

# Kotlin

Proseminar: Fortgeschrittene Programmierkonzepte

Christian Konersmann, Finn Paul Lippok, Paul Lukas

05.05.2025

# Was ist Kotlin?

- **Statisch typisierte** und **objektorientierte** Programmiersprache.
- **Basierend auf Java und der JVM** mit vollständiger **Interoperabilität** zu beiden.



# Was ist Kotlin?

- **Statisch typisierte** und **objektorientierte** Programmiersprache.
- **Basierend auf Java und der JVM** mit vollständiger **Interoperabilität** zu beiden.
- **Wichtigste Vorteile gegenüber Java:**
  - Klare und präzise Syntax.
  - Erweiterte Funktionen wie Null-Sicherheit.
  - Umfassende Multiplattform-Entwicklungsmöglichkeiten.



## Java Main-Methode

```
1 public class Main {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, World!");  
4     }  
5 }
```

## Java Main-Methode

```
1 public class Main {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, World!");  
4     }  
5 }
```

## Kotlin Main-Methode

```
1 fun main() {  
2     println("Hello, World!")  
3 }
```

## Java Main-Methode

```
1 public class Main {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, World!");  
4     }  
5 }
```

## Kotlin Main-Methode

```
1 fun main() {  
2     println("Hello, World!")  
3 }
```

- Keine explizite Klassendeklaration erforderlich.

## Java Main-Methode

```
1 public class Main {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, World!");  
4     }  
5 }
```

## Kotlin Main-Methode

```
1 fun main() {  
2     println("Hello, World!")  
3 }
```

- Keine explizite Klassendeklaration erforderlich.
- Verwendung des Schlüsselworts `fun` zur Funktionsdeklaration.

## Java Main-Methode

```
1 public class Main {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, World!");  
4     }  
5 }
```

## Kotlin Main-Methode

```
1 fun main() {  
2     println("Hello, World!")  
3 }
```

- Keine explizite Klassendeklaration erforderlich.
- Verwendung des Schlüsselworts `fun` zur Funktionsdeklaration.
- Standardzugriffsmodifikator ist `public`.



## Java Main-Methode

```
1 public class Main {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, World!");  
4     }  
5 }
```

## Kotlin Main-Methode

```
1 fun main() {  
2     println("Hello, World!")  
3 }
```

- Keine explizite Klassendeklaration erforderlich.
- Verwendung des Schlüsselworts `fun` zur Funktionsdeklaration.
- Standardzugriffsmodifikator ist `public`.
- `args`-Parameter ist optional.

## Java Main-Methode

```
1 public class Main {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, World!");  
4     }  
5 }
```

## Kotlin Main-Methode

```
1 fun main() {  
2     println("Hello, World!")  
3 }
```

- Keine explizite Klassendeklaration erforderlich.
- Verwendung des Schlüsselworts `fun` zur Funktionsdeklaration.
- Standardzugriffsmodifikator ist `public`.
- `args`-Parameter ist optional.
- Semikolons sind nicht erforderlich.

# Variablen-Deklaration

## Java

```
1 int a = 5;  
2 final String b = "Hallo";
```

## Kotlin

```
var a: Int = 5  
val b: String = "Hallo"
```

- `var` für veränderliche Variablen, `val` für unveränderliche Variablen.

# Variablen-Deklaration

## Java

```
1 int a = 5;  
2 final String b = "Hallo";
```

## Kotlin

```
var a: Int = 5  
val b: String = "Hallo"
```

- `var` für veränderliche Variablen, `val` für unveränderliche Variablen.
- Typangabe nach dem Variablennamen mit Doppelpunkt.

# Variablen-Deklaration

## Java

```
1 int a = 5;  
2 final String b = "Hallo";
```

## Kotlin

```
var a: Int = 5  
val b: String = "Hallo"
```

- `var` für veränderliche Variablen, `val` für unveränderliche Variablen.
- Typangabe nach dem Variablennamen mit Doppelpunkt.
- In Kotlin gibt es **keine** primitiven Typen.

# Variablen-Deklaration

## Java

```
1 int a = 5;  
2 final String b = "Hallo";
```

## Kotlin

```
var a: Int = 5  
val b: String = "Hallo"
```

- `var` für veränderliche Variablen, `val` für unveränderliche Variablen.
- Typangabe nach dem Variablennamen mit Doppelpunkt.
- In Kotlin gibt es **keine** primitiven Typen.

# Variablen-Deklaration

## Java

```
1 int a = 5;  
2 final String b = "Hallo";
```

## Kotlin

```
var a: Int = 5  
val b: String = "Hallo"
```

- `var` für veränderliche Variablen, `val` für unveränderliche Variablen.
- Typangabe nach dem Variablennamen mit Doppelpunkt.
- In Kotlin gibt es **keine** primitiven Typen.

Kotlin unterstützt **Typinferenz**, d.h. der Typ kann weggelassen werden.

- Der Compiler leitet den Typ aus dem initialisierten Wert ab.
- Beispiel: `var a = 5` ist auch möglich.

## Java

```
1 public class Verkaufsperson {  
2     public final String name;  
3     private double provision;  
4  
5     public Verkaufsperson (String name, double  
6         provision) {...}  
}
```



## Java

```
1 public class Verkaufsperson {  
2     public final String name;  
3     private double provision;  
4  
5     public Verkaufsperson (String name, double  
6         provision) {...}  
7 }
```

## Kotlin

```
1 class Verkaufsperson() {  
2  
3  
4  
5     val name: String  
6     private var provision: Double  
7 }
```

## Java

```
1 public class Verkaufsperson {  
2     public final String name;  
3     private double provision;  
4  
5     public Verkaufsperson (String name, double  
6         provision) {...}  
7 }
```

## Kotlin

```
1 class Verkaufsperson(  
2     name: String,  
3     provision: Double = 0.2  
4 ) {  
5     val name: String = name  
6     private var provision: Double = provision  
7 }
```

## Java

```
1 public class Verkaufsperson {  
2     public final String name;  
3     private double provision;  
4  
5     public Verkaufsperson (String name, double  
6         provision) {...}  
7 }
```

## Kotlin

```
1 class Verkaufsperson(  
2     val name: String,  
3     private var provision: Double = 0.2  
4 ) {  
5  
6  
7 }
```

## Java

```
1 public class Verkaufsperson {  
2     public final String name;  
3     private double provision;  
4  
5     public Verkaufsperson (String name, double  
6         provision) {...}  
7 }
```

## Kotlin

```
1 class Verkaufsperson(  
2     val name: String,  
3     private var provision: Double = 0.2  
4 ) {}
```

- Ähnlich wie Java-Records, aber flexibler.
- Nur vererbbar, wenn als open deklariert.

## Kotlin: Properties

```
1 class Verkaufsperson(val name: String,  
2     private var provision: Double = 0.2) {  
3  
4     var umsatz : Int = 0  
5  
6  
7  
8  
9  
10  
11 }
```

## Kotlin: Properties Zugriffsmodifikator

```
1 class Verkaufsperson(val name: String,  
2     private var provision: Double = 0.2) {  
3  
4     var umsatz : Int = 0  
5     private set  
6  
7  
8  
9  
10  
11 }
```

## Kotlin: Benutzerdefinierte Zugriffsmethoden

```
1 class Verkaufsperson(val name: String,  
2     private var provision: Double = 0.2) {  
3  
4     var umsatz : Int = 0  
5     private set(value) {  
6         if (value < 0)  
7             throw IllegalArgumentException("Umsatz_muss_  
8                 positiv_sein")  
9         field = value  
10    }  
}
```

## Kotlin: Benutzerdefinierte Zugriffsmethoden

```
1 class Verkaufsperson(val name: String,  
2     private var provision: Double = 0.2) {  
3  
4     var umsatz : Int = 0  
5     private set(value) {  
6         if (value < 0)  
7             throw IllegalArgumentException("Umsatz_muss_  
8                 positiv_sein")  
9         field = value  
10    }  
}
```

- Punkt-Notation ruft automatisch Setter/Getter auf.
- Beispiel: `verkaufsperson.umsatz = -1` wirft eine `IllegalArgumentException`.



