una vez conectadas nuevamente, las PWA pueden actualizar el contenido y administrar las solicitudes de red sin problemas.

Existen multitud de frameworks para el desarrollo de PWAs, entre los que se encuentran los principales frameworks de desarrollo web: React PWA Library, Angular PWA Framework, Vue PWA Framework, Ionic PWA Framework, Svelte, PWA Builder o Polymer.

	NATIVE	WEB	HYBRID	PROGRESSIVE
DESC	Built for a specific platform	Rely on a web browser and a working Internet connection to run	Combine the functionalities of native and web apps	Web apps designed to be faster, more lightweight, and borrow native app features
PROS	Faster speed Processing efficiency Smoother UI Hardware compatibility Access to the device's functionality	Cheaper to make No device memory or storage issues Easy maintenance Accessibility	Faster development Cross-platform compatibility Cost-effective Offline capability	No installation needed Data efficiency Versatility Automatic updates
CONS	Programming is not easy Takes time Multiple codebases for the same app	Browser dependent Useless without Internet Limited functionalities	Slower speed Limited hardware access Less smooth UI	Hardware integration issues Limited hardware access Browser UI issues

Ref: 4 - <https://www.bitstudios.com/blog/types-of-mobile-applications/>

## 2.5. APLICACIONES COMPILADAS

Se trata de tecnologías para desarrollar aplicaciones nativas más que un tipo de app móvil. Su objetivo es utilizar un solo lenguaje de programación para generar aplicaciones nativas. La idea general es trabajar con una única tecnología y lenguaje de programación y compilar el código para que sea nativo en las diferentes plataformas.

Algunas de las tecnologías más utilizadas en este tipo de aplicaciones son:

- React Native y Native Script: React es un framework creado por Facebook, que utiliza el lenguaje JavaScript y la biblioteca React y permite crear interfaces basadas en sus componentes. En estas aplicaciones, el código JavaScript se ejecuta en un hilo de ejecución separado, mientras que los elementos de la interfaz de usuario se compilan en lenguaje de máquina.

Por otro lado, Native Script permite crear aplicaciones nativas utilizando JavaScript puro o utilizando otras librerías como Angular o Vue. También viene con varios componentes prediseñados para interfaces de usuario. Al igual que React Native, no funcionan con HTML.

- Flutter: Es un framework desarrollado y mantenido por Google, que utiliza el lenguaje de programación Dart. El principio es que todo el código está escrito en Dart y compilado en código nativo que se ejecuta completamente en el dispositivo. Entonces, Flutter compila en ARM y genera bibliotecas C/C++, estando más cerca del código nativo y por lo tanto más rápido. La forma en que funciona Flutter es diseñar interfaces de usuario llamadas widgets. Flutter ya tiene varios widgets predeterminados, como botones, barras de navegación, etc.

### 3. IDIOMA KOTLIN

#### 3.1. INTRODUCCIÓN

Kotlin es un lenguaje de programación desarrollado por JetBrains, una empresa de software con sede en Rusia. Se anunció por primera vez en 2011 y se ha convertido en un lenguaje de programación popular en el mundo del desarrollo de software, especialmente en el desarrollo de aplicaciones móviles para Android.

Kotlin se caracteriza por ser un lenguaje de programación moderno y conciso que está diseñado para ser interoperable con Java, lo que significa que puedes usarlo en proyectos que ya están escritos en Java y viceversa. Algunas de las características clave de Kotlin incluyen:

- Seguridad: Kotlin incorpora características que ayudan a prevenir errores comunes en el código, como los relacionados con la seguridad nula. Esto reduce la probabilidad de errores de tiempo de ejecución relacionados con referencias nulas.

- Concisión: Kotlin le permite escribir código de forma más concisa en comparación con Java. Esto significa que puedes hacer más con menos líneas de código, lo que hace que el software sea más fácil de leer. •

Interoperabilidad: Kotlin puede interactuar perfectamente con el código y las bibliotecas Java existentes.

Esto significa que puedes migrar gradualmente un proyecto Java a Kotlin sin necesidad de reescribir todo el código desde cero.

- Programación funcional: Kotlin admite programación funcional, lo que significa que puede escribir más código declarativo y aprovechar conceptos como funciones de orden superior, lambdas y funciones de extensión.

- Orientado a objetos: Kotlin es un lenguaje orientado a objetos, lo que significa que puedes utilizar los conceptos de programación orientada a objetos de forma natural.

