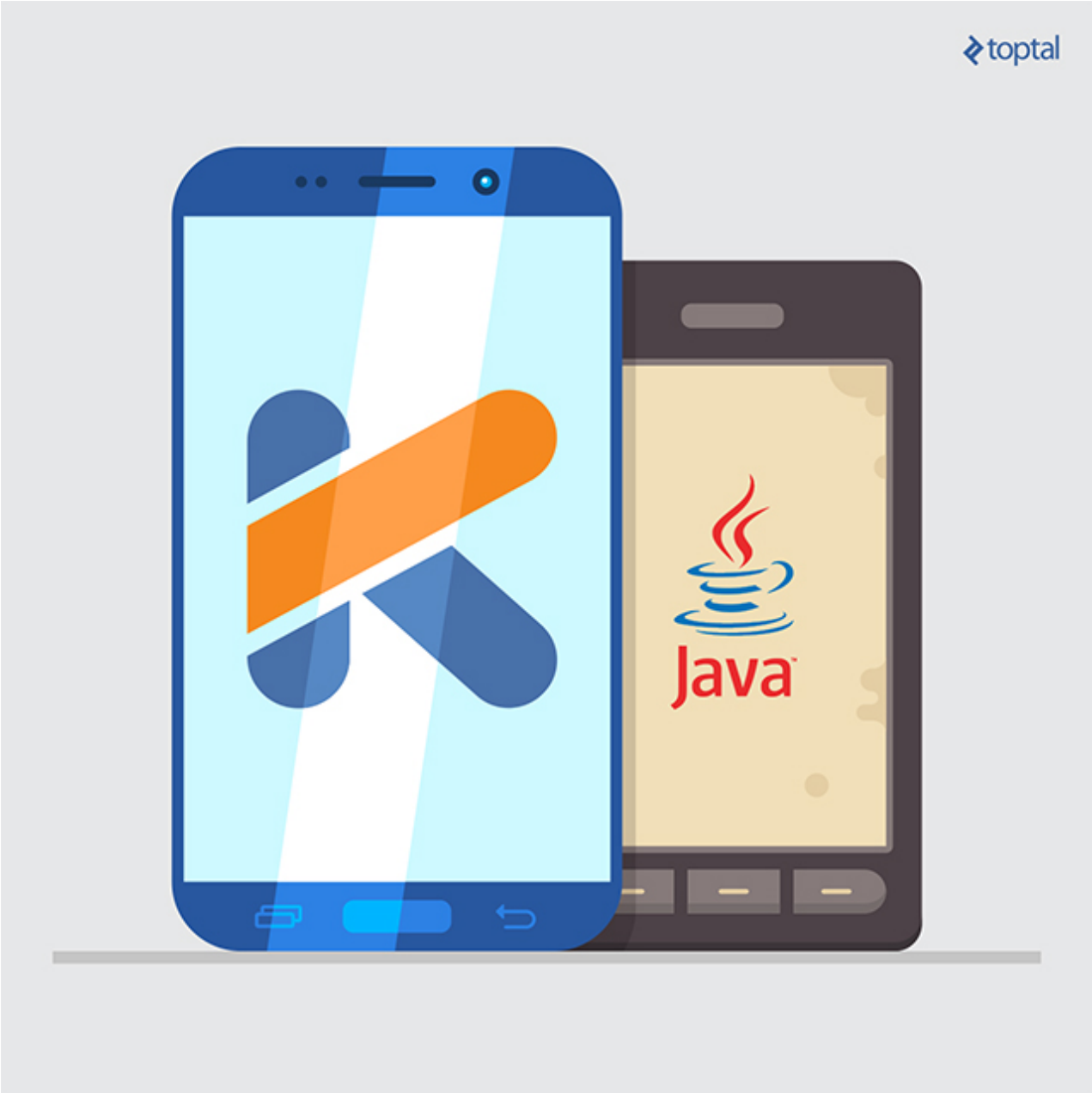
Introducción a Kotlin: Programación de Android Para Seres Humanos

View all articles

by **Tomasz Czura** - Freelance Software Engineer @ Toptal (translated by Isabella Rolz)

#Android #JVM #Kotlin #ProgrammingLanguage

En el perfecto mundo de Android, el idioma principal de Java es realmente moderno, claro y elegante, puedes escribir menos haciendo más y cada vez que aparezca una nueva función, los desarrolladores pueden usarla aumentando la versión en **Gradle**. Al crear una aplicación agradable, la función parece totalmente comprobable, extensible y mantenible. Nuestras actividades no son demasiado grandes y complicadas, podemos cambiar las fuentes de datos de la base de datos a la web sin demasiadas diferencias y así sucesivamente. Suena bien, ¿verdad? Desafortunadamente, en el mundo de Android no es este ideal. Google sigue luchando por la perfección, pero todos sabemos que los mundos ideales no existen. Por lo tanto, tenemos que ayudarnos a nosotros mismos en ese gran viaje en el mundo de Android.



