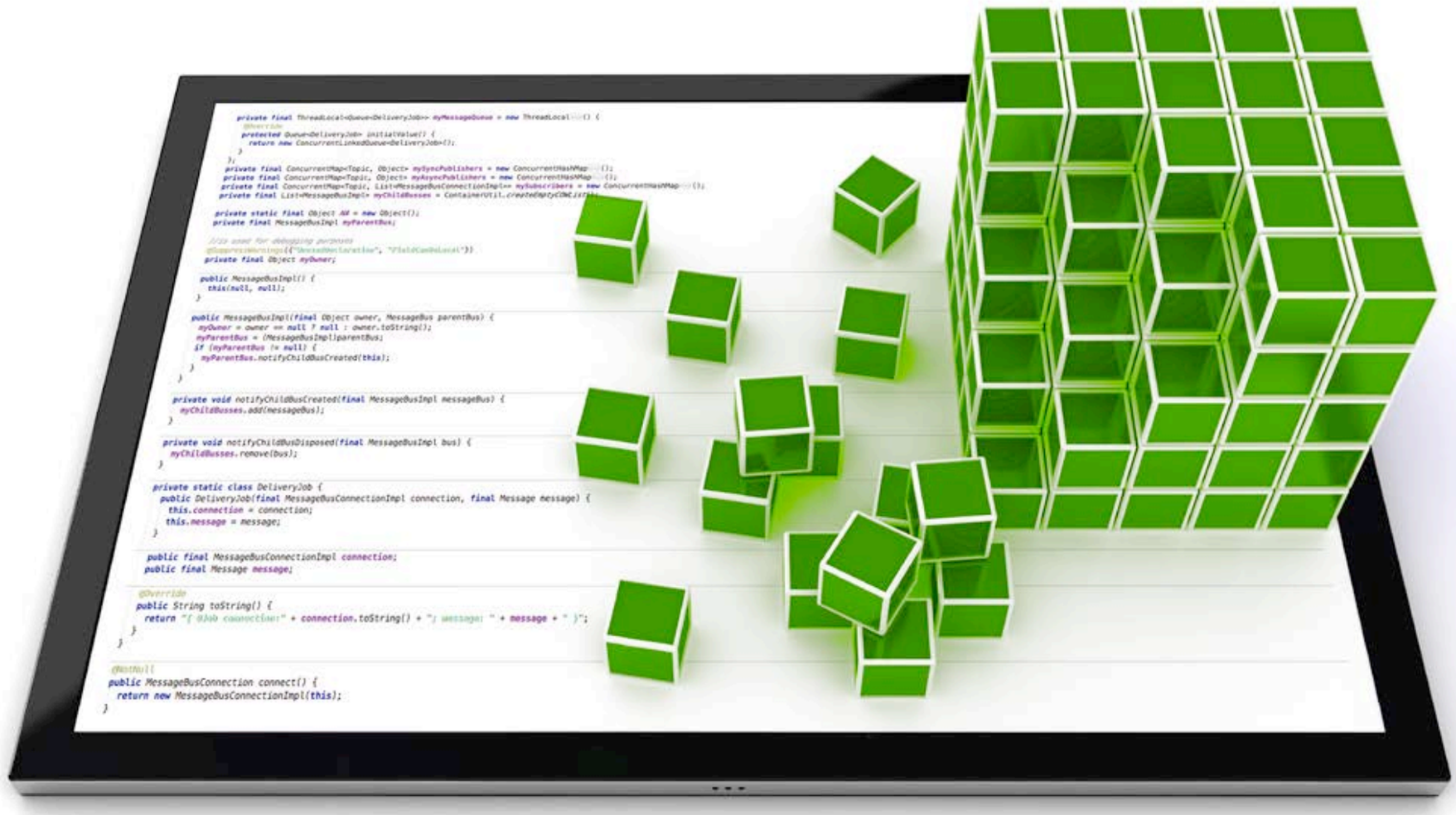
Kotlin

@dirkdittert



Was ist Kotlin?