## Motivation: Null Safety

## Motivation: Null Safety

### Java Beispiel

```
1 Verkaufsperson person = null;  
2 System.out.println(person.name);
```

## Motivation: Null Safety

### Java Beispiel

```
1 Verkaufsperson person = null;  
2 System.out.println(person.name);
```

- Code wirft `java.lang.NullPointerException`

## Motivation: Null Safety

### Java Beispiel

```
1 Verkaufsperson person = null;  
2 System.out.println(person.name);
```

- Code wirft `java.lang.NullPointerException`
- Kann zu Programmabbruch führen oder weitere Fehler nach sich ziehen

# Null Safety

```
var a : String = "a_is_non-nullable"  
var b : String? = "b_is_nullable"
```

# Null Safety

```
var a : String = "a_ist_non-nullable"  
var b : String? = "b_ist_nullable"
```

- Unterscheidung zwischen *nullable* und *non-nullable* types
- Programmierer muss Null safety gewährleisten

# Null Safety: Safe call Operator

Ziel: sicherer Zugriff auf Datenfelder und Methoden durch ?.

# Null Safety: Safe call Operator

Ziel: sicherer Zugriff auf Datenfelder und Methoden durch ?.

in Java

```
private final Verkaufsperson vorgesetzer;  
  
public void printVorgesetzer() {  
    if (vorgesetzer == null) System.out.println(null);  
    else System.out.println(vorgesetzer.name);  
}
```



# Null Safety: Safe call Operator

Ziel: sicherer Zugriff auf Datenfelder und Methoden durch ?.

in Java

```
private final Verkaufsperson vorgesetzer;  
  
public void printVorgesetzer() {  
    if (vorgesetzer == null) System.out.println(null);  
    else System.out.println(vorgesetzer.name);  
}
```

in Kotlin

```
val vorgesetzer: Verkaufsperson? = null  
  
fun printVorgesetzer() {  
    println(vorgesetzer?.name)  
}
```

## Verkettung des Operators

```
val name: String? = vorgesetzer?.vorgesetzer?.name
```

## Verkettung des Operators

```
val name: String? = vorgesetzer?.vorgesetzer?.name
```

## Zuweisungen mit dem Operator

```
vorgesetzer?.vorgesetzer?.provision = 0.0
```

- Weiterentwicklung des Safe call Operators

# Null Safety: Elvis Operator

- Weiterentwicklung des Safe call Operators
- Ermöglicht setzen von Default-Werten anstelle *null*

# Null Safety: Elvis Operator

- Weiterentwicklung des Safe call Operators
- Ermöglicht setzen von Default-Werten anstelle *null*

# Null Safety: Elvis Operator

- Weiterentwicklung des Safe call Operators
- Ermöglicht setzen von Default-Werten anstelle *null*

```
public void printVorgesetzter() {  
    if (vorgesetzter == null)  
        System.out.println("Kein_Vorgesetzter");  
    else System.out.println(vorgesetzter.name);  
}
```

# Null Safety: Elvis Operator

- Weiterentwicklung des Safe call Operators
- Ermöglicht setzen von Default-Werten anstelle *null*

```
public void printVorgesetzer() {  
    if (vorgesetzer == null)  
        System.out.println("Kein_Vorgesetzer");  
    else System.out.println(vorgesetzer.name);  
}
```

```
fun printVorgesetzer() {  
    println(vorgesetzer?.name ?: "Kein_Vorgesetzer")  
}
```



# Null Safety: Not-null assertion Operator

```
val a: String? = null  
var b: String = possiblyNull!!
```

# Null Safety: Not-null assertion Operator

```
val a: String? = null  
var b: String = possiblyNull!!
```

- Kann zu `NullPointerException`s führen

# Null Safety: Nullable Receiver

- Baut auf Extension functions

# Null Safety: Nullable Receiver

- Baut auf Extension functions
- erlauben Methodenaufruf auf nullable types

# Null Safety: Nullable Receiver

- Baut auf Extension functions
- erlauben Methodenaufruf auf nullable types
- Null Werte werden innerhalb der Methode behandelt

# Null Safety: Nullable Receiver

- Baut auf Extension functions
- erlauben Methodenaufruf auf nullable types
- Null Werte werden innerhalb der Methode behandelt

# Null Safety: Nullable Receiver

- Baut auf Extension functions
- erlauben Methodenaufruf auf nullable types
- Null Werte werden innerhalb der Methode behandelt

```
fun Verkaufsperson?.print() {  
    if (this == null) return println("Diese Person  
        existiert nicht")  
    return println("$name: $provision Anteil")  
}
```

# Null Safety: Nullable Receiver

- Baut auf Extension functions
- erlauben Methodenaufruf auf nullable types
- Null Werte werden innerhalb der Methode behandelt

```
fun Verkaufsperson?.print() {  
    if (this == null) return println("Diese Person  
        existiert nicht")  
    return println("$name: $provision Anteil")  
}
```

```
var sales: Verkaufsperson? = null  
sales.print()
```



```
1 public class Verkaufsperson {  
2     private final String name;  
3     private double provision;  
4  
5     public Verkaufsperson(String name, double  
6         provision) {...}  
7  
8     public String getName() {...}  
9     public double getProvision() {...}  
0     public void setProvision(double provision) {...}  
1 }  
2  
3  
4  
5  
6  
7  
8  
9  
0
```

# Interoperabilität

```
public class Verkaufsperson {  
    private final String name;  
    private double provision;  
  
    public Verkaufsperson(String name, double  
        provision) {...}  
  
    public String getName() {...}  
    public double getProvision() {...}  
    public void setProvision(double provision) {...}  
}
```

```
var carl = Verkaufsperson("Carl_Mueller", 0.1)  
println(carl.name)  
carl.provision = 0.2
```

```
public class Verkaufsperson {  
    private final String name;  
    private double provision;  
  
    public Verkaufsperson(String name, double  
        provision) {...}  
  
    public String getName() {...}  
    public double getProvision() {...}  
    public void setProvision(double provision) {...}  
}
```

```
var carl = Verkaufsperson("Carl_Mueller", 0.1)  
println(carl.name)  
carl.provision = 0.2
```

- Kotlin erstellt synthetic properties
- Aufruf über getter/setter Methoden weiterhin möglich

- normalerweise werden die java-Typen übernommen

# Interoperabilität: Mapped types

- normalerweise werden die java-Typen übernommen
- Manche haben einen zugehörige Kotlin Typ

# Interoperabilität: Mapped types

- normalerweise werden die java-Typen übernommen
- Manche haben einen zugehörige Kotlin Typ

# Interoperabilität: Mapped types

- normalerweise werden die java-Typen übernommen
- Manche haben einen zugehörige Kotlin Typ
- `java.lang.Object`  $\Rightarrow$  `kotlin.Any`!