Kotlin es un jugador importante en el mundo de Android. Pero, ¿podrá reemplazar a Java?

¿Qué es Kotlin, y Por Qué Deberías Usarlo?

Así que, el primer idioma. Creo que Java no es el maestro de la elegancia o la claridad y no es ni moderno ni expresivo (y supongo que estás de acuerdo). La desventaja es que por debajo de Android N, todavía estamos limitados a Java 6 (incluyendo algunas pequeñas partes de Java 7). Los desarrolladores también pueden adjuntar **RetroLambda** para utilizar expresiones lambda en el código, lo cual es muy útil al usar **RxJava**. Por encima de Android N, podemos usar algunas de las nuevas funcionalidades de Java 8, pero sigue siendo el viejo y pesado Java. Muy a menudo oigo a [desarrolladores de Android] (https://www.toptal.com/android) decir “Me gustaría que Android apoyara un lenguaje más agradable, como el iOS lo hace con Swift”. ¿Y si te dijera que

puedes usar un lenguaje muy agradable y sencillo, con seguridad nula, lambdas, y muchas otras nuevas características? Bienvenido a Kotlin.

¿Qué es Kotlin?

Kotlin es un nuevo idioma (a veces conocido como Swift para Android), desarrollado por el equipo de **JetBrains** y ahora está en su versión 1.0.2. Lo que lo hace útil en el desarrollo de Android es que compila a bytecode JVM, y también se puede compilar con JavaScript. Es totalmente compatible con Java y el código de Kotlin puede ser simplemente convertido a código Java y viceversa (hay un **plugin** de JetBrains). Esto significa que Kotlin puede usar cualquier marco, biblioteca, etc., escrito en Java. En Android, se integra por Gradle. Si tienes una aplicación Android existente y quieres implementar una nueva función en Kotlin sin volver a escribir la aplicación completa, solo empieza a escribir en Kotlin y funcionará.

Pero, ¿cuáles son las "grandes nuevas características"? Permíteme enumerar algunas:

Parámetros de Funciones Denominadas y Opcionales

```
Fecha de crear Diversión(day: Int, month: Int, year: Int, hour: Int
  print("TEST", "$day-$month-$year $hour:$minute:$second")
}
```

Podemos Llamar Al Método Fechacreación De Distintas Maneras

```
Fechacreación(1,2,2016) prints:  '1-2-2016 0:0:0'
Fechacreación(1,2,2016, 12) prints: '1-2-2016 12:0:0'
Fechacreación(1,2,2016, minute = 30) prints: '1-2-2016 0:30:0'
```

Seguridad Nula

Si una variable puede ser nula, el código no compilará a menos que los obliguemos a hacerlo. El siguiente código tendrá un error: **nullable Var** puede ser nulo:

```
var nullableVar: String? = "";
nullableVar.length;
```

Para compilar, debemos de verificar si es nulo o no:

```
if(nullableVar){
                nullableVar.length
}
```

O inclusive más corto:

```
nullableVar?.length
```

De esta manera, si el nullableVar es nulo, no pasa nada. De lo contrario, podemos marcar la variable como no anulable, sin un signo de interrogación después de

tipo:

```
var nonNullableVar: String = “”;
nonNullableVar.length;
```

Este código se compila y si queremos asignar nulo al no-NullableVar, el compilador mostrará un error.

También es muy útil el operador de Elvis:

```
var stringLength = nullableVar?.length ?: 0
```

Entonces, cuando nullableVar es nulo (así nullableVar?.length resulta en nulo), stringLength tendrá valor 0.

Variables Mutables e Inmutables

En el ejemplo anterior, utilizo **var** cuando se define una variable. Esto es mutable, podemos reasignarlo cuando queramos. Si queremos que esta variable sea inmutable (en muchos casos deberíamos), usamos **val** (como valor, no variable):

```
val immutable: Int = 1
```

Después de eso, el compilador no nos permitirá reasignar a inmutable.

Lambdas

Todos sabemos lo que es un lambda, así que aquí voy a mostrar cómo podemos usarlo en Kotlin:

```
button.setOnClickListener({ view -> Log.d("Kotlin","Click")})
```

O si la función es el único o el último argumento:

```
button.setOnClickListener { Log.d("Kotlin","Click")}
```

Extenciones

Las extensiones son una característica de lenguaje muy útil, gracias a la cual podemos “extender” las clases existentes, incluso cuando son finales o no tenemos acceso a su código fuente.