- Soporte multiplataforma: Kotlin también se puede utilizar para el desarrollo multiplataforma, lo que significa que puede escribir código que se ejecute en diferentes plataformas, como Android, iOS, web...

En resumen, Kotlin es un lenguaje de programación moderno y versátil que ha ganado popularidad en la comunidad de desarrollo de software debido a sus características y ventajas sobre otros lenguajes. Su uso principal es el desarrollo de aplicaciones para Android.

#### 3.1.1. INSTALACIÓN DE KOTLIN

Para instalar el lenguaje Kotlin seguiremos los siguientes pasos:

- Tenemos que instalar la versión deseada de java, en este caso openjdk-11-jdk

```
sudo apto instalar openjdk-11-jdk
```

- El siguiente paso es instalar Kotlin

```
instalación sudo snap --kotlin clásico
```

- Cree un editor de texto para crear un archivo llamado hello.kt

```
diversión principal() {  
  
println("¡Hola mundo!")  
  
}
```

- Compilar el archivo

```
kotlinc hola.kt -include-runtime -d hola.jar
```

- Ejecutarlo

```
java -jar hola.jar
```

También se puede comenzar a codificar y probar Kotlin utilizando un área de juegos de Kotlin:

- <https://play.kotlinlang.org>
- <https://developer.android.com/training/kotlinplayground>

### 3.1.2. VARIABLES Y TIPOS DE DATOS

En Kotlin los tipos de datos son clases, por lo que podemos acceder a sus propiedades y funciones miembro. Algunos de estos tipos, como números, caracteres o valores lógicos, pueden representarse internamente como valores primitivos en tiempo de ejecución, pero todo esto es transparente para el usuario. Recuerde que todos los tipos primitivos en Java también están disponibles en Kotlin.

Para definir valores en Kotlin usamos las palabras reservadas `var` o `val`:

- Usaremos `var` para definir variables mutables.
- Usaremos `val` para definir variables inmutables, o lo que en Java sería constante valores

En general, para tener mayor seguridad y rendimiento al trabajar en varios hilos de ejecución, se recomienda utilizar valores constantes si sabemos que no se modificarán.

Las siguientes asignaciones serían correctas:

```
valor pi = 3,14 // Constante
valor asunto = "PMDM"
var x = 1
x = x + 1
```

Si nos fijamos bien, vemos que no hemos definido el tipo de variables. Kotlin puede inferir el tipo de variables a partir de los valores con los que las inicializamos, por lo que no es necesario indicarlo explícitamente. Sólo será necesario indicar el tipo de una variable si no damos valor a la declaración. Para indicar el tipo haremos:

```
var nombreVariable: Tipo
```

---

## PLANTILLAS DE CUERDAS

Las plantillas de cadena o literales de cadena pueden contener expresiones de plantilla: fragmentos de código que se evaluarán y su resultado se concatenará en la cadena. Con esto podemos incluir valores, variables o incluso expresiones en una cadena.

Las expresiones de plantilla comienzan con el signo de dólar \$ y constan de un nombre de variable o una expresión entre las teclas { }.

```
temperatura del valor = 27

println("${temp}º ${ if (temp > 24) "Caliente" else "Frío" })"
```

---

## TIPOS NULLABLES Y OPERADOR ELVIS

Kotlin es un lenguaje seguro, y entre otras cosas nos evita errores de programación como NullPointerException ya que no permite que los valores de las variables sean nulos por defecto.

Si queremos especificar que una variable puede contener un valor nulo, es necesario definirla explícitamente como anulable. Para ello, cuando lo definimos, añadimos un signo de interrogación "?" a su tipo:

```
nombre de valor: ¿Cadena? = nulo
```

Además, Kotlin también nos proporciona el operador "?:" , conocido como operador de Elvis, para especificar un valor alternativo cuando la variable es nula.

```
nombre de valor: ¿Cadena? = nulo
println(nombre?.longitud?: -1)
```



### 3.1.3. OPERADORES

Ahora veremos los diferentes operadores que podemos utilizar en Kotlin, y su significado.

---

#### OPERADOR DE ASIGNACIÓN

Se utiliza para dar valor a una variable.

```
norte = 4
```

---

#### OPERADORES ARITMÉTICOS

Se utilizan para realizar operaciones aritméticas con variables. El resultado será un valor numérico. Pueden ser:

- + (operador de suma)
- - (operador de resta)
- % (operador de módulo)
- \* (operador de multiplicación)
- / (operador de división)

```
5 + 3
```

```
5++ // incremento
```

---

#### OPERADORES RELACIONALES

Se utilizan para hacer comparaciones entre variables y devolver un valor lógico.