# Interoperabilität: Mapped types

- normalerweise werden die java-Typen übernommen
- Manche haben einen zugehörige Kotlin Typ
- `java.lang.Object`  $\Rightarrow$  `kotlin.Any!`
- `java.lang.Integer`  $\Rightarrow$  `kotlin.Int?`



# Interoperabilität: Mapped types

- normalerweise werden die java-Typen übernommen
- Manche haben einen zugehörige Kotlin Typ
- `java.lang.Object`  $\Rightarrow$  `kotlin.Any!`
- `java.lang.Integer`  $\Rightarrow$  `kotlin.Int?`
- primitive typ `int`  $\Rightarrow$  `kotlin.Int`

# Interoperabilität: Mapped types

- normalerweise werden die java-Typen übernommen
- Manche haben einen zugehörige Kotlin Typ
- `java.lang.Object`  $\Rightarrow$  `kotlin.Any!`
- `java.lang.Integer`  $\Rightarrow$  `kotlin.Int?`
- primitiv typ `int`  $\Rightarrow$  `kotlin.Int`
- Rückgabewert `void`  $\Rightarrow$  `Unit`

# Interoperabilität: Null safety mit Java

```
public Verkaufsperson erstellePerson() {  
    return null;  
}
```

# Interoperabilität: Null safety mit Java

```
public Verkaufsperson erstellePerson() {  
    return null;  
}
```

```
val person: Verkaufsperson = erstellePerson()  
println(person.name)
```

# Interoperabilität: Null safety mit Java

```
public Verkaufsperson erstellePerson() {  
    return null;  
}
```

```
val person: Verkaufsperson = erstellePerson()  
println(person.name)
```

- Aus Java zurückgegebene Instanzen können *null* sein

# Interoperabilität: Null safety mit Java

```
public Verkaufsperson erstellePerson() {  
    return null;  
}
```

```
val person: Verkaufsperson = erstellePerson()  
println(person.name)
```

- Aus Java zurückgegebene Instanzen können *null* sein
- haben spezial-Typ: *platform type*

# Interoperabilität: Null safety mit Java

```
public Verkaufsperson erstellePerson() {  
    return null;  
}
```

```
val person: Verkaufsperson = erstellePerson()  
println(person.name)
```

- Aus Java zurückgegebene Instanzen können *null* sein
- haben spezial-Typ: *platform type*
- gelockerte Regeln bezüglich Null safety

# Interoperabilität: Null safety mit Java

```
public Verkaufsperson erstellePerson() {  
    return null;  
}
```

```
val person: Verkaufsperson = erstellePerson()  
println(person.name)
```

- Aus Java zurückgegebene Instanzen können *null* sein
- haben spezial-Typ: *platform type*
- gelockerte Regeln bezüglich Null safety
- anfälliger für NullPointerExceptions



- Es gibt keine primitiven Typen in Kotlin

# Interoperabilität: Java Arrays in Kotlin

- Es gibt keine primitiven Typen in Kotlin
- Extra Kotlin Klassen für primitive Arrays

- Es gibt keine primitiven Typen in Kotlin
- Extra Kotlin Klassen für primitive Arrays
- Diese kompilieren zu primitiven Arrays

# Interoperabilität: Java Arrays in Kotlin

- Es gibt keine primitiven Typen in Kotlin
- Extra Kotlin Klassen für primitive Arrays
- Diese kompilieren zu primitiven Arrays

# Interoperabilität: Java Arrays in Kotlin

- Es gibt keine primitiven Typen in Kotlin
- Extra Kotlin Klassen für primitive Arrays
- Diese kompilieren zu primitiven Arrays

```
public static void takeArray(int[] array) { ... }
```

# Interoperabilität: Java Arrays in Kotlin

- Es gibt keine primitiven Typen in Kotlin
- Extra Kotlin Klassen für primitive Arrays
- Diese kompilieren zu primitiven Arrays

```
public static void takeArray(int[] array) { ... }
```

```
var array: IntArray = intArrayOf(1, 2, 3)  
takeArray(array)
```

# Interoperabilität: Kotlin Properties in Java

```
var name: String
```

# Interoperabilität: Kotlin Properties in Java

```
var name: String
```

```
private String name;  
public String getName() { return name; }  
public void setName(String name) { this.name = name; }
```



# Interoperabilität: Instance Fields

```
Verkaufsperson person =  
    new Verkaufsperson("carl", 0.0);  
System.out.println(person.name);
```

```
Verkaufsperson person =  
    new Verkaufsperson("carl", 0.0);  
System.out.println(person.name);
```

- Aufruf von Attributen durch Punkt-Notation

```
Verkaufsperson person =  
    new Verkaufsperson("carl", 0.0);  
System.out.println(person.name);
```

- Aufruf von Attributen durch Punkt-Notation
- In Java ist der Aufruf durch die @JvmField Annotation möglich

```
Verkaufsperson person =  
    new Verkaufsperson("carl", 0.0);  
System.out.println(person.name);
```

- Aufruf von Attributen durch Punkt-Notation
- In Java ist der Aufruf durch die @JvmField Annotation möglich

# Interoperabilität: Instance Fields

```
Verkaufsperson person =  
    new Verkaufsperson("carl", 0.0);  
System.out.println(person.name);
```

- Aufruf von Attributen durch Punkt-Notation
- In Java ist der Aufruf durch die @JvmField Annotation möglich

```
class Verkaufsperson(@JvmField var name:String) {}
```

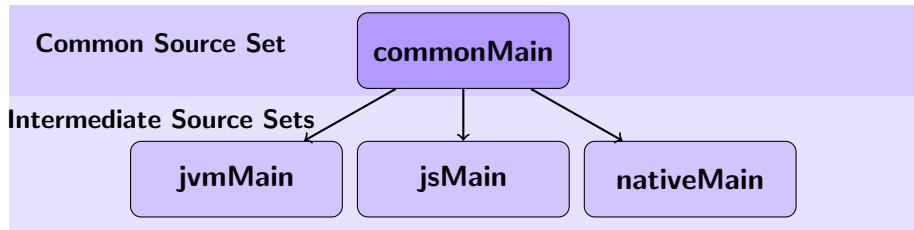
- native binarie
- verschiedene Targets
- hierarchische Projektstruktur

