team treehouse: enumerations and optionals

modelling finite data with enums

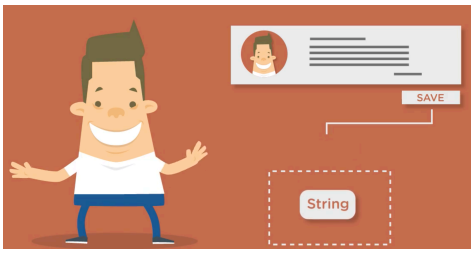
- **enums** are used to **model a finite data set**
  * days of the week are a great example of this
- ```
  enum Day {
      case monday
      case tuesday
      case wednesday
      case thursday
      case friday
      case saturday
      case sunday
  }
  ```
- there are *seven values* to this data set and it will **never change**
  * there will always *only be seven values*
  * the **set** contains a **fixed number of values** and it is *this kind of data* that an *enum models*
  * it seems like that's a limited use case but in fact there are *many ways in which the data presents itself like this*
    ** months of the year
    ** seasons
    ** compass directions
    ** turn by turn navigation directions
      ** but also *user sate in an app*
        ** it's **fixed too**, the user is either *logged in* or *logged out*
          ** there's two states
  * **the advantage of using enums** is that the **compiler can provide some checks** for you
- **exhaustive switch statement** is when **all the possible paths in the code are covered** and thus **there is no need for a default value**
  ```

getting rid of strings

- *a lot of app crashing errors* are just **caused by using strings** to **specify values incorrectly**
- **syntactic sugar** is a **language feature** that makes it **easier to read or express certain code**
- a feature of Swift called **Function** or **Method Overloading** and it allows you to **write two functions that have the same name** as long as they take **different parameters**

the absence of data

- so far every line of code that we've written
  * makes a pretty big assumption that the data we're working with *actually exists*
- let's say we're building a journaling app that lets the users record their daily thoughts
  * part of this app includes a *save* method
    ** users can *write down stuff* and *tap the save button*
      ** the **save button** then **calls a method** that **accepts a String** and **saves this String** in **some database**



- the save method **expects a string** but if the *user hits save too quickly* there's **no string to pass to the function**
  * that *data doesn't exist*
    ** if you *haven't taken this into account*, at this point the method will **return an error** and the app **will crush** and cause *terrible user experience*
- Swift was built with *three main things in mind*, it was to be
  * safe
  * modern
  * powerful

- one of which was **safety**
    * code like in the above example is *not safe* and *there are many situations like these* where you can crush
        ** swift has a *feature built in* just for *these kinds of situations* known as **optionals**

optionals

- **optionals** are **enums** under the hood
- an **optional** looks like this
- ```
  enum Optional<T> {
      case Some(T)
      case None
  }
  ```
    * **<T>** is a *language feature* called **generics** we will cover later
    * this is a **generic type** meaning that when we **create an Optional** the **type** that **add to the question mark** is **substituted for T**
- so when we create an Optional string like so
- `let middleName: String?`
    * the enum Optional type looks like this
        ** ```
           enum Optional {
               case Some(String)
               case None
           }
           ```
- an **optional** has **two members**
    * **None**
        ** when there's **no value**
    * and a member named **Some**
        ** that has an **associated value**
- if the **value exists**, the **compiler returns the associated value**

## optional binding

- the **safe** and the **correct** way to **unwrap** things is using **optional binding**

- **dictionaries always return an optional value**
    * that is because when we ask for a *particular value* using a *key*, there's a *chance* this key *might not exist*
        ** and rather than *crashing*, we *safely* **return nil** using an **optional**
- given a **dictionary** of type :
- `[String : String]`
    * where the value is a string
    * the *return type* of *getting a value* from the dictionary is an **optional string** so *string with a question mark*
    * `String?`
- **first choice** at our disposal that we can use is an **optional binding** using the **if let statement**

## downsides to using if let

- typically when we **retrive data from the web** which is what most iOS apps do
    * it comes with some sort of data format that is **most easily converted** to a **dictionary**
    * the **data is packaged up** as **key value pairs** and we can **retrieve values** for **each property** using the **relevant key**
        ** but because **dictionaries** are **optional**, we *can't simply just get the value out* because a **key might not exist**
- one should get into the **habit** of **never using force unwrapping** using **! operator**

## early exits using guard

- **return** is a **control transfer statement** and by **calling it**, we **exit the current scope**
- *in contrast*, an **early exit** is when we **exit the function as early as possible**
    * the **moment** we hit an **undesirable path of code** and we do that *using* the **guard statement**
- with the *if let statement*
- ```
  if let someValue = someOptionalExpression {
      print(someValue)
  }
  ```
  ```

