A Dozen Techniques for Everyday Kotlin DSLs

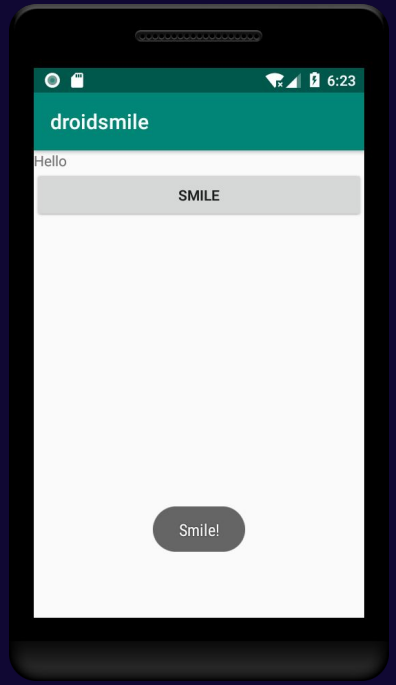
Greg Milette
Android Developer
@gregmilette

APRIL 08+09
DROIDCON BOS19

Hello Android, please make me a text view with the text “Hello”

Android Code

```
val layout = LinearLayout(this)
layout.orientation = LinearLayout.VERTICAL
val hello = TextView(this)
hello.text = "Hello"
val button = Button(this)
button.text = "Smile"
button.setOnClickListener {
     Toast.makeText(this, "Smile!",
     Toast.LENGTH_SHORT).show()
}
layout.addView(hello)
layout.addView(button)
```



Android Code with Anko

```
verticalLayout {
    textView { text = "Hello" }
    button("Smile") {
        onClick {
            toast("Smile!")
        }
    }
}
```

What Makes a Domain Specific Language

Goal: Concise, readable syntax

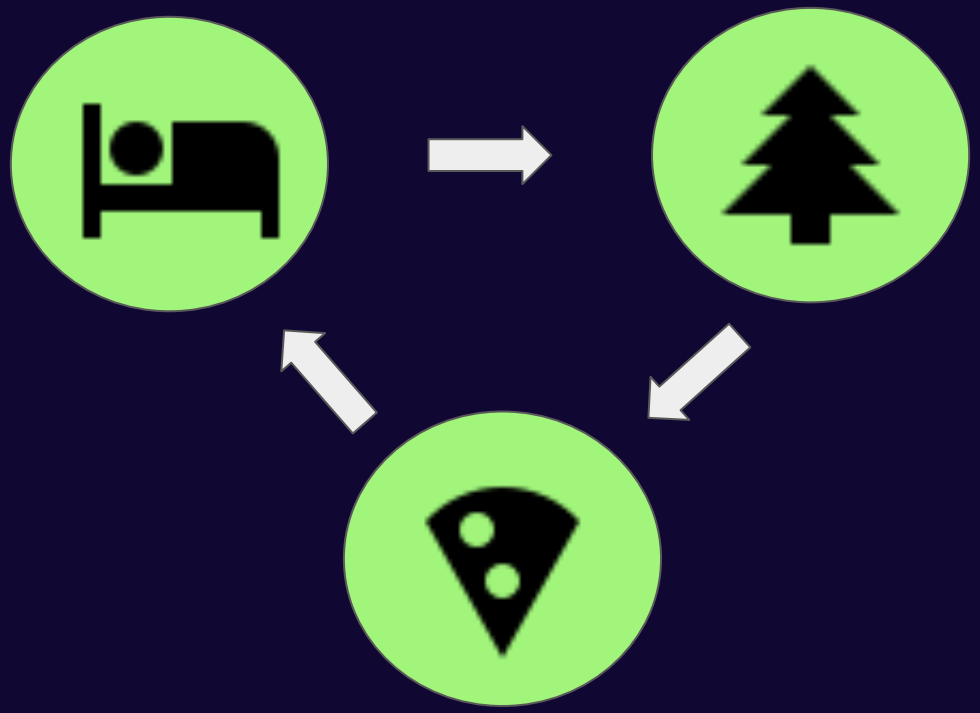
- Language Nature
- Domain Focus
- Limited Expressiveness

```
val layout = LinearLayout(this)
layout.orientation =
        LinearLayout.VERTICAL
val hello = TextView(this)
hello.text = "Hello"
layout.addView(hello)
```

```
verticalLayout {
  textView { text = "Hello" }
}
```

