

# Kotlin

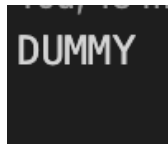
Proseminar: Fortgeschrittene Programmierkonzepte

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 Safety
  - Umfassende Multiplattform-Entwicklungsmöglichkeiten



1 Main-Methode

2 Variablen-Deklaration

3 Klassen

4 Properties

## 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 }
```

- Klassendeklaration: nicht 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 }
```

- Klassendeklaration: nicht erforderlich
- Schlüsselwort zur Funktionsdeklaration: fun



## 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 }
```

- Klassendeklaration: nicht erforderlich
- Schlüsselwort zur Funktionsdeklaration: `fun`
- Standardzugriffsmodifikator: `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 }
```

- Klassendeklaration: nicht erforderlich
- Schlüsselwort zur Funktionsdeklaration: fun
- Standardzugriffsmodifikator: public
- args-Parameter: 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 }
```

- Klassendeklaration: nicht erforderlich
- Schlüsselwort zur Funktionsdeklaration: fun
- Standardzugriffsmodifikator: public
- args-Parameter: optional
- Semikolons: nicht notwendig

# Variablen-Deklaration

## Java

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

## Kotlin

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

# 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
- **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
- **Keine** primitiven Typen
- Funktionen sind Objekte  $\Rightarrow$  Funktionale Programmierung möglich



# 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
- **Keine** primitiven Typen
- Funktionen sind Objekte  $\Rightarrow$  Funktionale Programmierung möglich

**Typinferenz** wird unterstützt:

- 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

# Java Getter und Setter

```
1 public class Verkaufsperson {  
2     private final String name;  
3     private double provision;  
4  
5  
6     public Verkaufsperson(String name, double  
7         provision) {...}  
8  
9     public String getName() {...}  
10  
11 }
```

# Java Getter und Setter

```
1 public class Verkaufsperson {  
2     private final String name;  
3     private double provision;  
4     private int umsatz;  
5  
6     public Verkaufsperson(String name, double  
7         provision) {...}  
8  
9     public String getName() {...}  
10    public int getUmsatz() {...}  
11    private void setUmsatz(int umsatz) {...}  
12 }
```



## 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    }  
}
```

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

- 5 Motivation
- 6 Safe call Operator
- 7 Elvis Operator
- 8 Not-null assertion Operator
- 9 Nullable Receiver Funktionen

## **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

## 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
- Konzept verhindert `NullPointerExceptions`

# Null Safety

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

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

- Unterscheidung zwischen *nullable* und *non-nullable* types

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

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

# Null Safety: Safe call Operator

in Java

```
1 private final Verkaufsperson vorgesetzter;  
2  
3 public void printVorgesetzter() {  
4     if (vorgesetzter == null)  
5         System.out.println(null);  
6     else System.out.println(vorgesetzter.name);  
7 }
```

# Null Safety: Safe call Operator

in Java

```
1 private final Verkaufsperson vorgesetzter;  
2  
3 public void printVorgesetzter() {  
4     if (vorgesetzter == null)  
5         System.out.println(null);  
6     else System.out.println(vorgesetzter.name);  
7 }
```

in Kotlin

```
1 val vorgesetzter: Verkaufsperson? = null  
2  
3 fun printVorgesetzter() {  
4     println(vorgesetzter?.name)  
5 }
```

## Verkettung des Operators

```
1 var name: String? = vorgesetzter?.vorgesetzter?.name
```



## Verkettung des Operators

```
1 var name: String? = vorgesetzter?.vorgesetzter?.name
```

## Zuweisungen mit dem Operator

```
1 vorgesetzter?.vorgesetzter?.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*

```
1 public void printVorgesetzter() {  
2     if (vorgesetzter == null)  
3         System.out.println("Kein_Vorgesetzter");  
4     else System.out.println(vorgesetzter.name);  
5 }
```

# Null Safety: Elvis Operator

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

```
1 public void printVorgesetzter() {  
2     if (vorgesetzter == null)  
3         System.out.println("Kein_Vorgesetzter");  
4     else System.out.println(vorgesetzter.name);  
5 }
```

```
1 fun printVorgesetzter() {  
2     println(vorgesetzter?.name ?: "Kein_Vorgesetzter")  
3 }
```

# Null Safety: Not-null assertion Operator

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

# Null Safety: Not-null assertion Operator

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

- Kann zu `NullPointerExceptions` führen

# Null Safety: Nullable Receiver

- Funktionen können extern deklariert werden



# Null Safety: Nullable Receiver

- Funktionen können extern deklariert werden
- Erlaubt auch Methodenaufruf auf nullable types

# Null Safety: Nullable Receiver

- Funktionen können extern deklariert werden
- Erlaubt auch Methodenaufruf auf nullable types
- Null Werte werden innerhalb der Methode behandelt

# Null Safety: Nullable Receiver

- Funktionen können extern deklariert werden
- Erlaubt auch Methodenaufruf auf nullable types
- Null Werte werden innerhalb der Methode behandelt

# Null Safety: Nullable Receiver

- Funktionen können extern deklariert werden
- Erlaubt auch Methodenaufruf auf nullable types
- Null Werte werden innerhalb der Methode behandelt

```
1 fun Verkaufsperson?.print() {  
2     if (this == null) return println("Diese Person  
    existiert nicht")  
3     return println("$name: $provision Anteil")  
4 }
```

# Null Safety: Nullable Receiver

- Funktionen können extern deklariert werden
- Erlaubt auch Methodenaufruf auf nullable types
- Null Werte werden innerhalb der Methode behandelt

