2 Basistypen und -variablen

Eine Variable sein

Es gibt eine Sache, von der jeder Code abhängt: Variablen.

In diesem Kapitel werfen wir einen Blick unter die Motorhaube und zeigen Ihnen, wie ***Kotlin-Variablen tatsächlich funktionieren***. Sie werden Kotlins **Basisdatentypen** wie *Integer*, *Floats* und *boolesche Werte* kennenlernen. Sie werden sehen, wie Kotlins Compiler den **Typ einer Variablen anhand des übergebenen Werts feststellen** kann. Außerdem lernen Sie den Einsatz von **String-Templates** für die Erstellung komplexer Strings mit wenig Code sowie das Anlegen von **Arrays**, um mehrere Werte zu speichern. Abschließend kümmern wir uns noch um die Frage: »*Warum sind Objekte für das Leben in Kotlinville so wichtig?*«

Ihr Code braucht Variablen

Bis jetzt haben Sie gelernt, einfache Anweisungen, Ausdrücke, `while`-Schleifen und `if`-Tests zu schreiben. Für wirklich guten Code fehlt Ihnen aber noch eine wichtige Zutat: Variablen.

Wie man Variablen deklariert, wissen Sie bereits:

```
var x = 5
```

Der Code sieht einfach aus. Was passiert aber hinter den Kulissen?

Eine Variable ist wie ein Becher

Stellen Sie sich eine Kotlin-Variable wie einen Becher vor. Becher gibt es in verschiedenen Formen und Größen – es gibt die riesigen Becher für das Popcorn im Kino oder auch einfache Zahnputzbecher. Dennoch haben sie etwas gemeinsam: die Aufgabe, etwas zu enthalten.

Die Deklaration einer Variablen ist vergleichbar mit der Bestellung eines Getränks bei einer Kaffeehauskette. Sie teilen dem (oder der) Barista mit, welches Getränk Sie haben möchten und welcher Name ausgerufen werden soll, wenn es fertig ist. Vielleicht sagen Sie noch, ob Sie einen schicken wiederverwendbaren Becher oder doch lieber einen Wegwerfbecher haben möchten. Zum Beispiel:

```
var x = 5
```

Mit diesem Code teilen Sie dem Kotlin-Compiler den Wert der Variablen mit und ob sie für andere Werte wiederverwendet werden kann.

Für die Erzeugung einer Variablen muss der Compiler drei Dinge wissen:

- **Welchen Namen soll die Variable haben?**
  Über den Namen können wir die Variable im Code ansprechen.
- **Ist die Variable wiederverwendbar?**
  Angenommen, die Variable hätte den Startwert 2. Soll es möglich sein, diesen später in 3 zu ändern, oder soll er für die gesamte Laufzeit des Programms 2 bleiben?
- **Welcher Datentyp wird für die Variable verwendet?**
  Ist es ein Integer (ganzzahliger Wert), ein String? Oder etwas Komplexeres?

Sie haben bereits gesehen, wie eine Variable benannt wird und wie durch die Schlüsselwörter `val` und `var` festgelegt wird, ob sie für andere Werte wiederverwendbar ist. Was ist jedoch mit dem Datentyp der Variablen?

Was passiert, wenn Sie eine Variable deklarieren

Für den Compiler ist es wichtig, welchen Datentyp eine Variable besitzt, damit er abwegige oder gefährliche Operationen verhindern kann, die zu Programmierfehlern (Bugs) führen können. Aus diesem Grund können Sie einer Int-Variablen beispielsweise nicht den String »Fisch« zuweisen. Der Compiler weiß, dass es zweckwidrig ist, mathematische Operationen an einem String durchzuführen.

Damit diese Art der Sicherheit funktioniert, muss der Compiler den Variablentyp kennen. Glücklicherweise kann er **den Datentyp einer Variablen aus dem zugewiesenen Wert ableiten**.

Und das funktioniert so:

Um eine Variable zu erzeugen, muss der Compiler ihren Namen und ihren Datentyp kennen und wissen, ob sie wiederverwendbar ist.

Der Wert wird in ein Objekt umgewandelt …

Wenn Sie eine Variable mit Code wie diesem deklarieren …

```
var x = 5
```

… wird der zugewiesene Wert verwendet, um ein neues Objekt zu erzeugen. In diesem Beispiel wird der Wert `5` einer neuen Variablen namens `x` zugewiesen. Der Compiler weiß, dass `5` ein ganzzahliger Wert (Int) ist. Also erzeugt der Compiler ein neues `Int`-Objekt mit dem Wert `5`:

Ein paar Seiten weiter sehen wir uns verschiedene Datentypen genauer an.

… und der Compiler leitet den Datentyp der Variablen von diesem Objekt ab

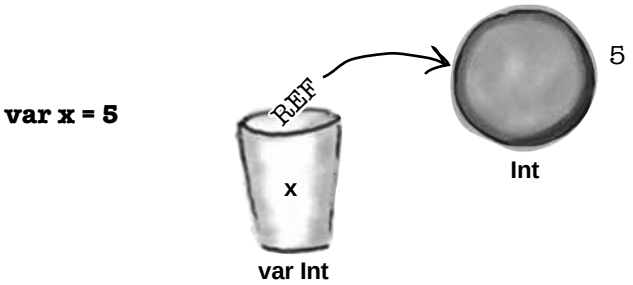
Danach benutzt der Compiler den Datentyp des Objekts, um den Datentyp der Variablen festzulegen. Im obigen Beispiel haben wir ein `Int`-Objekt, daher ist der Datentyp der Variablen ebenfalls `Int`. Dieser Typ kann während der Laufzeit nicht verändert werden.

Der Compiler weiß, dass Sie eine Variable vom Typ Int benötigen, damit sie zum Typ des Objekts passt.

Danach wird der Variablen das Objekt zugewiesen. Wie funktioniert das?

Die Variable enthält eine Referenz auf das Objekt

Wird einer Variablen ein Objekt zugewiesen, **enthält das Objekt nicht die Variable selbst.** Stattdessen wird in der Variablen eine *Referenz* auf das Objekt gespeichert.



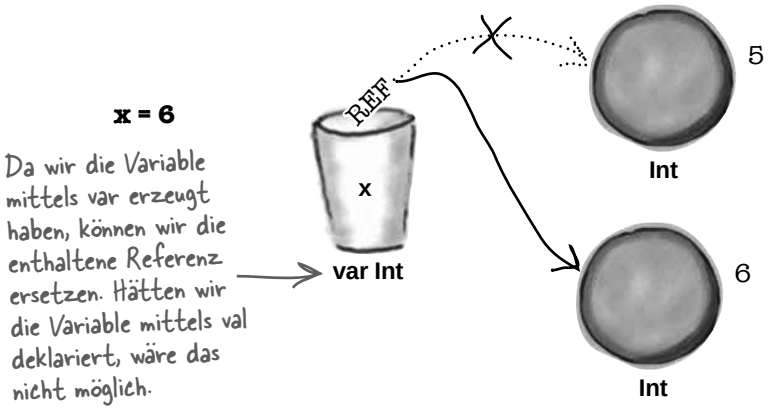
Da die Variable eine Referenz auf das Objekt enthält, können wir auf das Objekt zugreifen.

Ein zweiter Blick auf den Unterschied zwischen val und var

Wenn Sie die Variable mit `val` deklarieren, bleibt die Referenz während der Laufzeit in der Variablen und kann nicht ersetzt werden. Benutzen Sie dagegen das Schlüsselwort `var`, können Sie der Variablen einen anderen Wert zuweisen, zum Beispiel:

```
x = 6
```

Hier wird der Variablen `x` der Wert 6 zugewiesen. Dies erzeugt ein neues `Int`-Objekt mit dem Wert 6. Eine Referenz darauf wird in `x` gespeichert. Die ursprüngliche Referenz wird dabei überschrieben:

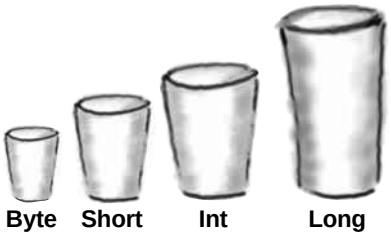


Nachdem Sie wissen, was bei der Deklaration einer Variablen passiert, wollen wir uns nun einige grundsätzliche Datentypen in Kotlin ansehen: ganzzahlige Werte (Integer), Fließkommazahlen (Floats), boolesche Werte, Zeichen (Char) und Zeichenketten (Strings).