See Domain-Specific Languages by Martin Fowler and Rebecca Parsons

Problem: Construct a Tour

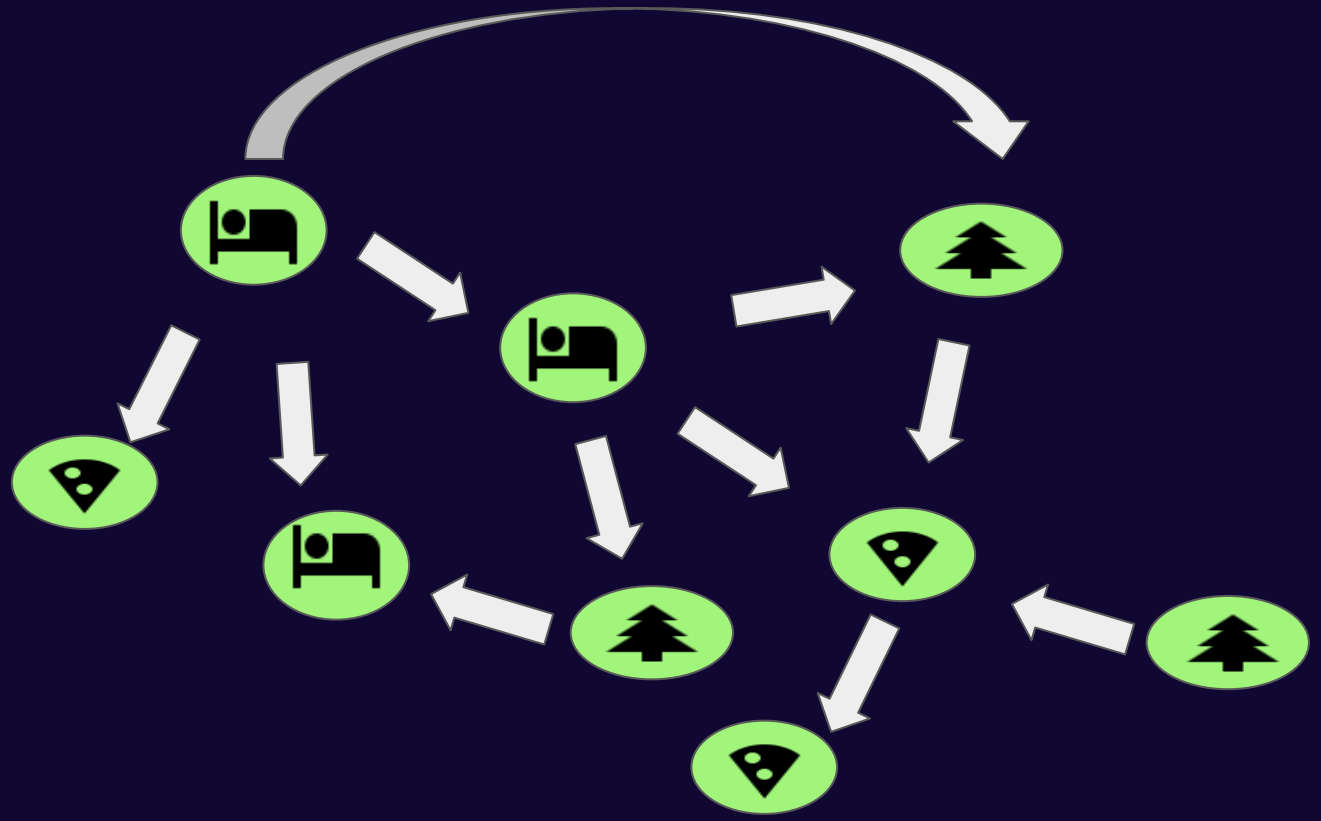


A Tour with 3 Places

```
val hotel = Hotel("Hotel", 4)
val restaurant = Restaurant("Pizza", "italian")
val park = Park("Park")

val tour = Tour()
tour.step(hotel, restaurant)
tour.step(restaurant, park)
tour.step(park, hotel)
```

World Wide Tour?



Add Tour 2

```
val sushi = Restaurant("Sushi", "Japanese")
val hotel2 = Hotel("Hotel2", 4)
val tour2 = Tour()
tour2.step(hotel2, park)
tour2.step(park, sushi)
tour2.step(sushi, hotel2)

// still more and more tours
```

Split into Functions

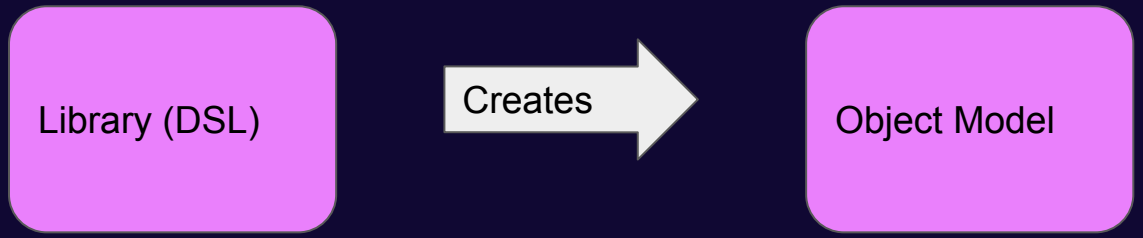
```
val places = mutableMapOf<String, Place>()

populatePlaces(places)

makeTour(places)
makeTour2(places)
makeTour3(places)
makeTour4(places)
```

Options for Improving Authoring of Tour

1. Improve Object Model
2. Configuration File
3. Code Library for Creating Object Model (expose a DSL)



A Home for the DSL

```
class TourBuilder {
    // TODO

    fun build(): Tour {

    }

}
```

DSL Toolbox

Function chains	Symbols	@DSLMarker
Naming	Lambda with receiver	Extension function
Nested builders	Override property	Infix
Context variables	Invoke operator	

Create a Function Chain

```
fun name(name: String): TourBuilder {
    this.name = name
    return this
}
```

