Kotlin

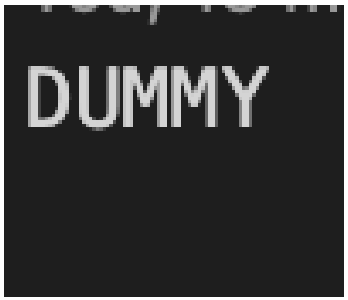
Proseminar: Fortgeschrittene Programmierkonzepte

Christian Konersmann, Finn Paul Lippok, Paul Lukas

05.05.2025

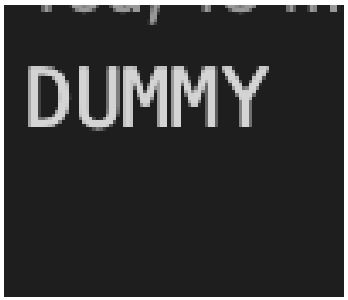
Was ist Kotlin?

- **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.

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!")
}
```

- Keine explizite Klassendeklaration 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!")
}
```

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

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!")
}
```

- Keine explizite Klassendeklaration erforderlich.
- Verwendung des Schlüsselworts `fun` zur Funktionsdeklaration.
- Standardzugriffsmodifikator ist `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!")
}
```

- Keine explizite Klassendeklaration erforderlich.
- Verwendung des Schlüsselworts `fun` zur Funktionsdeklaration.
- Standardzugriffsmodifikator ist `public`.
- `args`-Parameter ist 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!")
}
```

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

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.
- In Kotlin gibt es **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.
- In Kotlin gibt es **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.
- 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.

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.

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 }
```

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

Null Safety

Motiviation: Null Safety

Null Safety

Motiviation: Null Safety

Java Beispiel

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

Null Safety

Motiviation: Null Safety

Java Beispiel

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

- Code wirft `java.lang.NullPointerException`

Null Safety

Motiviation: 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

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

```java
private final Verkaufsperson vorgesetzer;

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

in Kotlin

```kotlin
val vorgesetzer: Verkaufsperson? = null

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

Null Safety: Safe call Operator

Verkettung des Operators

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

Null Safety: Safe call Operator

Verkettung des Operators

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

Zuweisungen mit dem Operator

```
vorgesetzer?.vorgesetzer?.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*

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);
}
```

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

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

Null Safety: Not-null assertion Operator

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

- Kann zu NullPointerExceptions 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()
```

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

```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
```

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

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

Interoperabilität: Mapped types

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

- `java.lang.Object` ⇒`kotlin.Any!`

Interoperabilität: Mapped types

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

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

Interoperabilität: Mapped types

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

- `java.lang.Object` ⇒`kotlin.Any!`
- `java.lang.Integer` ⇒ `kotlin.Int?`
- primitive typ `int` ⇒ `kotlin.Int`

Interoperabilität: Mapped types

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

- `java.lang.Object` ⇒`kotlin.Any!`
- `java.lang.Integer` ⇒ `kotlin.Int?`
- primitive typ `int` ⇒ `kotlin.Int`
- Rückgabewert `void` ⇒ `Unit`

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)
```

- Aus Java zurückgegebene Instanzen können *null* sein

Interoperabilität: Null safety mit Java

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

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

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

Interoperabilität: Null safety mit Java

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

```kotlin
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

Interoperabilität: Java Arrays in Kotlin

- 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

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

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

```kotlin
var name: String
```

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

Interoperabilität: Instance Fields

- Aufruf von Attributen durch Punkt-Notation

Interoperabilität: Instance Fields

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

Interoperabilität: Instance Fields

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

Interoperabilität: Instance Fields

- 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);
```

