

```
// Eine Ente tanzen lassen.  
func DuckDance(duck Duck) {  
    duck.Quack()  
    duck.Walk()  
    duck.Quack()  
    duck.Walk()  
    duck.Quack()  
}
```

Praktisch ist, dass ein Typ nirgendwo explizit definieren muss, dass er dieses Interface implementiert. Es genügt einfach, dass in unserem Beispiel für den Typ eines übergebenen Parameters die beiden Methoden `Quack()` und `Walk()` definiert sind. Der Compiler prüft bei einem Aufruf der Funktion `Dance()`, ob dies zutrifft. Damit würde ein Fehler wie oben mit dem Hund als Beispiel bereits zur Kompilierzeit gefunden. Gleichzeitig wird bei einem Papagei mit den beiden gewünschten Methoden kein Fehler ausgegeben. So bietet Go die Vorteile des Duck Typing ohne seine Nachteile.

1.13 Willkommen im Paralleluniversum

Unser Ziel ist nicht nur die Entwicklung eines funktionierenden Programms, wir wollen auch unsere Hardware ausnutzen – wozu haben wir einen Rechner mit mehreren Kernen? Und außerdem soll unser Programm auch robust sein. Go arbeitet hier nach dem Prinzip der von Tony Hoare entwickelten *Communicating Sequential Processes* (CSP).

Viele traditionelle Programmiersprachen greifen auf durch das Betriebssystem zur Verfügung gestellte *Threads* zurück, eine Form leichtgewichtiger Prozesse innerhalb des Applikationsprozesses. Der Datenaustausch zwischen ihnen findet auf der Basis von *Shared Memory* statt. Hier kommen Variablen zum Einsatz, die im Zugriff aller beteiligten Threads liegen. Die Kontrolle des Zugriffs (also das Verhindern eines gleichzeitig schreibenden Zugriffs oder des Lesens durch einen Thread, während ein weiter die Daten verändert) soll durch Sperren verhindert werden. Die Manipulation dieser Sperren wird idealerweise im Rahmen der Bibliothek vollzogen, in deren Hoheit die geteilten Daten liegen. Das Gegenteil ist das manuelle Sperren und Entsperren durch den eigenen Quellcode, der die geteilten Variablen nutzt. Hier können sich schnell Fehler einschleichen. Weiterhin kann es bei falschen Reihenfolgen oder nicht bedachten Ablaufpfaden zu Blockaden während der Programmausführung kommen.

Bei der Nutzung von Communicating Sequential Processes wird auf Shared Memory verzichtet. Die nebenläufigen Routinen, seien es Threads oder im Falle von Go die *Goroutinen*, tauschen ihre Informationen über Nachrichtenkanäle aus. Im Rahmen einer Schleife werden nun nach und nach innerhalb der empfangenden Routine Daten aus einem der überwachten *Channels* ausgelesen und sequenziell verarbeitet. Hierbei kann es sich um je einen Channel pro Aufgabe der Routine oder auch um einen Channel für unterschiedliche Aufgaben handeln, je nach Ausprägung der zu übertragenden Daten. Der wesentliche Kern dieser Arbeitsweise ist, dass pro Goroutine immer nur eine Aktion auf einmal ausgeführt wird, egal ob sie nur lesend oder auch schreibend ist. Kollisionen können so nicht auftreten.

Ist nun eine Architektur mit einer sehr großen Anzahl nebenläufiger Instanzen gewünscht, ist der Overhead durch die Threads des Betriebssystems vielfach zu hoch. Der Lösungsansatz seitens der Go-Entwickler sind an dieser Stelle besagte Goroutinen. Hierbei handelt es sich um ein noch leichtgewichtigeres Prozessmodell, das auf der Basis eines *Thread Pools* realisiert wurde. Die Ausführung der Goroutinen wird also auf eine Menge von Threads verteilt. Blockierende Goroutinen, beispielsweise wegen eines sperrenden Systemaufrufs, werden von der Laufzeitumgebung erkannt. Weitere dem gleichen Thread zugeordnete Goroutinen werden in diesem Fall anderen Threads aus dem Pool zugeordnet, sodass ihr Ablauf nicht gestört ist. Gleichzeitig wurde die Speicherverwaltung für Goroutinen optimiert. Dank eines segmentierten Stacks kann sie den initialen Verbrauch gering halten und flexibel auf mehr Bedarf zur Laufzeit reagieren. So ist es möglich, in einem Laufzeitsystem mehr als 100.000 parallele Routinen auszuführen. Eine solche Anzahl von Threads wäre hingegen unmöglich.

