Kotlin

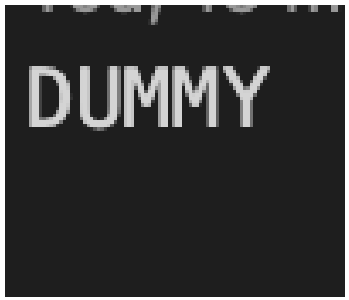
Proseminar: Fortgeschrittene Programmierkonzepte

Christian Konersmann, Finn Paul Lippok, Paul Lukas

05.05.2025

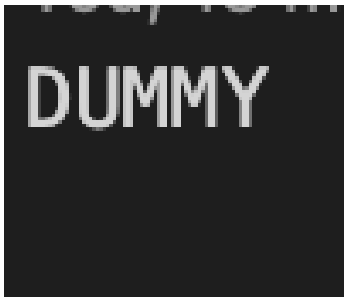
Was ist Kotlin?

- **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

Syntax

1. Main-Methode
2. Variablen-Deklaration
3. Klassen
4. Properties

Main-Methode

Java Main-Methode

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

Main-Methode

Java Main-Methode

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

Kotlin Main-Methode

```kotlin
fun main() {
    println("Hello, World!")
}
```

Main-Methode

Java Main-Methode

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

Kotlin Main-Methode

```kotlin
fun main() {
    println("Hello, World!")
}
```

- Klassendeklaration: nicht erforderlich

Main-Methode

Java Main-Methode

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

Kotlin Main-Methode

```kotlin
fun main() {
    println("Hello, World!")
}
```

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

Main-Methode

Java Main-Methode

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

Kotlin Main-Methode

```kotlin
fun main() {
    println("Hello, World!")
}
```

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

Main-Methode

Java Main-Methode

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

Kotlin Main-Methode

```kotlin
fun main() {
    println("Hello, World!")
}
```

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

Main-Methode

Java Main-Methode

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

Kotlin Main-Methode

```kotlin
fun main() {
    println("Hello, World!")
}
```

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

Variablen-Deklaration

Java

```java
int a = 5;
final String b = "Hallo";
```

Kotlin

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

Variablen-Deklaration

Java

```java
int a = 5;
final String b = "Hallo";
```

Kotlin

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

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

Variablen-Deklaration

Java

```java
int a = 5;
final String b = "Hallo";
```

Kotlin

```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

```java
int a = 5;
final String b = "Hallo";
```

Kotlin

```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

```java
int a = 5;
final String b = "Hallo";
```

Kotlin

```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

```java
int a = 5;
final String b = "Hallo";
```

Kotlin

```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.

Klassen

Java

```java
public class Verkaufsperson {
  public final String name;
  private double provision;

  public Verkaufsperson (String name, double
      provision) {...}
}
```

Klassen

Java

```java
public class Verkaufsperson {
  public final String name;
  private double provision;

  public Verkaufsperson (String name, double
      provision) {...}
}
```

Kotlin

```kotlin
class Verkaufsperson() {



  val name: String
  private var provision: Double
}
```

Klassen

Java

```java
public class Verkaufsperson {
  public final String name;
  private double provision;

  public Verkaufsperson (String name, double
      provision) {...}
}
```

Kotlin

```kotlin
class Verkaufsperson(
  name: String,
  provision: Double = 0.2
) {
  val name: String = name
  private var provision: Double = provision
}
```

Klassen

Java

```java
public class Verkaufsperson {
  public final String name;
  private double provision;

  public Verkaufsperson (String name, double
      provision) {...}
}
```

Kotlin

```kotlin
class Verkaufsperson(
  val name: String,
  private var provision: Double = 0.2
) {


}
```

Klassen

Java

```java
public class Verkaufsperson {
  public final String name;
  private double provision;

  public Verkaufsperson (String name, double
      provision) {...}
}
```

Kotlin

```kotlin
class Verkaufsperson(
  val name: String,
  private var provision: Double = 0.2
) {}
```

- 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
         provision) {...}
7
8    public String getName() {...}
9
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
         provision) {...}
7
8    public String getName() {...}
9    public int getUmsatz() {...}
10   private void setUmsatz(int umsatz) {...}
11 }
```

Properties

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 }
```

Properties

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 }
```

Properties

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␣
               positiv␣sein")
8        field = value
9      }
10 }
```

Properties

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
               positiv sein")
8        field = value
9      }
10 }
```

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

Null Safety

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

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

Null Safety

Motivation: Null Safety

Java Beispiel

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

- Code wirft `java.lang.NullPointerException`

Null Safety

Motivation: Null Safety

Java Beispiel

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

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

Null Safety

Motivation: Null Safety

Java Beispiel

```java
Verkaufsperson person = null;
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"
```

Null Safety

```
1 var a : String = "a ist non-nullable"
2 var b : String? = "b ist nullable"
```

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

Null Safety

```
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

```java
private final Verkaufsperson vorgesetzter;

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

Null Safety: Safe call Operator

in Java

```java
private final Verkaufsperson vorgesetzter;

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

in Kotlin

```kotlin
val vorgesetzter: Verkaufsperson? = null

fun printVorgesetzter() {
  println(vorgesetzter?.name)
}
```

Null Safety: Safe call Operator

Verkettung des Operators

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

Null Safety: Safe call Operator

Verkettung des Operators

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

Zuweisungen mit dem Operator

```
1 vorgesetzter?.vorgesetzter?.provision = 0.0
```

Null Safety: Elvis Operator

- 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*

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

```kotlin
fun printVorgesetzter() {
  println(vorgesetzter?.name ?: "Kein Vorgesetzter")
}
```

Null Safety: Not-null assertion Operator

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

Null Safety: Not-null assertion Operator

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

- 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()
```

Interoperabilität

Java in Kotlin benutzen

Kotlin in Java benutzen

Interoperabilität

```java
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) {...}
}
```

Interoperabilität

```java
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) {...}
}
```

```kotlin
var carl = Verkaufsperson("Carl␣Mueller", 0.1)
println(carl.name)
carl.provision = 0.2
```

Interoperabilität: Mapped Types

- 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` ⇒ `kotlin.Any!`
- Primitiver Typ `int` ⇒ `kotlin.Int`

Interoperabilität: Mapped Types

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

- `java.lang.Object` ⇒ `kotlin.Any!`
- Primitiver Typ `int` ⇒ `kotlin.Int`
- `java.lang.Integer` ⇒ `kotlin.Int?`

Interoperabilität: Null safety mit Java

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

Interoperabilität: Null safety mit Java

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

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

Interoperabilität: Null safety mit Java

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

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

- haben spezial-Typ: *platform type*

Interoperabilität: Null safety mit Java

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

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

- haben spezial-Typ: *platform type*
- gelockerte Regeln bezüglich Null safety

Interoperabilität: Null safety mit Java

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

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

- haben spezial-Typ: *platform type*
- gelockerte Regeln bezüglich Null safety
- anfälliger für `NullPointerExceptions`

Interoperabilität: Kotlin Properties in Java

```kotlin
class Verkaufsperson() {
  val name: String
}
```

Interoperabilität: Kotlin Properties in Java

```kotlin
class Verkaufsperson() {
  val name: String
}
```

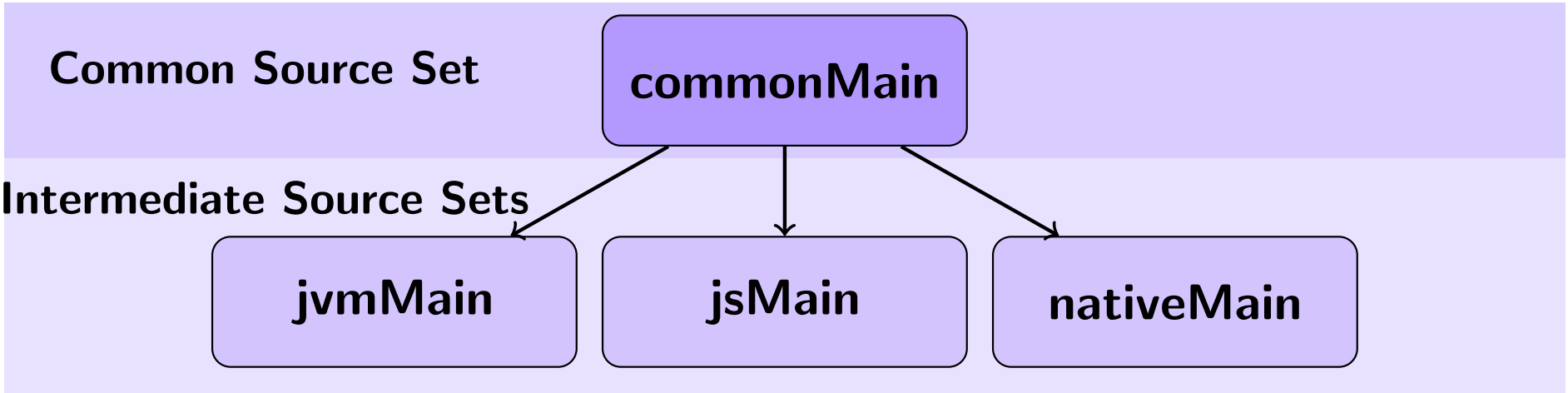
```java
public class Verkaufsperson {
  private String name;