Interoperabilität: Instance Fields

- 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);
```

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

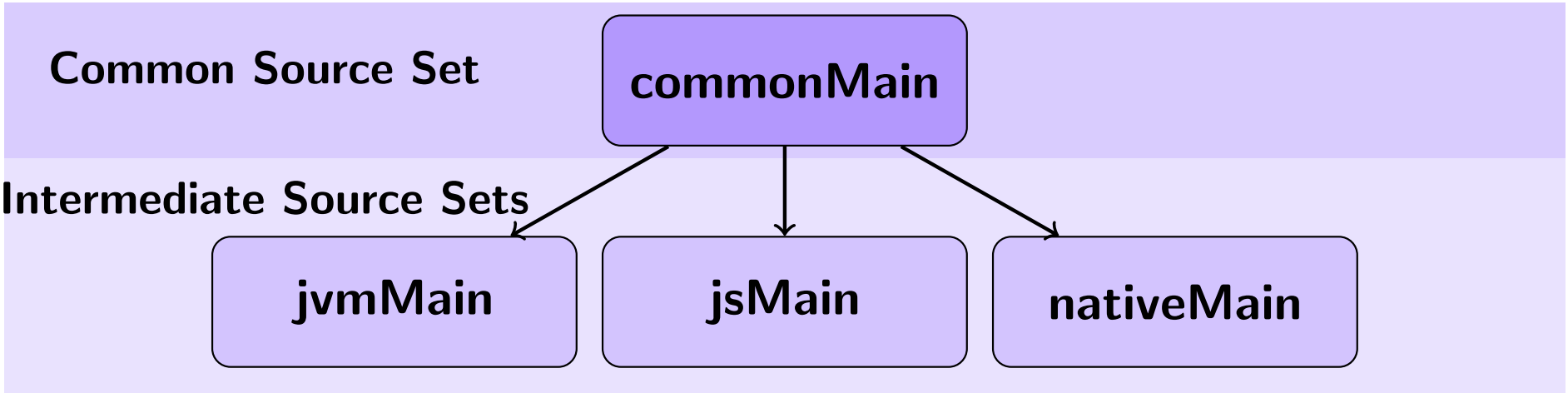
Multiplatform

- native binarie
- verschiedene Targets
- hierarchische Projektstruktur

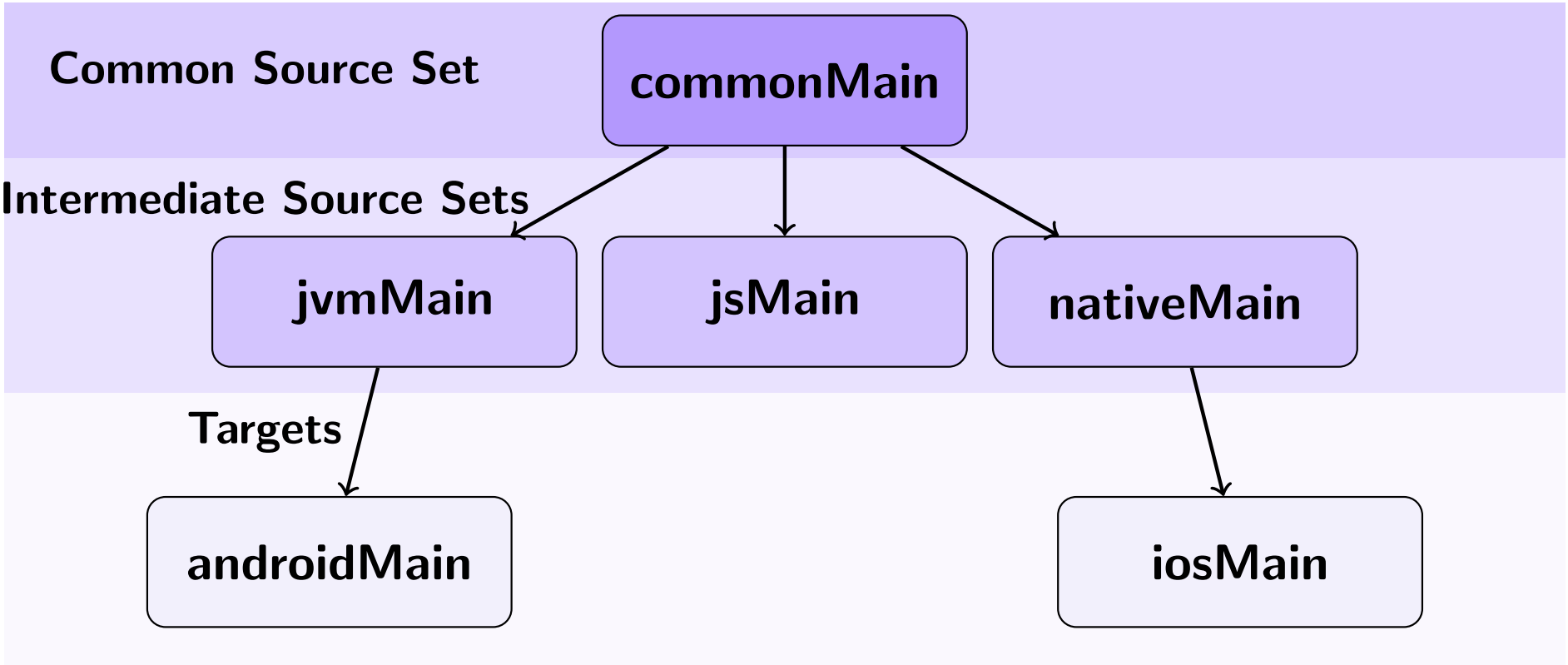
Multiplatform: hierarchische Projektstruktur

Common Source Set **commonMain**

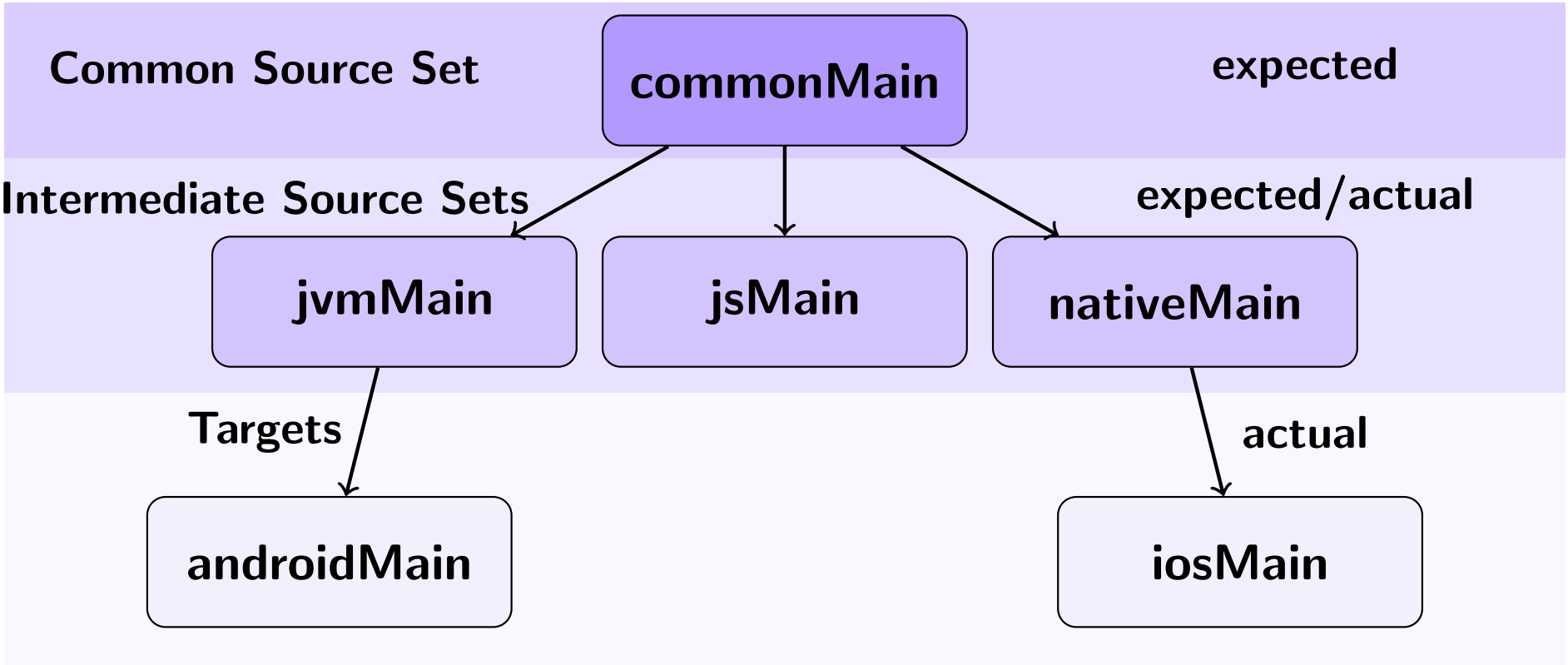
Multiplatform: hierarchische Projektstruktur



Multiplatform: hierarchische Projektstruktur



Multiplatform: hierarchische Projektstruktur



Android

- Jetpack compose
- Coroutines
- Beispiel

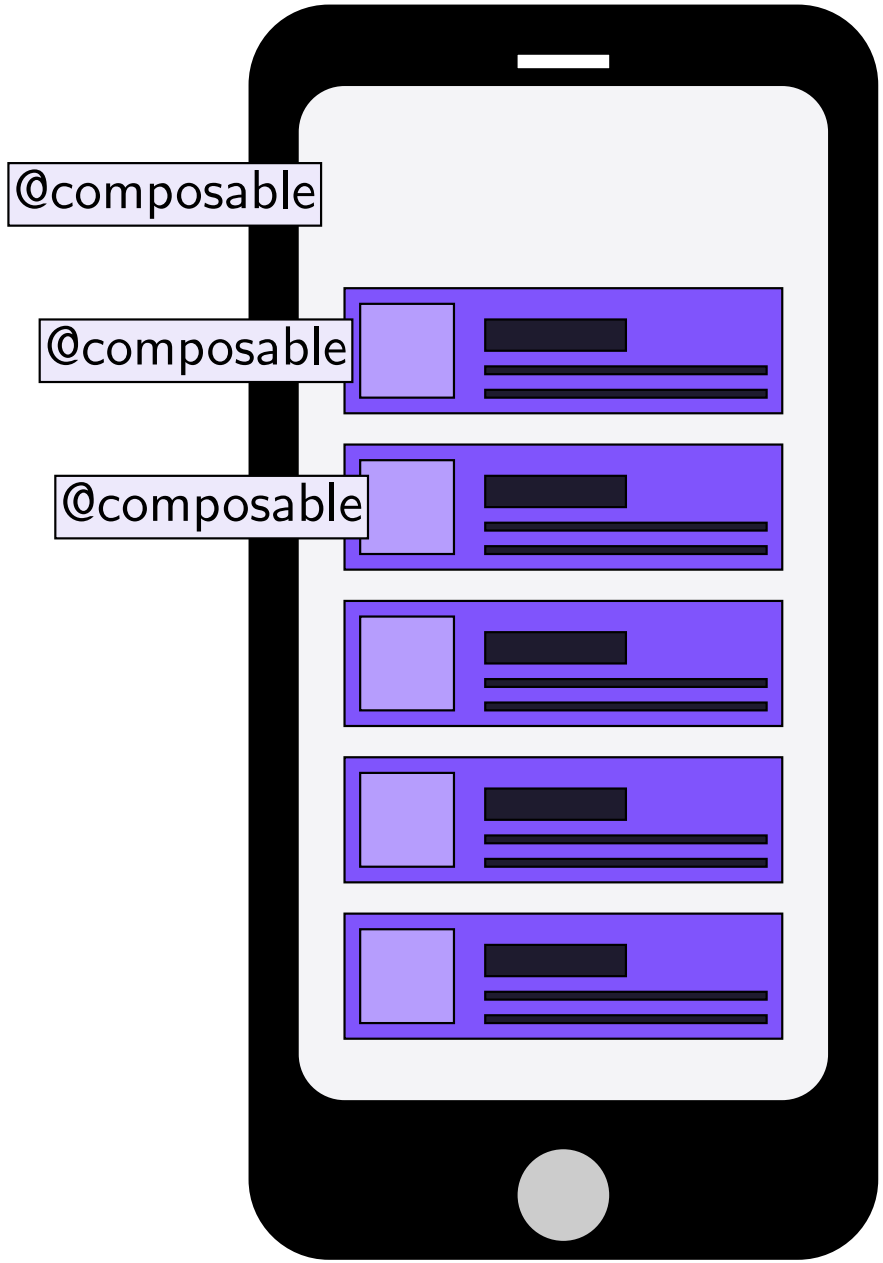
Android: Jetpack compose

- Ui Tool
- @composables
- Kotlin basiert

Android: Composable: Beispiel

States

Home

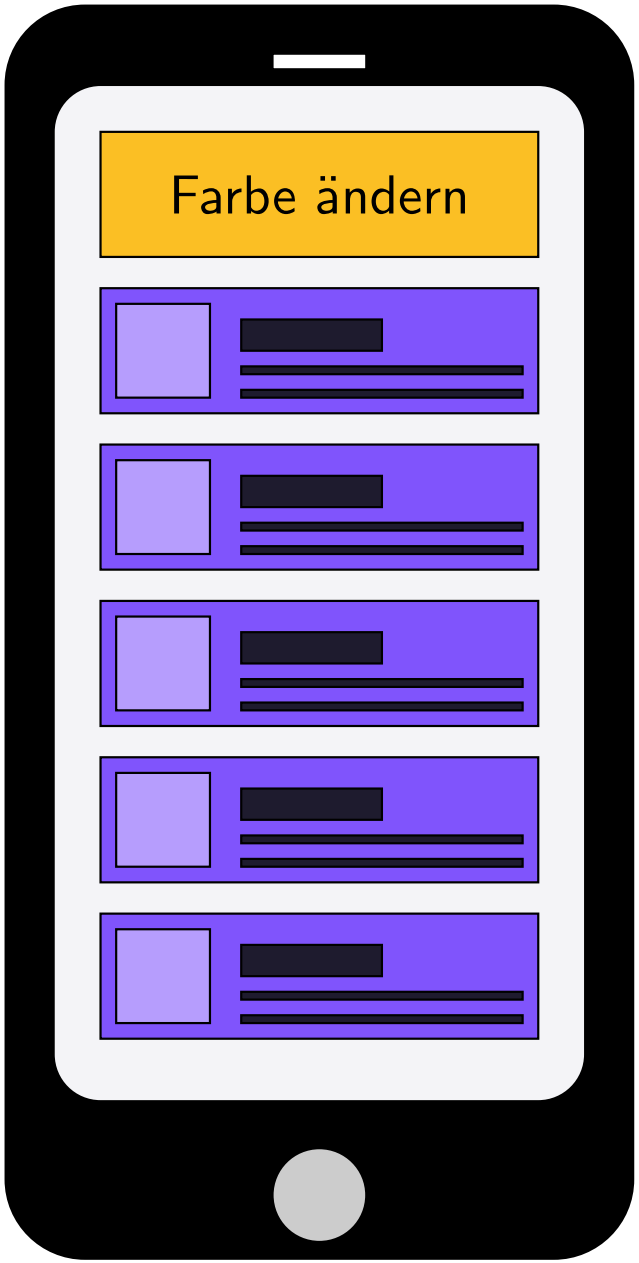


Android: Composable: Beispiel

States

Home

Farbe: Grau

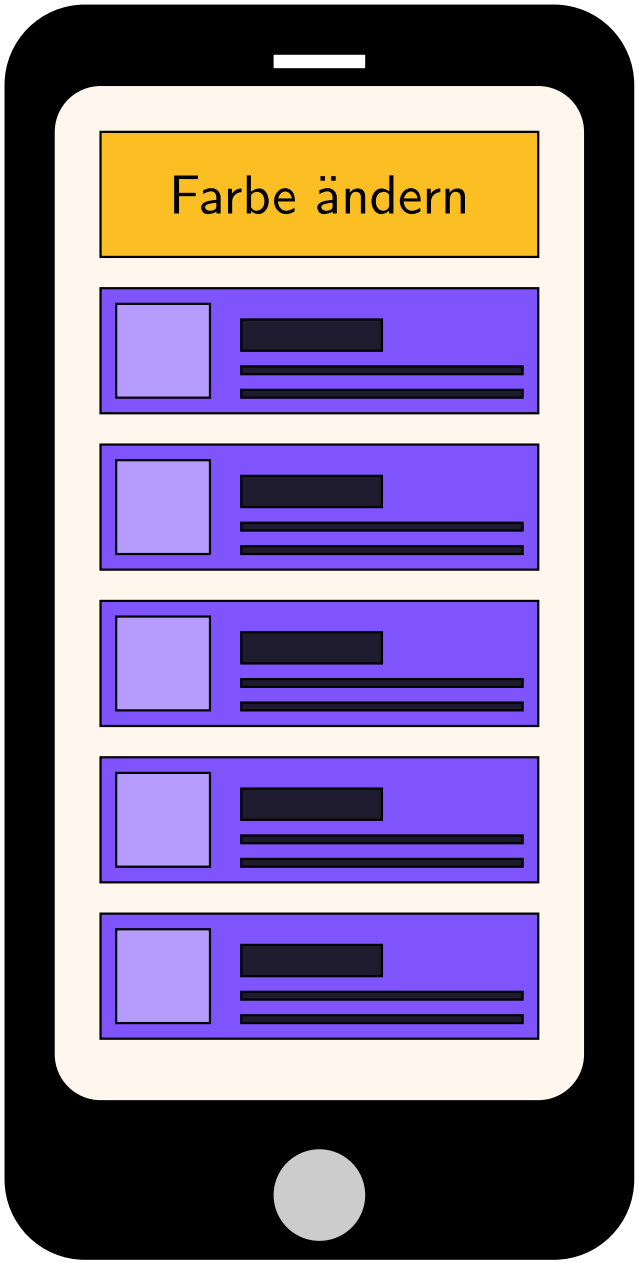


Android: Composable: Beispiel

States

Home

Farbe:
Grau Gelb

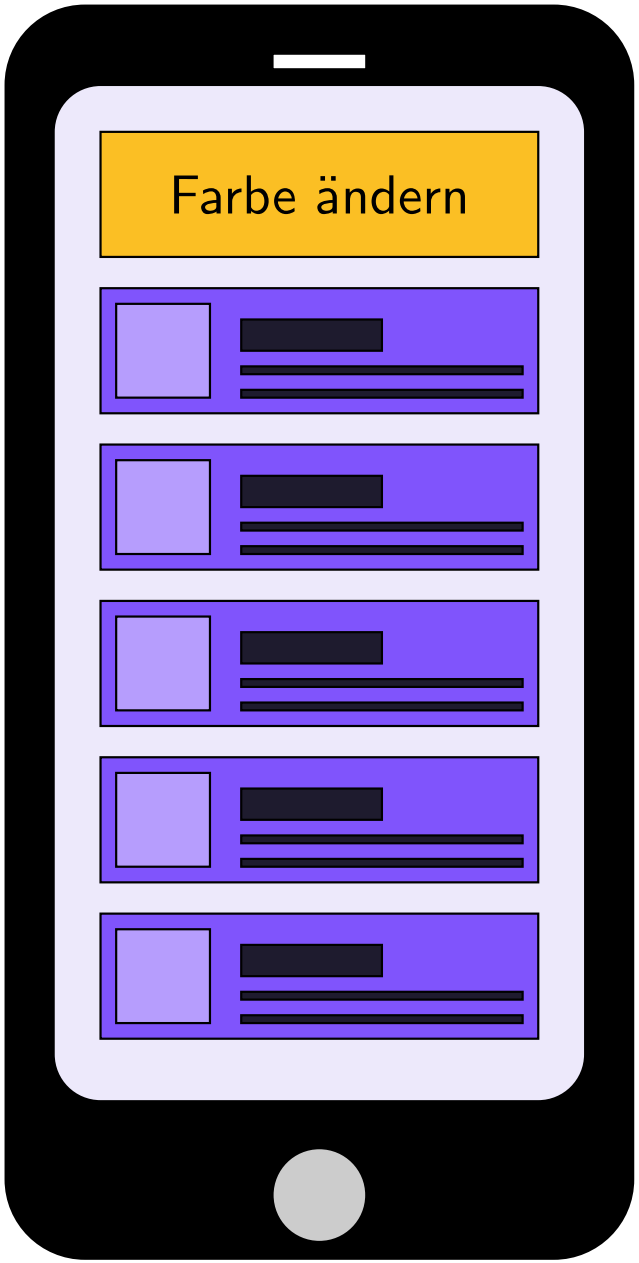


Android: Composable: Beispiel

States

Home

Farbe:
Grau Lila

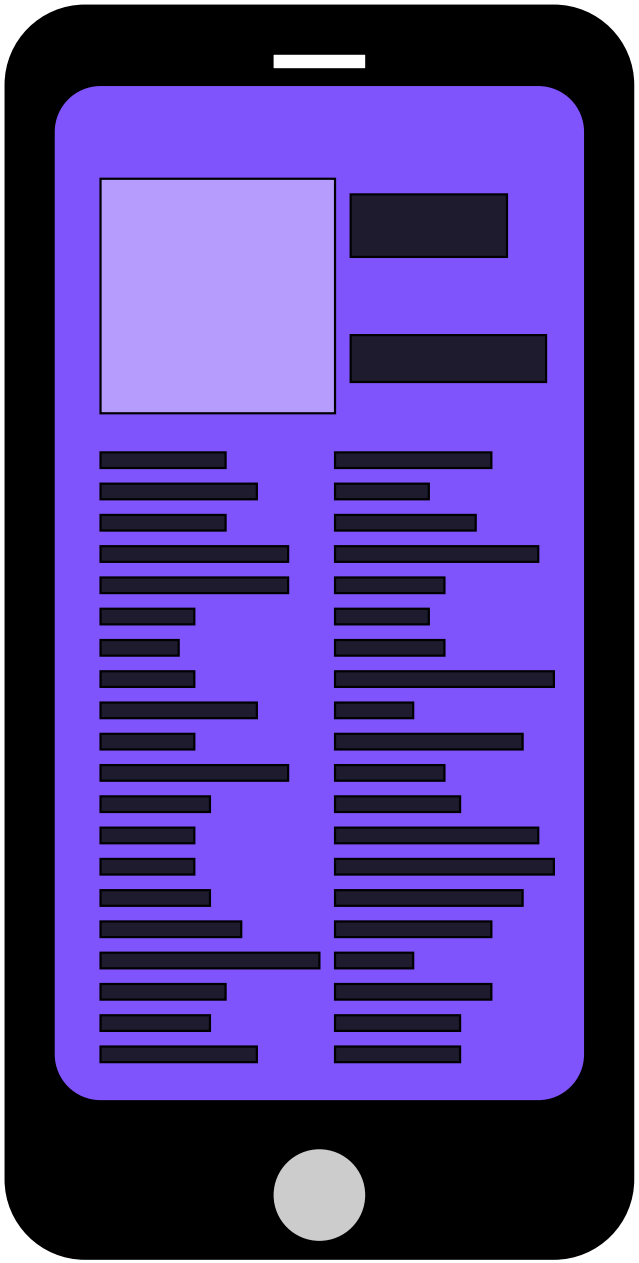


Android: Composable: Beispiel

States

Profil

Farbe:
Lila

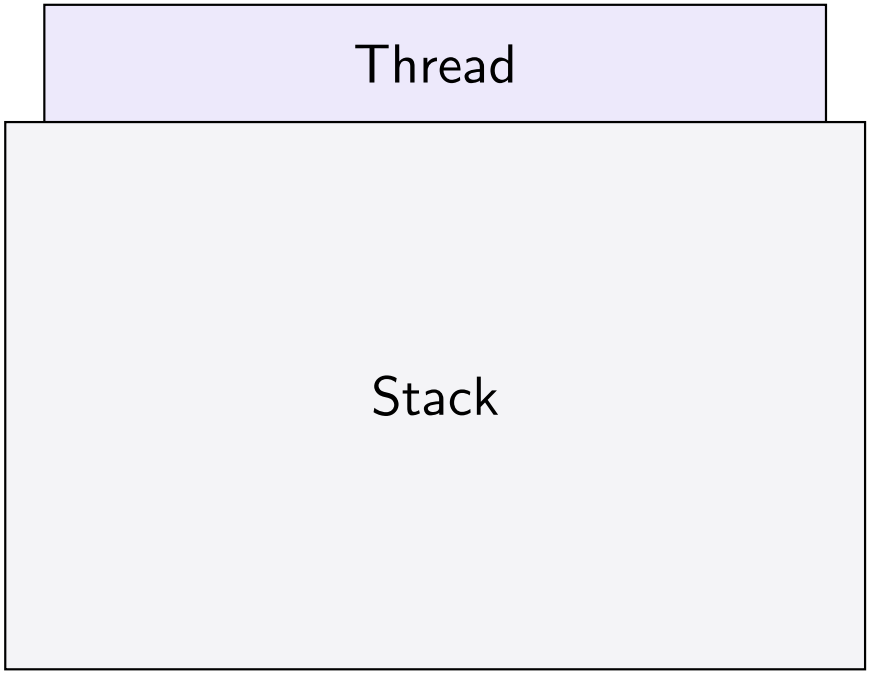


Android: Coroutines

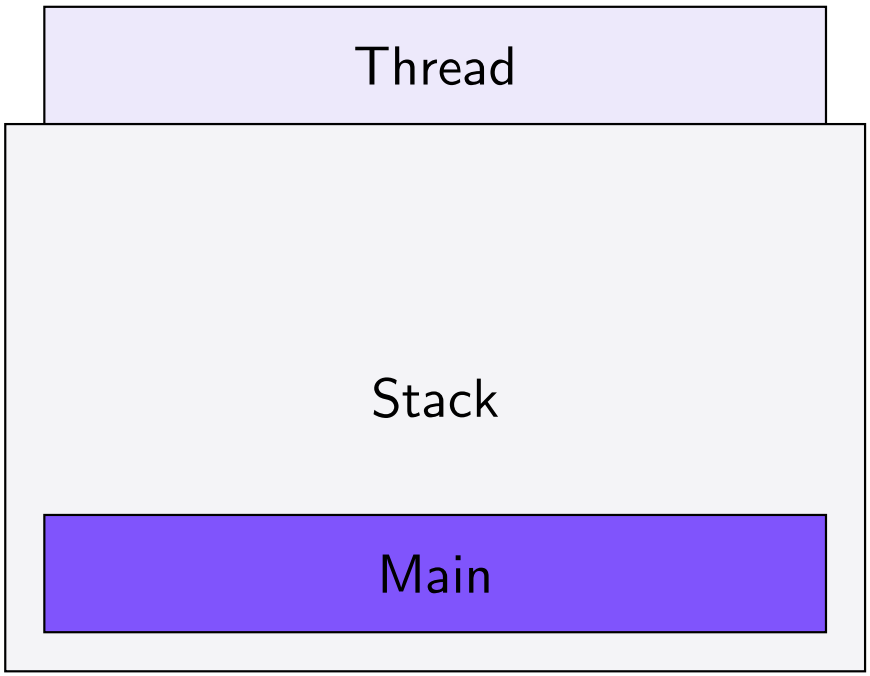
- Threads unterschiede
- Threads sind:
- subprocess
- Os heavy
- Build from(stack...)
- besseres organisier System
