

## 本日の内容

- ・ kotlin の基礎知識\_基本構文
- ・ Android studio のデバッグ方法
- ・ Activity のライフサイクル

2年生からは、スマートフォンアプリ開発演習 I で学んだ内容を活かしながら、次のステップへ進みます。  
言語を Kotlin に変更し、内容も応用編に近くなります。Kotlin 特有の記述方法に慣れることから始めましょう。

## ■ Kotlin 基礎知識

- ・ Kotlin とは

IntelliJ IDEA で有名な JetBrains が開発した**オブジェクト指向プログラミング言語**  
2017 年に **Android 公式開発言語**に追加

- ・ 特徴

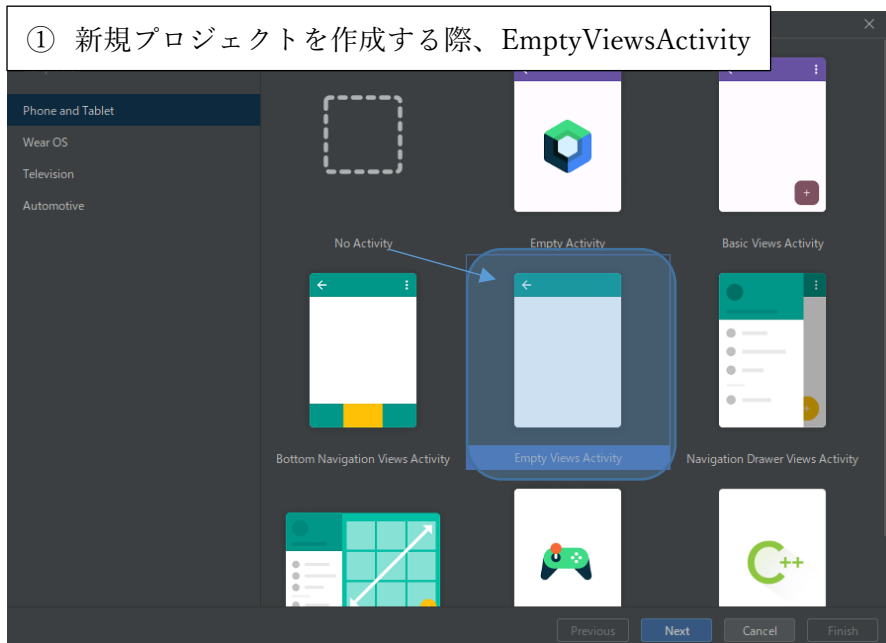
<b>Kotlin とは Java との相互運用性が高い</b>
Kotlin から Java を呼び出すことも、Java から Kotlin を呼び出すことも可能
<b>Null 安全 (Null Safety) を採用</b>
Kotlin の言語仕様で、Null 参照による実行時エラーを未然に防止する仕組み
<b>変数宣言に val と var の 2 つがある</b>
val は定数のような用途、var は変数のような用途で使い分ける
<b>行末の ; (セミコロン) を省略できる</b>
Java でコーディングする場合には行末に「;」を付ける必要がありましたが、Kotlin なら行末の「;」は不要です。
<b>型宣言は後ろに置く</b>
構文 (var 変数名: 型 = リテラル) 記述例 var num: Int = 1
<b>when 式</b>
when では switch と同様の処理を記述することができます、 Java の switch の条件分岐よりも Kotlin の when の方がシンプルでわかりやすいコードを記述することができます。
<b>ラムダ式</b>
Kotlin の標準ライブラリではラムダ式を多用していることから、 ラムダ式によってコレクションの操作がしやすくなっています。