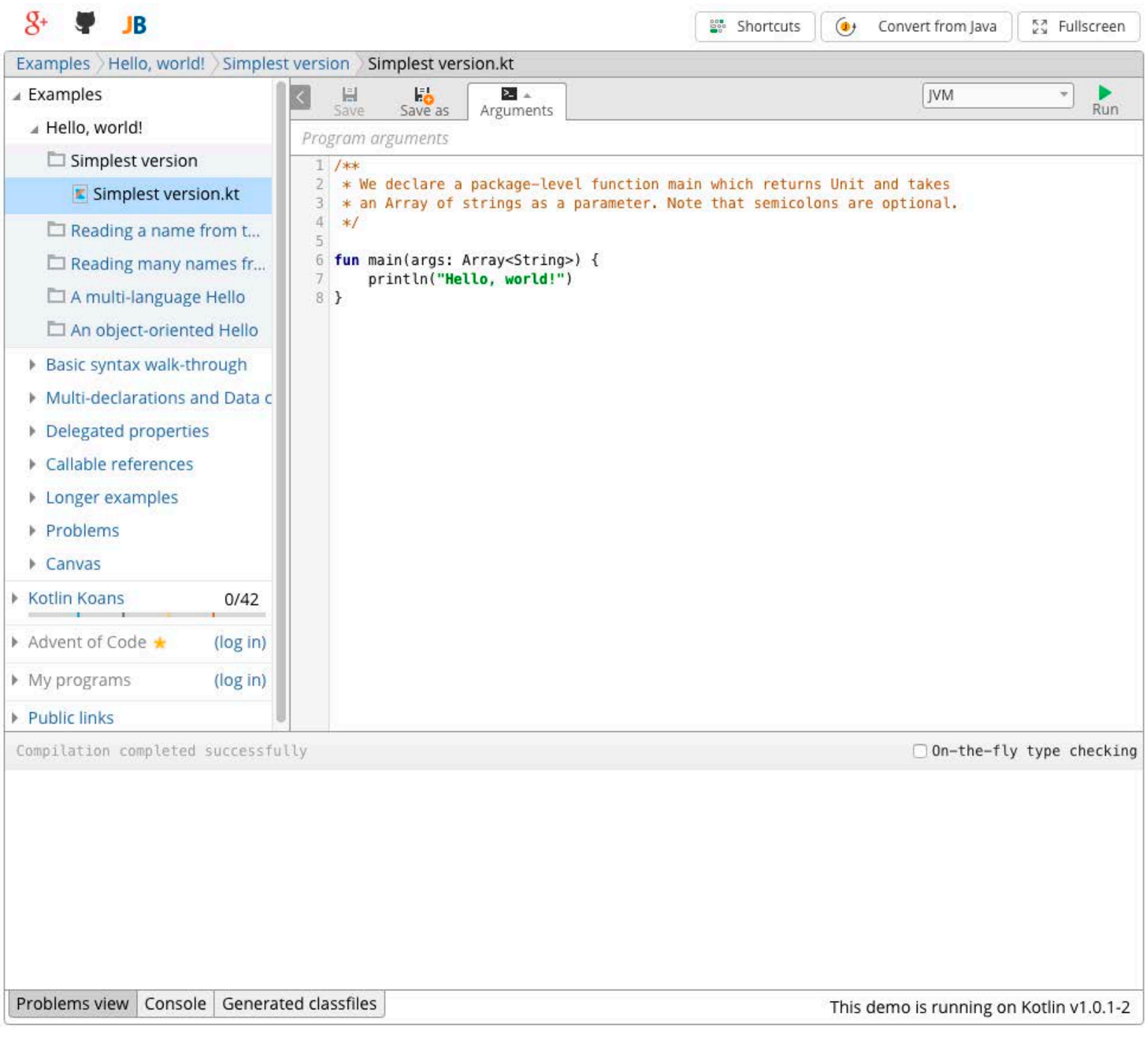
- Eine Insel, ca. 30km westlich von St. Petersburg in der Ostsee
- Peter der Große eroberte die Insel 1703 von den Schweden und gründete auf der Insel die Festungsstadt Kronstadt
- Kronstädter Pegel (am Fuße der Festung): Höhenbezug des früheren Ostblocks und bis 1993 auch der DDR