- Thread pools:

Pool name	(Max. Threads)	(Min. Threads)
Main	(1)	(1)
Default	(Cpu Cores)	(2)
I/O	(64)	(0)

Android: Coroutines



Android: Coroutines



-Subprocese

-Besteht aus:

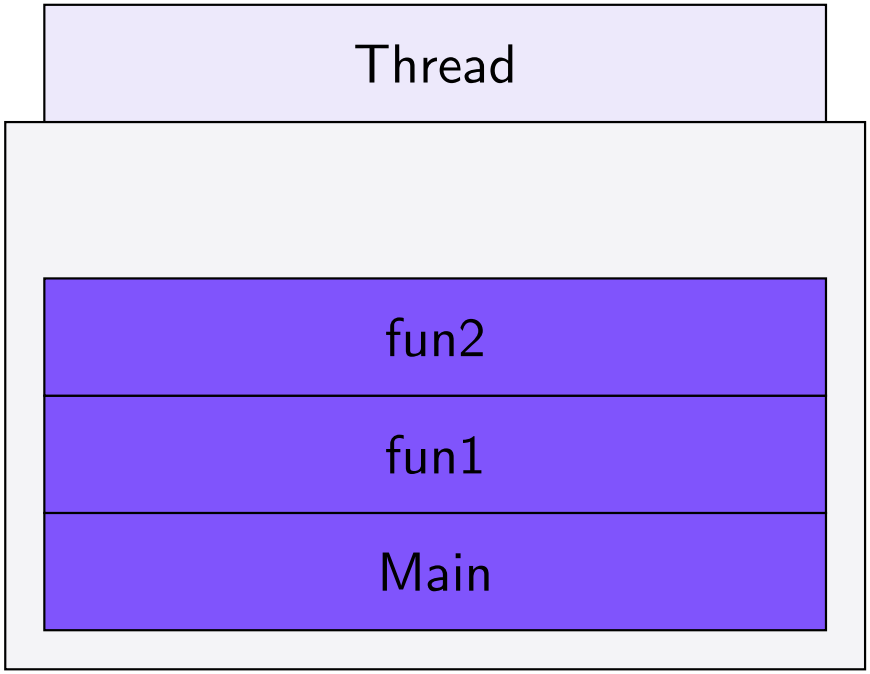
Stack

Thread Context

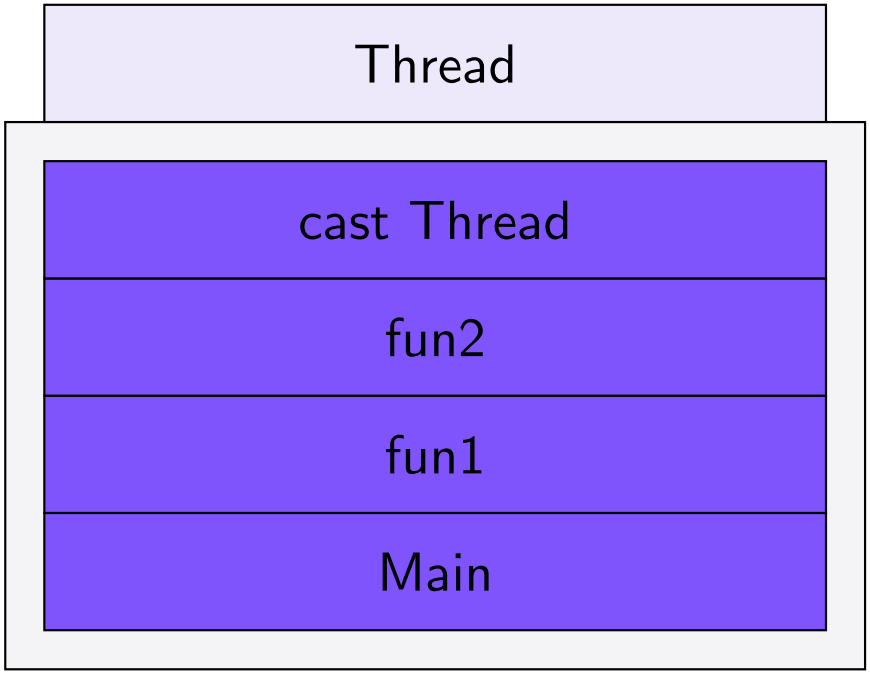
TCB

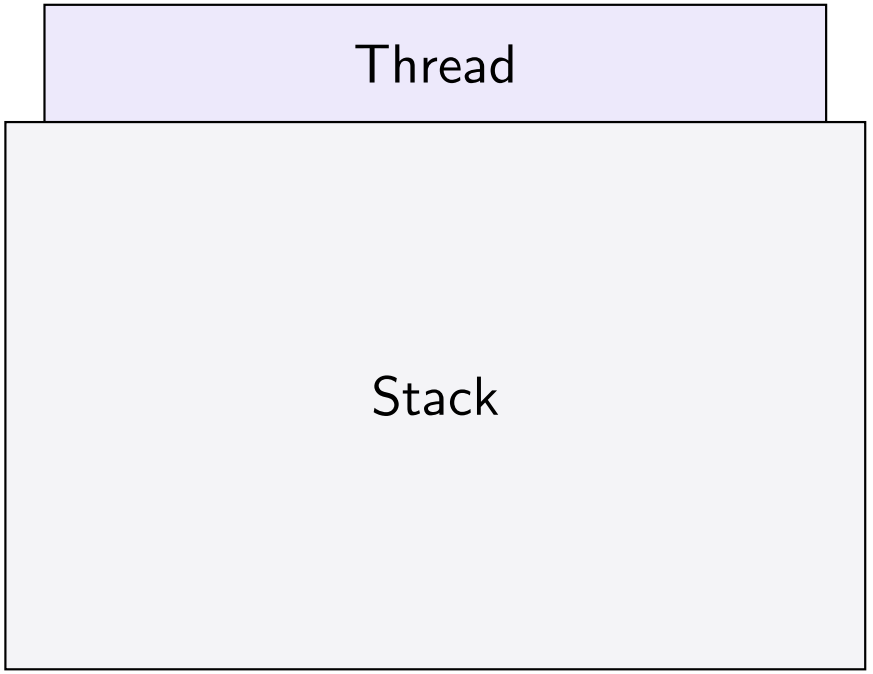
-OS Lastig

Android: Coroutines

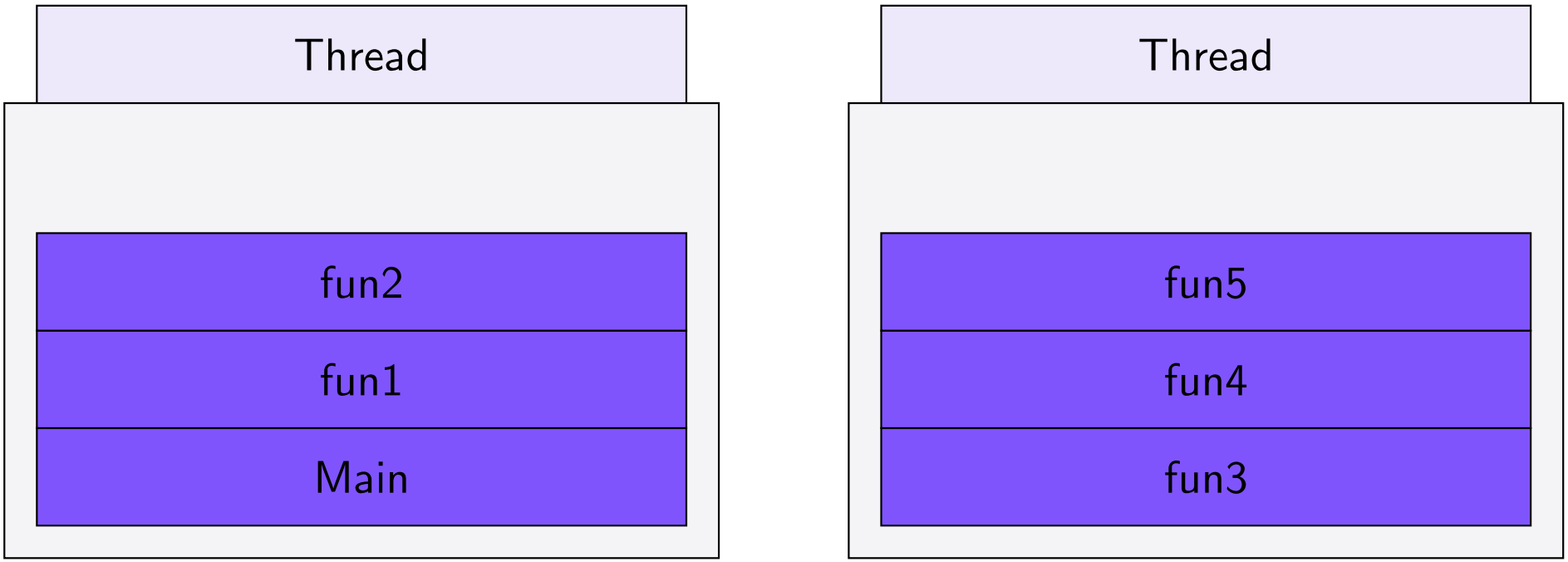


Android: Coroutines

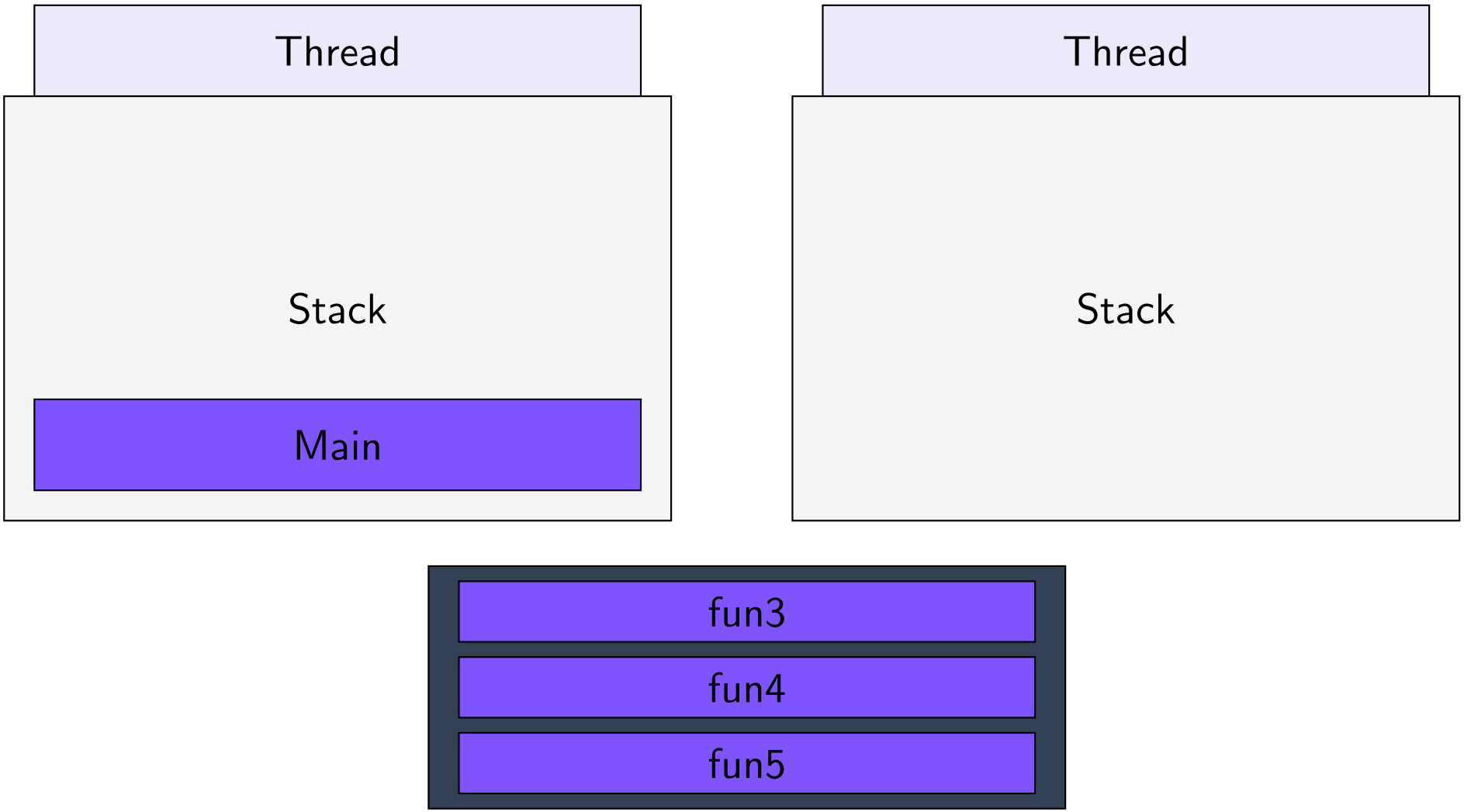




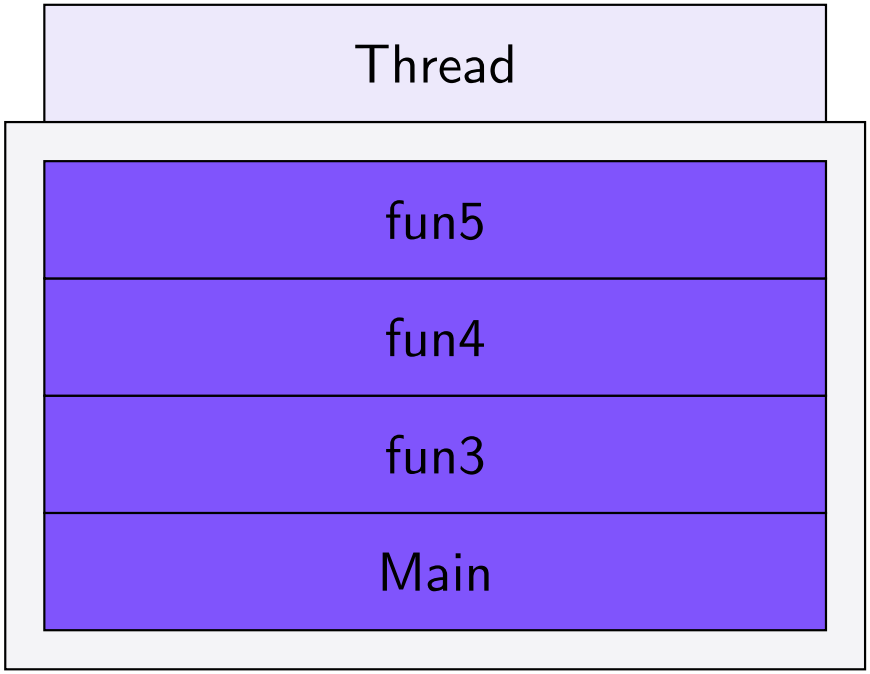
Android: Coroutines

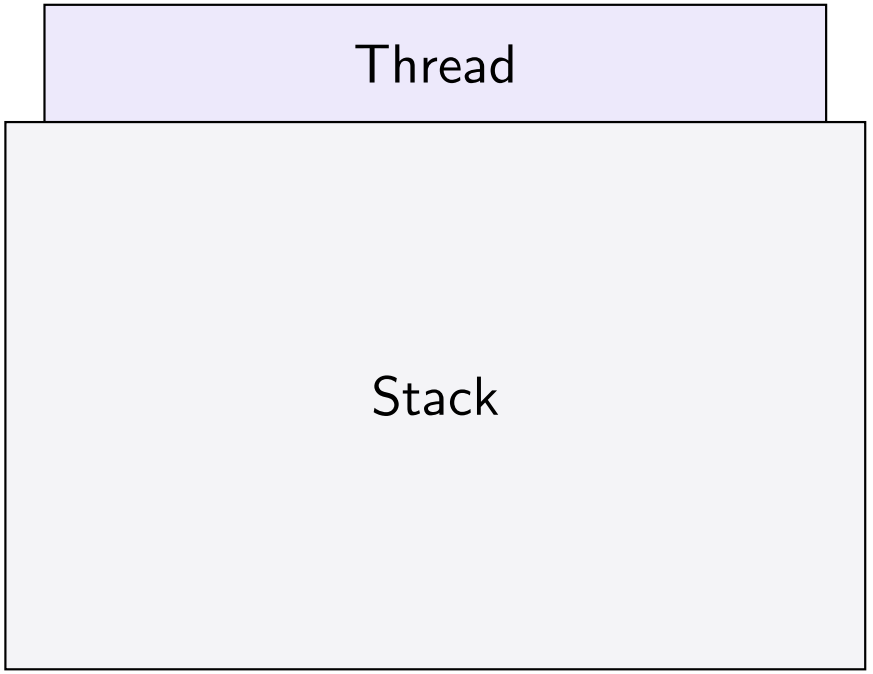


Android: Coroutines

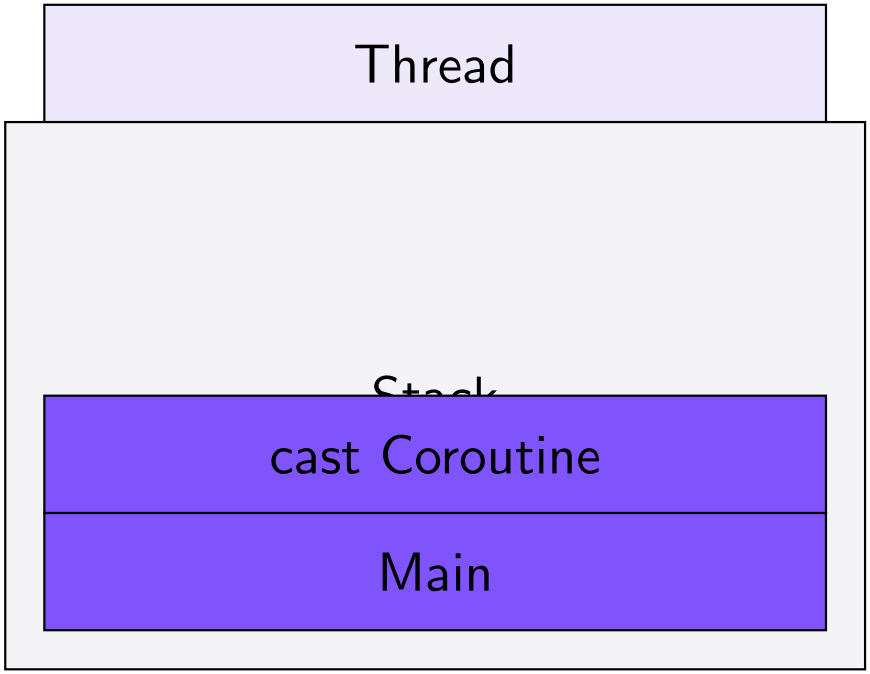


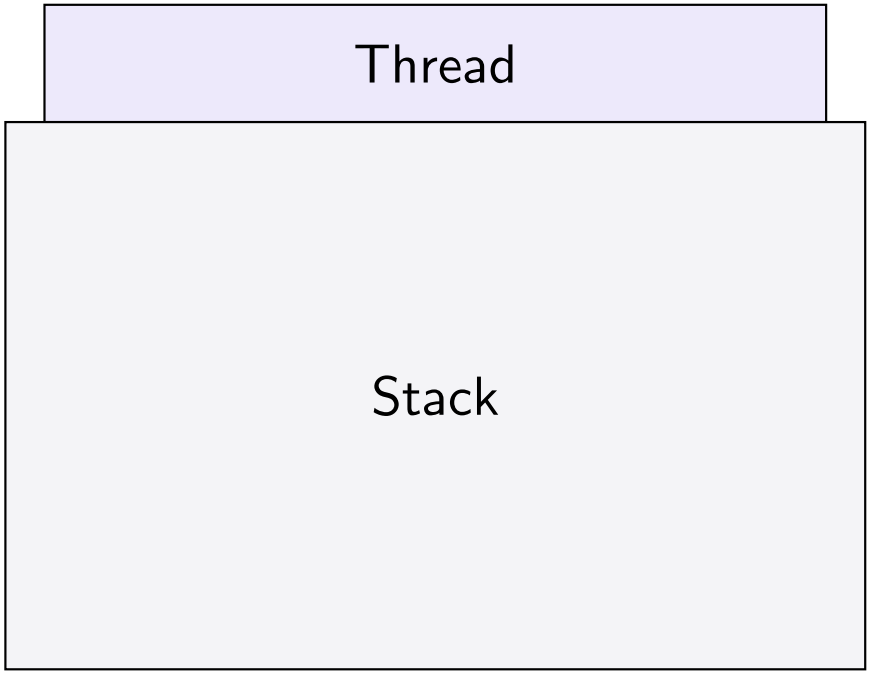
Android: Coroutines



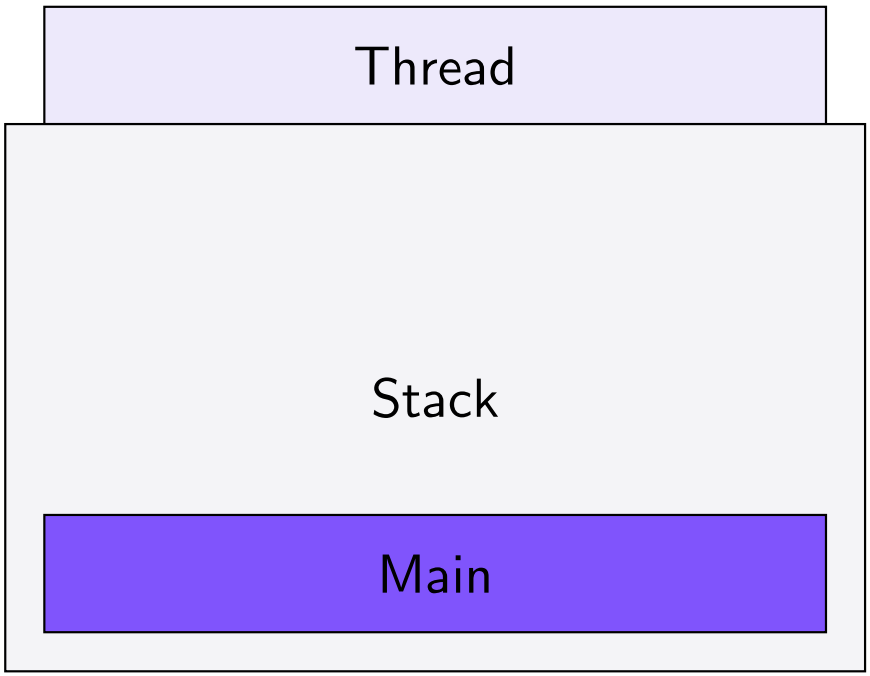


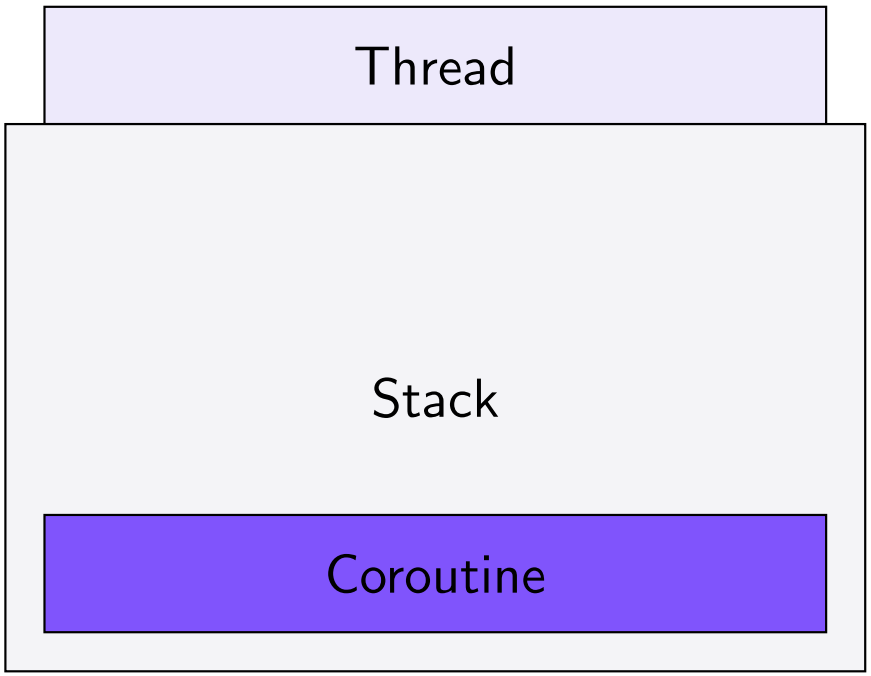
Android: Coroutines



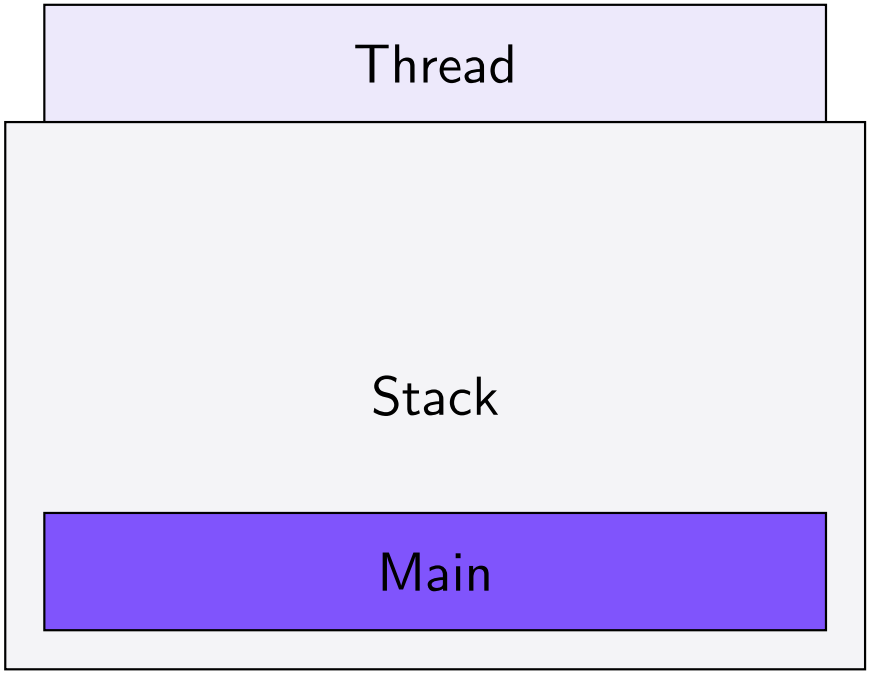


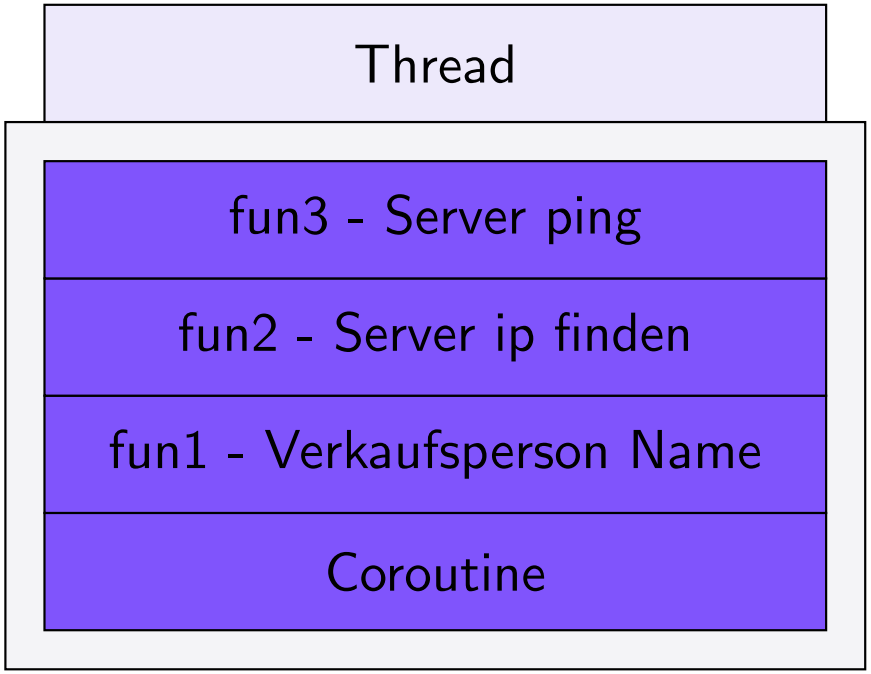
Android: Coroutines





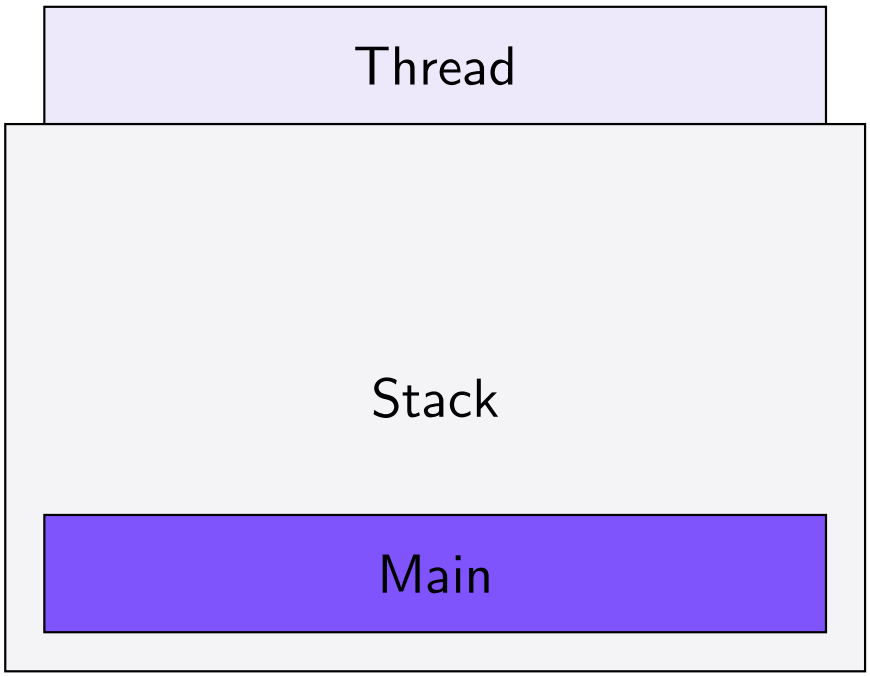
Android: Coroutines

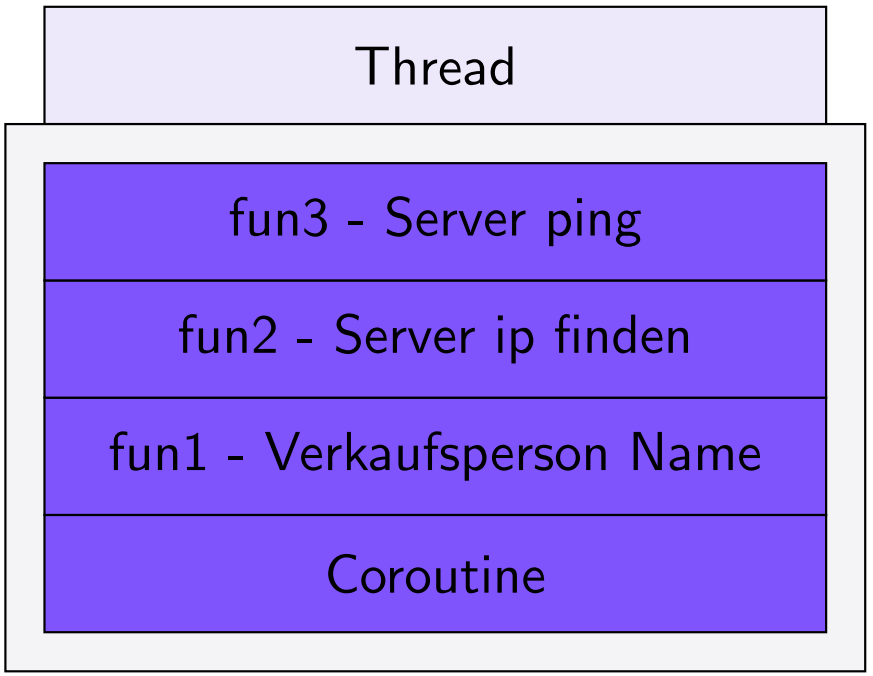




Android: Coroutines

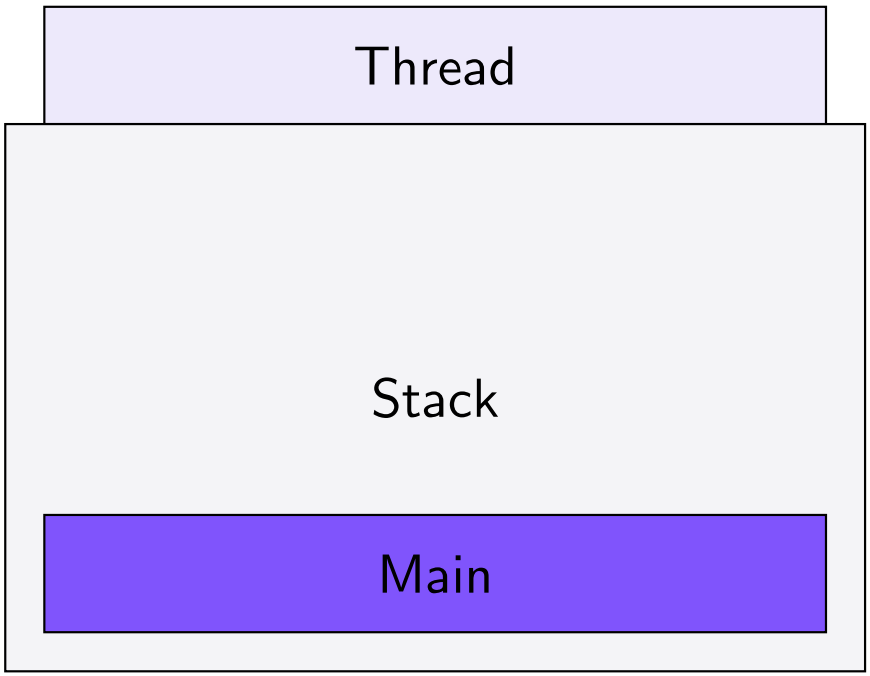
Suspends: delay(),yield(),withContext(),...

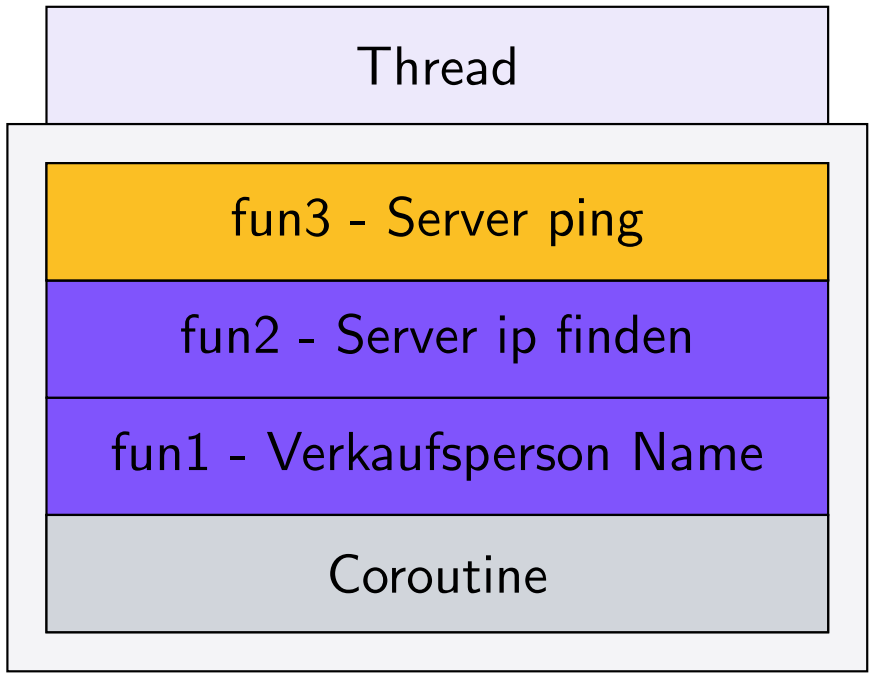




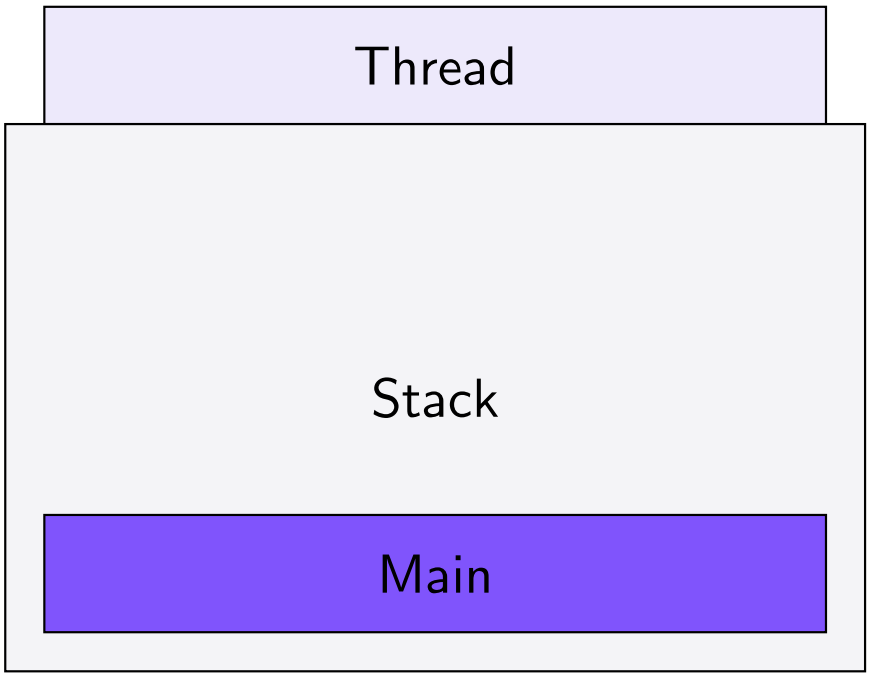
Android: Coroutines

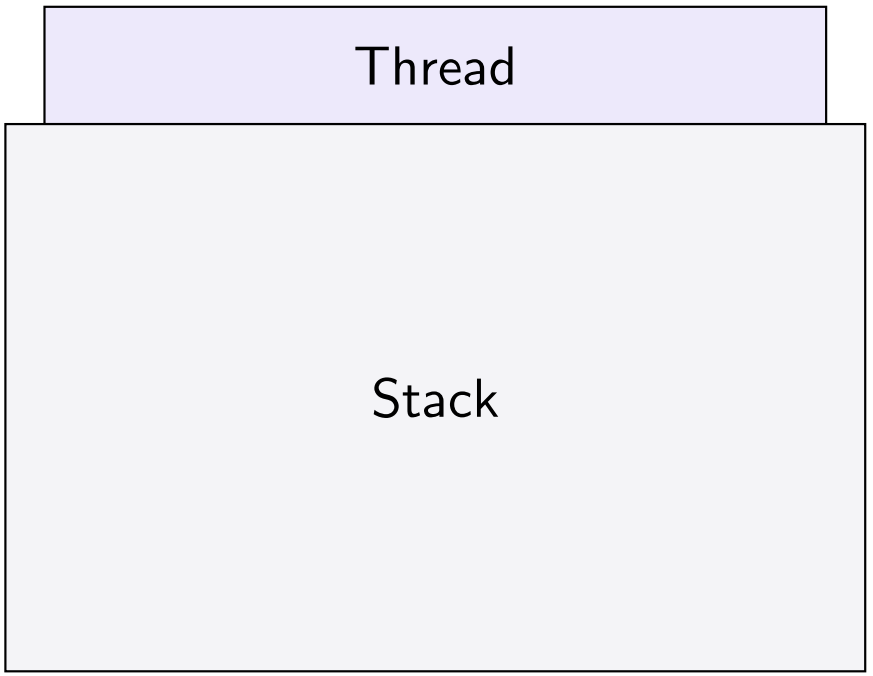
Suspends: delay(),yield(),withContext(),...





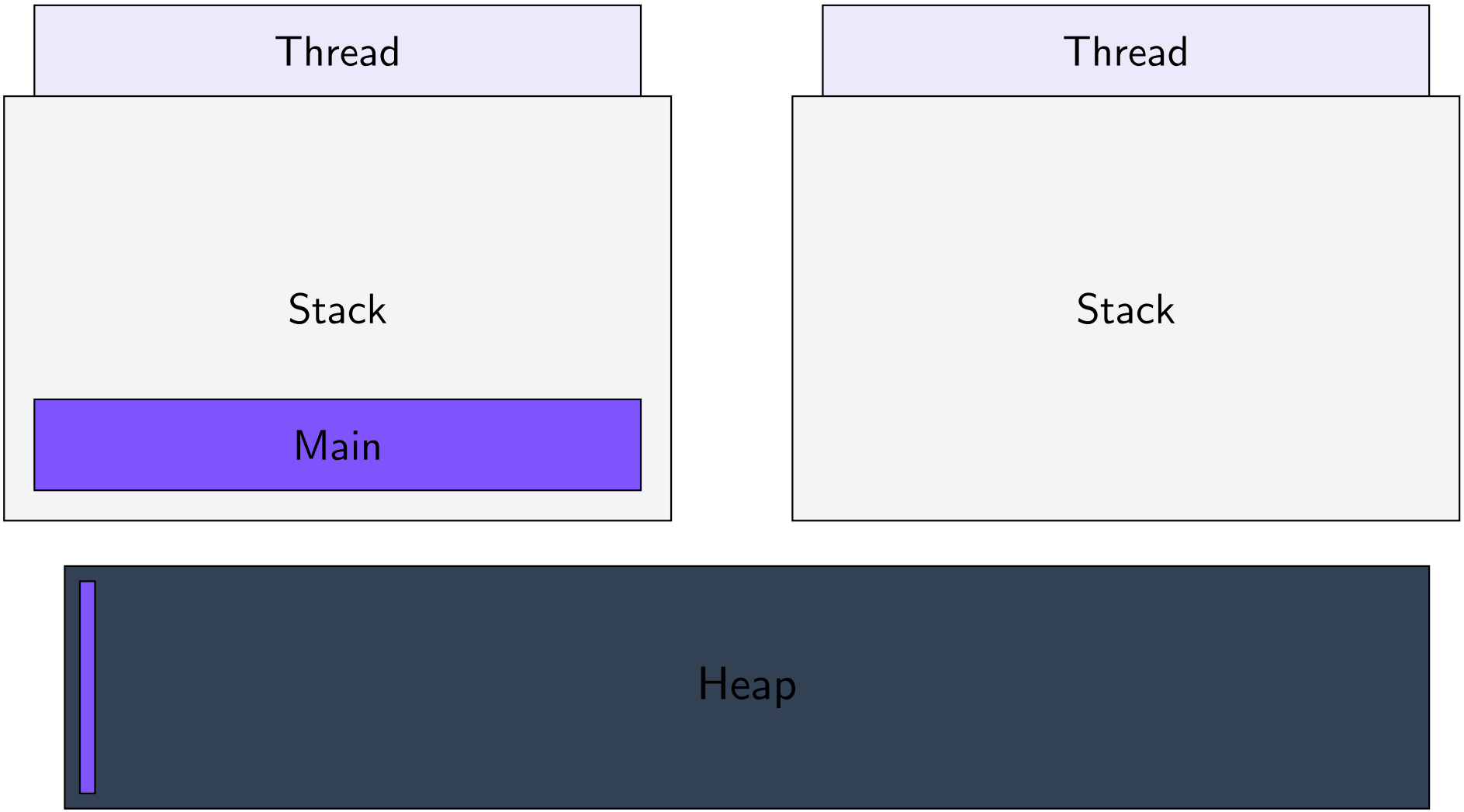
Android: Coroutines



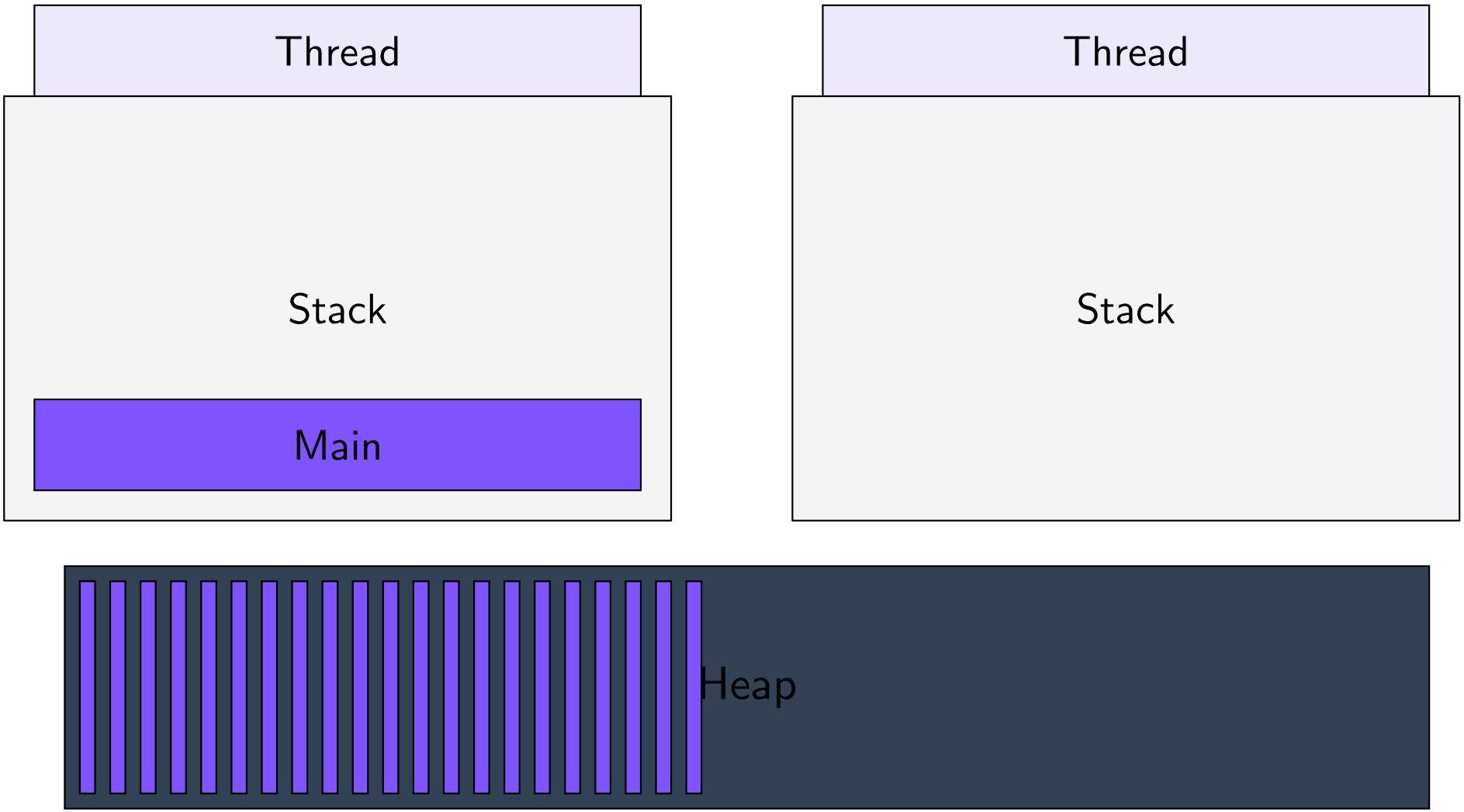



