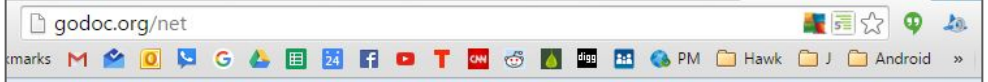
TCP Servers

bufio Scanners, strings.Fields, concurrency

but first, a review



GoDoc Home Index About Search

Go: net Index | Examples | Files | Directories

package net

```
import "net"
```

Package net provides a portable interface for network I/O, including TCP/IP, UDP, domain name resolution, and Unix domain sockets.

Although the package provides access to low-level networking primitives, most clients will need only the basic interface provided by the Dial, Listen, and Accept functions and the associated Conn and Listener interfaces. The crypto/tls package uses the same interfaces and similar Dial and Listen functions.

The Dial function connects to a server:

```
conn, err := net.Dial("tcp", "google.com:80")
if err != nil {
        // handle error
}
fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
status, err := bufio.NewReader(conn).ReadString('\n')
// ...
```

The Listen function creates servers:

```
ln, err := net.Listen("tcp", ":8080")
if err != nil {
        // handle error
}
for {
        conn, err := ln.Accept()
        if err != nil {
                // handle error
        }
        go handleConnection(conn)
}
```

main.go

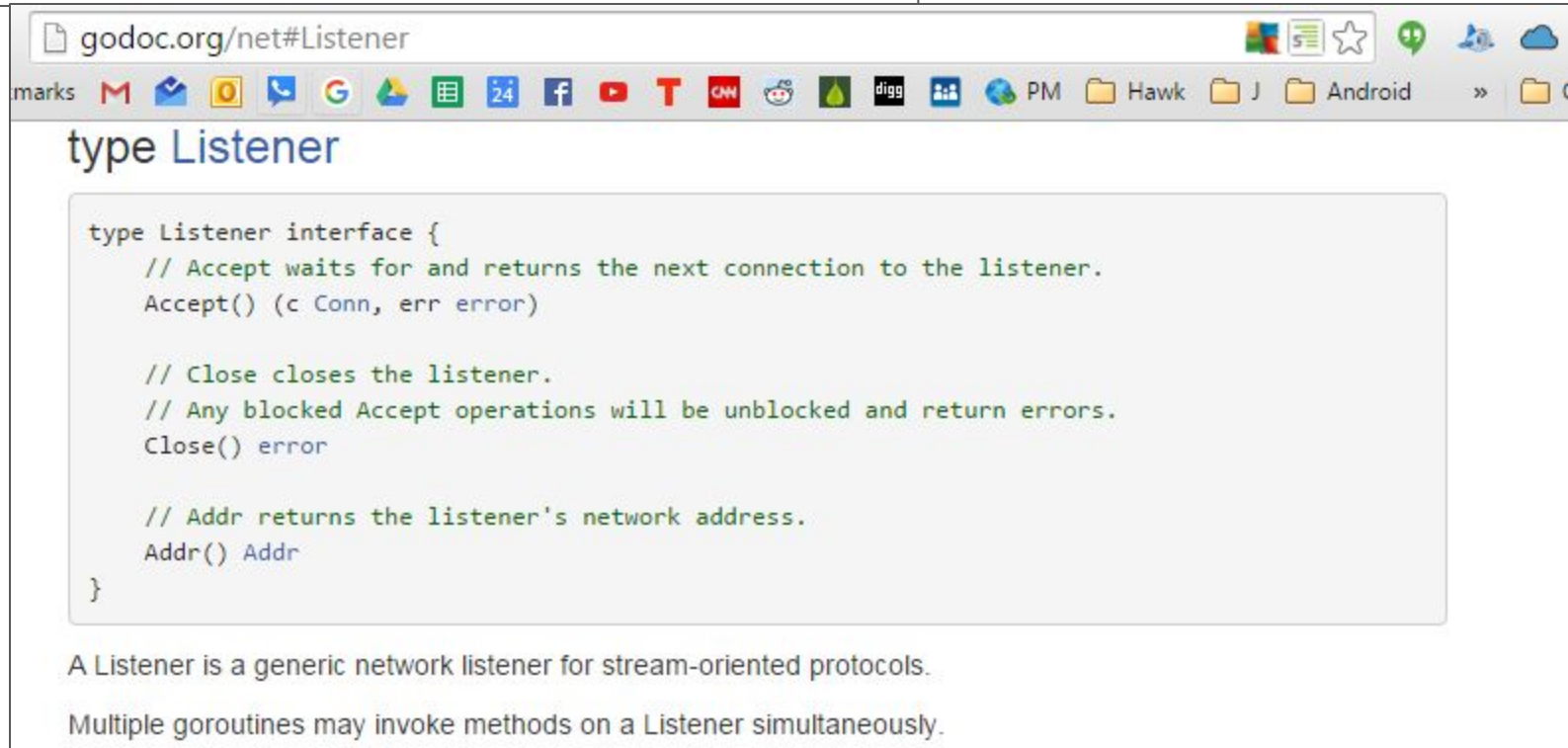
```go
package main

import (
    "net"
    "log"
    "bufio"
    "fmt"
)

func handle(conn net.Conn) {
    defer conn.Close()

    scanner := bufio.NewScanner(conn)
    for scanner.Scan() {
        ln := scanner.Text()
        fmt.Println(ln)
    }
}

func main() {
    li, err := net.Listen("tcp", ":9000")
    if err != nil{
        log.Fatalln(err)
    }
    defer li.Close()

    for {
        conn, err := li.Accept()
        if err != nil {
            log.Fatalln(err)
        }
        handle(conn)
    }
}

```

What will this code do?

func Listen ¶

```
func Listen(net, laddr string) (Listener, error)
```

Listen announces on the local network address laddr. The network net must be a stream-oriented network: "tcp", "tcp4", "tcp6", "unix" or "unixpacket". See Dial for the syntax of laddr.



type Listener

```
type Listener interface {
    // Accept waits for and returns the next connection to the listener.
    Accept() (c Conn, err error)

    // Close closes the listener.
    // Any blocked Accept operations will be unblocked and return errors.
    Close() error

    // Addr returns the listener's network address.
    Addr() Addr
}
```

A Listener is a generic network listener for stream-oriented protocols.

Multiple goroutines may invoke methods on a Listener simultaneously.

main.go

```go
package main

import (
    "net"
    "log"
    "bufio"
    "fmt"
)

func handle(conn net.Conn) {
    defer conn.Close()

    scanner := bufio.NewScanner(conn)
    for scanner.Scan() {
        ln := scanner.Text()
        fmt.Println(ln)
    }
}

func main() {
    li, err := net.Listen("tcp", ":9000")
    if err != nil{
        log.Fatalln(err)
    }
    defer li.Close()

    for {
        conn, err := li.Accept()
        if err != nil {
            log.Fatalln(err)
        }
        handle(conn)
    }
}

```

```
Tue Sep 15 23:27:04 PDT 2015
~ $ telnet localhost 9000
Trying ::1...
Connected to localhost.
Escape character is '^]'.
hello
look at that
from one client connection via telnet
the listening tcp server receives what was sent
by the client
_
```

```
Tue Sep 15 23:14:54 PDT 2015
GolangTraining $ cd 41_TCP/05_redis-clone/v01/
v01 $ go run main.go
hello
look at that
from one client connection via telnet
the listening tcp server receives what was sent
by the client
```

type Conn

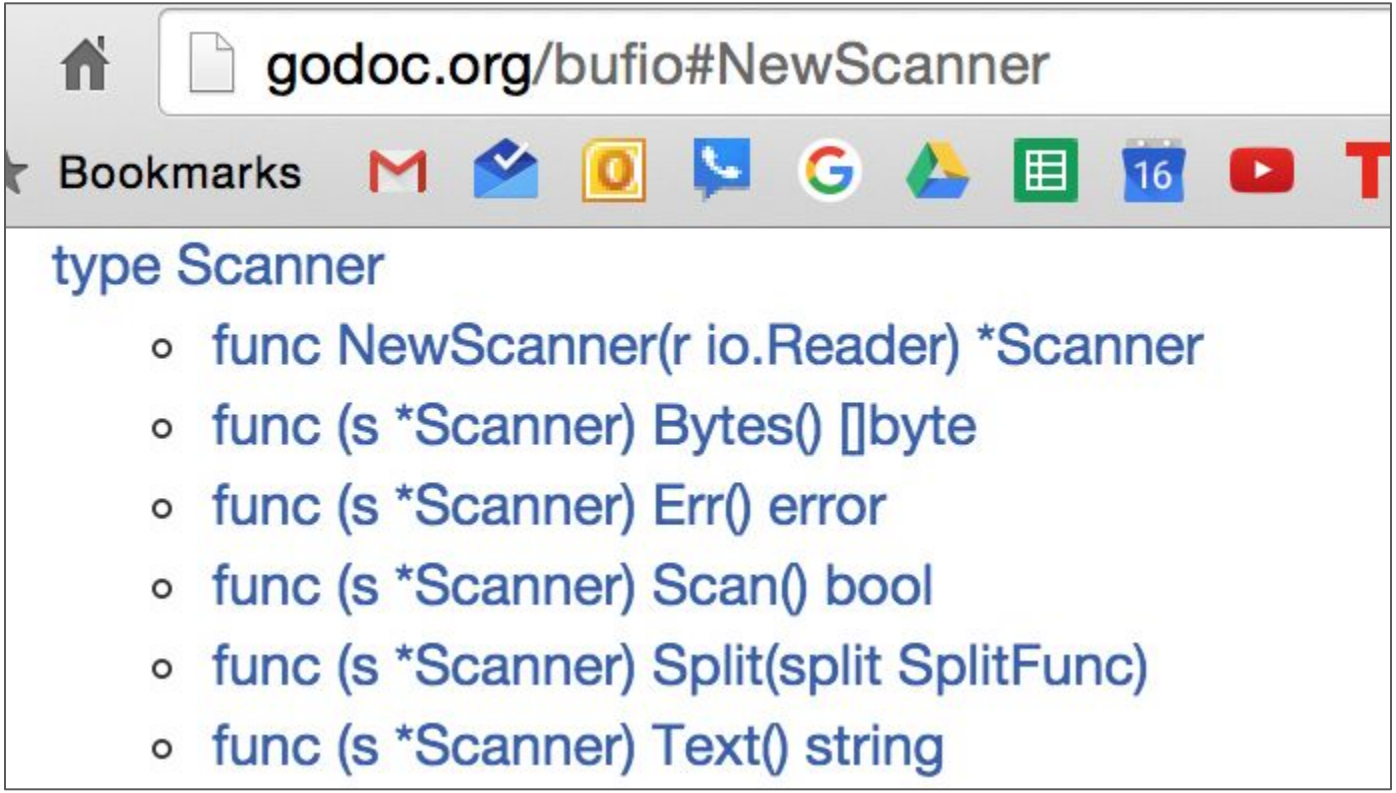
```
type Conn interface {
    // Read reads data from the connection.
    // Read can be made to time out and return a Error with Timeout() == true
    // after a fixed time limit; see SetDeadline and SetReadDeadline.
    Read(b []byte) (n int, err error)

    // Write writes data to the connection.
    // Write can be made to time out and return a Error with Timeout() == true
    // after a fixed time limit; see SetDeadline and SetWriteDeadline.
    Write(b []byte) (n int, err error)

    // Close closes the connection.
    // Any blocked Read or Write operations will be unblocked and return errors.
    Close() error
```

package bufio

package bufio

```
import "bufio"
```

Package bufio implements buffered I/O. It wraps an io.Reader or io.Writer object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.