- == igual que
- != diferente a
- < menos que

- > mayor que
- <= menor o igual
- >= mayor o igual

Además, con Kotlin podemos utilizar:

- === Es el mismo objeto.
- !== No es el mismo objeto

---

## OPERADORES LOGICOS

Se utilizan para realizar operaciones entre variables de tipo lógico. Su resultado también es de tipo lógico.

- ! Negación
- || O
- && Y

---

## OPERADOR CONDICIONAL TERNARIO

valor = condición? expresión\_1: expresión\_2

En Kotlin, una declaración condicional no es una declaración, sino una expresión, por lo que se puede asignar directamente a una variable.

### 3.1.4. ESTRUCTURA DE CONTROL

---

#### SI-ELSE

Esta estructura es la misma que la de las versiones de Java.

---

#### CUANDO

Switch no existe en Kotlin, la estructura más similar es el "Cuando". Puede expresarse como una declaración o como una expresión, lo cual veremos en el siguiente ejemplo:

```
cuando (eValor) {  
  
    valor1 -> si_valor1
```

```
valor2 -> if_valor2

...

valorN -> if_valorN

más -> _default

}

var x = cuando (valorexpr) {

    valor1 -> valor_para_1

    valor2 -> valor_para_2

    ...

    valorN -> valor_para_n

    más -> valor_predeterminado

}
```

También podemos usar el operador cuando sin argumentos (como si, entonces, si no) y con los operadores “es” y “en”

```
cuando {

    t < 15 -> println("FRÍO")

    t en 15..24 -> println("Aceptar")

    t > 25 -> println("CALIENTE")

}
```

```
cuando(mes) {

    en 1..3 -> println("invierno")

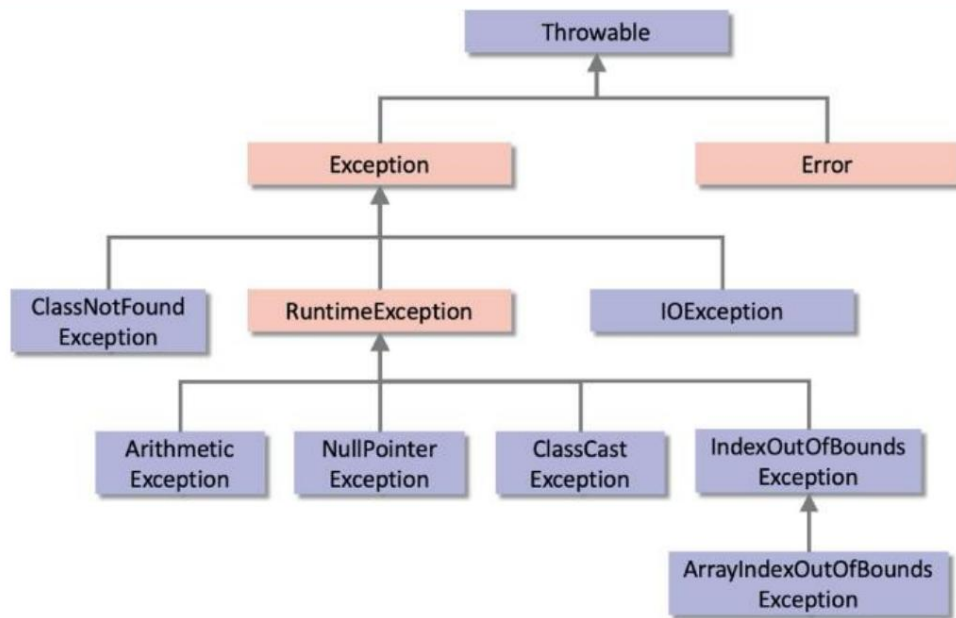
    en 4..6 -> println("primavera")

    en 7..9 -> println("verano")

    en 10..12 -> println("otoño")

}
```

## 3.1.5. EXCEPCIONES



Las excepciones en Kotlin son solo del tipo no revisado:

- `ArithmeticException` •
- `ArrayIndexOutOfBoundsException` •
- `NullPointerException`...

A nivel práctico, esto significa que los lanzamientos no deben incluirse en las declaraciones de funciones donde puedan ocurrir.