等々…他にも細かい特徴は沢山ありますが、紹介はここまでとしておきます。

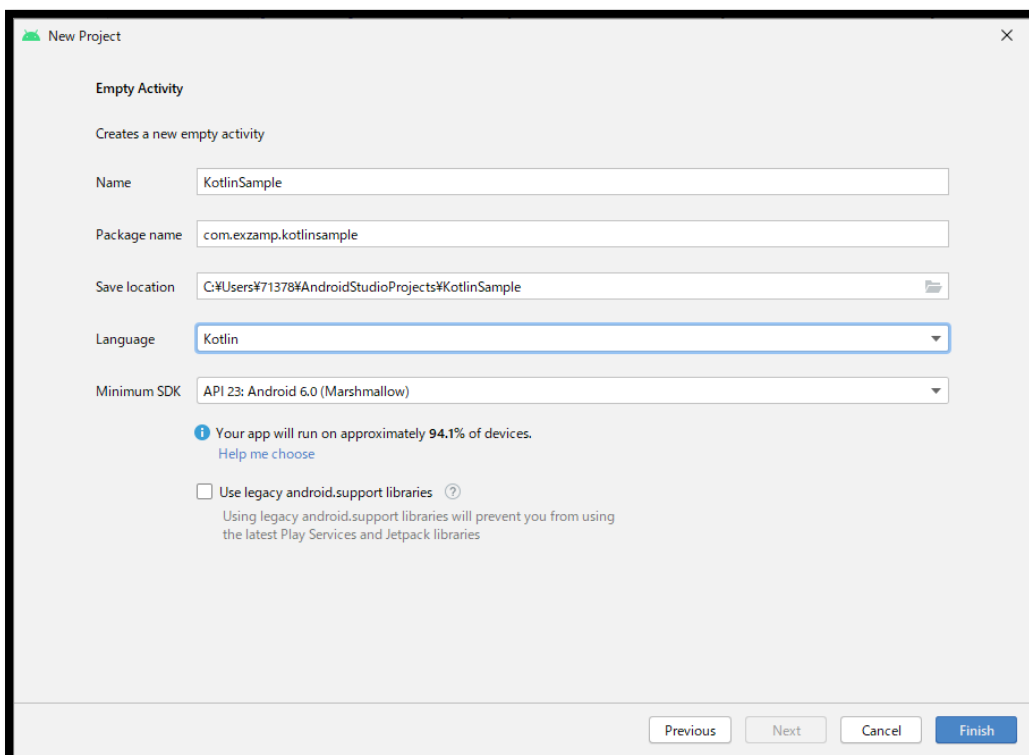
## ■ Kotlin 言語に触れてみる

### project 作成

まずは Android Studio で動かしながら慣れていきましょう



- ②Name を今回は[KotlinSample]と記入
- ③Language を[Kotlin]に変更
- ④Finish をクリック

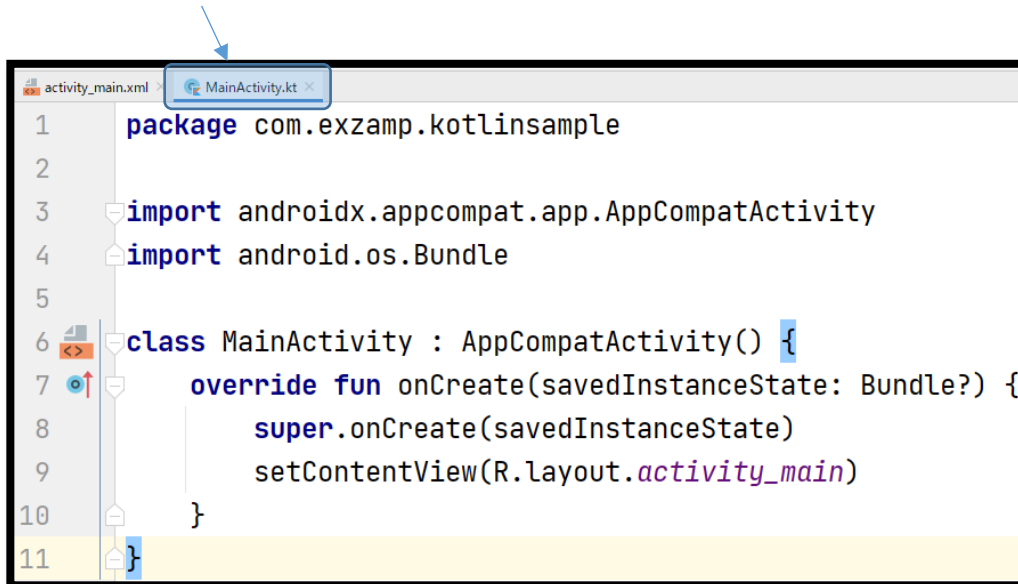


## スマートフォンアプリ開発演習

ビルドに少し時間が掛かります。

生成されたファイル名が.kt となっていますね！

これが Kotlin ファイルの拡張子となります。



事前準備 OK です

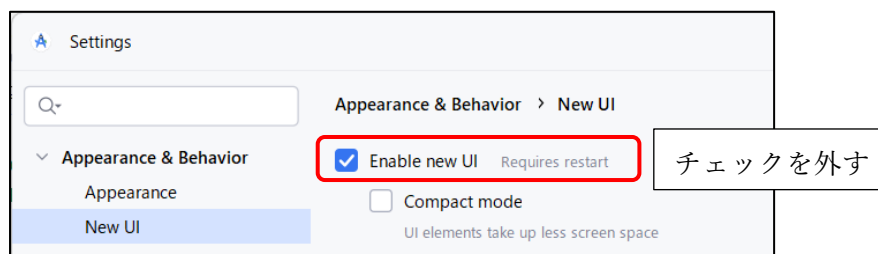
ここから、言語の構文を学んでいきますが、毎回 Android に書き込んで動作確認を行うと処理が重いため、KotlinREPL という機能で Kotlin 言語を学びます。

以前の UI に戻す

Android Studio Giraffe から開発環境の UI が大きく変化しています。

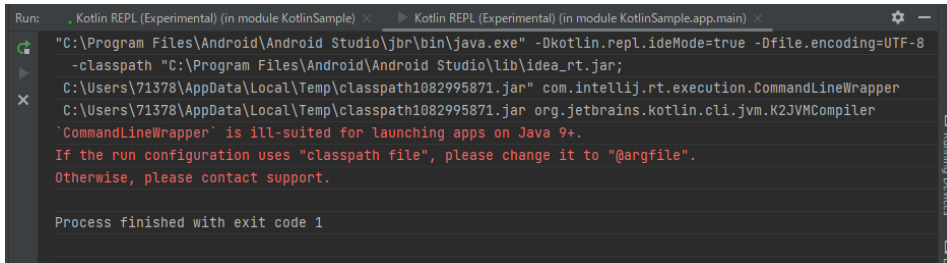
皆さんは UI 変更前にスマートフォンアプリ開発演習 I を学習していると思いますので、もし新しい UI に変更されている場合は、以下の手順で以前の UI に戻しておきましょう。

