Maps

You can find all the code for this chapter here

In arrays & slices, you saw how to store values in order. Now, we will look at a way to store items by a `key` and look them up quickly.

Maps allow you to store items in a manner similar to a dictionary. You can think of the `key` as the word and the `value` as the definition. And what better way is there to learn about Maps than to build our own dictionary?

First, assuming we already have some words with their definitions in the dictionary, if we search for a word, it should return the definition of it.

Write the test first

In `dictionary_test.go`

```go
package main

import "testing"

func TestSearch(t *testing.T) {
	dictionary := map[string]string{"test": "this is just a test"}

	got := Search(dictionary, "test")
	want := "this is just a test"

	if got != want {
		t.Errorf("got %q want %q given, %q", got, want, "test")
	}
}
```

Declaring a Map is somewhat similar to an array. Except, it starts with the `map` keyword and requires two types. The first is the key type, which is written inside the `[]`. The second is the value type, which goes right after the `[]`.

The key type is special. It can only be a comparable type because without the ability to tell if 2 keys are equal, we have no way to ensure that we are getting the correct value. Comparable types are explained in depth in the language spec.

The value type, on the other hand, can be any type you want. It can even be another map.

Everything else in this test should be familiar.

Try to run the test

By running `go test` the compiler will fail with `./dictionary_test.go:8:9: undefined: Search`.

Write the minimal amount of code for the test to run and check the output

In `dictionary.go`

```go
package main

func Search(dictionary map[string]string, word string) string {
	return ""
}
```

Your test should now fail with a *clear error message*

`dictionary_test.go:12: got '' want 'this is just a test' given, 'test'`.

Write enough code to make it pass

```go
func Search(dictionary map[string]string, word string) string {
	return dictionary[word]
}
```

Getting a value out of a Map is the same as getting a value out of Array `map[key]`.

Refactor

```go
func TestSearch(t *testing.T) {
	dictionary := map[string]string{"test": "this is just a test"}

	got := Search(dictionary, "test")
	want := "this is just a test"

	assertStrings(t, got, want)
}

func assertStrings(t testing.TB, got, want string) {
	t.Helper()

	if got != want {
		t.Errorf("got %q want %q", got, want)
	}
}
```

I decided to create an `assertStrings` helper to make the implementation more general.

Using a custom type

We can improve our dictionary's usage by creating a new type around map and making `Search` a method.

In `dictionary_test.go`:

```go
func TestSearch(t *testing.T) {
	dictionary := Dictionary{"test": "this is just a test"}

	got := dictionary.Search("test")
	want := "this is just a test"

```

```
        assertStrings(t, got, want)
}
```

We started using the `Dictionary` type, which we have not defined yet. Then called `Search` on the `Dictionary` instance.

We did not need to change `assertStrings`.

In `dictionary.go`:

```go
type Dictionary map[string]string

func (d Dictionary) Search(word string) string {
        return d[word]
}
```

Here we created a `Dictionary` type which acts as a thin wrapper around `map`. With the custom type defined, we can create the `Search` method.

Write the test first

The basic search was very easy to implement, but what will happen if we supply a word that's not in our dictionary?

We actually get nothing back. This is good because the program can continue to run, but there is a better approach. The function can report that the word is not in the dictionary. This way, the user isn't left wondering if the word doesn't exist or if there is just no definition (this might not seem very useful for a dictionary. However, it's a scenario that could be key in other usecases).

```go
func TestSearch(t *testing.T) {
        dictionary := Dictionary{"test": "this is just a test"}

        t.Run("known word", func(t *testing.T) {
                got, _ := dictionary.Search("test")
                want := "this is just a test"

                assertStrings(t, got, want)
        })
```

```go

	t.Run("unknown word", func(t *testing.T) {
		_, err := dictionary.Search("unknown")
		want := "could not find the word you were looking for"

		if err == nil {
			t.Fatal("expected to get an error.")
		}

		assertStrings(t, err.Error(), want)
	})
}
```

The way to handle this scenario in Go is to return a second argument which is an `Error` type.

`Error`s can be converted to a string with the `.Error()` method, which we do when passing it to the assertion. We are also protecting `assertStrings` with `if` to ensure we don't call `.Error()` on `nil`.