```
Coroutine:
  Variablen:      Name,Ip
    State:            1
     Path:  fun1-fun2-fun3
```

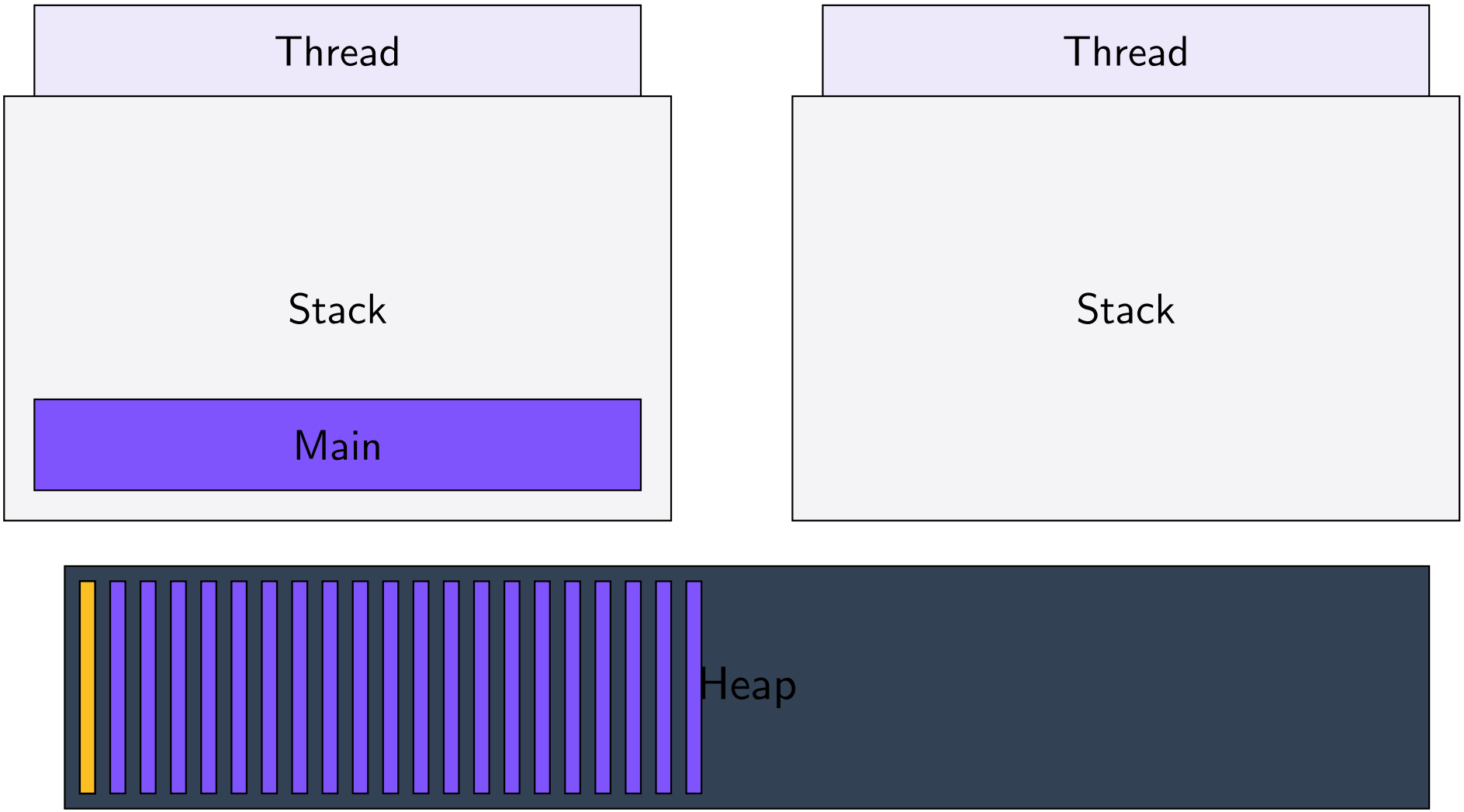
Android: Coroutines



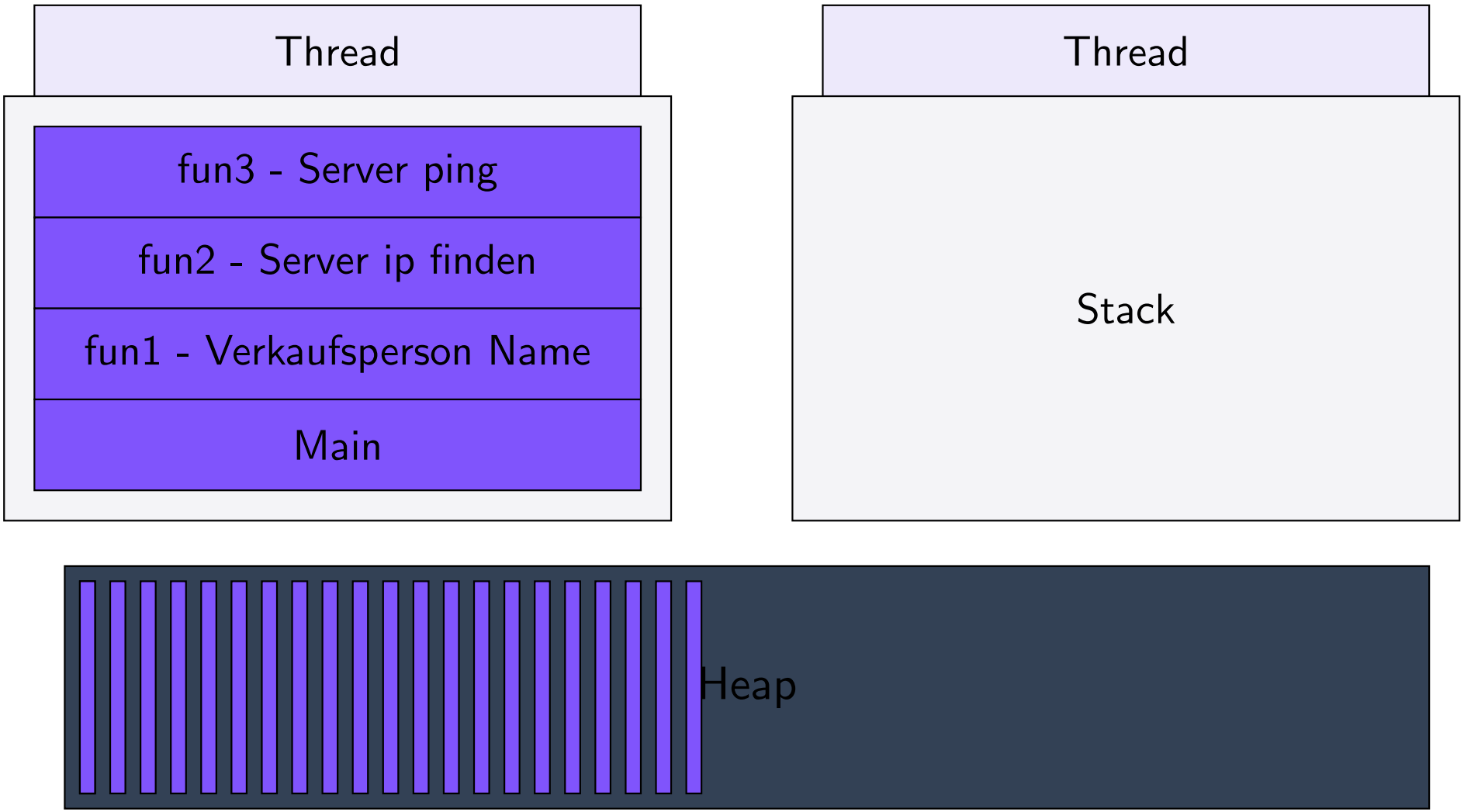
Android: Coroutines



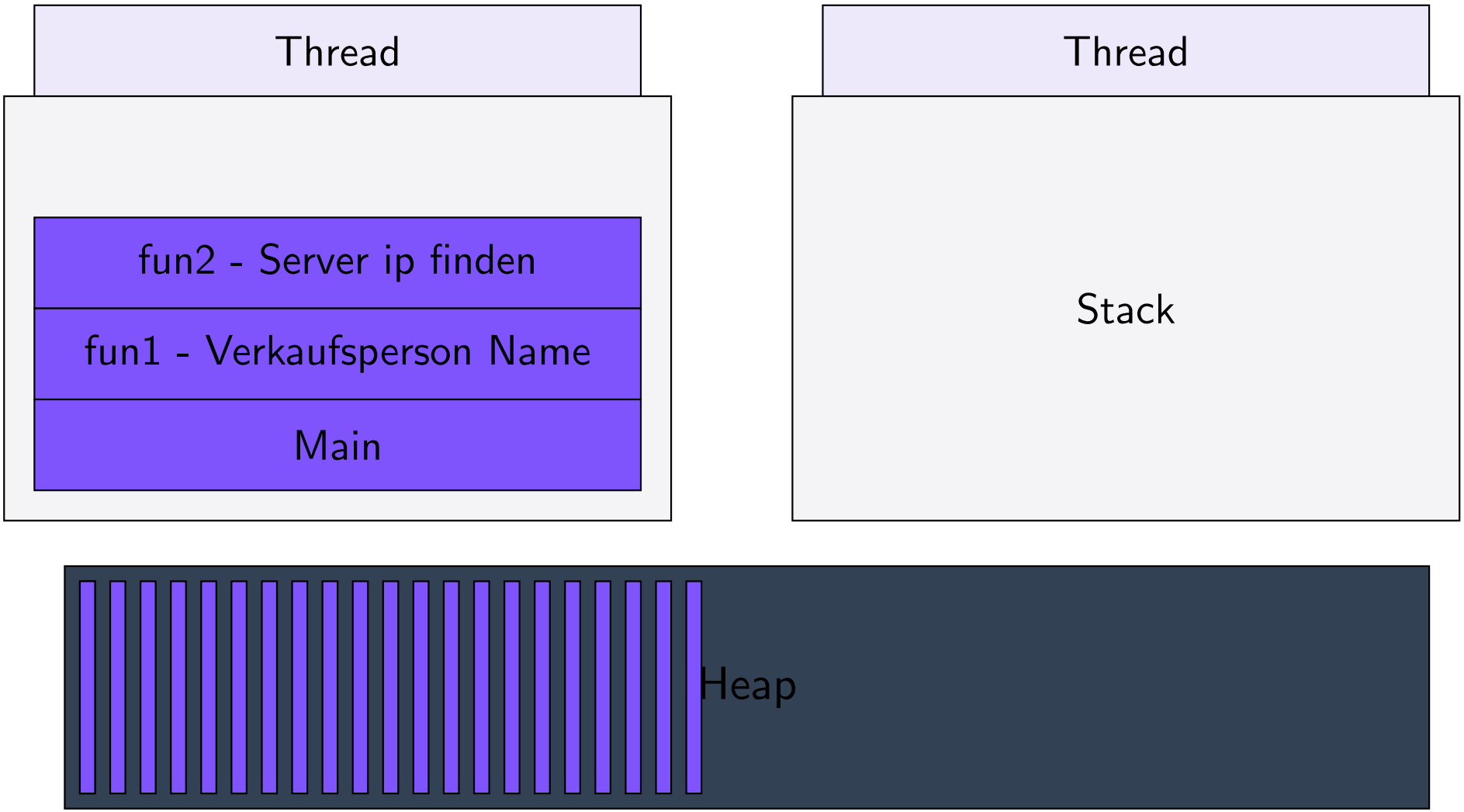
Android: Coroutines



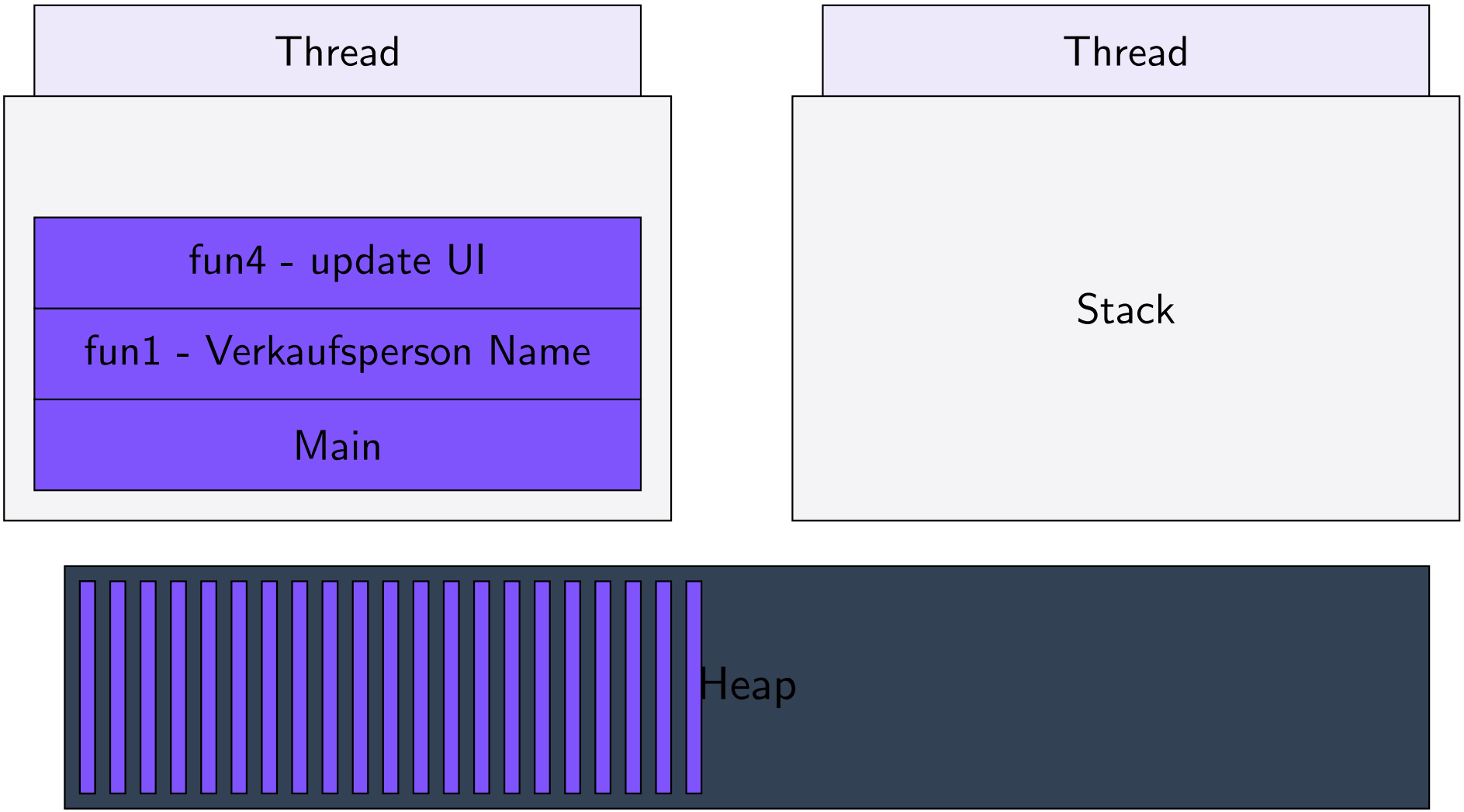
Android: Coroutines



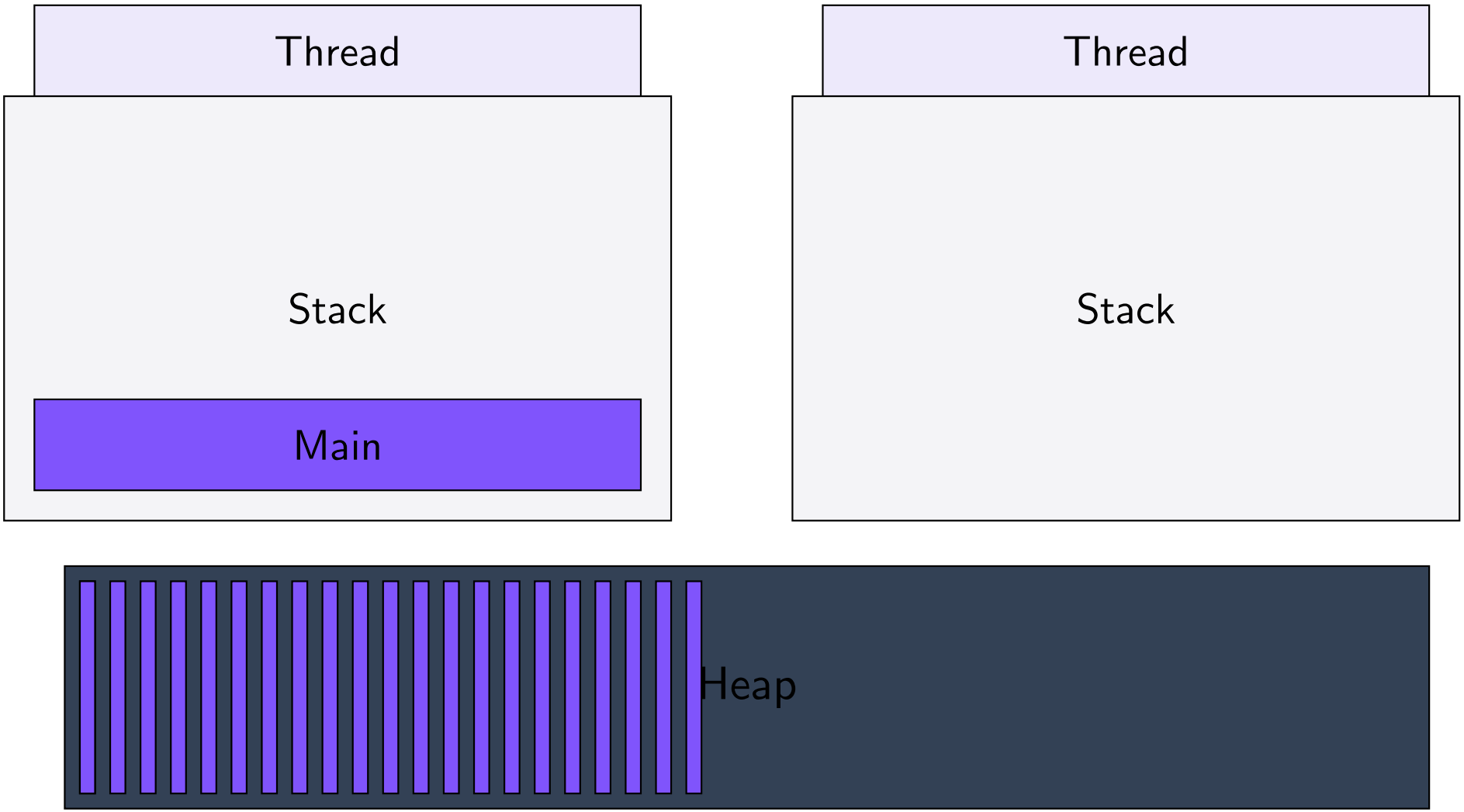
Android: Coroutines



Android: Coroutines



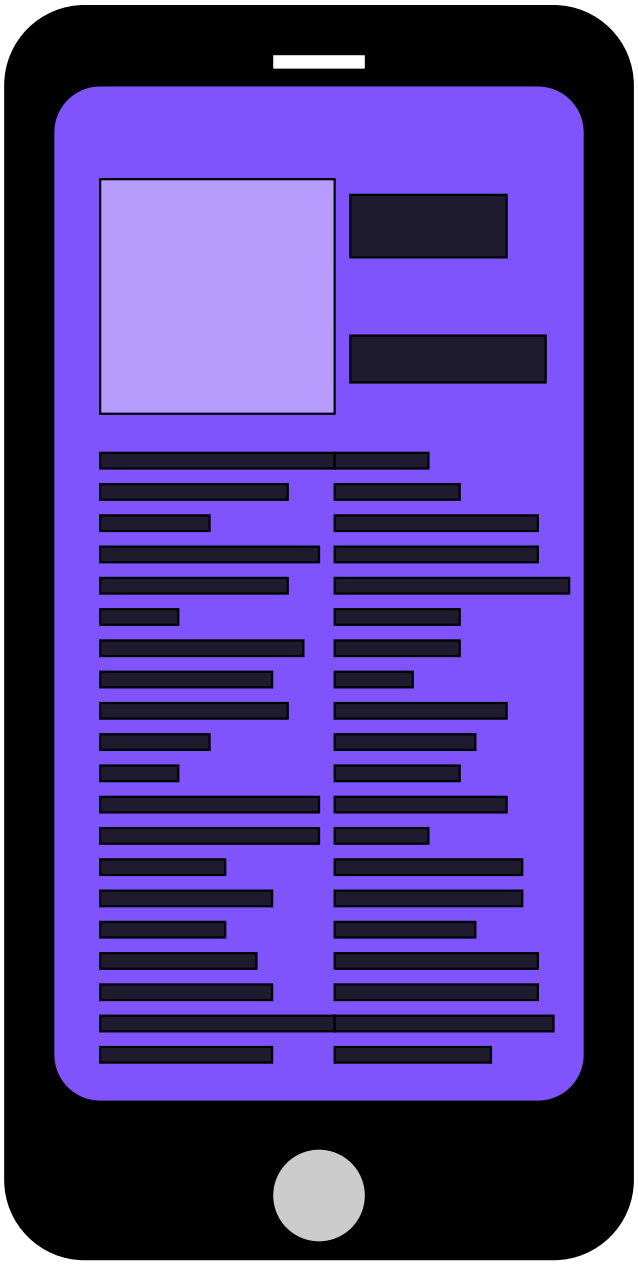
Android: Coroutines



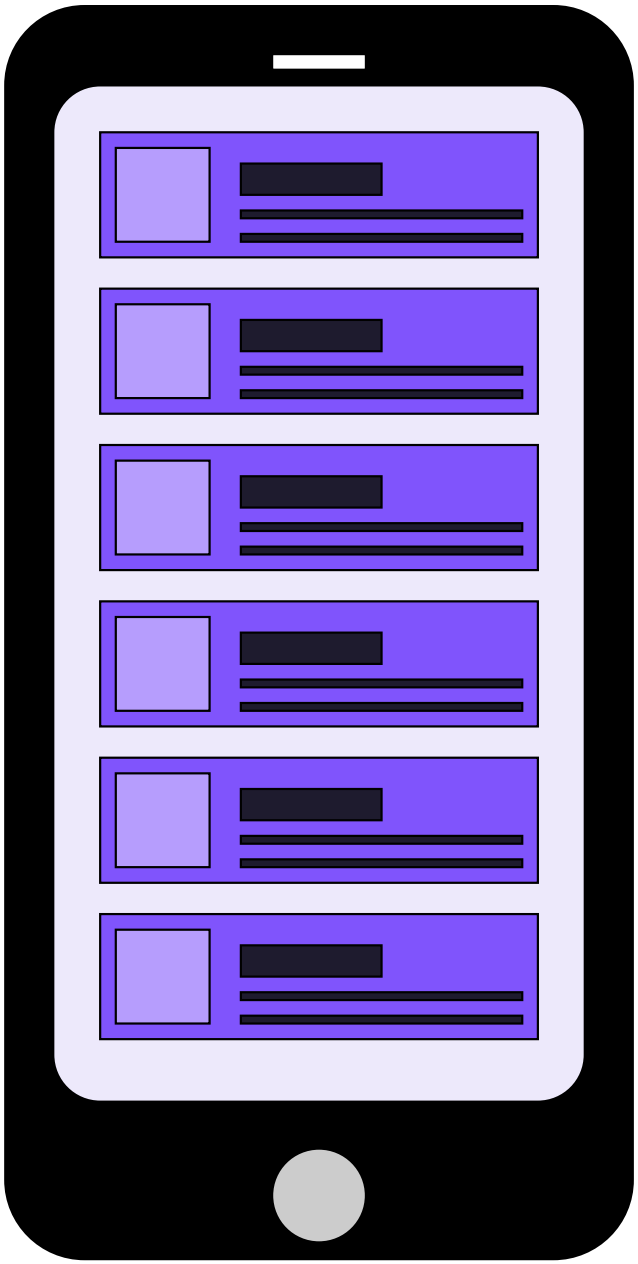
Android: Coroutines: Beispiel

Multithreading mit Coroutines
ein Thread
Multithreading
Multithreading mit Coroutines