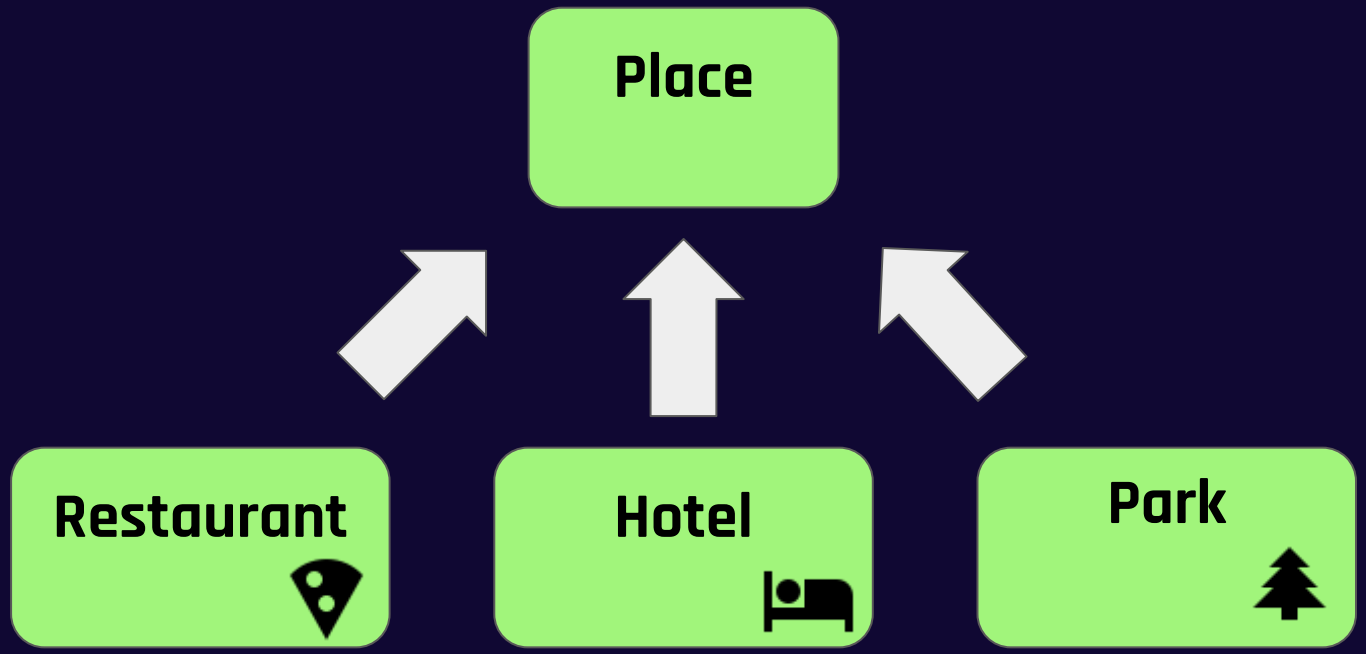
Builders: Create Function Chains

```
val restaurant =
    TourBuilder()
        .name("Pizza")
        .kind("Restaurant")
        .build()
```

Builders: Named Like a Language

```
val restaurant =
    TourBuilder()
        .named("Pizza")
        .isKindOf("Restaurant")
        .build()
```

3 Types of Places



Nested Builders: Enforcing a Language

```
val restaurant = TourBuilder()
    .named("Pizza")
    .asRestaurant()
    .cuisine("Italian")
    .build()
```

Restaurant-specific

Nested Builders: Enforcing a Language

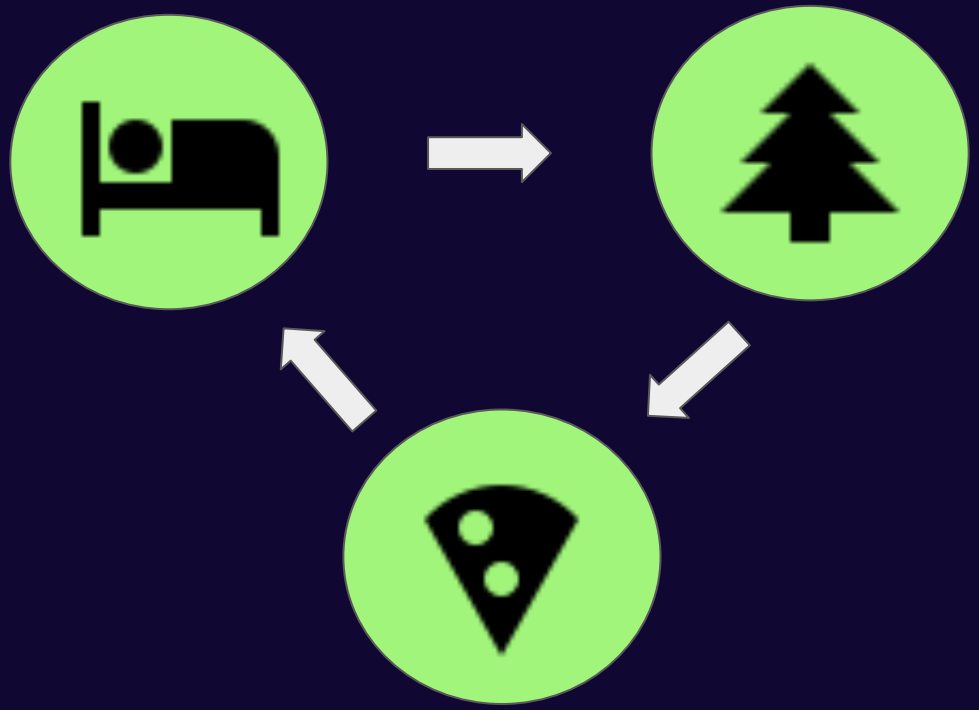
```
class TourBuilder {
    fun asRestaurant() = RestaurantBuilder(this)
    fun asHotel() = HotelBuilder(this)
}
```