Hello World!

```
package tfo

data class KotlinEvent(val name: String, val ort: String)


fun main(args: Array<String>) {
    val event = KotlinEvent("Java Forum", "Stuttgart")
    println("Herzlich willkommen ${event.name} in ${event.ort}!")
}
```

```
> Herzlich willkommen Java Forum in Stuttgart!
>
> Process finished with exit code 0
```

http://try.kotlinlang.org



Die Menge der Kleinigkeiten ist groß!

- Keine Strichpunkte, kein new
- Variablendeklaration mit **val** und **var**; Typ kann automatisch abgeleitet werden

```
val str = """A multiline string
             with Backslashes \ from ${Date()}""".trimIndent()
```

```
var firstExecution: LocalTime? = null
```

- Einfache Funktionen können sehr kurz formuliert werden, da viele Sprachkonstrukte Ausdrücke sind:

```
fun isEven(i: Int) = if (i.mod(2) == 0) "gerade" else "ungerade"
```

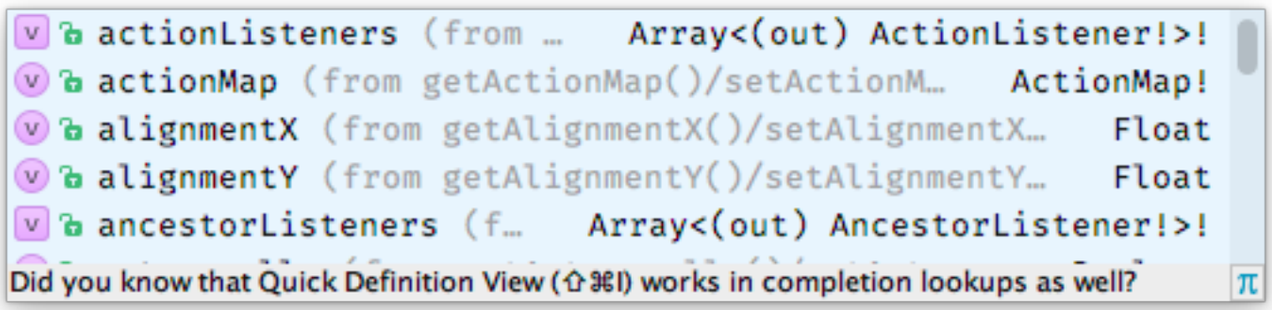
- Einige Java Sprachelemente sind in Kotlin Bibliotheksfunktionen:
  - **try-with-Resources** (`use(block: (T) → R): R`)
  - **synchronized** (`synchronized(lock: Any, block: () → R): R`)

Properties

- **Es gibt keine Member Variablen, nur Properties**
  `println(firstExecution.hour)`
  - **Compiler generiert Getter (und Setter) für Properties; diese können durch eigenen Code ersetzt werden**
  - **Properties können delegiert werden**

```
class PropertiesExample {
    var p1: String by Delegates.notNull()
}
```

- **Getter/Setter aus Java-Code können in Kotlin als Properties verwendet werden**



Operatoren (1)

- Operatoren können überladen werden

```
data class Complex(val re: Double, val im: Double) {
    operator fun plus(other: Complex)  = Complex(re + other.re, im + other.im)
    operator fun minus(other: Complex) = Complex(re - other.re, im - other.im)
}
```

```
val c1 = Complex(3.0, 5.0)
val c2 = Complex(4.0, 6.0)
val result = c1 + c2
```

- die Auswertungsreihenfolge und die Menge der Operatoren ist fest vorgegeben

Operatoren (2)

- **Folgendes ist also nicht direkt möglich:**

```
fa <*> fb { (a, b) ⇒ c }                          // apply, produces F[C]
fa |@| fb |@| fc |@| ... { (a, b, c, ... ) ⇒ x } // apply, produces F[X]
```