[File]-[Settings]で設定画面を開く。



## ■ KotlinREPL の起動

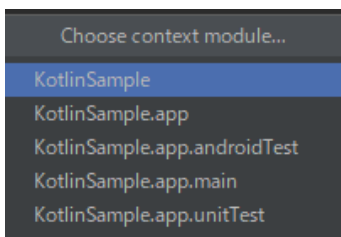
上部タブの「Tools」 > 「Kotlin」 > 「Kotlin REPL」をクリック



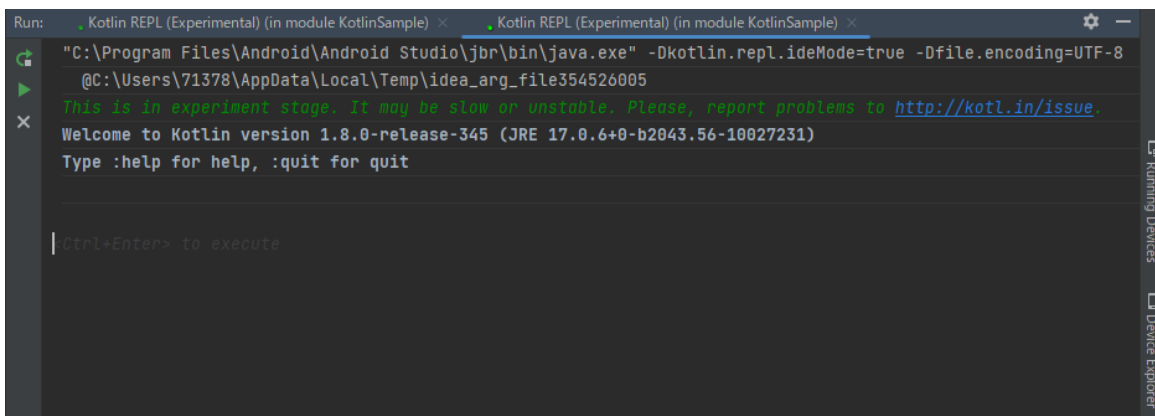
```
Run: Kotlin REPL (Experimental) (in module KotlinSample) × Kotlin REPL (Experimental) (in module KotlinSample.app.main) ×
"C:\Program Files\Android\Android Studio\jbr\bin\java.exe" -Dkotlin.repl.ideMode=true -Dfile.encoding=UTF-8
-classpath "C:\Program Files\Android\Android Studio\lib\idea_rt.jar;
C:\Users\71378\AppData\Local\Temp\classpath1082995871.jar" com.intellij.rt.execution.CommandLineWrapper
C:\Users\71378\AppData\Local\Temp\classpath1082995871.jar org.jetbrains.kotlin.cli.jvm.K2JVMCompiler
'CommandLineWrapper' is ill-suited for launching apps on Java 9+.
If the run configuration uses "classpath file", please change it to "@argfile".
Otherwise, please contact support.

Process finished with exit code 1
```

初回はエラーが出るので、この window を選択している状態で再度  
上部タブの「Tools」 > 「Kotlin」 > 「Kotlin REPL」をクリック



この作成したプロジェクトの一番上の KotlinSample を選択



```
Run: Kotlin REPL (Experimental) (in module KotlinSample) × Kotlin REPL (Experimental) (in module KotlinSample) ×
"C:\Program Files\Android\Android Studio\jbr\bin\java.exe" -Dkotlin.repl.ideMode=true -Dfile.encoding=UTF-8
@C:\Users\71378\AppData\Local\Temp\idea_arg_file354526005
This is in experiment stage. It may be slow or unstable. Please, report problems to http://kotl.in/issue
Welcome to Kotlin version 1.8.0-release-345 (JRE 17.0.6+0-b2043.56-10027231)
Type :help for help, :quit for quit

| Ctrl+Enter> to execute
```

上記のようになっていれば OK です。

## ■ Kotlin の基本的な構文

ではここからは、実際にプログラミングしながら、構文と動きを確認していきましょう。

### ステップ 1: 数値演算子を調べる

他の言語と同様に、Kotlin は、プラス、マイナス、乗算、除算を使用します。

Kotlin は `+-*/` / `Int`、`Long`、`Double`、`Float` などのさまざまな数値タイプもサポートしています。

REPL に次の式を入力します。結果を確認するには、それぞれの後に `Control+Enter`

(Mac の場合: `Command+Enter`)を押します。

#### 1.1 試しに、5つの式を順番に実行してみましょう

```
1+1
⇒ res8: kotlin.Int = 2
53-3
⇒ res9: kotlin.Int = 50
50/10
⇒ res10: kotlin.Int = 5
1.0/2.0
⇒ res11: kotlin.Double = 0.5
2.0*3.5
⇒ res12: kotlin.Double = 7.0
```