Kotlins grundsätzliche Datentypen

Ganzzahlige Werte (Integer)

Kotlin besitzt vier Basisdatentypen: **`Byte`**, **`Short`**, **`Int`** und **`Long`**. Jeder Typ kann eine feste Anzahl von Bits enthalten. Der Typ **`Byte`** kann beispielsweise 8 Bits enthalten. Daher kann ein **`Byte`** ganzzahlige Werte zwischen –128 und 127 enthalten. `Int`-Werte können dagegen 32 Bits groß sein. Das entspricht Zahlen zwischen –2.147.483.648 und 2.147.483.647.



Das Standardverfahren für die Zuweisung eines Integerwerts an eine Variable sieht so aus:

```
var x = 1
```

Das erzeugt ein Objekt und eine Variable vom Typ `Int`. Ist der zugewiesene Wert zu lang, um in einen `Int` zu passen, wird der Datentyp `Long` verwendet. Alternativ können Sie ein `Long`-Objekt (und eine Variable) erzeugen, indem Sie dem Integer ein »L« nachstellen, wie hier gezeigt:

```
var hugeNumber = 6L
```

Diese Tabelle zeigt die verschiedenen Integertypen, ihre Bitgrößen und Wertebereiche:

Typ	Bits	Wertebereich
Byte	8 Bits	–128 bis 127
Short	16 Bits	–32768 bis 32767
Int	32 Bits	–2147483648 bis 2147483647
Long	64 Bits	–huge bis (huge – 1)

Hexadezimal und binäre Zahlen

- Um eine binäre Zahl zuzuweisen, stellen Sie ihr ein 0b voran:

```
x = 0b10
```

- Um eine hexadezimale Zahl zu erzeugen, stellen Sie der Zahl ein 0x voran:

```
y = 0xAB
```

- Oktale Zahlen werden nicht unterstützt.

Fließkommazahlen (Floats)

Für Fließkommazahlen gibt es zwei grundsätzliche Datentypen: **`Float`** und **`Double`**. `Float`s können 32 Bits enthalten, während `Double`s 64 Bits aufnehmen können.

Hier die Standardmethode, um einer Variablen einen Fließkommawert zuzuweisen:

```
var x = 123.5
```

Hierbei werden ein Objekt und eine Variable vom Typ `Double` erzeugt. Stellen Sie dem Wert ein »F« oder ein »f« nach, wird stattdessen ein `Float` erzeugt.

```
var x = 123.5F
```

Boolesche Werte

Für Werte, die entweder `true` (wahr) oder `false` (falsch) sein können, gibt es **`boolesche`** Variablen. Boolesche Objekte (und Variablen) können Sie mit Code wie diesem erzeugen:

```
var isBarking = true
var isTrained = false
```

Zeichen und Strings

Es gibt noch zwei weitere Basistypen: **`Char`** und **`String`.**

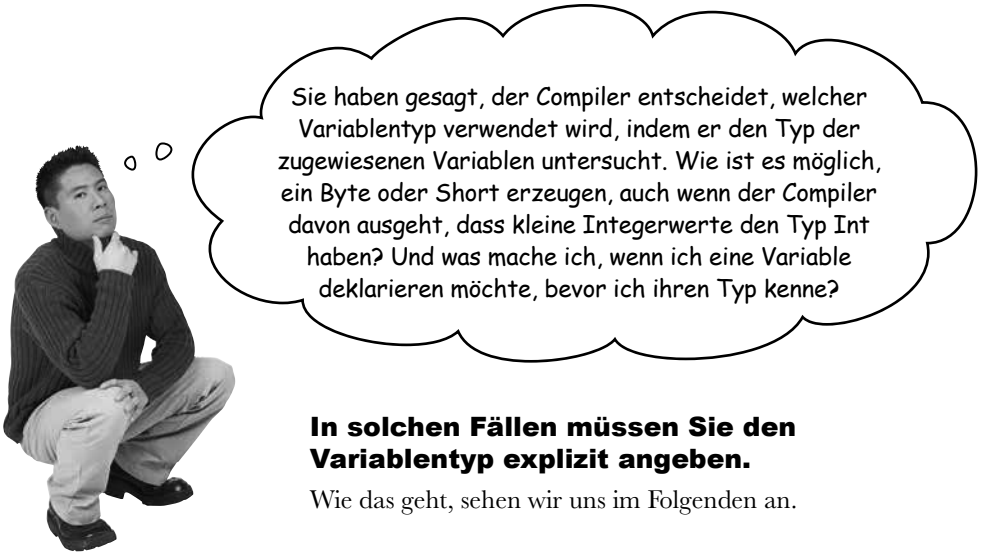
Variablen vom Typ `Char` werden für einzelne Zeichen benutzt. Eine `Char`-Variable wird erstellt, indem das Zeichen bei der Zuweisung mit einzelnen Anführungszeichen ('…') umgeben wird:

```
var letter = 'D'
```

Variablen vom Typ `String` werden verwendet, um zusammenhängende Zeichenketten zu speichern. Eine `String`-Variable wird erzeugt, indem die Zeichenkette bei der Zuweisung mit doppelten Anführungszeichen umgeben wird:

```
var name = "Fido"
```

Char-Variablen werden für einzelne Zeichen verwendet. String-Variablen für Zeichenketten.



In solchen Fällen müssen Sie den Variablentyp explizit angeben.

Wie das geht, sehen wir uns im Folgenden an.

Variablentypen explizit angeben

Inzwischen wissen Sie, wie eine Variable durch Zuweisung eines Werts erzeugt wird, wobei der Compiler den Datentyp aus dem Wert ableitet. Gelegentlich müssen Sie *dem Compiler den Datentyp einer Variablen auch explizit mitteilen*. Vielleicht wollen Sie `Byte` oder `Short` anstelle von `Int` verwenden, weil sie effizienter sind. Oder Sie möchten eine Variable am Anfang Ihres Codes deklarieren und den Wert erst später zuweisen.

Sie können den Datentyp einer Variablen folgendermaßen explizit angeben:

```
var smallNum: Short
```

Durch die explizite Angabe des Variablentyps erhält der Compiler genug Informationen für die Erzeugung der Variablen: ihren Namen, ihren Typ und ob sie wiederverwendet werden kann.

Anstatt sich darauf zu verlassen, dass der Compiler den Variablentyp von ihrem Wert ableitet, können Sie dem Variablennamen einen Doppelpunkt (`:`) nachstellen und den gewünschten Datentyp danach explizit angeben. Der oben stehende Code besagt also: »Erzeuge eine wiederverwendbare Variable namens *smallNum* und verwende *Short* als Datentyp.«

Hier ein entsprechendes Beispiel für das Deklarieren einer `Byte`-Variablen:

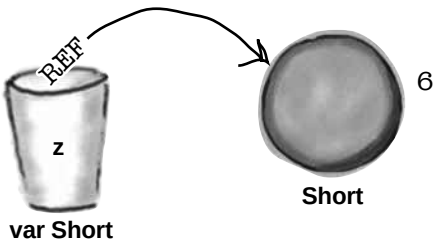
```
var tinyNum: Byte
```

Datentyp deklarieren UND einen Wert zuweisen

Die obigen Beispiele erzeugen Variablen, ohne ihnen Werte zuzuweisen. Sie können den Datentyp einer Variablen aber auch explizit festlegen *und* im gleichen Schritt einen Wert zuweisen. Im folgenden Beispiel erzeugen wir eine Variable vom Typ `Short` namens `z` und weisen ihr den Wert 6 zu:

```
var z: Short = 6
```

Dies erzeugt eine Variable namens `z` vom Typ `Short`. Ihr Wert 6 ist klein genug, um in einen `Short` zu passen. Also wird ein `Short`-Objekt mit dem Wert 6 erzeugt. Danach wird eine Referenz auf das `Short`-Objekt in der Variablen gespeichert.



Wenn Sie einer Variablen einen Wert zuweisen, müssen Sie sicherstellen, dass Wert und Variablentyp zusammenpassen. Auf der folgenden Seite sehen wir uns das noch einmal genauer an.

Die anfängliche Zuweisung eines Werts an eine Variable heißt Initialisierung. Eine Variable MUSS vor ihrer Verwendung initialisiert werden, ansonsten wird ein Compilerfehler ausgelöst. Der folgende Code wird nicht kompiliert, weil x kein Wert zugewiesen wurde:

```
var x: Int
var y = x + 6
```

x wurde kein Wert zugewiesen. Der Compiler ist »not amused«.

Den richtigen Wert für den Variablentyp verwenden

Wie in diesem Kapitel schon gesagt, ist dem Compiler der Variablentyp wirklich wichtig. Denn so kann er unzweckmäßige Operationen verhindern, die ansonsten zu Fehlern in Ihrem Code führen könnten. Versuchen Sie beispielsweise, einer Int-Variablen eine Fließkommazahl wie 3,12 zuzuweisen, wird der Compiler die Kompilierung Ihres Codes verweigern. Der folgende Code funktioniert also nicht:

```
var x: Int = 3.12
```

Der Compiler bemerkt, dass 3,12 nicht ohne Präzisionsverlust (alles nach dem Komma) in einem `Int` gespeichert werden kann. Also verweigert er stumpf die Kompilierung Ihres Codes.