Por ejemplo, para obtener un valor de cadena de texto de edición, en lugar de escribir cada vez editText.text.toString () podemos escribir la función:

```
fun EditText.textValue(): String{
  return text.toString()

}
```

O inclusive más corta:

```
fun EditText.textValue() = text.toString()
```

Y ahora, con cada instancia de **EditText**:

```
editText.textValue()
```

O, podemos agregar una propiedad que devuelve lo mismo:

```
var EditText.textValue: String
  get() = text.toString()
  set(v) {setText(v)}
```

Sobrecarga del Operador

A veces es útil si queremos agregar, multiplicar o comparar objetos. Kotlin permite la sobrecarga de: operadores binarios (más, menos, plusAssign, rango, etc.), operadores de matriz (get, set, get range, set range) y de operaciones iguales y de unarios (+ a, -a, etc.).

Clase de Datos

¿Cuántas líneas de código necesita para implementar una clase de usuario en Java con tres propiedades: copiar, igual, **hashCode** y **toString**? En Kotlin sólo necesitas una línea:

```
Usuario de Clase de Datos(nombre del val: String, apellido del val:
```

Esta clase de datos proporciona métodos iguales (), hashCode () y copiar (), y también **toString** (), que imprime al Usuario como:

```
Usuario(nombre=John, apellido=Doe, edad=23)
```

Las clases de datos también proporcionan algunas otras funciones y propiedades útiles que se pueden ver en la documentación de Kotlin.

Extensiones de Anko

Utiliza **Butterknife** o extensiones de Android, ¿no? ¿Qué pasa si no necesitas ni siquiera usar esta biblioteca y después de declarar las vistas en XML simplemente lo usas desde el código por su ID (como con XAML en C #):

```
<Button
       android:id="@+id/loginBtn"
       android:layout_width="match_parent"
       android:layout_height="wrap_content" />
```

```
loginBtn.setOnClickListener{}
```

Kotlin tiene muy útiles [extensiones de Anko] (https://github.com/Kotlin/anko) y con esto no necesitas decirle a su actividad lo que es **loginBtn**, lo sabe simplemente “importando” xml:

```
import kotlinx.android.synthetic.main.activity_main.*
```

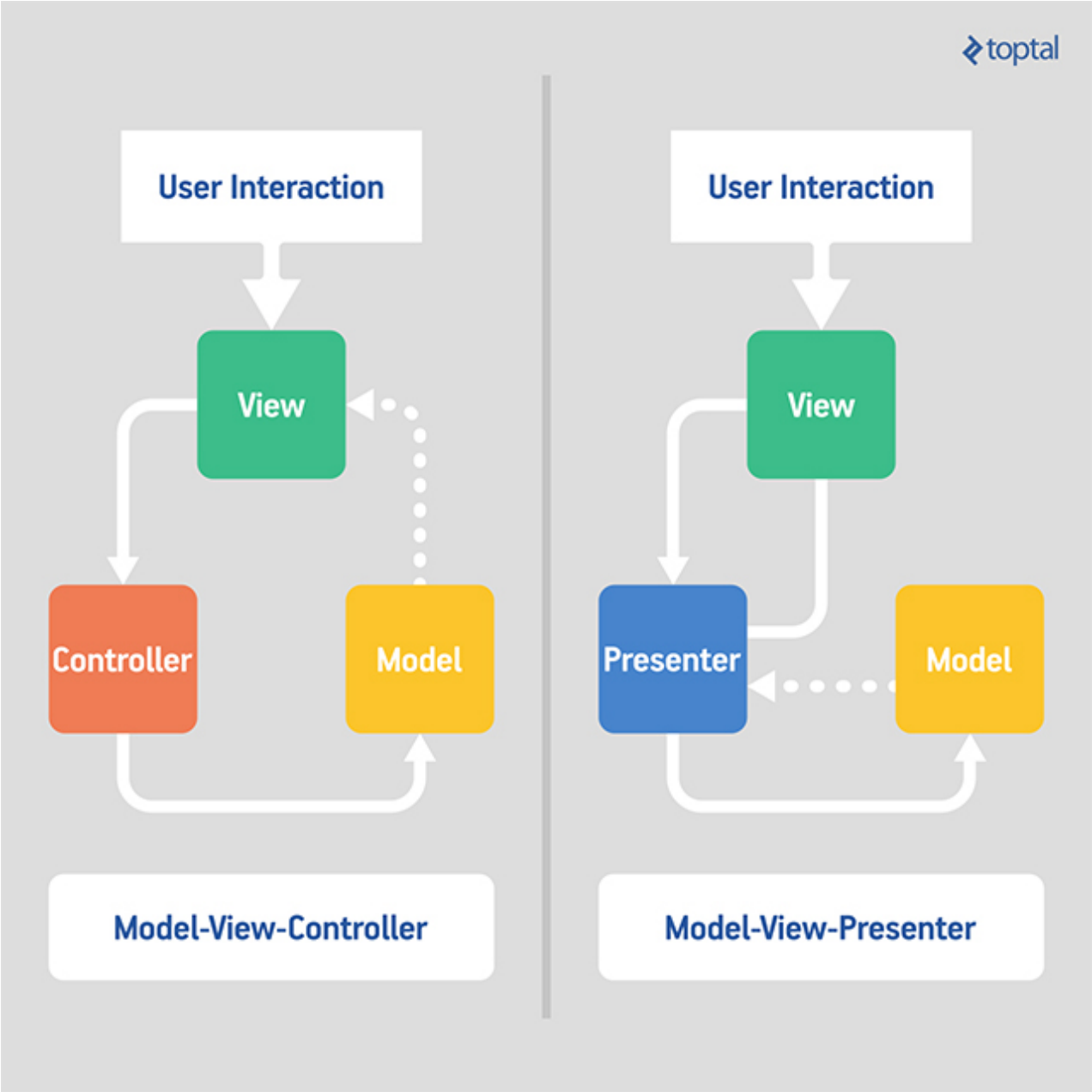
Hay muchas otras cosas útiles en Anko, incluyendo actividades de inicio, actividades medias y así sucesivamente. Este no es el objetivo principal de Anko - está diseñado para crear diseños de código fácilmente. Así que si necesitas crear un diseño de programación, esta es la mejor manera.

