team treehouse : object oriented swift

- structs in **swift** follow an **upper camel casing** convention
- the struct definition acts as a **blueprint**
- *a class* is a *template for an object*
- *object* is an *instance of a class*

object oriented programming

- **object oriented programming** is a **style of programming** wherein we **model information** to **data structurers** or **objects**
  * one particular *kind of object* is a **structure**
    ** we create a *structure* using a *keyword*
      ** `struct User {}`
- using this **blueprint**, we can then create an **instance** of this **structure**
  * `let user = User(name: "Jim", email: "jim@j.com"`

methods

- in **object oriented programming** a **method** is any **function** that is associated with a **particular type**

recap

- `struct Point {}`
  * is one of the objects we can use in swift to create **custom types**
  * struct can define **stored properties** that **encapsulate values**
  * a *struct* can also have *functions* that work on the **data** that is *stored* in these *properties*

    * a **function associated** with an object is called a **method**
    * both the *stored properties* and *functions* that we've seen so far are **scoped instances** of the **struct**
        ** meaning we **cannot use it** unless we *specifically create* an **instance**
- to **create an instance**
- `let point = Point(x: 0, y: 0)`
    * we start with a *name of the struct* and then use a *memberwise initialiser method* that swift *automatically creates* to *assign values* to *each of the stored properties*
        ** swift does this by simply *creating a method* with *parameters* for *each of the stored properties* specified in the *definition of the struct*
        ** we can write out an *initialiser method* ourselves by creating a *special method* using the **init keyword**
        **
```
init (x: Int, y: Int) {
self.x = x
self.y = y
```
        ** the **purpose of the initialiser** is to **assign values** to all **stored properties** during **creation**
        ** an **initaliser method** is a *special method without a name* and a *list of paremeters* that we use to *pass in values* to *our stored properties*
        ** if we need to *refer to the instance* from *inside the instance method* we use the **self** keyword
            ** that is how we can create our first object

classes

- class is another *type of object*
-
```
class Enemy {
   var life: Int
}
```
- stored properties can either be **constants** or **variables**
    * this determines *how* the *properties behave* after an *instance has been created*
- in *programming terminology* when we have a value where we can only *get* the value *from* like a *constant*
    * we say it's a **read-only** or **gettable property**
- if we can also *change the value* at any time
    * we say it's **read-write** or **settable property**

- for **classes** we **always** have to **write the init method** *ourselves*
- instance methods should be narrow in scope and carry out a single task
- **helper methods**
    * i.e. methods that we do not call directly on an instance but help compute the output for another function
- you *can't inherit from multiple classes* in Swift
    * but you can *adopt multiple Protocols* that are *similar to classes* in the sense that they *set a blueprint* for other
        ** classes
        ** enumerations
        ** or structures

inheritance

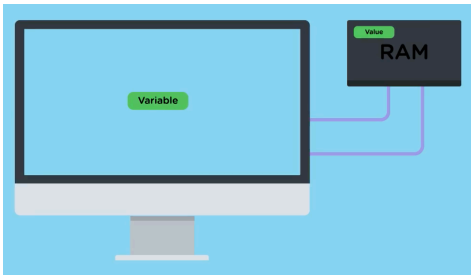
- classes support an interesting mechanism known as **inheritance**
- when we talk about *class inheritance*, we are essentially saying the *interface* or *implementation* is the **same** for *both these classes*
- a **class** can **inherit methods**, **properties** and *other characteristics* from another class
- when one class inherits from another
    * the *inheriting class* is called the **subclass**
    * and the *class it is inheriting from* is called the **superclass** or **parent class**
- two *advantages* of *inheritance*
    * we *avoid code duplication*
    * we can *refine* a *subclass* and either *modify* the methods or *add* new ones
- that is not all *inheritance* provides
- we can *overwrite* both the *properties* and the *methods* on the *superclass*
- when we subclass a superclass to create a new base class
    * we are not just creating a new class and copying the contents of the class into it
    * when we do this, we create an **inheritance chain**
        ** thus *subclass is connected to the superclass*

