Data Classes & Lambdas

https://antonioleiva.com/kotlin-android-developers-book

Data Classes

```java
public class Artist {
    private long id;
    private String name;
    private String url;
    private String mbid;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public String getMbid() {
        return mbid;
    }

    public void setMbid(String mbid) {
        this.mbid = mbid;


    @Override public String toString() {
        return "Artist{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", url='" + url + '\'' +
                ", mbid='" + mbid + '\'' +
                '}';
    }
```

Simple Java “POJO” or “Bean” Class

Kotlin Equivalent- Data Class

```
data class Artist(
    var id: Long,
    var name: String,
    var url: String,
    var mbid: String)
```

This data class auto-generates all the fields and property accessors, as well as some useful methods such as toString().

You also get equals() and hashCode() for free, which are very verbose and can be dangerous if they are incorrectly implemented.

Data Classes

Edit Page

We frequently create classes whose main purpose is to hold data. In such a class some standard functionality and utility functions are often mechanically derivable from the data. In Kotlin, this is called a *data class* and is marked as `data`:

```
data class User(val name: String, val age: Int)
```

The compiler automatically derives the following members from all properties declared in the primary constructor:

— `equals()` / `hashCode()` pair;

— `toString()` of the form `"User(name=John, age=42)"`;

— componentN() functions corresponding to the properties in their order of declaration;

— `copy()` function (see below).

To ensure consistency and meaningful behavior of the generated code, data classes have to fulfill the following requirements:

— The primary constructor needs to have at least one parameter;

— All primary constructor parameters need to be marked as `val` or `var`;

— Data classes cannot be abstract, open, sealed or inner;

— (before 1.1) Data classes may only implement interfaces.

Lambdas

A lambda expression is a simple way to define an anonymous function.

Lambdas are very useful because they prevent us from having to write the specification of the function in an abstract class or interface, and then the implementation of the class.

In Kotlin, lambdas are first class citizens, which means that a function behaves as a type, so it can be passed as an argument to another function, can be returned by a function, saved into a variable or a property...

Implementing onClickListener

If we want to implement a click listener behaviour in Java, we first need to write the OnClickListener interface:

```
public interface OnClickListener {
    void onClick(View v);
}
```

And then we write an anonymous class that implements this interface:

```
view.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Toast.makeText(v.getContext(), "Click", Toast.LENGTH_SHORT).show();
    }
});
```

This would be the transformation of the code into Kotlin (using toast function from Anko):

```
view.setOnClickListener(object : OnClickListener {
    override fun onClick(v: View) {
        toast("Click")
    }
})
```

Kotlin allows some optimisations over Java libraries, and any function that receives an interface with a single function can be substituted by a lambda.

It will work as if we had defined setOnClickListener() like this:

```
fun setOnClickListener(listener: (View) -> Unit)
```

A lambda expression is defined by specifying the function input arguments at the left of the arrow (surrounded by parentheses), and the return type at the right.

In this case, we get a View and return Unit (nothing). So with this in mind, we can simplify the previous code a little:

```
view.setOnClickListener({ view -> toast("Click")})
```

While defining a function, we must use brackets and specify the argument values at the left of the arrow and the body of the function at the right.

We can even get rid of the left part if the input values are not being used:

```
view.setOnClickListener({ toast("Click") })
```

If the last argument of a function is also a function,
we can move it out of the parentheses:

```
view.setOnClickListener() { toast("Click") }
```

And, finally, if the function is the only parameter,
we can get rid of the parentheses:

```
view.setOnClickListener { toast("Click") }
```

```
view.setOnClickListener(object : OnClickListener {
    override fun onClick(v: View) {
        toast("Click")
    }
})
```

```
view.setOnClickListener({ view -> toast("Click")})
```

```
view.setOnClickListener({ toast("Click") })
```

```
view.setOnClickListener() { toast("Click") }
```

```
view.setOnClickListener { toast("Click") }
```

```
    btnAdd.setOnClickListener() {
      info("add Button Pressed")
    }
```

```
    btnAdd.setOnClickListener() {
      placemark.title = placemarkTitle.text.toString()
      if (placemark.title.isNotEmpty()) {
        info("add Button Pressed: $placemarkTitle")
      }
      else {
        toast ("Please Enter a title")
      }
    }
```