演算の結果はオペランドの型を保持するため、`1/2 = 0` ですが、`1.0/2.0 = 0.5` であることに注意してください。

#### 1.2 数値に対していくつかのメソッドを呼び出します。

Kotlin は数値をプリミティブとして保持しますが、数値のメソッドをオブジェクトであるかのように呼び出すことができます。

```
2.times(3)
⇒ res5: kotlin.Int = 6

3.5.plus(4)
⇒ res8: kotlin.Double = 7.5

2.4.div(2)
⇒ res9: kotlin.Double = 1.2
```

## ステップ 2: 型の使用を練習する

Kotlin は暗黙的に数値型を変換しないので、long 変数に直接 short 値を代入したり、.NET 変数に直接 a を代入したりすることはできません。これは、暗黙的な数値変換がプログラムにおけるエラーの一般的な原因であるためです。異なる型の値を代入するには、常にキャストする必要があります。

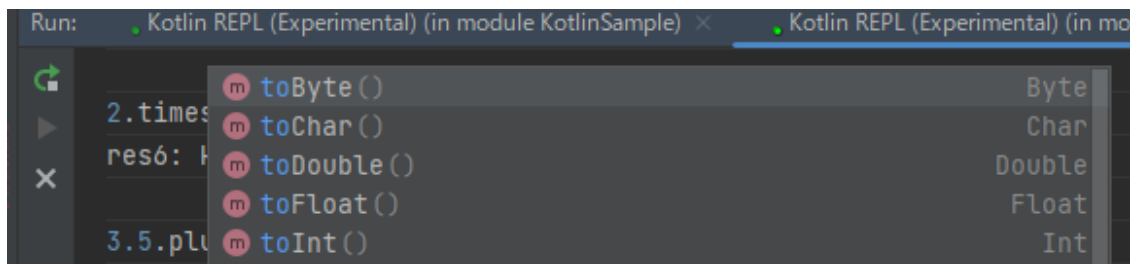
2.1 可能なキャストのいくつかを見るために、REPL.Int で型の変数を定義する。

```
val i: Int = 6
```

2.2 新しい変数を作成し、上に示した変数名の後に.to を入力する。

```
val b1 = i.to
```

補完可能なリストが表示されるので toByte() を選択



変数を出力してみましょう

```
println(b1)
```

```
⇒ 6
```

2.4 異なる型の変数に値を割り当てる。

```
val b2: Byte = 1 // OK, literals are checked statically
println(b2)
⇒ 1

val i1: Int = b2
⇒ error: type mismatch: inferred type is Byte but Int was expected

val i2: String = b2
⇒ error: type mismatch: inferred type is Byte but String was expected

val i3: Double = b2
⇒ error: type mismatch: inferred type is Byte but Double was expected
```

2.5 エラーを返した代入については、代わりにキャストしてみてください。

```
val i4: Int = b2.toInt() // OK!
println(i4)
⇒ 1

val i5: String = b2.toString()
println(i5)
⇒ 1

val i6: Double = b2.toDouble()
println(i6)
⇒ 1.0
```

2.6 長い数値定数を読みやすくするために、Kotlin では数字の中にアンダースコアを入れることができます。異なる数値定数を入力してみてください。

```
val oneMillion = 1_000_000
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

### ステップ3：変数タイプの値を学ぶ

Kotlin は、変更可能な変数と変更不可能な変数の2種類をサポートしています。

`val` を使えば、**一度だけ値を代入することができます**。もう一度値を代入しようとするエラーになる。

`var` を使えば、**値を代入し、後でプログラムの中で値を変更することができます**。

3.1 `val` `var` を使用して変数を定義し、新しい値を代入する。

```
var fish = 1
fish = 2
val aquarium = 1
aquarium = 2
```

⇒ `error: val cannot be reassigned`

`fish` は `var` 定義されているので、値を代入し、新しい値を代入することができます。

`aquarium` は `val` で定義されているので、新しい値を代入しようするとエラーになります。

変数に格納する型は、コンパイラが文脈から割り出すことができる場合に推測されます。

必要であれば、コロン記法を使って変数の型を明示的に指定することもできます。

3.2 いくつかの変数を定義し、型を明示的に指定する。

```
var fish: Int = 12
var lakes: Double = 2.5
```

一度、コンパイラによって型が割り当てられると、その型を変更することは出来ません。



#### ステップ 4：Strings について学ぶ

Kotlin の文字列は、他のプログラミング言語の文字列とほとんど同じように、文字列と単一文字に対して機能し、演算子を使って文字列を連結することができる。変数名は値を表すテキストに置き換えられます。これは変数補間と呼ばれます。

```
val numberOfFish = 5
val numberOfPlants = 12
"I have $numberOfFish fish" + " and $numberOfPlants plants"
```

```
⇒ res20: kotlin.String = I have 5 fish and 12 plants
```

他の言語と同様、値は式の結果であることが出来ます。中括弧を使って式を定義します。

```
"I have ${numberOfFish + numberOfPlants} fish and plants"
```

```
⇒ res21: kotlin.String = I have 17 fish and plants
```

## ステップ5 タスク 条件とブール値の比較

他の言語と同様に、

Kotlin にもブーリアンと、less than、equal to、greater than などのブーリアン演算子があります。

```
val numberOfFish = 50
val numberOfPlants = 23
if (numberOfFish > numberOfPlants) {
    println("Good ratio!")
} else {
    println("Unhealthy ratio")
}
⇒ Good ratio!
```

### 5.1 ステートメントで範囲を試す

Kotlin では、テストする条件にも範囲を使うことができます。