Natürlich lassen sich Goroutinen auch ohne Channels einsetzen, beispielsweise für im Hintergrund zu erledigende Aufgaben, denen beim Aufruf alle notwendigen Daten mitgegeben werden können. Rückgaben können ohne Channels jedoch nur in Variablen erfolgen. Als Goroutine eignet sich jede Funktion, sei sie anonym, benannt oder eine Methode. Eine eventuelle Rückgabe wird ignoriert. Ihre Verarbeitung im Hintergrund wird mit dem Schlüsselwort **go** angestoßen, der Aufruf hat keine Rückgabe.

Quelltext 1.32
Einfache Verarbeitung
im Hintergrund

```
// Funktion longCalculation() für eine lange  
// Berechnung im Hintergrund starten.  
  
go longCalculation(inputData)
```

1.13.1 Kanalarbeiter

Auch wenn ein Datenaustausch zwischen Goroutinen über gemeinsame Variablen möglich ist, so ist dies nicht der empfohlene und in Go übliche Weg. Als Sprachmittel für die Kommunikation zwischen verschiedenen parallel arbeitenden Einheiten kommen die bereits genannten Channels zum Einsatz. Bei ihnen handelt es sich um Referenztypen, deren Instanzen mit `make()` erzeugt werden. Bei der Erzeugung wird definiert, über welchen Typ die zu transportierenden Daten verfügen müssen. Diese können dann durch eine Goroutine in einen Channel geschrieben und durch eine weitere wieder ausgelesen werden. Diese Operationen sind atomar.

Über Variablen oder als Parameter sind die erzeugten Channels nun allen nutzenden Goroutinen zur Verfügung zu stellen. Dies bedeutet, dass es beliebig viele lesende und schreibende Goroutinen zu einem Channel geben darf. Regulär sind Channels ungepuffert, sie können jedoch auch mit einem Puffer erzeugt werden. Während erstere beim Schreibzugriff blockieren, solange ein zuvor geschriebenes Datum noch nicht gelesen wurde, können gepufferte Channels hingegen die definierte Menge an Daten aufnehmen, bevor auch sie blockieren.

Der Operator für das Lesen und Schreiben der Daten ist ein Pfeil in der Form `<-`. Beim Versand der Informationen steht der Channel links des Pfeils, beim Empfang rechts.

```
// Goroutine 1.
func GoroutineOne() {
    flagChan := make(chan bool)

    // Goroutine 2 starten.
    go GoroutineTwo(flagChan)

    // Daten versenden.
    flagChan <- true
}

// Goroutine 2.
func GoroutineTwo(fc chan bool) {
    // Endlosschleife.
    for {
        // Daten empfangen.
        flag := <- fc

        println("Signal empfangen.")
    }
}
```

Quelltext 1.33
Umgang mit Channels

Im Beispiel wird in der Schleife nur aus einem Channel gelesen, was in der Form `for flag := range fc { ... }` eleganter möglich ist. In der Regel wird jedoch aus mehreren Channels gelesen. Hier kommt das in Abschnitt 3.9 vorgestellte `select`-Konstrukt zum Einsatz. Im Quelltext 1.37 ist dessen Anwendung zu sehen.

1.13.2 Aufgabenverteilung

Als nächster Schritt soll in unserem Beispiel eine Person nun eine Liste ihrer Kumpel ausdrucken können. Das Drucken ist in der Regel ein langsamer Vorgang. Für das eigentliche Programm ist der Druckvorgang dabei uninteressant, es möchte seinen Programmfluss fortsetzen. Insofern eignet es sich sehr gut für eine Ausführung im Hintergrund. Ein in der Druckfunktion genutzter neuer Datentyp `Printer` soll dies bieten.