  public String getName() {
    return name;
  }

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

Multiplatform Entwicklung

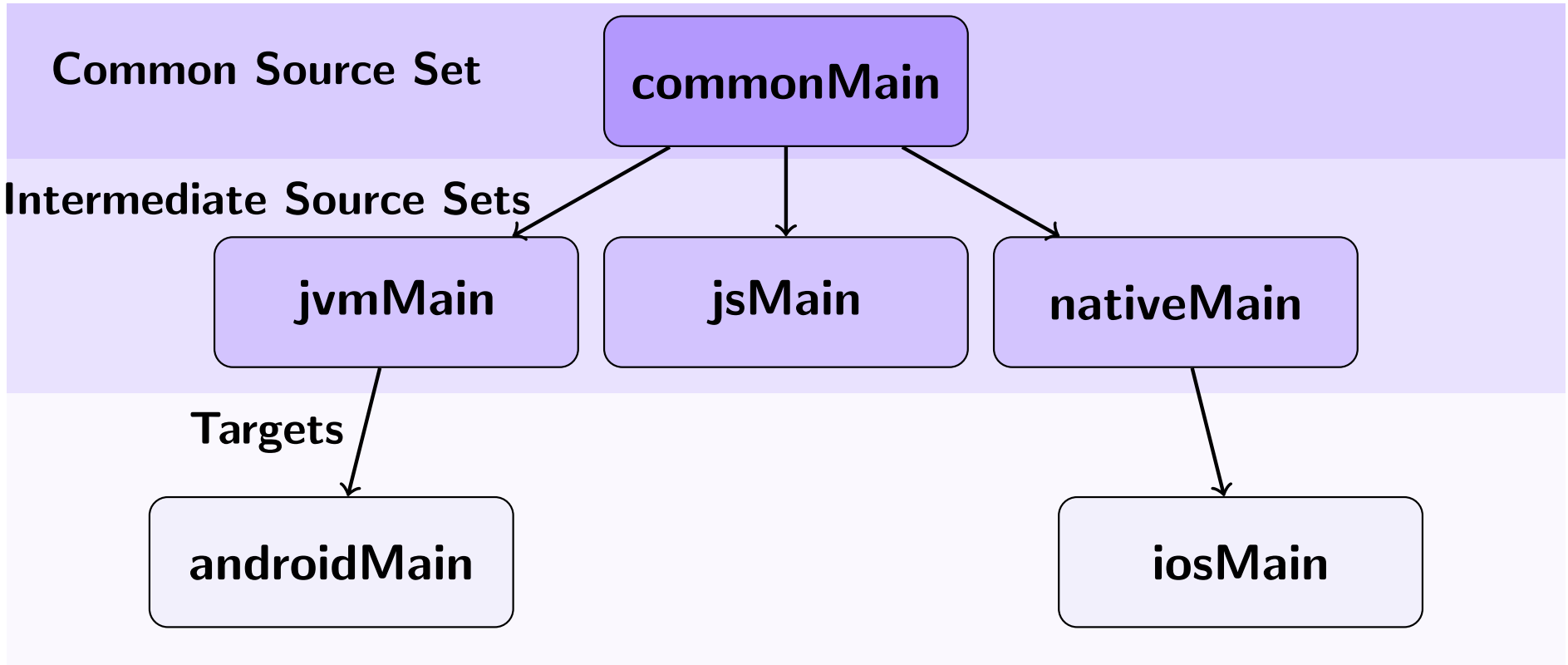
Multiplatform Entwicklung

Common Source Set **commonMain**

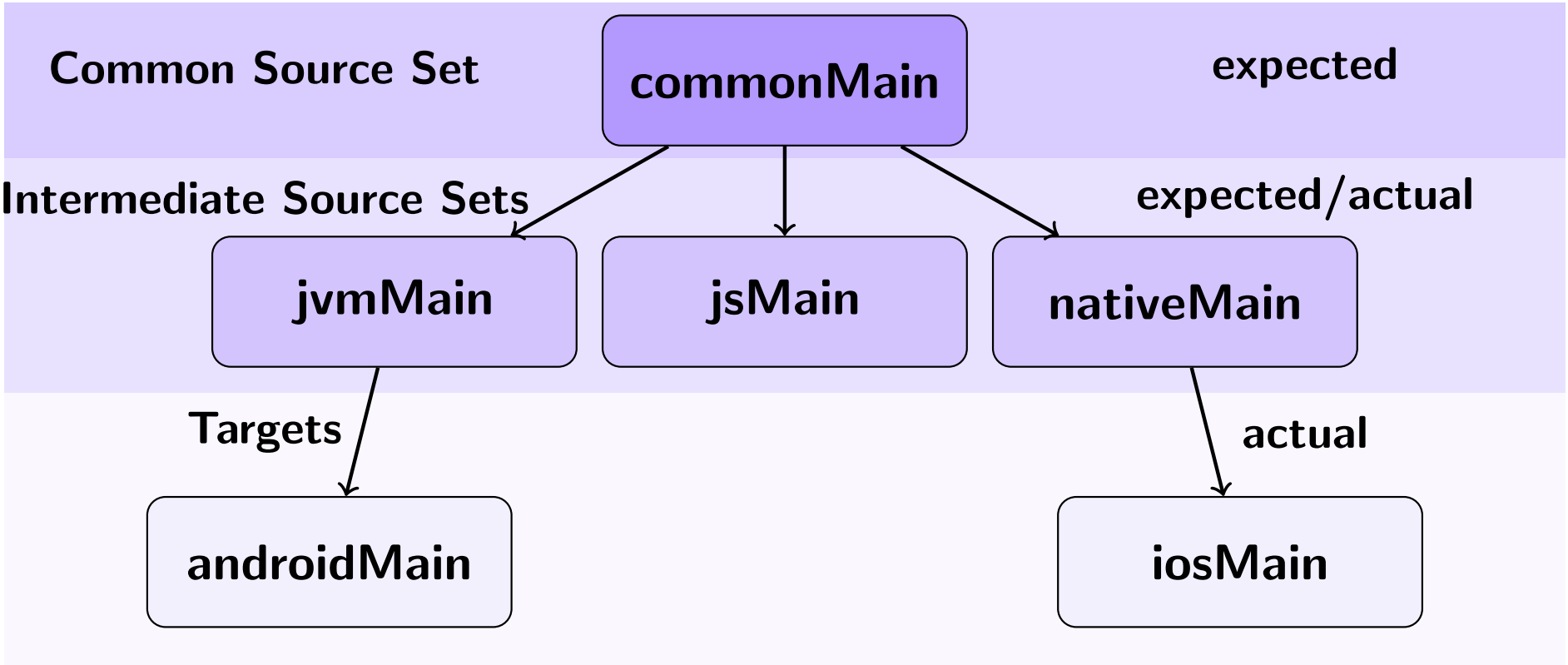
Multiplatform Entwicklung



Multiplatform Entwicklung



Multiplatform Entwicklung

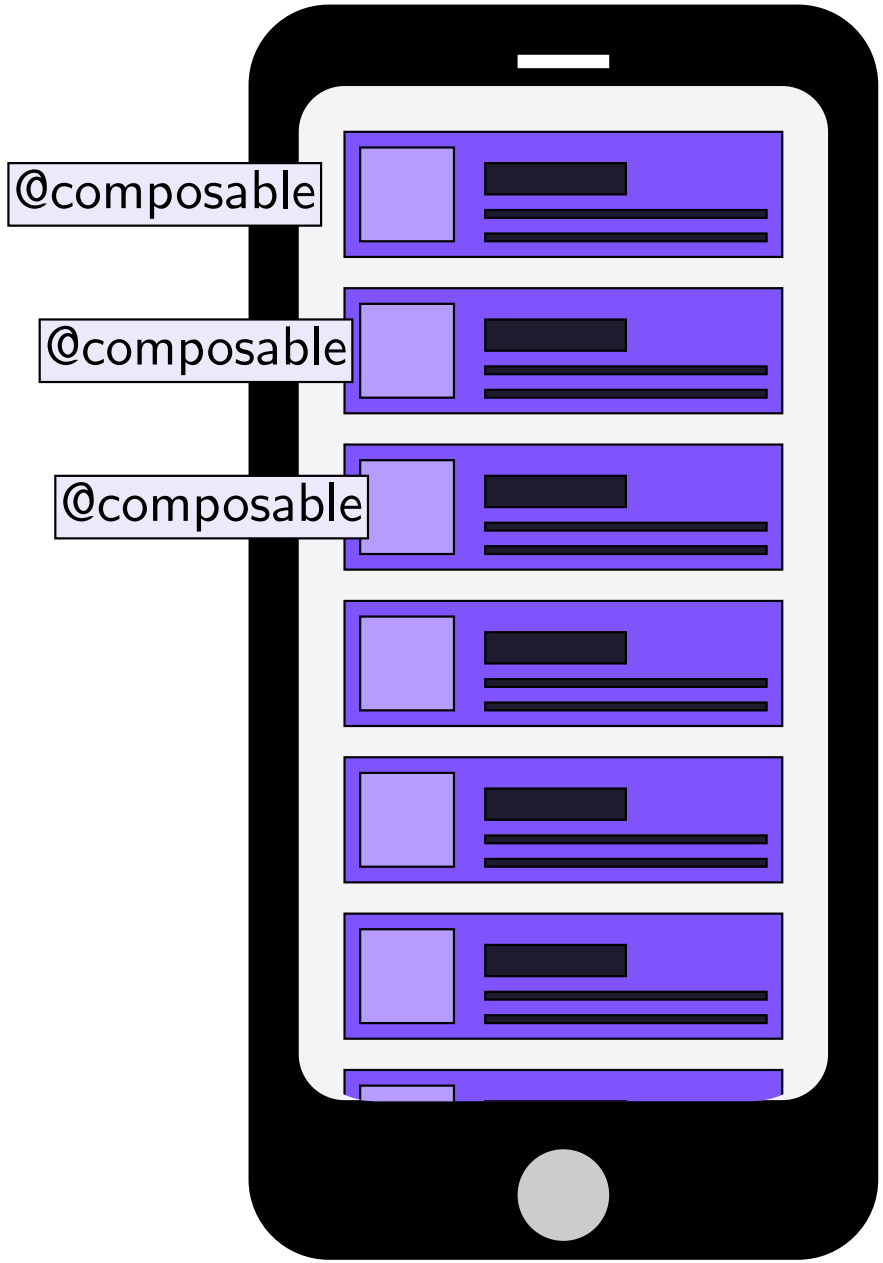


Android