# Multiplatform: hierarchische Projektstruktur

**Common Source Set**

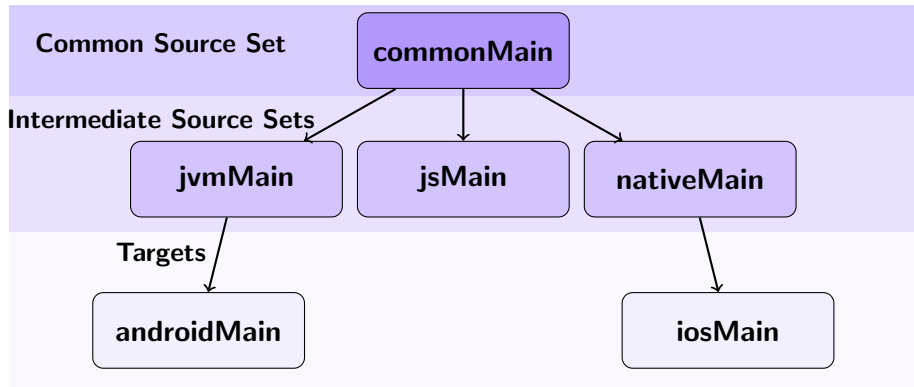
**commonMain**

# Multiplatform: hierarchische Projektstruktur

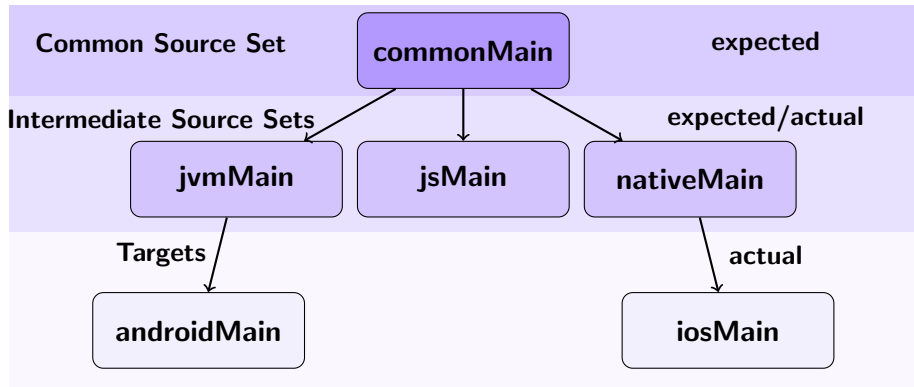




# Multiplatform: hierarchische Projektstruktur



# Multiplatform: hierarchische Projektstruktur



- Jetpack compose
- Coroutines
- Beispiel

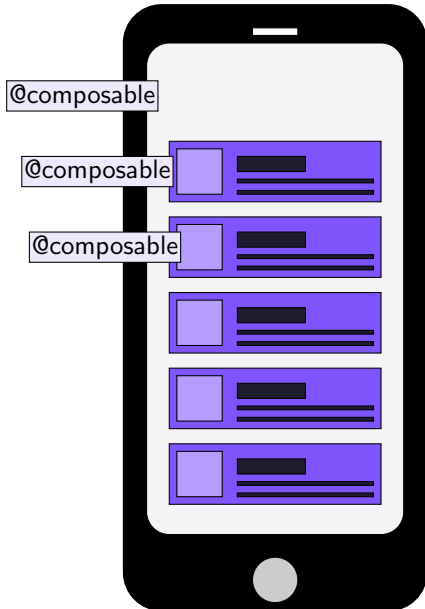
# Android: Jetpack compose

- Ui Tool
- @composables
- Kotlin basiert

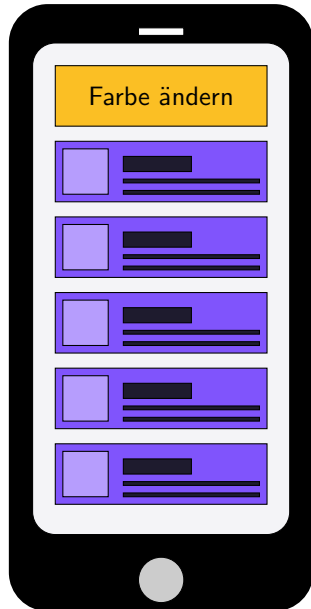
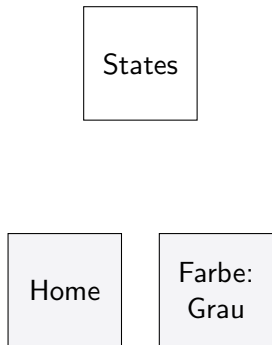
# Android: Composable: Beispiel

States

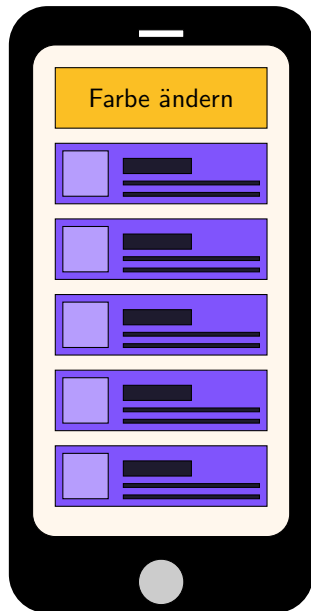
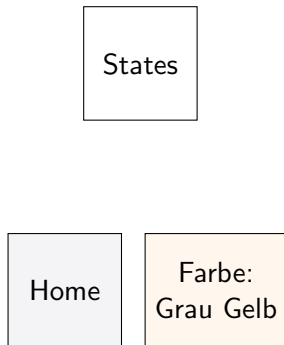
Home



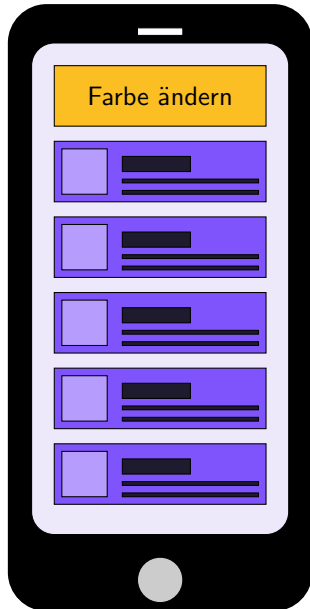
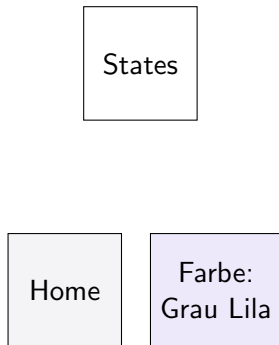
# Android: Composable: Beispiel



# Android: Composable: Beispiel

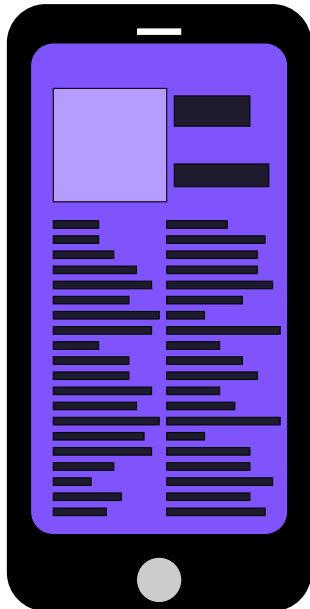
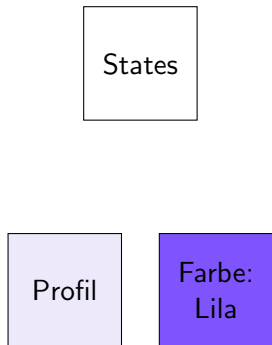


# Android: Composable: Beispiel

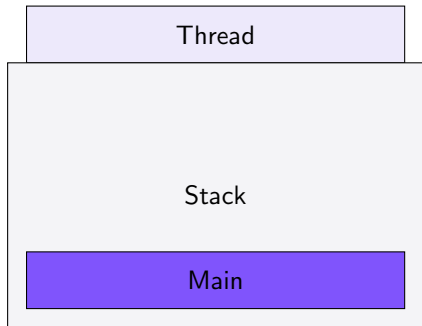




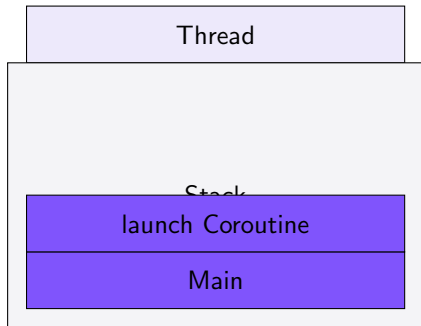
# Android: Composable: Beispiel



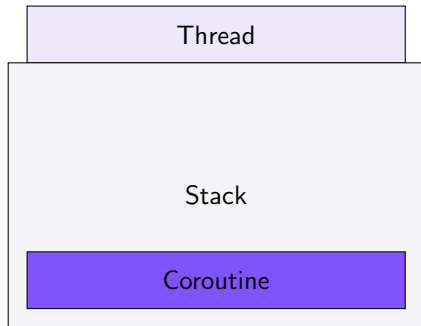
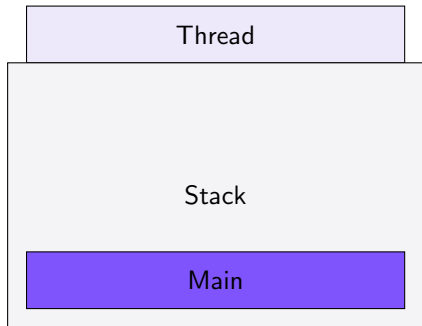
# Android: Coroutines



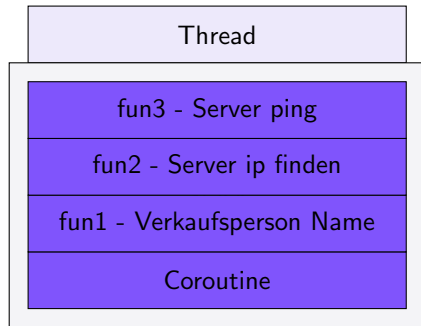
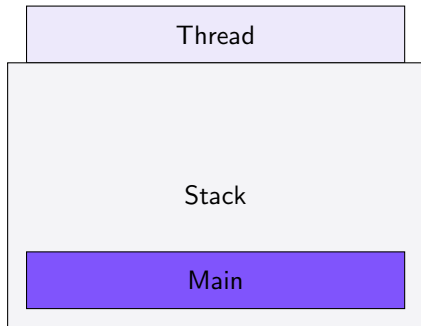
# Android: Coroutines