```
1 fun Verkaufsperson?.print() {  
2     if (this == null) return println("Diese Person  
    existiert nicht")  
3     return println("$name: $provision Anteil")  
4 }
```

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

## Java in Kotlin benutzen

- 10 Zugriff auf Klassen und Instanzen
- 11 Mapped Types
- 12 Null safety mit Java

## Kotlin in Java benutzen

- 13 Kotlin Properties in Java

```
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() {...}  
10    public void setProvision(double provision) {...}  
11 }
```

```
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() {...}
10    public void setProvision(double provision) {...}
11 }
```

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



- 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
- `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!`
- 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!`
- primitive typ `int`  $\Rightarrow$  `kotlin.Int`
- `java.lang.Integer`  $\Rightarrow$  `kotlin.Int?`

# Interoperabilität: Null safety mit Java

```
1 public Verkaufsperson erstellePerson() {  
2     return null;  
3 }
```

# Interoperabilität: Null safety mit Java

```
1 public Verkaufsperson erstellePerson() {  
2     return null;  
3 }
```

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

# Interoperabilität: Null safety mit Java

```
1 public Verkaufsperson erstellePerson() {  
2     return null;  
3 }
```

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

- Aus Java zurückgegebene Instanzen können *null* sein



# Interoperabilität: Null safety mit Java

```
1 public Verkaufsperson erstellePerson() {  
2     return null;  
3 }
```

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

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

# Interoperabilität: Null safety mit Java

```
1 public Verkaufsperson erstellePerson() {  
2     return null;  
3 }
```

```
1 val person: Verkaufsperson = erstellePerson()  
2 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

```
1 public Verkaufsperson erstellePerson() {  
2     return null;  
3 }
```