Benefits of Nested Builders

```
val restaurant = TourBuilder()
    .named("Pizza")
    .asRestaurant()
    .cuisine("Italian")
    .build()
```

- Single builder
- Only valid objects
- IDE can assist

Author a Tour



Create a Tour

```
val hotel = builder.place("Hotel") //...
val park = builder.place("Park") //..
val pizza = builder.place("Pizza") //..

builder.step(hotel, park)
builder.step(park, pizza)
builder.step(pizza, hotel)

val tour = builder.build()
```

Repetition

Context Variable Implementation

```
private var previous:PlaceBuilder? = null

fun next(next: PlaceBuilder): TourBuilder {
    connect(previous, next)
    previous = next
    return this
}
```

Builder With a Context Variable

```
//.. places

builder.next(hotel).next(park).next(pizza).next(hotel)
```

Gradle Builds Have Symbols

```
compile(project(":core"))
compile(project(":lib"))
compile(project(":common"))
```

Temporary variables

```
val hotel = builder.place("Hotel") //...
val park = builder.place("Park") //..
val pizza = builder.place("Pizza") //..

builder.step(hotel, park)
builder.step(park, pizza)
builder.step(pizza, hotel)

val tour = builder.build()
```

Builder with Symbols

```
builder.place("Hotel").asHotel().star(4)
builder.place("Pizza").asRestaurant().cuisine("Italian")
builder.place("Park").asPark()

builder.step("Hotel", "Park")
builder.step("Park", "Pizza")
builder.step("Pizza", "Hotel")
```

Benefits of Using Symbols

- No need for temporary variables
- Handle dependencies

DSL Toolbox

~~Function chains~~	~~Symbols~~	@DSLMarker
~~Naming~~	Lambda with receiver	Extension function
~~Nested builders~~	Override property	Infix
~~Context variables~~	Invoke operator	

Where are the Lambdas?

Structured TourBuilder DSL

```
val tour = builder.build() {
    hotel {
        name = "Hotel"
        star(4)
    }
    restaurant {
        cuisine("Italian")
    }
    park {
        hiking()
    }
}
```

{ } provides structure

properties

Functions for TourBuilder

```
class TourBuilder {

 fun build(initialize: TourBuilder.() -> Unit): Tour

 fun hotel(init: HotelBuilder.() -> Unit)
 fun restaurant(init: RestaurantBuilder.() -> Unit)
 fun park(init: ParkBuilder.() -> Unit)

}
```

Build Function

```
class TourBuilder {
 fun build(initialize: TourBuilder.() -> Unit): Tour {
    initialize()
}
```

Lambda with receiver

Implementation using Lambda with Receiver

```
fun restaurant(init: RestaurantBuilder.() -> Unit) {
    val builder = RestaurantBuilder(placeBuilder)
    placeBuilders.add(builder)
    builder.init()
}
```

Delayed execution

Improve Property Setting Code

```
builder.build() {
    hotel {
        star(4)
    }
    restaurant {
        cuisine("Italian")
    }
    park {
        hiking()
    }
}
```