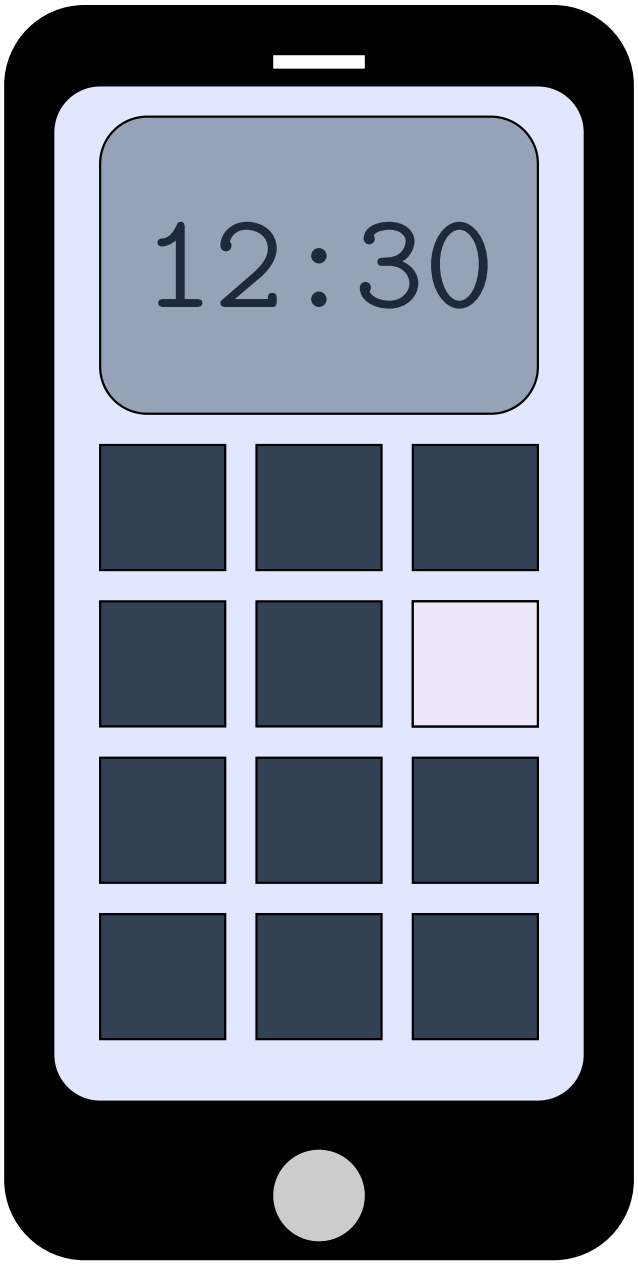
Android: Coroutines: Beispiel



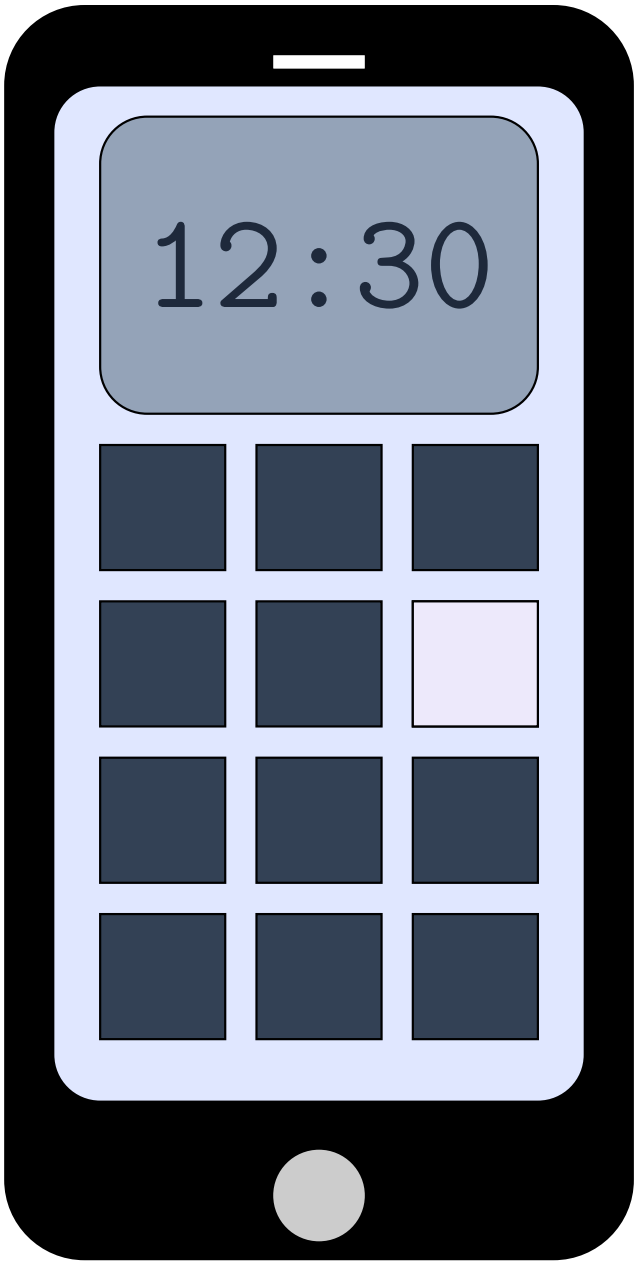
Android: Coroutines: Beispiel

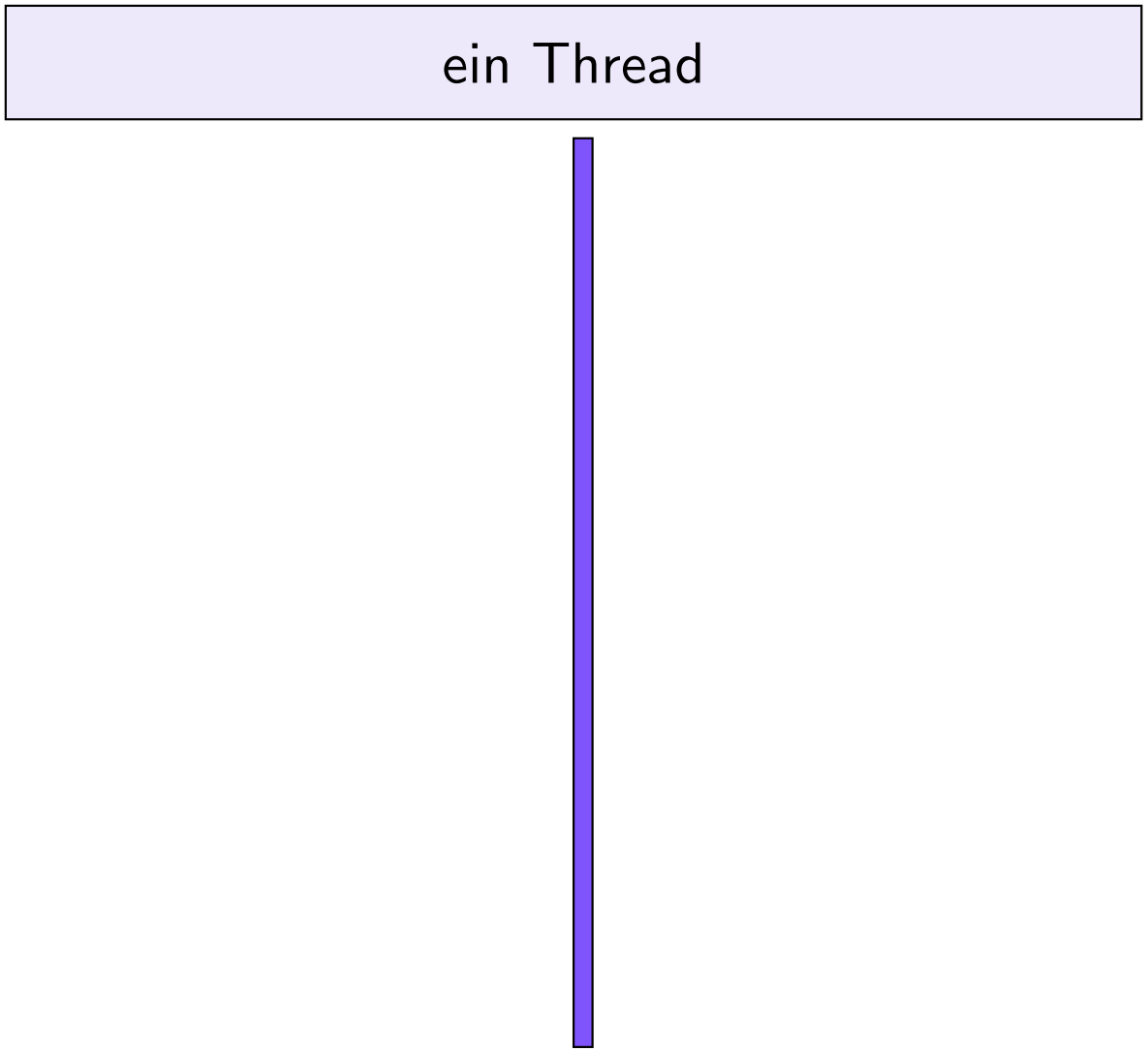


Android: Coroutines: Beispiel

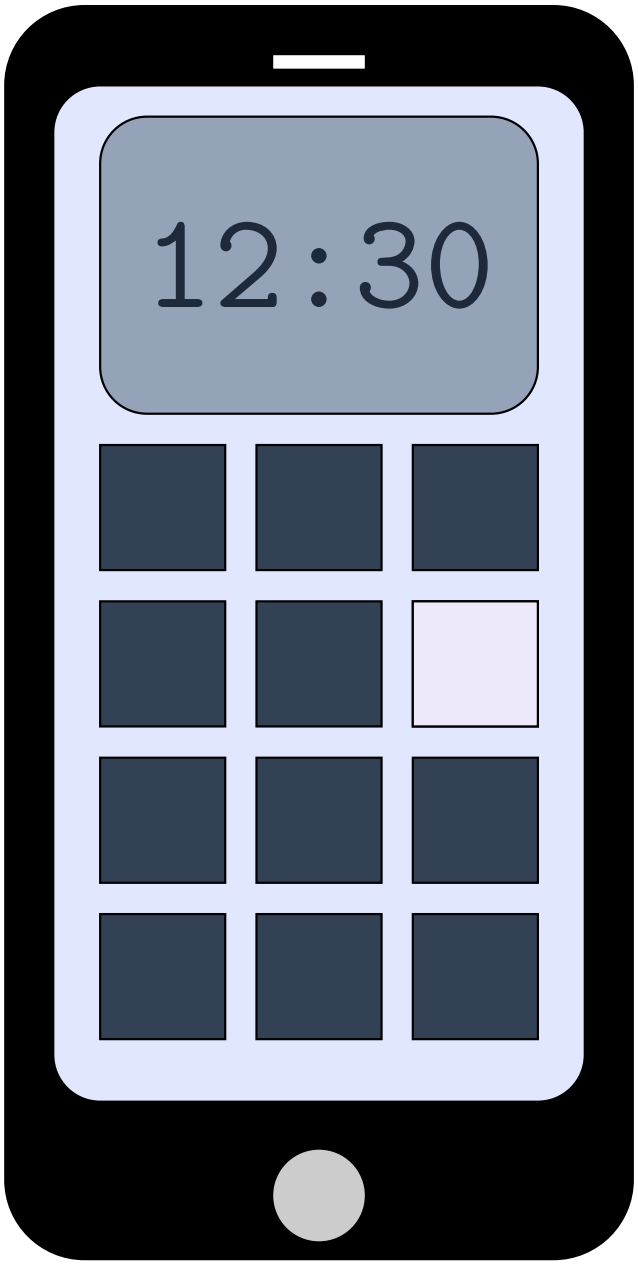


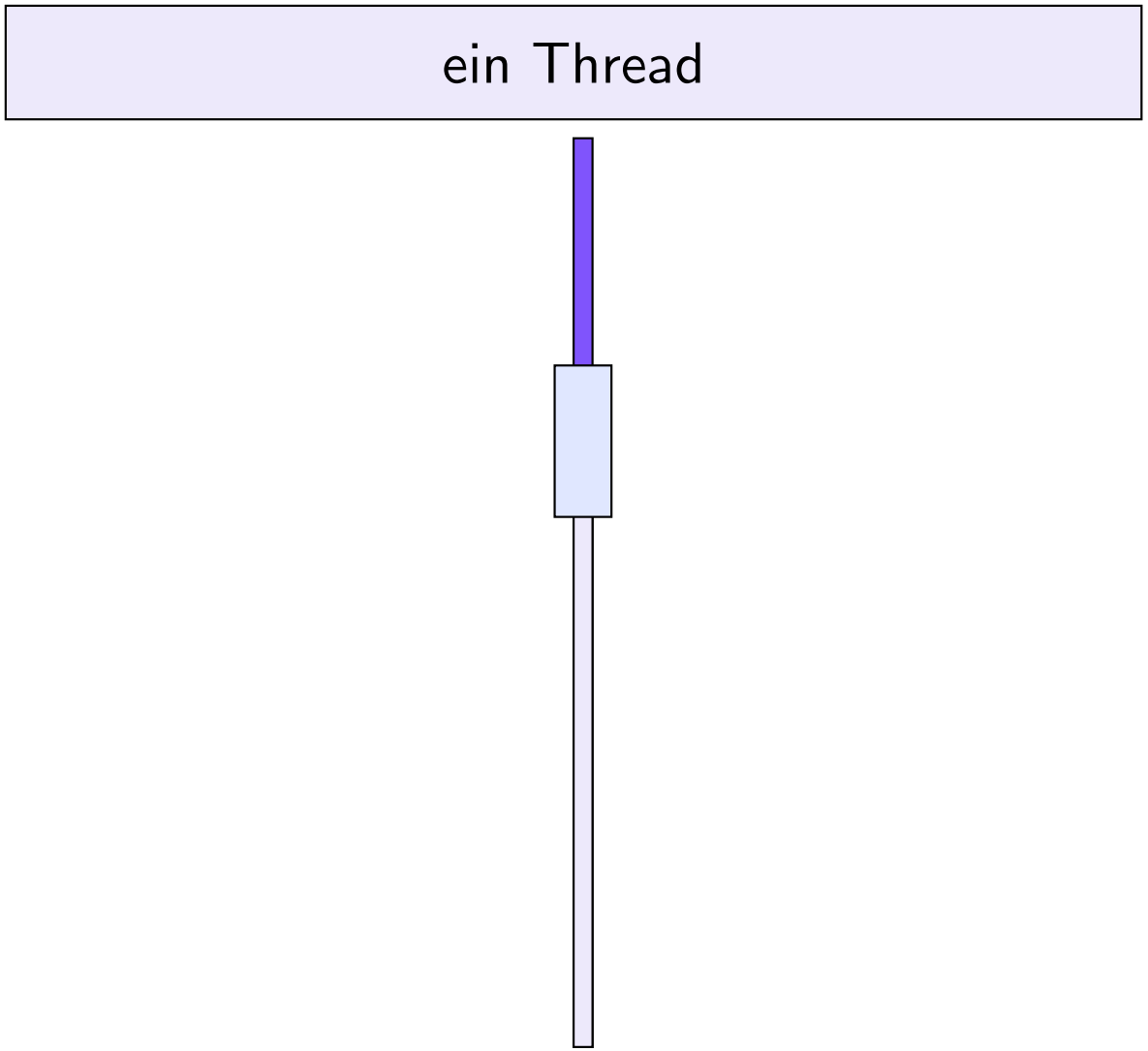
Android: Coroutines: Beispiel



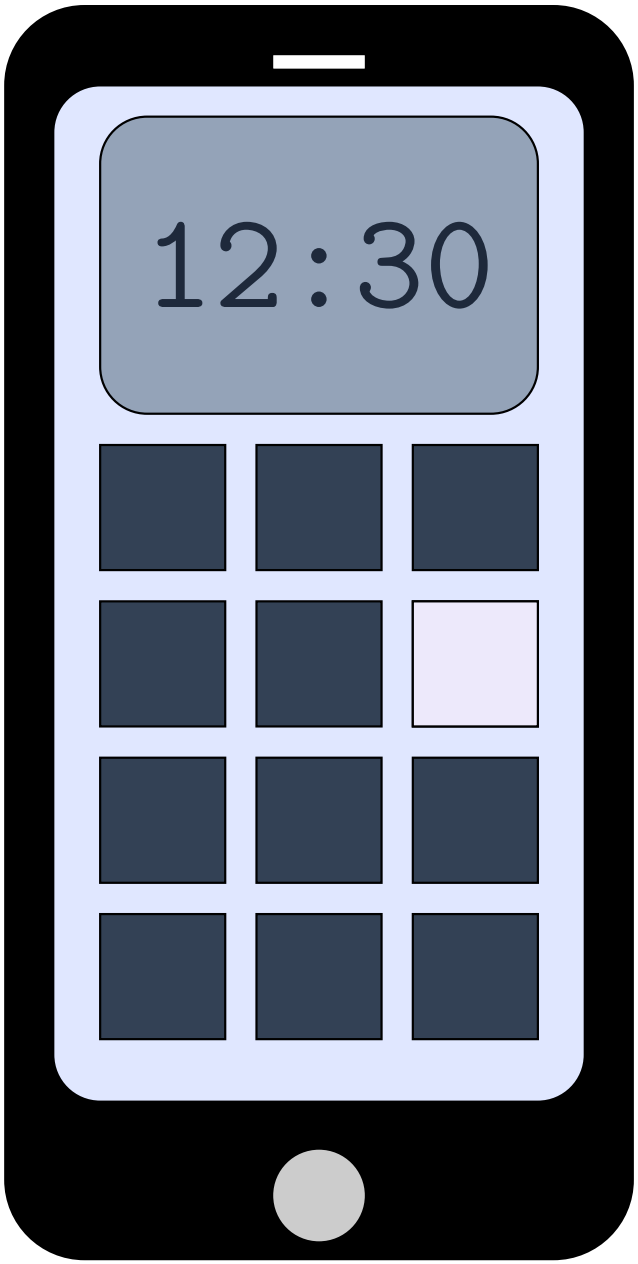


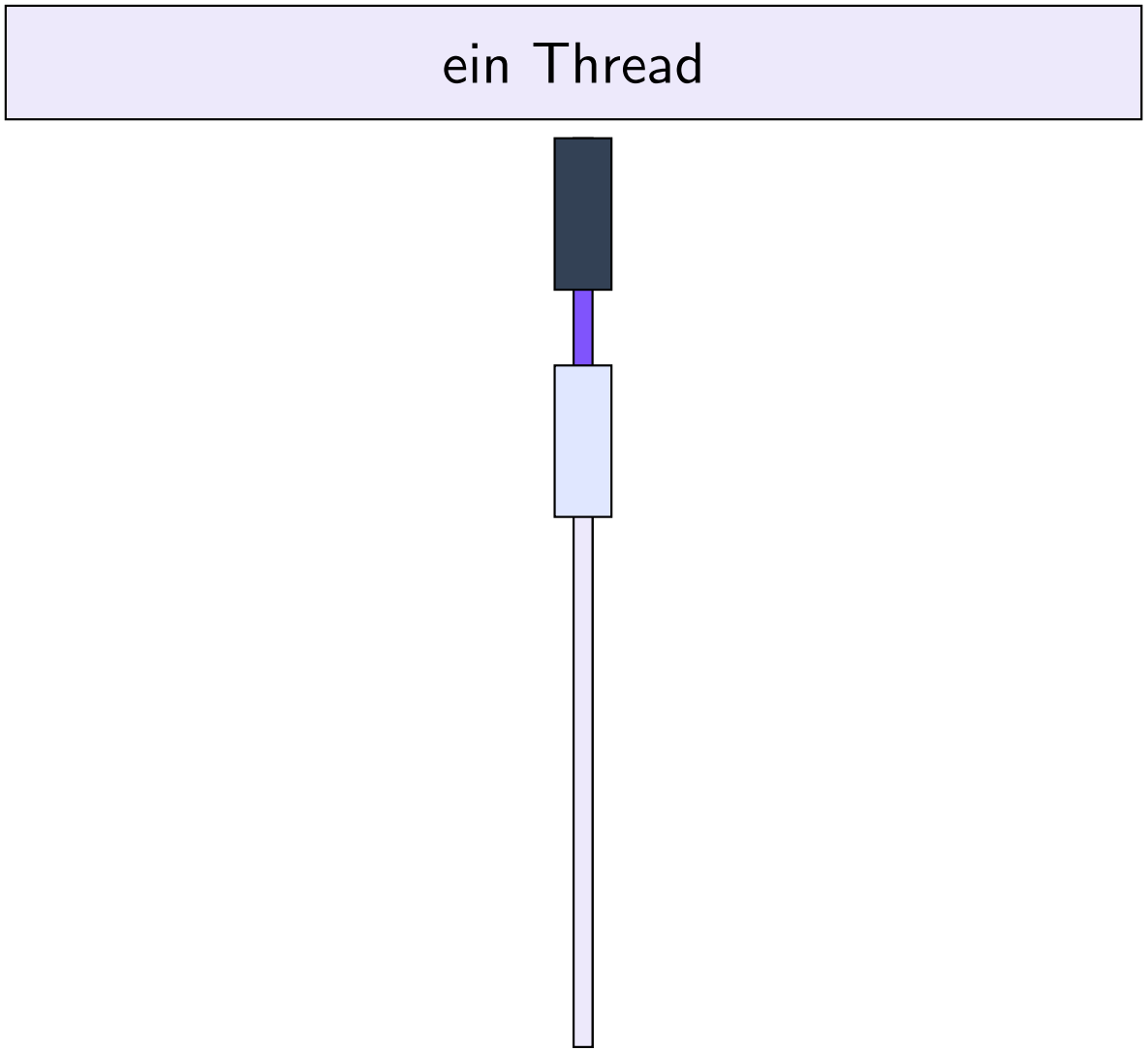
Android: Coroutines: Beispiel



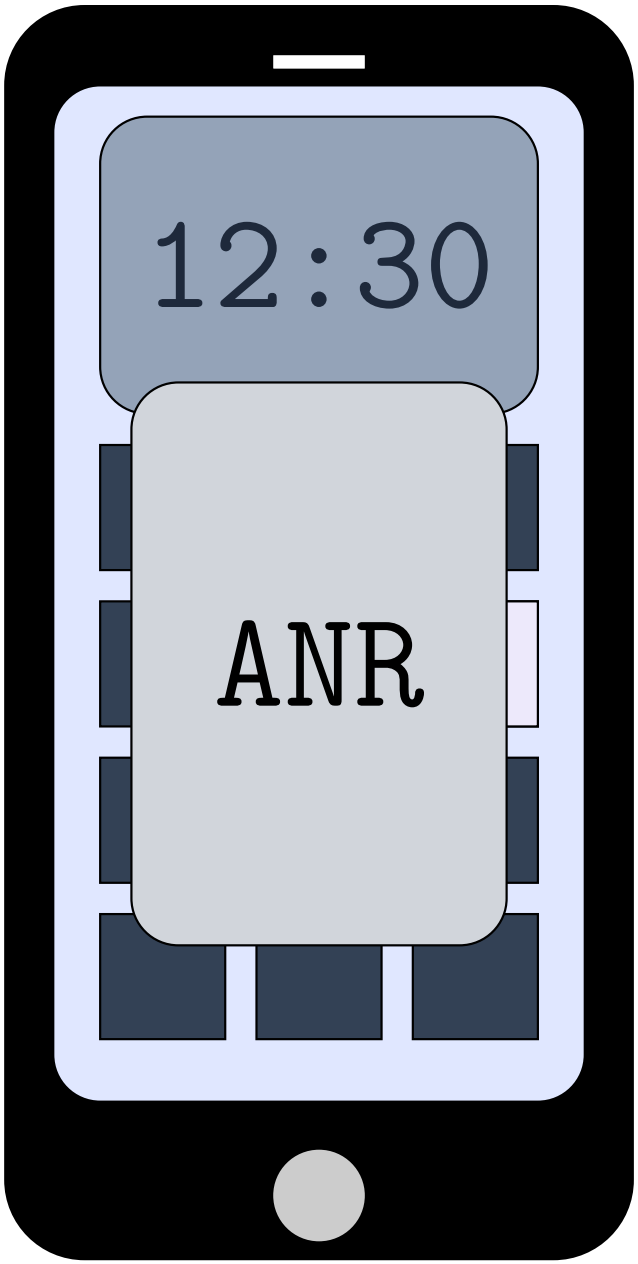


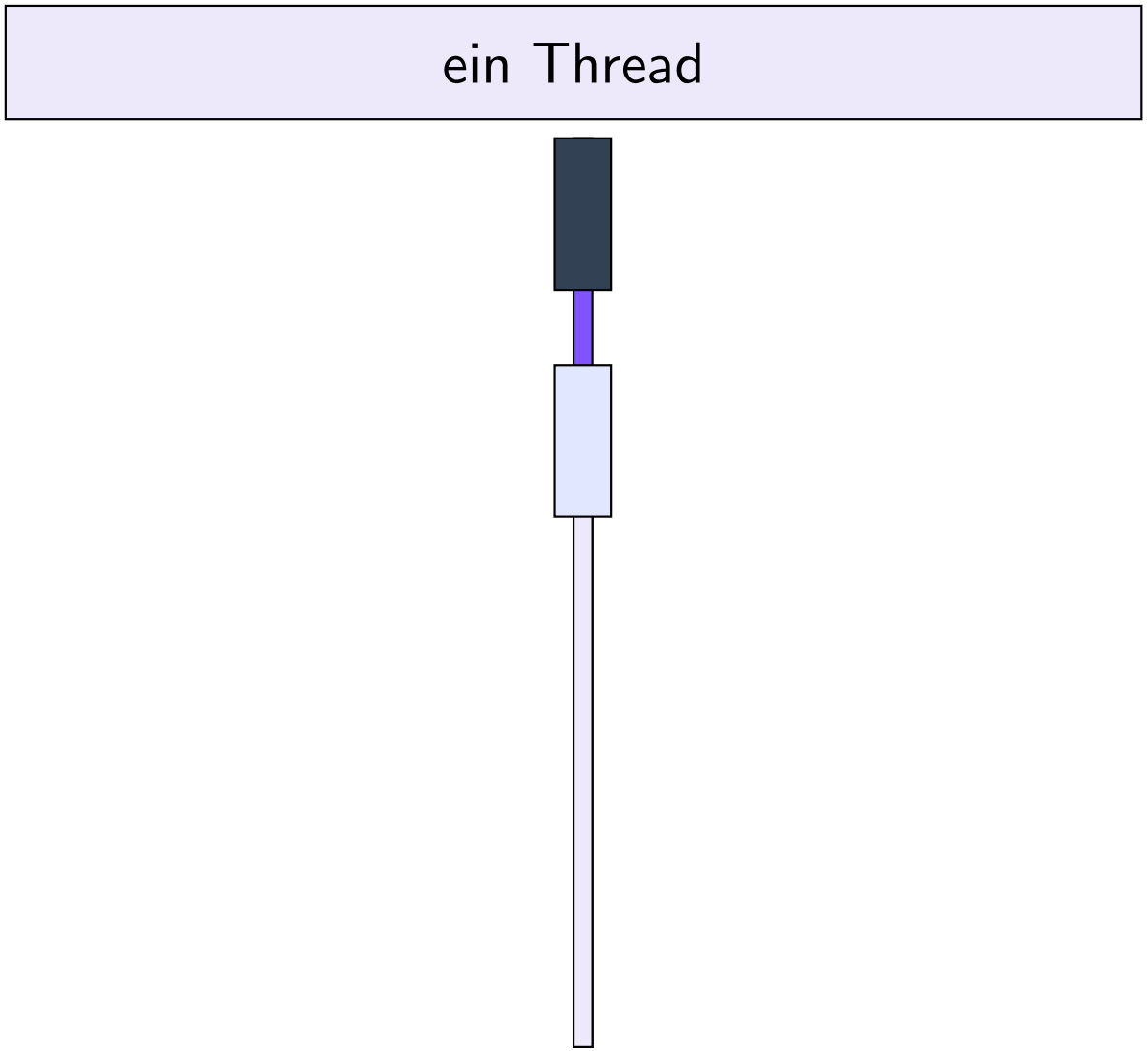
Android: Coroutines: Beispiel



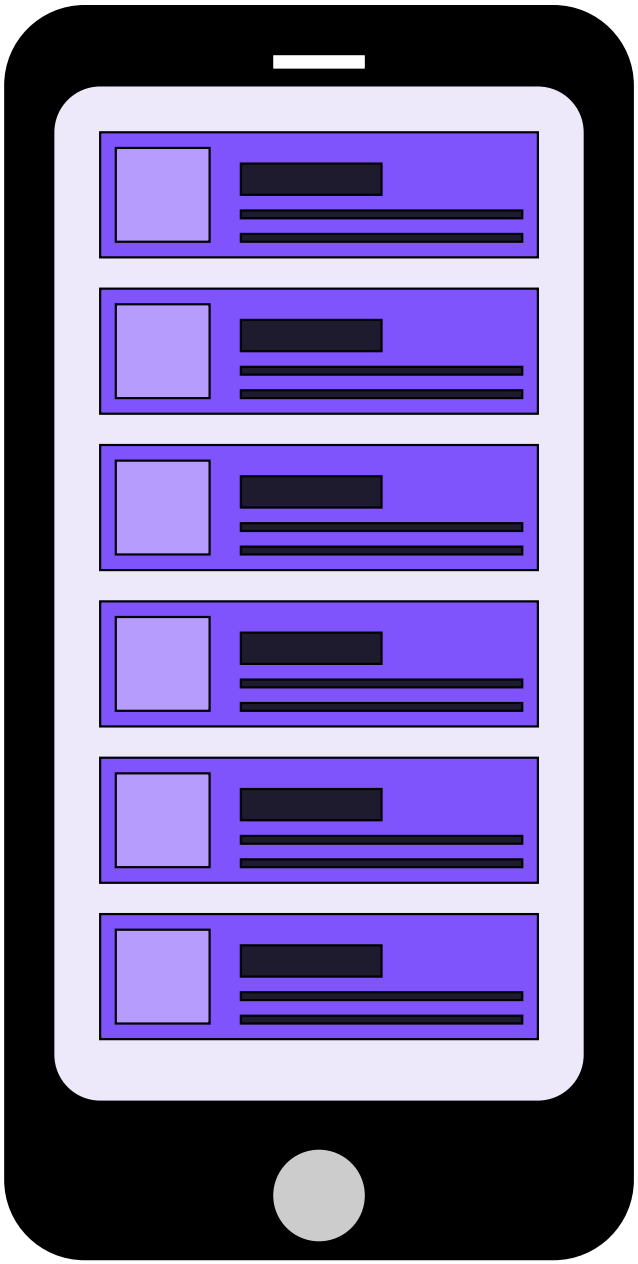


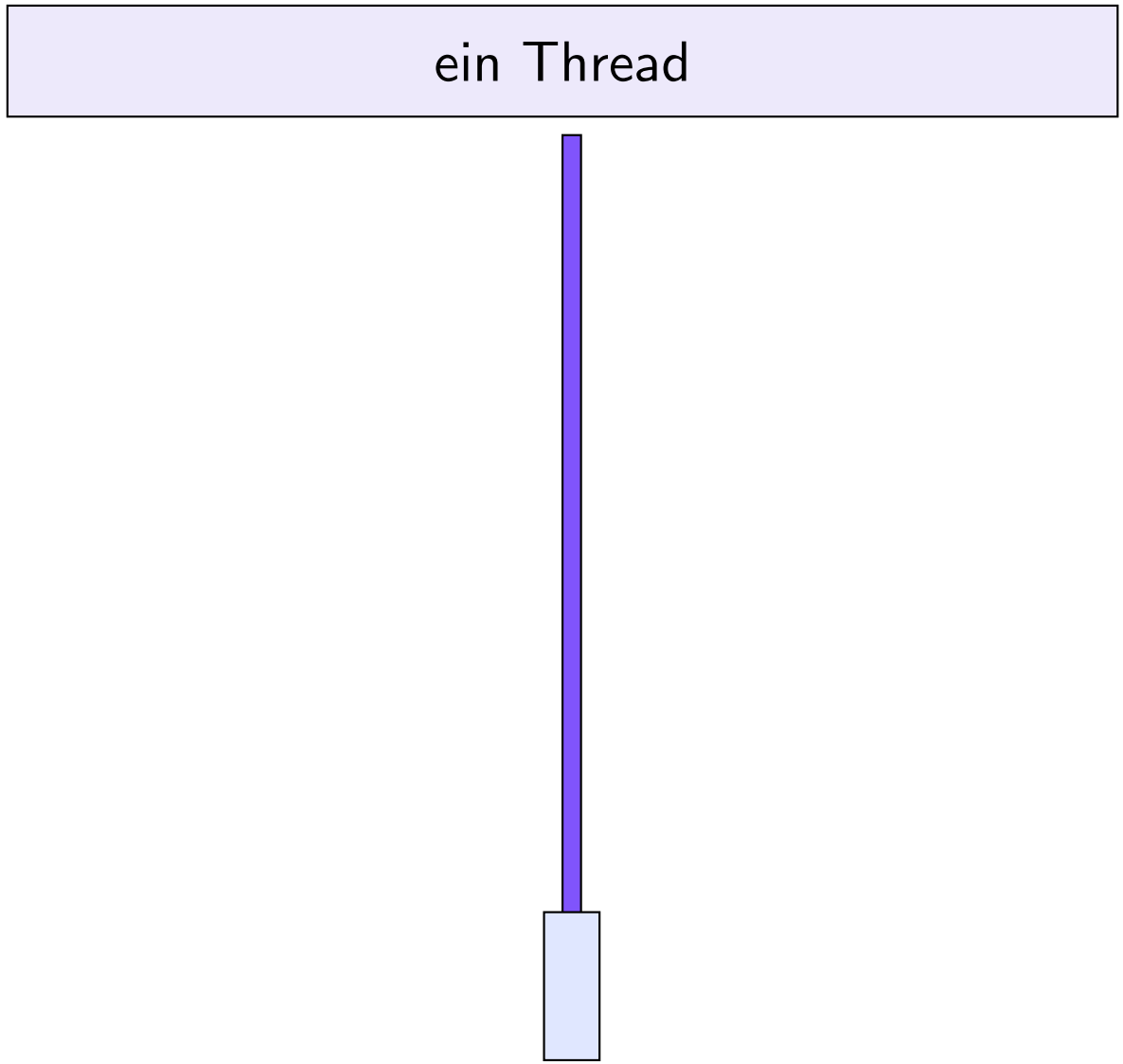
Android: Coroutines: Beispiel