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

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

# Interoperabilität: Kotlin Properties in Java

```
1 var name: String
```

# Interoperabilität: Kotlin Properties in Java

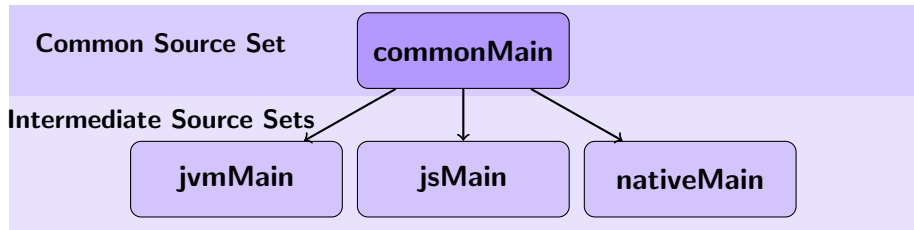
```
1 var name: String
```

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

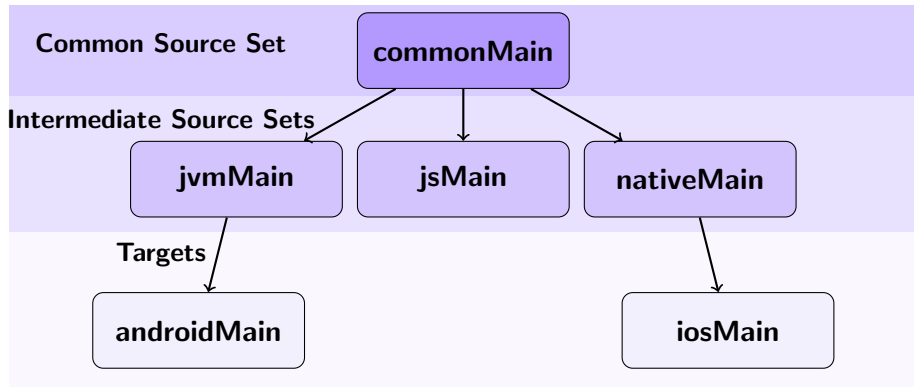
**Common Source Set**

**commonMain**

# Multiplatform

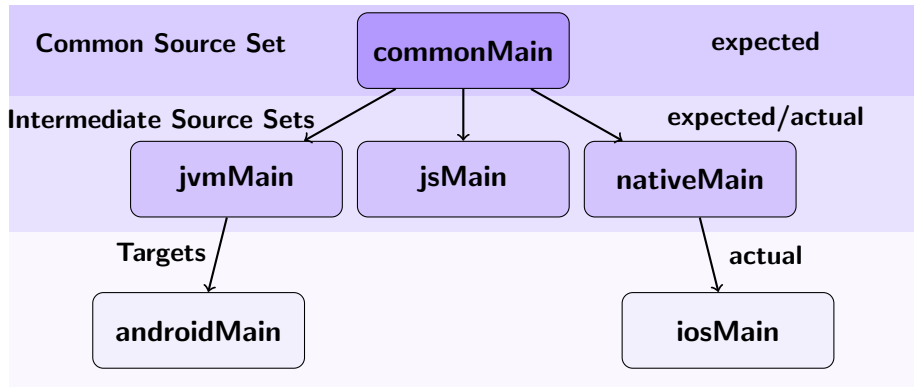


# Multiplatform





# Multiplatform



14 Jetpack Compose

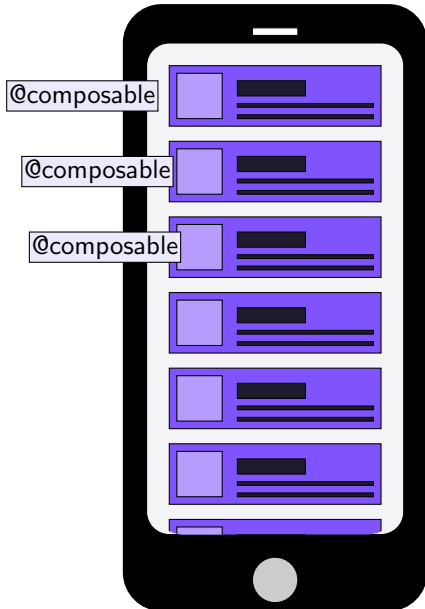
15 Coroutines

16 Beispiel

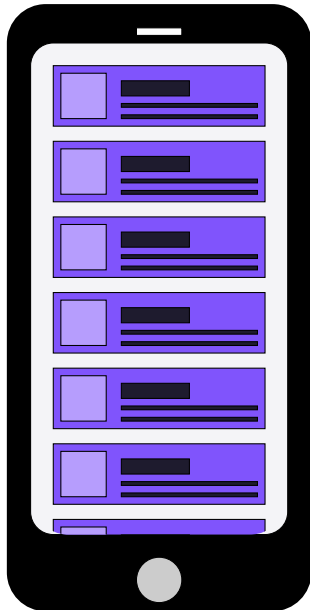
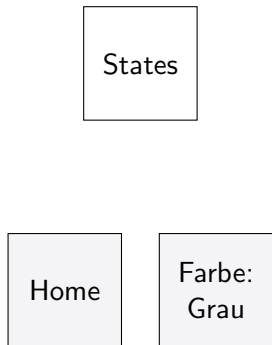
# Android: Jetpack compose

States

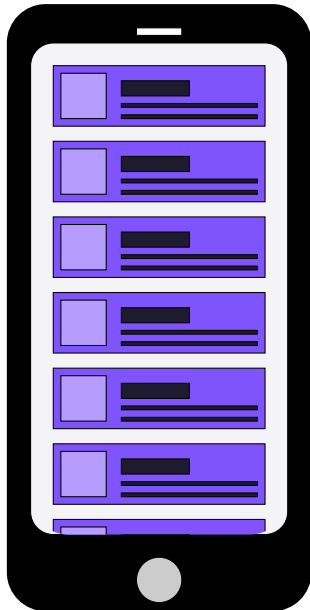
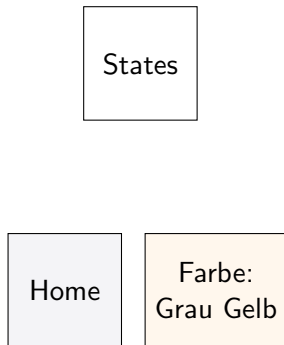
Home



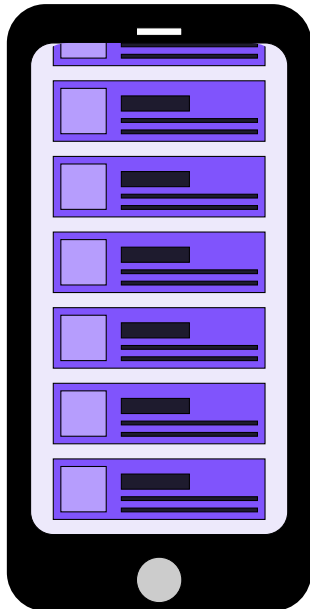
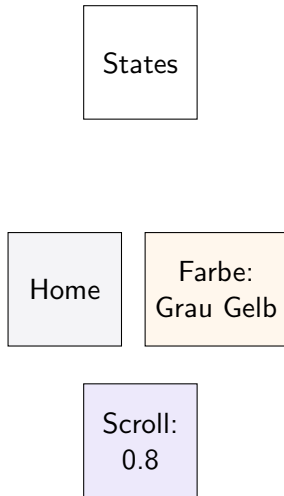
# Android: Jetpack compose



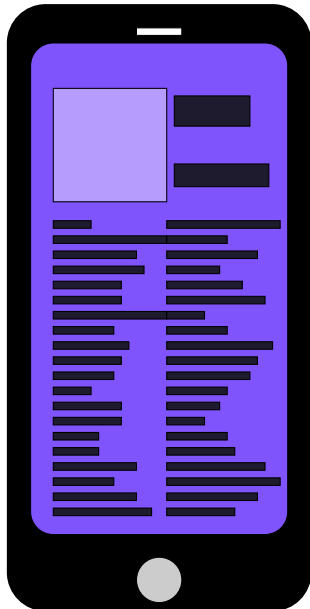
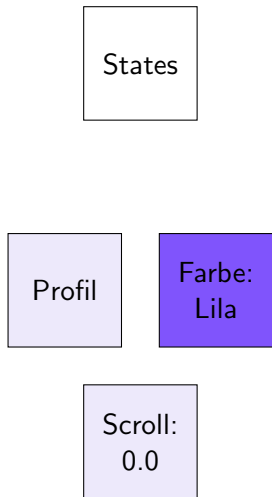
# Android: Jetpack compose