godoc.org/bufio#NewScanner
Bookmarks
16
type Scanner
func NewScanner(r io.Reader) *Scanner
func (s *Scanner) Bytes() []byte
func (s *Scanner) Err() error
func (s *Scanner) Scan() bool
func (s *Scanner) Split(split SplitFunc)
func (s *Scanner) Text() string

type Scanner

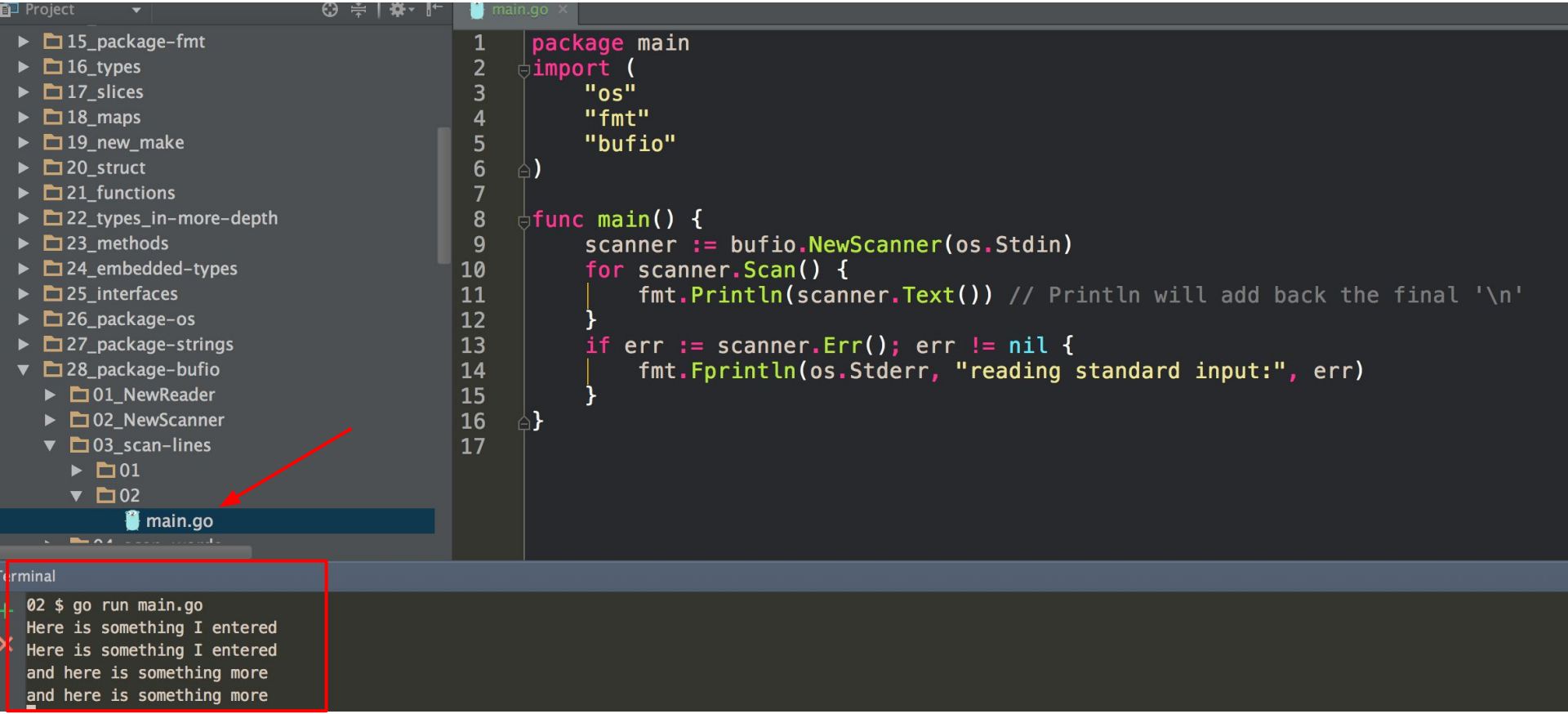
```
type Scanner struct {
    // contains filtered or unexported fields
}
```

Scanner provides a convenient interface for reading data such as a file of newline-delimited lines of text. Successive calls to the Scan method will step through the 'tokens' of a file, skipping the bytes between the tokens. The specification of a token is defined by a split function of type SplitFunc; the default split function breaks the input into lines with line termination stripped. Split functions are defined in this package for scanning a file into lines, bytes, UTF-8-encoded runes, and space-delimited words. The client may instead provide a custom split function.

Scanning stops unrecoverably at EOF, the first I/O error, or a token too large to fit in the buffer. When a scan stops, the reader may have advanced arbitrarily far past the last token. Programs that need more control over error handling or large tokens, or must run sequential scans on a reader, should use bufio.Reader instead.

Code: play

```
scanner := bufio.NewScanner(os.Stdin)
for scanner.Scan() {
    fmt.Println(scanner.Text()) // Println will add back the final '\n'
}
if err := scanner.Err(); err != nil {
    fmt.Fprintln(os.Stderr, "reading standard input:", err)
}
```


Project
15_package-fmt
16_types
17_slices
18_maps
19_new_make
20_struct
21_functions
22_types_in-more-depth
23_methods
24_embedded-types
25_interfaces
26_package-os
27_package-strings
28_package-bufio
01_NewReader
02_NewScanner
03_scan-lines
01
02
main.go
main.go
1 package main
2 import (
3 "os"
4 "fmt"
5 "bufio"
6)
7
8 func main() {
9 scanner := bufio.NewScanner(os.Stdin)
10 for scanner.Scan() {
11 fmt.Println(scanner.Text()) // Println will add back the final '\n'
12 }
13 if err := scanner.Err(); err != nil {
14 fmt.Fprintln(os.Stderr, "reading standard input:", err)
15 }
16 }
17
rminal
02 $ go run main.go
Here is something I entered
Here is something I entered
and here is something more
and here is something more

func NewScanner

```
func NewScanner(r io.Reader) *Scanner
```

NewScanner returns a new Scanner to read from r. The split function defaults to ScanLines.

func (*Scanner) Scan

```
func (s *Scanner) Scan() bool
```

Scan advances the Scanner to the next token, which will then be available through the Bytes or Text method. It returns false when the scan stops, either by reaching the end of the input or an error. After Scan returns false, the Err method will return any error that occurred during scanning, except that if it was io.EOF, Err will return nil. Scan panics if the split function returns 100 empty tokens without advancing the input. This is a common error mode for scanners.

func (*Scanner) Text

```
func (s *Scanner) Text() string
```

Text returns the most recent token generated by a call to Scan as a newly allocated string holding its bytes.

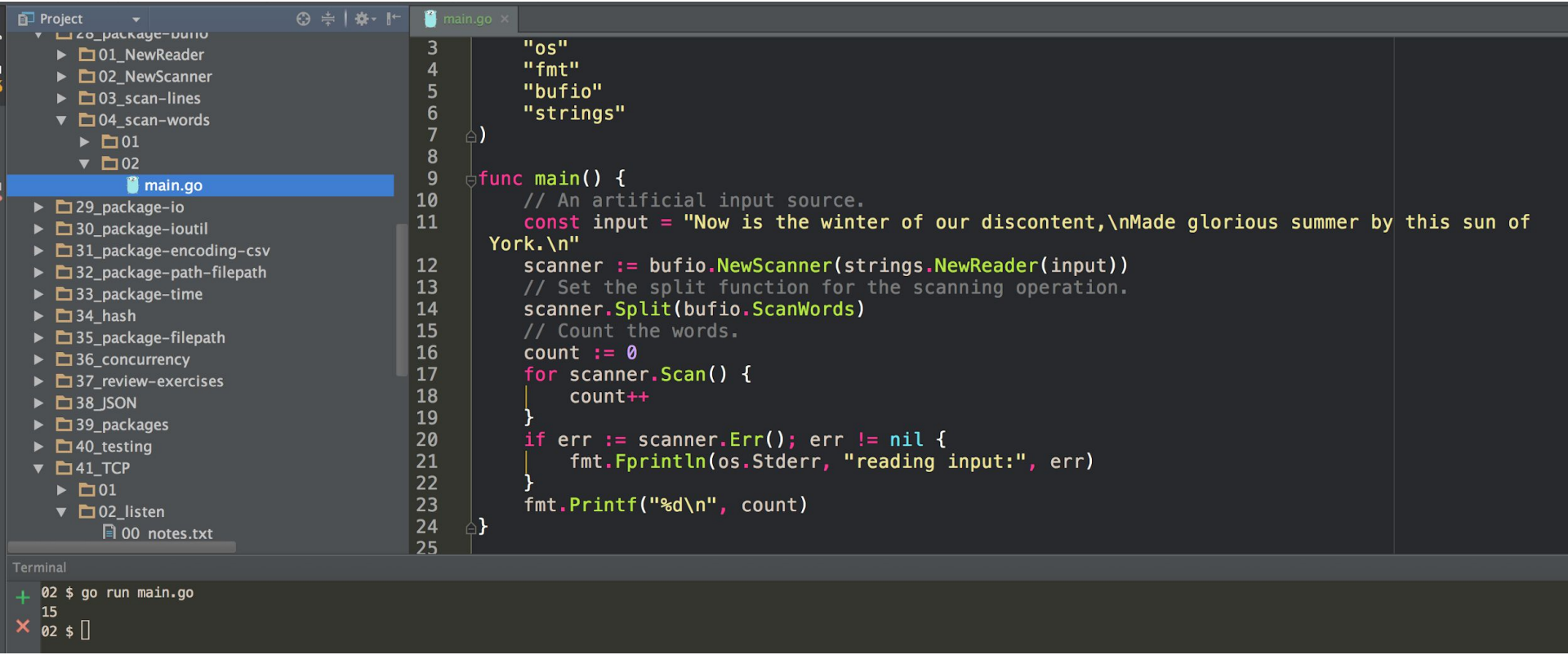
func (*Scanner) Err

```
func (s *Scanner) Err() error
```

Err returns the first non-EOF error that was encountered by the Scanner.