```
val fish = 50
if (fish in 1..100) {
    println(fish)
}
⇒ 50
```

より複雑な条件の場合は、論理と論理または論理を使用します。

他の言語と同様に、 if && || else if を使用することで複数のケースを持つことができます。

```
if (numberOfFish == 0) {
    println("Empty tank")
} else if (numberOfFish < 40) {
    println("Got fish!")
} else {
    println("That's a lot of fish!")
}
```

### 5.2 ステートメントを使ってみよう。

他の言語のステート Kotlin では、if else をより簡潔に書く方法があります。それは when ステートメントを使用する方法です。この when は、他の言語の switch のようなものです。また、when の条件では範囲も使用できます。従って、他の言語の if-else if-else のような条件分岐も、Kotlin では when を使って表現することが可能です。

```
when (numberOfFish) {
    0 -> println("Empty tank")
    in 1..39 -> println("Got fish!")
    else -> println("That's a lot of fish!")
}
⇒ That's a lot of fish!
```

## ステップ 6: null 値の許容について学習する

null 変数と非 null 変数について学ぶ。NULL を含むプログラミング・エラーは数え切れないほどのバグの原因となってきました。Kotlin では、nullable でない変数を導入することで、バグを減らそうとしています。

### 6.1 null 値の許容について

デフォルトでは、変数を null にすることはできません。

Int 型の null を宣言して割り当てます。

```
var rocks: Int = null
⇒ error: null can not be a value of a non-null type Int
```

型の後にクエションマーク演算子を使って、変数が null になる可能性があることを示す。

```
var marbles: Int? = null
```

### 6.2 ? と演算子について学ぶ

? 演算子を使ってテストできるので、多くの null チェック文を書く手間が省けます。

【従来の書き方】

```
var fishFoodTreats = 6
if (fishFoodTreats != null) {
    fishFoodTreats = fishFoodTreats.dec()
}
```

【? 演算子を使った Kotlin の書き方】

```
var fishFoodTreats = 6
fishFoodTreats = fishFoodTreats?.dec()
```

エルビス演算子を使って null のテストを連鎖させることもできる。

```
fishFoodTreats = fishFoodTreats?.dec() ?: 0
```

これは、null でない場合は、デクリメントして使用します。それ以外の場合は、値 (0) を使用します。

not-null アサーション演算子 (!!)

値が の場合は例外をスローします。

```
val len = s!!.length // throws NullPointerException if s is null
```

## ステップ 7: 配列、リスト、ループ

### 7.1 リストを作成する

リストは Kotlin の基本的なタイプであり、他の言語のリストに似ています。

```
val school = listOf("mackerel", "trout", "halibut")
println(school)
⇒ [mackerel, trout, halibut]
```

7.2 `mutableListOf` を使用して **変更できるリスト** を宣言します。アイテムを削除します。

```
val myList = mutableListOf("tuna", "salmon", "shark")
myList.remove("shark")
⇒ res36: kotlin.Boolean = true
```

### 7.3 配列を作成する

他の言語と同様に、Kotlin には配列があります。

変更可能なバージョンと不変バージョンがある Kotlin のリストとは異なり、**Array 配列**を作成すると、**サイズは固定**されます。新しい配列にコピーする場合を除き、**要素を追加または削除することはできません**。

```
val school = arrayOf("shark", "salmon", "minnow")
println(java.util.Arrays.toString(school))
⇒ [shark, salmon, minnow]
```

`arrayOf` で宣言された配列には要素に関連付けられた型がないため、型を混在させることができ、便利です。異なる型の配列を宣言します。

```
val mix = arrayOf("fish", 2)
```

すべての要素に対して 1 つの型で配列を宣言することもできます。`intArrayOf()` を使用して整数の配列を宣言します。他の型の配列に対応するビルダーまたはインスタンス化関数があります。

```
val numbers = intArrayOf(1,2,3)
```

7.4 2 つの配列を `+` 演算子で結合します。

```
val numbers = intArrayOf(1,2,3)
val numbers3 = intArrayOf(4,5,6)
val foo2 = numbers3 + numbers
println(foo2[5])
⇒ 3
```

## ステップ 8 ループを作成する

8.1 配列を作成します。for ループを使用して配列を反復処理し、要素を出力します。

```
val school = arrayOf("shark", "salmon", "minnow")
for (element in school) {
    print(element + " ")
}
⇒ shark salmon minnow
```

8.2 Kotlin では、要素とインデックスを同時にループ処理できます。

```
for ((index, element) in school.withIndex()) {
    println("Item at $index is $element\n")
}
⇒ Item at 0 is shark
Item at 1 is salmon
Item at 2 is minnow
```

8.3 まざまなステップサイズと範囲を試してください。数字または文字の範囲をアルファベット順に指定できます。また、他の言語と同様に、1 ずつ前進する必要はありません。downTo を使用して後方にステップできます。

```
for (i in 1..5) print(i)
⇒ 12345

for (i in 5 downTo 1) print(i)
⇒ 54321

for (i in 3..6 step 2) print(i)
⇒ 35

for (i in 'd'..'g') print (i)
⇒ defg
```

8.4 いくつかのループを試してみましょう。

他の言語と同様に、Kotlin には while ループ、do...while ループ、++および--演算子があります。

Kotlin には repeat ループもあります。