Esto no significa que no podamos manejar excepciones o lanzar excepciones en nuestro código.

```

intentar {

    // algún código

} captura (e: AlgunaExcepción) {

    // manipulador

} finalmente {

    // bloque finalmente opcional

}
  
```

```
1 fun foo() {  
2     try {  
3         throw Exception("Exception message")  
4     } catch (e: Exception) {  
5         println("Exception handled")  
6     } finally {  
7         println("inside finally block")  
8     }  
9 }
```

Ref: 5 - <https://code.tutsplus.com/es/kotlin-from-scratch-exception-handling--cms-29820t>

### 3.1.6. FUNCIONES

#### DEFINICIÓN E INVOCACIÓN

Para declarar una función en Kotlin hacemos:

```
divertido nombrefunción (param1: tipo1, parámetro2: tipo2...): tipo de retorno {  
  
    // cuerpo de la función  
  
    devolver  
  
}
```

Analizamos algunas características de las declaraciones de funciones:

- Se declaran usando la palabra clave fun
- Los nombres comienzan con minúsculas y se expresan en camelCase
- Los parámetros de función se especifican después del nombre, entre paréntesis y en el formato parámetro: Tipo. Estos tipos deben necesariamente especificarse.
- El tipo de retorno de la función se puede especificar después del paréntesis con el lista de argumentos, seguida de ":".
- Cuando la función no devuelve ningún valor significativo, su tipo de devolución predeterminado es Unidad, que sería el equivalente a void en Java o C.

#### FUNDAMENTOS DE LAS EXPRESIONES LAMBDA

Una expresión lambda representa el bloque de una función y simplifica el código.

Características de las expresiones lambda:

- Está expresado en { } • No tiene la palabra clave fun

- No tiene modificadores de acceso (privado, público o protegido) ya que no pertenece a ningún clase
- Es una función anónima, que no tiene nombre
- No especifica el tipo de retorno, como lo infiere el compilador.
- La última expresión de una lambda se considera el valor de retorno.
- Los parámetros no están entre paréntesis • Además, podemos asignar una expresión lambda a una variable y ejecutarla

En esta sección, veremos algunas formas de crear expresiones lambda:

- Expresión lambda sin parámetros y asignada a una variable

```
val msg = { println("¡Hola! Soy una función lambda") }  
  
mensaje()
```

- Expresión lambda con parámetros

```
val msg = {texto: Cadena -> println(texto) }  
  
msg("¡Hola Kotlin!")  
  
msg("¡Buenos días!")  
  
  
val writeSum = { s1: Int, s2: Int ->  
  
    println("Sumamos $s1 y $s2")  
  
    resultado valor = s1 + s2  
  
    println("La suma es: $resultado")  
  
}  
  
escribirSuma(3,2)
```

- Expresión lambda que omite parámetros: se usa cuando solo hay un parámetro

```
monedas val: (Int) -> Cadena = {cantidad ->
```

```
        "$cantidad cuartos"

    }

    println(monedas(3))    // 3 cuartos


    val monedas: (Int) -> String = { "$it quarters" }

    println(monedas(3))    // 3 cuartos
```

---

## FUNCIONES ANÓNIMAS

En Kotlin, se pueden crear funciones anónimas usando la palabra clave `fun` de manera similar a como se definen las funciones regulares, pero sin darles un nombre. Estas funciones se pueden asignar a variables o pasar como argumentos a otras funciones. A menudo se utilizan en Kotlin para implementar interfaces funcionales, como `Runnable` u `OnClickListener`. Aquí hay algunos ejemplos de funciones anónimas simples con argumentos en Kotlin:

```
val suma = diversión(x: Int, y: Int): Int {

    volver x + y

}

println(suma(5, 3))    // Imprime "8"


cálculo divertido (a: Int, b: Int, operación: (Int, Int) -> Int): Int {

    operación de retorno (a, b)

}

val suma = calcular(10, 5, diversión(x: Int, y: Int): Int {retorno x + y})

val diff = calcular(10, 5, diversión(x: Int, y: Int): Int {retorno x - y})

println("SUM: $suma")    // Imprime "SUM: 15"

println("DIFF: $diff ")    // Imprime "DIFF: 5"
```

### 3.1.7. POO

La programación orientada a objetos (POO) en Kotlin es una de las características clave del lenguaje, ya que gira en torno a objetos y clases para organizar y estructurar el código. Encuentre a continuación los conceptos fundamentales de la POO en Kotlin.