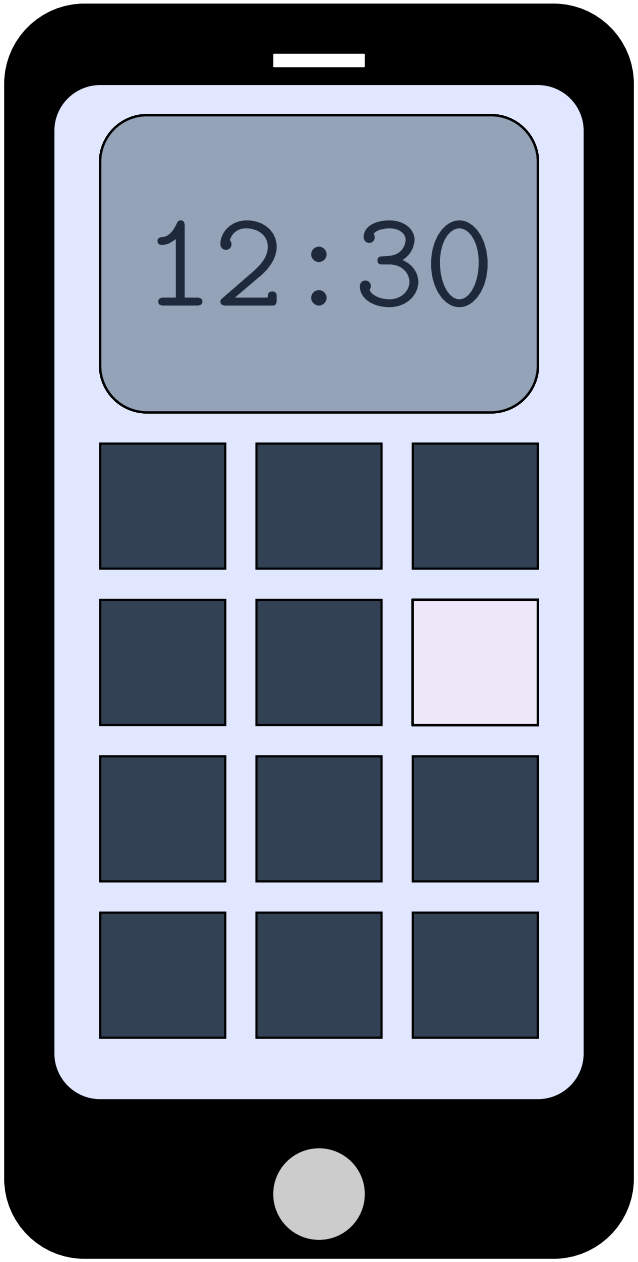


Android: Coroutines: Beispiel



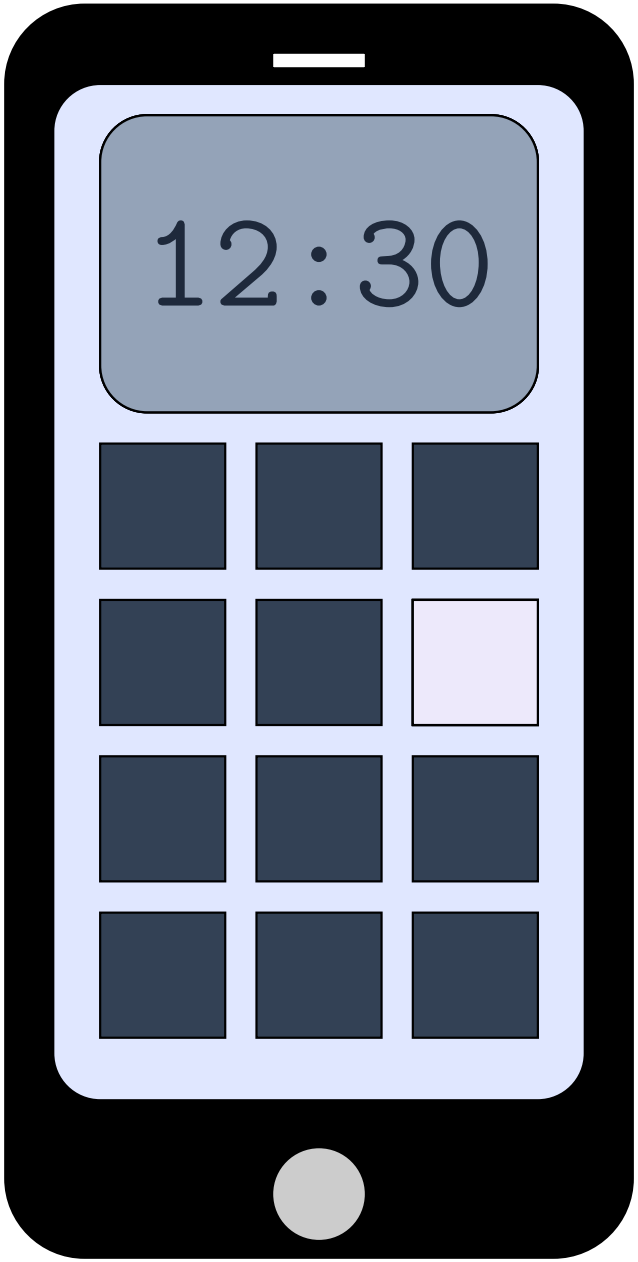


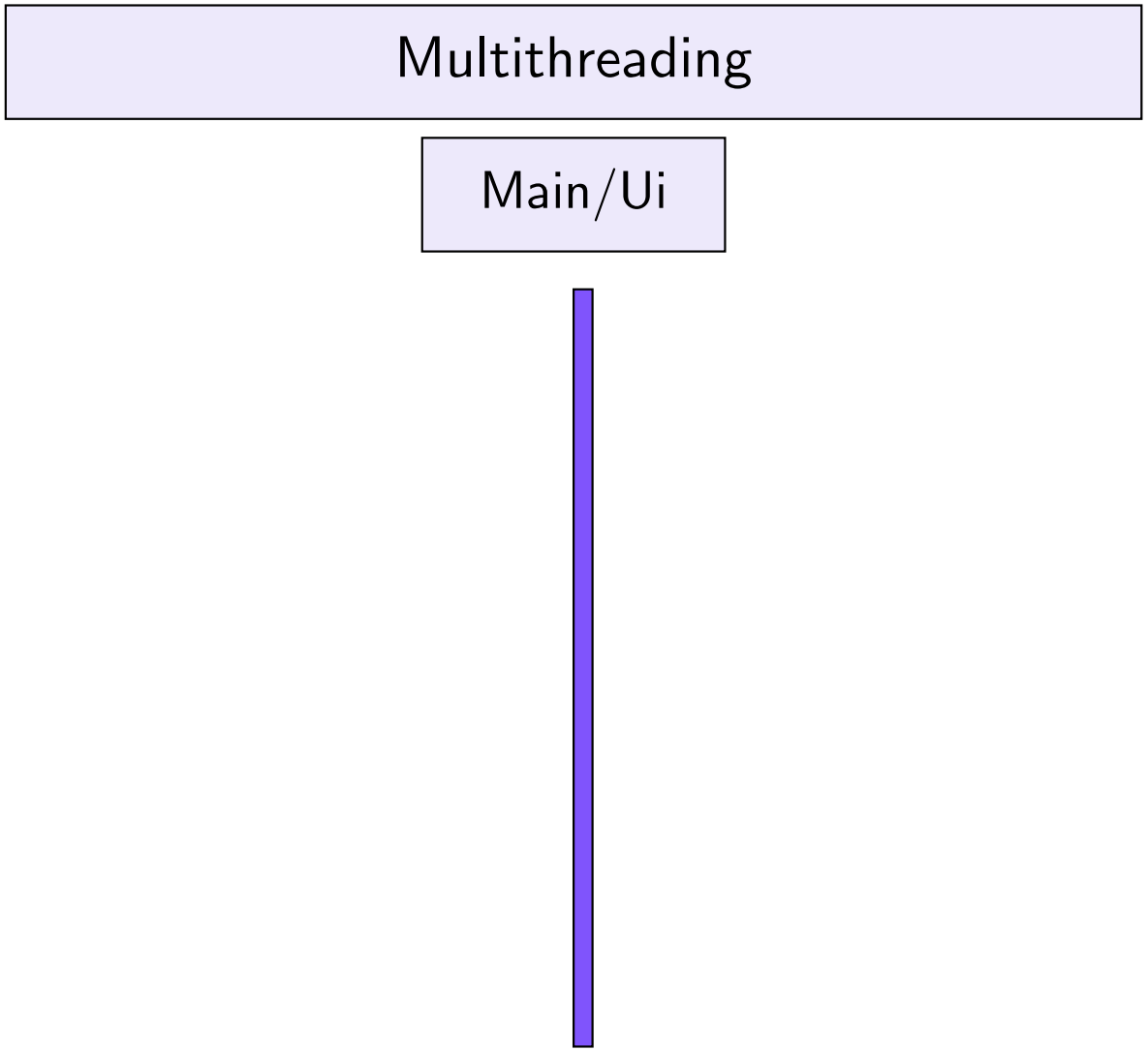
Android: Coroutines: Beispiel



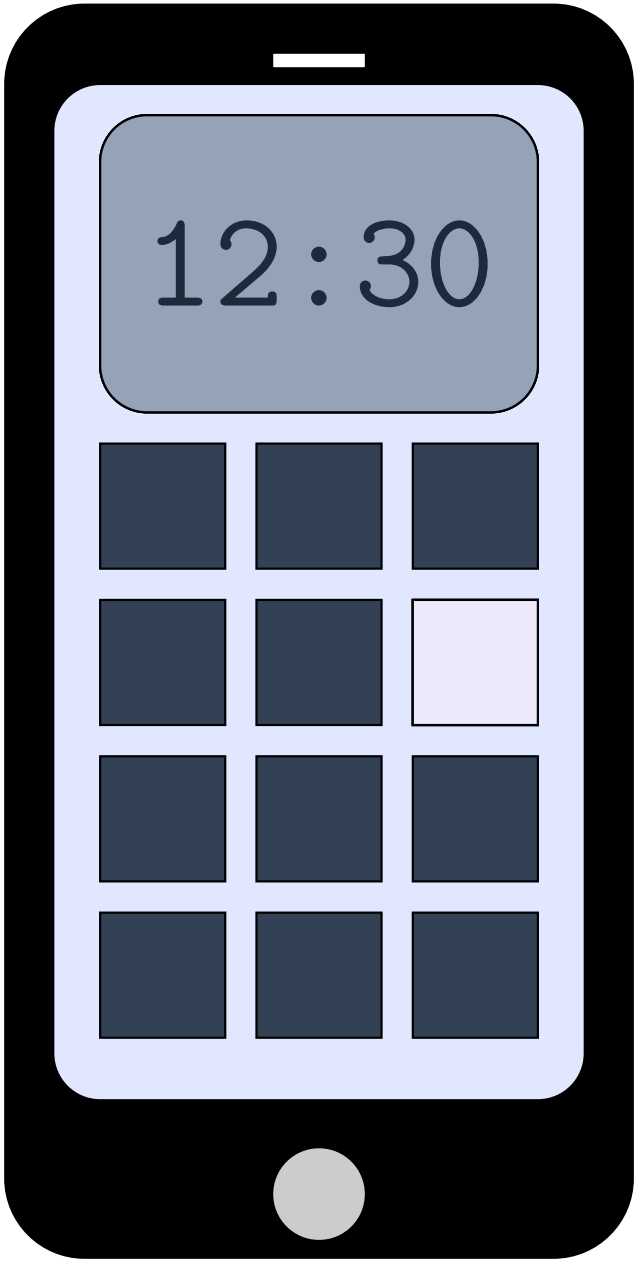
Multithreading

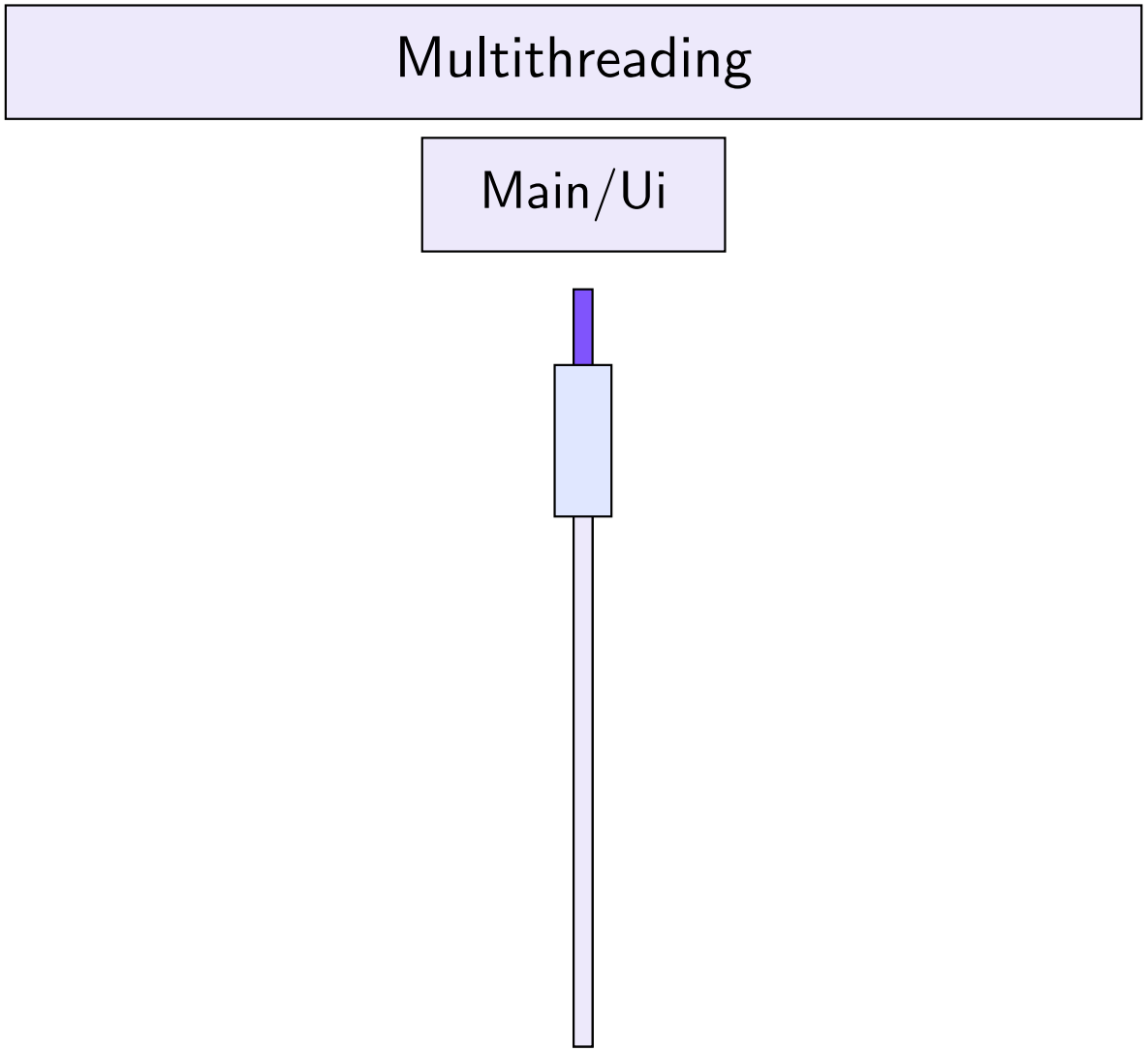
Android: Coroutines: Beispiel



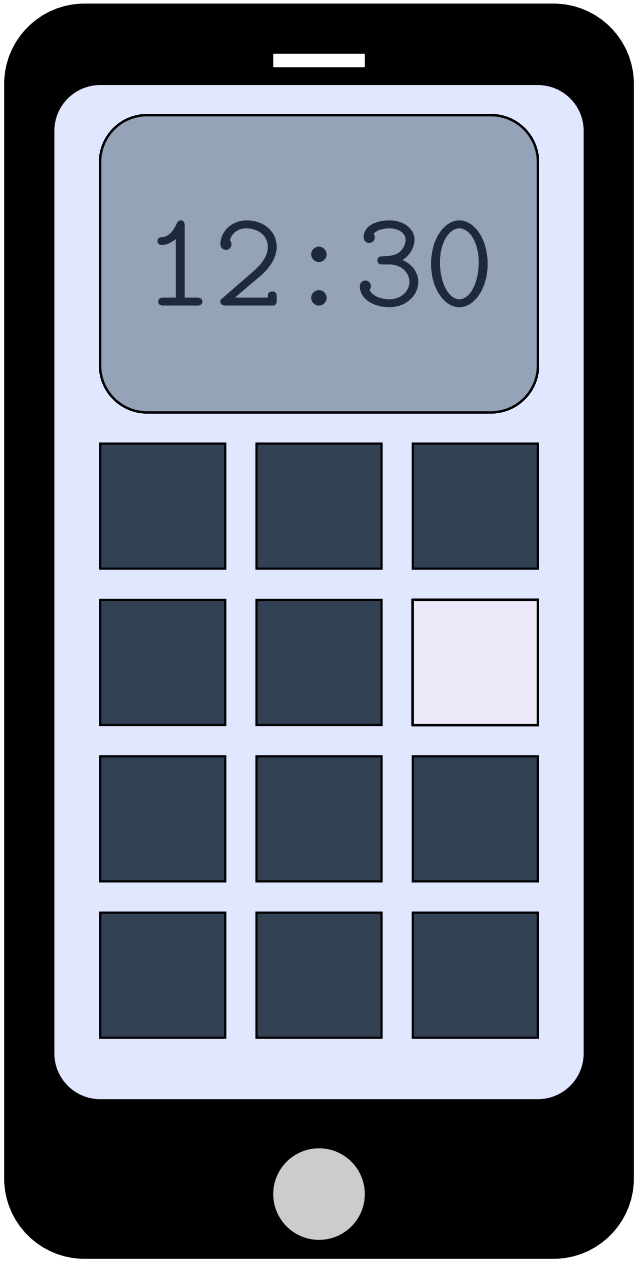


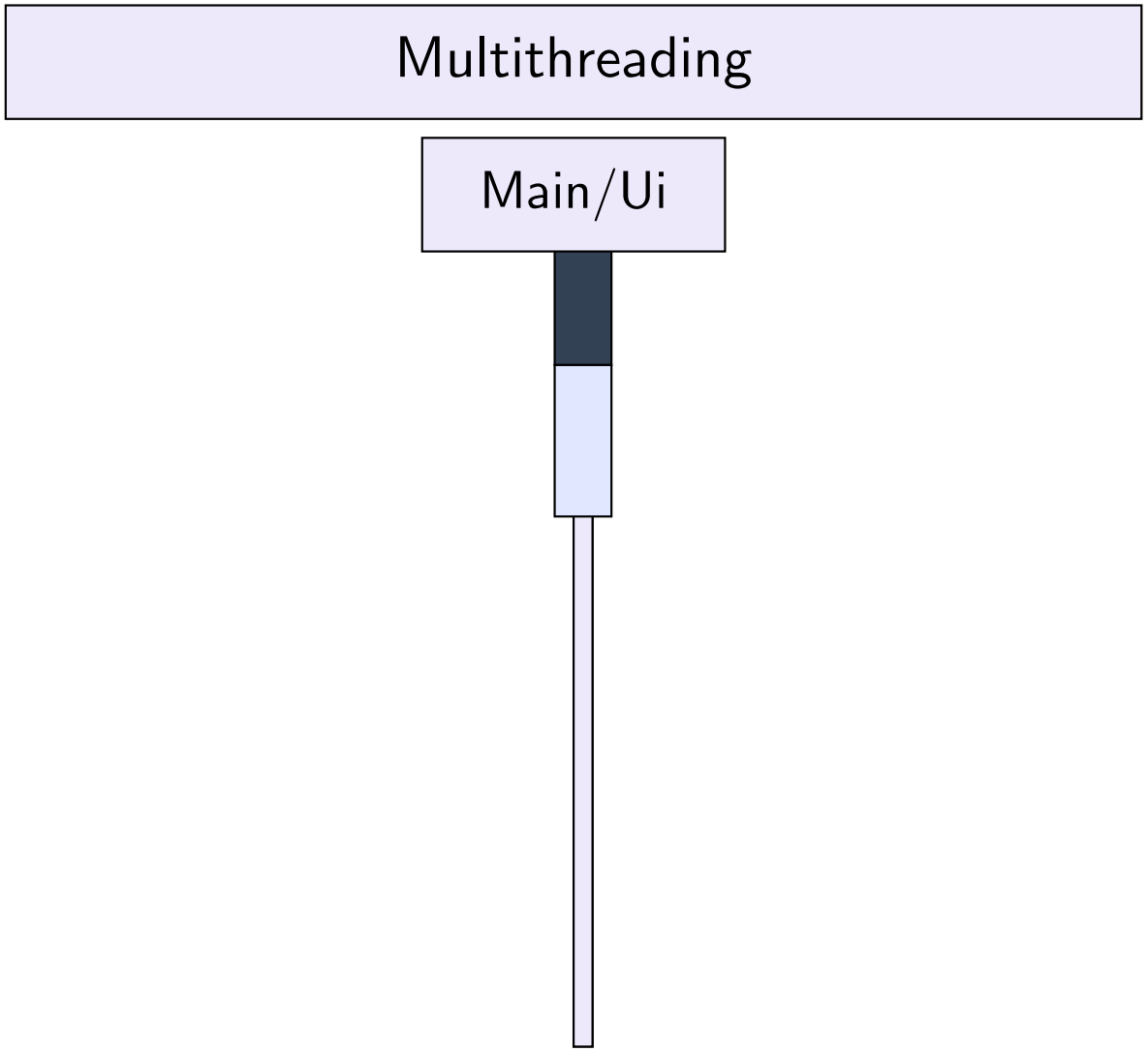
Android: Coroutines: Beispiel



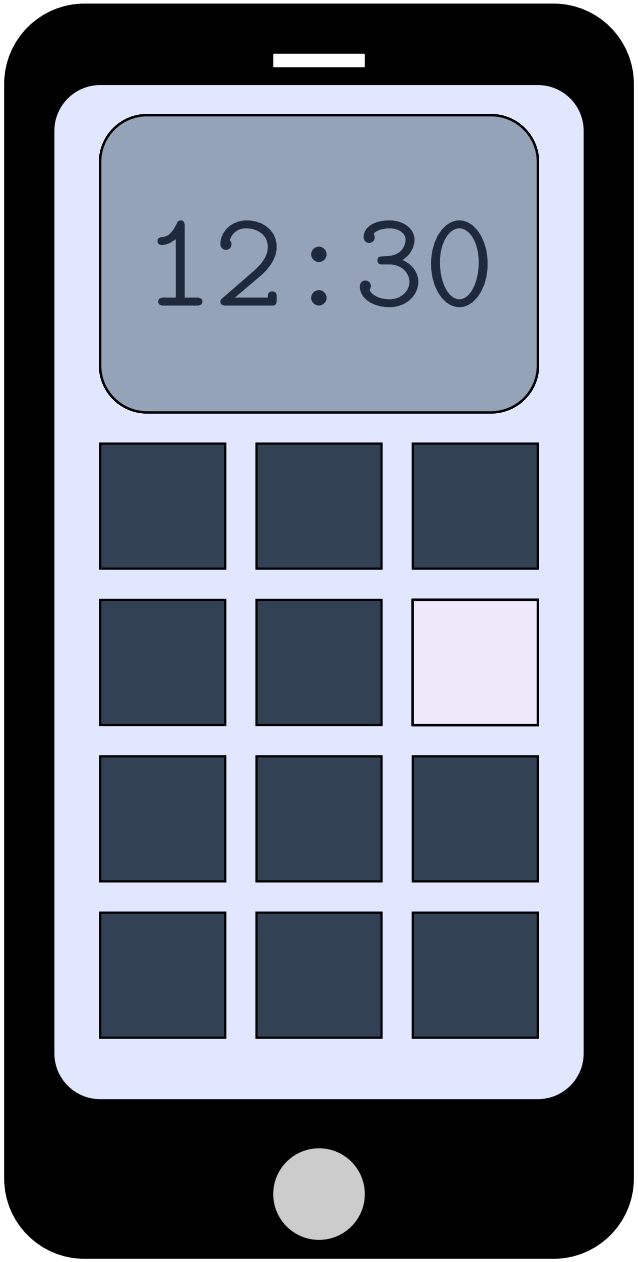


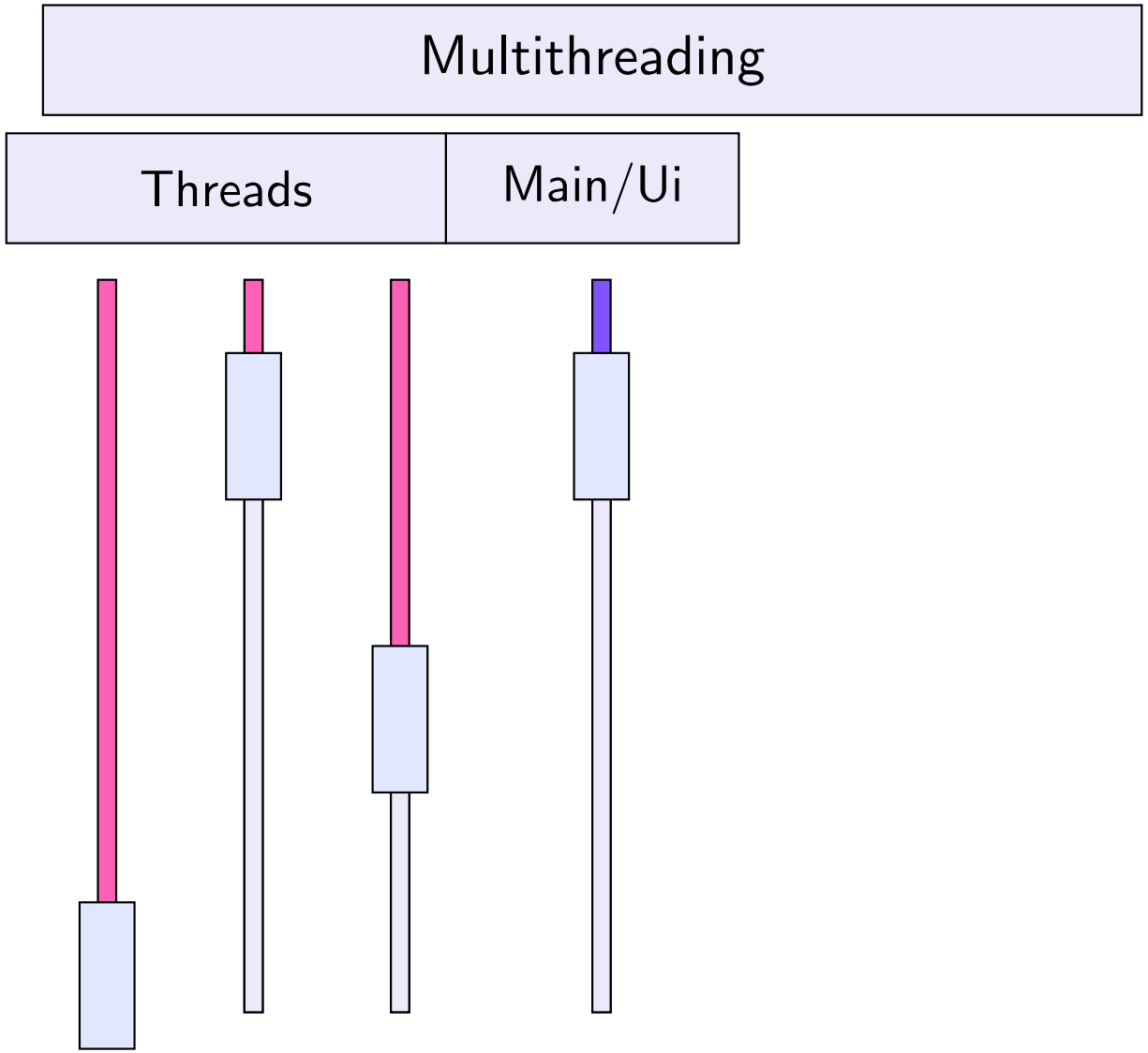
Android: Coroutines: Beispiel



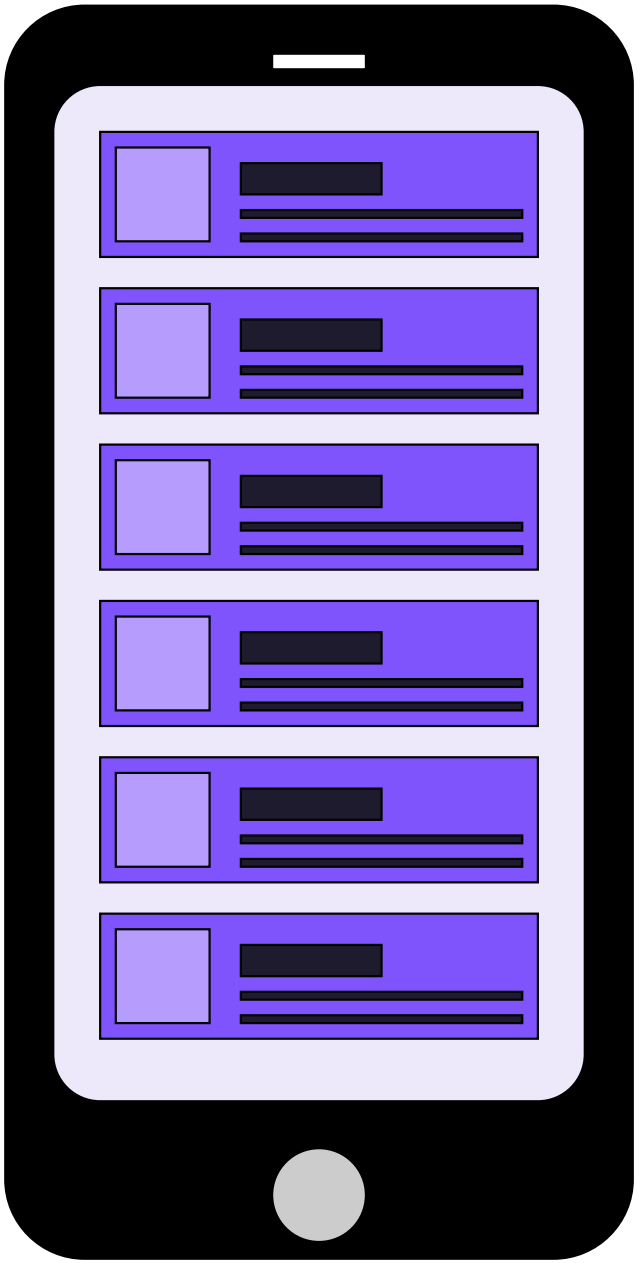


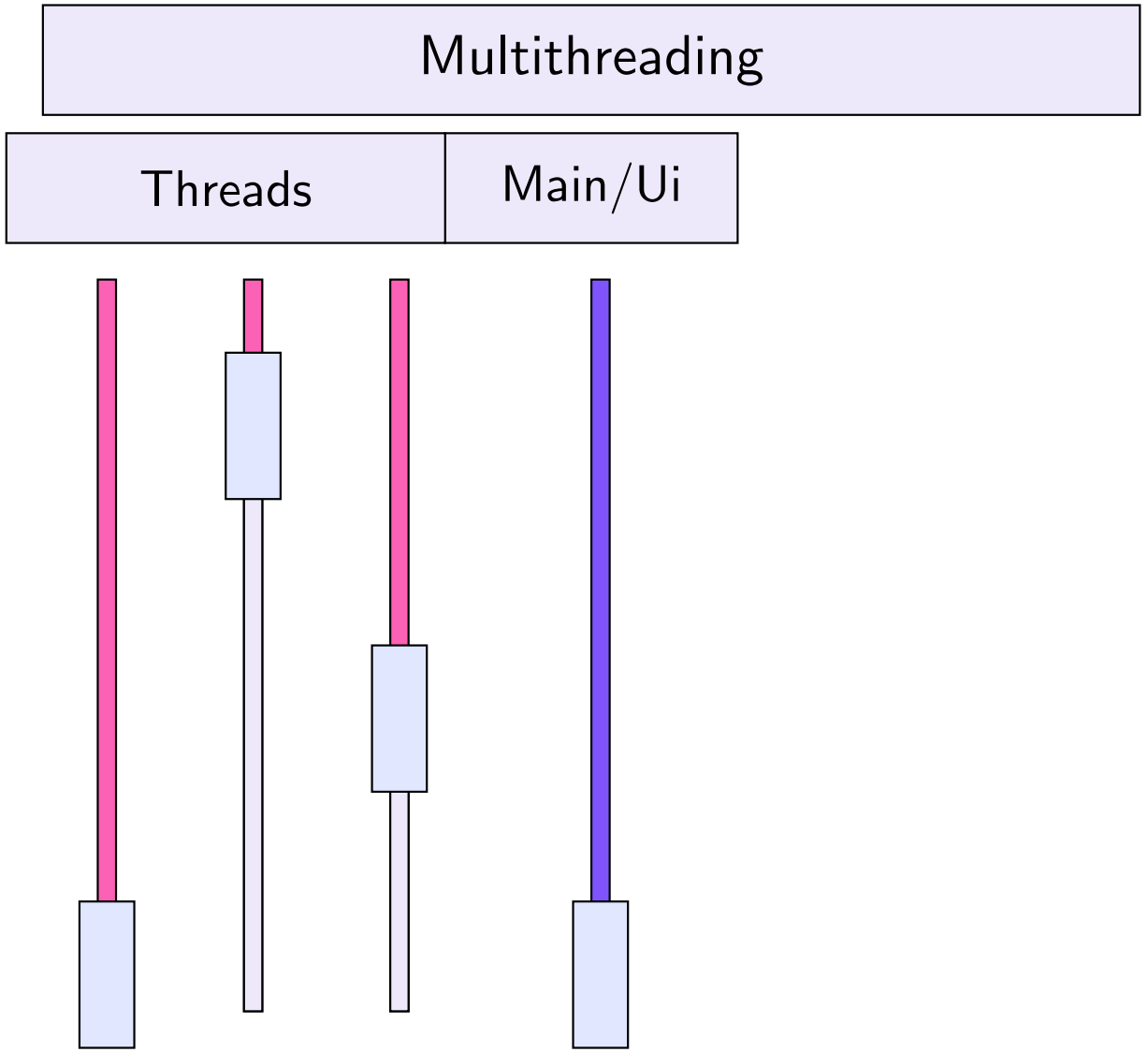