Android: Jetpack compose

States

Home

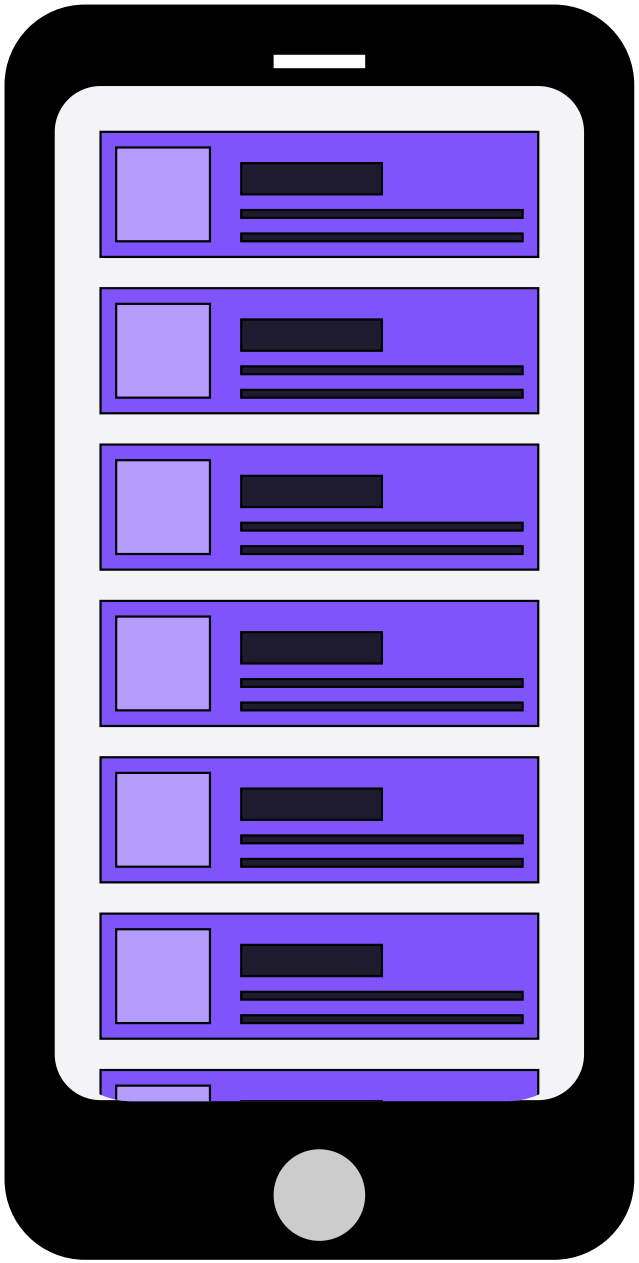


Android: Jetpack compose

States

Home

Farbe:
Grau

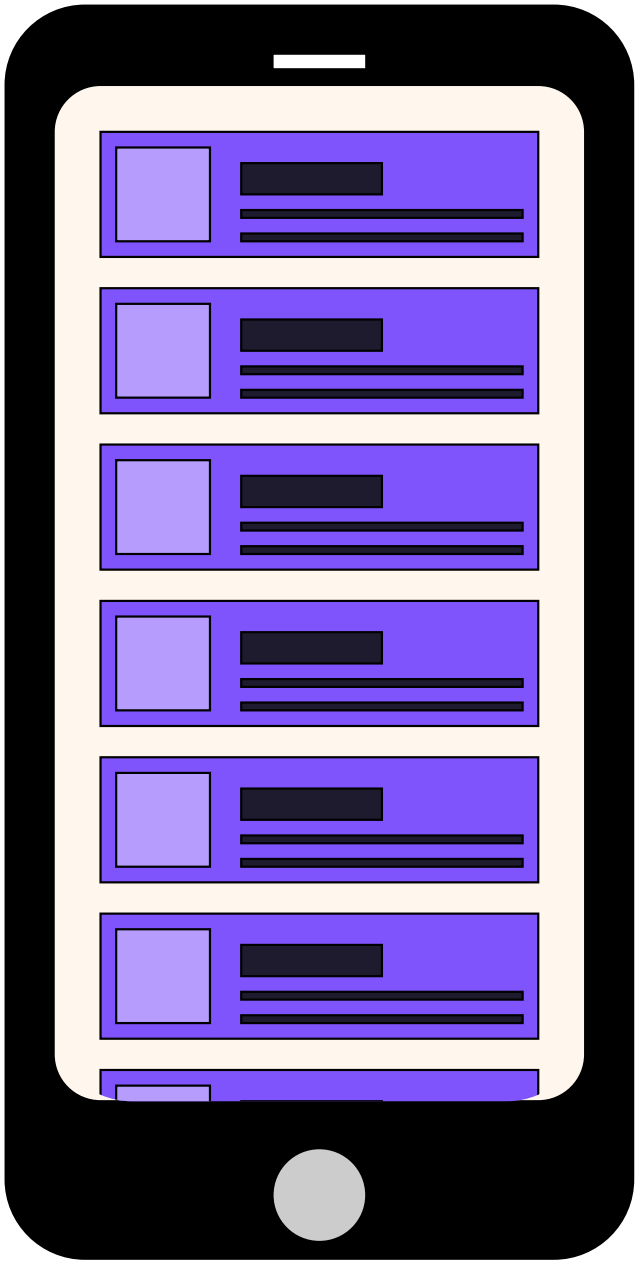


Android: Jetpack compose

States

Home

Farbe:
Grau Gelb



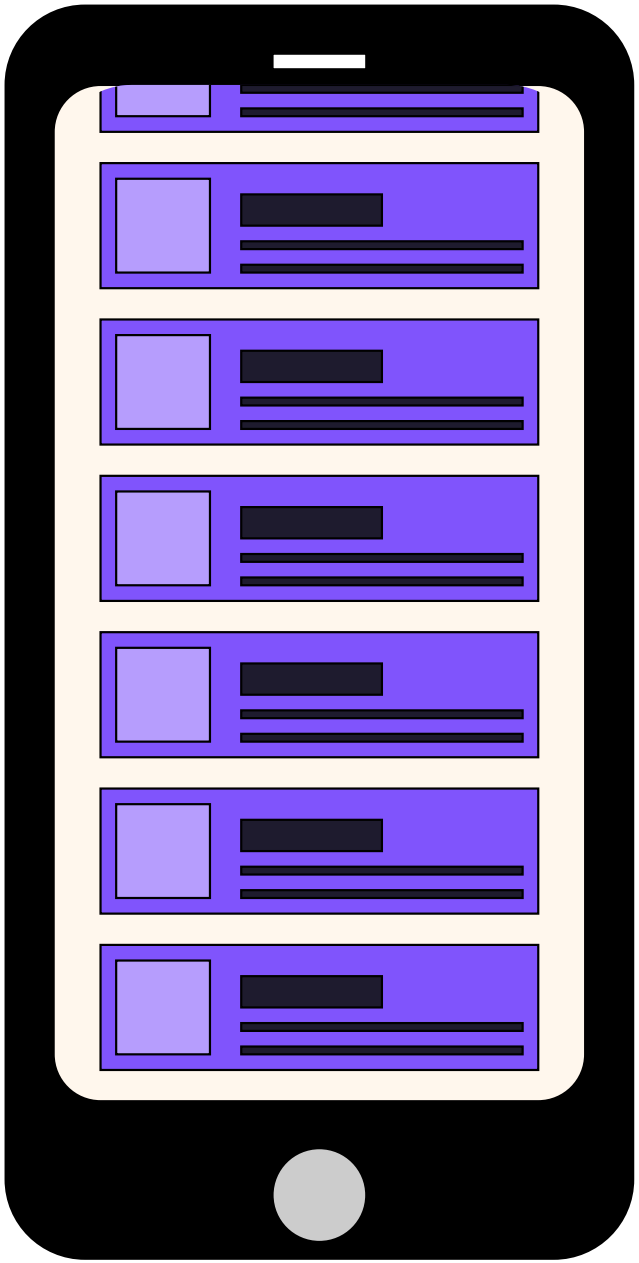
Android: Jetpack compose

States

Home

Farbe:
Grau Gelb

Scroll:
0.8



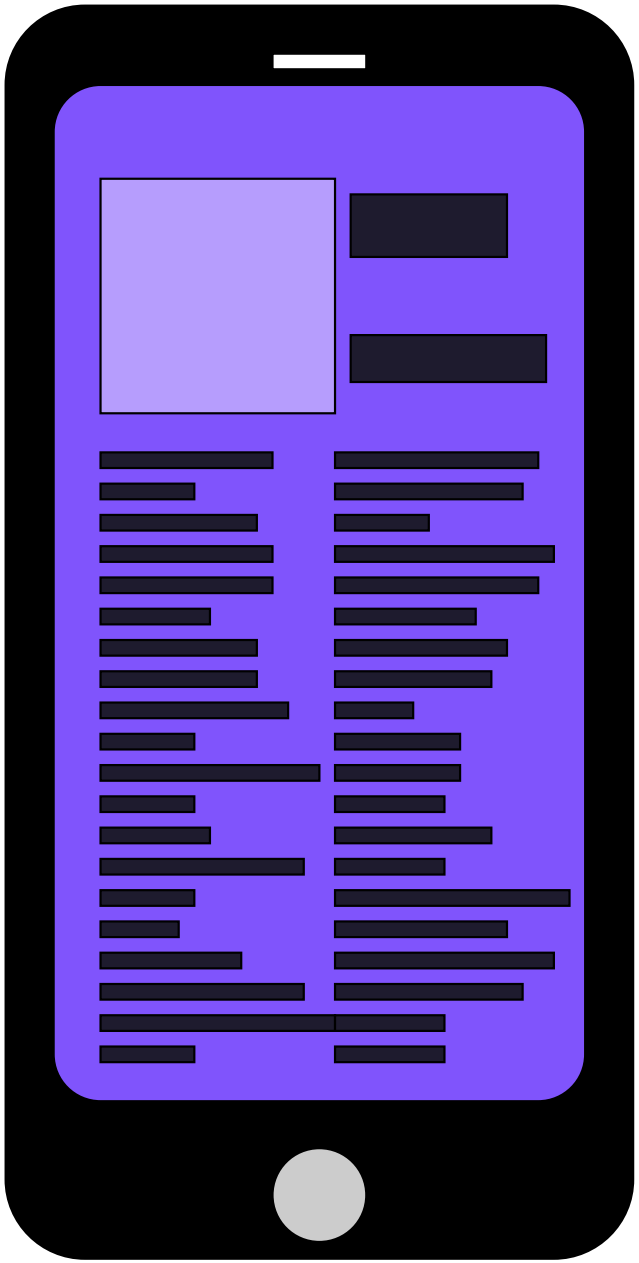
Android: Jetpack compose

States