---

## CLASES Y OBJETOS

```
class Persona (nombre val: Cadena, edad val: Int) {  
  
    saludo divertido() {  
  
        println("Hola, soy $nombre y tengo $edad años.")  
  
    }  
  
}  
  
diversión principal() {  
  
    val persona1 = Persona("Juan", 30)  
  
    val persona2 = Persona("María", 25)  
  
    persona1.saludar()  
  
    persona2.saludar()  
  
}
```

En este ejemplo, hemos definido una clase Persona con dos propiedades (nombre y edad) y un saludo . método. Luego, creamos dos objetos de la clase Persona y llamamos al método greet en cada uno de ellos.

---

## HERENCIA

Kotlin admite la herencia, lo que significa que puedes crear una nueva clase basada en una clase existente. Las clases de Kotlin y sus funciones son finales de forma predeterminada (en Java están abiertas de forma predeterminada). Para permitir que una clase se extienda, debe estar marcada como abierta. Para permitir que se anulen funciones y campos de clase, también deben marcarse como abiertos.

A continuación se muestra un ejemplo de herencia:



```
clase abierta Animal (nombre val: Cadena) {

    abrir diversión makeSound() {

        println("$nombre emite un sonido.")

    }

}

clase Perro (nombre: Cadena): Animal (nombre) {

    anular divertido makeSound() {

        println("$nombre ladra.")

    }

}

clase Gato (nombre: Cadena): Animal (nombre) {

    anular divertido makeSound() {

        println("$nombre maúlla.")

    }

}

diversión principal() {

    val perro = Perro("Max")

    val gato = Gato("Bigotes")

    perro.makeSound()

    gato.makeSound()

}
```

En este ejemplo, definimos una clase base Animal con un método makeSound y luego creamos dos clases derivadas (Perro y Gato) que heredan de Animal. Cada clase derivada proporciona su propia implementación del método makeSound .

---

## ENCAPSULACIÓN

Kotlin le permite controlar el acceso a las propiedades y métodos de una clase utilizando modificadores de acceso como privado, protegido, interno (módulo) y público (predeterminado). Esto facilita la implementación del principio de encapsulación (<https://kotlinlang.org/docs/visibility-modifiers.html>)

```
cuenta bancaria de clase (saldo var privado: doble)
{ depósito divertido (monto: doble) {
    si (monto > 0) {
        saldo += monto
    }
}

retiro divertido (monto: doble) {
    if (monto > 0 && saldo >= monto) {
        saldo -= cantidad
    }
}

divertido getBalance() : Doble
{ devolver saldo
}
}

fun main()
{ val cuenta = BankAccount(1000.0)
  account.deposit(500.0)
  account.withdraw(200.0)
  println("Saldo actual: ${account.getBalance()}")
}
```

En este ejemplo, la propiedad del saldo se declara como privada, lo que significa que solo se puede acceder a ella desde la clase BankAccount . Los métodos de depósito, retiro y getBalance se utilizan para interactuar con la propiedad del saldo.

Estos son los conceptos básicos de la Programación Orientada a Objetos en Kotlin. Puedes crear clases, objetos, aplicar herencia y controlar el acceso a propiedades y métodos para construir. Código modular y mantenible.

---

## USO DE GENÉRICOS

Kotlin permite el uso de parámetros de tipo en la definición de clases.

Definición:

```
class class name < generic data type > (  
    val property name : generic data type  
)  
class Question<T>(  
    val questionText: String,  
    val answer: T,  
    val difficulty: String  
)
```

Usar:

```
val instance name = class name < generic data type > ( parameters )  
  
fun main() {  
    val q1 = Question<String>("Capital of China is __", "Beijing", "medium")  
    val q2 = Question<Boolean>("The sky is green. True or false", false, "easy")  
    val q3 = Question<Int>("How many days are in July?", 31, "easy")  
}
```

Referencia: 6 - <https://kotlinlang.org/docs/generics.html>

---

## CLASES DE NUMERO

Las clases de enumeración se utilizan para evitar que los programadores y usuarios escriban incorrectamente. Usando enumeración

Las clases obligan a que un conjunto determinado de valores sean los únicos aceptados (tipo seguro).