Android: Coroutines: Beispiel



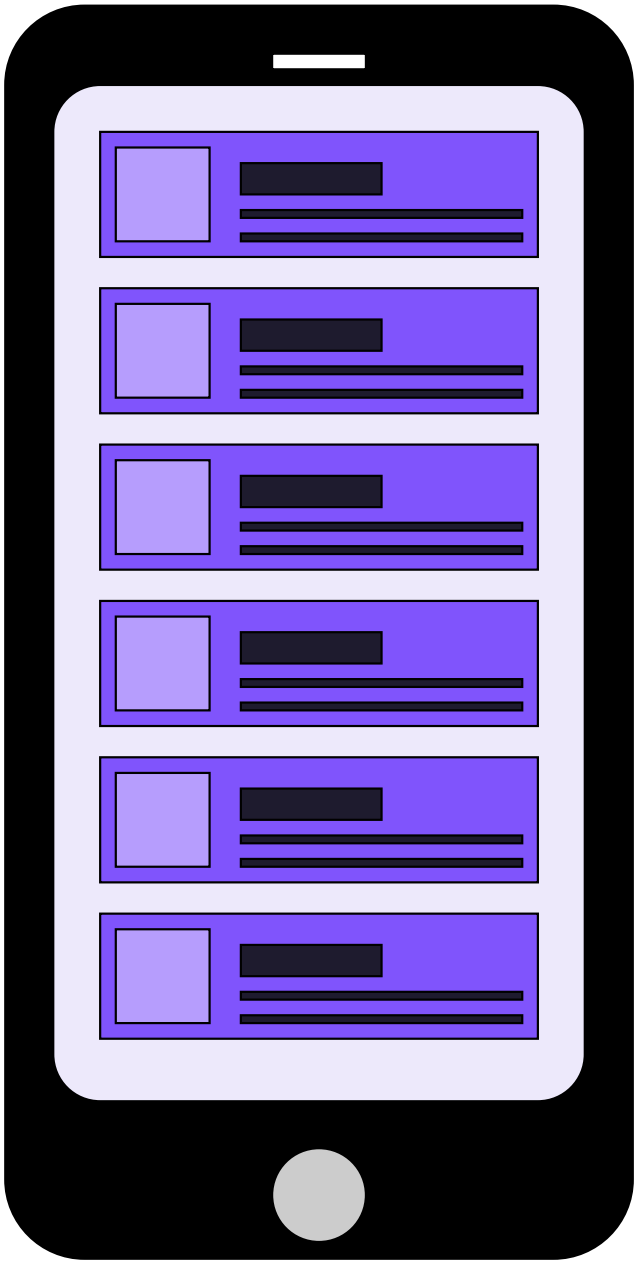


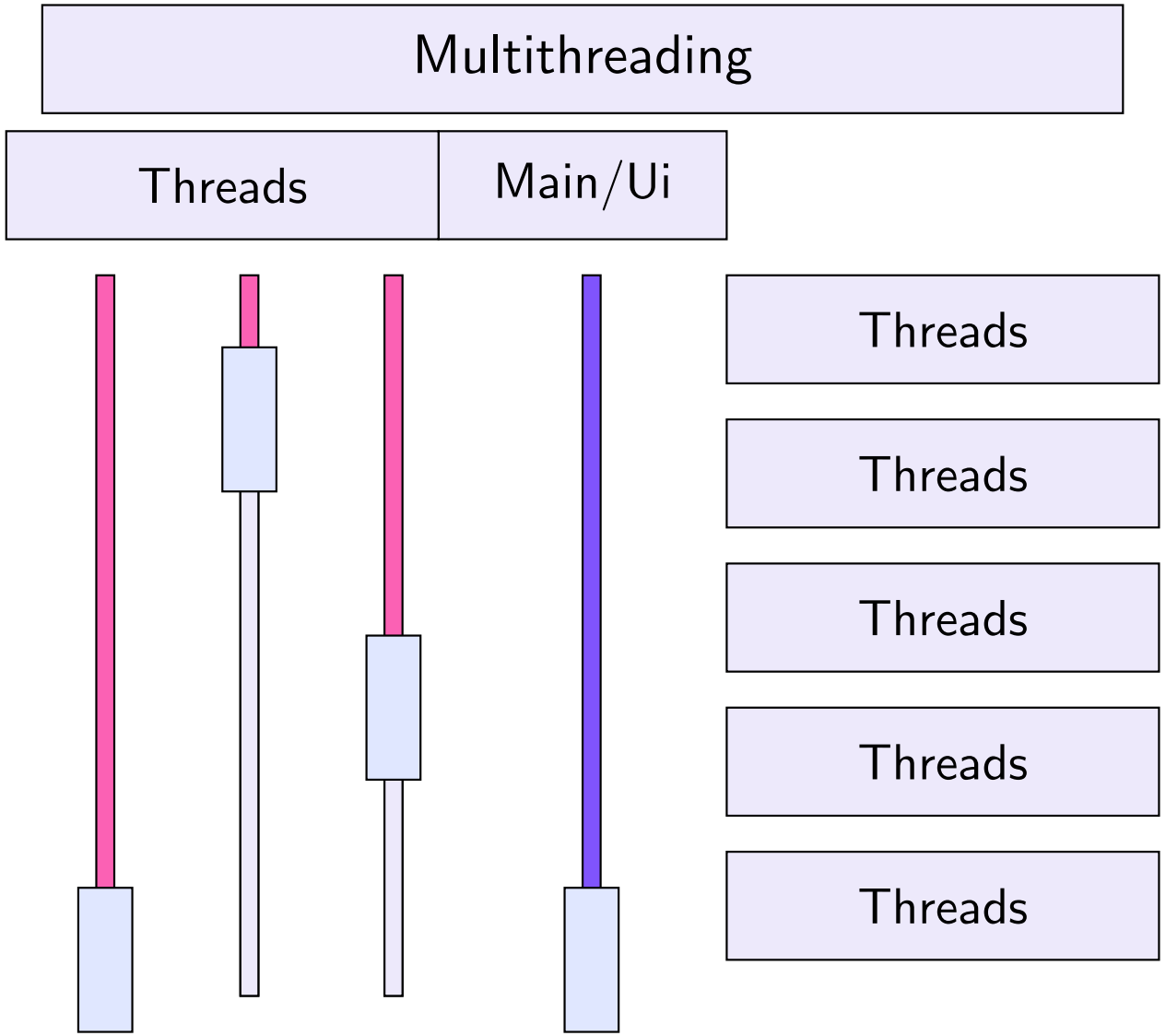
Android: Coroutines: Beispiel



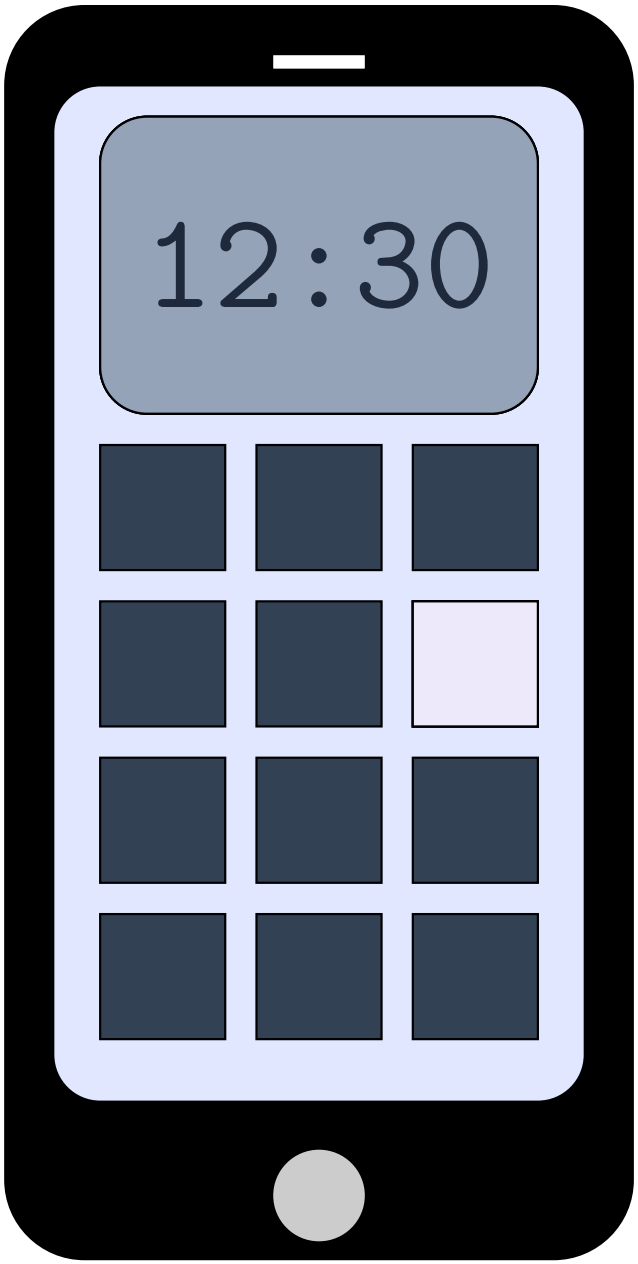


Android: Coroutines: Beispiel



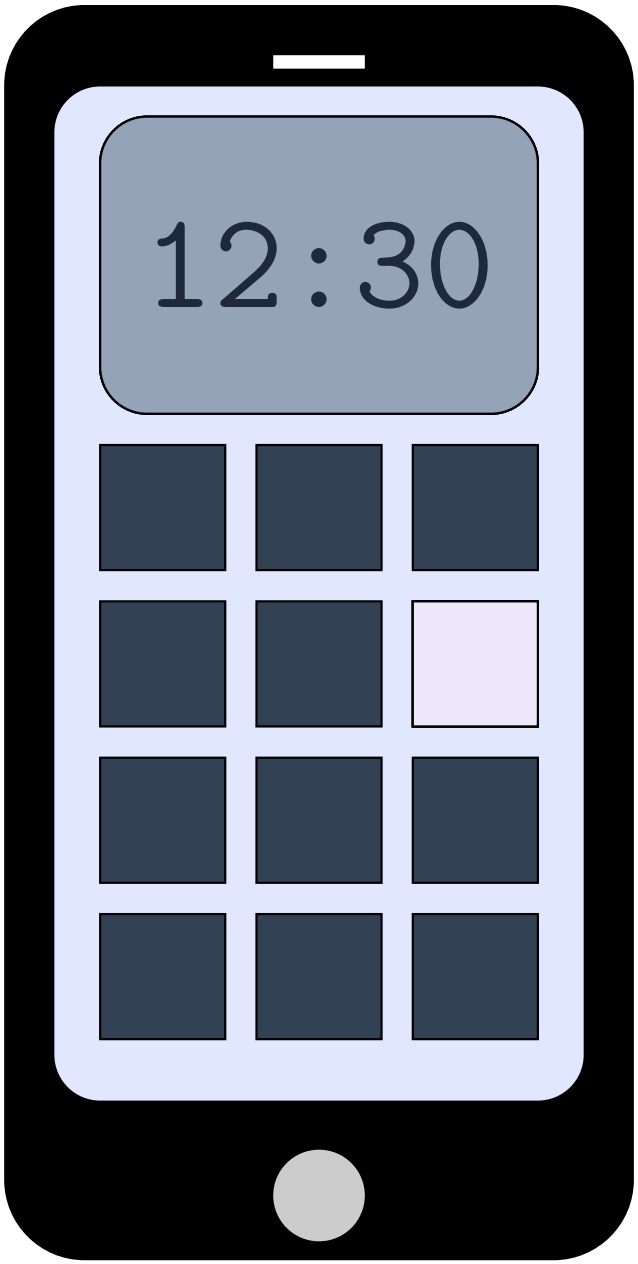


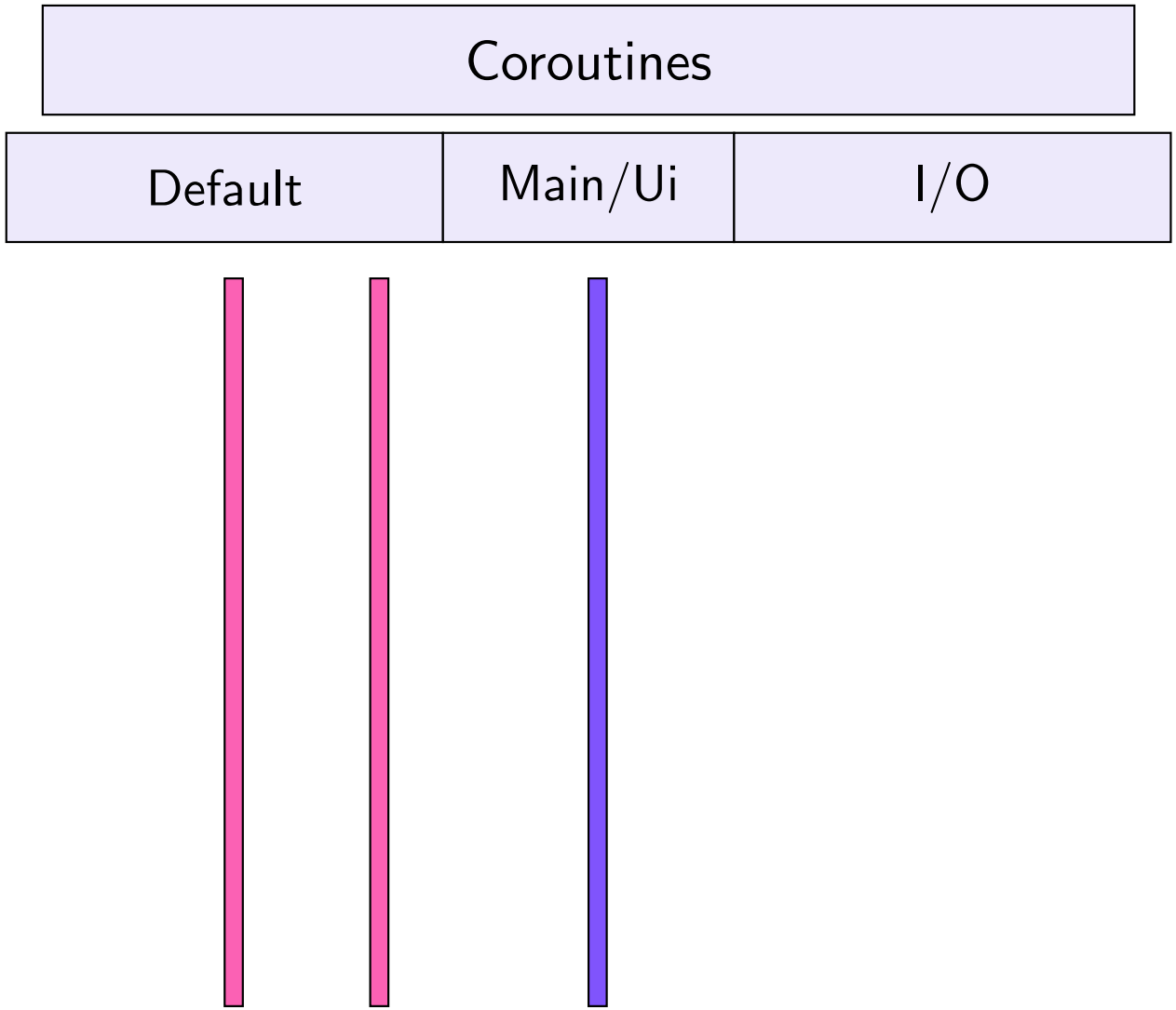
Android: Coroutines: Beispiel



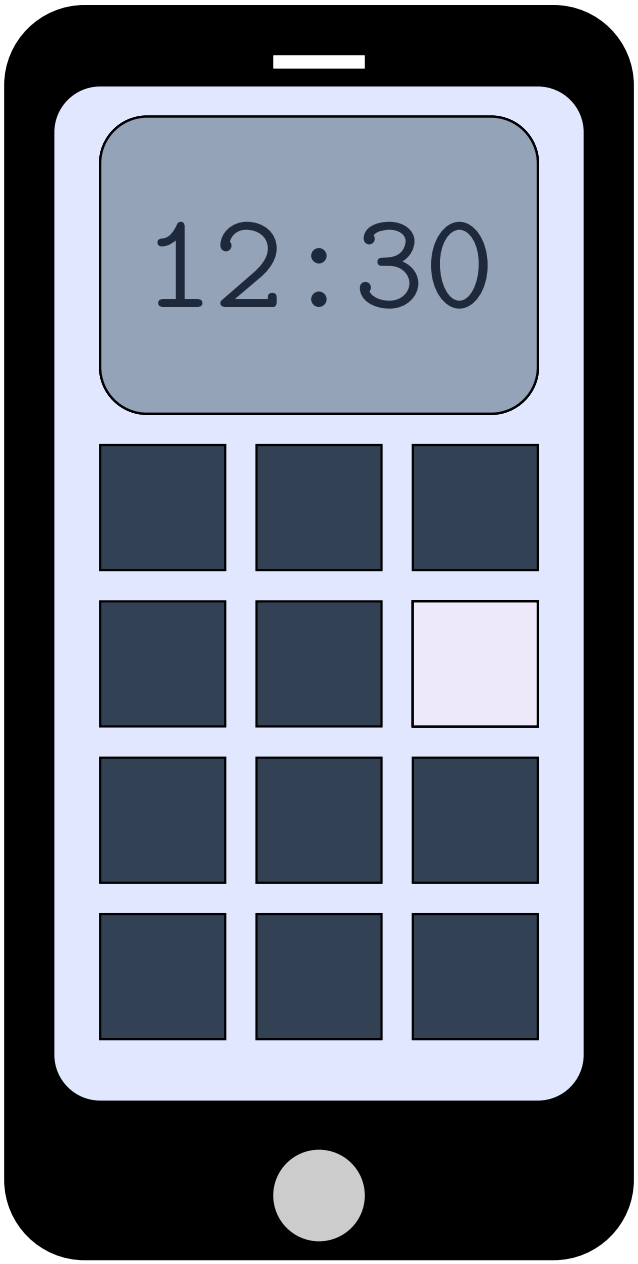
Coroutines

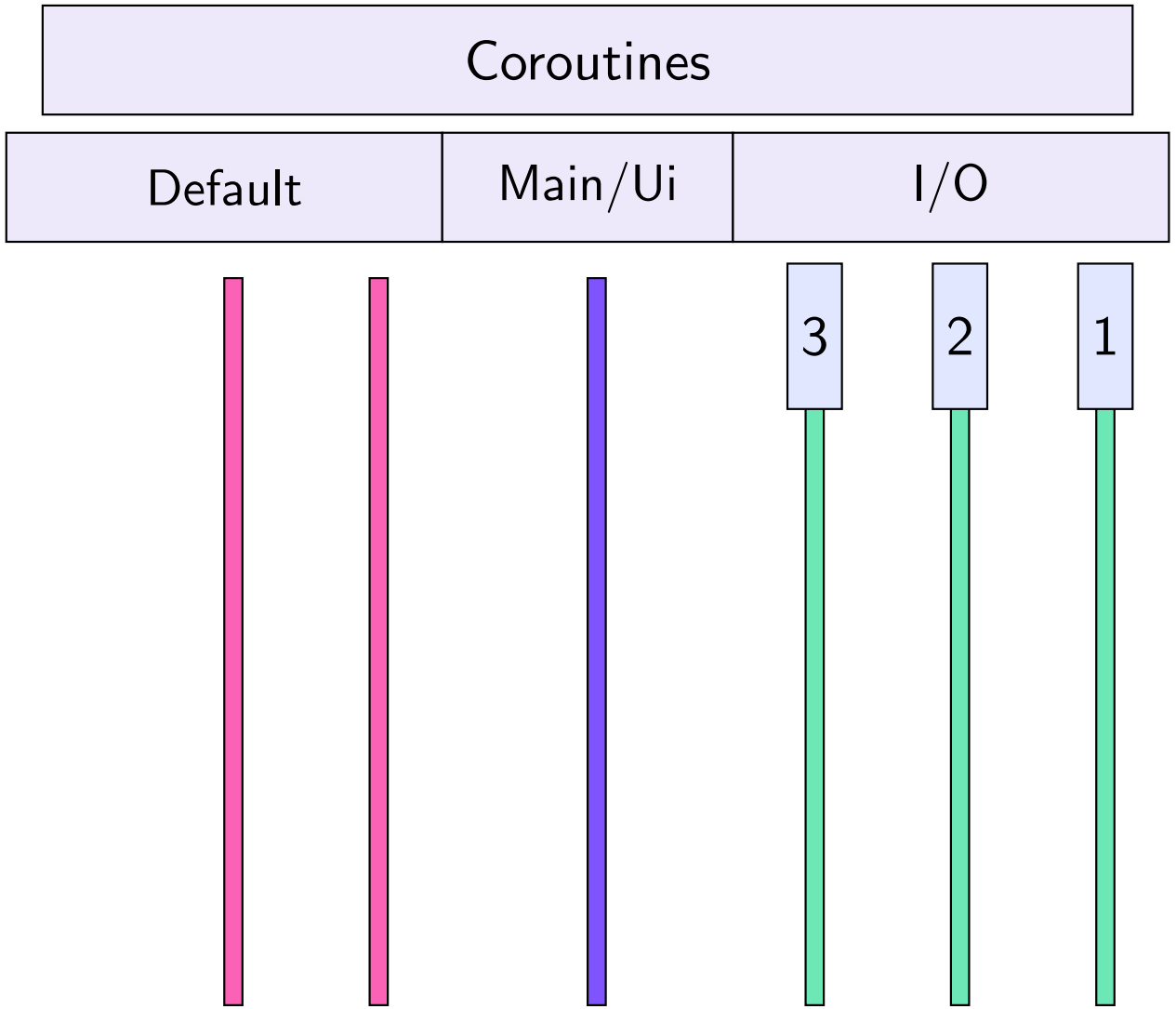
Android: Coroutines: Beispiel



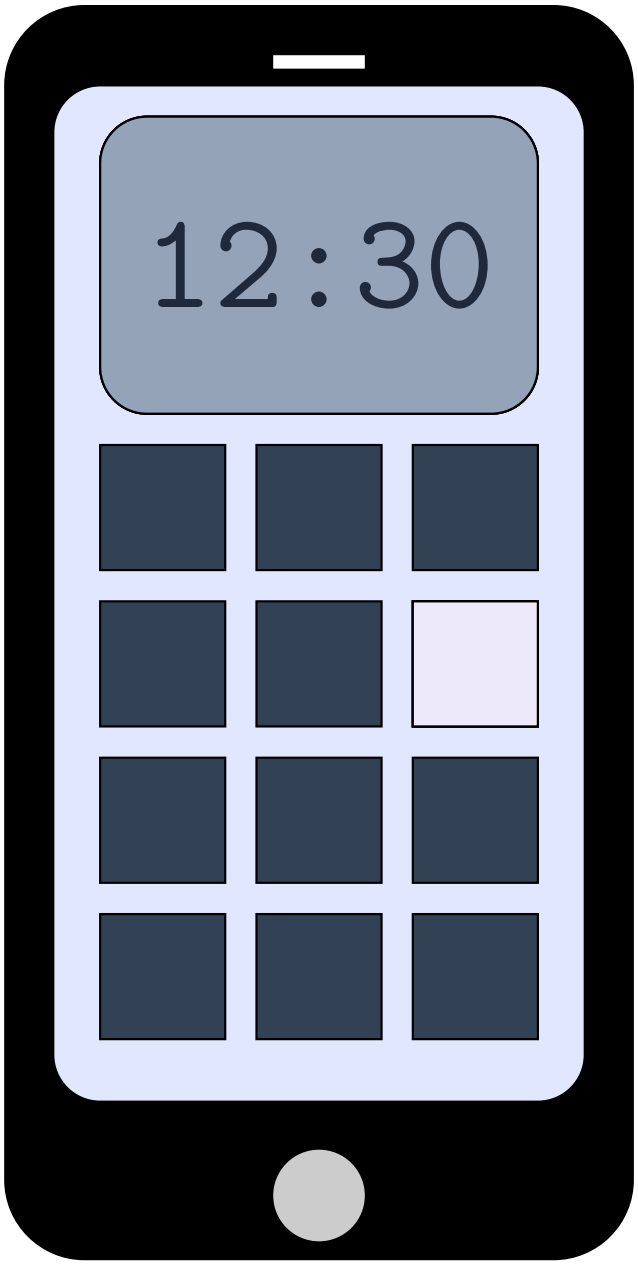


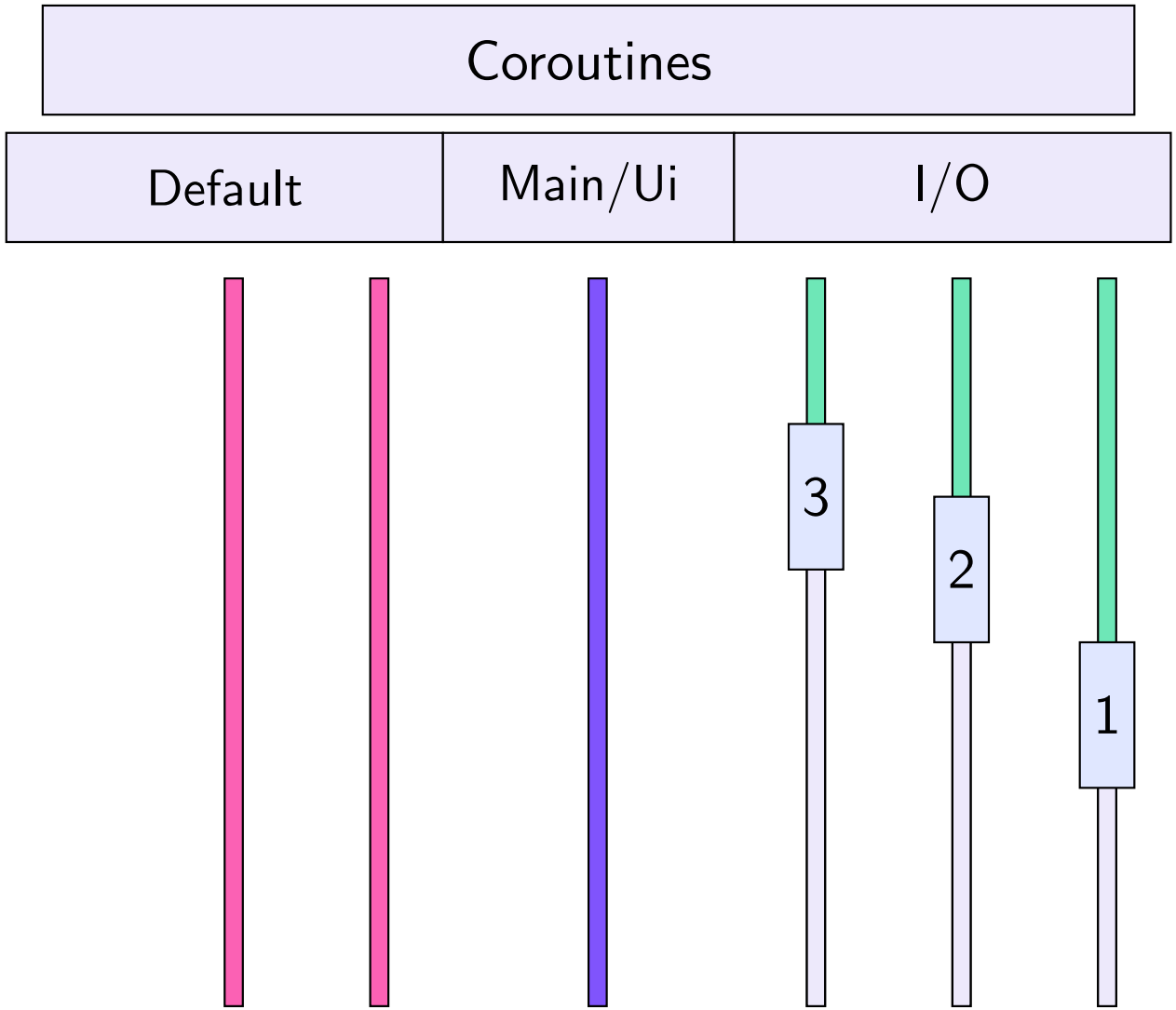
Android: Coroutines: Beispiel



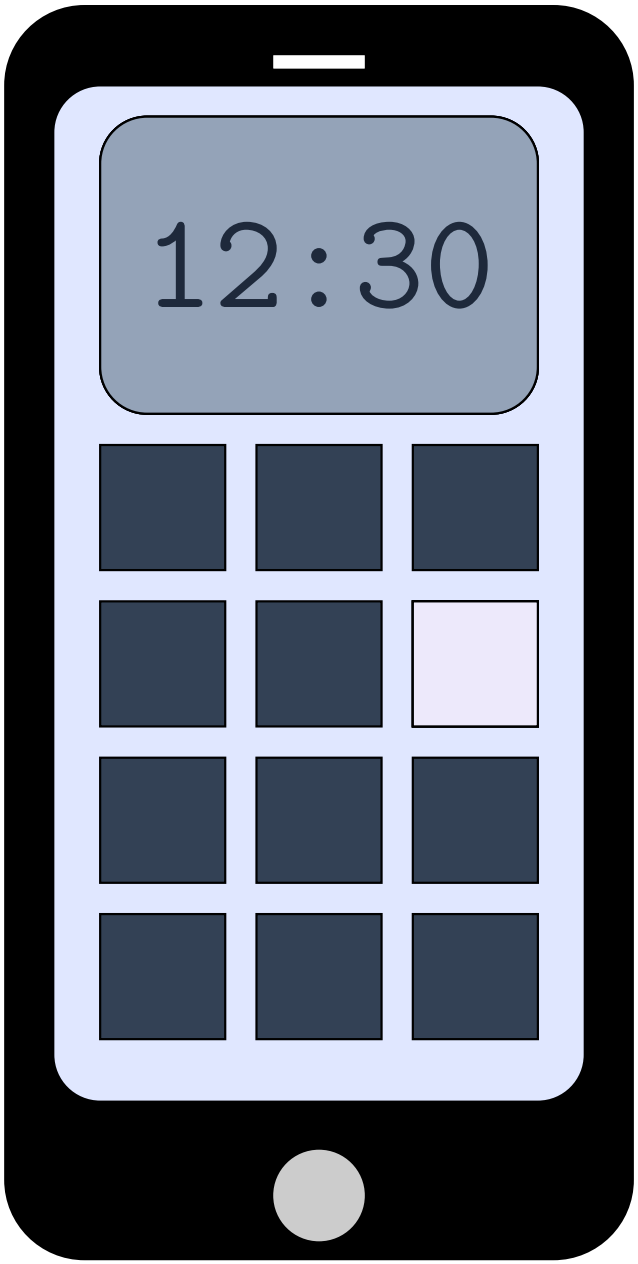


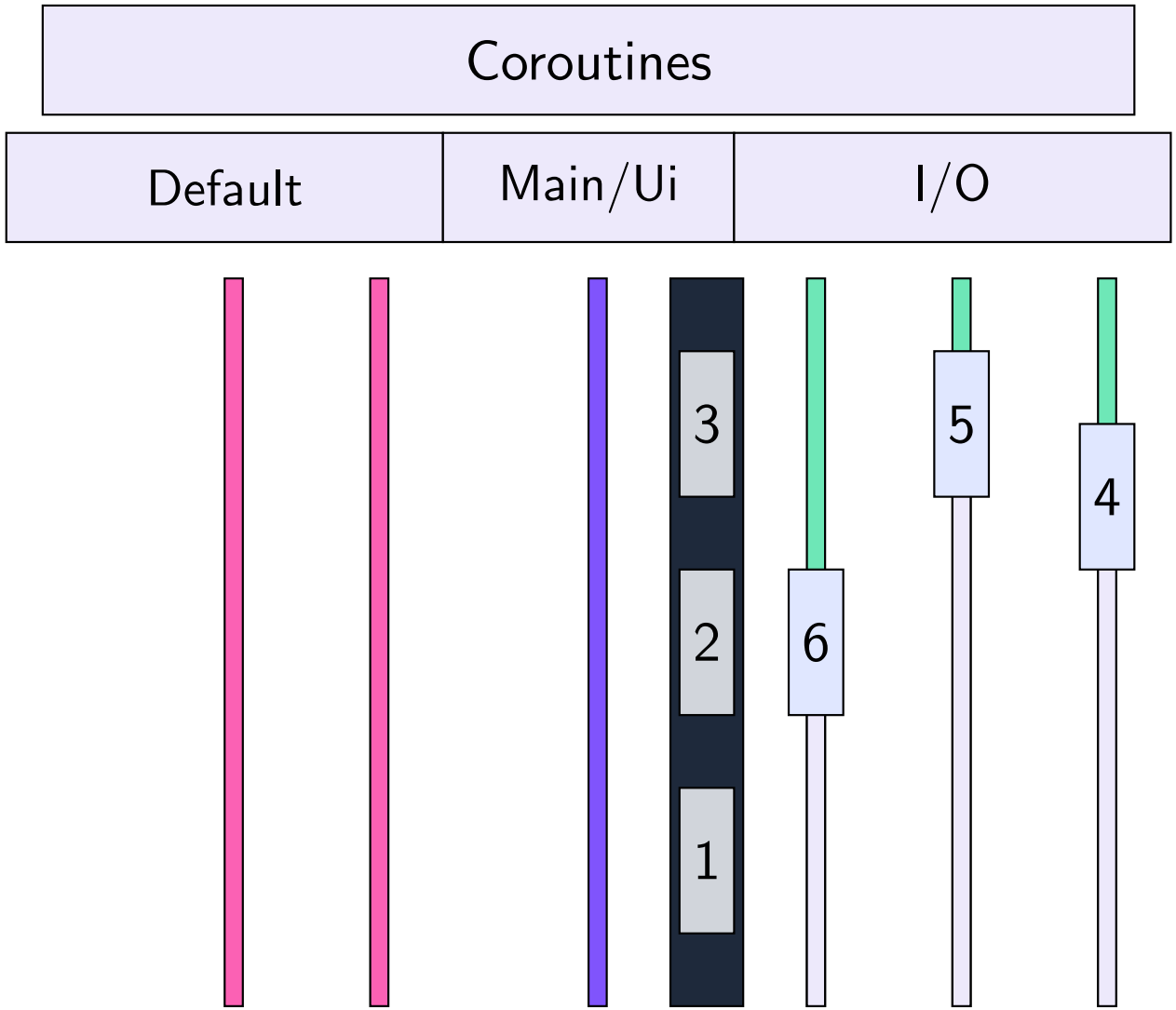
