SCALA ON ANDROID

Scala on Android

How to do efficient Android programming with Scala

Geoffroy Couprie

This book is for sale at http://leanpub.com/ScalaOnAndroid

This version was published on 2014-04-24

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Geoffroy Couprie by spreading the word about this book on Twitter!

The suggested tweet for this book is:

With ”Scala on Android”, my apps will shine like rainbows! https://leanpub.com/ScalaOnAndroid via @gcouprie

The suggested hashtag for this book is #ScalaOnAndroid.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#ScalaOnAndroid

This book is dedicated to my trusted Galaxy S2, which suffered through a lot of null pointer exceptions before I was enlightened by Scala

Contents

This sample chapter presents some of the useful features made available by Scala.

The power of Scala

Working with Scala on Android is a real joy. Its syntax helps in writing less code, and cleaner code.

Use implicit type conversions

Scala offers us a very powerful tool. By declaring a function as implicit, it will be used by the compiler to directly convert a type to another.

Take a look at the following code, setting up an OnClickListener for a button:

```
1  val button = findViewById(R.id.button).asInstanceOf[Button]
2    button.setOnClickListener(new View.OnClickListener() {
3      def onClick(v : View) {
4        Toast.makeText(this, "Clicked", Toast.LENGTH_LONG).show()
5      }
6    })
```

Yes, we need to create a new instance of the OnClickListener and override its `onClick` method. Doing that for each and every button is cumbersome.

With implicit conversions, we can get a much better syntax. First, let us define a `Helpers` object:

```
1  import android.view.View
2
3  object Helpers {
4    implicit def onClick(handler: View => Unit):View.OnClickListener = new View.OnC\
5  lickListener() {
6      override def onClick(source: View) = handler(source)
7    }
8  }
```

We wrote a method to convert a function taking a `View` as argument and returning nothing, to an OnClickListener instance. What can we do with that? We can now change the code like this:

```
import Helpers._
[...]
val button = findViewById(R.id.button).asInstanceOf[Button]
  button.setOnClickListener((v : View) => {
    Toast.makeText(this, "Clicked", Toast.LENGTH_LONG).show()
  })
```

Isn't that cleaner? And you can apply the same idea to other listeners, like `OnLongClickListener`.

Traits

Activities will often share common behaviour, like the menus, or how to handle some specific UI elements. In common Java code, we would create a `BehaviourActivity` extending `Activity`, and make our other activities inherit the behaviour. Or for more flexibility, create an interface, implemented by our activities? We should not have to choose between flexibility and code reuse.

That's where Scala traits are useful: we can share some code very easily. Here is an example of a trait settin up the activity's menu.

```
trait Menu extends Activity {

  override def onCreateOptionsMenu(menu: Menu) : Boolean = {
    menu.addSubMenu("Hello")
    true
  }
}
```

We can now easily add this trait to our activities:

```
class MainActivity extends Activity with TypedActivity, Menu {
   override def onCreate(bundle: Bundle) {
   super.onCreate(bundle)
   setContentView(R.layout.main)

    findView(TR.textview).setText("hello, world!")
  }
}
```

Don't hesitate to use traits extensively, even for a single class: you can separate concerns like basic UI management and data management easily.

Option

Android development is plagued by a very common error: the null pointer exception. We always need to check for null, for almost every result, or we will see lots of crashes. You do it in Java by littering your code with if/else, but Scala has the much more powerful `Option` type.

An example: how to get the number of satellites determining the phone's coordinates in Java?

```
1  String locProvider = LocationManager.GPS_PROVIDER;
2
3  Location loc = locManager.getLastKnownLocation(locProvider);
4
5  int satellites = 0;
6  if(loc != null) {
7    Bundle extras = loc.getExtras();
8    if(extras != null) {
9      satellites = extras.getInt("satellites");
10   }
11 }
```

The `getLastKnownLocation` method can return a `Location` instance, or null[1], and `getExtras` can return null too[2]. Instead of using the result directly in Scala, we will wrap it in an `Option`:

```
1  val locProvider = LocationManager.GPS_PROVIDER
2
3  val loc = Option(locManager.getLastKnownLocation(locProvider))
4
5  val nb = loc.map{_.getExtras()}.map{_.getInt("satellites")}.getOrElse(0)
```

This means that the type of `lastKnownLocation` is either `None`, or `Some[Location]`. If we call `map(f)` on a `None`, it returns `None`, but if we call it on `Some(a)`, it returns `Option(f(a))`, so `None` if `f(a) == null`, or `Some(f(a))` in the other case. We can chain map calls, until we really need the result. `getOrElse(x)` gives `a` if the option is `Some(a)`, or `x` if `None`.

Options are an interesting tool to reduce the size of your code and make it safer. There is a lot of nice patterns[3] availables with this construct.

[1] http://developer.android.com/reference/android/location/LocationManager.html#getLastKnownLocation%28java.lang.String%29
[2] http://developer.android.com/reference/android/location/Location.html#getExtras%28%29
[3] http://blog.tmorris.net/posts/scalaoption-cheat-sheet/