Esta es sólo una visión corta de Kotlin. Recomiendo leer el blog de Antonio Leiva y su libro - **Kotlin for Android Developers**, y por supuesto el sitio oficial de Kotlin.

¿Qué Es MVP y Por Qué?

Un lenguaje agradable, potente y claro no es suficiente. Es muy fácil escribir aplicaciones desordenadas con todos los idiomas sin una buena arquitectura. Los desarrolladores de Android (principalmente los que están empezando, pero también los más avanzados) a menudo dan a la Actividad la responsabilidad de todo lo que les rodea. La actividad (o Fragmento, u otra vista) descarga datos, guarda para enviar, los presenta, responde a interacciones de usuarios, edita datos y administra todas las vistas de niños. . . Y muchas veces mucho más. Es demasiado para objetos tan inestables como Actividades o Fragmentos (basta con rotar la pantalla y la Actividad dice 'Adiós …').

Una muy buena idea es aislar las responsabilidades de las opiniones y hacerlas tan estúpidas como sea posible. Las vistas (actividades, fragmentos, vistas personalizadas o lo que sea que presente datos en la pantalla) deberían ser responsables únicamente de la gestión de sus subvistas. Las vistas deben tener presentadores, que se comuniquen con el modelo y decirles lo que deben hacer. Esto en resumen, es el patrón **Model-View-Presenter** (para mí, debe ser nombrado **Model-Presenter-View** para mostrar las conexiones entre capas).



“¡Hey, yo sé de algo así, y se llama MVC!” - ¿no crees? No, MVP no es lo mismo que MVC. En el patrón MVC, tu vista puede comunicarse con el modelo. Mientras utilizas MVP, no permites ninguna comunicación entre estas dos capas, la única forma en que **View** puede comunicarse con **Model** es a través de **Presenter**. Lo único que **View** sabe sobre **Model** puede ser la estructura de datos. **View** sabe cómo, por ejemplo, mostrar al **Usuario**, pero no sabe cuándo. Aquí hay un ejemplo sencillo:

La vista sabe que “soy actividad, tengo dos **EditTexts** y un botón. Cuando alguien hace clic en el botón, debo decírselo a mi presentador, y le paso los valores de **EditTexts**. Y eso es todo, puedo dormir hasta el siguiente clic o el presentador me dice qué hacer”.

Presenter sabe que en algún lugar es una vista y sabe qué operaciones puede realizar esta vista. También sabe que cuando recibe dos cadenas, debe crear el usuario a partir de estas dos cadenas y enviar datos al modelo para guardar, y si se guarda con éxito, decirle a la vista “Muestra información de éxito”.