(scalaz, http://bit.ly/1WNVWls)

- **Infix Funktionen in Verbindung mit Unicode erlauben Kreativität**

```
infix fun Int.▒▒(to: Int): IntProgression =
        IntProgression.fromClosedRange(this.toInt(), to.toInt(), -1)
infix fun Int.`↣↠`(to: Int) = to - this

fun main(args: Array<String>) {
    println(1 ▒▒ 1)
    println(1 `↣↠` 1)
}
```

Die großen Unterschiede zu Java

- Kotlin kennt kein static!
- Java Klassenmethoden werden in Kotlin in Companion Objects untergebracht

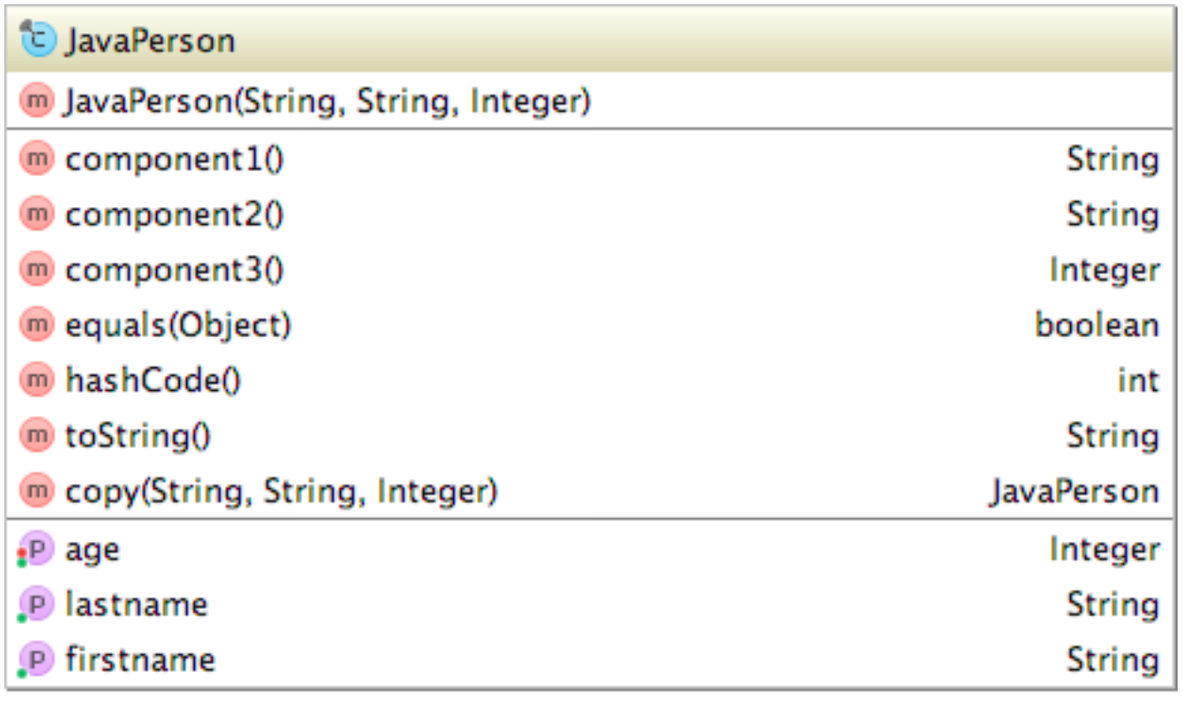
```
class ContentParser {
    companion object {
        fun create(): ContentParser {
            return ContentParser()
        }
    }
}

// Aufruf
ContentParser.create()
```

- Companion Objects sind normale Objektinstanzen und können dadurch auch Interfaces implementieren

Data Classes (1)

- Data Classes sind Klassen, die als Wert-Container dienen
- Der Compiler generiert alle zugehörigen Hilfsmethoden automatisch
- Data Classes können um eigene Methoden erweitert werden
- Aktuell ist keine Vererbung möglich (aber ab Version 1.1)

```
data class JavaPerson(
    val lastname: String,
    val firstname: String,
    var age: Int?
)
```