Code: play

```
// An artificial input source.
const input = "Now is the winter of our discontent,\nMade glorious summer by this
scanner := bufio.NewScanner(strings.NewReader(input))
// Set the split function for the scanning operation.
scanner.Split(bufio.ScanWords)
// Count the words.
count := 0
for scanner.Scan() {
    count++
}
if err := scanner.Err(); err != nil {
    fmt.Fprintln(os.Stderr, "reading input:", err)
}
fmt.Printf("%d\n", count)
```


Project
01_NewReader
02_NewScanner
03_scan-lines
04_scan-words
01
02
main.go
29_package-io
30_package-ioutil
31_package-encoding-csv
32_package-path-filepath
33_package-time
34_hash
35_package-filepath
36_concurrency
37_review-exercises
38_JSON
39_packages
40_testing
41_TCP
01
02_listen
00 notes.txt
main.go
"os"
"fmt"
"bufio"
"strings"
)
func main() {
// An artificial input source.
const input = "Now is the winter of our discontent,\nMade glorious summer by this sun of York.\n"
scanner := bufio.NewScanner(strings.NewReader(input))
// Set the split function for the scanning operation.
scanner.Split(bufio.ScanWords)
// Count the words.
count := 0
for scanner.Scan() {
count++
}
if err := scanner.Err(); err != nil {
fmt.Fprintln(os.Stderr, "reading input:", err)
}
fmt.Printf("%d\n", count)
}
Terminal
02 $ go run main.go
15
02 $

func NewScanner

```
func NewScanner(r io.Reader) *Scanner
```

NewScanner returns a new Scanner to read from r. The split function defaults to ScanLines.

func (*Scanner) Split

```
func (s *Scanner) Split(split SplitFunc)
```

Split sets the split function for the Scanner. If called, it must be called before Scan. The default split function is ScanLines.

func ScanWords

```
func ScanWords(data []byte, atEOF bool) (advance int, token []byte, err error)
```

ScanWords is a split function for a Scanner that returns each space-separated word of text, with surrounding spaces deleted. It will never return an empty string. The definition of space is set by unicode.IsSpace.

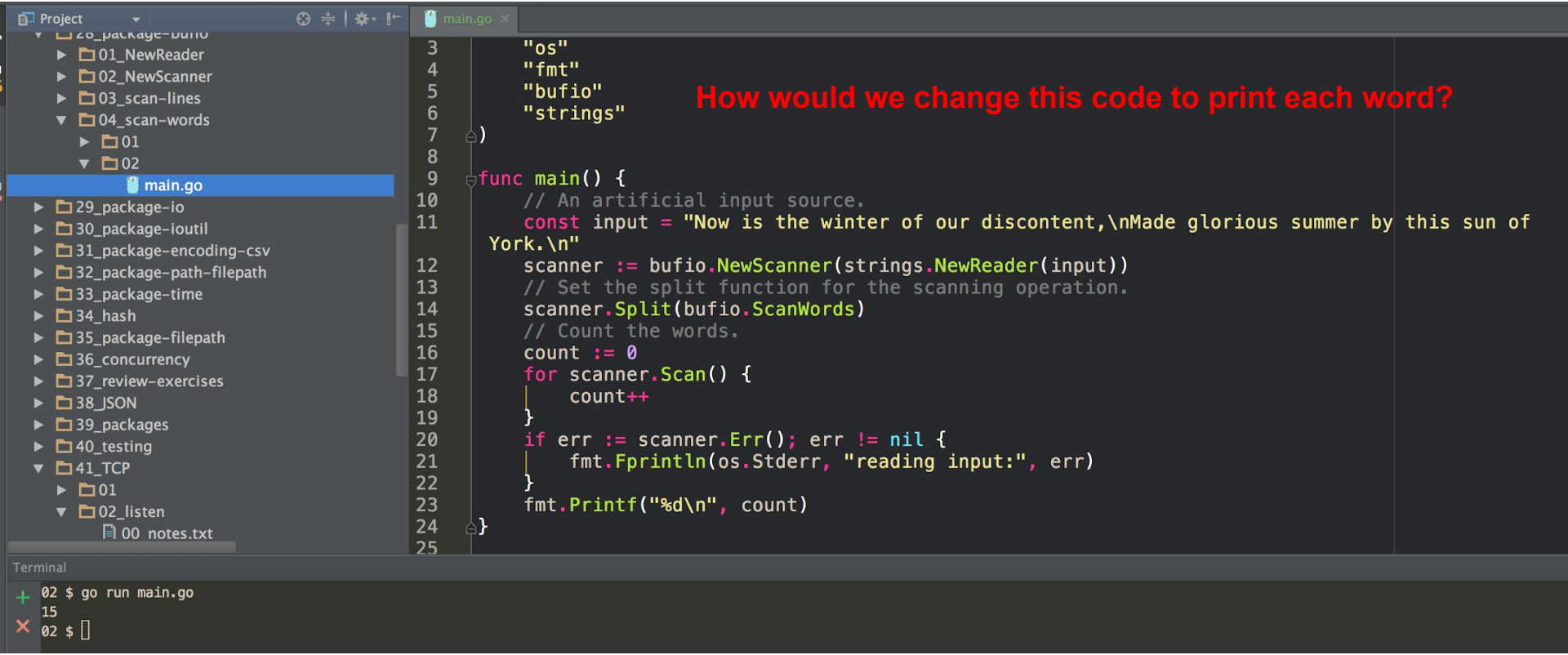
func (*Scanner) Scan

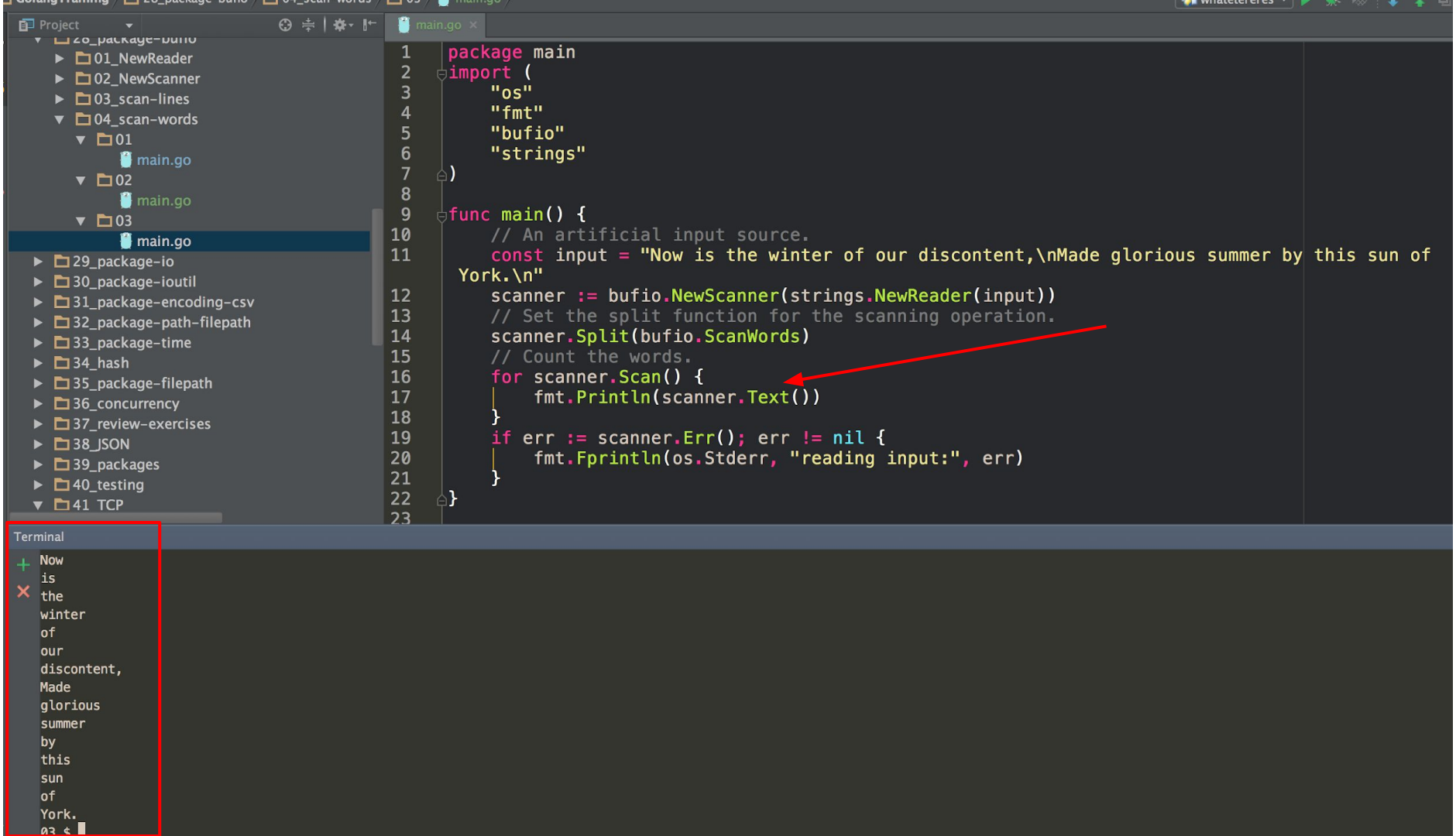
```
func (s *Scanner) Scan() bool
```

Scan advances the Scanner to the next token, which will then be available through the Bytes or Text method. It returns false when the scan stops, either by reaching the end of the input or an error. After Scan returns false, the Err method will return any error that occurred during scanning, except that if it was io.EOF, Err will return nil. Scan panics if the split function returns 100 empty tokens without advancing the input. This is a common error mode for scanners.

Index

Constants
Variables
func ScanBytes(data []byte, atEOF bool) (advance int, token []byte, err error)
func ScanLines(data []byte, atEOF bool) (advance int, token []byte, err error)
func ScanRunes(data []byte, atEOF bool) (advance int, token []byte, err error)
func ScanWords(data []byte, atEOF bool) (advance int, token []byte, err error)


Project
01_NewReader
02_NewScanner
03_scan-lines
04_scan-words
01
02
main.go
29_package-io
30_package-ioutil
31_package-encoding-csv
32_package-path-filepath
33_package-time
34_hash
35_package-filepath
36_concurrency
37_review-exercises
38_JSON
39_packages
40_testing
41_TCP
01
02_listen
00 notes.txt
main.go
"os"
"fmt"
"bufio"
"strings"
)
func main() {
// An artificial input source.
const input = "Now is the winter of our discontent,\nMade glorious summer by this sun of York.\n"
scanner := bufio.NewScanner(strings.NewReader(input))
// Set the split function for the scanning operation.
scanner.Split(bufio.ScanWords)
// Count the words.
count := 0
for scanner.Scan() {
count++
}
if err := scanner.Err(); err != nil {
fmt.Fprintln(os.Stderr, "reading input:", err)
}
fmt.Printf("%d\n", count)
}
How would we change this code to print each word?
Terminal
02 $ go run main.go
15
02 $


Project
01_NewReader
02_NewScanner
03_scan-lines
04_scan-words
01
main.go
02
main.go
03
main.go
29_package-io
30_package-ioutil
31_package-encoding-csv
32_package-path-filepath
33_package-time
34_hash
35_package-filepath
36_concurrency
37_review-exercises
38_JSON
39_packages
40_testing
41_TCP
main.go
1 package main
2 import (
3 "os"
4 "fmt"
5 "bufio"
6 "strings"
7)
8
9 func main() {
10 // An artificial input source.
11 const input = "Now is the winter of our discontent,\nMade glorious summer by this sun of York.\n"
12 scanner := bufio.NewScanner(strings.NewReader(input))
13 // Set the split function for the scanning operation.
14 scanner.Split(bufio.ScanWords)
15 // Count the words.
16 for scanner.Scan() {
17 fmt.Println(scanner.Text())
18 }
19 if err := scanner.Err(); err != nil {
20 fmt.Fprintln(os.Stderr, "reading input:", err)
21 }
22 }
23
Terminal
Now
is
the
winter
of
our
discontent,
Made
glorious
summer
by
this
sun
of
York.
03 $

func NewScanner

```
func NewScanner(r io.Reader) *Scanner
```

NewScanner returns a new Scanner to read from r. The split function defaults to ScanLines.