Old Syntax

```
restaurant {
    cuisine("Italian")
}
```

Cleaner Syntax

```
restaurant {
    italian
}
```

Old Syntax

```
restaurant {
    cuisine("Italian")
}
```

Cleaner Syntax

```
restaurant {
    italian
}
```

```
val italian: Boolean
    get() {
        cuisine = "Italian"
        return true
    }
```

Use Invoke Operator to Remove Call to Build

```
val builder = TourBuilder()
val tour = builder.build() {
    // ...
}
```

No call to .build()

Use Invoke Operator to Remove Call to Build

```
val builder = TourBuilder()
val tour = builder {
    // ...
}
```

```
operator fun invoke(init: TourBuilder.() -> Unit): Tour {
    init()
}
```

@DSLMarker

```
hotel {
    star = 4
    hotel {
        star = 5
    }
}
```

```
@DslMarker
annotation class TourDsl

@TourDsl
class TourBuilder

@TourDsl
class HotelBuilder
```

Lambdas, Properties, Invoke, @DSLMarker

Lambdas

- Structure

Properties

- Clean code for inside the lambdas

Invoke

- Options for executing

DSLMarker

- Make scoping work

DSL Toolbox

~~Function chains~~	~~Symbols~~	~~@DSLMarker~~
~~Naming~~	~~Lambda with receiver~~	Extension function
~~Nested builders~~	~~Override property~~	Infix
~~Context variables~~	~~Invoke operator~~	

```kotlin
val places = builder.build() {
    //… places

    step("Hotel", "Park")
    step("Park", "Pizza")
    step("Pizza", "Hotel")
}
```

Is this readable?

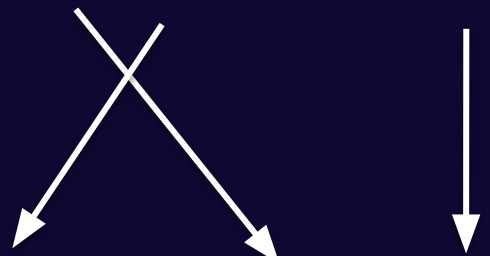
Swap Function Name and Parameter Order

```
assertEquals(2, sum)  // JUnit
```

```
sum.shouldEqual(2)    // kluent
```

Swap Function Name and Parameter Order

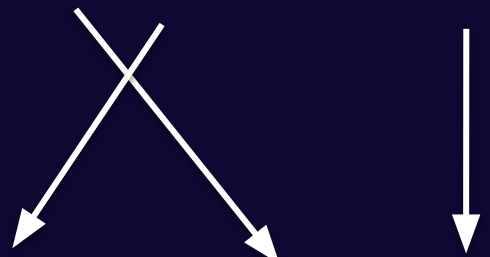
```
step("Hotel", “Park”)
```



```
"Hotel".then("Park")
```

Swap Arguments

```
step("Hotel", “Park”)
```



```
"Hotel".then("Park")
```

```
fun String.then(next: String)
{
    connect(this, next)
}
```

Use Extension

```
val tour = builder.build() {
    // .. places

    "Hotel".then("Park")
    "Park".then("Pizza")
    "Pizza".then("Hotel")

}
```

Use Extension (as infix)

```
val tour = builder.build() {
    // .. places

    "Hotel" then "Park"
    "Park" then "Pizza"
    "Pizza" then "Hotel"

}
```

No () or .