Data Classes (2)

```kotlin
data class JavaPerson(
    val lastname: String,
    val firstname: String,
    var age: Int?
)
```

```java
public final class JavaPerson {
  private final String lastname, firstname;
  private Integer age;

  public JavaPerson(@NotNull  String last,
                    @NotNull  String first, @Nullable Integer age) {
    this.lastname = Objects.requireNonNull(lastname);
    this.firstname = Objects.requireNonNull(firstname);
    this.age = age;
  }

  @NotNull  public String getLastname() { return lastname; }
  @NotNull  public String getFirstname() { return firstname; }
  @Nullable public Integer getAge() { return age; }
  @NotNull  public String component1() { return lastname; }
  @NotNull  public String component2() { return firstname; }
  @Nullable public Integer component3() { return age; }

  public void setAge(Integer a) {
    this.age = Objects.requireNonNull(a);
  }
```

```java
  @Override public boolean equals(Object o) {
        if (this == o) { return true; }
        if (o == null || getClass() != o.getClass()) { return false; }
        JavaPerson that = (JavaPerson) o;
        if (!lastname.equals(that.lastname)) { return false; }
        if (!firstname.equals(that.firstname)) { return false; }
        return age != null ? age.equals(that.age) : that.age == null;
    }

  @Override
  public int hashCode() {
    int result = lastname.hashCode();
    result = 31 * result + firstname.hashCode();
    result = 31 * result + (age != null ? age.hashCode() : 0);
    return result;
  }

  @Override
  public String toString() {
    final StringBuilder sb = new StringBuilder("JavaPerson{");
    sb.append("lastname='").append(lastname).append('\'');
    sb.append(", firstname='").append(firstname).append('\'');
    sb.append(", age=").append(age);
    sb.append('}');
    return sb.toString();
  }

  @NotNull public JavaPerson copy(@NotNull String lastname,
                @NotNull String firstname, @Nullable Integer age) {
      return new JavaPerson(lastname, firstname, age);
  }
}
```

Funktionen

- Funktionen sind eigenständige Sprachelemente und können in Kotlin ohne Klassen existieren
- Funktionen können innerhalb von Funktionen deklariert werden

```
fun traverse(list: List<String>) {
    fun recurse(list: List<String>, index: Int) {

    }

    recurse(list, 0)
}
```

- Funktionen können als **`inline`** deklariert werden

```
public inline fun <T> Iterable<T>.forEach(action: (T) → Unit): Unit {
    for (element in this) action(element)
}
```

Code von Inline Funktionen funktioniert wie Makros und wird direkt an der Stelle des Aufrufs generiert (d.h. kein Funktionsaufruf im ByteCode)

Extension Functions

- Extension Functions ermöglichen die Erweiterung beliebiger Klassen (oder Companion Objects) um eigene Methoden
- Extension Functions werden durch den Compiler als statische Methoden realisiert und müssen dadurch zustandslos sein
- Beispiel:

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
  val tmp = this[index1] // 'this' entspricht der Instanz der Liste
  this[index1] = this[index2]
  this[index2] = tmp
}
```

- Extension Functions werden **immer statisch aufgelöst**, nicht virtuell (also nicht abhängig vom Typ der tatsächlichen Objektinstanz)!

Eine Frage des Standpunkts

- Kotlin erlaubt zu bestimmen, was im Code unter `this` zu verstehen ist
- Bei Extension Functions ist dies die Instanz der Klasse, auf der die Funktion aufgerufen wird
- Bei Lambdas kann ein Objekt, das als „Empfänger“ dienen soll, angegeben werden:

```
fun <T, R> with(rec: T, block: T.() → R): R = rec.block()
```

`rec:` Receiver vom Typ `T`

`block:` Funktion ohne Parameter mit einem Receiver vom Typ `T` und einem Rückgabewert vom Typ `R`

- Extrem mächtig, aber die „fremdeste“ Syntax für Java Entwickler in Kotlin.

Praktische Idiome

- `with` – Ruft den Block mit dem ersten Parameter als Empfänger (`this`) auf und gibt das Ergebnis zurück