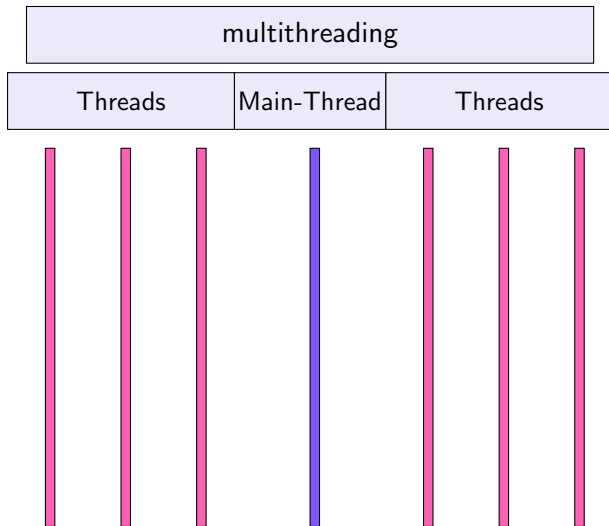
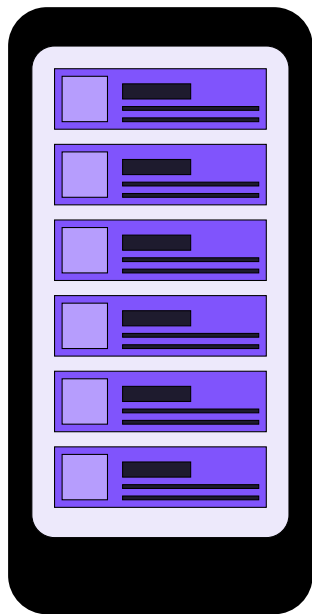
# Android: Jetpack compose



# Android: Jetpack compose

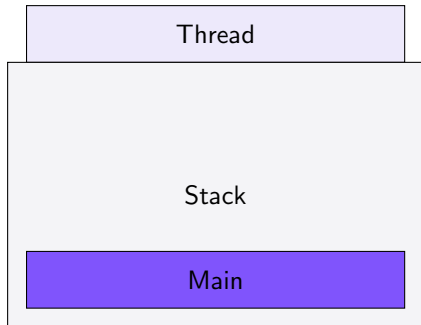


# Android: Coroutines

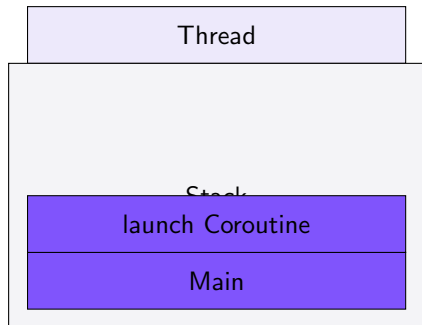




# Android: Coroutines



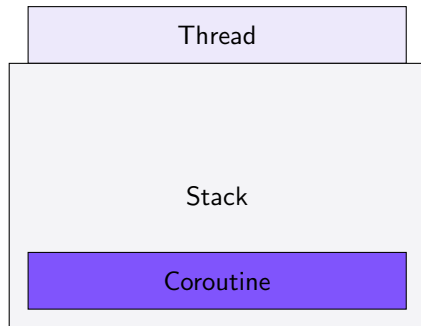
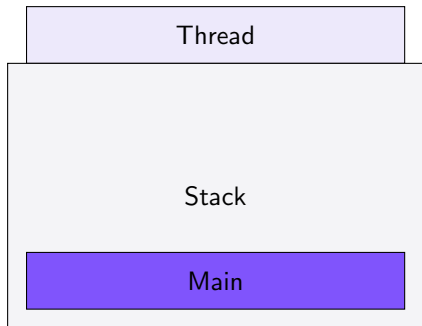
# Android: Coroutines



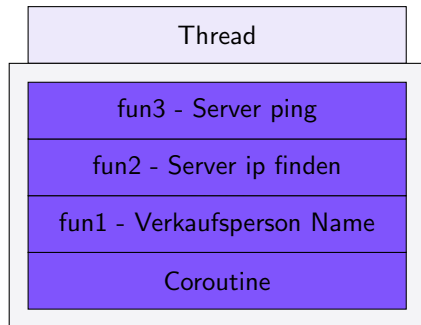
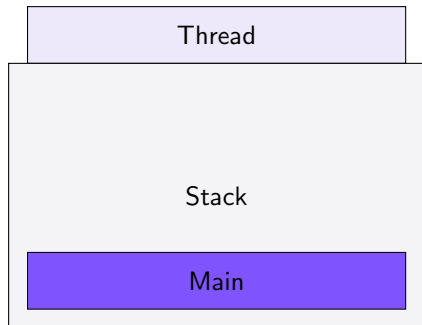
```
1 new Fiber<Void>(() -> {  
2     //coroutine  
3     return null;  
4 }) .start();
```

```
1 launch {  
2     //coroutine  
3 }
```

# Android: Coroutines

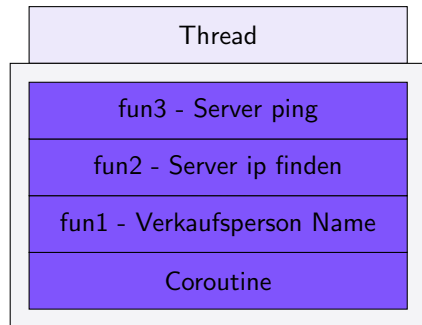
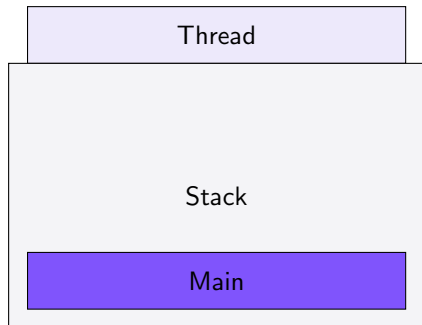