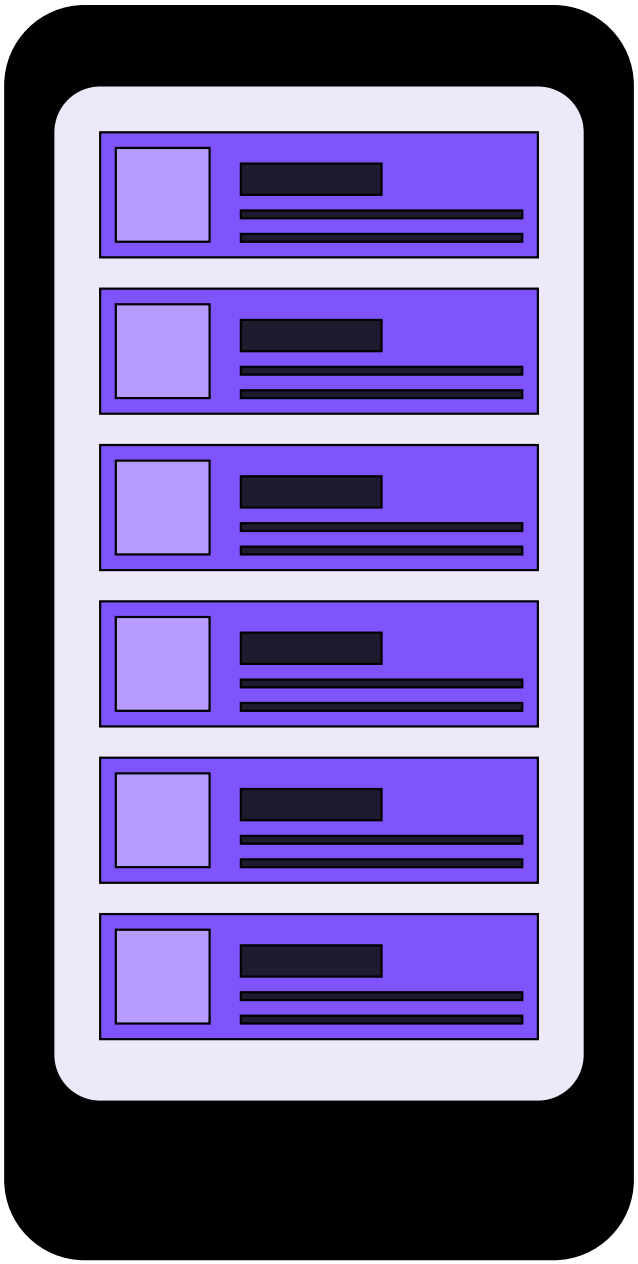
Profil

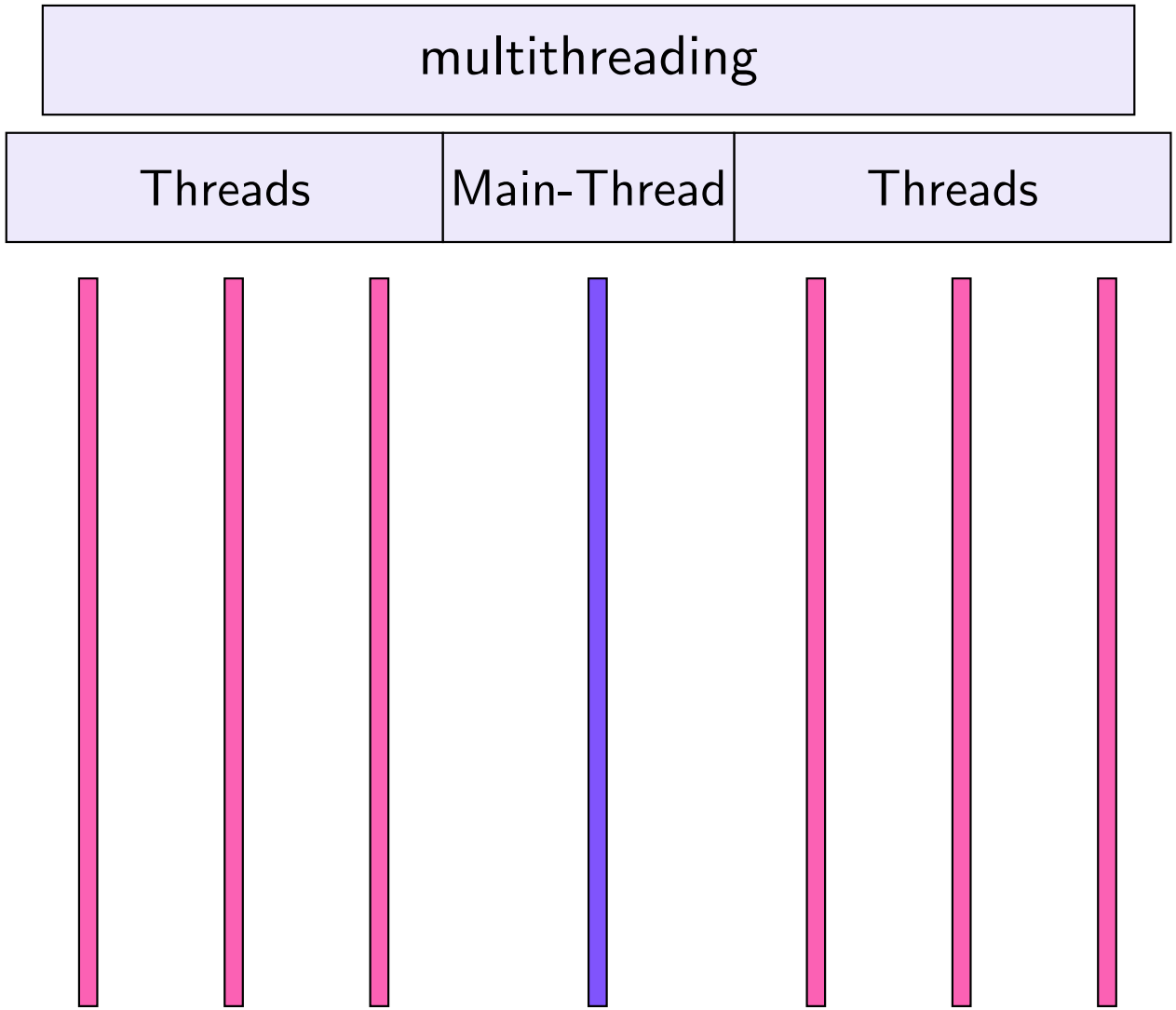
Farbe:
Lila

Scroll:
0.0

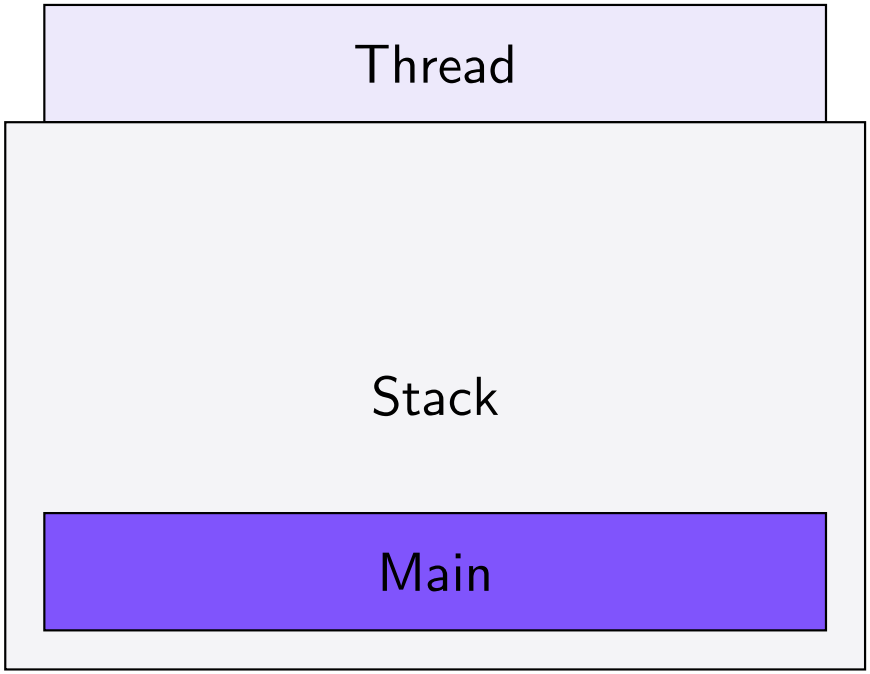


Android: Coroutines

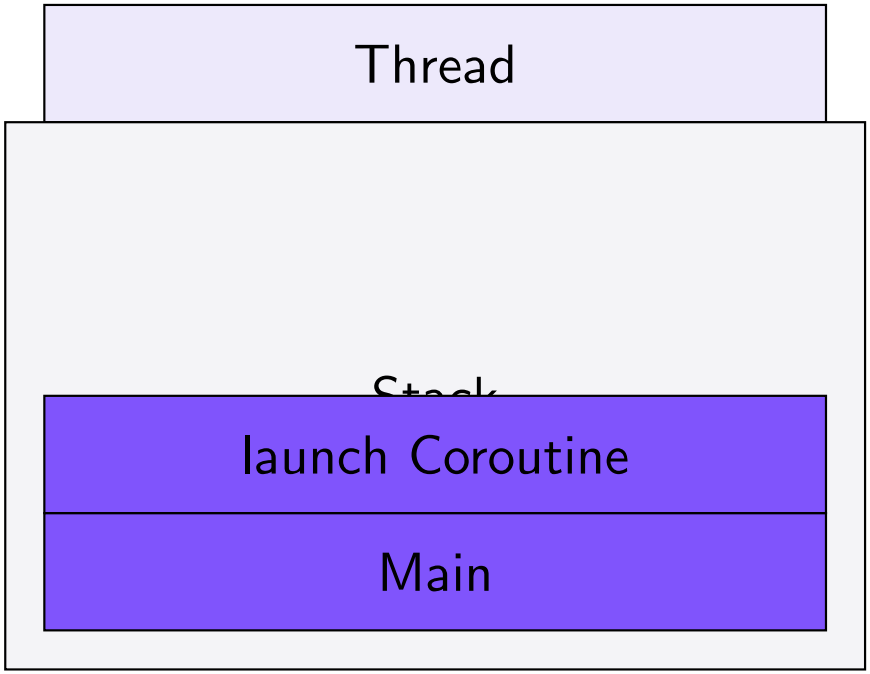




Android: Coroutines



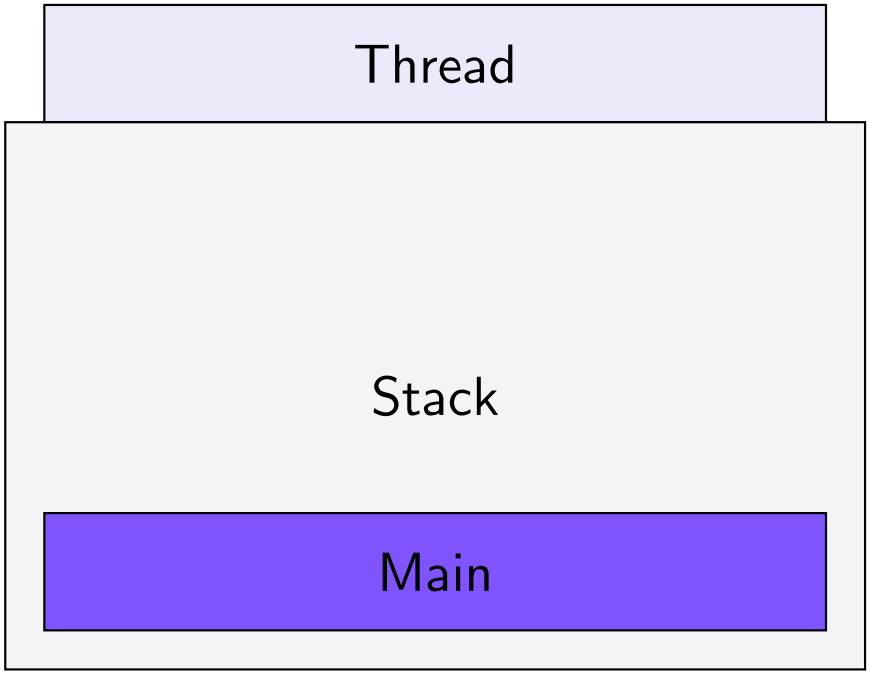
Android: Coroutines

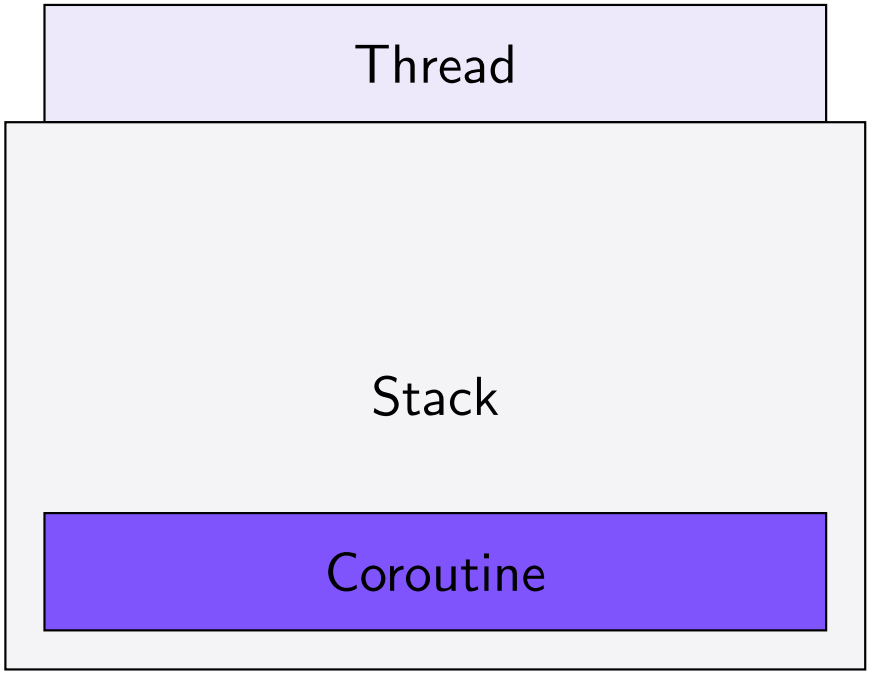



```
new Fiber<Void>(() -> {
    //coroutine
    return null;
}).start();
```