```
var bubbles = 0
while (bubbles < 50) {
    bubbles++
}
println("$bubbles bubbles in the water\n")

do {
    bubbles--
} while (bubbles > 50)
println("$bubbles bubbles in the water\n")

repeat(2) {
    println("A fish is swimming")
}
```

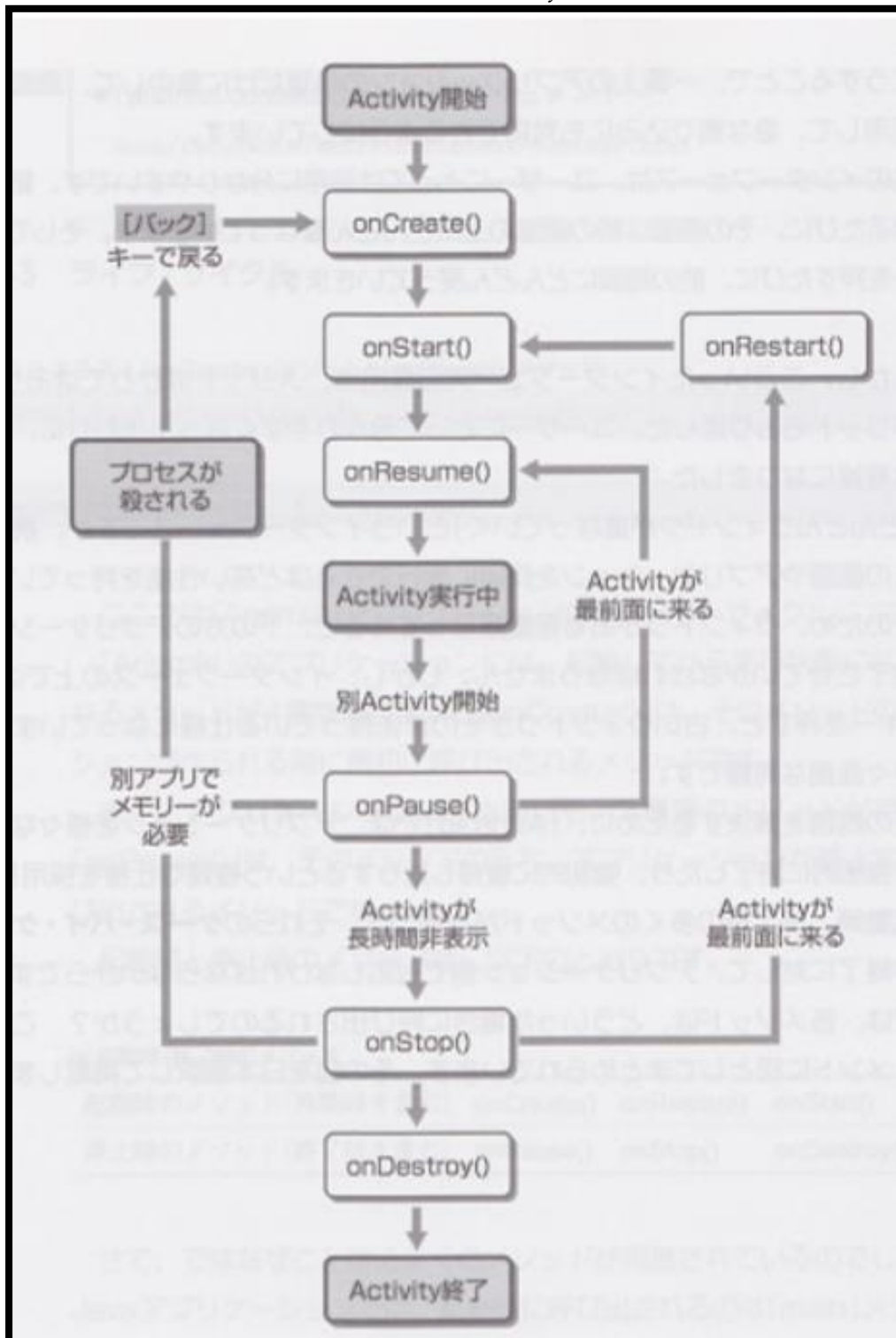
## ■Activity のライフサイクル

Android は画面遷移やバックグラウンドへの移動、アプリキルなど様々な動作により Activity の生成と消滅を繰り返しています。

このような1つ1つの Activity が生成されてから破棄されるまでの工程を **ライフサイクル** といいます。

### ◆Activity ライフサイクルのコールバック

Activity のライフサイクルの各段階を移動するために、Activity クラスから提供されているコールバックが7つ



各コールバックの説明

## ◆onCreate()

Activity が生成される際、最初に呼ばれる。

ライフサイクルの中で1回のみ実行するものなので

基本的な初期設定やパラメータ savedInstanceState の受け取りを行うことが多い

## ◆onStart()

Activity がフォアグラウンドに移動し、操作可能な状態になるまでの間に呼ばれる。