# Android: Coroutines



# Android: Coroutines

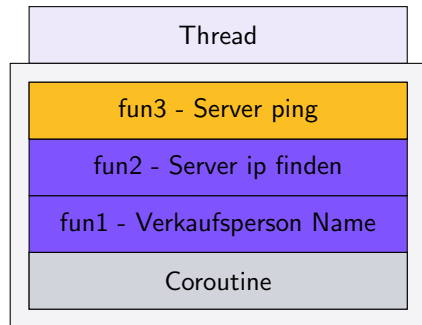
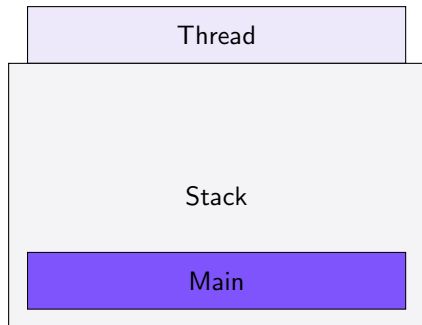
suspends: `delay()`, `yield()`, `withContext()`, ...



```
1 suspend fun pingServer() {  
2     //ping Server  
3 }
```

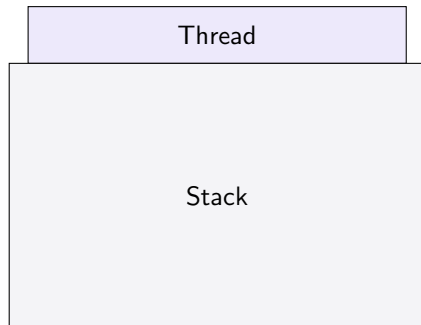
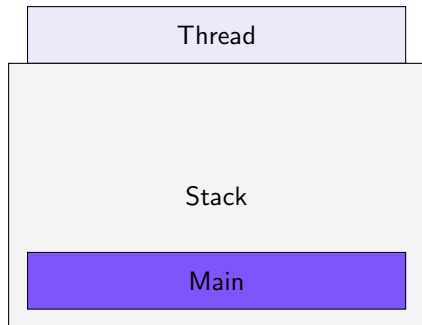
# Android: Coroutines

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



```
1 suspend fun pingServer() {  
2     //ping Server  
3 }
```

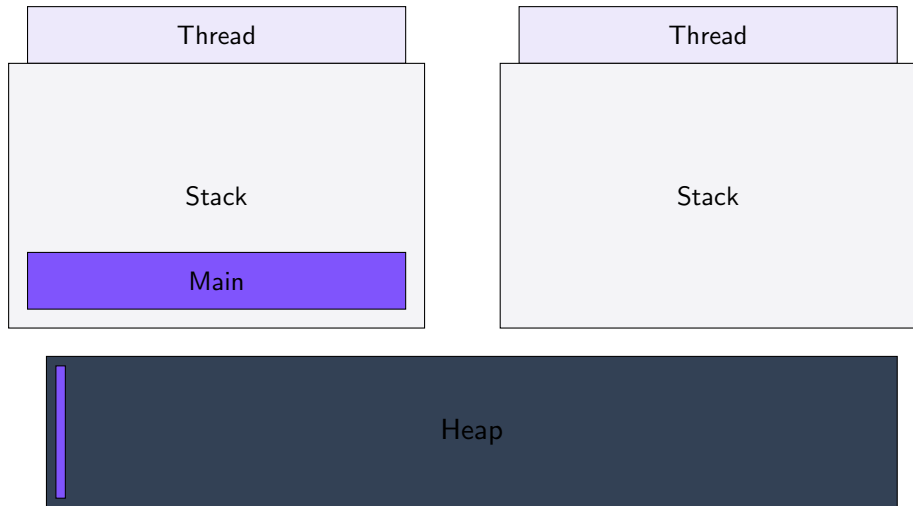
# Android: Coroutines



Coroutine:

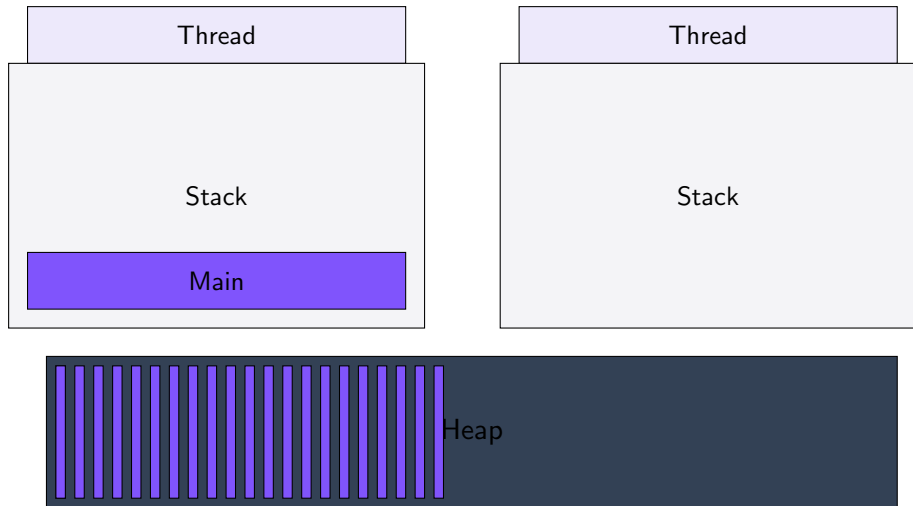
Variablen:	Name,lp
State:	1
Path:	fun1-fun2-fun3