```
val s = with(StringBuilder()) {
    append("Starting ")
    append("Countdown ")
    (10 downTo 1).joinTo(this, ", ")
    toString()
}
```

- `apply` – ähnlich wie `with`; gibt aber immer den Empfänger (`this`) zurück

```
val b = JButton().apply {
    isVisible = true
    text = "Hello!"
    minimumSize = Dimension(100, 20)
    addActionListener {
        println("Button was clicked")
    }
}
```

Praktische Idiome

- `let` – ruft den Block mit dem Wert als Argument auf

```
findButton()?.let { btn →
    btn.isVisible = true
    btn.text = "Hello!"
    btn.minimumSize = Dimension(100, 20)
    btn.addActionListener {
        println("Button was clicked")
    }
}
```

- `let` stellt dem zurück gegebenen Wert einen eigenen (beschränkten) Gültigkeitsbereich zur Verfügung, so dass keine temporäre Variable verwendet werden muss

Vergleich

```kotlin
val s = with(StringBuilder()) {
    append("Starting ")
    append("Countdown ")
    (10 downTo 1).joinTo(this, ", ")
    toString()
}
```

```java
final StringBuilder builder = new StringBuilder();
builder.append("Starting ");
builder.append("Countdown ");
for (int i = 10; i >= 2; i--) {
    builder.append(i);
    builder.append(", ");
}
builder.append("1");
final String s = builder.toString();
```

- Kotlin ist sehr prägnant und lenkt den Fokus auf die Intention während die Syntax von Java eher davon ablenkt
- Kotlin Variante funktioniert überall (d.h. auch für Properties einer Klasse/ globale Properties)
- Kein Overhead, da `with` eine inline Funktion ist

Null (1)

- **`null`** ist fester Bestandteil des Typsystems:

```
var nullableString: String? = null
var nonNullableString: String = "Hello world!"
```

- Compiler generiert für alle Methodenaufrufe automatisch entsprechende Annotationen und Prüfungen

```
fun process(str: String) { /*  ...  */ }                    // Kotlin

public void process(@NotNull String string) {               // Java
    Objects.requireNonNull(string);
    //  ...
}
```

- Compiler verwendet Prüfungen für automatische Typkonvertierungen:

```
val firstExecution: LocalTime? = null
println(firstExecution.hour)      // Compilerfehler
if (firstExecution ≠ null) {
    println(firstExecution.hour)  // ok (da firstExecution als val deklariert!)
}
```

Null (2)

- Annotierte Java-Methoden integrieren sich nahtlos mit Kotlin. Fehlen diese, verhält sich Kotlin ähnlich wie Java (d.h. reduzierte Prüfungen durch den Compiler)
- Manchmal gelten andere Regeln (z.B. bei Dependency Injection):

```
class ClassWithInjection {
    @Inject
    lateinit var shouldNeverBeNull: String
    fun prinLength() = shouldNeverBeNull.length
}
```

- Praktische Prüfungen:

  Erzwinge Wert ungleich Null: `str!!.toUpperCase().toByteArray().size`

  Aufruf, falls nicht Null: `str?.toUpperCase()?.toByteArray()?.size`

Beispiel

Für eine Reihe von Sendungen (`Shipment`) sollen zugehörige Versandunterlagen (`Manifest`) erstellt werden. Hierfür gilt eine komplexe Geschäftslogik.

Beispiel

```
data class Shipment(val country: String, val city: String, val numItems: Int)

fun dispatch(s: Shipment): Manifest {
    if (s.requiresTrucking()) {
        return manifest {
            location {
                fromShipment(s)
                usePreferredShipper = s.numItems ⩽ 33
            }
        }
    } else if (s.requiresAirFreight()) {
        return manifest {
            location {
                fromShipment(s)
            }
            instructions {
                ! "Handle with special care!"
                ! "Do not transport on passenger planes!"
            }
        }
    } else {
        return putInStorage()
    }
}
```

Arrays

Integration von Arrays ist problematisch und Verwendung wird nicht empfohlen:

- **Vergleich von Arrays verhält sich anders als der Vergleich von Listen**