Definición:	Usar:
<pre>enum class <b>enum name</b> {     <b>Case 1</b> , <b>Case 2</b> , <b>Case 3</b> }</pre>	<pre><b>enum name</b> . <b>case name</b></pre>

```
class Question <T> {  
    val questionText : String,  
    val answer : T,  
    val difficulty: Difficulty  
}  
  
enum class Difficulty{  
    EASY, MEDIUM, HARD  
}  
  
fun main() {  
    val q3 = Question<Int>("Fingers in a hand?", 5, Difficulty.EASY)  
}
```

Ref: 7 - <https://kotlinlang.org/docs/enum-classes.html>

---

## OBJETOS UNICOS

Hay muchos escenarios en los que desea que una clase solo tenga una instancia. Por ejemplo:

- Estadísticas del jugador en un juego móvil para el usuario
- actual • Interactuar con un solo dispositivo de hardware, como enviar audio a través de un altavoz
- Autenticación, donde solo un usuario debe iniciar sesión a la vez

En los escenarios anteriores, probablemente necesitarás usar una clase. Sin embargo, sólo necesitarás crear una instancia de esa clase --> Objeto Singleton

Un objeto singleton no puede tener un constructor ya que no se pueden crear instancias directamente. En cambio, todas las propiedades se definen entre llaves y se les asigna un valor inicial.

```
object object name {  


class body 1

  
}
```

```

data class Question<T>{
    val questionText: String,
    val answer: T,
    val difficulty: Difficulty
}
enum class Difficulty {
    EASY, MEDIUM, HARD
}
object StudentProgress {
    var total: Int = 10
    var answered: Int = 3
}

fun main() {
    println("${StudentProgress.answered} of ${StudentProgress.total} answered")
    val q3 = Question<Int>("Fingers in a hand?", 5, Difficulty.EASY)
    StudentProgress.answered = StudentProgress.answered + 1
    println("${StudentProgress.answered} of ${StudentProgress.total} answered")
}

```

Ref: 8 - <https://kotlinlang.org/docs/object-declarations.html#object-declarations-overview>

### 3.1.8. TIPOS DE DATOS COMPLEJOS

En Kotlin, los tipos de datos complejos a menudo se implementan mediante clases, clases de datos o colecciones (como listas, conjuntos y mapas) para representar datos estructurados. A continuación se muestran algunos ejemplos de tipos de datos complejos en Kotlin:

---

#### CLASES Y CLASES DE DATOS

Las clases se utilizan para crear tipos de datos complejos personalizados con propiedades y métodos. Las clases de datos, por otro lado, son un tipo especial de clase diseñada principalmente para almacenar datos. Proporcionan automáticamente funciones útiles como `toString()`, `equals()` y `hashCode()` según sus propiedades. Una clase de datos necesita tener al menos un parámetro en su constructor.

```

clase de datos Persona (nombre val: Cadena, edad val: Int)

val persona = Persona("Alicia", 30)

println(persona)           // Salida: Persona(nombre=Alicia, edad=30)

```

---

## LISTA / LISTAMUTABLE

Las listas son colecciones ordenadas que pueden almacenar elementos del mismo o de diferentes tipos. Puede crear listas de tipos de datos complejos. List es una interfaz que define propiedades y métodos relacionados con una colección ordenada de elementos de solo lectura. MutableList amplía la interfaz List definiendo métodos para modificar una lista, como agregar y eliminar elementos.

```
val frutas = listaDe("Manzana", "Plátano", "Cereza")

val personas = listaDe(Persona("Alicia", 30), Persona("Bob", 25))

val solarSystem = listOf("Mercurio", "Venus", "Tierra", "Marte")

println(sistemasolar.tamaño)

println(sistemasolar[2])

println(sistemasolar.get(2))

println(sistemasolar.indexOf("Tierra"))


val solarSystem = mutableListOf("Mercurio", "Venus", "Tierra", "Marte")

solarSystem.add("Plutón")

solarSystem.add(3, "Theia")

solarSystem[3] = "Luna futura"

solarSystem.removeAt(9)

solarSystem.remove(" Luna del futuro")

println(sistemasolar.contiene("Plutón"))

println("Luna del Futuro" en el Sistema solar)
```

Las listas son colecciones fáciles de iterar mediante un bucle for .

```
for ( element name in collection name ) {  
    body  
}  
  
for (planet in solarSystem) {  
    println(planet)  
}
```

<https://kotlinlang.org/docs/collections-overview.html>

---

#### CONJUNTO / CONJUNTO MUTABLE

Un conjunto es una colección que no tiene un orden específico y no permite valores duplicados.

El secreto es un código hash. Un código hash es un Int producido por el método hashCode() de cualquier clase Kotlin. Un pequeño cambio en el objeto, como agregar un carácter a una cadena, da como resultado un valor hash muy diferente.

