Level 2–1

Underneath the Tracks

Variables & Types

ON TRACK with GOLANG

Repetitive References

We are currently referencing the same os.Args array from two different places. This is the start of what could become a code smell and unnecessary repetition.

```
package main

import
...

func main() {
  if len(os.Args) > 1 {
    fmt.Println(os.Args[1])
  } else {
    fmt.Println("Hello, Gopher")
  }
}
```

Multiple references to the same array

Declaring Variables With Type Inference

The `:=` operator tells Go to **automatically find out the data type** on the right being assigned to the **newly declared variable** on the left. This is known as **type inference.**

```
package main

import
...

func main() {
  args := os.Args
  if len(args) > 1 {
    fmt.Println(args[1])
  } else {
    fmt.Println("Hello, Gopher")
  }
}
```

Automatically finds out data type for the value returned from os.Args

Storing Values as Data Types

Given a value, we must determine **how much space** is needed to store this value for later reuse.

A value, like 1 *or* "hello"

HERE'S AN ACORN...

A data type, like int *or* string

...AND SOME AVAILABLE STORAGE SPACE: A BASKET.

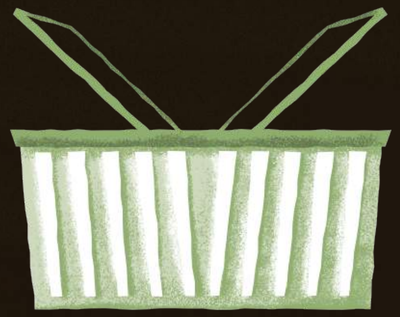
Assigning a value to a variable of a specific data type

THE BASKET CAN PERFECTLY ACCOMMODATE THE ACORN.

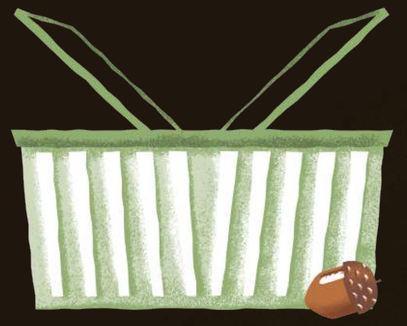
One Size Does Not Fit All

Taking up too much storage is a **waste of precious space**. On the other hand, not reserving enough space can limit the amount of data we can store.

HERE'S A DIFFERENT ACORN...



...AND MORE STORAGE THAN NEEDED.



OUR ACORN FITS IN THE BASKET, BUT THERE'S A LOT OF SPACE LEFT UNUSED!

Using more memory than necessary!