# Android: Coroutines

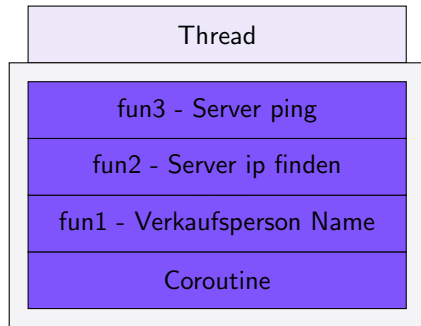
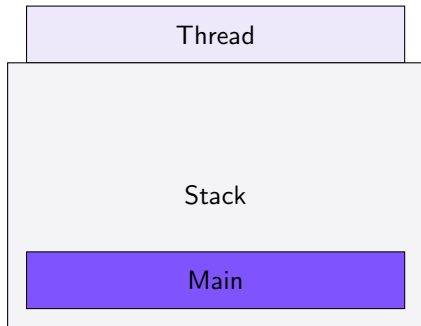


# Android: Coroutines



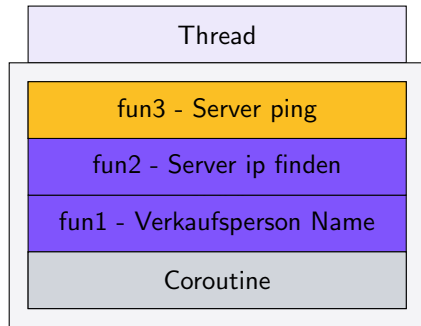
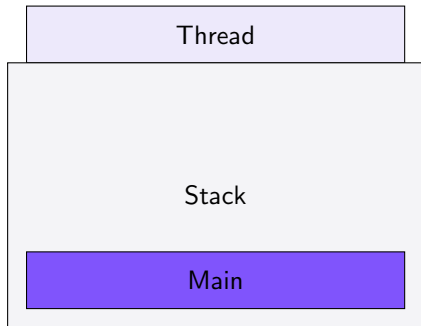
# Android: Coroutines

Suspends: `delay()`, `yield()`, `withContext()`, ...

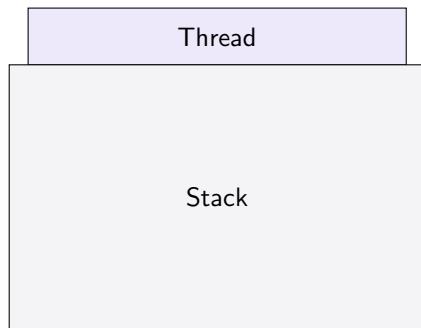
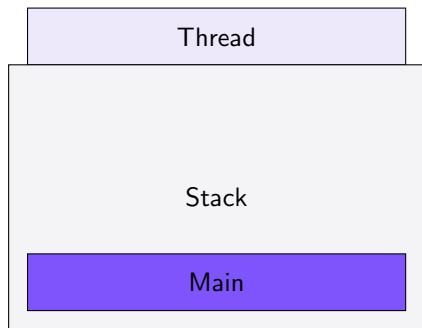


# Android: Coroutines

Suspends: `delay()`, `yield()`, `withContext()`, ...



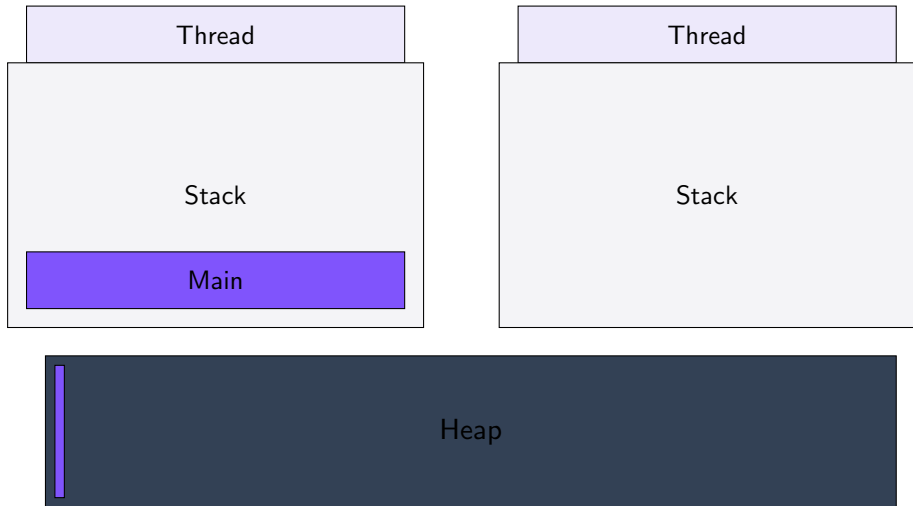
# Android: Coroutines



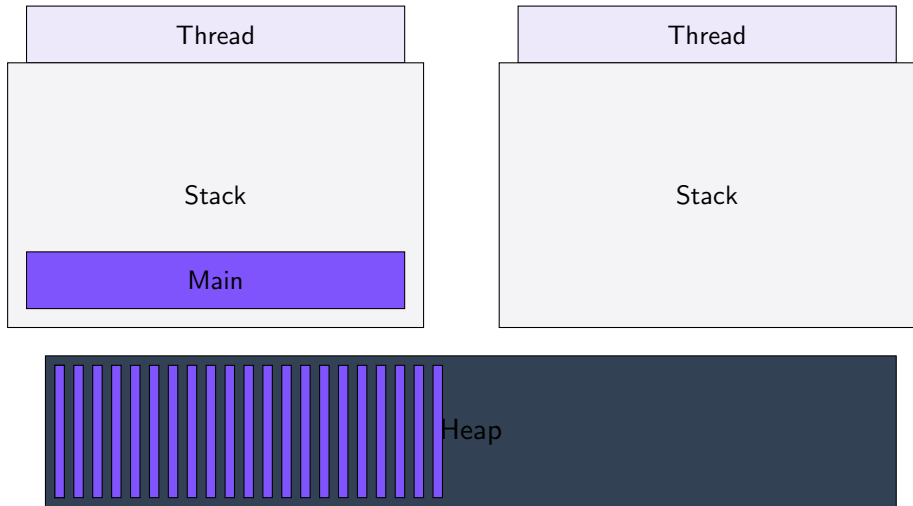
```
Coroutine:  
Variablen:      Name,lp  
State:          1  
Path:           fun1-fun2-fun3
```