Wenn Sie auf ähnliche Weise versuchen, eine große Ganzzahl (Large) in einer Variablen zu speichern, die dafür zu klein ist, bekommt der Compiler ebenfalls schlechte Laune. Versuchen Sie, den Wert 500 einer `Byte`-Variablen zuzuweisen, erhalten Sie einen Compilerfehler:

```
// Das funktioniert nicht:
var tinyNum: Byte = 500
```

Deshalb müssen Sie sicherstellen, dass der Wert mit dem Variablentyp kompatibel ist, um einer Variablen einen literalen Wert zuzuweisen. Das ist besonders wichtig, wenn der Wert einer Variablen einer anderen Variablen zugewiesen werden soll. Dies werden wir uns als Nächstes ansehen.

Der Kotlin-Compiler erlaubt die Zuweisung von Werten an Variablen nur, wenn beide kompatibel sind. Ist der Wert zu groß oder der Datentyp falsch, wird der Code nicht kompiliert.

Es gibt keine Dummen Fragen

F: In Java sind Zahlen primitive Datentypen. Eine Variable enthält also die tatsächliche Zahl. Ist das in Kotlin nicht auch so?

A: Nein. In Kotlin sind Zahlen Objekte, und die Variable enthält anstelle des Objekts eine Referenz darauf.

F: Warum ist Kotlin der Variablentyp so wichtig?

A: Weil damit Ihr Code sicherer und weniger fehleranfällig ist. Das klingt zwar etwas kleinkariert, aber glauben Sie uns: Das ist eine gute Sache!

F: In Java können Sie primitive Zeichenwerte als Zahlen behandeln. Funktioniert das auch mit `Char`-Werten in Kotlin?

A: Nein, `Char`-Werte in Kotlin sind Zeichen, keine Zahlen. Sprechen Sie uns nach: »Kotlin ist nicht Java!«

F: Kann ich meinen Variablen einen beliebigen Namen geben?

A: Nein. Die Regeln sind zwar recht flexibel, dennoch können Sie keine reservierten Wörter als Variablennamen verwenden. Der Versuch, eine Variable *while* zu nennen, bringt Sie mit Sicherheit in Schwierigkeiten. Es gibt aber auch gute Nachrichten: Versuchen Sie, einer Variablen einen nicht erlaubten Namen zu geben, wird IntelliJ IDEA das Problem sofort erkennen und Sie darauf hinweisen.

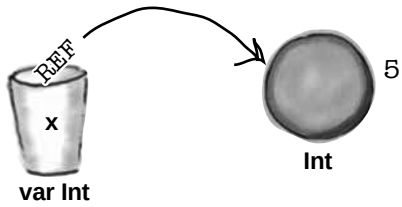
Einen Wert einer anderen Variablen zuweisen

Wenn Sie den Wert einer Variablen einer anderen Variablen zuweisen, müssen Sie dafür sorgen, dass die Datentypen kompatibel sind. Hierzu ein Beispiel:

```
var x = 5
var y = x
var z: Long = x
```

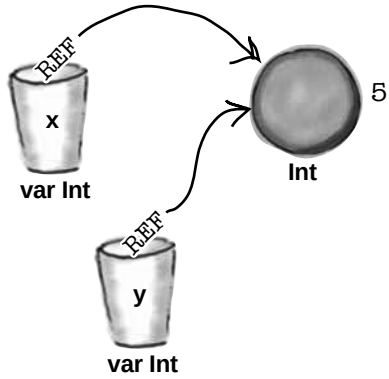
1 **var x = 5**

Dies erzeugt eine `Int`-Variable namens `x` sowie ein `Int`-Objekt mit dem Wert 5. `x` enthält eine Referenz auf das Objekt.



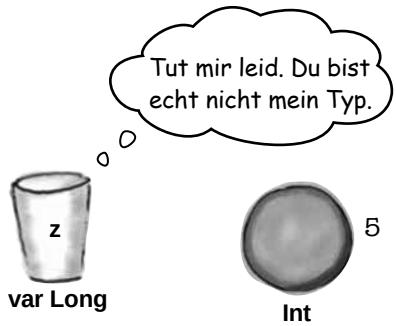
2 **var y = x**

Der Compiler sieht, dass `x` ein `Int`-Objekt ist. Er weiß also, dass `y` ebenfalls ein `Int` sein muss. Anstatt ein zweites `Int`-Objekt zu erzeugen, wird der Wert der Variablen `x` an `y` zugewiesen. Man könnte also auch sagen: »Nimm die Bits in `x`, erstelle eine Kopie und speichere die Kopie in `y`.« **Das heißt, dass `x` und `y` Referenzen auf das gleiche Objekt enthalten**.



3 **var z: Long = x**

Der Compiler sieht, dass Sie eine neue Variable vom Typ `Long` mit dem Namen `z` erzeugen wollen, und weist ihr den Wert von `x` zu. Dabei gibt es allerdings ein Problem. Die Variable `x` enthält eine Referenz auf ein `Int`-Objekt mit dem Wert 5, aber kein `Long`-Objekt. Wir wissen, dass das Objekt den Wert 5 hat und das 5 in ein `Long`-Objekt passt. Aber weil die Variable `z` einen anderen Typ hat als das `Long`-Objekt, schmollt der Compiler und weigert sich, den Code zu kompilieren.



Wie kann man einen Variablenwert einer Variablen eines anderen Typs zuweisen?

Wir müssen den Wert konvertieren

Angenommen, wir wollten den Wert einer Variablen von `Int` nach `Long` umwandeln. Der Compiler erlaubt keine direkte Zuweisung, weil die Variablen unterschiedliche Typen haben: Eine `Long`-Variable kann nur auf ein `Long`-Objekt verweisen. Versuchen Sie, ein `Int`-Objekt zuzuweisen, wird der Code nicht kompiliert.

Um das zu ermöglichen, müssen Sie den Wert zunächst in den korrekten Typ konvertieren. Soll der Wert einer `Int`-Variablen einer Variablen vom Typ `Long` zugewiesen werden, müssen Sie den Wert also zunächst in einen `Long`-Wert umwandeln. Das geht anhand der *Funktionen* des `Int`-Objekts.

Ein Objekt besitzt Zustand und Verhalten

Als Objekt hat man zwei Dinge: **Zustand** und **Verhalten**.

Der *Zustand* eines Objekts bezieht sich auf die Daten, die mit dem Objekt verbunden sind: seine Werte und Eigenschaften. Ein numerisches Objekt kann zum Beispiel Werte wie 5, 42 oder 3,12 besitzen – je nach Typ. Der Wert eines `Char`-Objekts ist dagegen ein einzelnes Zeichen. Ein Objekt vom Typ `Boolean` ist entweder `true` (wahr) oder `false` (falsch).

Das *Verhalten* eines Objekts beschreibt die Dinge, die es tun kann bzw. die mit ihm getan werden können. Ein `String` kann beispielsweise in Großbuchstaben verwandelt werden. Numerische Objekte »wissen«, wie einfache mathematische Operationen an ihnen ausgeführt werden können und wie man ihre Werte in Objekte eines anderen numerischen Typs konvertiert. Das Verhalten eines Objekts ist über seine Funktionen zugänglich.

Typumwandlung für numerische Werte

In unserem Beispiel wollen wir den Wert einer `Int`-Variablen nach `Long` umwandeln. Jedes numerische Objekt besitzt eine Funktion namens `toLong()`. Sie übernimmt den Wert des Objekts und erzeugt daraus ein neues `Long`-Objekt. Das geht beispielsweise mit Code wie diesem :

```
var x = 5
var z: Long = x.toLong()
```

Dies ist der Punktoperator.

Mithilfe des Punktoperators (`.`) können Sie die Funktionen eines Objekts aufrufen. Der Aufruf `x.toLong()` bedeutet also: »Gehe zu dem Objekt, das in der Variablen `x` referenziert wird, und rufe dessen `toLong()`-Funktion auf.«

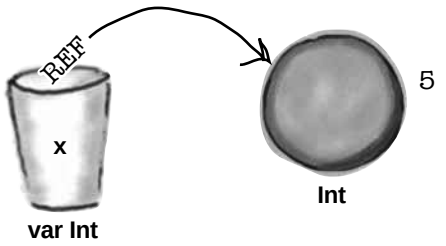
Auf den folgenden Seiten gehen wir genauer darauf ein, was dieser Code macht.