func (*Scanner) Split

```
func (s *Scanner) Split(split SplitFunc)
```

Split sets the split function for the Scanner. If called, it must be called before Scan. The default split function is ScanLines.

func ScanWords

```
func ScanWords(data []byte, atEOF bool) (advance int, token []byte, err error)
```

ScanWords is a split function for a Scanner that returns each space-separated word of text, with surrounding spaces deleted. It will never return an empty string. The definition of space is set by unicode.IsSpace.

func (*Scanner) Scan

```
func (s *Scanner) Scan() bool
```

Scan advances the Scanner to the next token, which will then be available through the Bytes or Text method. It returns false when the scan stops, either by reaching the end of the input or an error. After Scan returns false, the Err method will return any error that occurred during scanning, except that if it was io.EOF, Err will return nil. Scan panics if the split function returns 100 empty tokens without advancing the input. This is a common error mode for scanners.

func (*Scanner) Text

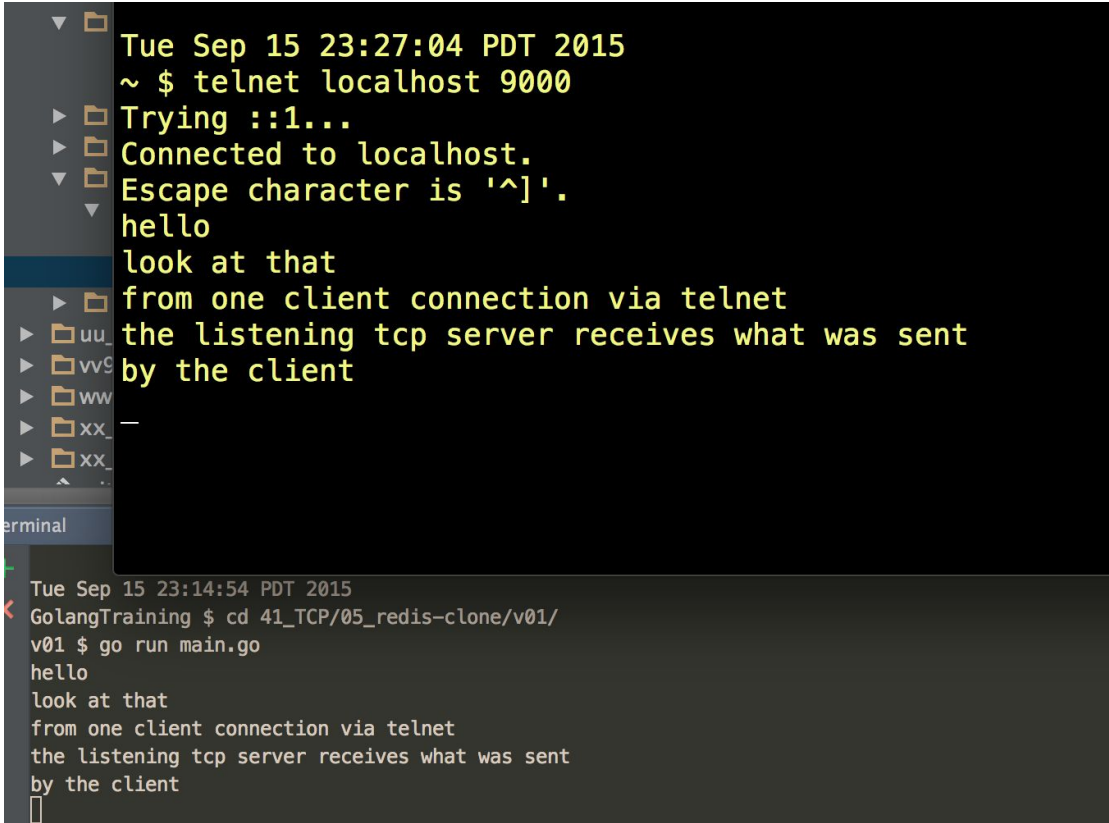
```
func (s *Scanner) Text() string
```

Text returns the most recent token generated by a call to Scan as a newly allocated string holding its bytes.

main.go

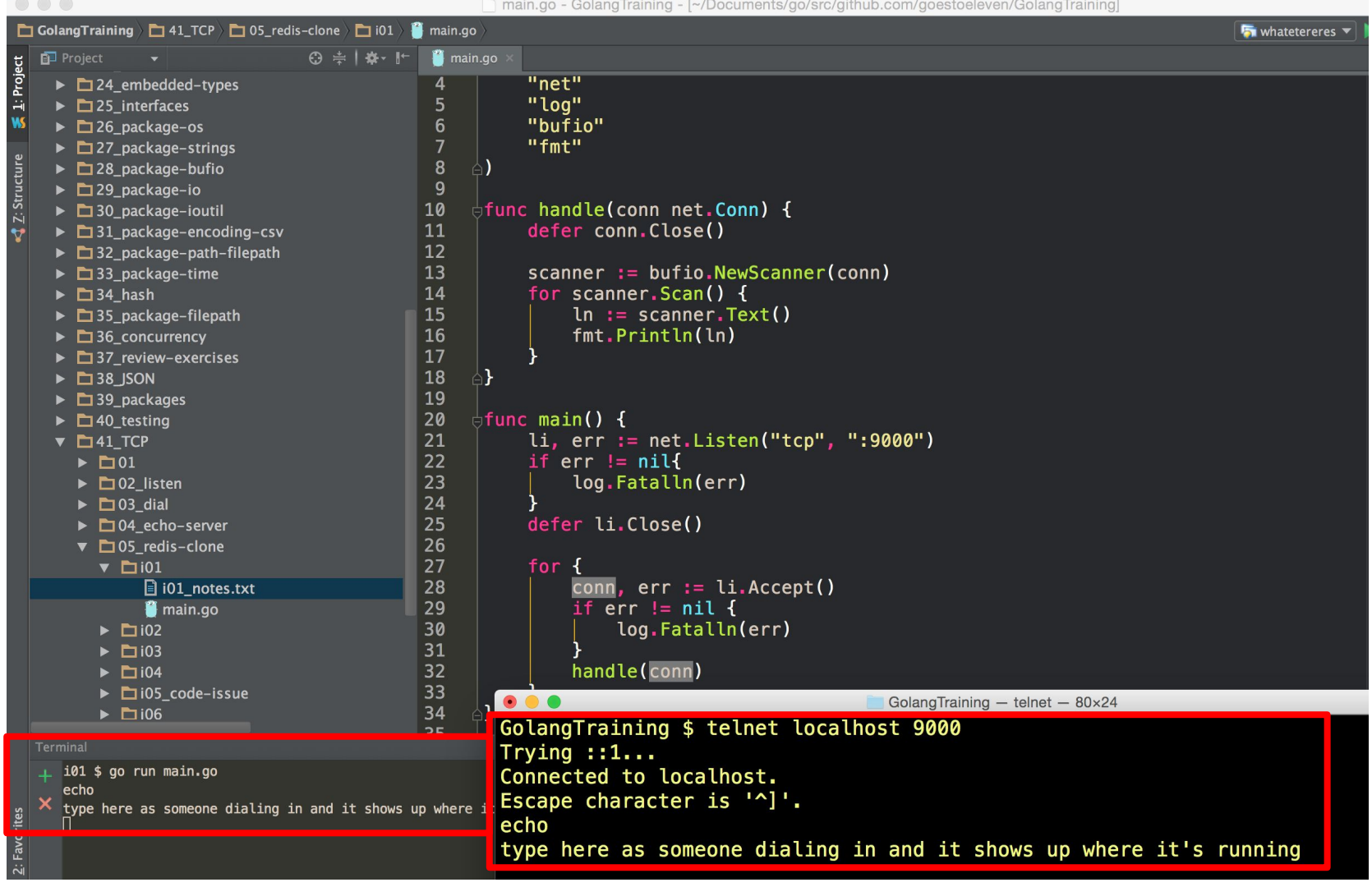
```go
package main

import (
    "net"
    "log"
    "bufio"
    "fmt"
)

func handle(conn net.Conn) {
    defer conn.Close()

    scanner := bufio.NewScanner(conn)
    for scanner.Scan() {
        ln := scanner.Text()
        fmt.Println(ln)
    }
}

func main() {
    li, err := net.Listen("tcp", ":9000")
    if err != nil{
        log.Fatalln(err)
    }
    defer li.Close()

    for {
        conn, err := li.Accept()
        if err != nil {
            log.Fatalln(err)
        }
        handle(conn)
    }
}
```

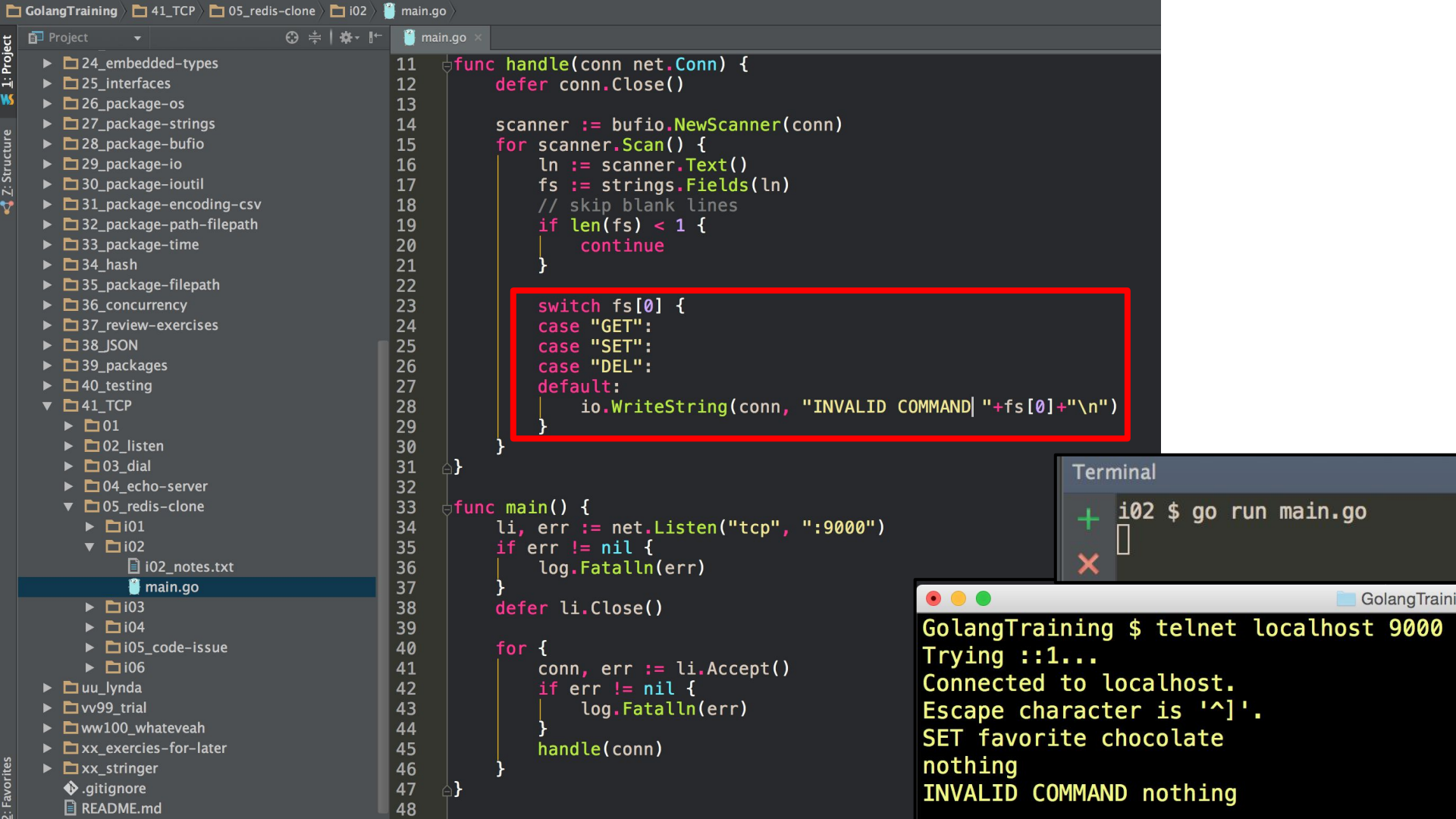
Show of hands:
Does this code make more sense now that we've talked about scanners?