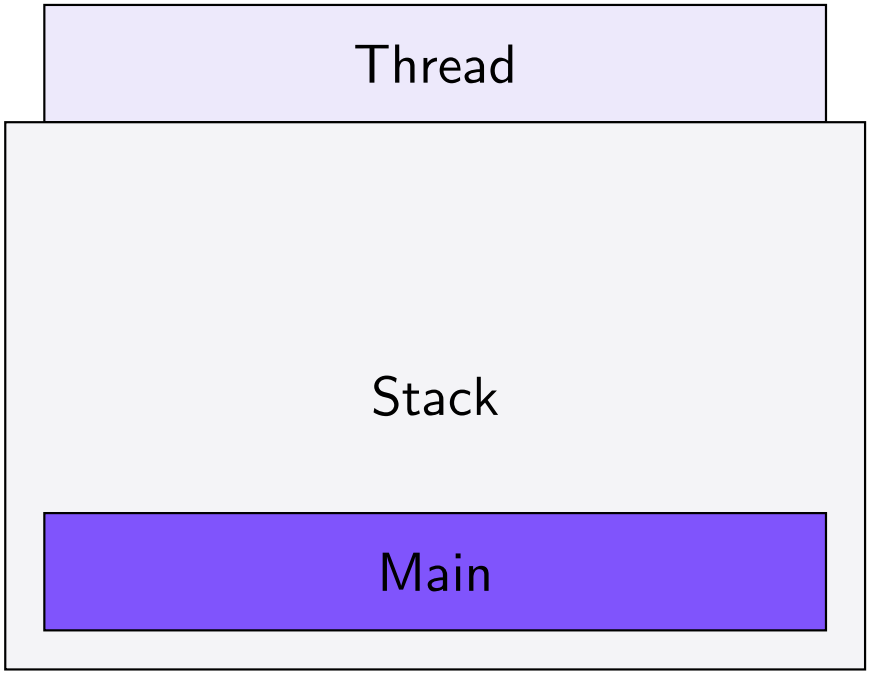
```
launch {
    //coroutine
}
```

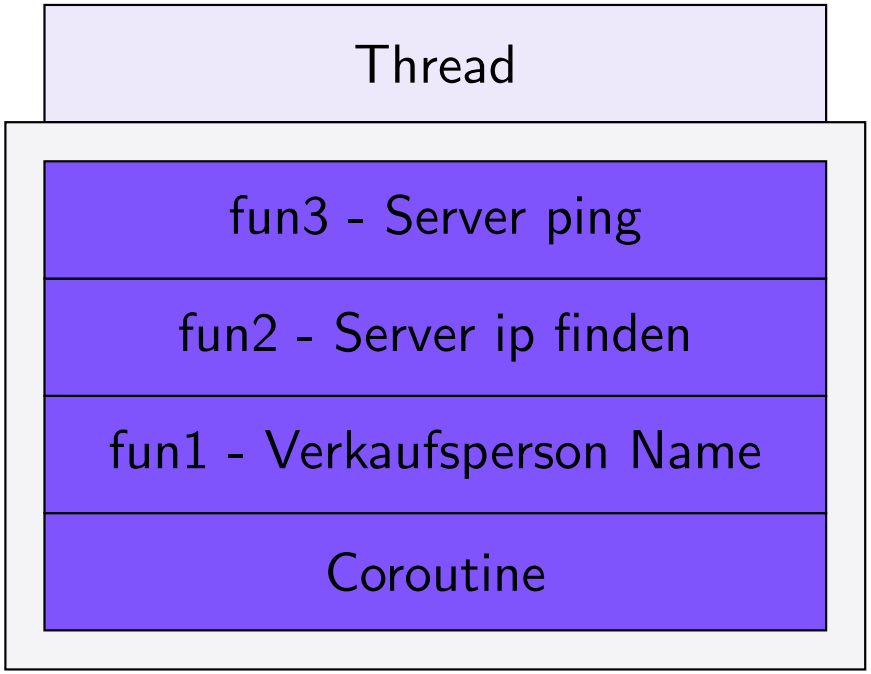
Android: Coroutines



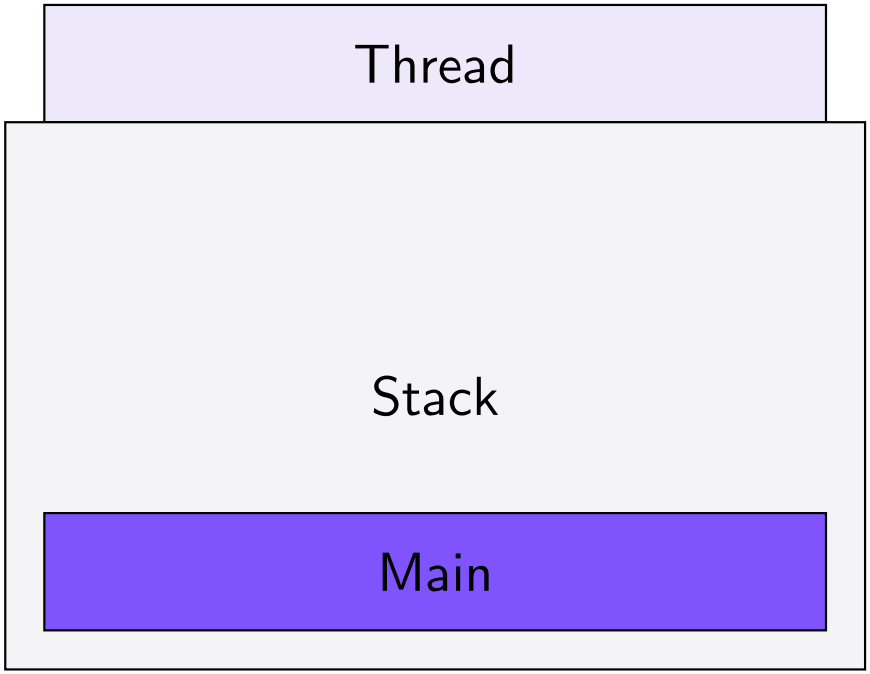


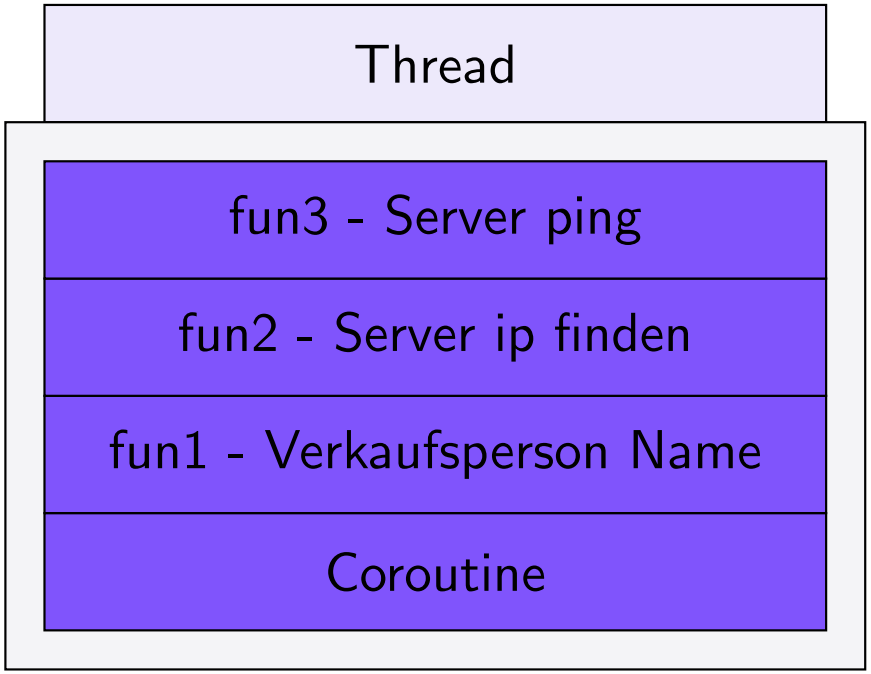
Android: Coroutines





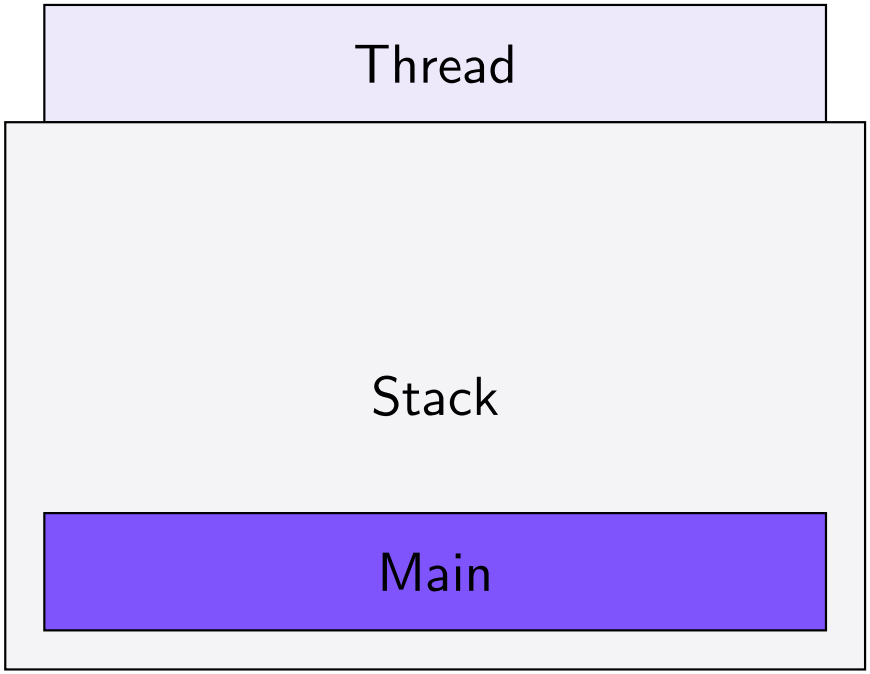
Android: Coroutines

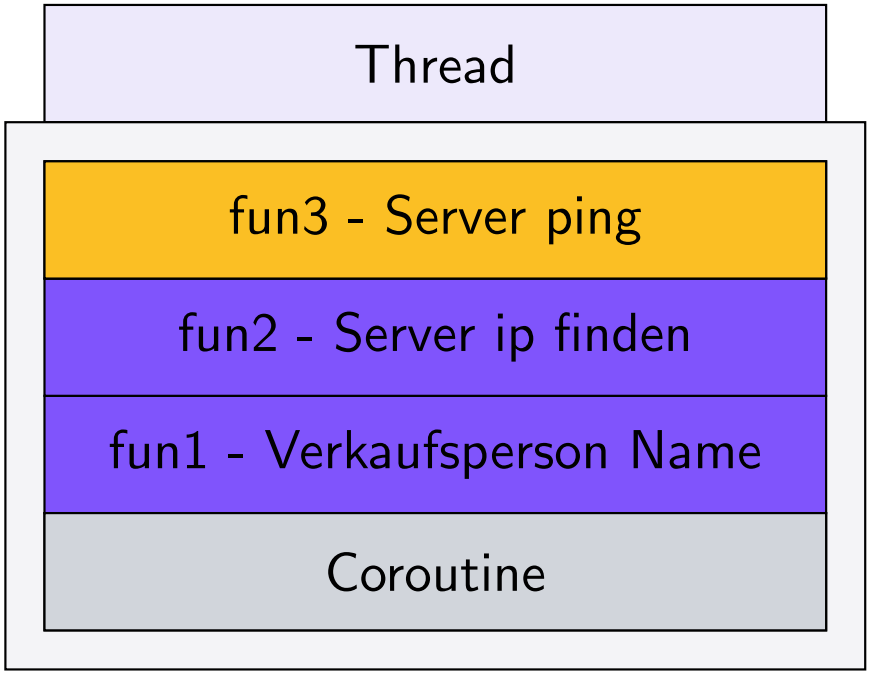





```
1  suspend fun pingServer() {
2          //ping Server
3      }
```