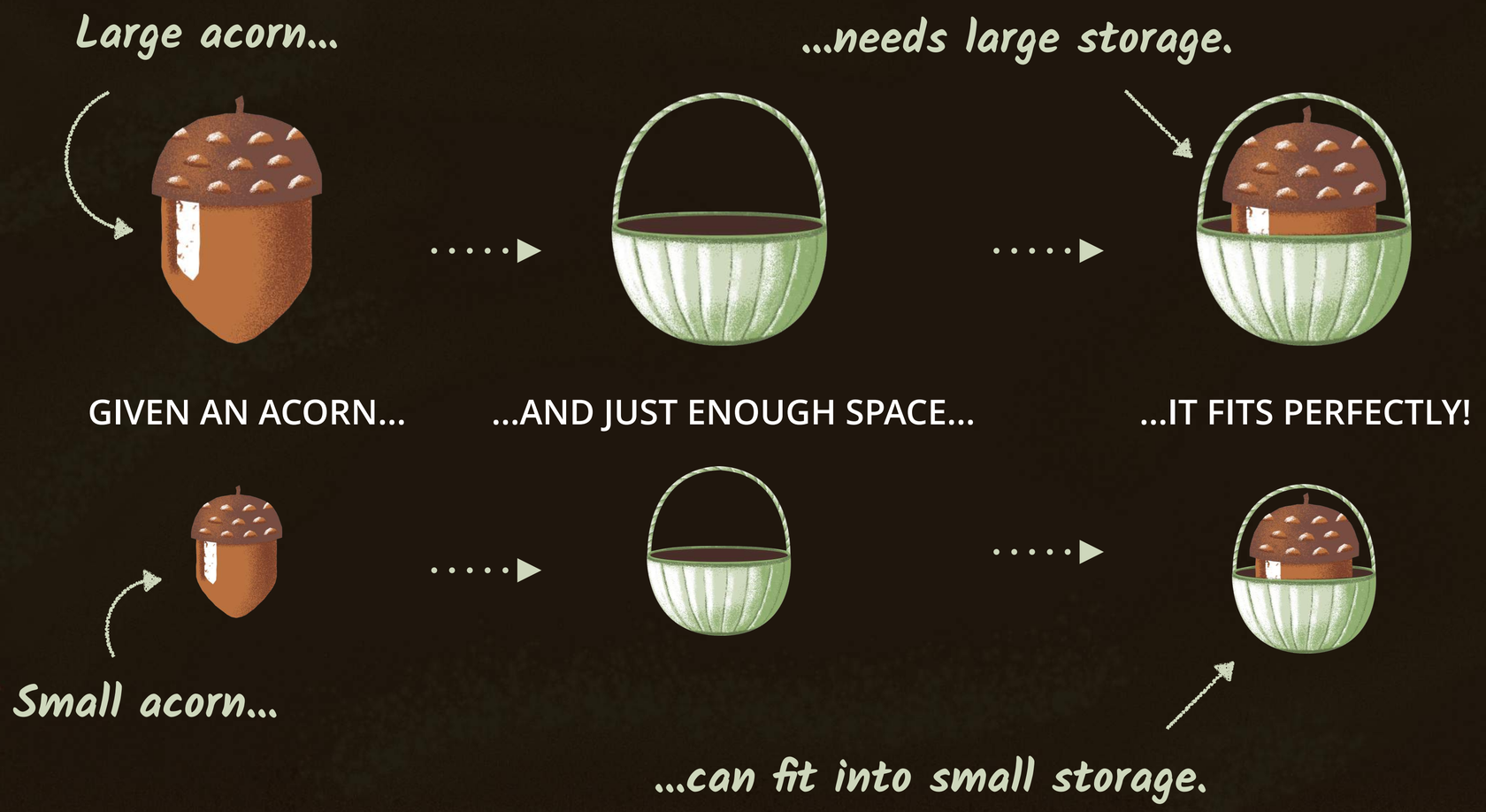
...AND NOT ENOUGH STORAGE SPACE.

CAN'T FIT!

Not enough memory for storage

Storing Values as Data Types

For every value, there's always a **proper data type** that determines how the value should be stored and the operations that can be done on values of that type.



Type Inference vs. Manual Type Declaration

There are **two ways** to declare variables in Go: **type inference** and **manual type declaration.**

Type Inference

Data type is inferred during assignment.

```
<variable-name> := <some-value>
```

Notice the special := operator.

The most common and preferred way

```
message := "Hello, Gopher"
```

Manual Type Declaration

Data type is declared prior to assignment.

```
var <variable-name> <data-type>
<variable-name> = <some-value>
```

The var keyword...

...and the = operator.

More verbose, but useful and often necessary

```
var message string
message = "Hello, Gopher"
```

How Static Typing Can Help

Static typing allows the Go compiler to check for type errors **before the program is run.**

Assigning a different value than what was expected makes the Go compiler mad... 42 is NOT a string!

```
var age string
age = 42
```

```
./main.go:5: cannot use 42 (type int)
                as type string in assignment
```

No errors when the correct data type is used... 42 is an int!

```
var age int
age = 42
```

Type Inference Requires Less Code

Most times, type inference and manual type declaration **can be used interchangeably,** but type inference is **less code to write and read.**

```
package main

import
...

func main() {
  args := os.Args
  if len(args) > 1 {
    fmt.Println(args[1])
  } else {
    fmt.Println("Hello, Gopher")
  }
}
```

Same thing

Manually declaring type

```
var args []string
args = os.Args
```

Brackets prefix indicates this is a collection of strings.

Common Data Types

Here are a few built-in data types commonly found in most Go programs.

Type	Data
int	42
string	"Hello"
bool	true or false
[]string	["acorn", "basket"]

Also called primitive data values