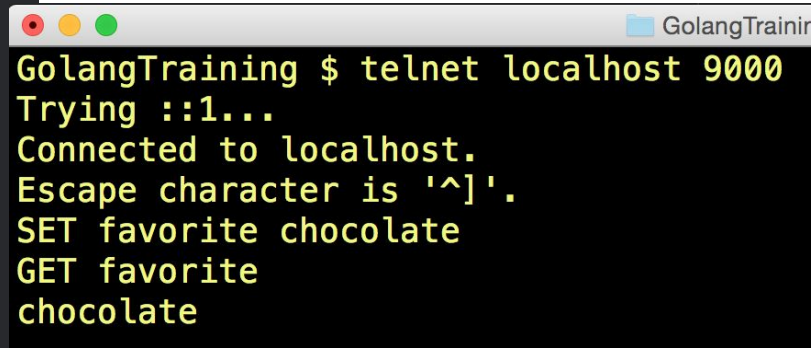
exercise

Create a simplified redis clone which accepts GET, SET and DEL commands. ‘GET <KEY>’ should write the value of <KEY> followed by a new line. ‘SET <KEY> <VALUE>’ should set the value of <KEY>. ‘DEL <KEY>’ should remove the value. Data should be stored in memory.

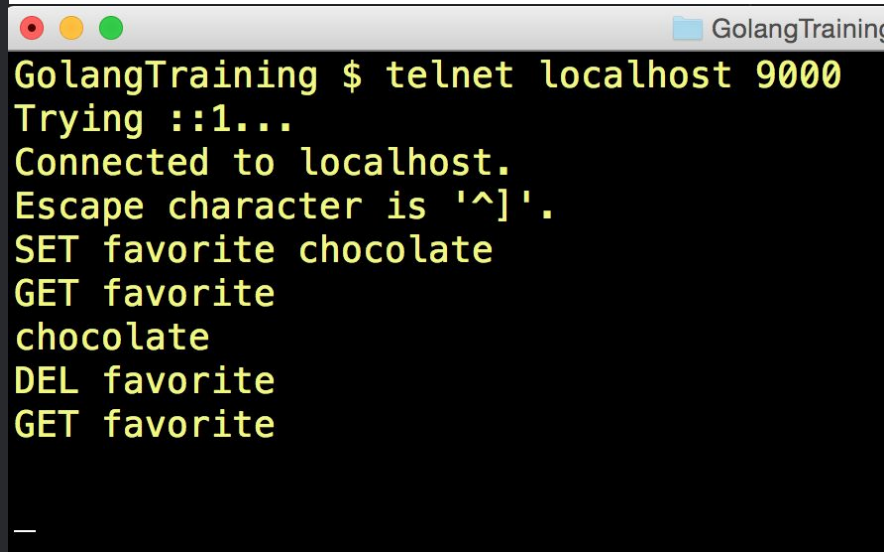

main.go - GolangTraining - [~/Documents/go/src/github.com/goestoeleven/GolangTraining]
GolangTraining 41_TCP 05_redis-clone i01 main.go
whatetereres
Project
main.go
24_embedded-types
25_interfaces
26_package-os
27_package-strings
28_package-bufio
29_package-io
30_package-ioutil
31_package-encoding-csv
32_package-path-filepath
33_package-time
34_hash
35_package-filepath
36_concurrency
37_review-exercises
38_JSON
39_packages
40_testing
41_TCP
01
02_listen
03_dial
04_echo-server
05_redis-clone
i01
i01_notes.txt
main.go
i02
i03
i04
i05_code-issue
i06
"net"
"log"
"bufio"
"fmt"
)
func handle(conn net.Conn) {
defer conn.Close()
scanner := bufio.NewScanner(conn)
for scanner.Scan() {
ln := scanner.Text()
fmt.Println(ln)
}
}
func main() {
li, err := net.Listen("tcp", ":9000")
if err != nil{
log.Fatalln(err)
}
defer li.Close()
for {
conn, err := li.Accept()
if err != nil {
log.Fatalln(err)
}
handle(conn)
Terminal
i01 $ go run main.go
echo
type here as someone dialing in and it shows up where i
GolangTraining — telnet — 80×24
GolangTraining $ telnet localhost 9000
Trying ::1...
Connected to localhost.
Escape character is '^]'.
echo
type here as someone dialing in and it shows up where it's running


GolangTraining
41_TCP
05_redis-clone
i02
main.go
Project
main.go
24_embedded-types
25_interfaces
26_package-os
27_package-strings
28_package-bufio
29_package-io
30_package-ioutil
31_package-encoding-csv
32_package-path-filepath
33_package-time
34_hash
35_package-filepath
36_concurrency
37_review-exercises
38_JSON
39_packages
40_testing
41_TCP
01
02_listen
03_dial
04_echo-server
05_redis-clone
i01
i02
i02_notes.txt
main.go
i03
i04
i05_code-issue
i06
uu_lynda
vv99_trial
ww100_whateveah
xx_exercies-for-later
xx_stringer
.gitignore
README.md
func handle(conn net.Conn) {
defer conn.Close()
scanner := bufio.NewScanner(conn)
for scanner.Scan() {
ln := scanner.Text()
fs := strings.Fields(ln)
// skip blank lines
if len(fs) < 1 {
continue
}
switch fs[0] {
case "GET":
case "SET":
case "DEL":
default:
io.WriteString(conn, "INVALID COMMAND "+fs[0]+"\n")
}
}
}
func main() {
li, err := net.Listen("tcp", ":9000")
if err != nil {
log.Fatalln(err)
}
defer li.Close()
for {
conn, err := li.Accept()
if err != nil {
log.Fatalln(err)
}
handle(conn)
}
}
Terminal
i02 $ go run main.go
GolangTraini
GolangTraining $ telnet localhost 9000
Trying ::1...
Connected to localhost.
Escape character is '^]'.
SET favorite chocolate
nothing
INVALID COMMAND nothing

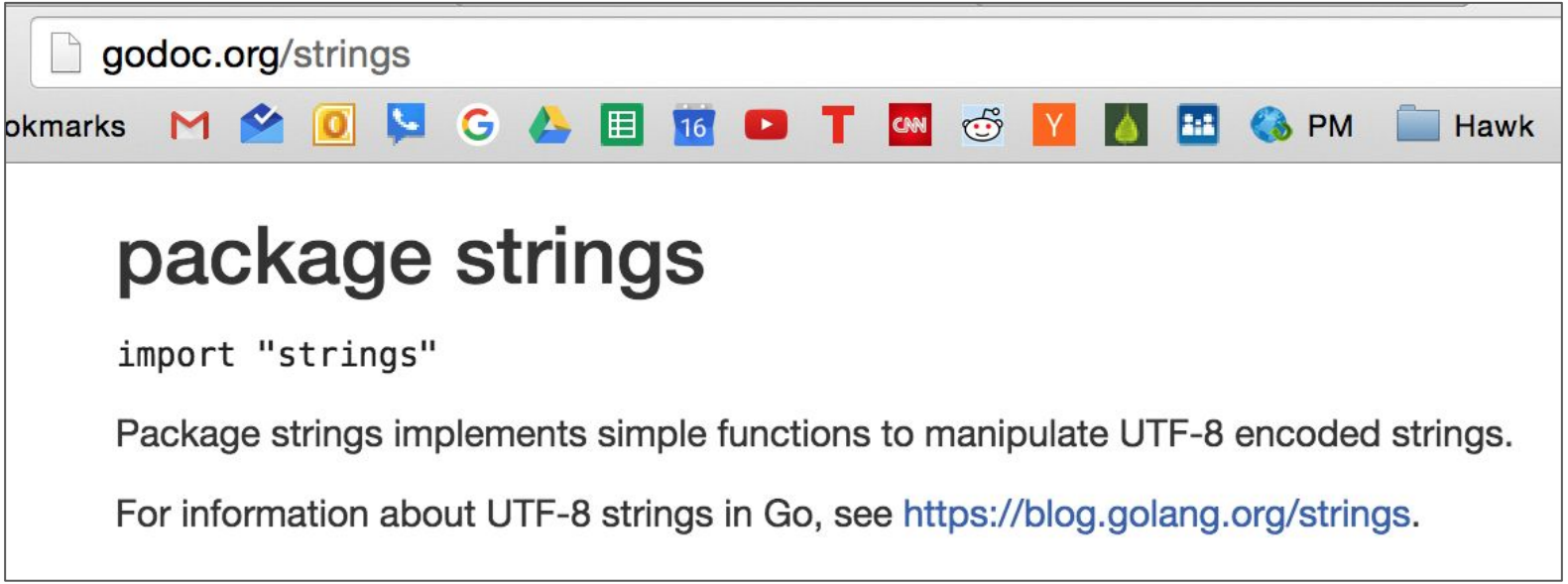
```go
var data = make(map[string]string)

func handle(conn net.Conn) {
    defer conn.Close()

    scanner := bufio.NewScanner(conn)
    for scanner.Scan() {
        ln := scanner.Text()
        fs := strings.Fields(ln)
        // skip blank lines
        if len(fs) < 2 {
            continue
        }

        switch fs[0] {
        case "GET":
            key := fs[1]
            value := data[key]
            fmt.Fprintf(conn, "%s\n", value)
        case "SET":
            if len(fs) != 3 {
                io.WriteString(conn, "EXPECTED VALUE\n")
                continue
            }
            key := fs[1]
            value := fs[2]
            data[key] = value
        case "DEL":
        default:
            io.WriteString(conn, "INVALID COMMAND "+fs[0]+"\n")
        }
    }
}
```