El modelo sólo sabe dónde están los datos, dónde deben guardarse y qué operaciones deben realizarse en los datos.

Las aplicaciones escritas en MVP son fáciles de probar, mantener y reutilizar. Un presentador puro no debe saber nada sobre la plataforma Android. Debe ser puro clase de (o Kotlin, en nuestro caso) Java. Gracias a esto podemos reutilizar a nuestro presentador en otros proyectos. También podemos escribir fácilmente pruebas de unidad, probando por separado **Model**, **View** y **Presenter**.

El patrón MVP conduce a una base de código mejor y menos compleja al mantener la interfaz de usuario y la lógica de negocio verdaderamente separadas..

Una pequeña digresión: el MVP debe ser una parte de **Uncle Bob's Clean Architecture** para hacer aplicaciones aún más flexibles y bien diseñadas. Voy a tratar de escribir sobre eso la próxima vez.

Aplicación De Ejemplo Con MVP Y Kotlin

Esto es suficiente teoría, ¡vamos a ver algunos códigos! Bueno, intentemos crear una aplicación sencilla. El objetivo principal de esta aplicación es crear un usuario. La primera pantalla tendrá dos EditTexts (nombre y apellido) y un botón (Save). Después de introducir el nombre y el apellido y hacer clic en "Guardar", la aplicación debe mostrar "Usuario se guarda" y pasar a la siguiente pantalla, donde se muestra el nombre y apellido guardados. Cuando el nombre o apellido está vacío, la aplicación no debe guardar al usuario y entonces muestra un error que indica lo que está mal.

Lo primero que se debe hacer después de crear el proyecto Android Studio es configurar Kotlin. Deberías instalar el complemento Kotlin y después de reiniciar, en Herramientas > Kotlin puedes hacer clic en 'Configurar Kotlin en Proyecto'. IDE agregará dependencias de Kotlin a Gradle. Si tienes algún código existente, puedes convertirlo fácilmente en Kotlin (Ctrl + Mayús + Alt + K o Codificar & gt; Convertir archivo de Java en Kotlin). Si algo no funciona y el proyecto no se compila, o Gradle no ve Kotlin, puedes comprobar el código de la aplicación disponible en GitHub.

Kotlin no sólo interopera bien con los marcos de Java y las bibliotecas, sino que le permite seguir utilizando la mayoría de las mismas herramientas con lasque ya estás familiarizado.

Ahora que tenemos un proyecto, comencemos creando nuestra primera vista: **Create User View**. Esta vista debe tener las funcionalidades mencionadas anteriormente, por lo que podemos escribir una interfaz para eso:

```
interface CreateUserView
: View {
  fun
showEmptyNameError() /* show error when name is empty */
  fun
showEmptySurnameError() /* show error when surname is empty */
  fun
showUserSaved() /* show user saved info */
  fun
```

```
showUserDetails(user: User) /* show user details */
}
```

Como se puede ver, Kotlin es similar a Java en declarar funciones. Todas son funciones que no devuelven nada y las últimas tienen un parámetro. Esta es la diferencia, el tipo de parámetro viene después del nombre. La interfaz de View no es de Android: es nuestra interfaz simple y vacía:

```
interface View
```

La interfaz de **Presenter** básica debe tener una propiedad del tipo vista, y al menos en el método (onDestroy por ejemplo), donde esta propiedad se establecerá como nula:

```
interface Presenter<T : View> {
   var view: T?

   fun onDestroy(){
       view = null
   }
}
```

Aquí puede ver otra característica de Kotlin: puedes declarar propiedades en interfaces y también implementar métodos allí.

Nuestro **CreateUserView** necesita comunicarse con **CreateUserPresenter**. La única función adicional que necesita este presentador es **saveUser** con dos argumentos de cadena:

```
interface CreateUserPresenter<T : View>: Presenter<T> {
   fun saveUser(name: String, surname: String)
}
```

También necesitamos una definición de Modelo – es mencionado con anterioridad en el tipo de clase:

```
data class User(val name: String, val surname: String)
```

Después de declarar todas las interfaces, podemos empezar a implementar.

CreateUserPresenter será implementado en **CreateUserPresenterImpl**:

```
class
CreateUserPresenterImpl(override var view: CreateUserView?):
CreateUserPresenter<CreateUserView> {

  override fun
saveUser(name: String, surname: String) {
  }
}
```

La primera línea, con definición de clase:

```
CreateUserPresenterImpl(override
var view: CreateUserView?)
```

Es un constructor, lo utilizamos para asignar la propiedad de vista, definida en la interfaz.