La búsqueda de un elemento específico en un conjunto es rápida (en comparación con las listas), especialmente para colecciones grandes, pero los conjuntos tienden a utilizar más memoria que las listas para la misma cantidad de datos.

```
val UniquePeople = setOf(Persona("Alicia", 30), Persona("Bob", 25), Persona("Alicia", 30))  
  
println(personasÚnicas)           // [Persona(nombre=Alice, edad=30), Persona(nombre=Bob, edad=25)]  
  
val solarSystem = mutableSetOf("Mercurio", "Venus", "Tierra", "Marte")  
  
println(sistemasolar.tamaño)  
  
solarSystem.add("Plutón")  
  
println(sistemasolar.contiene("Plutón"))           // "Plutón" en solarSystem es equivalente  
  
solarSystem.remove("Plutón")
```

---

#### MAPA / MUTABLEMAP

Los mapas asocian claves con valores, lo que le permite crear estructuras de datos complejas para representar relaciones o configuraciones. Se llama mapa porque las claves únicas se asignan a otras valores.

```
val map name = mapOf(  
    key to value ,  
    key to value ,  
    key to value ,  
)  
mutableMapOf<key type , value type> ( )
```

```
val edades = mutableMapOf("Alice" a 30, "Bob" a 25)  
  
println(edades) // {Alicia=30, Bob=25}  
  
println(edades.get("Bob"))  
  
edades.remove("Alicia")  
  
val peopleMap = mapOf(1 a Persona("Alice", 30), 2 a Persona("Bob", 25))  
  
println(peopleMap) // {1=Persona(nombre=Alice, edad=30), 2=Persona(nombre=Bob, edad=25)}
```

---

## Matrices

Los arreglos son colecciones de tamaño fijo que pueden almacenar elementos del mismo tipo. El tipo de datos es opcional como se puede inferir.

```
val variable name = arrayOf<data type> ( element1 , element2 , ... )
```

```
números de valor = matriz de (1, 2, 3, 4, 5)
```

---

## FUNCIONES DE ALTO ORDEN CON COLECCIONES

Una función de orden superior es una función que toma funciones como parámetros o devuelve una función.

### PARA CADA()

La función `forEach()` se puede combinar con plantillas de cadena y lambdas para iterar a lo largo de una colección y realizar acciones en cada elemento de la colección.



```

class Cookie(
    val name: String,
    val softBaked: Boolean,
    val hasFilling: Boolean,
    val price: Double
)

val cookies = listOf(
    Cookie(
        name = "Chocolate Chip",
        softBaked = false,
        hasFilling = false,
        price = 1.69
    ),
    Cookie(
        name = "Banana Walnut",
        softBaked = true,
        hasFilling = false,
        price = 1.49
    ), ...
)

```

```

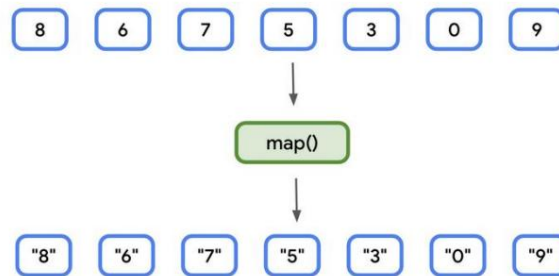
fun main() {
    cookies.forEach {
        println("Menu item: ${it.name}")
    }
}

```

Ref: 9 - <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/for-each.html>

## MAPA()

La función map() le permite transformar una colección en una nueva colección con la misma cantidad de elementos mientras agrega alguna transformación.



Ref: 10 - <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map.html>

```

fun main() {
    val fullMenu = cookies.map {
        "${it.name} - ${it.price}"
    }
    println("Full menu:")
    fullMenu.forEach {
        println(it)
    }
}

```

Full menu:

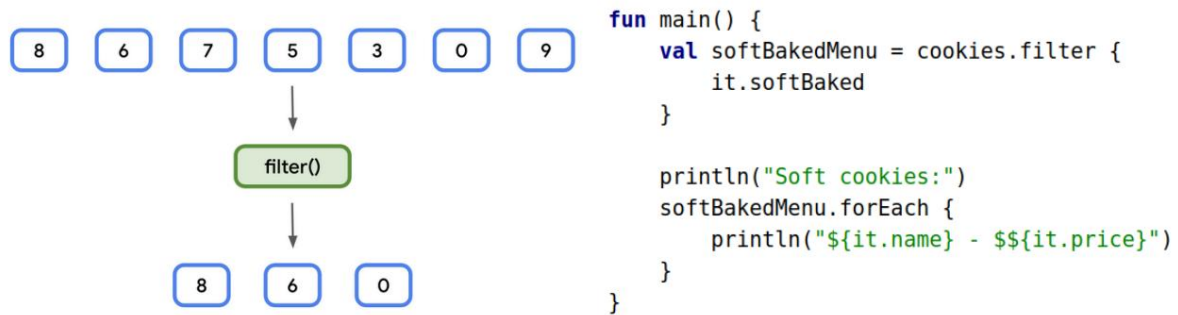
```

Chocolate Chip - $1.69
Banana Walnut - $1.49
Vanilla Creme - $1.59