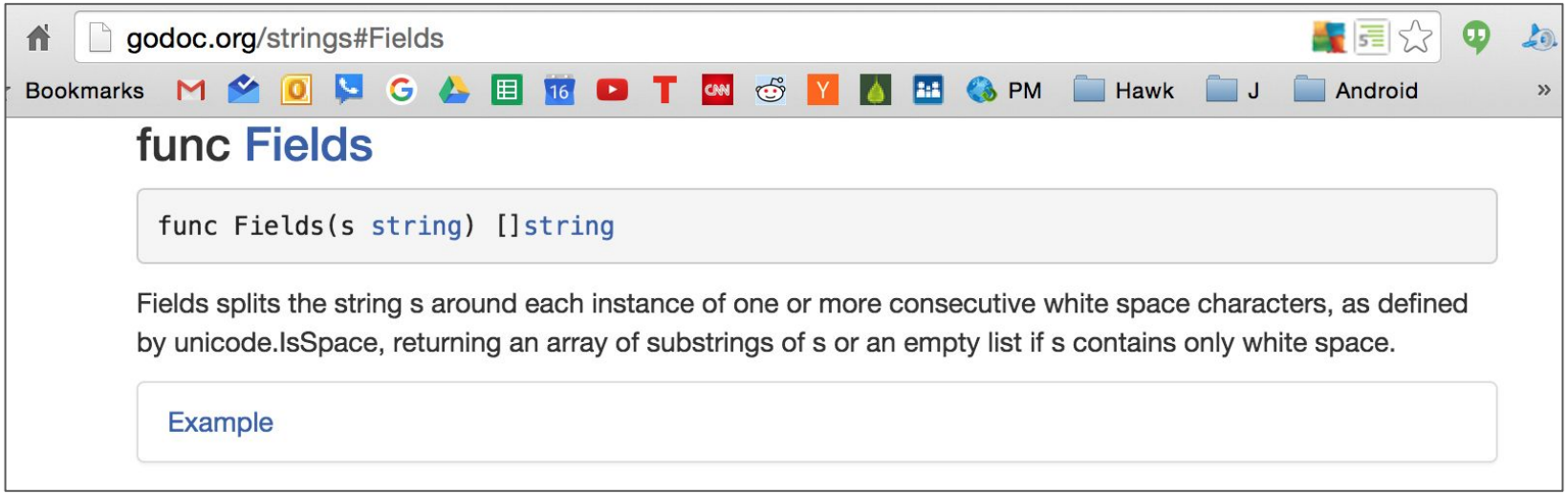
```go
var data = make(map[string]string)

func handle(conn net.Conn) {
    defer conn.Close()

    scanner := bufio.NewScanner(conn)
    for scanner.Scan() {
        ln := scanner.Text()
        fs := strings.Fields(ln)
        // skip blank lines
        if len(fs) < 2 {
            continue
        }

        switch fs[0] {
        case "GET":
            key := fs[1]
            value := data[key]
            fmt.Fprintf(conn, "%s\n", value)
        case "SET":
            if len(fs) != 3 {
                io.WriteString(conn, "EXPECTED VALUE\n")
                continue
            }
            key := fs[1]
            value := fs[2]
            data[key] = value
        case "DEL":
            key := fs[1]
            delete(data, key)
        default:
            io.WriteString(conn, "INVALID COMMAND "+fs[0]+"\n")
        }
    }
}
```



package strings


godoc.org/strings
okmarks
PM
Hawk
package strings
import "strings"
Package strings implements simple functions to manipulate UTF-8 encoded strings.
For information about UTF-8 strings in Go, see https://blog.golang.org/strings.


godoc.org/strings#NewReader
marks
PM
Hawk
J
Android
GO
func NewReader
func NewReader(s string) *Reader
NewReader returns a new Reader reading from s. It is similar to bytes.NewBufferString but more efficient and read-only.


godoc.org/strings#Fields
Bookmarks
PM
Hawk
J
Android
func Fields
func Fields(s string) []string
Fields splits the string s around each instance of one or more consecutive white space characters, as defined by unicode.IsSpace, returning an array of substrings of s or an empty list if s contains only white space.
Example

Example

Code: play

```
fmt.Printf("Fields are: %q", strings.Fields("  foo bar  baz   "))
```

Output:

```
Fields are: ["foo" "bar" "baz"]
```

```go
var data = make(map[string]string)

func handle(conn net.Conn) {
    defer conn.Close()

    scanner := bufio.NewScanner(conn)
    for scanner.Scan() {
        ln := scanner.Text()
        fs := strings.Fields(ln)
        // skip blank lines
        if len(fs) < 2 {
            continue
        }

        switch fs[0] {
        case "GET":
            key := fs[1]
            value := data[key]
            fmt.Fprintf(conn, "%s\n", value)
        case "SET":
            if len(fs) != 3 {
                io.WriteString(conn, "EXPECTED VALUE\n")
                continue
            }
            key := fs[1]
            value := fs[2]
            data[key] = value
        case "DEL":
            key := fs[1]
            delete(data, key)
        default:
            io.WriteString(conn, "INVALID COMMAND "+fs[0]+"\n")
        }
    }
}
```

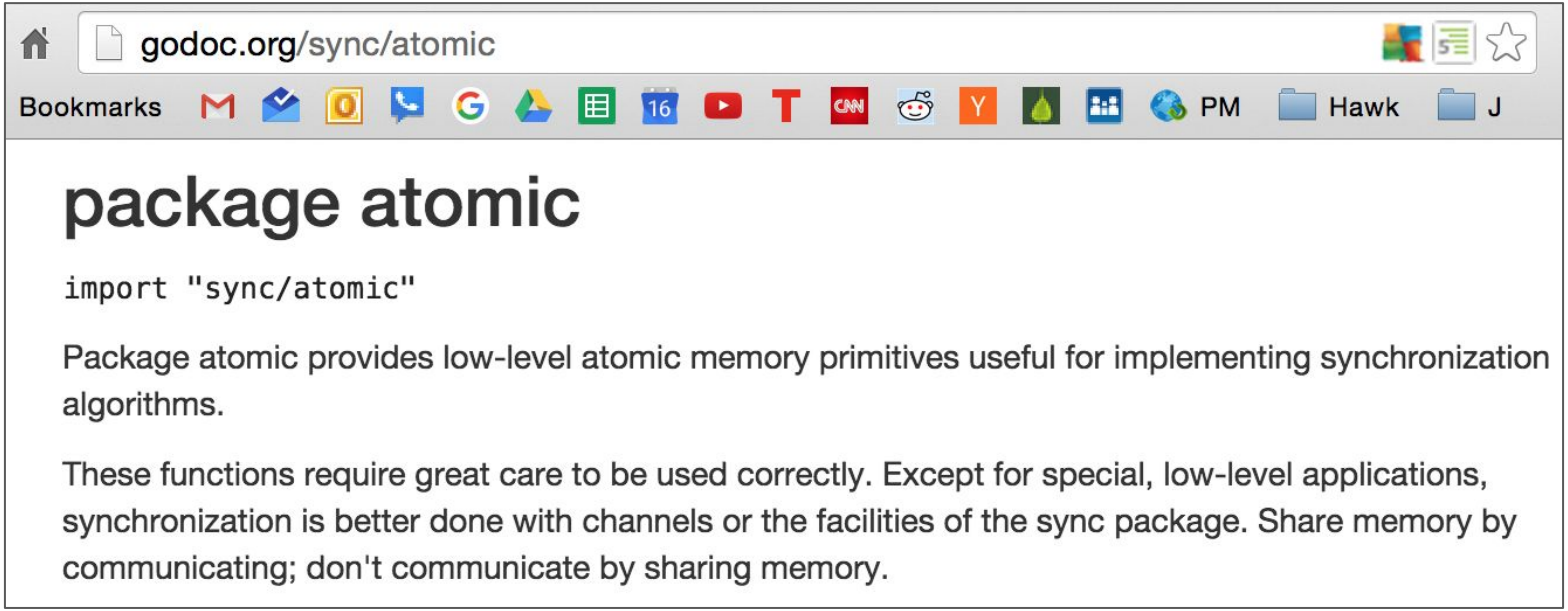
Continuing with this example ...

```
GolangTraining $ telnet localhost 9000
Trying ::1...
Connected to localhost.
Escape character is '^]'.
SET favorite chocolate
GET favorite
chocolate
DEL favorite
GET favorite
```

```go
func main() {
    li, err := net.Listen("tcp", ":9000")
    if err != nil {
        log.Fatalln(err)
    }
    defer li.Close()

    for {
        conn, err := li.Accept()
        if err != nil {
            log.Fatalln(err)
        }
        handle(conn)
        // ONLY HANDLES ONE CONNECTION AT A TIME
        // Could we make it concurrent? How?
        // What are the considerations?
    }
}
```

Here is our func main which calls handle

atomicity, mutexes, concurrency

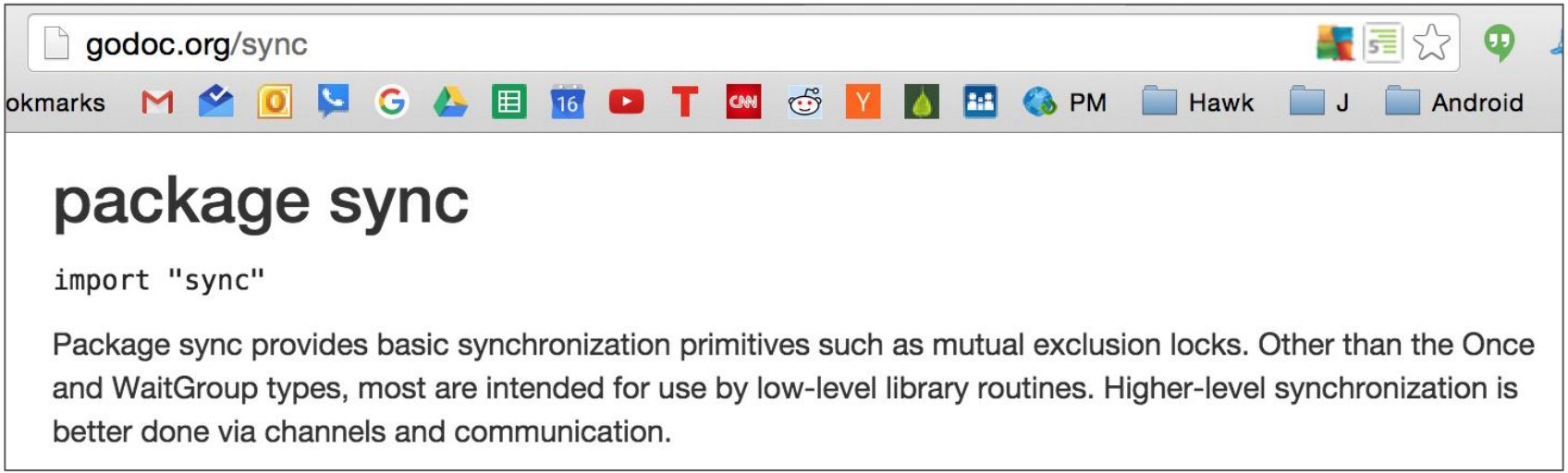


package atomic

```
import "sync/atomic"
```

Package atomic provides low-level atomic memory primitives useful for implementing synchronization algorithms.