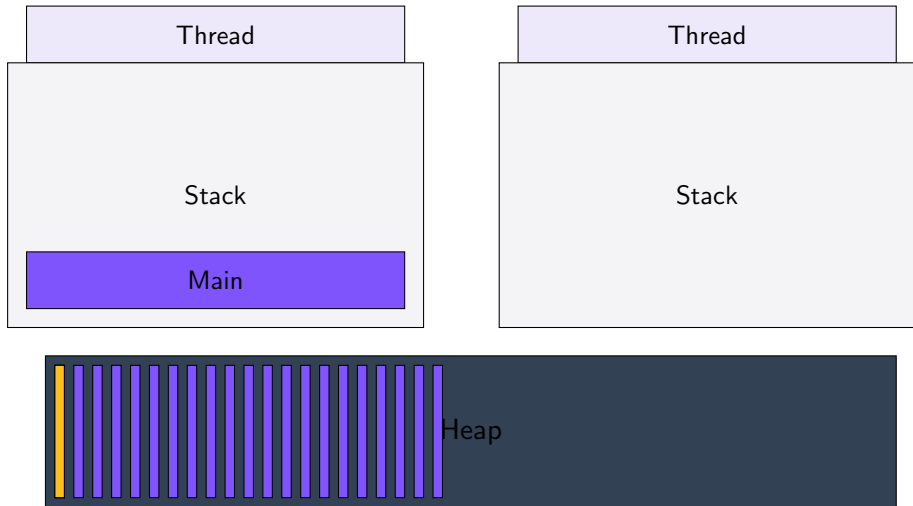
# Android: Coroutines



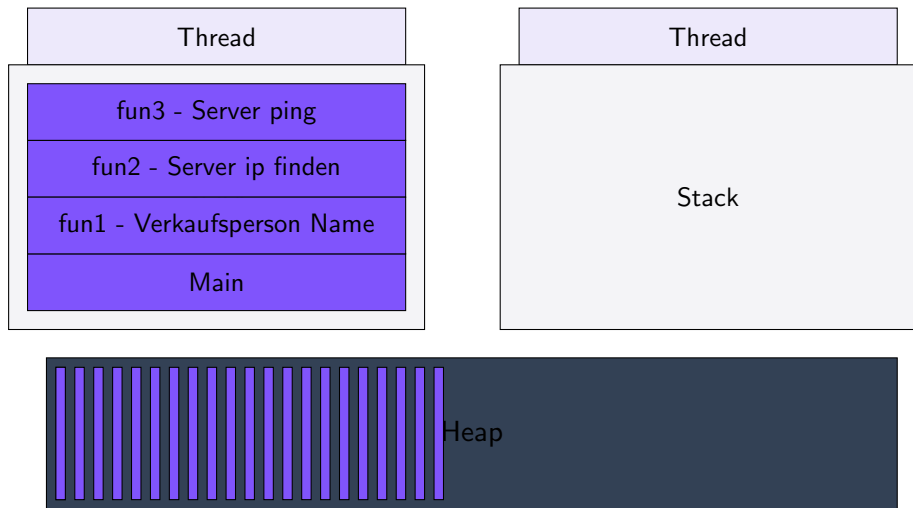
# Android: Coroutines



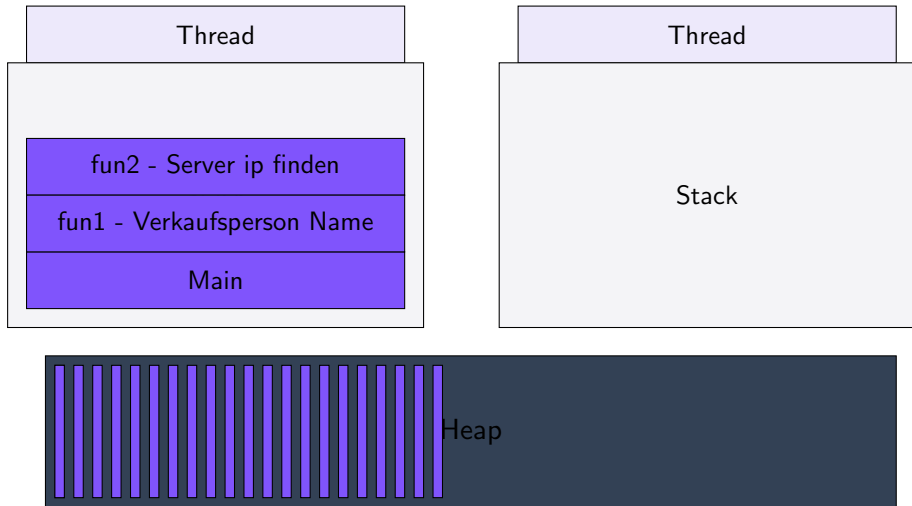
# Android: Coroutines



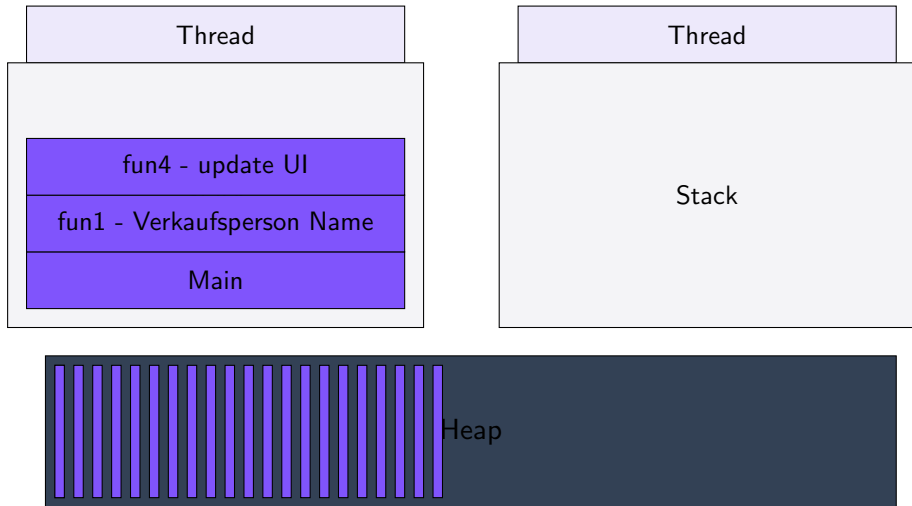
# Android: Coroutines



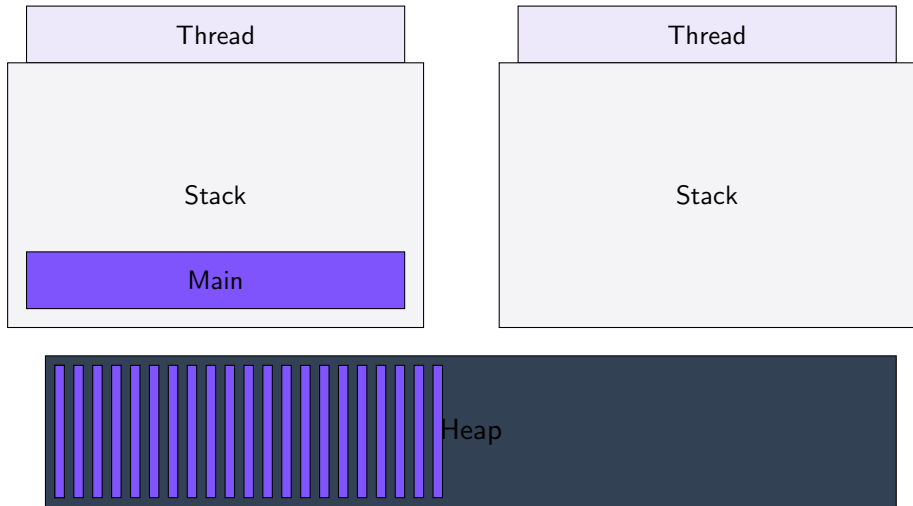
# Android: Coroutines



# Android: Coroutines



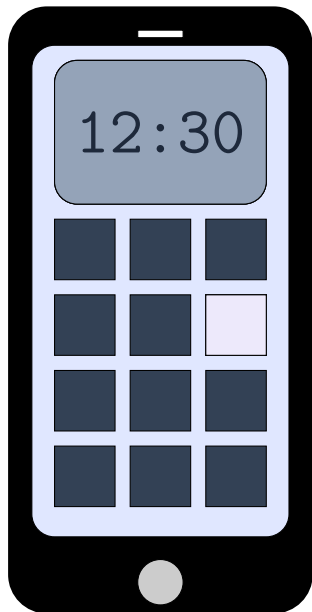
# Android: Coroutines



## Beispiel Coroutines

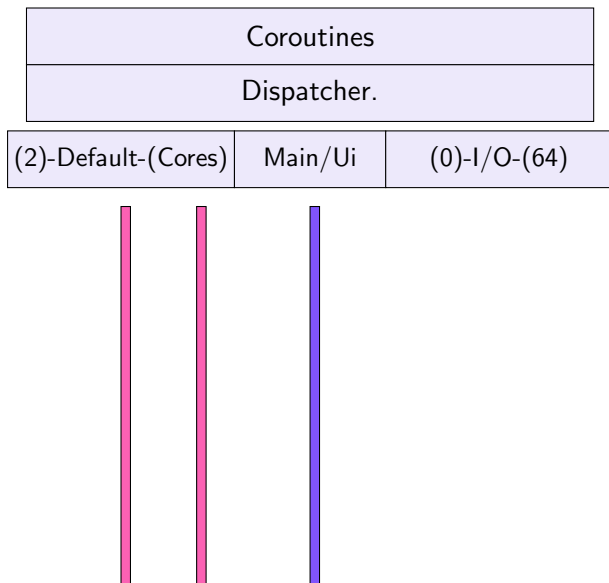
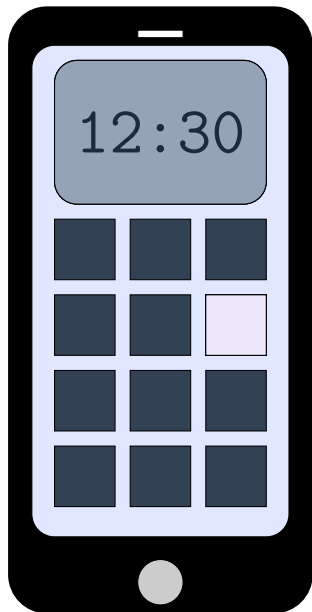