Try and run the test

This does not compile

```
./dictionary_test.go:18:10: assignment mismatch: 2 variables but 1 values
```

Write the minimal amount of code for the test to run and check the output

```go
func (d Dictionary) Search(word string) (string, error) {
	return d[word], nil
}
```

Your test should now fail with a much clearer error message.

`dictionary_test.go:22: expected to get an error.`

Write enough code to make it pass

```go
func (d Dictionary) Search(word string) (string, error) {
	definition, ok := d[word]
	if !ok {
		return "", errors.New("could not find the word you were looking for")
	}

	return definition, nil
}
```

In order to make this pass, we are using an interesting property of the map lookup. It can return 2 values. The second value is a boolean which indicates if the key was found successfully.

This property allows us to differentiate between a word that doesn't exist and a word that just doesn't have a definition.

Refactor

```go
var ErrNotFound = errors.New("could not find the word you were looking for")

func (d Dictionary) Search(word string) (string, error) {
	definition, ok := d[word]
	if !ok {
		return "", ErrNotFound
	}

	return definition, nil
}
```

We can get rid of the magic error in our `Search` function by extracting it into a variable. This will also allow us to have a better test.

```go
t.Run("unknown word", func(t *testing.T) {
	_, got := dictionary.Search("unknown")

	assertError(t, got, ErrNotFound)
})
```

```go
func assertError(t testing.TB, got, want error) {
	t.Helper()

	if got != want {
		t.Errorf("got error %q want %q", got, want)
	}
}
```

By creating a new helper we were able to simplify our test, and start using our `ErrNotFound` variable so our test doesn't fail if we change the error text in the future.

Write the test first

We have a great way to search the dictionary. However, we have no way to add new words to our dictionary.

```go
func TestAdd(t *testing.T) {
	dictionary := Dictionary{}
	dictionary.Add("test", "this is just a test")

	want := "this is just a test"
	got, err := dictionary.Search("test")
	if err != nil {
		t.Fatal("should find added word:", err)
	}

	if got != want {
		t.Errorf("got %q want %q", got, want)
	}
}
```

In this test, we are utilizing our `Search` function to make the validation of the dictionary a little easier.

Write the minimal amount of code for the test to run and check output

In `dictionary.go`

```go
func (d Dictionary) Add(word, definition string) {
}
```

Your test should now fail

```
dictionary_test.go:31: should find added word: could not find the word you were looki
```

Write enough code to make it pass

```go
func (d Dictionary) Add(word, definition string) {
	d[word] = definition
}
```

Adding to a map is also similar to an array. You just need to specify a key and set it equal to a value.

Pointers, copies, et al

An interesting property of maps is that you can modify them without passing as an address to it (e.g `&myMap`)

This may make them *feel* like a "reference type", but as Dave Cheney describes they are not.

> A map value is a pointer to a runtime.hmap structure.

So when you pass a map to a function/method, you are indeed copying it, but just the pointer part, not the underlying data structure that contains the data.

A gotcha with maps is that they can be a `nil` value. A `nil` map behaves like an empty map when reading, but attempts to write to a `nil` map will cause a runtime panic. You can read more about maps here.

Therefore, you should never initialize an empty map variable:

```go
var m map[string]string
```

Instead, you can initialize an empty map like we were doing above, or use the `make` keyword to create a map for you:

```go
var dictionary = map[string]string{}

// OR

var dictionary = make(map[string]string)
```

Both approaches create an empty `hash map` and point `dictionary` at it. Which ensures that you will never get a runtime panic.

Refactor

There isn't much to refactor in our implementation but the test could use a little simplification.

```go
func TestAdd(t *testing.T) {
	dictionary := Dictionary{}
	word := "test"
	definition := "this is just a test"

	dictionary.Add(word, definition)

	assertDefinition(t, dictionary, word, definition)
}

func assertDefinition(t testing.TB, dictionary Dictionary, word, definition string) {
	t.Helper()

	got, err := dictionary.Search(word)
	if err != nil {
		t.Fatal("should find added word:", err)
	}

	if definition != got {
		t.Errorf("got %q want %q", got, definition)
	}
}
```

We made variables for word and definition, and moved the definition assertion into its own helper function.