# Android: Coroutines

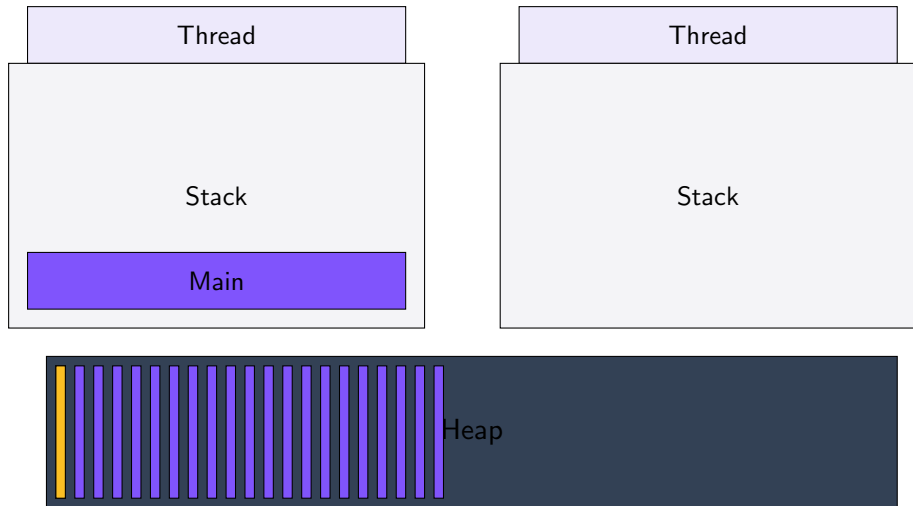




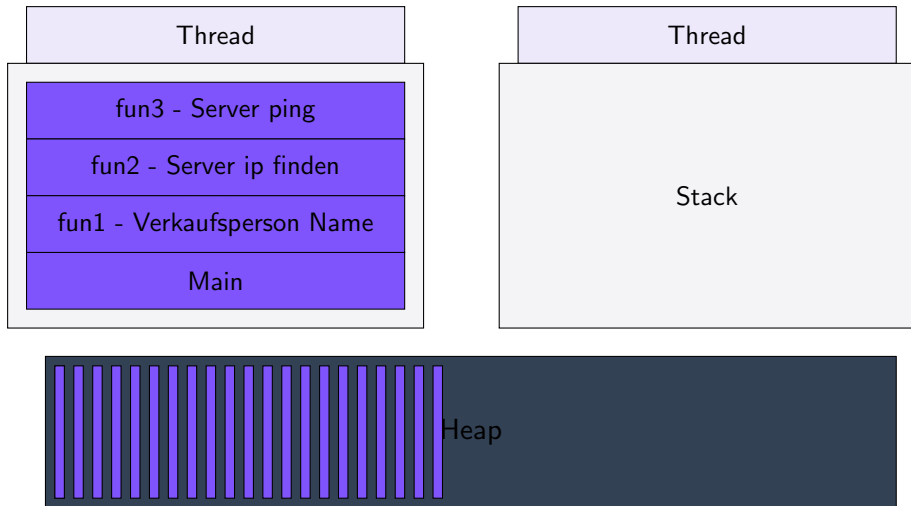
# Android: Coroutines



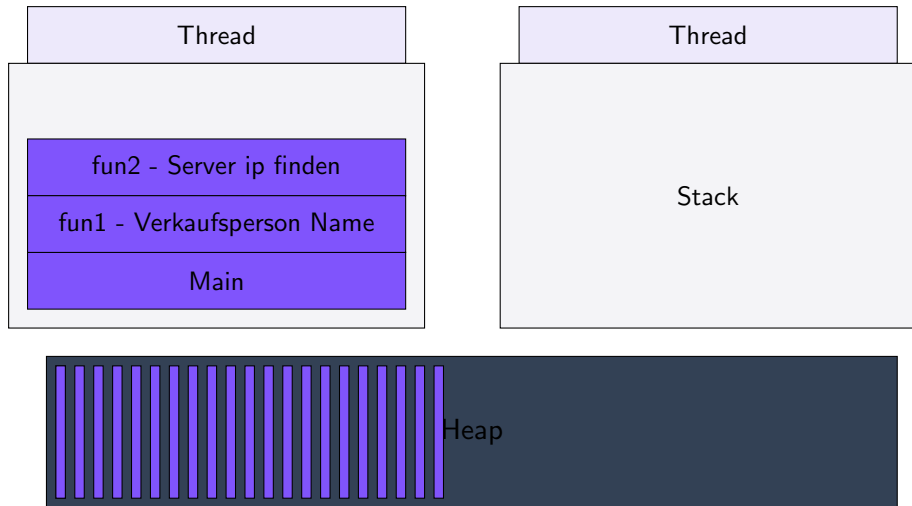
# Android: Coroutines



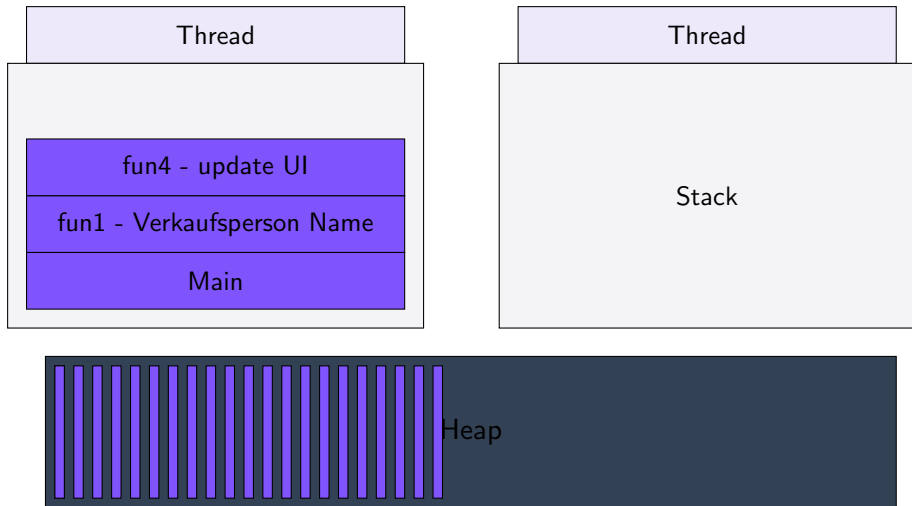
# Android: Coroutines



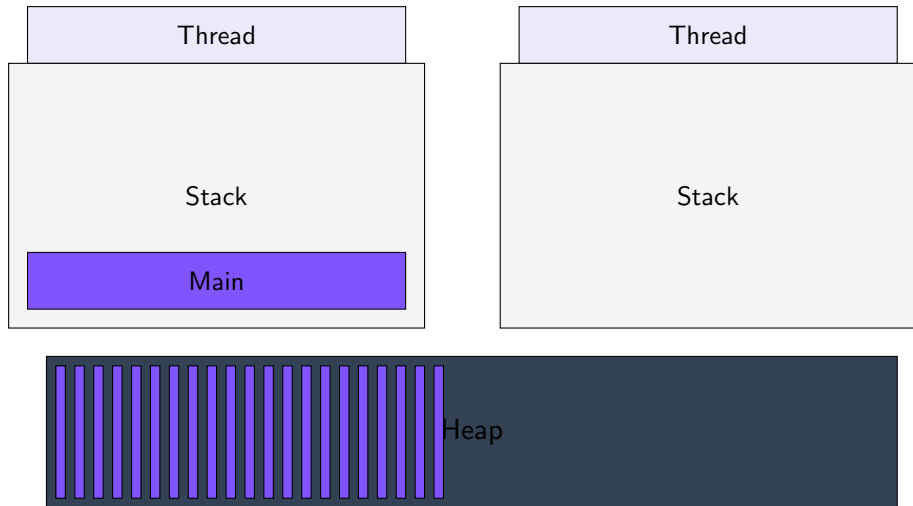