# Android: Coroutines: Beispiel



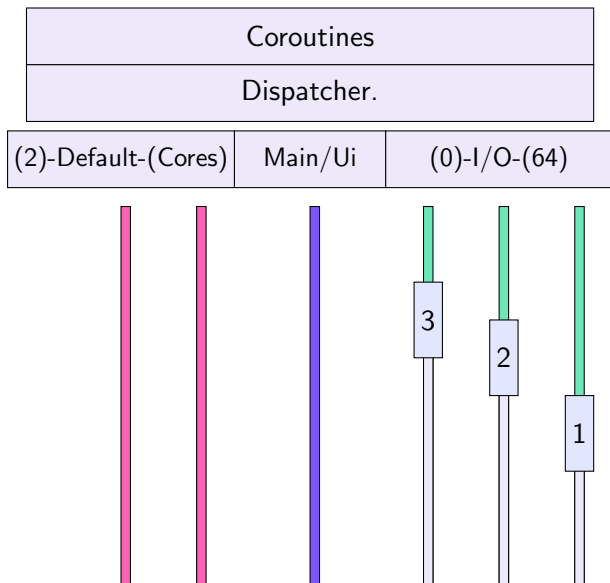
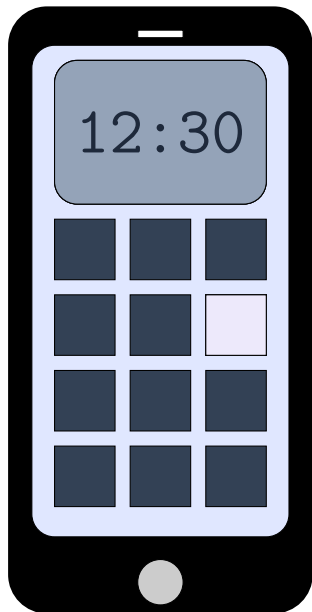
Coroutines

# Android: Coroutines: Beispiel

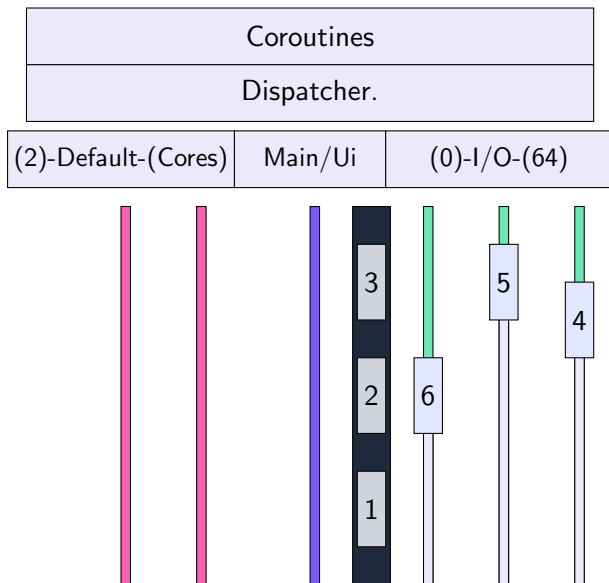
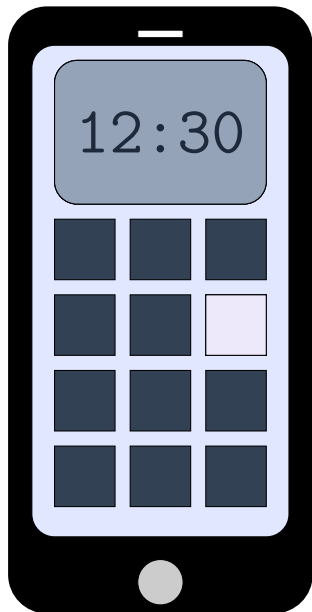




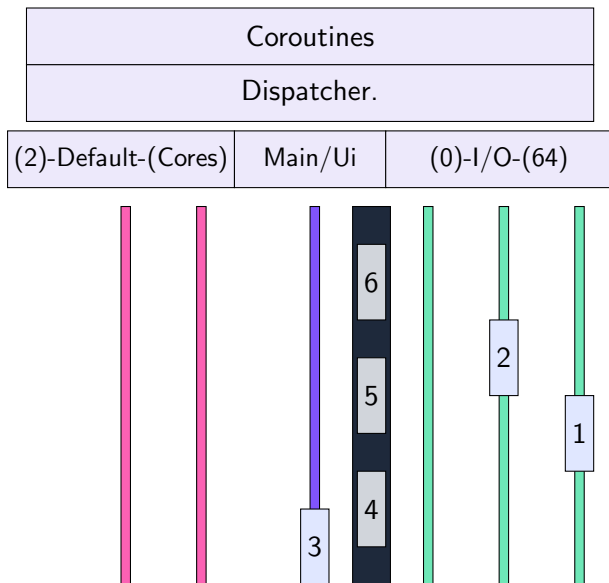
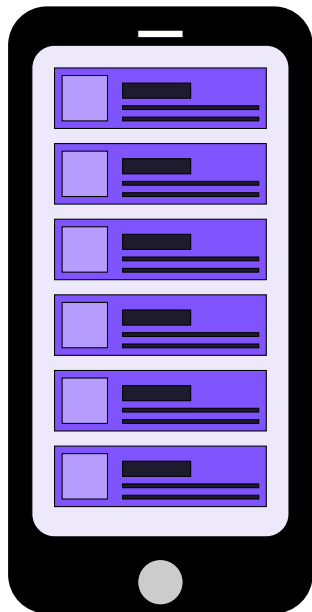
# Android: Coroutines: Beispiel



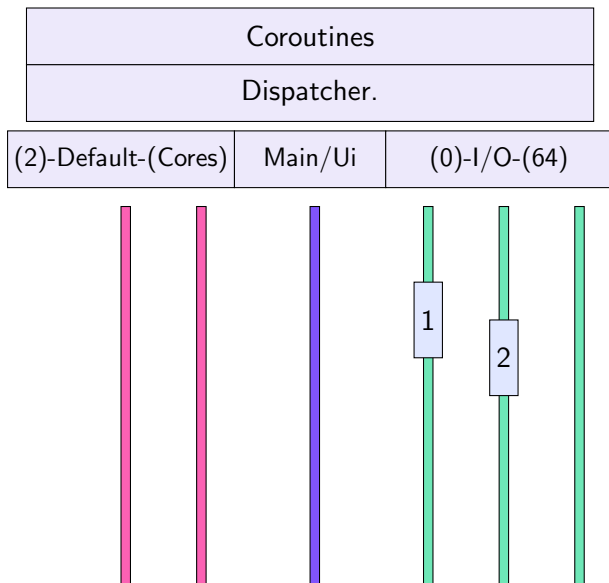
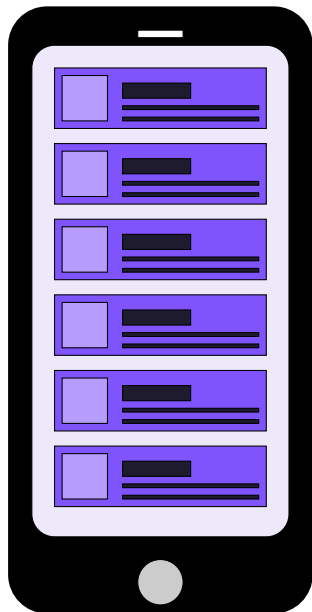
# Android: Coroutines: Beispiel