MainActivity, which is our **CreateUserView** implementation, needs a reference to **CreateUserPresenter**:

```
class MainActivity :
AppCompatActivity(), CreateUserView {

  private val
presenter: CreateUserPresenter<CreateUserView> by lazy {
      CreateUserPresenterImpl(this)
  }

  override fun
onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)

      saveUserBtn.setOnClickListener{
          presenter.saveUser(userName.textValue(),
userSurname.textValue()) /*use of textValue() extension, mentioned e

      }
  }



  override fun
showEmptyNameError() {
      userName.error
= getString(R.string.name_empty_error) /* it's equal to userName.set
Kotlin allows us to use property */

  }

  override fun
showEmptySurnameError() {
      userSurname.error
= getString(R.string.surname_empty_error)
  }

  override fun
showUserSaved() {
      toast(R.string.user_saved)
/* anko extension - equal to Toast.makeText(this, R.string.user_save
Toast.LENGTH_LONG) */

  }

  override fun
showUserDetails(user: User) {

  }

override fun onDestroy()
{
  presenter.onDestroy()
}
}
```

Al principio del ejemplo, definimos nuestro presentador:

```
private val presenter:
CreateUserPresenter<CreateUserView> by lazy {
      CreateUserPresenterImpl(this)
}
```

Se define como inmutable (val), y es creado por delegado perezoso, que se asignará la primera vez que sea necesario. Por otra parte, estamos seguros de que no será nulo (sin signo de interrogación después de la definición).

Cuando el usuario hace clic en el botón Guardar, View envía la información al Presenter con valores EditTexts. Cuando esto sucede, el usuario debe ser guardado, por lo que tenemos que implementar el método saveUser en Presenter (y algunas de las funciones del modelo):

```
override fun
saveUser(name: String, surname: String) {
  val user =
User(name, surname)
  when(UserValidator.validateUser(user)){
      UserError.EMPTY_NAME
-> view?.showEmptyNameError()
      UserError.EMPTY_SURNAME
-> view?.showEmptySurnameError()
      UserError.NO_ERROR
-> {
          UserStore.saveUser(user)
view?.showUserSaved()
          view?.showUserDetails(user)
      }
  }
}
```

Cuando se crea un usuario, se envía a **UserValidator** para comprobar su validez. Entonces, según el resultado de la validación, se llama el método apropiado. La construcción cuando () {} es igual que switch / case en Java. Pero es más potente - Kotlin permite el uso no sólo de enum o int en 'case', sino también de rangos, cadenas o tipos de objetos. Debes contener todas las posibilidades o tener una expresión. A continuación, cubre todos los valores UserError.

Al usar view? .showEmptyNameError () (con un signo de interrogación después del View), nos protegemos del **NullPointer** La vista puede anularse en el método **onDestroy**, y con esta construcción, no sucederá nada.

Cuando un modelo de usuario no tiene errores, le dice al **UserStore** que lo guarde y luego le indica al View que muestre el éxito y muestre los detalles.

Como se mencionó anteriormente, tenemos que implementar algunas cosas esenciales:

```
enum class UserError {
  EMPTY_NAME,
  EMPTY_SURNAME,
  NO_ERROR
}

object UserStore {
  fun
saveUser(user: User){
      //Save
```

```
user somewhere: Database, SharedPreferences, send to web...
  }
}

object UserValidator {

  fun
validateUser(user: User): UserError {
      with(user){
          if(name.isNullOrEmpty())
return UserError.EMPTY_NAME
          if(surname.isNullOrEmpty())
return UserError.EMPTY_SURNAME
      }

      return
UserError.NO_ERROR
  }
}

Lo más interesante aquí es *UserValidator.* Mediante el uso de la pa

Adicionalmente: en el método validateUser (usuario), existe con (usu

También hay otra pequeña cosa. Todo el código anterior, de enum a *U

Cuando un usuario se guarda, nuestra aplicación debería mostrarlo. N

Por lo tanto, vamos a definir la interfaz de*UserDetailsView*. Puede

~~~ kotlin
interface
UserDetailsView {
  fun
showUserDetails(user: User)
  fun
showNoUserError()
}
```

Next, UserDetailsPresenter. It should have a user property:

```
interface
UserDetailsPresenter<T: View>: Presenter<T> {
  var user:
User?
}
```

Esta interfaz se implementará en **UserDetailsPresenterImpl**. Tienes que sobrescribir la propiedad del usuario. Cada vez que se asigna esta propiedad, el usuario debe actualizarse en la vista. Podemos usar un **setter** de propiedad para esto:

```
class
UserDetailsPresenterImpl(override var view: UserDetailsView?):
UserDetailsPresenter<UserDetailsView> {
  override var
user: User? = null
      set(value)
{
          field
= value
          if(field
```

```
!= null){
                view?.showUserDetails(field!!)
            }
else {
                view?.showNoUserError()
            }
        }

}
```

La implementación **UserDetailsView**, **UserDetailsActivity**, es muy simple. Al igual que antes, tenemos un objeto presentador creado por la carga perezosa. El usuario al mostrar debe ser pasado por intención. Hay un pequeño problema con esto por ahora y lo resolveremos en un momento. Cuando tengamos el usuario de la intención, View necesita asignarlo a su presentador. Después de eso, el usuario se actualizará en la pantalla o, si es nulo, aparecerá el error (y la actividad terminará, pero el presentador no lo sabe):

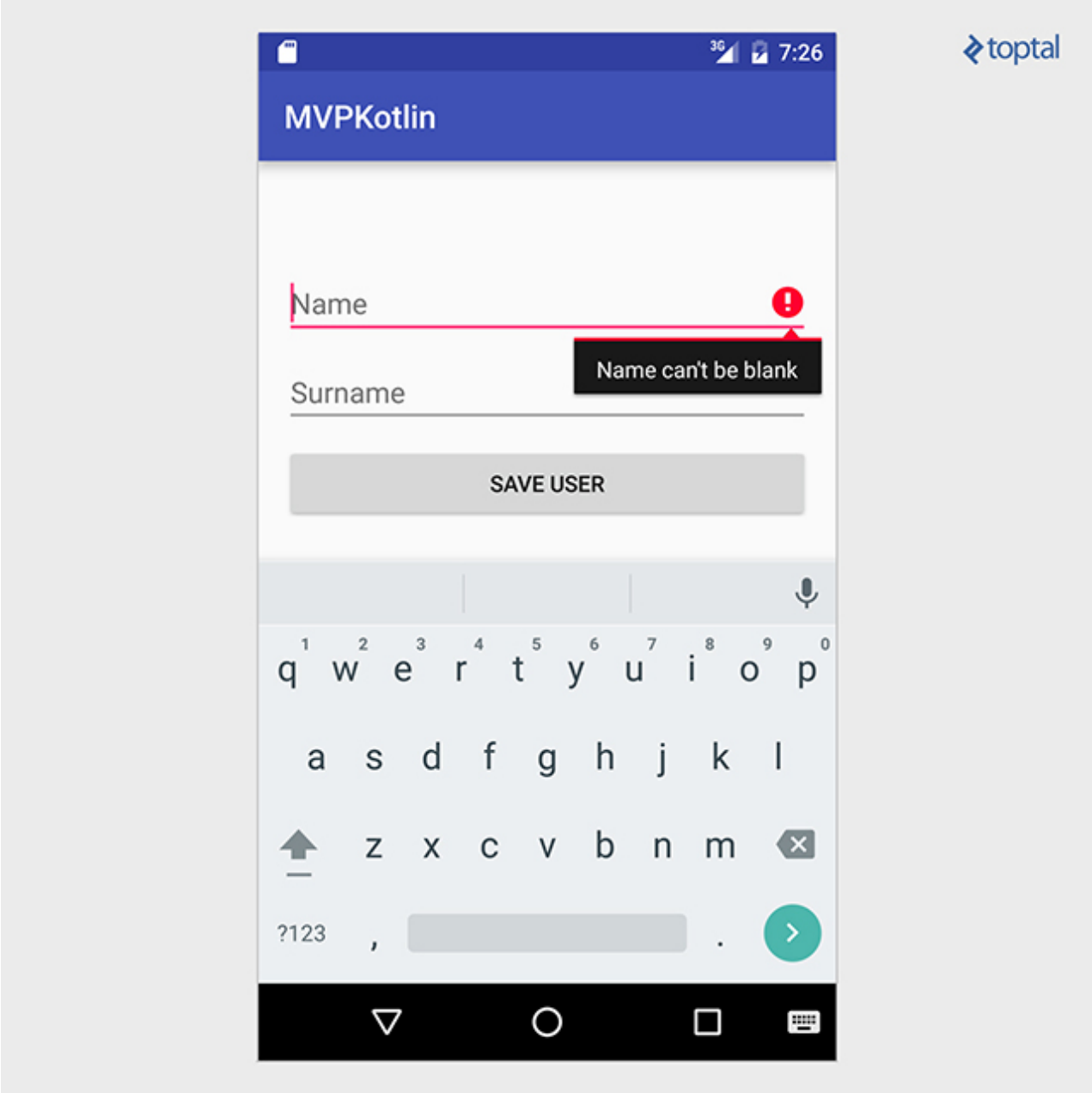
```
class
UserDetailsActivity: AppCompatActivity(), UserDetailsView {