These functions require great care to be used correctly. Except for special, low-level applications, synchronization is better done with channels or the facilities of the sync package. Share memory by communicating; don't communicate by sharing memory.


godoc.org/sync
okmarks
PM
Hawk
J
Android
package sync
import "sync"
Package sync provides basic synchronization primitives such as mutual exclusion locks. Other than the Once and WaitGroup types, most are intended for use by low-level library routines. Higher-level synchronization is better done via channels and communication.

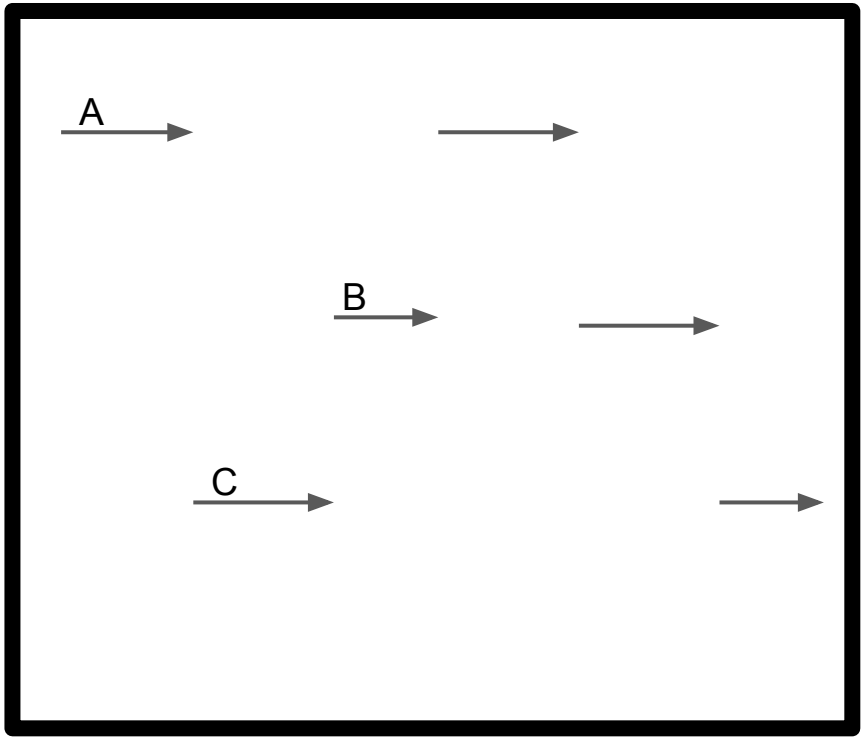
concurrency vs parallelism

concurrency and parallelism

“**Concurrency** is the composition of independently executing processes, while **parallelism** is the simultaneous execution of (possibly related) computations. **Concurrency** is about dealing with lots of things at once. **Parallelism** is about doing lots of things at once.”

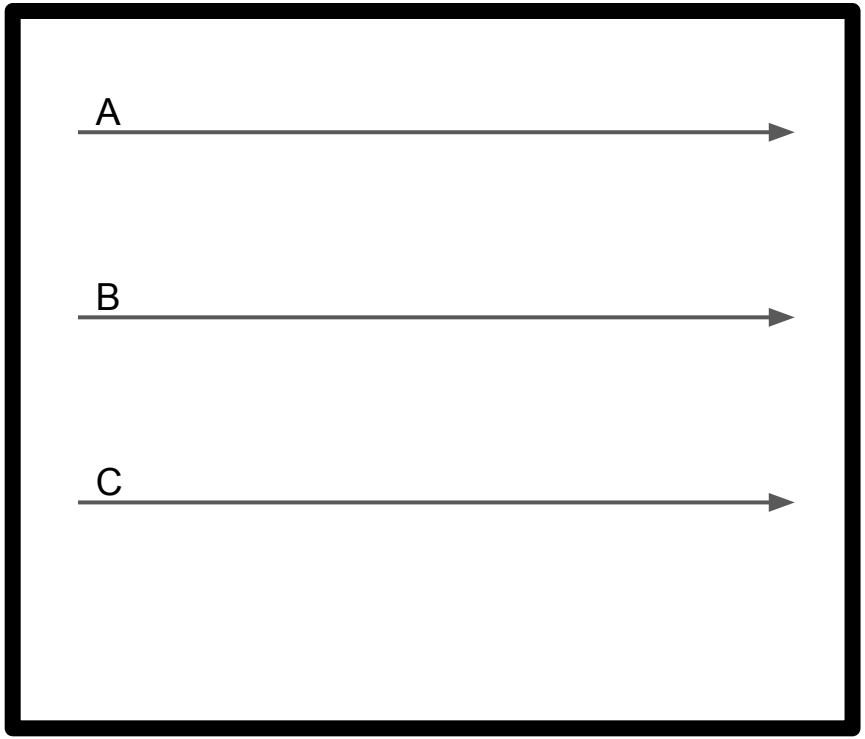
concurrency

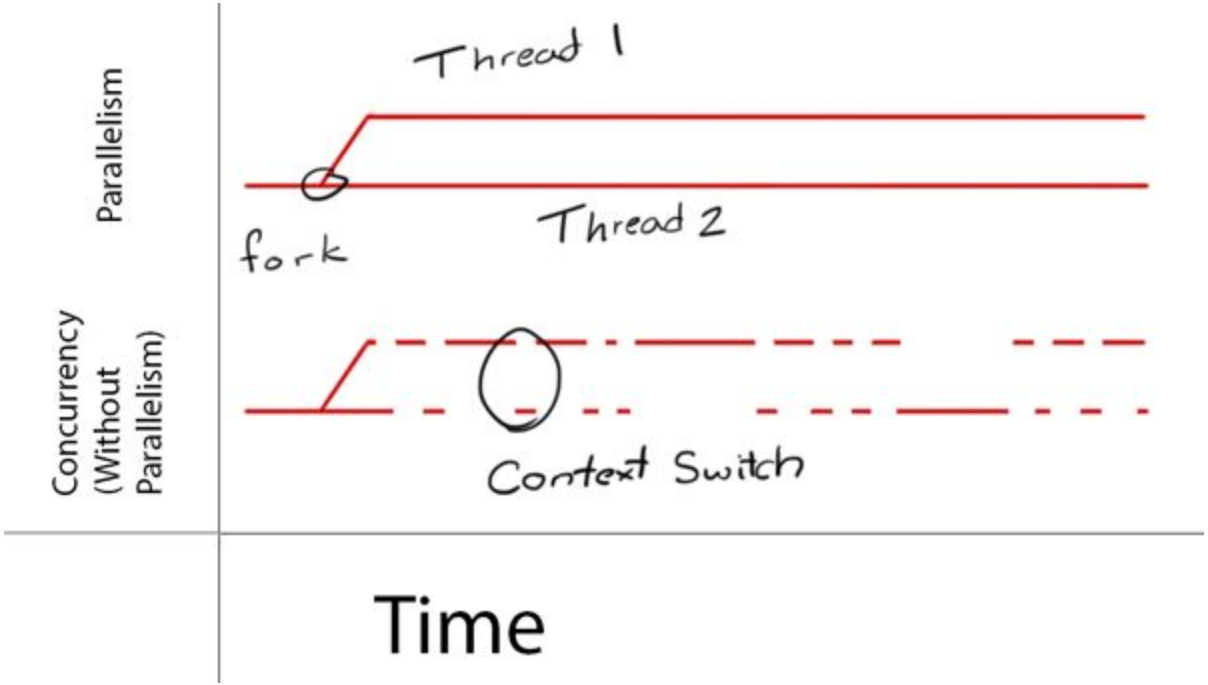
doing many things, but only one at a time
“multitasking”



parallelism

doing many things at the same time





http://cdn1.raywenderlich.com/wp-content/uploads/2014/01/Concurrency_vs_Parallelism.png

Continuing with our example …
adding concurrency

```go
func main() {
    li, err := net.Listen("tcp", ":9000")
    if err != nil {
        log.Fatalln(err)
    }
    defer li.Close()

    for {
        conn, err := li.Accept()
        if err != nil {
            log.Fatalln(err)
        }
        handle(conn)
        // ONLY HANDLES ONE CONNECTION AT A TIME
        // Could we make it concurrent? How?
        // What are the considerations?
    }
}
```

Here is our func main before concurrency

adding concurrency

```
func main() {
    li, err := net.Listen("tcp", ":9000")
    if err != nil {
        log.Fatalln(err)
    }
    defer li.Close()

    commands := make(chan Command)
    go redisServer(commands)

    for {
        conn, err := li.Accept()
        if err != nil {
            log.Fatalln(err)
        }

        go handle(commands, conn)
    }
}
```

adding concurrency

```go
func main() {
    li, err := net.Listen("tcp", ":9000")
    if err != nil {
        log.Fatalln(err)
    }
    defer li.Close()

    commands := make(chan Command)
    go redisServer(commands)

    for {
        conn, err := li.Accept()
        if err != nil {
            log.Fatalln(err)
        }

        go handle(commands, conn)
    }
}
```

```go
11
12  type Command struct {
13      Fields []string
14      Result chan string
15  }
16
```

https://gobyexample.com/goroutines

```go
func redisServer(commands chan Command) {
    var data = make(map[string]string)
    for cmd := range commands {
        if len(cmd.Fields) < 2 {
            cmd.Result <- "Expected at least 2 arguments"
            continue
        }

        fmt.Println("GOT COMMAND", cmd)

        switch cmd.Fields[0] {
        // GET <KEY>
        case "GET":
            key := cmd.Fields[1]
            value := data[key]

            cmd.Result <- value

        // SET <KEY> <VALUE>
        case "SET":
            if len(cmd.Fields) != 3 {
                cmd.Result <- "EXPECTED VALUE"
                continue
            }
            key := cmd.Fields[1]
            value := cmd.Fields[2]
            data[key] = value
            cmd.Result <- ""
        // DEL <KEY>
        case "DEL":
            key := cmd.Fields[1]
            delete(data, key)
            cmd.Result <- ""
        default:
            cmd.Result <- "INVALID COMMAND " + cmd.Fields[0] + "\n"
        }
    }
}
```

https://gobyexample.com/range-over-channels

(next slide please)

```go
func redisServer(commands chan Command) {
    var data = make(map[string]string)
    for cmd := range commands {
        if len(cmd.Fields) < 2 {
            cmd.Result <- "Expected at least 2 arguments"
            continue
        }

        fmt.Println("GOT COMMAND", cmd)

        switch cmd.Fields[0] {
        // GET <KEY>
        case "GET":
            key := cmd.Fields[1]
            value := data[key]

            cmd.Result <- value

        // SET <KEY> <VALUE>
        case "SET":
            if len(cmd.Fields) != 3 {
                cmd.Result <- "EXPECTED VALUE"
                continue
            }
            key := cmd.Fields[1]
            value := cmd.Fields[2]
            data[key] = value
            cmd.Result <- ""
        // DEL <KEY>
        case "DEL":
            key := cmd.Fields[1]
            delete(data, key)
            cmd.Result <- ""
        default:
            cmd.Result <- "INVALID COMMAND " + cmd.Fields[0] + "\n"
        }
    }
}
```

https://gobyexample.com/range-over-channels

https://gobyexample.com/channels

(next slide please)

```go
func redisServer(commands chan Command) {
    var data = make(map[string]string)
    for cmd := range commands {
        if len(cmd.Fields) < 2 {
            cmd.Result <- "Expected at least 2 arguments"
            continue
        }

        fmt.Println("GOT COMMAND", cmd)

        switch cmd.Fields[0] {
        // GET <KEY>
        case "GET":
            key := cmd.Fields[1]
            value := data[key]

            cmd.Result <- value

        // SET <KEY> <VALUE>
        case "SET":
            if len(cmd.Fields) != 3 {
                cmd.Result <- "EXPECTED VALUE"
                continue
            }
            key := cmd.Fields[1]
            value := cmd.Fields[2]
            data[key] = value
            cmd.Result <- ""
        // DEL <KEY>
        case "DEL":
            key := cmd.Fields[1]
            delete(data, key)
            cmd.Result <- ""
        default:
            cmd.Result <- "INVALID COMMAND " + cmd.Fields[0] + "\n"
        }
    }
}
```

https://gobyexample.com/range-over-channels

https://gobyexample.com/channels

```go
type Command struct {
    Fields []string
    Result chan string
}
```