```
arrayOf(1, 2, 3) == arrayOf(1, 2, 3) // false
listOf(1, 2, 3) == listOf(1, 2, 3)   // true
```

- **Initialisierung mehrdimensionaler Arrays ist für „Profis“**

```
val data: Array<Array<Boolean>> =
          Array(width) { Array(height) { false } }
```

- **Annotationen werden mit Arrays etwas länglich**

```
public @interface RequestMapping { // gekürzt, aus Spring
  String[] path() default {};
}

@RequestMapping(path = arrayOf("/"))
```

Unterstützung für Java 8/9

- Version 1.0 ist für Java 6 ausgelegt, d.h. der erzeugte Bytecode verwendet für Lambdas Klassen
- In den Basistypen fehlen Methoden und Möglichkeiten, die in Java 7 und Java 8 hinzugekommen sind
  - Klassen können nicht mehrfach mit Annotationen mit Runtime Retention versehen werden
  - z.B. bei den Exceptions sind manche Methoden erst mit Java 7/8 hinzu gekommen/`AutoCloseable`
- Java 8 Stream API funktioniert nicht in Kotlin (da keine Default Methods in Interfaces möglich sind)
- Workaround: Erweiterungsbiliothek `kotlinx-support-jdk8` http://bit.ly/1XUngoP

Frameworks

- Spring Boot: http://bit.ly/1YM9PON
- Kotlin als DSL für Gradle: http://bit.ly/1swfQVp
- Exposed (Kotlin SQL Framework): http://bit.ly/1RfSWGv
- Anko (DSL für Android Entwicklung): http://bit.ly/1swfqyh
- funKTionale (Funktionale Erweiterungen): http://bit.ly/1U5Vzz1
- RxKotlin (Adapter für RxJava): http://bit.ly/1OReE9e
- HamKrest (Kotlin Portierung der Hamcrest Matcher): http://bit.ly/241LMwC
- KotlinTest (Testing Framework): http://bit.ly/29lgLQT
- und vieles andere mehr!

Roadmap für Kotlin 1.1

Spracherweiterungen werden im Kotlin Evolution and Enhancement Process (KEEP) diskutiert: https://github.com/Kotlin/KEEP

- async/await/yield (http://bit.ly/1Wi6hpR)
- Vererbung bei Data Classes (implementiert)
- Type Aliases (implementiert)

```
typealias MouseEventHandler = (MouseEvent) → Unit
```

- Destructuring Lambdas

```
for ((k, v) in myMap) {    /* 1.0 */  }
myMap.forEach { (k, v) → /* 1.1. */ }
```

- Unterstützung für Java 8/9
- Unterstützung für Project Jigsaw

Roadmap für Kotlin 1.1

- Unterstützung von Javascript als Zielplattform
- Unterstützung aller Kotlin Sprachkonstrukte
  - 1.0.2: Nested/Inner/Local classes & non-local returns
  - Keine Reflection Unterstützung unter Javascript
  - Viele Kleinigkeiten (`println(2147483647 + 2147483647)` oder `println(intArrayOf(1)[1])`)
- Integration der Typdefinitionen der wichtigsten Javascript Frameworks (aus der Typescript Community)

Fazit

- Produktionstauglich: Ja (ab 1.0.2)! Für Javasscript und Vorsichtige ab Version 1.1.1
- Lernkurve ist für Java Entwickler relativ flach
- Syntax ist modern, orientiert sich an der Praxis und macht Spaß
- Behandlung von `null` ist nicht neu (IntelliJ IDEA), aber jetzt deutlich besser lesbar
- Leider muss auch Kotlin mit den Fehlern der JVM Implementierung leben:
  - Arrays
  - Type Erasure bei Generics
  - Fehlende Value Types (d.h. int vs. Integer) mit den Konsequenzen (Boxing)