```

## FILTRAR()

La función `filter()` te permite crear un subconjunto de una colección. La lambda tiene un único parámetro que representa cada elemento de la colección y devuelve un valor booleano.



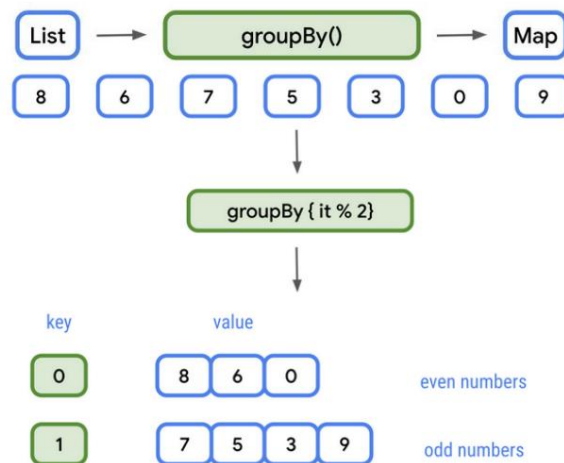
Ref: 11 - <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter.html>

Para cada artículo de la colección:

- Si el resultado de la expresión lambda es verdadero, entonces el elemento se incluye en la nueva recopilación.
- Si el resultado es falso, el artículo no se incluye en la nueva colección.

## AGRUPAR POR()

La función `groupBy()` se puede utilizar para convertir una lista en un mapa, según una función. Cada valor de retorno único de la función se convierte en una clave en el mapa resultante. Los valores de cada clave son todos los elementos de la colección que produjeron ese valor de retorno único.



Ref: 12 - <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/group-by.html>

```
fun main() {  
    val groupedMenu = cookies.groupBy {it.softBaked}  
    val softBakedMenu = groupedMenu[true] ?: emptyList()  
    val crunchyMenu = groupedMenu[false] ?: emptyList()  
  
    println("Soft cookies:")  
    softBakedMenu.forEach {  
        println("${it.name} - ${it.price}")  
    }  
    println("Crunchy cookies:")  
    crunchyMenu.forEach {  
        println("${it.name} - ${it.price}")  
    }  
}
```

#### DOBLAR()

Se utiliza para generar un valor único a partir de una colección.

La función fold() toma dos parámetros:

- Un valor inicial
- Una expresión lambda que devuelve un valor del mismo tipo que el valor inicial

La expresión lambda tiene además dos parámetros:

- Acumulador: cada vez que se llama a la expresión lambda, el acumulador es igual al valor de retorno de la vez anterior que se llamó a la expresión lambda.
- El segundo es del mismo tipo que cada elemento de la colección.

Initial value of accumulator

```
val totalPrice = cookies.fold(0.0) {total, cookie ->  
    total + cookie.price  
}  
println("Total price: ${totalPrice}")
```

Accumulator

```
total = total + cookie.price  
return total
```

Inferred

Ref: 13 - <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/fold.html>

## ORDENADO POR()

Le permite especificar una lambda que devuelve la propiedad por la que desea ordenar.

En la medida en que el tipo de datos del valor tenga un orden de clasificación natural (las cadenas se ordenan alfabéticamente, los valores numéricos se ordenan en orden ascendente), se ordenará como una colección de ese tipo.

```
val alphabeticalMenu = cookies.sortedBy {  
    it.name  
}  
println("Alphabetical menu:")  
alphabeticalMenu.forEach {  
    println(it.name)  
}
```

Ref: 14 - <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted-by.html>

Estos ejemplos demuestran varias formas en que puede trabajar con tipos de datos complejos en Kotlin.

Dependiendo de las necesidades de su aplicación, puede usar clases, clases de datos, listas, conjuntos, mapas o matrices para representar y manipular datos estructurados de manera efectiva.