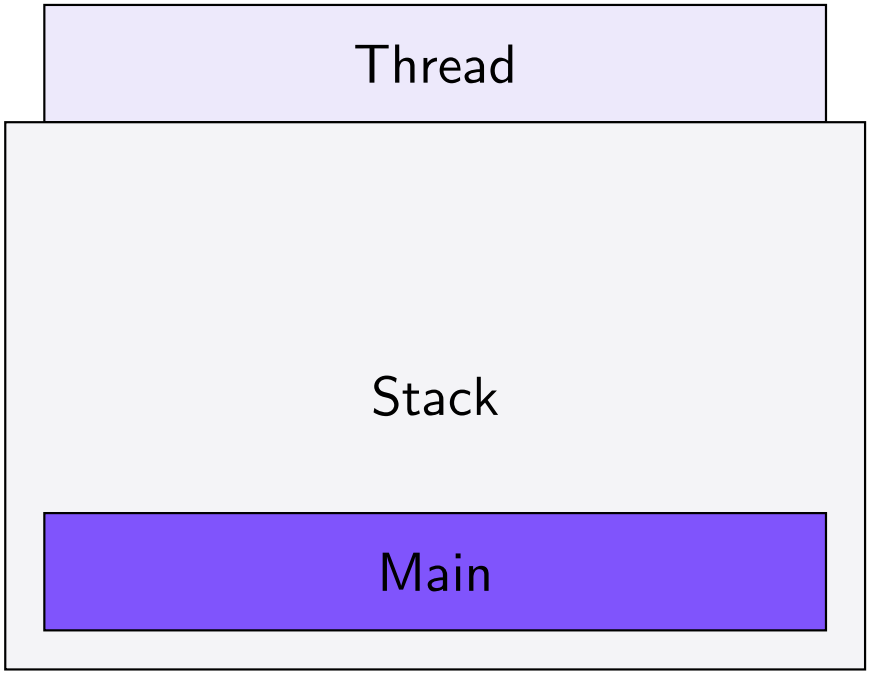
Android: Coroutines

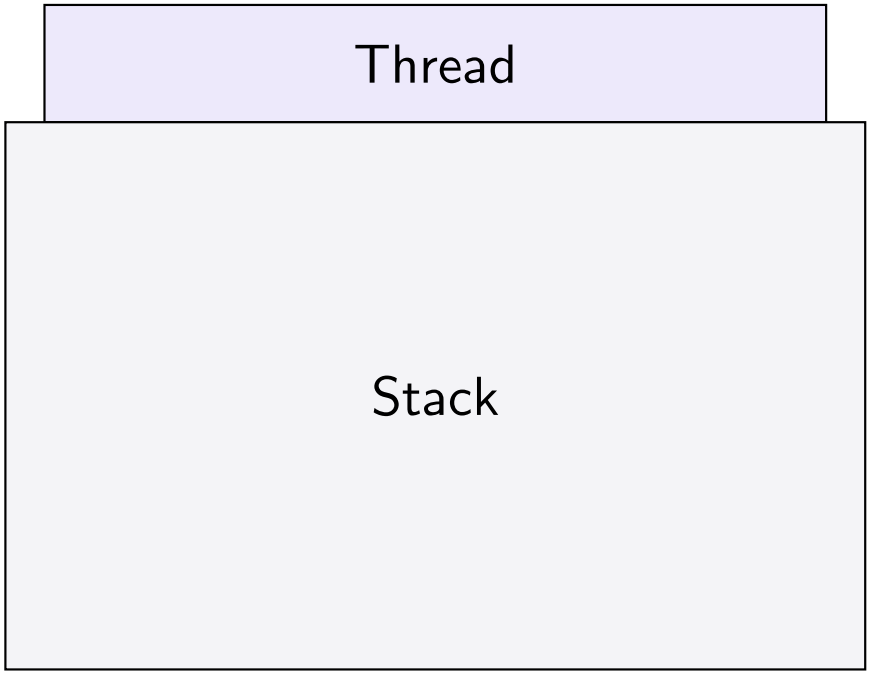





```
1  suspend fun pingServer() {
2          //ping Server
3      }
```