Jeder numerische Datentyp besitzt die folgenden Konvertierungsfunktionen: toByte(), toShort(), toInt(), toLong(), toFloat() und toDouble().

Was passiert, wenn Sie einen Wert konvertieren?

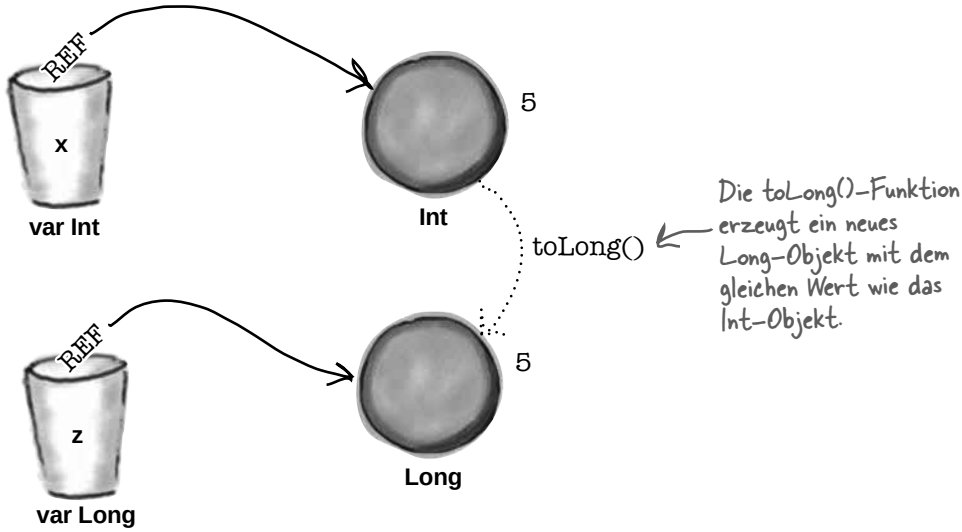
1 **`var x = 5`**

Das erzeugt eine `Int`-Variable namens `x` und ein `Int`-Objekt mit dem Wert 5. `x` enthält eine Referenz auf das Objekt.



2 **`var z: Long = x.toLong()`**

Dieser Code erzeugt eine `Long`-Variable namens `z`. Am Objekt von `x` wird die `toLong()`-Funktion aufgerufen. Das erzeugt ein neues `Long`-Objekt mit dem Wert 5. Eine Referenz auf das `Long`-Objekt wird in der Variablen `z` gespeichert.



Das funktioniert gut, wenn Sie einen Wert in ein größeres Objekt konvertieren wollen. Was passiert aber, wenn das neue Objekt zu klein ist, um den Wert zu enthalten?

Aufpassen, dass nichts überläuft

Wenn Sie versuchen, einen großen Wert in einer kleinen Variablen zu speichern, ist das, als versuchten Sie, einen ganzen Eimer Kaffee in eine kleine Tasse zu zwängen. Ein Teil des Kaffees passt in die Tasse, der Rest läuft über.

Angenommen, Sie wollten den Wert aus einem `Long`-Objekt in einem `Int` speichern. Wie wir bereits wissen, kann ein `Long`-Objekt größere Zahlen enthalten als ein `Int`.

Befindet sich der Wert des `Long` innerhalb des Wertebereichs für `Int`-Objekte, bleibt der Wert nach der Konvertierung von `Long` nach `Int` unverändert 42.

```
var x = 42L
var y: Int = x.toInt()  // Wert ist 42
```

Ist der `Long`-Wert dagegen zu groß für ein `Int`-Objekt, schneidet der Compiler den Wert ab, und Sie erhalten eine recht seltsame (aber berechenbare) Zahl. Wenn Sie beispielsweise versuchen, ein `Long`-Objekt mit dem Wert 1234567890123 in ein `Int`-Objekt zu konvertieren, wird der `Int`-Wert 1912276171 lauten:

Das hat mit Vorzeichen, Bits, Binärzahlen und anderem geekigen Zeug zu tun, auf das wir hier nicht weiter eingehen. Wenn Sie richtig neugierig sind, suchen Sie nach dem Begriff >>Zweierkomplement<<.

```
var x = 1234567890123
var y: Int = x.toInt()  // Wert ist 1912276171!
```

Der Compiler geht davon aus, dass Sie das absichtlich tun, und der Code wird kompiliert. Vielleicht haben Sie aber auch eine Fließkommazahl und brauchen nur den ganzzahligen Teil. Bei der Konvertierung der Zahl in ein `Int`-Objekt wird der Compiler alles nach dem Komma abschneiden und verwerfen:

```
var x = 123.456
var y: Int = x.toInt()  // Wert ist 123
```

Obacht: Wie in fast allen Programmiersprachen wird das Komma auch in Kotlin als Dezimalpunkt (.) ausgedrückt.

Der wichtigste Punkt ist also: Wenn Sie numerische Werte von einem Typ in einen anderen konvertieren, müssen Sie dafür sorgen, dass der Typ groß genug für den Wert ist, ansonsten erhalten Sie unerwartete Ergebnisse.

Nachdem Sie sich einigermaßen mit der Funktionsweise von Variablen und Kotlins Basisdatentypen auskennen, ist es Zeit für die folgende Übung.

Spitzen Sie Ihren Bleistift

Die folgende `main`-Funktion kompiliert nicht. Kreisen Sie die ungültigen Zeilen ein und notieren Sie, warum sie die Kompilierung verhindern.

```
fun main(args: Array<String>) {

    var x: Int = 65.2

    var isPunk = true

    var message = 'Hello'

    var y = 7

    var z: Int = y

    y = y + 50

    var s: Short

    var bigNum: Long = y.toLong()

    var b: Byte = 2

    var smallNum = b.toShort()

    b = smallNum

    isPunk = "false"

    var k = y.toDouble()

    b = k.toByte()

    s = 0b10001

}
```

Spitzen Sie Ihren Bleistift
Lösung

Die folgende `main`-Funktion kompiliert nicht. Kreisen Sie die ungültigen Zeilen ein und notieren Sie, warum sie die Kompilierung verhindern.

```
fun main(args: Array<String>) {

    var x: Int = 65.2

    var isPunk = true

    var message = 'Hello'

    var y = 7

    var z: Int = y

    y = y + 50

    var s: Short

    var bigNum: Long = y.toLong()

    var b: Byte = 2

    var smallNum = b.toShort()

    b = smallNum

    isPunk = "false"

    var k = y.toDouble()

    b = k.toByte()

    s = 0b10001

}
```

65.2 ist kein gültiger Int-Wert.

Einfache Anführungszeichen werden für die Definition von Char-Werten (einzelnen Zeichen) verwendet.

smallNum ist ein Short-Objekt. Daher kann sein Wert nicht einer Byte-Variablen zugewiesen werden.

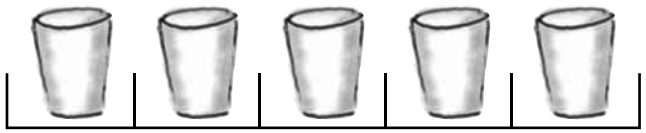
isPunk ist eine boolesche Variable. Daher wird ihr Wert (false) nicht mit doppelten Anführungszeichen umgeben.

Mehrere Werte in einem Array speichern

Wir möchten Ihnen jetzt eine weitere Objektart vorstellen: das Array. Angenommen, Sie wollten die Namen von 50 Eiscremesorten speichern oder auch die Balkencodes aller Bücher einer Bücherei. Hier wird der Einsatz von Variablen ziemlich schnell unhandlich. Stattdessen können Sie ein Array verwenden.

Arrays sind großartig, wenn schnell mal ein paar Dinge gruppiert werden sollen. Sie sind leicht zu erstellen, und Sie können schnell auf die einzelnen Elemente eines Arrays zugreifen.

Stellen Sie sich ein Array wie ein Tablett mit Bechern vor, bei dem jedes Arrayelement des Arrays einer Referenz auf ein Objekt entspricht.

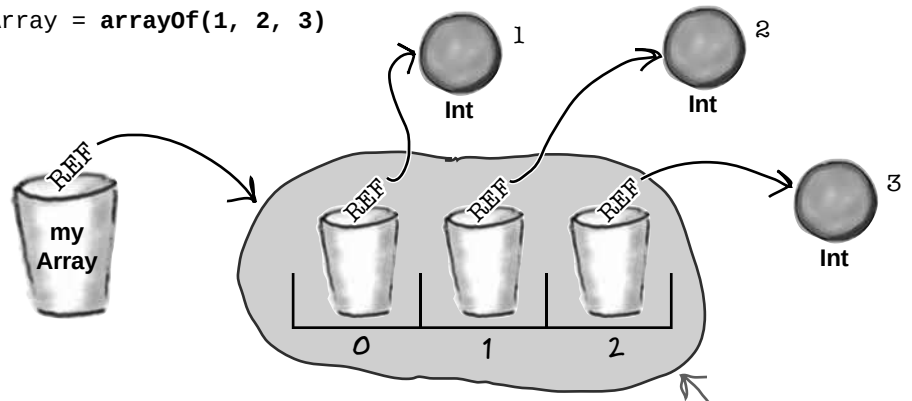


Stellen Sie sich ein Array als Tablett mit Bechern vor.