Extensions

- Function name/argument swap creates readable code
- Be careful of scope

DSL Toolbox

Function chains	Symbols	@DSLMarker
Naming	Lambda with receiver	Extension function
Nested builders	Override property	Infix
Context variables	Invoke operator	

No DSL

```
val hotel =
 Hotel("Hotel", 4)
val restaurant =
 Restaurant("Pizza", "Italian")

val tour = Tour()
tour.step(hotel, restaurant)
tour.step(restaurant, park)
tour.step(park, hotel)
```

DSL

```
val tour = build {
    hotel {
        name = "Hotel"
        star = 4
    }
    restaurant {
        name = "Pizza"
        italian
    }

    "Hotel" then "Park"
    "Park" then "Pizza"
    "Pizza" then "Hotel"
}
```

DSL Toolbox

Function chains	Symbols	@DSLMarker
Naming	Lambda with receiver	Extension function
Nested builders	Override property	Infix
Context variables	Invoke operator	

Extension Function for Function Name/Argument Swap

```
toLatLon(location)          // util function style

location.asLatLon()         // readable
```

Function Name/Argument Swap

```
class MapWrapper {
    // ..
    private fun Location.asLatLon(): LatLon {
        return LatLon(latitude, longitude)
    }
}
```

private extension function improve syntax

DSL Toolbox

Function chains	Symbols	@DSLMarker
Naming	Lambda with receiver	Extension function
Nested builders	Override property	Infix
Context variables	Invoke operator	

Sooo Many Parameters

```
private fun showDetail(
    fragmentManager: FragmentManager,
    tag: String,
    id: Int,
    slideIn: Boolean = false,
    slideOut: Boolean = false,
    fadeIn: Boolean = false,
    fadeOut: Boolean = false,
    addToBackStack: Boolean = false,
    backStackTag: String = tag,
    container: Int = R.id.fragment_container)
```

How Improve this with a Lambda

- Capture variation in arguments in a lambda
- Delay execution

Delayed Execution

```
// preprocessing
val transaction = fragmentManager.beginTransaction()
// execute the lambda
transaction.transactionBlock()
// post processing
transaction.show(fragmentToPlace)
transaction.commit()
```

Execute lambda

Pass argument to Lambda

```
private fun showDetail(
    fragmentManager: FragmentManager,
    id: Int,
    transactionBlock: (FragmentTransaction) -> Unit)


showDetail(fragmentManager, id) {
    it.setCustomAnimations(android.R.anim.fade_in,
                           android.R.anim.fade_out)
    it.addToBackStack("hotel")
}
```

Lambda with Receiver

```
private fun showDetail(
    fragmentManager: FragmentManager,
    id: Int,
    transactionBlock: FragmentTransaction.() -> Unit)


showDetail(fragmentManager, id) {
    setCustomAnimations(android.R.anim.fade_in,
                        android.R.anim.fade_out)
    addToBackStack("hotel")
}
```

Custom Builder as Receiver

```
private fun showDetail(
    fragmentManager: FragmentManager,
    id: Int,
    transactionBlock: TransactionBuilder.() -> Unit)


showDetail(fragmentManager, id) {
    slideIn
    backStackTag = "hotel"
}
```

Changes to Our Function

```
slideIn: Boolean = false //and other params
```

```
transactionBlock: (FragmentTransaction) -> Unit
```

```
transactionBlock: FragmentTransaction.() -> Unit
```

```
transactionBlock: TransactionBuilder.() -> Unit
```

DSL Toolbox

Function chains	Symbols	@DSLMarker
Naming	Lambda with receiver	Extension function
Nested builders	Override property	Infix
Context variables	Invoke operator	

Resources

Books

- Domain Specific Language, (Fowler, Parsons) https://martinfowler.com/books/dsl.html
- Kotlin in Action (Dmitry Jemerov and Svetlana Isakova)

Presentations

- KotlinConf 2018 - Creating Internal DSLs in Kotlin by Venkat Subramaniam: https://www.youtube.com/watch?v=JzTeAM8N1-o

Github

- Village DSL: https://github.com/zsmb13/VillageDSL
- Castle builder DSL: https://github.com/gmilette/kotlin-castle-dsl/tree/master/src/dsl/castlebuilder

Courses

- Domain Specific Languages in Kotlin (Pluralsight)
  https://www.pluralsight.com/courses/kotlin-fundamentals-domain-specific-languages

Shutterstock | Aphelleon