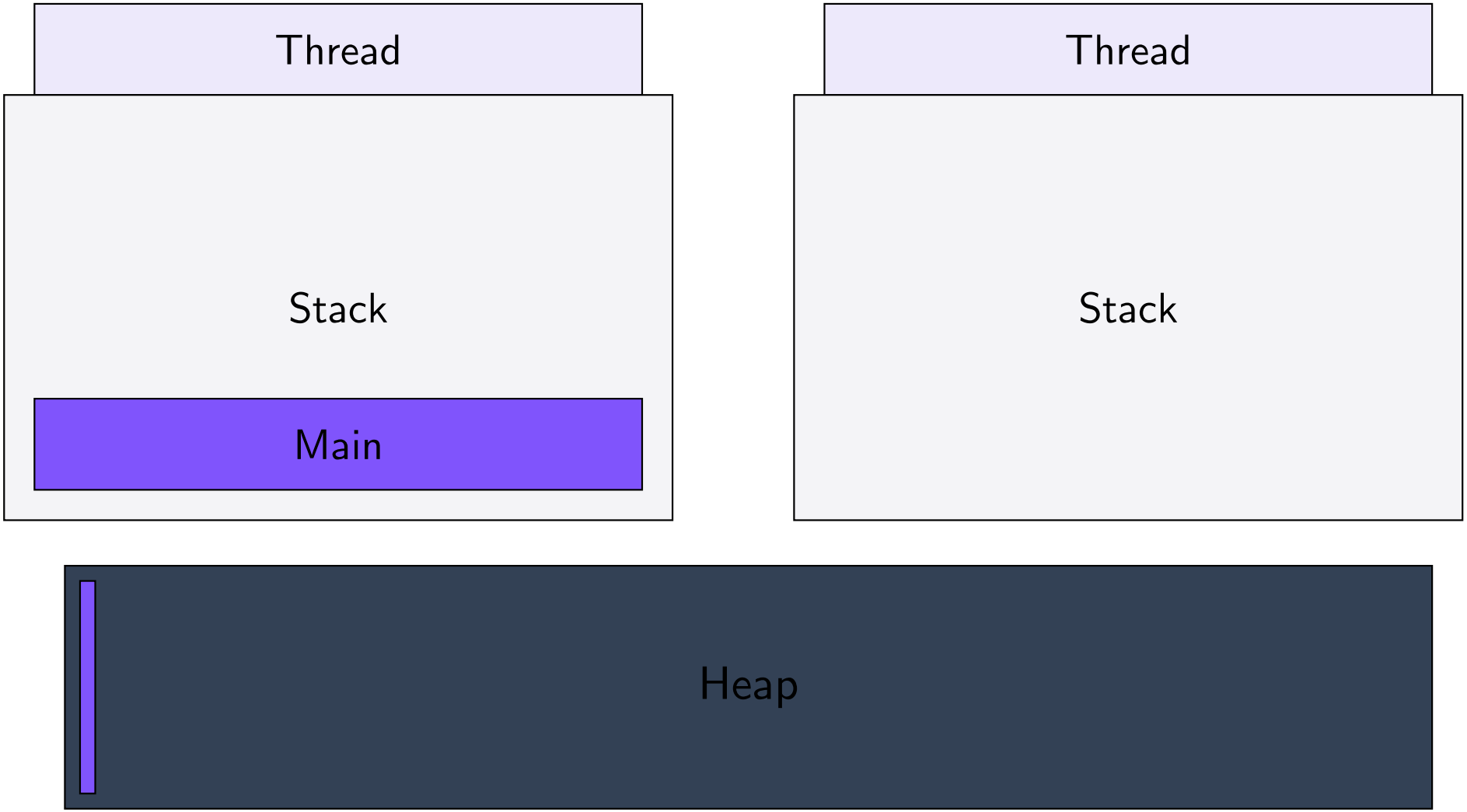
Android: Coroutines



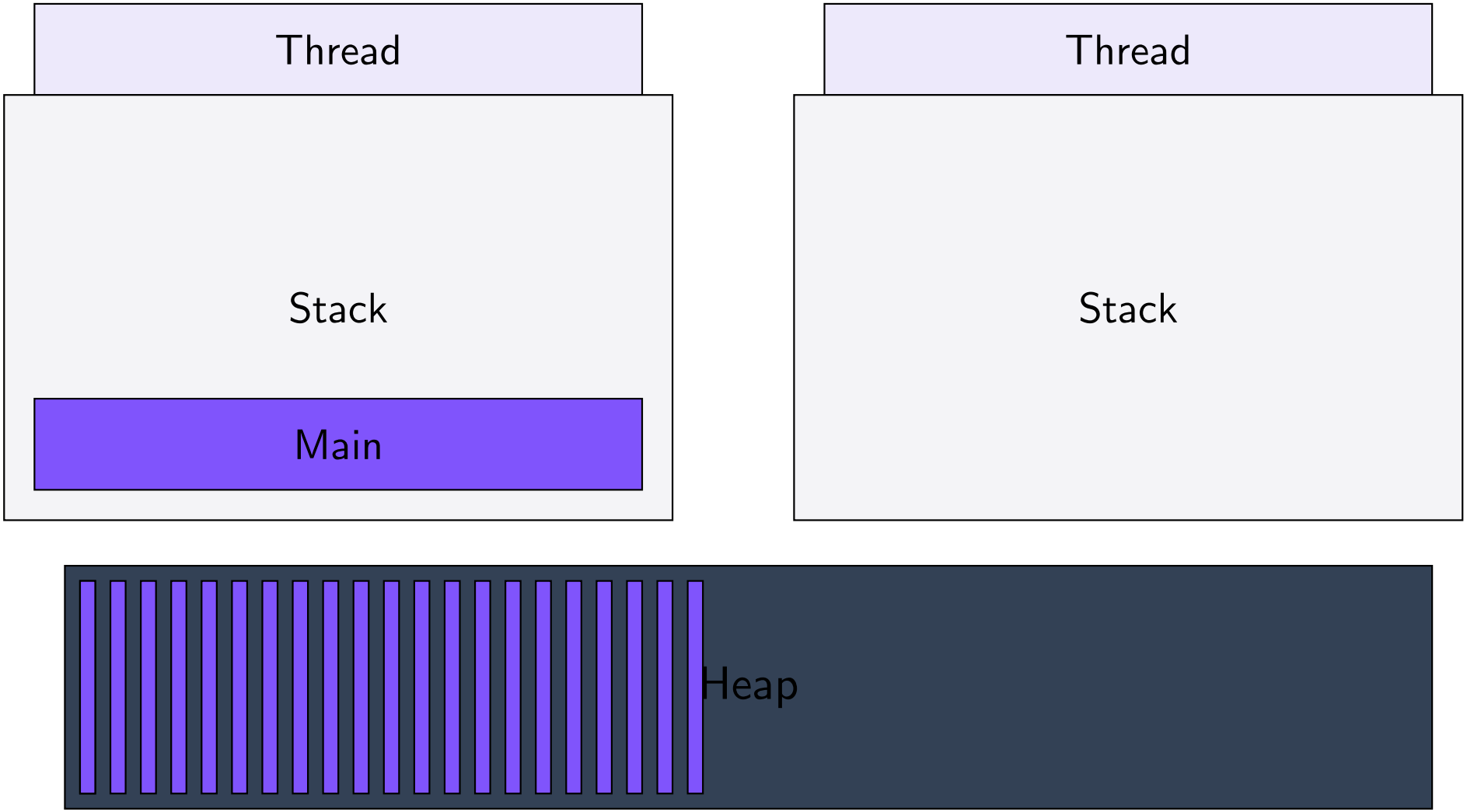



```
Coroutine:
 Variablen:        Name,Ip
   State:             1
    Path:     fun1-fun2-fun3
```