Wie man ein Array erstellt

Mit der Funktion `arrayOf()` können Sie ein Array erstellen. Hier ein Beispiel, in dem wir die Funktion verwenden, um ein Array mit drei Elementen (den Integerwerten 1, 2 und 3) zu erstellen und einer Variablen namens `myArray` zuzuweisen:

```
var myArray = arrayOf(1, 2, 3)
```



Auch das Array befindet sich in einem Objekt, und die Variable enthält eine Referenz darauf.

Um den Wert eines Arrayelements zu erhalten, verwenden Sie die Arrayvariable mit einem **Index**. Das erste Element können Sie beispielsweise so ausgeben:

```
println(myArray[0])
```

Und die Länge des Arrays können Sie so ermitteln:

```
myArray.size
```

Auf der folgenden Seite wollen wir das Wissen anwenden, um ein ernsthaftes Businessprogramm zu schreiben – den Phras-O-Matic.

Die Phras-O-Matic-Applikation erstellen

Wir bauen eine neue Applikation, die nützliche Marketingsprüche für den englischsprachigen Markt erzeugt.

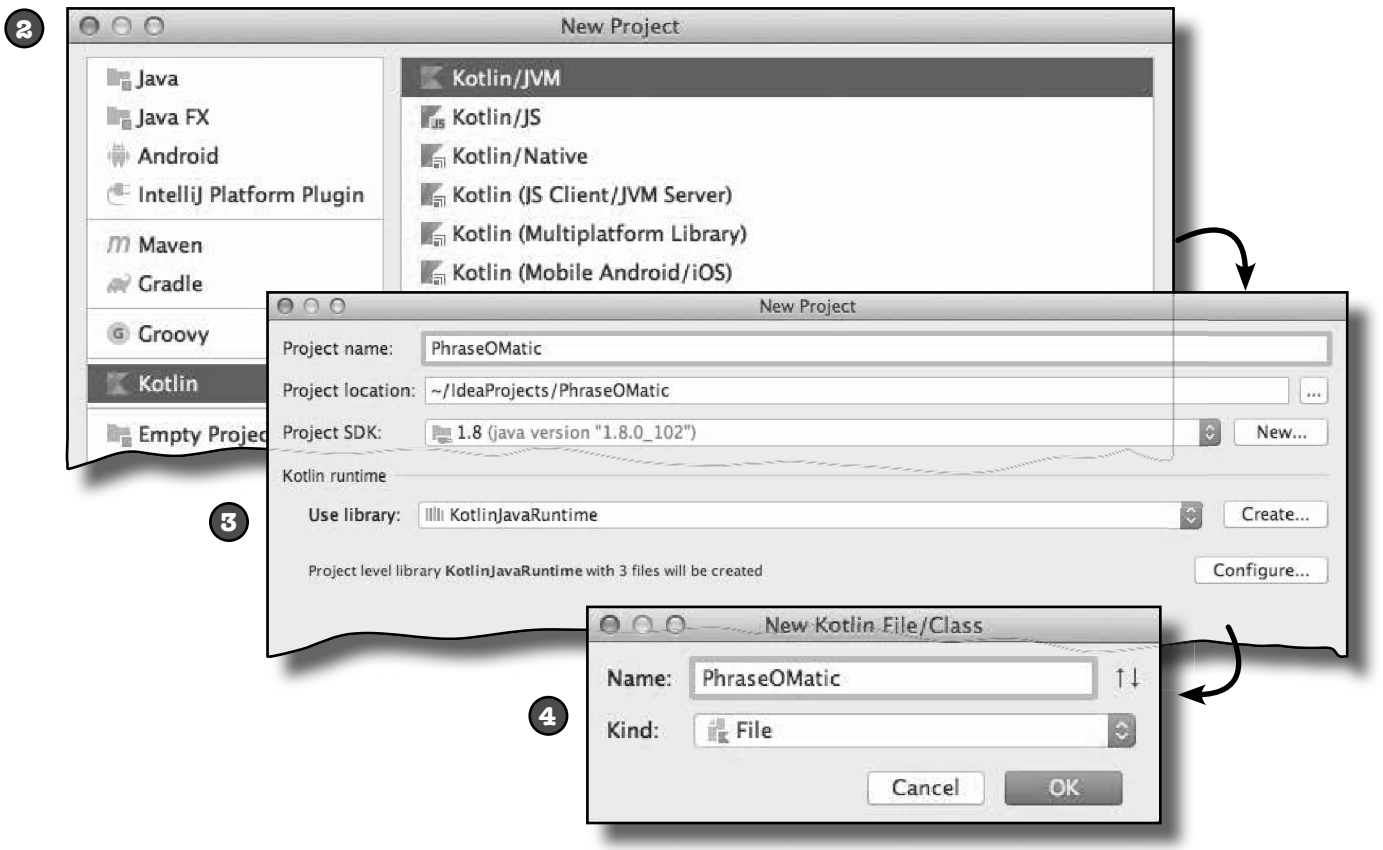
Hierfür müssen wir zuerst ein neues Projekt in IntelliJ IDEA erstellen:

1. Öffnen Sie IntelliJ IDEA und wählen Sie im Startfenster die Option »Create New Project«. Dies startet das aus Kapitel 1 bekannte Helferprogramm (»Wizard«).

2. Wählen Sie die Option zur Erstellung eines Kotlin-Projekts für die JVM.

3. Nennen Sie das Projekt »PhrasOMatic«, akzeptieren Sie die übrigen Standardwerte und klicken Sie auf den Finish-Button.

4. Wenn Ihr Projekt in der IDE angezeigt wird, erstellen Sie eine neue Kotlin-Datei namens *PhrasOMatic.kt*. Markieren Sie hierfür den *src*-Ordner und wählen Sie dann den Befehl New → Kotlin File/Class aus dem File-Menü. Geben Sie der Datei den Namen »PhrasOMatic« und wählen Sie in der zweiten Zeile (Kind) die Option File.



Den Code zu PhrasOMatic.kt hinzufügen

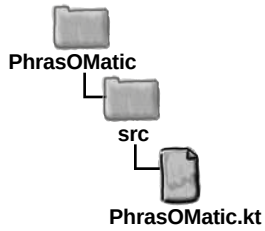
Der Code für unser Phras-O-Matic-Programm besteht aus einer `main`-Funktion, die drei Arrays mit Wörtern anlegt, zufällig ein Wort aus jedem Array auswählt und diese miteinander verbindet. Fügen Sie den unten stehenden Code *PhrasOMatic.kt*: hinzu:

```
fun main(args: Array<String>){
    val wordArray1 = arrayOf("24/7", "multi-tier", "B-to-B", "dynamic", "pervasive")
    val wordArray2 = arrayOf("empowered", "leveraged", "aligned", "targeted")
    val wordArray3 = arrayOf("process", "paradigm", "solution", "portal", "vision")

    val arraySize1 = wordArray1.size
    val arraySize2 = wordArray2.size
    val arraySize3 = wordArray3.size

    val rand1 = (Math.random() * arraySize1).toInt()
    val rand2 = (Math.random() * arraySize2).toInt()
    val rand3 = (Math.random() * arraySize3).toInt()

    val phrase = "${wordArray1[rand1]} ${wordArray2[rand2]} ${wordArray3[rand3]}"
    println(phrase)
}
```



Die meisten Teile des Codes sollten Sie bereits kennen. Einige Zeilen wollen wir uns dennoch etwas genauer ansehen. Die folgende Zeile erzeugt eine Zufallszahl:

```
val rand1 = (Math.random() * arraySize1).toInt()
```

Daher müssen wir sie mit der Anzahl der Elemente im Array multiplizieren. Danach benutzen wir die Funktion `toInt()`, um ein ganzzahliges Ergebnis zu erhalten.

Die folgende Zeile verwendet ein String-Template, um drei Wörter auszuwählen und zu verbinden:

```
val phrase = "${wordArray1[rand1]} ${wordArray2[rand2]} ${wordArray3[rand3]}"
```

Auf den folgenden Seiten finden Sie noch mehr Informationen zu String-Templates und weitere Dinge, die mit Arrays möglich sind.