    * we start with **if**
    * then a **temporary constant** to **assign the value to** and then an **expression**
        ** this **expression** has to be one that **returns an optional value**
            ** *if this* **succeeds inside** the **if let statement**, we have **access** to that **temporary variable containing** the **unwrapped value**
- **in contrast**, we **start the guard statement** with a **guard keyword**
    * like *if let*, we then **create a constant** to **assign the value** and then we **provide an expression** to **evaluate**
    * `guard let someValue = someOptionalExpression`
        ** this expression like before **must return an optional value**
            ** if the expression **succeedes**, that is the **optional contains a value** and **not nil**, it is **assigned to the constant**
                ** so far it is the same as *if let* however this is where it *differs*
    * instead of *opening brace*, we write the **else keyword** and **then the brace**
    *
```
guard let someValue = someOptionalExpression else {
  return nil
}
```
- **nested code** makes it **hard** to **deduce** the **flow of your code** because you have **all these branching paths for each check**

recap on optionals

- *in many languages any type* can be set to **nil**
    * this was a **common problem** in **objective-c** and caused **crashes** *all over the place*
        ** this of **nil** as an **exploding value**
            ** if you *see it* but *aren't prepared* for it, it **explodes** and your **app crashes**
    * in Swift, we can **annotate** a **type** and **indicate** that **it can be nil** by making it an **optional type**
        ** we do this by adding a *question mark after the type declaration*
        ** `String?`
            ** it's important to note that in Swift, **only an optional value can be nil**
                ** what it *means to the compiler* is that **if the type isn't an optional**, it **doesn't need to worry** about it **being nil** and the **app crashing**
                ** but when a **type** is **optional** and you *try to simply use it*, the *compiler will tell you*
- **once we have an optional**

* we use the value by **unwrapping it**
*
```
enum Optional {
   case Some(String)
   case None
}
```
* because an **optional enum** *under the hood*, we want the *associated value* with this *Some case*
* Swift comes with **two nice syntax constructs** that helps us **get the value out** without worrying about poking around in the enum
* **the first** is the **if let** syntax also known as **optional bind**
  **
```
if let someValue = someOptionalExpression {
    print(someValue)
}
```
  ** when we use an if let statement, we **provide an expression** to **evaluate** that **returns an optional**
    ** if the **optional contains a value**, this is then **assigned** to a **temporary constant** that is **scoped to the if statement** (we can only use it inside the if statement but if the value is nil, we go back to our regular path)
    ** **if let statements allow us** to **combine expressions** so that we can **evaluate multiple optional expressions** where **each one depends on the previous expression containing a non-nil value**
      ** however *if let statements* **can lead** to **nested code** where the **happy path isn't straightforward**
      ** for this we have an alternate construct
* **the guard statement**
  ** **the guard statement** allows for an **early exit** and much like the *if let statement*
    ** it **evaluates the results** of an **optional expression** and **binds the value** to a **constant in the local scope**
    ** the **difference** here is that **using an else clause**, we can **nest the error** or **failure case inside** while the **happy path of code** continues **along the same scope**

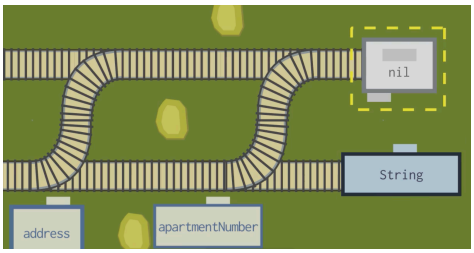
initialising with raw values

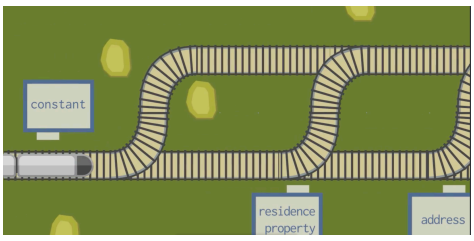
- we've learned about **initialisation** before
  * the **way** in which we **create an object**

- like structs and classes, we can **initialise an object**, but **only if it has a raw value**
- *typically* **when we make a request on the web**, let's say by *entering an address*
  * **the server first returns a response**
  * these **responses** have **various status codes** that **tell your browser** what the **state of the response was**
- for example **if we enter an address** and **the request is successful** the **status code is 200**
  * if something **goes wrong**, you **get a number in the 400 range back**
- many of you have seen the **404 NOT FOUND error** when **browsing the web**
- since there is a **limited set of status codes**
  * **enums provide** a **great way** to **model this data**

optional chaining

- **optional chaining** is a **process** for **querying** and **calling properties**, **methods** or **subscripts on an optional** that **might be currently nil**
  * **if** the **optional contains** a **value**, the *property method* or a *subscript call* **succeeds**
    ** but **if it's nil**, this **entire call returns nil**
- you can think of this operation as a set of **two parallel railroad tracks**
  * one **final truck ends with the final value** which in our case was a *String*
  * the **other truck ends in nil**





- the track starts as susan constant, we know that it isn't nil

recap

- it's *okay* to *be overwhelmed*