Our `Add` is looking good. Except, we didn't consider what happens when the value we are trying to add already exists!

Map will not throw an error if the value already exists. Instead, they will go ahead and overwrite the value with the newly provided value. This can be convenient in practice, but makes our function name less than accurate. `Add` should not modify existing values. It should only add new words to our dictionary.

Write the test first

```go
func TestAdd(t *testing.T) {
	t.Run("new word", func(t *testing.T) {
		dictionary := Dictionary{}
		word := "test"
		definition := "this is just a test"

		err := dictionary.Add(word, definition)

		assertError(t, err, nil)
		assertDefinition(t, dictionary, word, definition)
	})

	t.Run("existing word", func(t *testing.T) {
		word := "test"
		definition := "this is just a test"
		dictionary := Dictionary{word: definition}
		err := dictionary.Add(word, "new test")

		assertError(t, err, ErrWordExists)
		assertDefinition(t, dictionary, word, definition)
	})
}

func assertError(t testing.TB, got, want error) {
	t.Helper()
	if got != want {
		t.Errorf("got %q want %q", got, want)
	}
}
```

For this test, we modified `Add` to return an error, which we are validating against a new error variable, `ErrWordExists`. We also modified the previous test to check for a `nil` error, as well as the `assertError` function.

Try to run test

The compiler will fail because we are not returning a value for `Add`.

```
./dictionary_test.go:30:13: dictionary.Add(word, definition) used as value
./dictionary_test.go:41:13: dictionary.Add(word, "new test") used as value
```

Write the minimal amount of code for the test to run and check the output

In `dictionary.go`

```go
var (
	ErrNotFound   = errors.New("could not find the word you were looking for")
	ErrWordExists = errors.New("cannot add word because it already exists")
)

func (d Dictionary) Add(word, definition string) error {
	d[word] = definition
	return nil
}
```

Now we get two more errors. We are still modifying the value, and returning a `nil` error.

```
dictionary_test.go:43: got error '%!q(<nil>)' want 'cannot add word because it alreac
dictionary_test.go:44: got 'new test' want 'this is just a test'
```

Write enough code to make it pass

```go
func (d Dictionary) Add(word, definition string) error {
	_, err := d.Search(word)

	switch err {
	case ErrNotFound:
		d[word] = definition
	case nil:
		return ErrWordExists
	default:
		return err
	}

	return nil
}
```

Here we are using a `switch` statement to match on the error. Having a `switch` like this provides an extra safety net, in case `Search` returns an error other than `ErrNotFound`.

Refactor

We don't have too much to refactor, but as our error usage grows we can make a few modifications.

```go
const (
	ErrNotFound   = DictionaryErr("could not find the word you were looking for")
	ErrWordExists = DictionaryErr("cannot add word because it already exists")
)

type DictionaryErr string

func (e DictionaryErr) Error() string {
	return string(e)
}
```

We made the errors constant; this required us to create our own `DictionaryErr` type which implements the `error` interface. You can read more about the details in this excellent article by Dave Cheney. Simply put, it makes the errors more reusable and immutable.

Next, let's create a function to `Update` the definition of a word.

Write the test first

```go
func TestUpdate(t *testing.T) {
	word := "test"
	definition := "this is just a test"
	dictionary := Dictionary{word: definition}
	newDefinition := "new definition"

	dictionary.Update(word, newDefinition)

	assertDefinition(t, dictionary, word, newDefinition)
}
```

`Update` is very closely related to `Add` and will be our next implementation.

Try and run the test

```
./dictionary_test.go:53:2: dictionary.Update undefined (type Dictionary has no field
```

Write minimal amount of code for the test to run and check the failing test output

We already know how to deal with an error like this. We need to define our function.

```go
func (d Dictionary) Update(word, definition string) {}
```

With that in place, we are able to see that we need to change the definition of the word.

```
dictionary_test.go:55: got 'this is just a test' want 'new definition'
```

Write enough code to make it pass

We already saw how to do this when we fixed the issue with `Add`. So let's implement something really similar to `Add`.

```go
func (d Dictionary) Update(word, definition string) {
	d[word] = definition
}
```

There is no refactoring we need to do on this since it was a simple change. However, we now have the same issue as with `Add`. If we pass in a new word, `Update` will add it to the dictionary.