## ◆onResume()

Activity が前面に表示され、ユーザとやりとりが可能になる直前に呼び出される

そのため、表示に必要な処理や初期化をここで処理することが多い

## ◆onPause()

アプリがバックグラウンドに移動したことを示すために呼び出される。

そのため、続行しない操作を停止、続行処理はバックグラウンドにあることを想定して調整を行う

なお、実行は非常に短時間で終了するため、状態の保存には適していない

## ◆onStop()

アプリが完全にバックグラウンドに移動した時に呼び出される

Activity は非表示になり、停止している。

そのため、高負荷の状態保存はここで行う(アプリデータの保存、ネットワークの呼び出し)

## ◆onRestart()

Activity の再表示の際に呼び出される

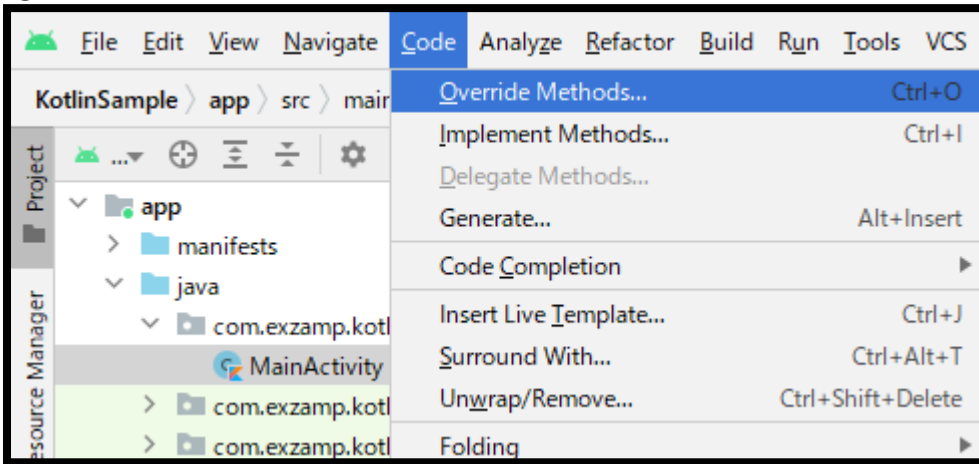
## ◆onDestoroy()

以下の内容で Activity が破棄される前に呼び出される

- ・アプリキルなどして Activity が終了した時
- ・画面の回転、テーマ変更、マルチウィンドウモードにより Activity が一時的に破棄される

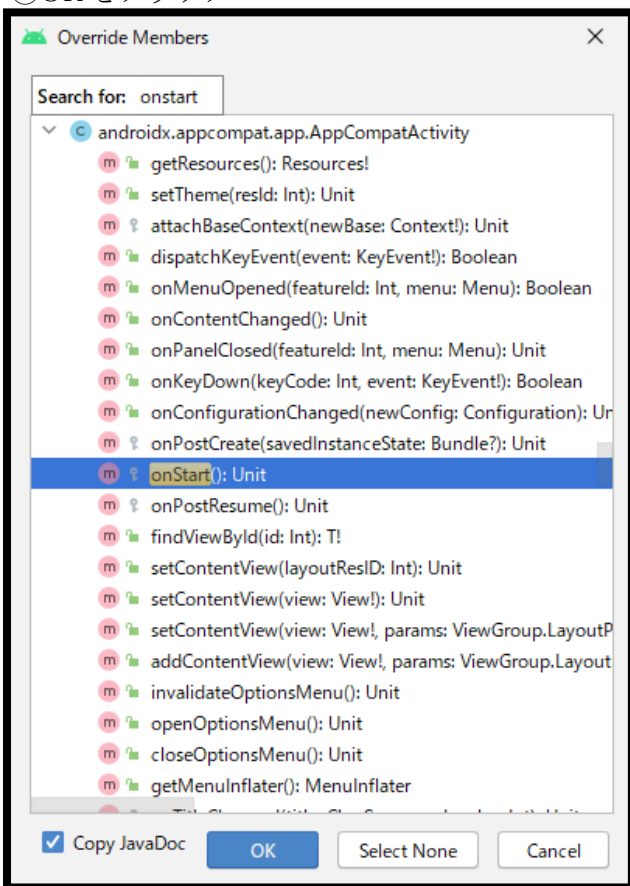
◆各コールバックの実装、および動作タイミングの確認

①Code タブの Override Methods..をクリック



②出てきた window 内で検索したい文字を入力[例:onStart]しコールバックメソッドを探す

③OK をクリック



指定したコールバック関数が追加されました