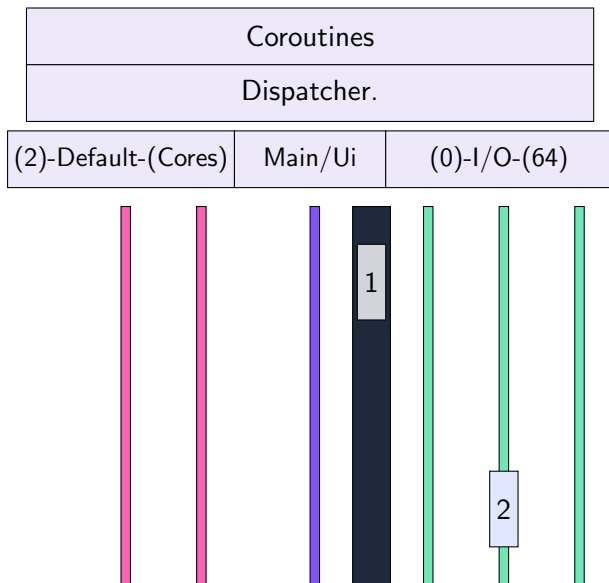
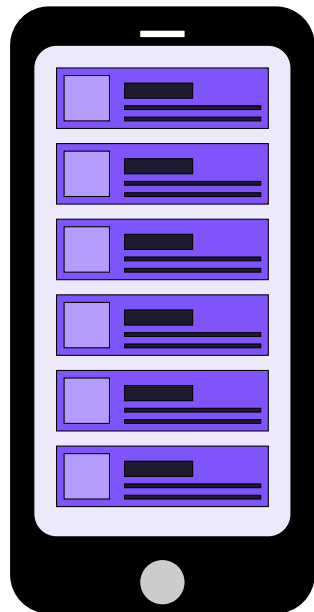
# Android: Coroutines: Beispiel



# Android: Coroutines: Beispiel

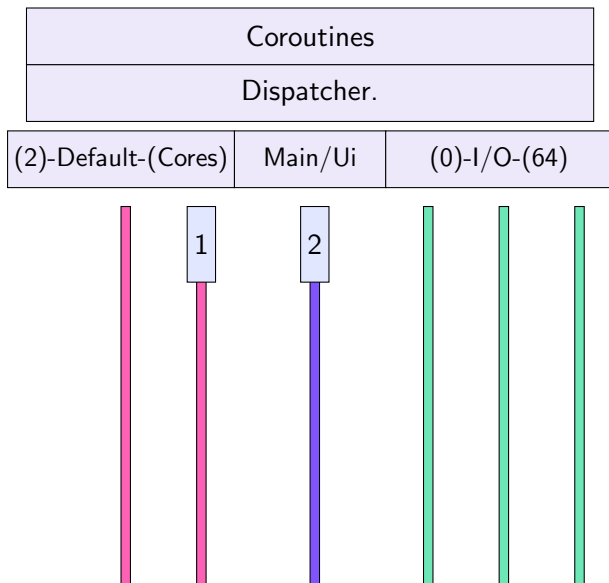
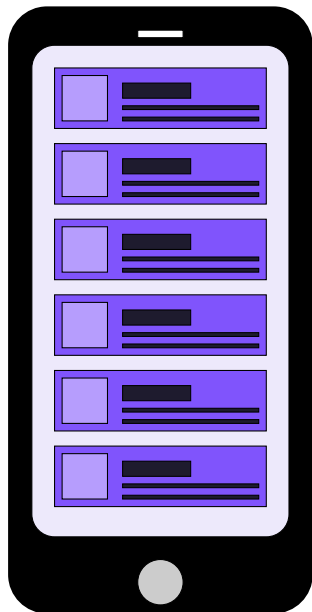


# Android: Coroutines: Beispiel

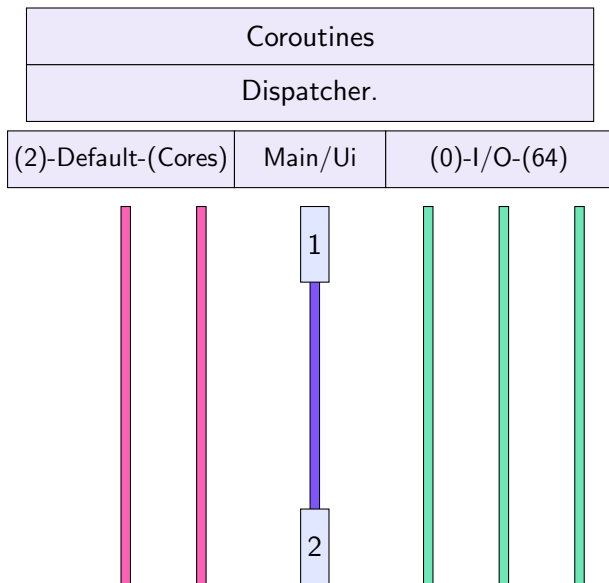
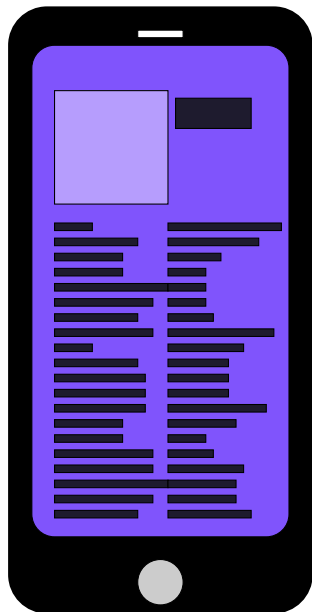




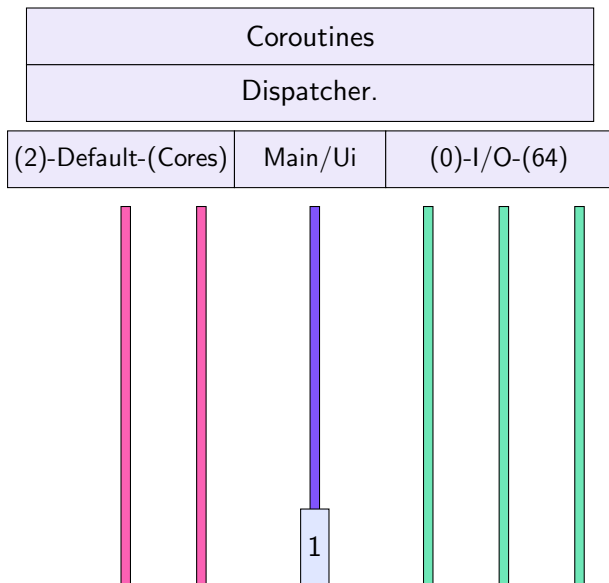
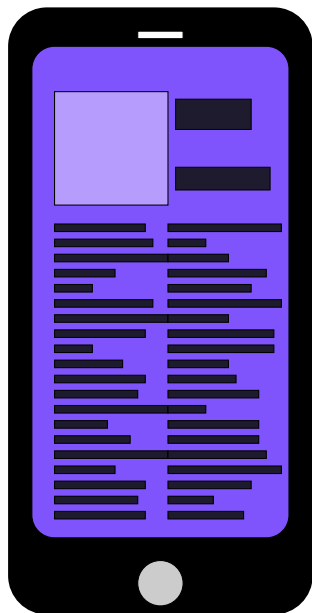
# Android: Coroutines: Beispiel



# Android: Coroutines: Beispiel



# Android: Coroutines: Beispiel



- Moderne Programmiersprache mit präziser Syntax und innovativen Features.
- Verbesserte Klassenstrukturen und Null-Sicherheit.
- Nahtlose Interoperabilität mit Java
- Multiplattform-Entwicklung (Android)

- Moderne Programmiersprache mit präziser Syntax und innovativen Features.
- Verbesserte Klassenstrukturen und Null-Sicherheit.
- Nahtlose Interoperabilität mit Java
- Multiplattform-Entwicklung (Android)

## **Aussicht:**

- Erweiterte Features: Smart Casts, Delegation, Destructuring ...
- Unterstützung funktionaler Programmierparadigmen