  private val
presenter: UserDetailsPresenter<UserDetailsView> by lazy {
      UserDetailsPresenterImpl(this)
  }

  override fun
onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_user_details)

      val
user = intent.getParcelableExtra<User>(USER_KEY)
      presenter.user
= user
  }

  override fun
showUserDetails(user: User) {
      userFullName.text
= "${user.name} ${user.surname}"
  }

  override fun
showNoUserError() {
      toast(R.string.no_user_error)
      finish()
  }

override fun onDestroy()
{
  presenter.onDestroy()
}

}
```

Pasar objetos a través de intentos requiere que este objeto implemente la interfaz Parcelable. Esto es un trabajo muy “sucio”. Personalmente, odio hacer esto debido a todos los creadores, las propiedades, el ahorro, la restauración, y así sucesivamente. Afortunadamente, hay un **plugin** adecuado, Parcelable para Kotlin. Después de instalarlo, podemos generar la Parcelable con sólo un clic.

Lo último que debemos hacer es implementar **showUserDetails** (usuario: Usuario) en nuestra ActividadPrincipal:

```
anular divertido showUserDetails(usuario: Usuario) {
  startActivity<UserDetailsActivity>(USER_KEY to user)
/* extensión anko – empieza UserDetailsActivity y pasa el usuario a
intent */
}
```

Y eso es todo.



Tenemos una aplicación simple que guarda un usuario (en realidad, no se guarda, pero podemos agregar esta funcionalidad sin tocar el presentador o la vista) y lo presenta en la pantalla. En el futuro, si queremos cambiar la forma en la que se presenta el usuario en la pantalla, como por ejemplo: dos actividades a dos fragmentos en una actividad, o dos vistas personalizadas; los cambios solo se verán en clases Ver. Por supuesto, si no cambiamos la funcionalidad o la estructura

del modelo. El presentador, que no sabe exactamente qué es View, no necesitará ningún cambio.

¿Qué Sigue?

En nuestra aplicación, Presenter se crea cada vez que se crea una actividad. Este enfoque, o su opuesto, si Presenter debe persistir a través de instancias de actividad, es un tema de mucha discusión a través del Internet. Para mí, depende de la aplicación, sus necesidades y el desarrollador. A veces es mejor destruir el presentador, a veces no. Si decides persistir uno, una técnica muy interesante es usar **LoaderManager** para eso.

Como se mencionó anteriormente, MVP debe ser parte de **Uncle Bob's Clean architecture**. Por otra parte, los buenos reveladores deben utilizar **Dagger** para inyectar dependencias de los presentadores a las actividades. También ayuda a mantener, probar y reutilizar el código en el futuro. Actualmente, Kotlin trabaja muy bien con Dagger (antes del lanzamiento oficial no era tan fácil) y también con otras bibliotecas útiles de Android.

Conclusión

Para mí, Kotlin es un gran lenguaje. Es moderno, claro y expresivo, mientras se sigue desarrollando por grandes personas. Y podemos usar cualquier versión nueva en cualquier dispositivo y versión Android. Lo que me hace enojar en Java, Kotlin lo mejora.

Por supuesto, como he dicho nada es ideal. Kotlin también tiene algunas desventajas. Las versiones más recientes de plugins de Gradle (principalmente de alfa o beta) no funcionan bien con este idioma. Mucha gente se queja de que el tiempo de construcción es un poco más largo que el Java puro, y los apks tienen algunos MBs adicionales. Sin embargo, **Android Studio** y **Gradle** siguen mejorando y los teléfonos tienen cada vez más espacio para aplicaciones. Es por eso que creo que Kotlin puede ser un lenguaje muy agradable para todos los desarrolladores de Android. Sólo inténtalo y comparte en la sección de comentarios debajo, lo que piensas.

El código fuente de la aplicación de ejemplo está disponible en Github.

About the author

Hire the Author
Tomasz Czura, Poland
member since November 29, 2015
ReactJavaScriptFull-stackJavaAndroid StudioButterknifeAppsAndroidAndroid APIJIRAGitHubGit
Tomasz is an Android Developer who specializes in creating user-friendly and useful applications with the newest technologies. He enjoys experimenting with different languages and technologies, extending his skill set beyond Android to tasks leveraging React or iOS. Tomasz makes it a personal goal to ensure the client is fully satisfied, and often likens the process of software development to writing a good, interesting novel. [click to continue...]