Wir brauchen …

- multi-tier leveraged solution
- dynamic targeted vision
- 24/7 aligned paradigm
- B-to-B empowered portal

String-Templates unter der Lupe

String-Templates bieten eine einfache und schnelle Möglichkeit, innerhalb eines Strings auf eine Variable zuzugreifen.

Um den Wert einer Variablen in einem String zu verwenden, stellen Sie dem Variablennamen ein Dollarzeichen ($) voran. Um den Wert einer `Int`-Variablen namens `x` in einem String zu verwenden, können Sie beispielsweise schreiben:

```
var x = 42
var value = "Value of x is $x"
```

In String-Templates kann außerdem auf die Eigenschaften und Funktionen eines Objekts zugegriffen werden. Dafür wird der Ausdruck mit geschweiften Klammern umgeben. In den folgenden Zeilen verwenden wir die Größe eines Arrays und den Wert des ersten Arrayelements in einem String:

```
var myArray = arrayOf(1, 2, 3)
var arraySize = "myArray has ${myArray.size} items"
var firstItem = "The first item is ${myArray[0]}"
```

String-Templates können auch komplexe Ausdrücke enthalten. Hier nutzen wir einen `if`-Ausdruck, um abhängig von der Größe von `myArray` unterschiedlichen Text in der Variablen `result` zu speichern:

Beachten Sie, dass der auszuwertende Ausdruck mit geschweiften Klammern ({}) umgeben ist.

```
var result = "myArray is ${if (myArray.size > 10) "large" else "small"}"
```

Durch die Verwendung von String-Templates können Sie schon mit sehr wenig Code komplexe Strings erzeugen.

Es gibt keine Dummen Fragen

F: Ist `Math.random()` die Standardmethode zum Erzeugen einer Zufallszahl in Kotlin?

A: Das kommt darauf an, welche Version von Kotlin Sie verwenden.

Vor Version 1.3 hatte Kotlin keine eigene Methode, um Zufallszahlen zu erzeugen. Applikationen, die in einer JVM laufen, können allerdings die `random()`-Methode aus der Java-`Math`-Bibliothek verwenden, wie in unserem Beispiel.

Benutzen Sie dagegen Version 1.3 oder höher, können Sie Kotlins eigene `Random`-Funktionen einsetzen. Der folgende Code verwendet die Funktion `nextInt()` aus Kotlins `Random`-Klasse, um ein zufälliges `Int`-Objekt zu erzeugen:

```
kotlin.random.Random.nextInt()
```

In diesem Buch benutzen wir trotzdem `Math.random()` für die Erzeugung von Zufallszahlen, weil dies für alle Kotlin-Versionen funktioniert, die in einer JVM laufen.

Der Compiler leitet den Arraytyp aus dessen Werten ab

Nachdem Sie wissen, wie man ein Array erstellt und auf dessen Elemente zugreift, wollen wir nun sehen, wie man die enthaltenen Werte verändern kann.

Angenommen, wir hätten ein Array mit `Int`-Werten namens `myArray`:

```
var myArray = arrayOf(1, 2, 3)
```

Der folgende Code ändert den Wert des zweiten Elements auf 15:

```
myArray[1] = 15
```

Die Sache hat allerdings einen Haken: **Der Wert muss den richtigen Typ haben.**

Der Compiler untersucht den Typ jedes Arrayelements und leitet daraus ab, welche Datentypen die einzelnen Elemente während der Laufzeit des Programms haben sollen. Im obigen Beispiel haben wir ein Array mit `Int`-Werten deklariert. Der Compiler geht deshalb davon aus, dass das Array nur `Int`-Werte enthalten kann. Versuchen Sie, etwas anderes im Array zu speichern, wird der Code nicht kompilieren:

```
myArray[1] = "Fido"  // Das wird nicht kompiliert
```

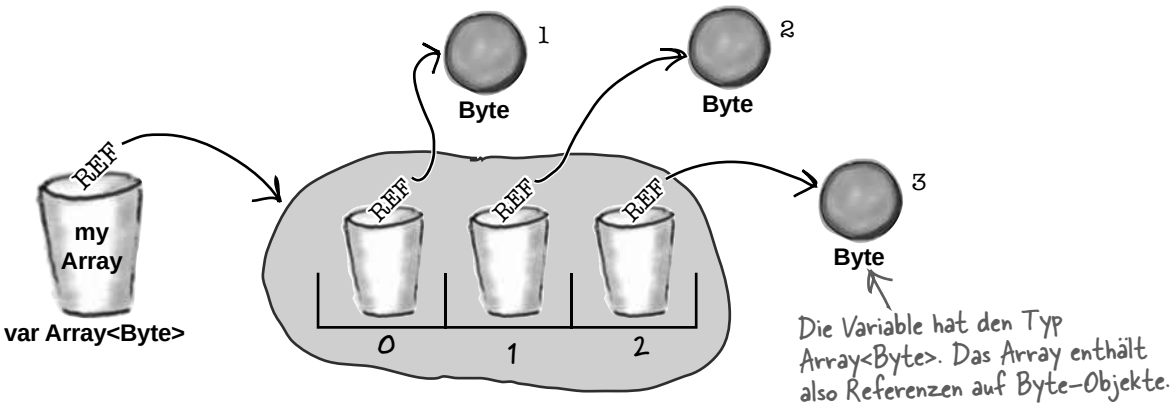
Den Arraytyp explizit angeben

Wie bei anderen Variablen können Sie auch für Arrays explizit angeben, welche Datentypen die Arrayelemente enthalten sollen. Angenommen, Sie definierten ein Array, das `Byte`-Werte enthalten soll, wie hier gezeigt:

```
var myArray: Array<Byte> = arrayOf(1, 2, 3)
```

Der Code `Array<Byte>` teilt dem Compiler mit, dass Sie ein Array mit `Byte`-Werten erstellen. Um den Arraytyp explizit anzugeben, umgeben Sie den gewünschten Typ mit spitzen Klammern (`<>`).

Arrays enthalten Elemente eines bestimmten Typs. Entweder lassen Sie den Compiler den Typ aus den Arraywerten ermitteln, oder Sie geben ihn explizit an, indem Sie die Schreibweise Array<Typ> verwenden.



var heißt, die Variable kann auf ein anderes Array verweisen

Zum Abschluss müssen wir uns noch ansehen, welche Auswirkungen `val` und `var` auf die Deklaration von Arrays haben.

Wie Sie bereits wissen, enthält eine Variable eine Referenz auf ein Objekt. Wurde die Variable per `var` deklariert, können Sie die Variable aktualisieren, sodass sie auf ein anderes Objekt verweist. Enthält die Variable eine Referenz auf ein Array, können Sie die Variable aktualisieren, sodass sie auf ein anderes Array gleichen Typs verweist. Der folgende Code ist beispielsweise gültig und wird kompiliert:

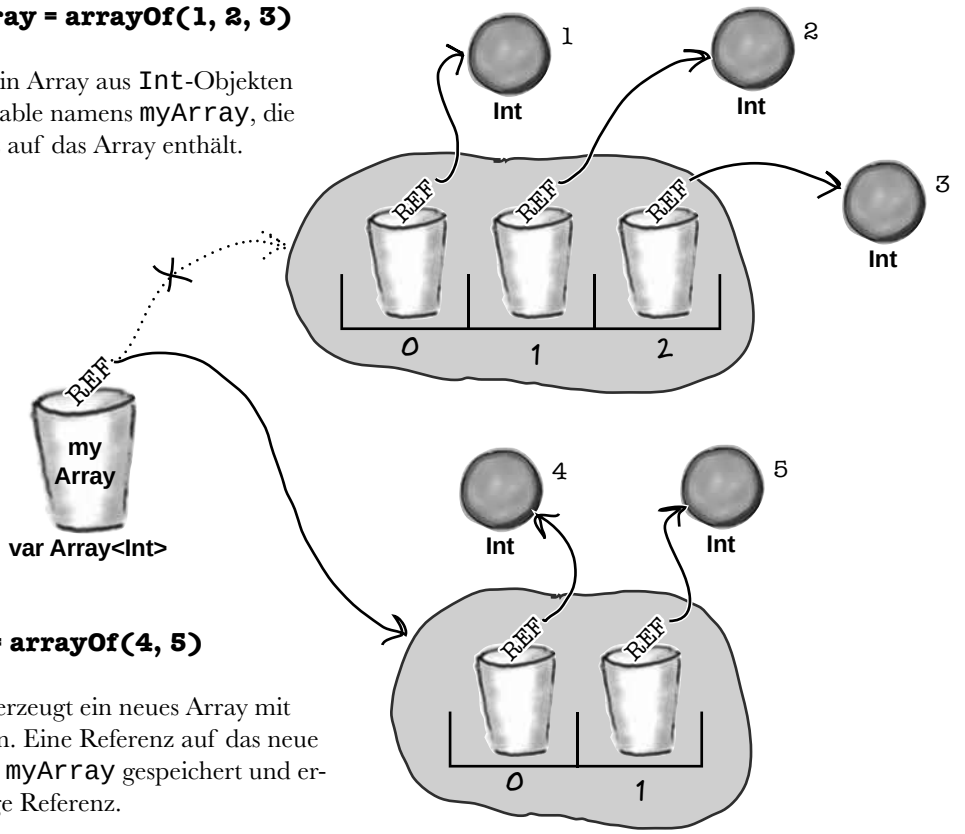
```
var myArray = arrayOf(1, 2, 3)
myArray = arrayOf(4, 5)
```

← Ein brandneues Array.