# Android: Coroutines



# Android: Coroutines

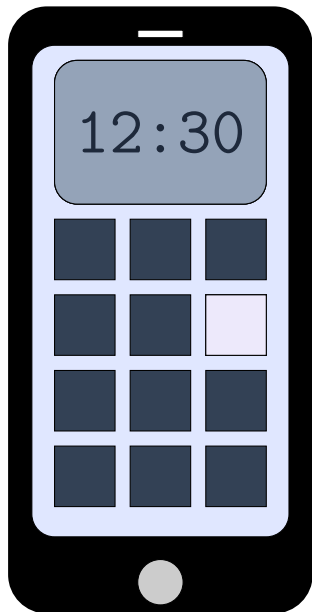


# Android: Coroutines



## Beispiel Coroutines

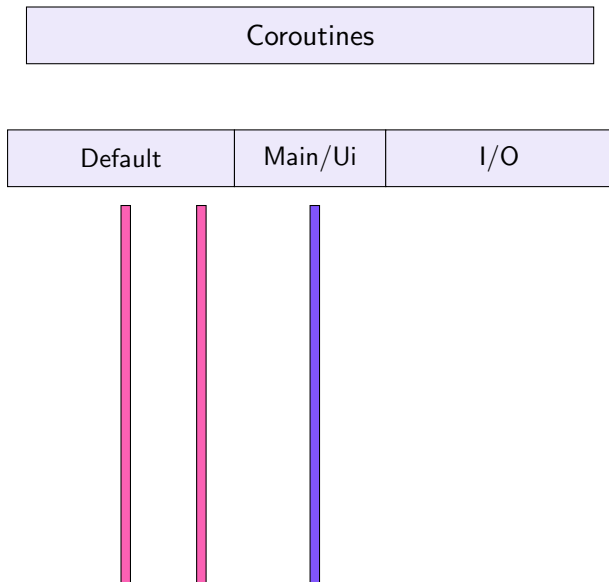
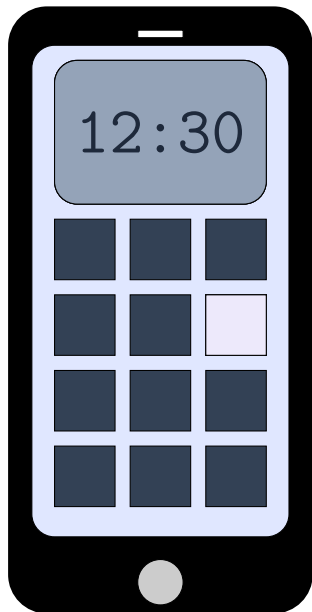
# Android: Coroutines: Beispiel