```
override fun onStart() {  
    super.onStart()  
}
```



## スマートフォンアプリ開発演習

同様に

- ・ onResume()
- ・ onPause()
- ・ onStop()
- ・ onRestart()
- ・ onDestroy()

も追加しましょう。

◆各コールバックに確認用のログを出力するように処理を追加

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    Log.d("test", msg: "call onCreate()")  
}
```

```
override fun onStart() {  
    super.onStart()  
    Log.d("test", msg: "call onStart()")  
}
```

```
override fun onResume() {  
    super.onResume()  
    Log.d("test", msg: "call onResume()")  
}
```

```
override fun onPause() {  
    super.onPause()  
    Log.d("test", msg: "call onPause()")  
}
```

```
override fun onStop() {  
    super.onStop()  
    Log.d("test", msg: "call onStop()")  
}
```

```
override fun onRestart() {  
    super.onRestart()  
    Log.d("test", msg: "call onRestart()")  
}
```

```
override fun onDestroy() {  
    super.onDestroy()  
    Log.d("test", msg: "call onDestroy()")  
}
```

◆アプリ起動時のライフサイクル

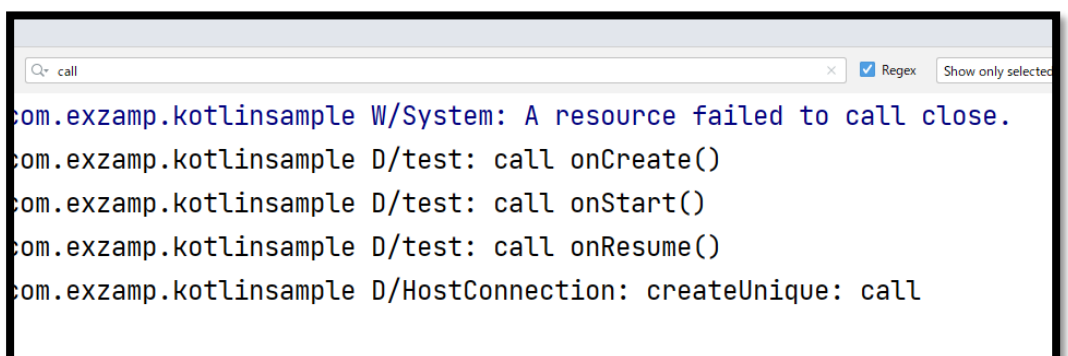
では実行してログを確認していきます。

まずは実行してみましょう。



ログはそのままと見つけにくいので

検索窓に「call」と入力して文字を絞り込んでいます



## スマートフォンアプリ開発演習

アプリ起動時の onCreate()

アプリ表示の onStart()

ユーザ操作可能な onResume()

のタイミングでログが出力されていますね！

### ◆ホームボタン押下のライフサイクル

ではホームボタンを押下してアプリをバックグラウンド状態にさせてみましょう



```
om.exzamp.kotlinsample W/System: A resource failed to call close.  
om.exzamp.kotlinsample D/test: call onCreate()  
om.exzamp.kotlinsample D/test: call onStart()  
om.exzamp.kotlinsample D/test: call onResume()  
om.exzamp.kotlinsample D/HostConnection: createUnique: call  
om.exzamp.kotlinsample D/test: call onPause()  
om.exzamp.kotlinsample D/test: call onStop()
```

バックグラウンドへ移動開始時の onPause()

バックグラウンド移動完了時の onStop()

のタイミングでログが出力されていますね！

### ◆アプリ復帰時のライフサイクル

では□ボタンを押下した状態で、アプリを選択することで

再度アプリをフォアグラウンドに戻しましょう



```
om.exzamp.kotlinsample W/System: A resource failed to call  
om.exzamp.kotlinsample D/test: call onCreate()  
om.exzamp.kotlinsample D/test: call onStart()  
om.exzamp.kotlinsample D/test: call onResume()  
om.exzamp.kotlinsample D/HostConnection: createUnique: call  
om.exzamp.kotlinsample D/test: call onPause()  
om.exzamp.kotlinsample D/test: call onStop()  
om.exzamp.kotlinsample D/test: call onRestart()  
om.exzamp.kotlinsample D/test: call onStart()  
om.exzamp.kotlinsample D/test: call onResume()
```

## スマートフォンアプリ開発演習

アプリの再表示開始時の `onRestart()`

アプリ表示時の `onStart()`

ユーザ操作可能な `onResume()`

のタイミングでログが出力されていますね！

### ◆アプリキル時のライフサイクル

では□ボタンを押下して、更にアプリを上からスワイプすることでアプリキルになります。



```
call
om.exzamp.kotlinsample D/HostConnection: createUnique: call
om.exzamp.kotlinsample D/test: call onPause()
om.exzamp.kotlinsample D/test: call onStop()
om.exzamp.kotlinsample D/test: call onRestart()
om.exzamp.kotlinsample D/test: call onStart()
om.exzamp.kotlinsample D/test: call onResume()
om.exzamp.kotlinsample D/test: call onPause()
om.exzamp.kotlinsample D/test: call onStop()
om.exzamp.kotlinsample D/test: call onDestroy()
```

バックグラウンドへ移動開始時の `onPause()`

バックグラウンド移動完了時の `onStop()`

アプリがキルされる直前の `onDestroy()`

のタイミングでログが出力されていますね！

## ■デバッグについて

- ・デバッグ方法：その 1.ログで確認

デバッグとは、コンピュータプログラムに潜む欠陥を探し出して取り除くこと。

エラー内容を確認するだけでなく、プログラムの途中で変数に何が入っているかの確認もデバッグ機能で確認することが可能です。

- ・デバッグ方法：その 2.ブレイクポイントで止めて確認する

ログに出力するのではなく、特定行の時点で変数には何が格納されているのかを Android Studio では確認することが出来ます。

特定の行で処理を止める為に「**ブレイクポイント**」を設定します。

一時的に止めたい処理の行の場所をクリックすることでブレイクポイントを設定できます

ブレイクポイントを設定した状態でデバッグ実行をします。デバッグ実行は虫アイコンをクリックで行います