Quelltext 1.34
Nutzung eines
Druckers

```
// Kumpel drucken.
func (p *Person) PrintBuddies() int {
    // Kumpel für Druck aufbereiten.
    printData := p.RenderBuddyTable()

    // Druck aufrufen, wird intern im Hintergrund
    // erledigt (s.u.). Das Programm wird unmittelbar
    // fortgesetzt.
    printer := system.Printer()
    jobId := printer.Print(printData)

    return jobId
}
```

Für den Drucker selbst ist es wichtig, dass sich mehrere parallel abgeschickte Druckaufträge nicht vermischen. Gleichzeitig sollen später ja auch Anfragen über den Druckfortschritt verarbeitet oder Aufträge sogar storniert werden. Hierfür kommen die Channels und ein weiteres Muster zum Einsatz.

Den Start macht eine Erzeugung eines Druckers mit der Anlage eines oder mehrerer Channel für die Kommunikation sowie dem Start einer Goroutine für die Nachrichtenverarbeitung im Hintergrund. Außerdem werden eigene Typen zur Kommunikation benötigt. Diese werden jedoch nicht exportiert, da wir auch hier Implementierungsdetails vor dem Nutzer verbergen wollen. Die so definierte Funktion erzeugt eine Instanz des Typs `Printer`. Intern benötigte Informationen über den Druckertyp kann sie beispielsweise wieder über die bereits vorgestellte Konfiguration ermitteln.

```
// Druckauftrag.
type printRequest struct {
    data          string
    responseChan chan int
}

// Stornoauftrag.
type cancelRequest struct {
    jobId        int
    responseChan chan int
}

// Statusanfrage.
type statusRequest struct {
    jobId        int
    responseChan chan int
}

// Drucker.
type Printer struct {
    ...
}

// Drucker erzeugen.
func NewPrinter() *Printer {
    printer := new(Printer)

    ...

    printer.printRequestChan = make(chan *printRequest)
    printer.cancelRequestChan = make(chan *cancelRequest)
    printer.statusRequestChan = make(chan *statusRequest)

    ...

    // Start der Goroutine für die Verarbeitung
    // im Hintergrund.

    go printer.backend()

    ...

    return printer
}

```

Quelltext 1.35
Drucker-Erzeugung
mit Goroutine

Der Typ `printRequest` kapselt die für den Versand eines Druckauftrags notwendigen Informationen. Allerdings soll für eine spätere Statusabfrage auch ein Identifikator für den Auftrag zurückgeliefert werden. Hierfür ist ein spezieller Channel für die Antwort an den Aufrufer notwendig. In der Methode `Print()` wird nun die Anfrage zusammen mit einem Channel für die Antwort erzeugt. Anschließend wird auf die Antwort gewartet und diese an den Aufrufer zurückgegeben.

Quelltext 1.36
Frage und Antwort

```
func (p *Printer) Print(data string) int {
    // Anfrage mit Antwort-Channel erzeugen.
    request := &printRequest{data, make(chan int)}

    // Anfrage an Goroutine im Hintergrund senden.
    p.printRequestChan <- request

    // Auf Antwort warten und zurück liefern.
    return <-request.responseChan
}
```

Hier passiert also eigentlich nichts Spannendes, nur eine Kapselung der Kommunikation. Das andere Ende des Channels für neue Druckaufträge wird innerhalb der `backend()`-Methode überwacht. Innerhalb einer Endlosschleife werden alle den Drucker betreffenden Channels überwacht und eventuell anliegende Nachrichten entgegengenommen. Durch diese sequenzielle Ausführung wird wie schon erwähnt immer nur eine Anfrage auf einmal verarbeitet. So können sich parallel eintreffende Anfragen nicht vermischen und zu einem inkonsistenten Zustand führen. Eventuell notwendige interne Goroutinen kommunizieren ebenfalls auf diesem Weg mit der übergeordneten Instanz, sodass auch sie keine Bedrohung des Zustands darstellen.