Android: Coroutines: Beispiel



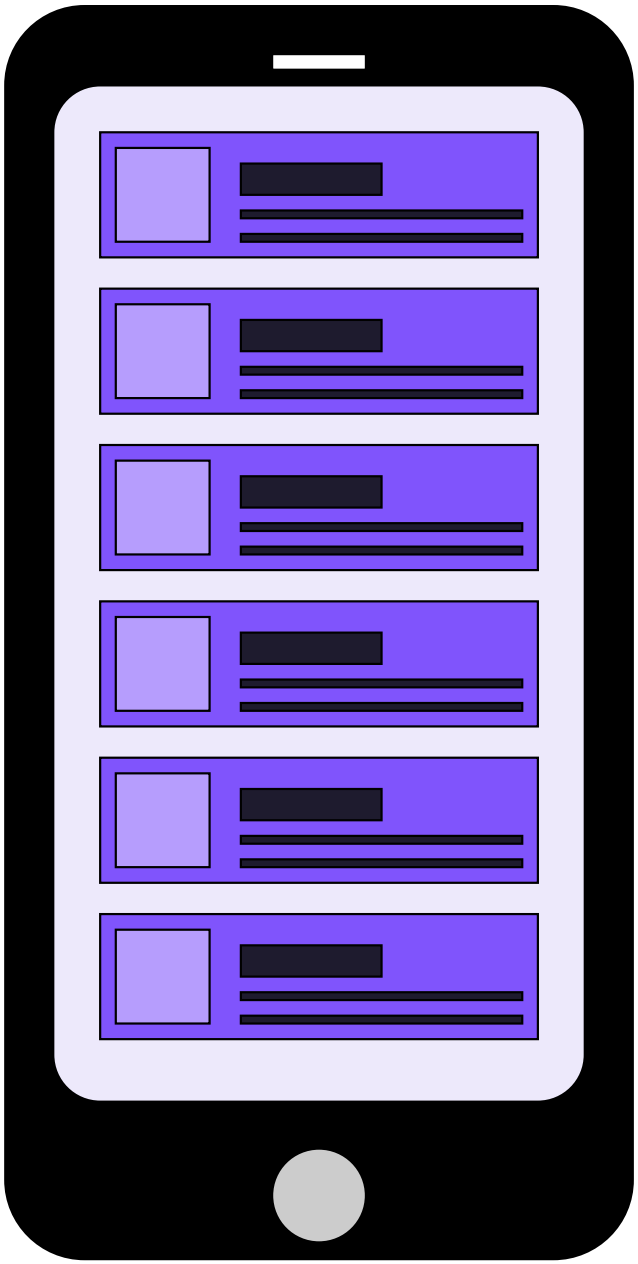


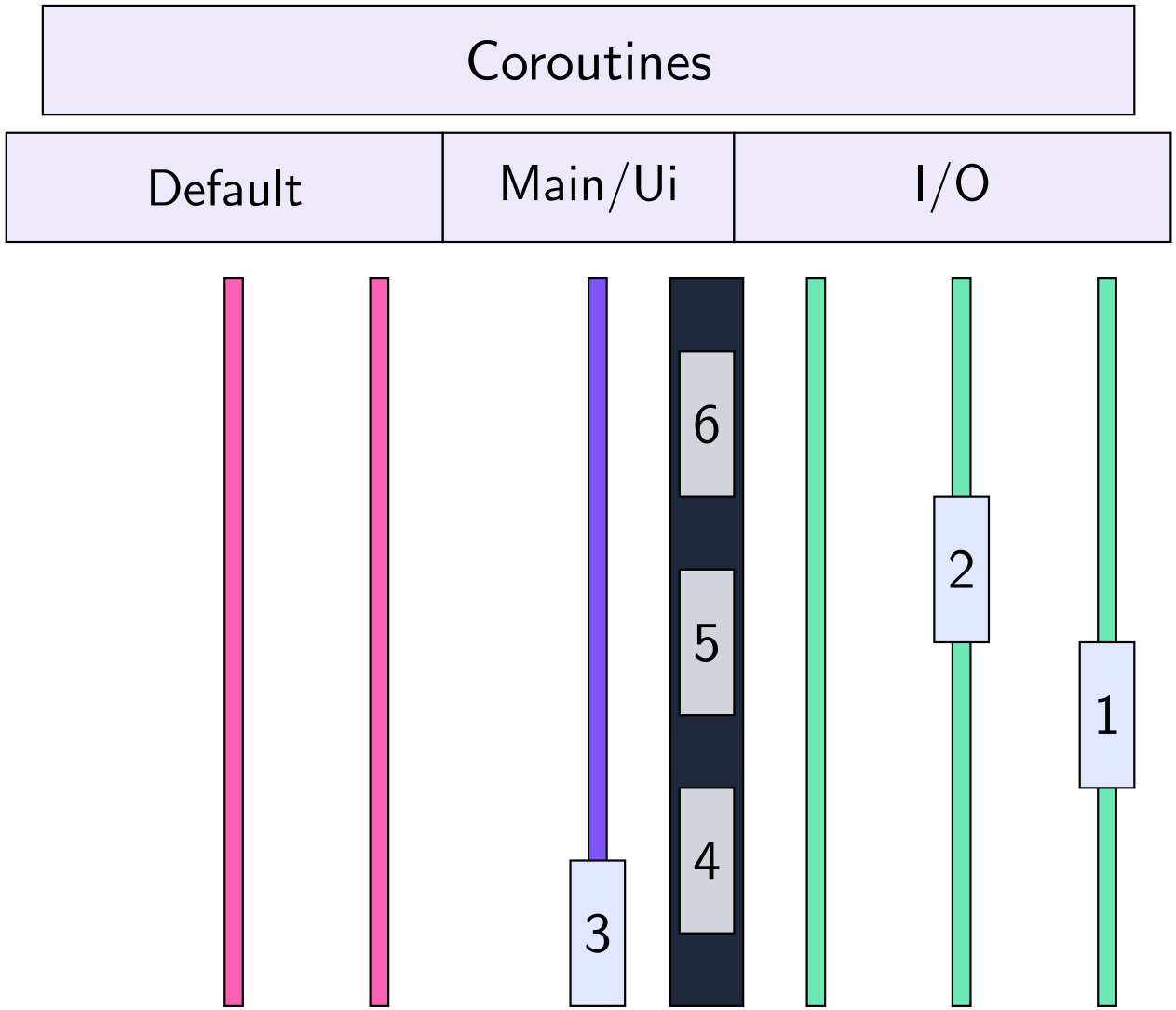
Android: Coroutines: Beispiel



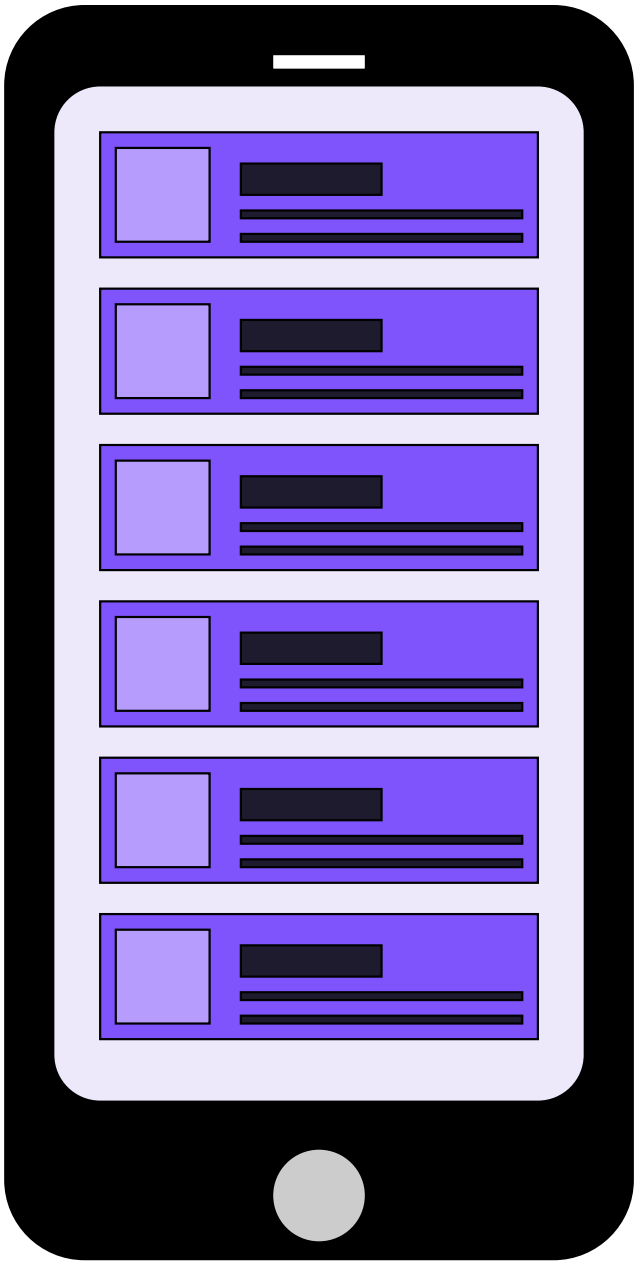


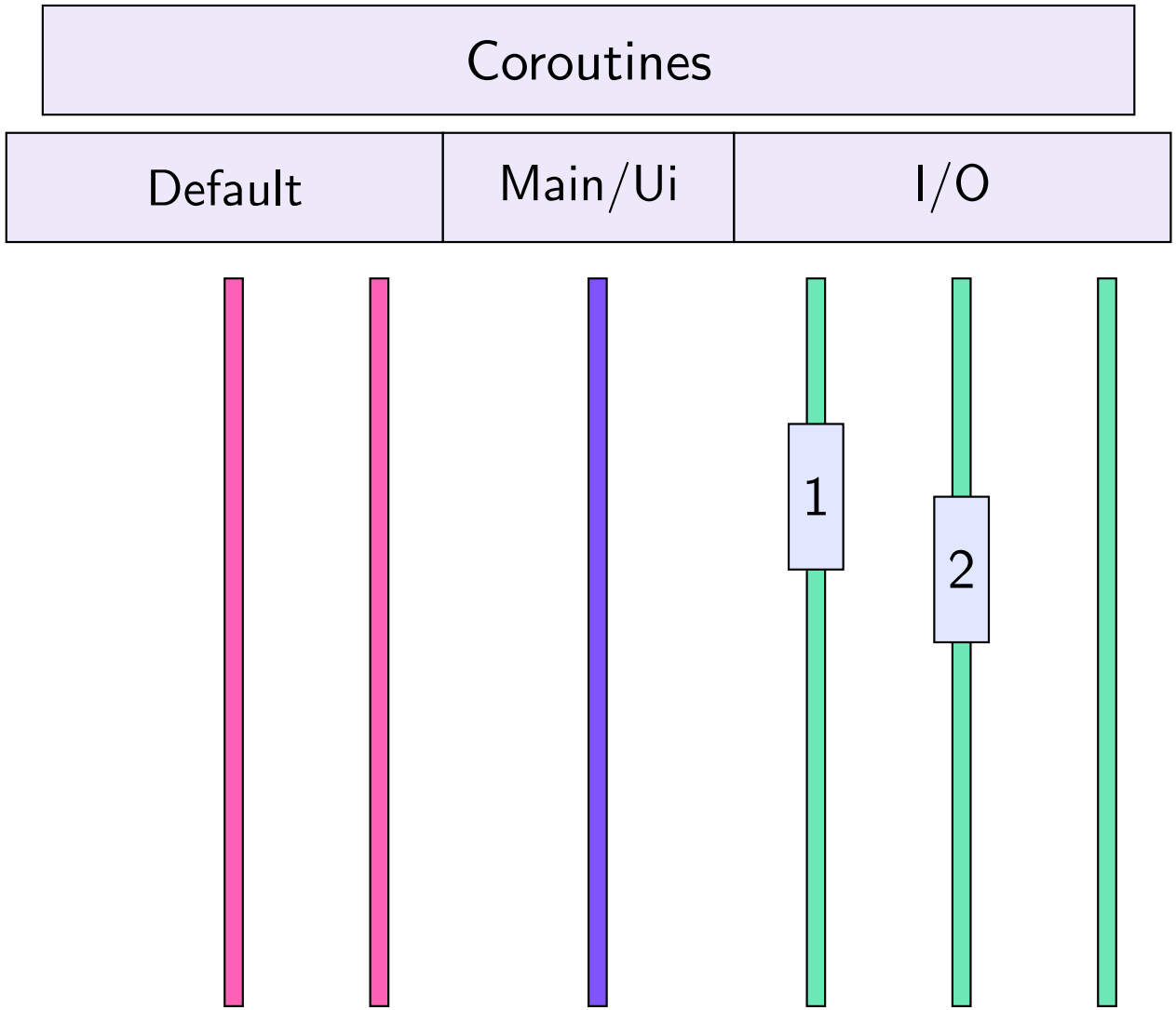
Android: Coroutines: Beispiel



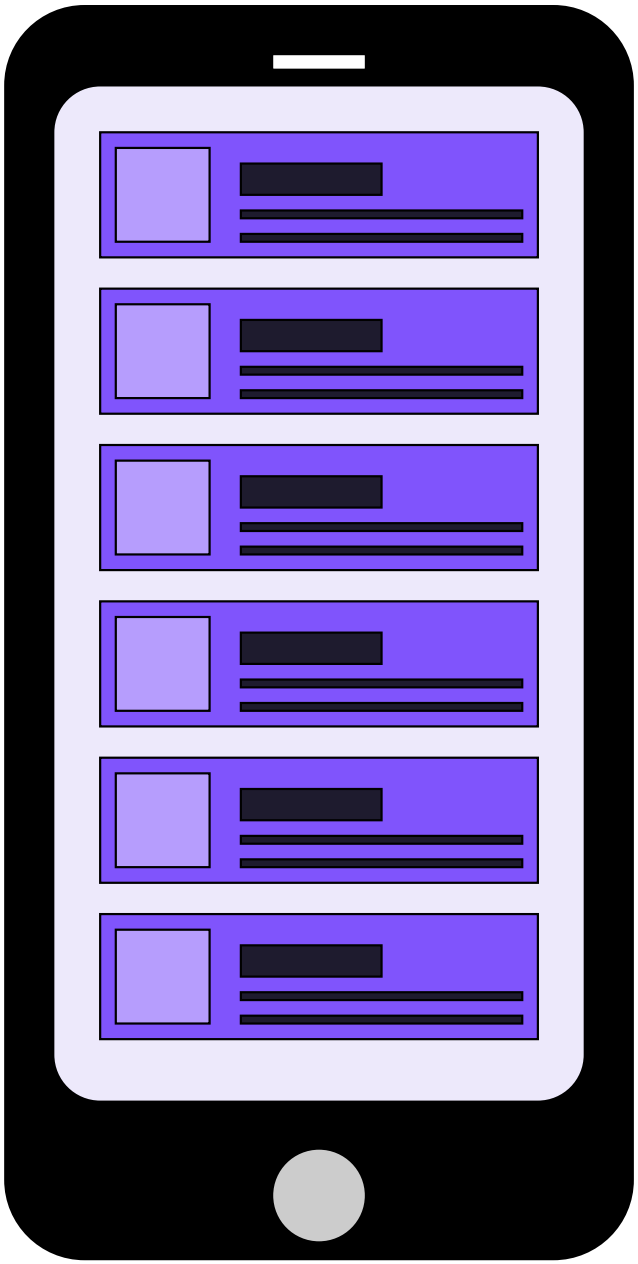


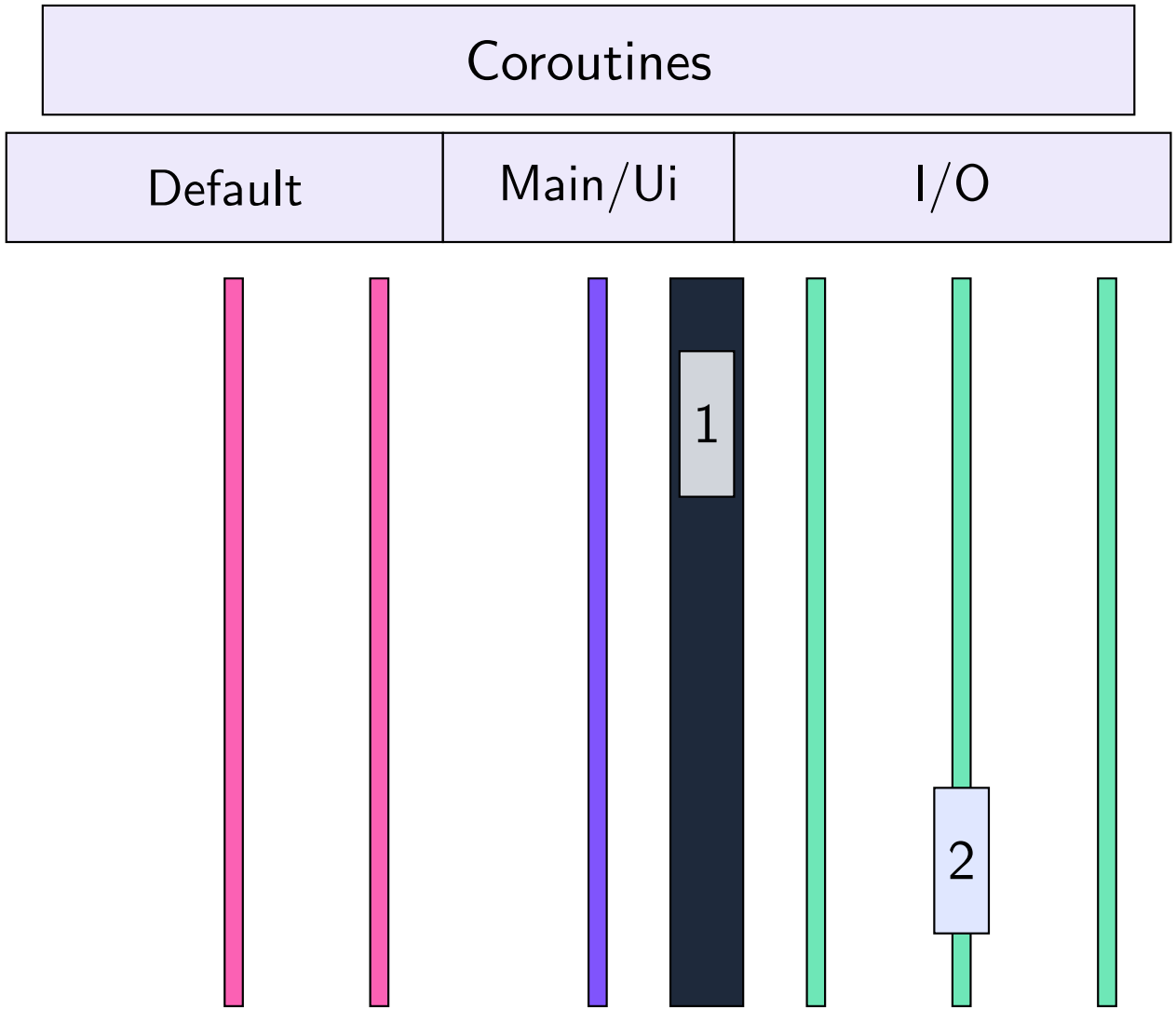
Android: Coroutines: Beispiel



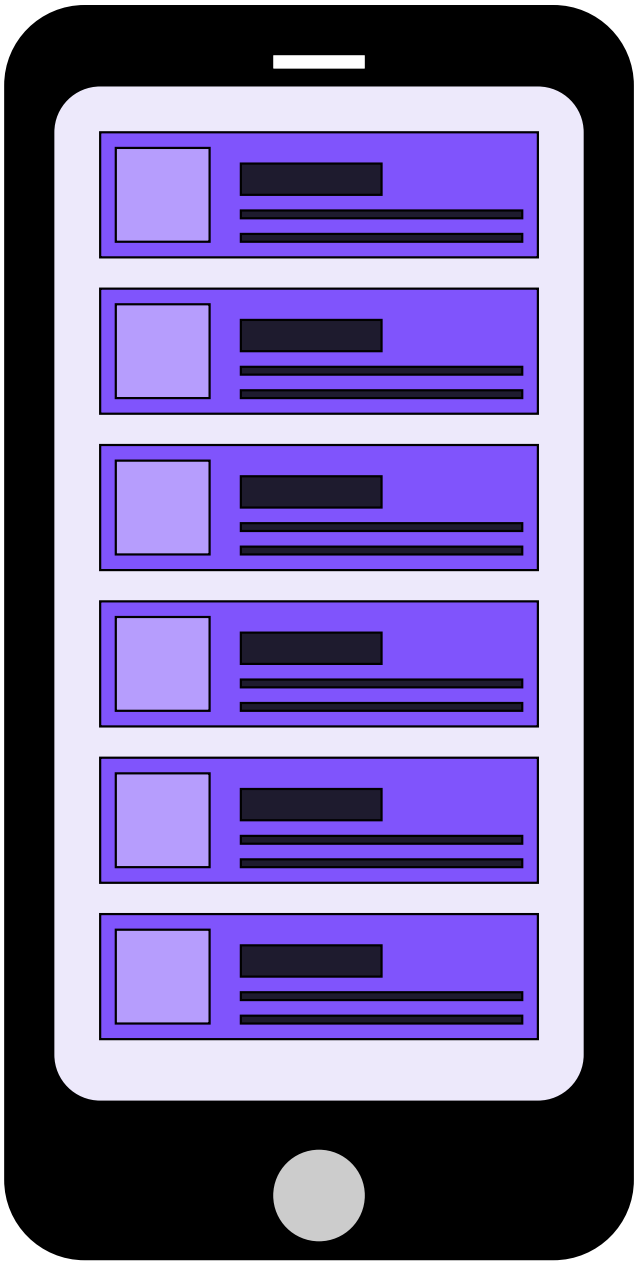


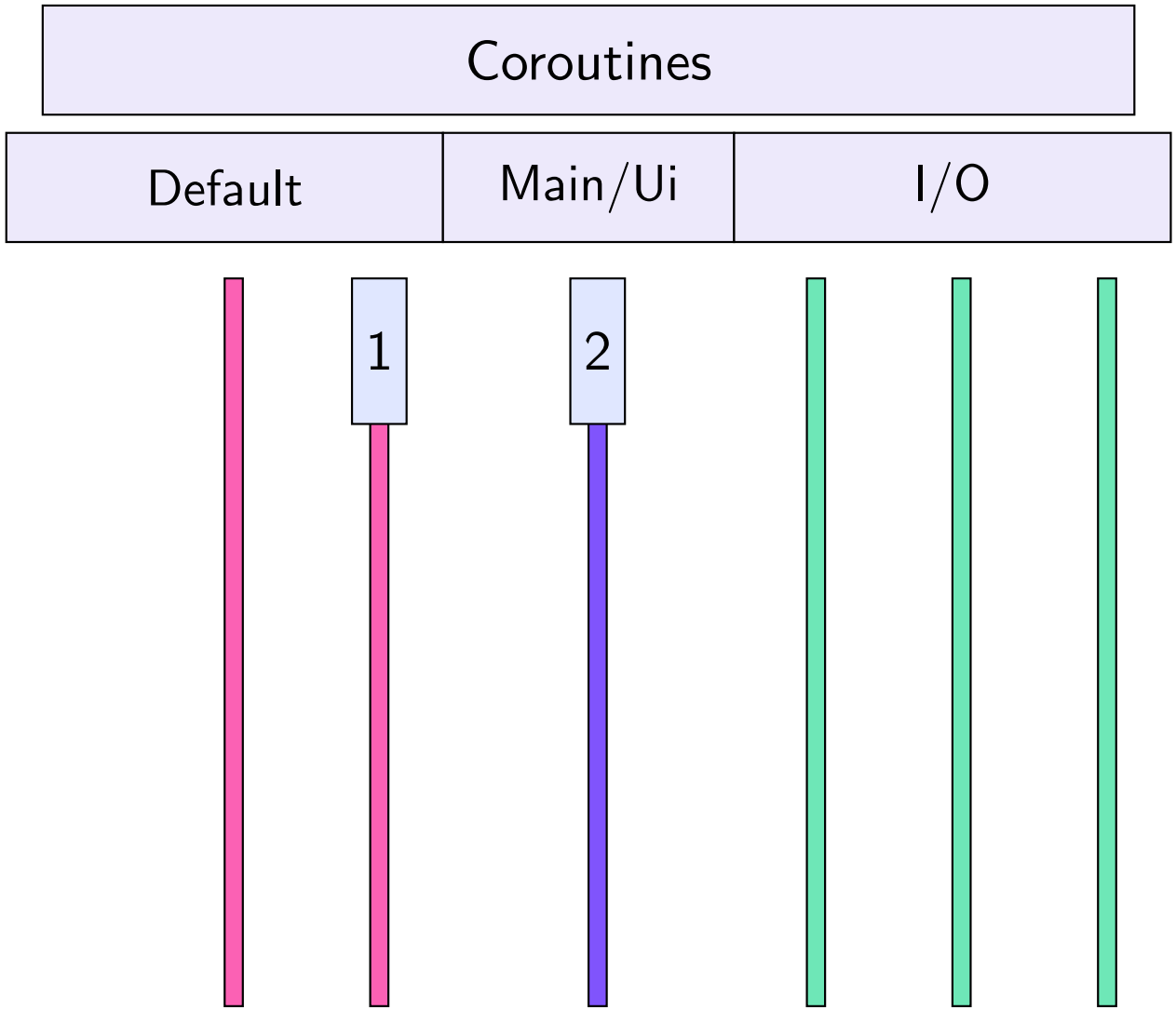
Android: Coroutines: Beispiel



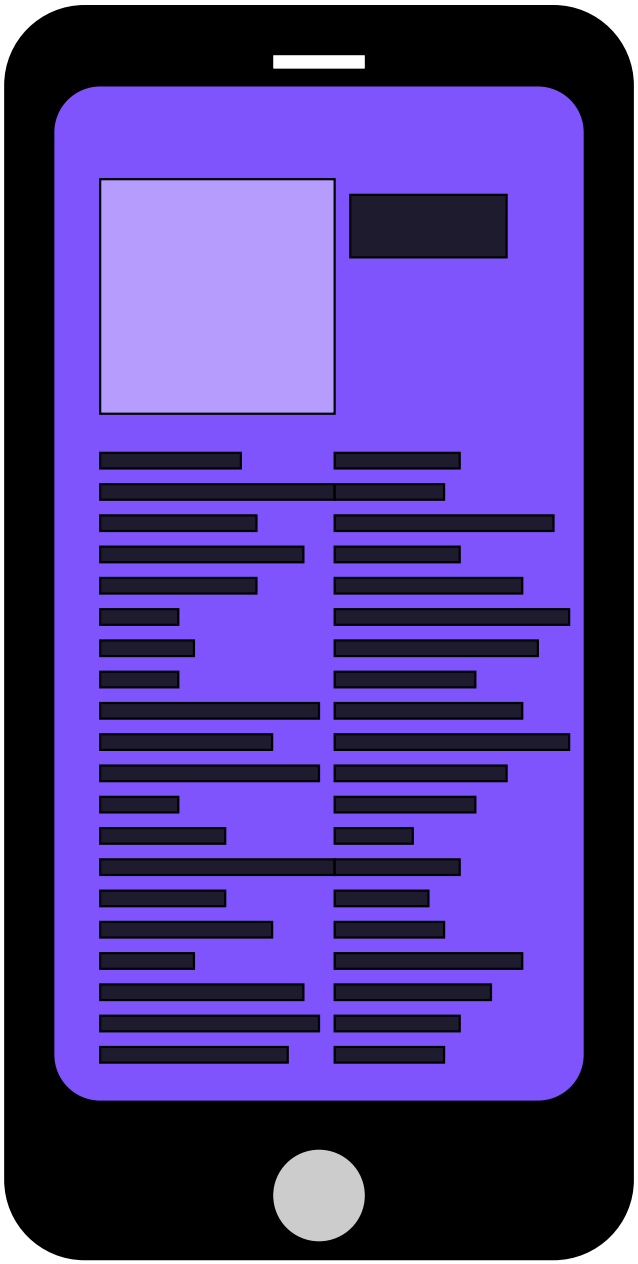


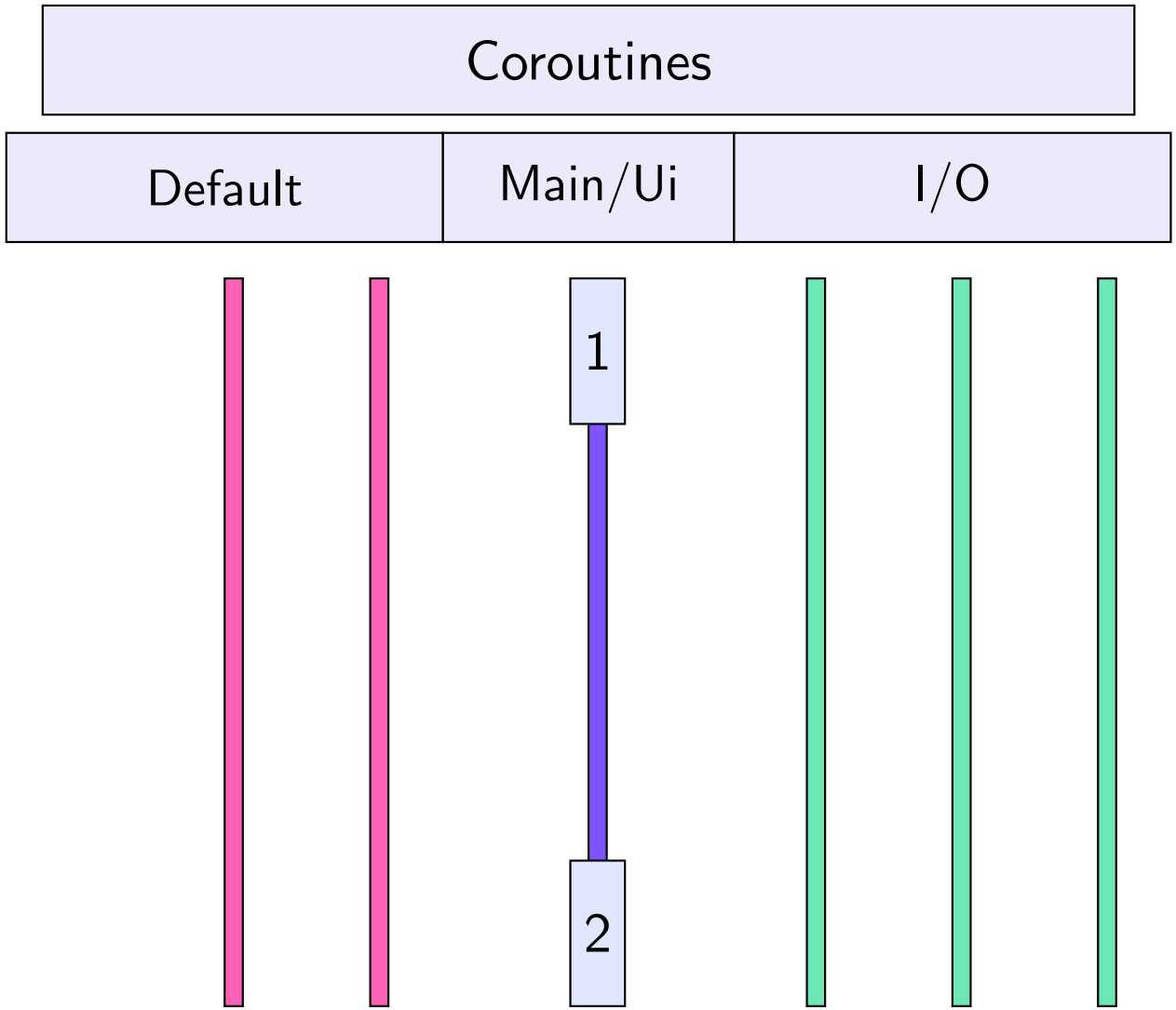
Android: Coroutines: Beispiel





Android: Coroutines: Beispiel





Android: Coroutines: Beispiel