(next slide please)

```go
func redisServer(commands chan Command) {
    var data = make(map[string]string)
    for cmd := range commands {
        if len(cmd.Fields) < 2 {
            cmd.Result <- "Expected at least 2 arguments"
            continue
        }

        fmt.Println("GOT COMMAND", cmd)

        switch cmd.Fields[0] {
        // GET <KEY>
        case "GET":
            key := cmd.Fields[1]
            value := data[key]

            cmd.Result <- value

        // SET <KEY> <VALUE>
        case "SET":
            if len(cmd.Fields) != 3 {
                cmd.Result <- "EXPECTED VALUE"
                continue
            }
            key := cmd.Fields[1]
            value := cmd.Fields[2]
            data[key] = value
            cmd.Result <- ""
        // DEL <KEY>
        case "DEL":
            key := cmd.Fields[1]
            delete(data, key)
            cmd.Result <- ""
        default:
            cmd.Result <- "INVALID COMMAND " + cmd.Fields[0] + "\n"
        }
    }
}
```

https://gobyexample.com/range-over-channels

https://gobyexample.com/channels

```go
type Command struct {
    Fields []string
    Result chan string
}
```

(next slide please)

```go
func redisServer(commands chan Command) {
    var data = make(map[string]string)
    for cmd := range commands {
        if len(cmd.Fields) < 2 {
            cmd.Result <- "Expected at least 2 arguments"
            continue
        }

        fmt.Println("GOT COMMAND", cmd)

        switch cmd.Fields[0] {
        // GET <KEY>
        case "GET":
            key := cmd.Fields[1]
            value := data[key]

            cmd.Result <- value

        // SET <KEY> <VALUE>
        case "SET":
            if len(cmd.Fields) != 3 {
                cmd.Result <- "EXPECTED VALUE"
                continue
            }
            key := cmd.Fields[1]
            value := cmd.Fields[2]
            data[key] = value
            cmd.Result <- ""
        // DEL <KEY>
        case "DEL":
            key := cmd.Fields[1]
            delete(data, key)
            cmd.Result <- ""
        default:
            cmd.Result <- "INVALID COMMAND " + cmd.Fields[0] + "\n"
        }
    }
}
```

```go

type Command struct {
    Fields []string
    Result chan string
}

```

```go

func handle(commands chan Command, conn net.Conn) {
    defer conn.Close()

    scanner := bufio.NewScanner(conn)
    for scanner.Scan() {
        ln := scanner.Text()
        fs := strings.Fields(ln)

        result := make(chan string)
        commands <- Command{
            Fields: fs,
            Result: result,
        }

        io.WriteString(conn, <-result+"\n")
    }

}

func main() {
    li, err := net.Listen("tcp", ":9000")
    if err != nil {
        log.Fatalln(err)
    }
    defer li.Close()

    commands := make(chan Command)
    go redisServer(commands)

    for {
        conn, err := li.Accept()
        if err != nil {
            log.Fatalln(err)
        }

        go handle(commands, conn)
    }
}
```

(next slide please)

```go
func redisServer(commands chan Command) {
    var data = make(map[string]string)
    for cmd := range commands {
        if len(cmd.Fields) < 2 {
            cmd.Result <- "Expected at least 2 arguments"
            continue
        }

        fmt.Println("GOT COMMAND", cmd)

        switch cmd.Fields[0] {
        // GET <KEY>
        case "GET":
            key := cmd.Fields[1]
            value := data[key]

            cmd.Result <- value

        // SET <KEY> <VALUE>
        case "SET":
            if len(cmd.Fields) != 3 {
                cmd.Result <- "EXPECTED VALUE"
                continue
            }
            key := cmd.Fields[1]
            value := cmd.Fields[2]
            data[key] = value
            cmd.Result <- ""
        // DEL <KEY>
        case "DEL":
            key := cmd.Fields[1]
            delete(data, key)
            cmd.Result <- ""
        default:
            cmd.Result <- "INVALID COMMAND " + cmd.Fields[0] + "\n"
        }
    }
}
```

```go

type Command struct {
    Fields []string
    Result chan string
}

```

```go

func handle(commands chan Command, conn net.Conn) {
    defer conn.Close()

    scanner := bufio.NewScanner(conn)
    for scanner.Scan() {
        ln := scanner.Text()
        fs := strings.Fields(ln)

        result := make(chan string)
        commands <- Command{
            Fields: fs,
            Result: result,
        }

        io.WriteString(conn, <-result+"\n")
    }

}

func main() {
    li, err := net.Listen("tcp", ":9000")
    if err != nil {
        log.Fatalln(err)
    }
    defer li.Close()

    commands := make(chan Command)
    go redisServer(commands)

    for {
        conn, err := li.Accept()
        if err != nil {
            log.Fatalln(err)
        }

        go handle(commands, conn)
    }
}
```

(next slide please)

```go
func redisServer(commands chan Command) {
    var data = make(map[string]string)
    for cmd := range commands {
        if len(cmd.Fields) < 2 {
            cmd.Result <- "Expected at least 2 arguments"
            continue
        }

        fmt.Println("GOT COMMAND", cmd)

        switch cmd.Fields[0] {
        // GET <KEY>
        case "GET":
            key := cmd.Fields[1]
            value := data[key]

            cmd.Result <- value

        // SET <KEY> <VALUE>
        case "SET":
            if len(cmd.Fields) != 3 {
                cmd.Result <- "EXPECTED VALUE"
                continue
            }
            key := cmd.Fields[1]
            value := cmd.Fields[2]
            data[key] = value
            cmd.Result <- ""
        // DEL <KEY>
        case "DEL":
            key := cmd.Fields[1]
            delete(data, key)
            cmd.Result <- ""
        default:
            cmd.Result <- "INVALID COMMAND " + cmd.Fields[0] + "\n"
        }
    }
}
```

```go

type Command struct {
    Fields []string
    Result chan string
}

```

```go

func handle(commands chan Command, conn net.Conn) {
    defer conn.Close()

    scanner := bufio.NewScanner(conn)
    for scanner.Scan() {
        ln := scanner.Text()
        fs := strings.Fields(ln)

        result := make(chan string)
        commands <- Command{
            Fields: fs,
            Result: result,
        }

        io.WriteString(conn, <-result+"\n")
    }

}

func main() {
    li, err := net.Listen("tcp", ":9000")
    if err != nil {
        log.Fatalln(err)
    }
    defer li.Close()

    commands := make(chan Command)
    go redisServer(commands)

    for {
        conn, err := li.Accept()
        if err != nil {
            log.Fatalln(err)
        }

        go handle(commands, conn)
    }
}
```

Do this exercise
even if you're only retyping code
build it up step-by-step
understand each step

exercise

Create a simplified redis clone which accepts GET, SET and DEL commands. 'GET <KEY>' should write the value of <KEY> followed by a new line. 'SET <KEY> <VALUE>' should set the value of <KEY>. 'DEL <KEY>' should remove the value. Data should be stored in memory.

redis

- nosql database
- stored in memory
  - basically a map
    - key, value pairs
  - very fast
- people use redis like cache
  - in conjunction with a sql server
  - load what you’re using into redis
    - it’s super fast

redis

- nosql database
- stored in memory
  - basically a map
    - key, value pairs
  - very fast
- people use redis like cache
  - in conjunction with a sql server
  - load what you're using into redis
    - it's super fast

How do you scale this?

scaling

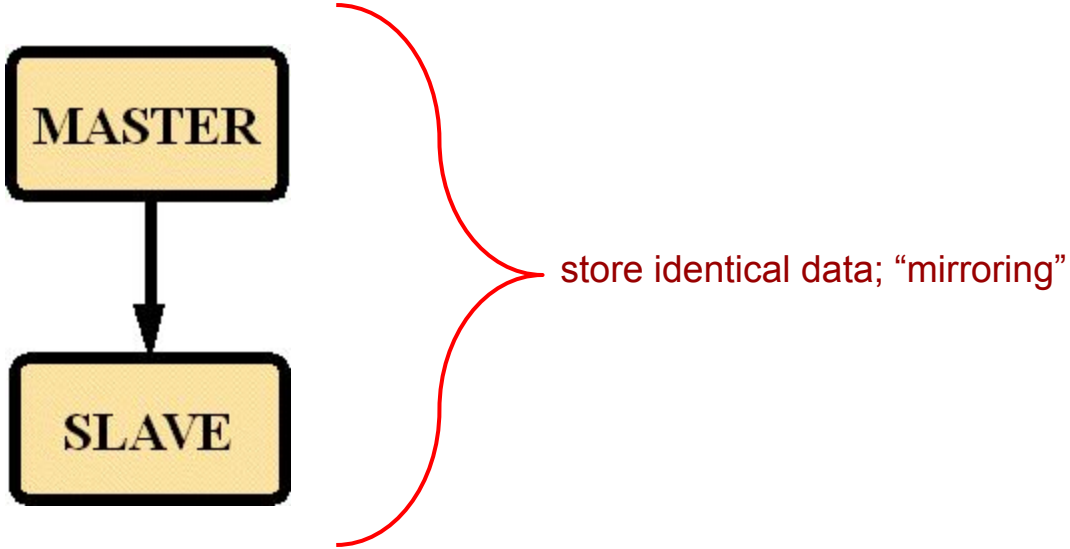
- Vertical
  - more memory in one computer
- Horizontal
  - more computers
  - partition your data with “hashing into buckets”
    - <hash %2> for two computers
    - <hash %10> for 10 computers
      - 11 % 10 → store in computer 1
      - 12 % 10 → store in computer 2
      - 73 % 10 → store in computer 3
      - 134 % 10 → store in computer 4
      - 1095 % 10 → store in computer 5
      - ….

redis

- nosql database
- stored in memory
  - basically a map
    - key, value pairs
  - very fast
- people use redis like cache
  - in conjunction with a sql server
  - load what you're using into redis
    - it's super fast

How do you scale this?
What happens if the power goes out?


{MASTER}
{slave}


MASTER
SLAVE
store identical data; “mirroring”

CAP Theorem

https://en.wikipedia.org/wiki/CAP_theorem

exercise

Modify ‘echo’ so that it returns messages as rot 13

optional exercise

Create a chat room server. A client can connect and send messages to the server. Those messages will be broadcast to any other currently connected clients.