Quelltext 1.37
Arbeit im Hintergrund

```
// Hintergrundfunktion des Druckers.
func (p *Printer) backend() {
    for {
        select {
            case printRequest := <-p.printRequestChan:
                // Druckauftrag empfangen.
                jobId := p.addToQueue(printRequest.data)

                printRequest.responseChan <- jobId
            case cancelRequest := <-p.cancelRequestChan:
                // Druckauftrag stornieren.
                status := p.cancelJob(statusRequest.jobId)

                statusRequest.responseChan <- status
            case statusRequest := <-p.statusRequestChan:
```

```

// Statusanfrage empfangen.
status := p.jobStatus(statusRequest.jobId)

statusRequest.responseChan <- status
}
}
}

```

1.13.3 Kollegiales Miteinander

Dieses Muster ist nicht für alle Typen notwendig. Werden aber Datenstrukturen durch mehrere Goroutinen gleichzeitig genutzt und sogar verändert, so eignet sich die Serialisierung aller Anfragen über Channels sehr gut. Dennoch ist bei den Methoden darauf zu achten, dass die auszuführenden Operationen und Modifikationen atomar sind. So darf der Drucker beispielsweise nicht über zwei notwendigerweise in Folge auszuführende Schritte `Prepare()` und `Print()` angesteuert werden müssen. In diesem Fall könnte sich ein zweites `Prepare()` dazwischendrängen, gegebenenfalls rutscht je nach Ausführungszeit auch der Druck dazwischen. Es ist leicht ersichtlich, dass dies nicht das gewünschte Verhalten ist.

Beim Design eines nebenläufigen Systems kommt es vielmehr darauf an, dass ein Netzwerk von Komponenten aufgebaut wird, die über definierte Protokolle kommunizieren. Sie versenden untereinander Anfragen und erhalten bei Bedarf Antworten. In unserem Beispiel wird beispielsweise in den Methoden im Anschluss an den Versand einer Anfrage direkt auf die Antwort gewartet. Techniken zur Vermeidung von Blockaden an dieser Stelle werden später noch vorgestellt.

Dieses Modell eigenständig agierender und miteinander kommunizierender Komponenten ist nicht der einzige Anwendungsfall für Nebenläufigkeit. Goroutinen lassen sich auch zur Beschleunigung individueller Aufgaben einsetzen. So lässt sich die Suche von Elementen in einem Array durch die Suche in einzelnen Abschnitten parallelisieren. Hierfür werden Slices aus diesen Abschnitten erzeugt, was in Go sehr einfach ist.

```

// Array mit 12 Elementen.
a := [12]int{0, 5, 10, 15, 20, 25,
          30, 35, 40, 45, 50, 55}

// Drei das Array überlagernde Slices.
s1 := a[0:3]
s2 := a[4:7]
s3 := a[8:11]

```

Quelltext 1.38
Slices aus Arrays erzeugen

Diese Slices überlagern das Array, Veränderungen in ihnen wirken sich auch auf das Array aus. Im Fall der Suche wird allerdings nur gelesen. Die so erzeugten Slices können nun mehreren Such-Goroutinen übergeben werden, diese melden ihre Ergebnisse über einen Antwort-Channel zurück.

Das Kapitel 6 befasst sich mit weiteren in Go typischen Mustern. Sie machen deutlich, wie Goroutinen und Channel für eine Vielzahl von Lösungen optimal zusammenspielen.

1.14 Zusammenfassung

Nach dieser Einführung haben Sie die wichtigsten Eigenschaften der Sprache Go kennengelernt und sind nun in der Lage, erste kleine Programme zu entwickeln. Neben Installation und Kompilierung haben Sie den Aufbau des Hauptprogramms, die Modularisierung durch Funktionen und Packages, Typen, Interfaces und im Kleinen auch die Nebenläufigkeit kennengelernt. Nun also viel Spaß bei den ersten Versuchen, bevor es mit den Details der Sprache weitergeht.