Write the test first

```go
t.Run("existing word", func(t *testing.T) {
	word := "test"
	definition := "this is just a test"
	newDefinition := "new definition"
	dictionary := Dictionary{word: definition}

	err := dictionary.Update(word, newDefinition)

	assertError(t, err, nil)
	assertDefinition(t, dictionary, word, newDefinition)
})

t.Run("new word", func(t *testing.T) {
	word := "test"
	definition := "this is just a test"
	dictionary := Dictionary{}

	err := dictionary.Update(word, definition)

	assertError(t, err, ErrWordDoesNotExist)
})
```

We added yet another error type for when the word does not exist. We also modified `Update` to return an `error` value.

Try and run the test

```
./dictionary_test.go:53:16: dictionary.Update(word, newDefinition) used as value
./dictionary_test.go:64:16: dictionary.Update(word, definition) used as value
./dictionary_test.go:66:23: undefined: ErrWordDoesNotExist
```

We get 3 errors this time, but we know how to deal with these.

Write the minimal amount of code for the test to run and check the failing test output

```go
const (
	ErrNotFound         = DictionaryErr("could not find the word you were looking
	ErrWordExists       = DictionaryErr("cannot add word because it already exist
	ErrWordDoesNotExist = DictionaryErr("cannot update word because it does not e
)

func (d Dictionary) Update(word, definition string) error {
	d[word] = definition
	return nil
}
```

We added our own error type and are returning a `nil` error.

With these changes, we now get a very clear error:

```
dictionary_test.go:66: got error '%!q(<nil>)' want 'cannot update word because it doe
```

Write enough code to make it pass

```go
func (d Dictionary) Update(word, definition string) error {
	_, err := d.Search(word)

	switch err {
	case ErrNotFound:
		return ErrWordDoesNotExist
	case nil:
		d[word] = definition
	default:
		return err
	}

```

```go
		return nil
}
```

This function looks almost identical to `Add` except we switched when we update the `dictionary` and when we return an error.

Note on declaring a new error for Update

We could reuse `ErrNotFound` and not add a new error. However, it is often better to have a precise error for when an update fails.

Having specific errors gives you more information about what went wrong. Here is an example in a web app:

> You can redirect the user when `ErrNotFound` is encountered, but display an error message when `ErrWordDoesNotExist` is encountered.

Next, let's create a function to `Delete` a word in the dictionary.

Write the test first

```go
func TestDelete(t *testing.T) {
	word := "test"
	dictionary := Dictionary{word: "test definition"}

	dictionary.Delete(word)

	_, err := dictionary.Search(word)
	if err != ErrNotFound {
		t.Errorf("Expected %q to be deleted", word)
	}
}
```

Our test creates a `Dictionary` with a word and then checks if the word has been removed.

Try to run the test

By running `go test` we get:

```
./dictionary_test.go:74:6: dictionary.Delete undefined (type Dictionary has no field
```

Write the minimal amount of code for the test to run and check the failing test output

```go
func (d Dictionary) Delete(word string) {

}
```

After we add this, the test tells us we are not deleting the word.

```
dictionary_test.go:78: Expected 'test' to be deleted
```

Write enough code to make it pass

```go
func (d Dictionary) Delete(word string) {
	delete(d, word)
}
```

Go has a built-in function `delete` that works on maps. It takes two arguments. The first is the map and the second is the key to be removed.

The `delete` function returns nothing, and we based our `Delete` method on the same notion. Since deleting a value that's not there has no effect, unlike our `Update` and `Add` methods, we don't need to complicate the API with errors.

Wrapping up

In this section, we covered a lot. We made a full CRUD (Create, Read, Update and Delete) API for our dictionary. Throughout the process we learned how to:

- Create maps
- Search for items in maps
- Add new items to maps
- Update items in maps
- Delete items from a map
- Learned more about errors
    - How to create errors that are constants
    - Writing error wrappers