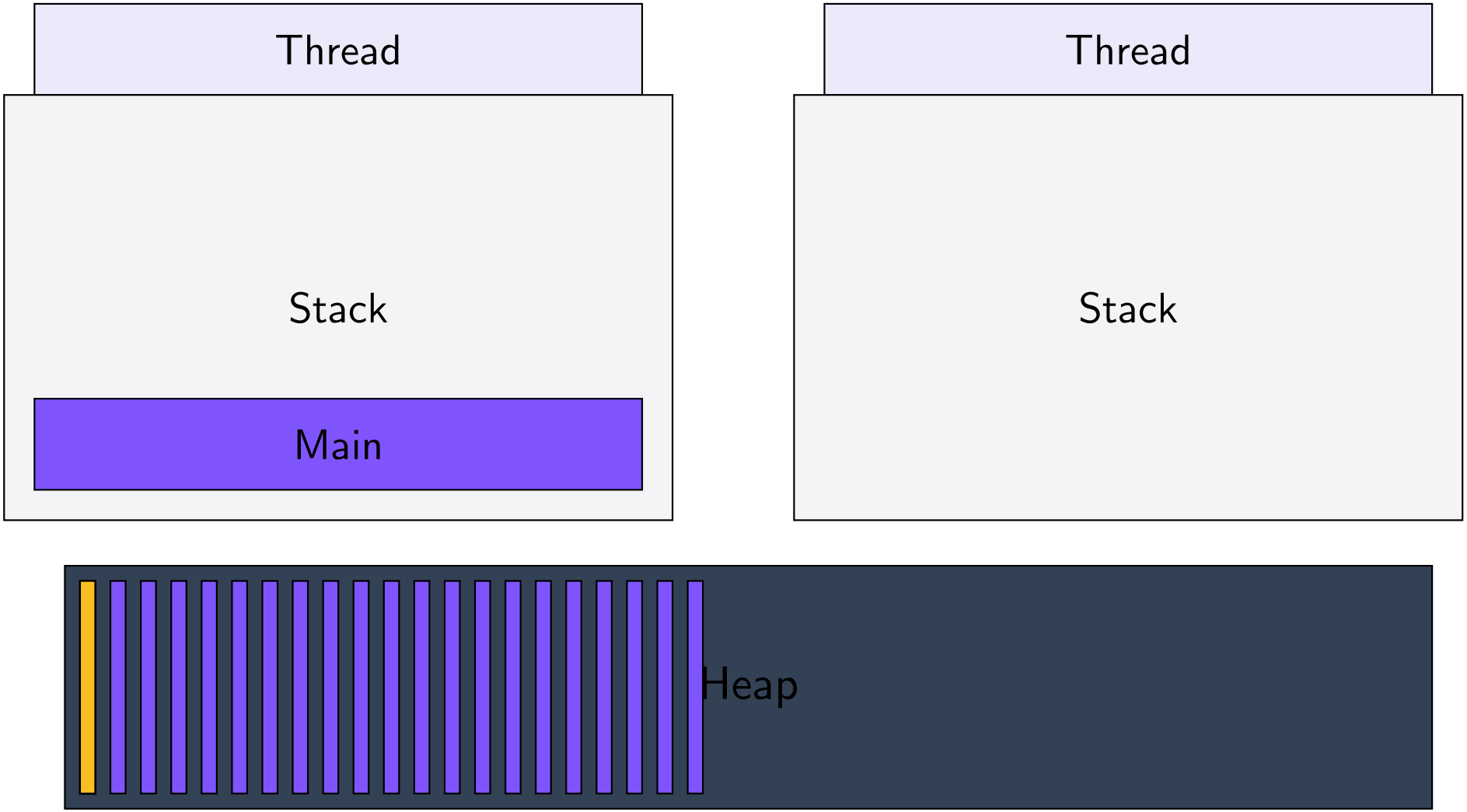
Android: Coroutines



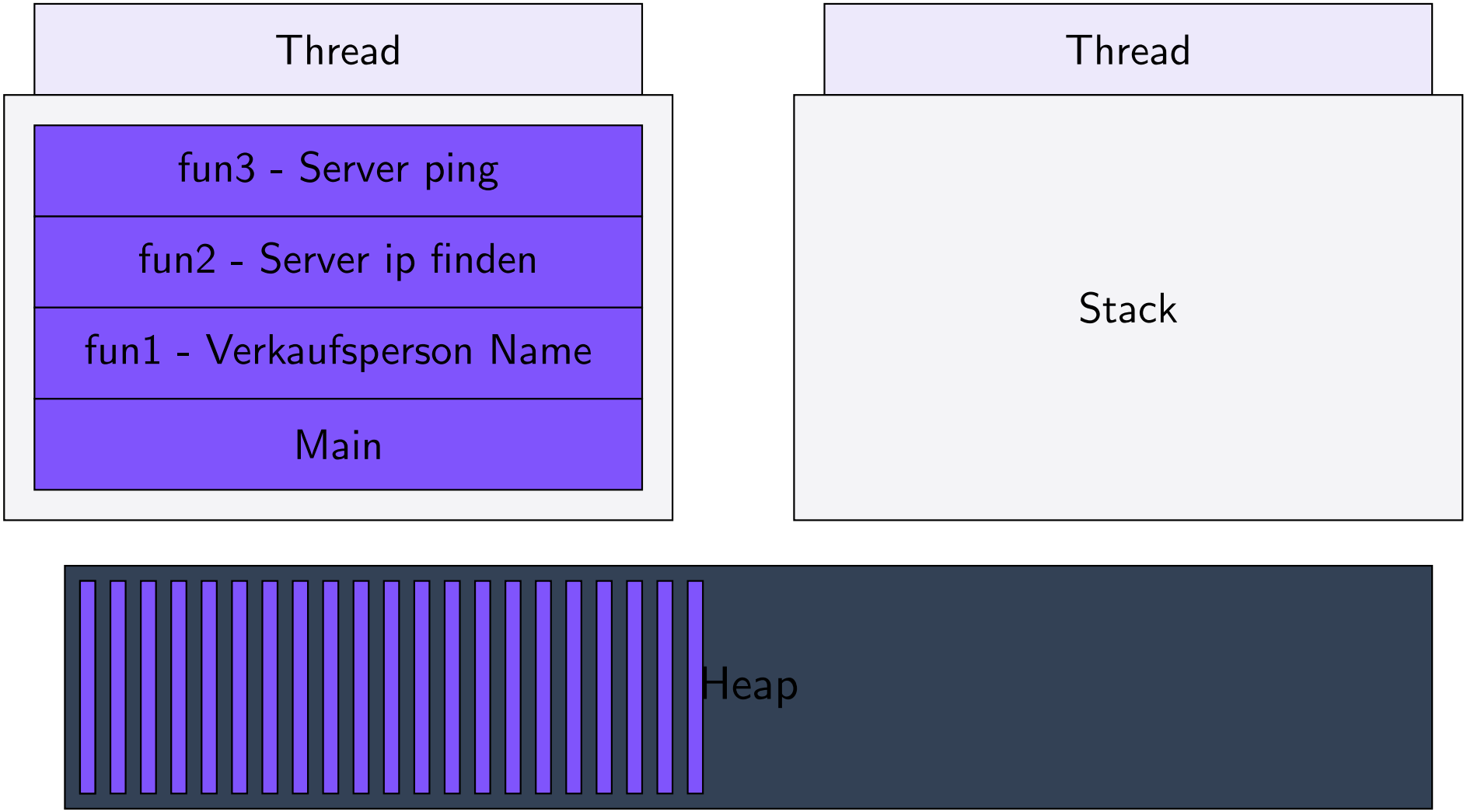
Android: Coroutines



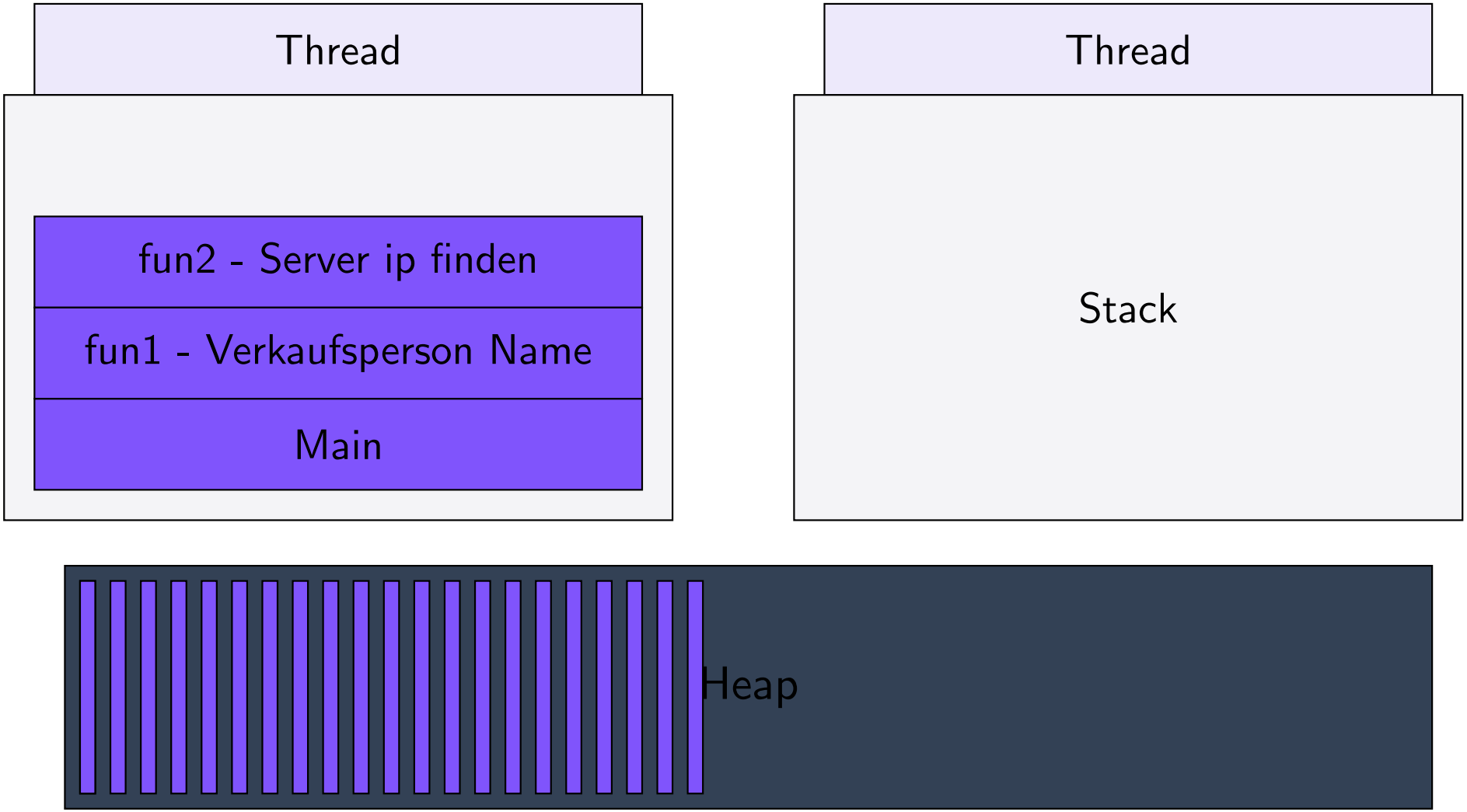
Android: Coroutines



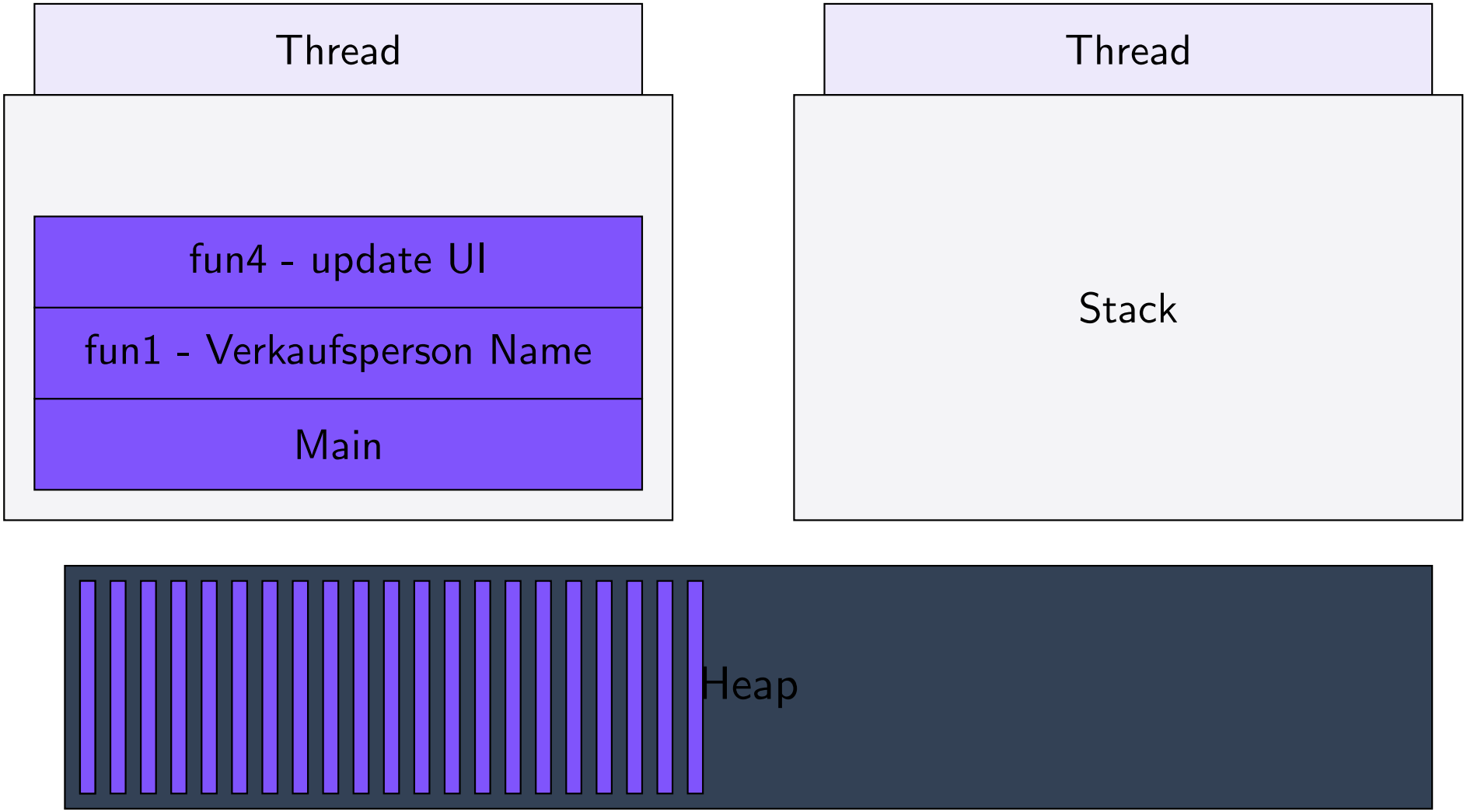
Android: Coroutines



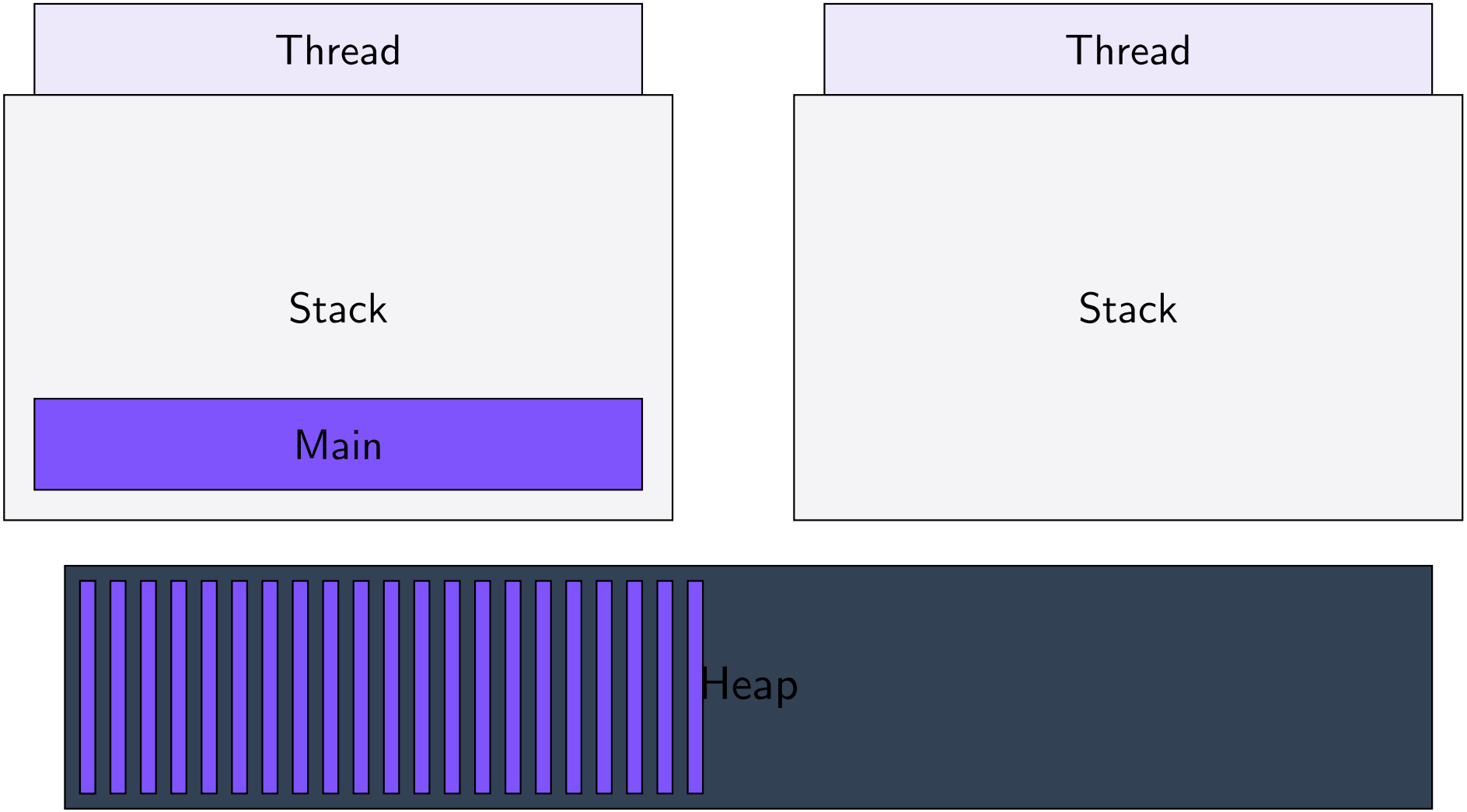
Android: Coroutines



Android: Coroutines



Android: Coroutines



Zusammenfassung

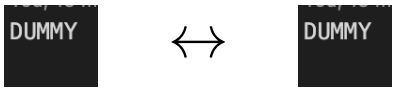
- Moderne Programmiersprache mit präziser Syntax

  ```
  public static void main(String[] args) ⇒ fun main()
  ```

- Innovative Features wie Null-Sicherheit

  ```
  verkaufsperson?.vorgesetzter ?: "Kein␣Vorgesetzter"
  ```

- Nahtlose Interoperabilität mit Java

- Multiplattform-Entwicklung (Android)

Zusammenfassung

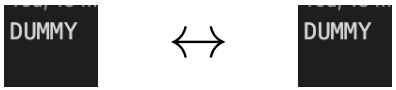
- Moderne Programmiersprache mit präziser Syntax

```
public static void main(String[] args) ⇒ fun main()
```

- Innovative Features wie Null-Sicherheit

```
verkaufsperson?.vorgesetzter ?: "Kein␣Vorgesetzter"
```

- Nahtlose Interoperabilität mit Java



- Multiplattform-Entwicklung (Android)

Aussicht:

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