Das gehen wir am besten Schritt für Schritt durch.

1 **`var myArray = arrayOf(1, 2, 3)`**

Das erzeugt ein Array aus `Int`-Objekten und eine Variable namens `myArray`, die eine Referenz auf das Array enthält.



2 **`myArray = arrayOf(4, 5)`**

Dieser Code erzeugt ein neues Array mit `Int`-Objekten. Eine Referenz auf das neue Array wird in `myArray` gespeichert und ersetzt die vorige Referenz.

Was passiert, wenn wir stattdessen `val` verwenden?

val bedeutet, die Variable verweist während der gesamten Laufzeit auf dasselbe Array ...

Wenn Sie ein Array per `val` deklarieren, kann die Variable nicht mehr aktualisiert werden, sodass sie eine Referenz auf ein anderes Array enthält. Der folgende Code wird nicht kompiliert:

```
val myArray = arrayOf(1, 2, 3)
myArray = arrayOf(4, 5, 6)
```

Wenn Sie eine Arrayvariable mittels val deklarieren, kann sie nicht auf ein anderes Array verweisen.

Sobald der Variablen ein bestimmtes Array zugewiesen wurde, verweist sie während der gesamten Laufzeit des Programms auf dieses Array. Trotzdem kann **das Array selbst** auch weiterhin aktualisiert werden.

... die Elemente im Array können trotzdem aktualisiert werden

Verwenden Sie `val`, um eine Variable zu deklarieren, sagen Sie dem Compiler, dass die Variable nicht für andere Werte zur Verfügung steht. Diese Anweisung bezieht sich aber nur auf die Variable selbst. Enthält die Variable eine Referenz auf ein Array, können die enthaltenen Elemente trotzdem verändert werden.

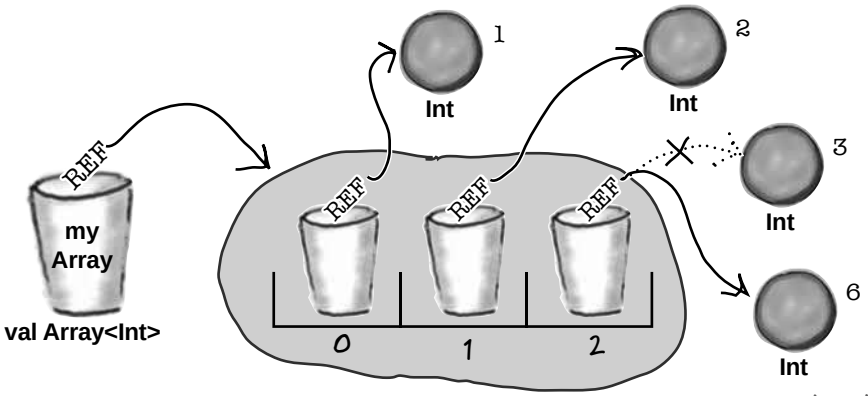
Nehmen wir beispielsweise den folgenden Code:

```
val myArray = arrayOf(1, 2, 3)
myArray[2] = 6
```

Hier wird das dritte Arrayelement aktualisiert.

Hier erzeugen wir eine Variable namens `myArray`. Sie enthält eine Referenz auf ein Array, dessen Elemente auf mehrere `Int`-Objekte verweisen. Dieses wurde per `val` deklariert. Die Variable muss also für die Laufzeit des Programms eine Referenz auf dasselbe Array enthalten. Danach wird das dritte Arrayelement erfolgreich mit dem Wert 6 aktualisiert, denn das Array selbst kann weiterhin verändert werden.

Die Deklaration einer Variablen per val bedeutet, dass Sie der Variablen kein anderes Objekt zuweisen können. Das referenzierte Objekt selbst kann aber trotzdem verändert werden.



Das Array selbst kann weiterhin verändert werden, auch wenn die Variable per val deklariert wurde.

Nachdem Sie wissen, wie Arrays hier in Kotlinville funktionieren, ist es Zeit für ein paar Übungen.

SEIEN Sie der Compiler

Die Codeabschnitte auf dieser Seite stehen jeweils für eine vollständige Kotlin-Quelldatei. Spielen Sie Compiler und finden Sie heraus, welche der Dateien kompiliert und fehlerfrei ausgeführt werden und welche nicht. Wie würden Sie die Fehler beheben?

A

```
fun main(args: Array<String>) {

    val hobbits = arrayOf("Frodo", "Sam", "Merry", "Pippin")
    var x = 0;

    while (x < 5) {
        println("${hobbits[x]} is a good Hobbit name")
        x = x + 1
    }

}
```

Jeder Hobbitname im Array soll auf einer eigenen Zeile ausgegeben werden.

B

```
fun main(args: Array<String>) {

    val firemen = arrayOf("Pugh", "Pugh", "Barney McGrew", "Cuthbert", "Dibble", "Grub")
    var firemanNo = 0;

    while (firemanNo < 6) {
        println("Fireman number $firemanNo is $firemen[firemanNo]")
        firemanNo = firemanNo + 1
    }

}
```

Die Namen der Feuerwehrleute im Array sollen jeweils auf einer eigenen Zeile ausgegeben werden.

→ Die Antworten finden Sie auf Seite 55.

Code-Magnete

Die Kühlschrankmagnete für ein funktionsfähiges Kotlin-Programm sind durcheinandergekommen. Können Sie die Codeabschnitte wieder in die richtige Reihenfolge bringen? Wir brauchen eine Funktion, die folgende Ausgabe erzeugt:

```
Fruit = Banana
Fruit = Blueberry
Fruit = Pomegranate
Fruit = Cherry
```

```
fun main(args: Array<String>) {
```

Hier kommen die Magnete hin.

```
}
```

```
x = x + 1
```

```
y = index[x]
```

```
var x = 0
```

```
while (x < 4) {
```

```
var y: Int
```

```
val index = arrayOf(1, 3, 4, 2)
```

```
}
```

```
println("Fruit = ${fruit[y]}")
```

```
val fruit = arrayOf("Apple", "Banana", "Cherry", "Blueberry", "Pomegranate")
```

→ Die Antworten finden Sie auf Seite 56.

Vermischte Referenzen

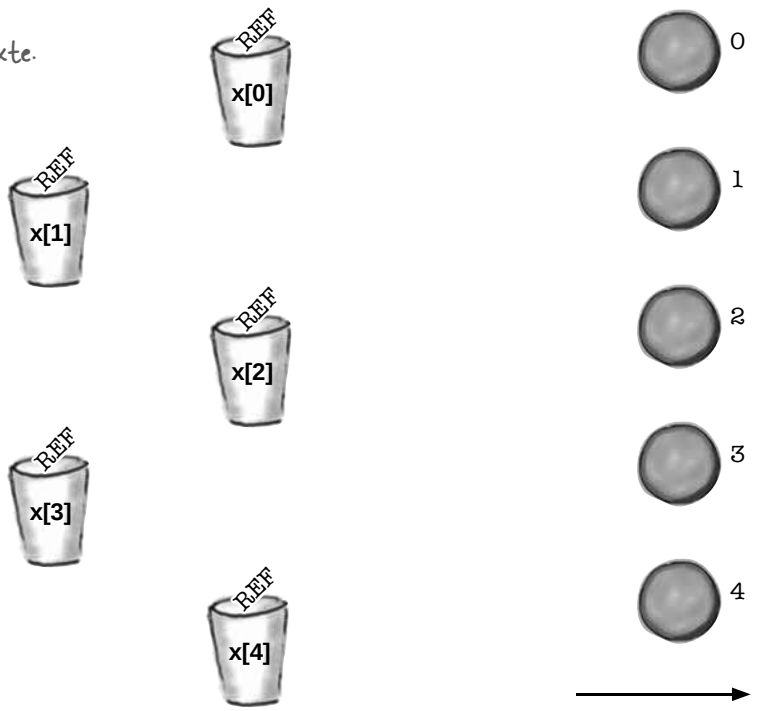
Unten sehen Sie ein kurzes Kotlin-Programm. Bei Erreichen der Zeile `// Mach was` sind bereits mehrere Variablen und Objekte erzeugt worden. Sie sollen herausfinden, welche Variablen auf welche Objekte verweisen, wenn `// Mach was` erreicht ist. Auf einige Objekte wird möglicherweise mehr als einmal verwiesen. Zeichnen Sie Linien, um die Variablen und Objekte miteinander zu verbinden.

```
fun main(args: Array<String>) {
    val x = arrayOf(0, 1, 2, 3, 4)
    x[3] = x[2]
    x[4] = x[0]
    x[2] = x[1]
    x[1] = x[0]
    x[0] = x[1]
    x[4] = x[3]
    x[3] = x[2]
    x[2] = x[4]
    // Mach was
}
```

Variablen:

Objekte:

Verbinden Sie die Variablen und Objekte.