Coroutines

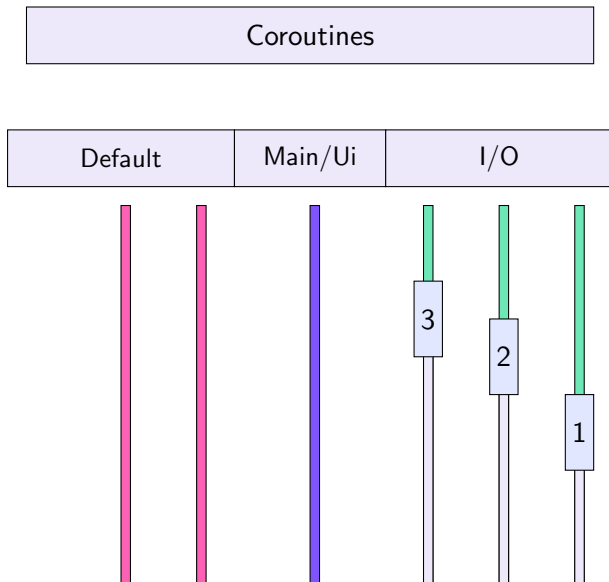
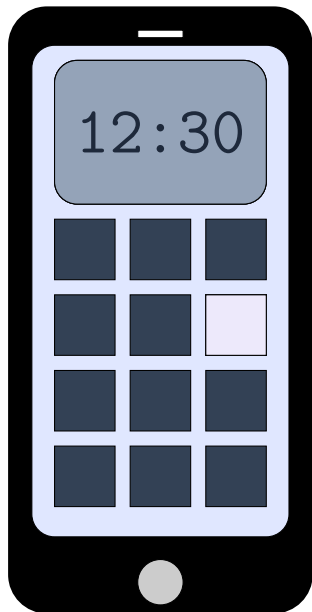


# Android: Coroutines: Beispiel

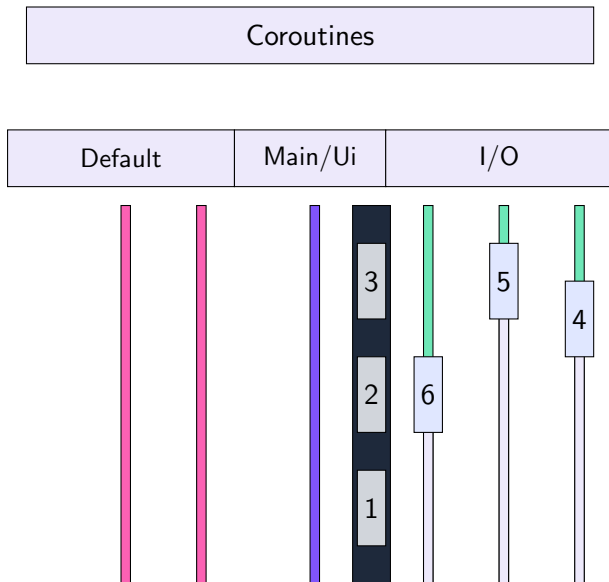
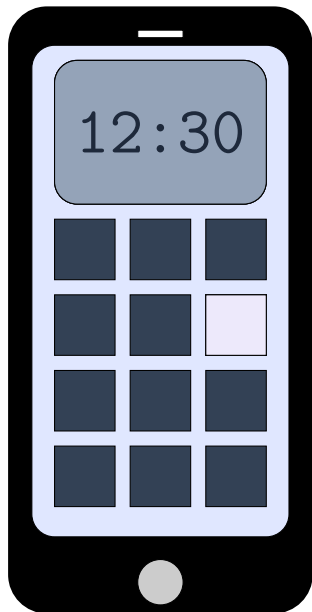




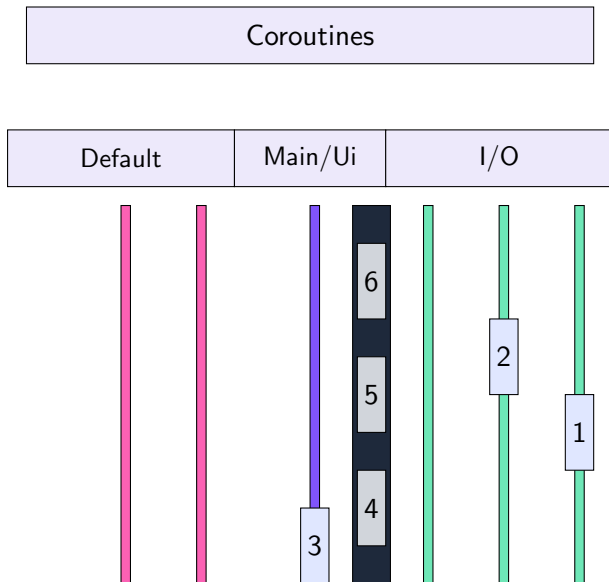
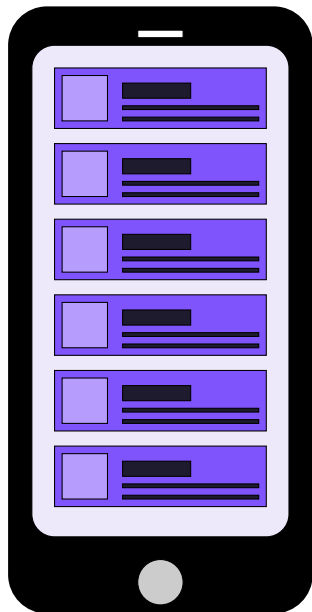
# Android: Coroutines: Beispiel



# Android: Coroutines: Beispiel



# Android: Coroutines: Beispiel



- 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