Antworten auf Seite 57.

SEIEN Sie der Compiler, Lösung

Die Codeabschnitte auf dieser Seite stehen jeweils für eine vollständige Kotlin-Quelldatei. Spielen Sie Compiler und finden Sie heraus, welche der Dateien kompiliert und fehlerfrei ausgeführt werden und welche nicht. Wie würden Sie die Fehler beheben?

A

```
fun main(args: Array<String>) {

    val hobbits = arrayOf("Frodo", "Sam", "Merry", "Pippin")
    var x = 0;
    while (x < 5) {
        println("${hobbits[x]} is a good Hobbit name")
        x = x + 1
    }

}
```

Der Code kompiliert, erzeugt aber einen Laufzeitfehler. Vergessen Sie nicht, dass die Nummerierung von Array-elementen bei 0 beginnt und bei (Länge – 1) endet.

B

```
fun main(args: Array<String>) {

    val firemen = arrayOf("Pugh", "Pugh", "Barney McGrew", "Cuthbert", "Dibble", "Grub")
    var firemanNo = 0;

    while (firemanNo < 6) {
        println("Fireman number $firemanNo is ${firemen[firemanNo]}")
        firemanNo = firemanNo + 1
    }

}
```

Der Ausdruck firemen[firemanNo] muss von geschweiften Klammern umgeben werden, damit die Namen der Feuerwehrleute ausgegeben werden können.

Code-Magnete, Lösung

Die Kühlschrankmagnete für ein funktionsfähiges Kotlin-Programm sind durcheinandergekommen. Können Sie die Codeabschnitte wieder in die richtige Reihenfolge bringen? Wir brauchen eine Funktion, die folgende Ausgabe erzeugt:

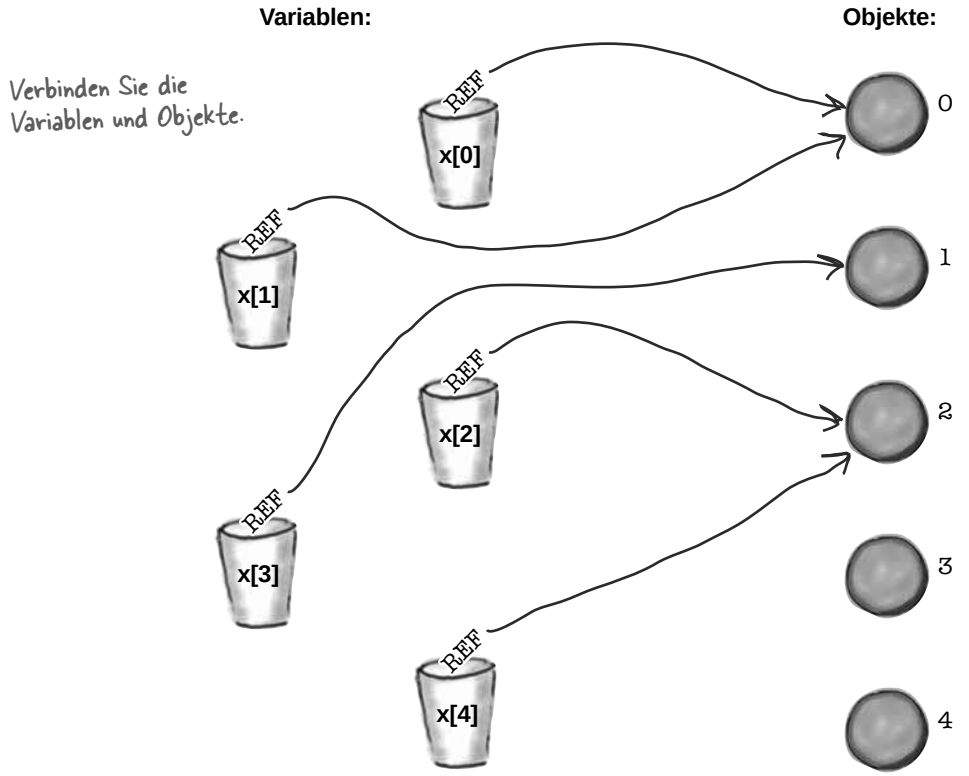
```
Fruit = Banana
Fruit = Blueberry
Fruit = Pomegranate
Fruit = Cherry
```

```
fun main(args: Array<String>) {
    val index = arrayOf(1, 3, 4, 2)
    val fruit = arrayOf("Apple", "Banana", "Cherry", "Blueberry", "Pomegranate")
    var x = 0
    var y: Int
    while (x < 4) {
        y = index[x]
        println("Fruit = ${fruit[y]}")
        x = x + 1
    }
}
```

Vermischte Referenzen, Lösung

Unten sehen Sie ein kurzes Kotlin-Programm. Bei Erreichen der Zeile `// Mach was` sind bereits mehrere Variablen und Objekte erzeugt worden. Sie sollen herausfinden, welche Variablen auf welche Objekte verweisen, wenn `// Mach was` erreicht ist. Auf einige Objekte wird möglicherweise mehr als einmal verwiesen. Zeichnen Sie Linien, um die Variablen und Objekte miteinander zu verbinden.

```
fun main(args: Array<String>) {
    val x = arrayOf(0, 1, 2, 3, 4)
    x[3] = x[2]
    x[4] = x[0]
    x[2] = x[1]
    x[1] = x[0]
    x[0] = x[1]
    x[4] = x[3]
    x[3] = x[2]
    x[2] = x[4]
    // Mach was
}
```



Ihr Kotlin-Werkzeugkasten

Sie haben das zweite Kapitel hinter sich gebracht und Ihren Werkzeugkasten um Basisdatentypen und Variablen erweitert.

Den kompletten Code dieses Kapitels können Sie hier herunterladen: https://tinyurl.com/HFKotlin.

Punkt für Punkt

- Damit der Compiler eine Variable erzeugen kann, muss er ihren Namen und ihren Datentyp kennen und wissen, ob sie wiederverwendet werden kann.
- Wurde der Datentyp der Variablen nicht explizit festgelegt, leitet der Compiler ihn vom angegebenen Wert ab.
- Eine Variable enthält eine Referenz auf ein Objekt.
- Ein Objekt besitzt Zustand und Verhalten. Über Funktionen kann auf das Verhalten zugegriffen werden.
- Die Deklaration einer Variablen per `var` bedeutet, dass die enthaltene Objektreferenz ersetzt werden kann. Die Deklaration per `val` sorgt dafür, dass die Referenz auf das Objekt während der Laufzeit des Programms nicht verändert werden kann.
- Kotlin besitzt mehrere Basisdatentypen: `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Boolean`, `Char` und `String`.
- Um den Datentyp einer Variablen explizit anzugeben, setzen Sie hinter den Namen der Variablen einen Doppelpunkt, gefolgt vom Typ:

```
var tinyNum: Byte
```

- Um einer Variablen einen Wert zuweisen zu können, muss dieser einen kompatiblen Datentyp besitzen.
- Passt der Wert nicht in den neuen Typ, kann die Präzision leiden.
- Um ein Array zu erzeugen, verwenden Sie die `arrayOf`-Funktion:

```
var myArray = arrayOf(1, 2, 3)
```

- Anhand des Index kann auf die einzelnen Arrayelemente zugegriffen werden, z. B. `myArray[0]`. Das erste Element eines Arrays hat den Index 0.
- Die Anzahl der Elemente in einem Array erhält man per `myArray.size`.
- Der Compiler kann den Datentyp für ein Array aus dessen Elementen ableiten. Um den Typ eines Arrays explizit festzulegen, können Sie schreiben:

```
var myArray: Array<Byte>
```

- Auch wenn Sie ein Array per `val` deklarieren, können Sie die enthaltenen Elemente verändern.
- Um schnell und einfach innerhalb eines Strings auf Variablen zuzugreifen oder Ausdrücke auszuwerten, können Sie String-Templates verwenden.