initialising a subclass

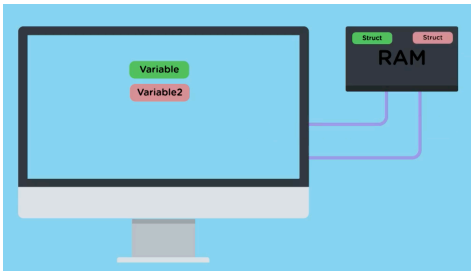
- *provide values* for properties of the subclass
- once the subclass is *initialised*
  * *provide values* for *properties* of the *base class*
- when we **override** a *method*
  * we are essentially saying, we are going to use this method from the *superclass*
  * but we are going to change the body of the method to do something different
- the subclass inherits all the methods and properties from its superclass
  * from here we can customise the subclass as we need to
- the way we *initialise any class* in swift is by *calling its init method*
- the way *initialising* works in a *subclass* is that we **first** *initialise the subclass*
  * then *swift goes up the chain* and *initialises* the *parent class*

value vs reference types

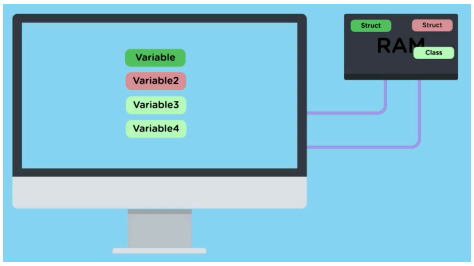
- a **value type** is a *type* whose *underlying value* is *copied* when it is *assigned* to a *new variable* or *constant* or *when it is passed into a function*
- a **reference type** is *not copied* when assigned to a new variable or constant or when passed into a function
  * rather than a copy, a **reference to an existing instance** is used
- all **structs** are **value types** in swift
  * this means that the *values are copied*
- a **class** is a **reference type**
  * when you assign a *reference type* to *another variable*
    ** we simply *assign* a **reference**
- to understand this better :
- when we *declare* a *variable* or a *constant* and *assign it some value* we are *storing this value* in the *computer's memory*



- the *name of the variable* simply *points to this place in memory* where the data lives
- if this data is a *value type* (i.e. struct) then when we *copy the data over*, then what we are actually doing is *making a copy* of the data and *storing it in a new place in memory* and the *new variable or constant* points to a *new variable*



- in contrast, if the data is a **reference type** and we *assign it to a new variable*, the *new variable simply points to the same place in memory*



- this way when we *change the property of the first instance*
  * since the *new instance* is just pointing to the *first one*, it *changes as well*
- *value types* are quite *common in swift*
  * arrays
  * dictionaries
  * Strings
  * Ints

- this means that we have to keep in mind that when they get copied
  * that is when we *assign them to new variables*
    ** *we are working with a copy*

recap

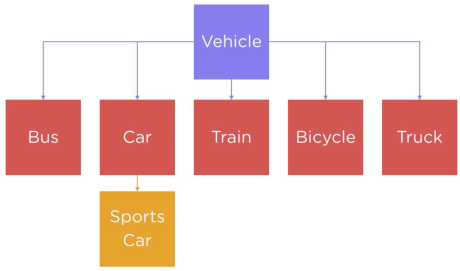
- we have two **main objects**
  * structs
  * classes
- both objects can contain stored properties
  * in its simplest form a stored property is a constant or variable that is stored as part of an instance of a particular class or structure
- stored properties can also take default values as part of the class or structure definition
- using this blueprint of a class or struct definition
  * we can create an instance to use
    ** creating an instance *involves giving our stored properties initial values*
      ** and we do this through a process called **initialisation**
- structs *automatically* get *member wise initialisers* for its *stored properties*
- with classes
  * we need to **define an init method**

```
class Person {
    let name: String
    var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}
```

- an **init method** is simply
  * *a special instance method* that can *take parameters* like any function
  * inside the init method we can *set* or *modify* the *initial values* for any *stored properties*
- both structs and classes *can contain methods* that we *call on an instance*
  * these are known as **instance methods** and are written just like we write a function

    - ** except that they are *contained within the class or struct definition* and *scoped to it*
- we *access both stored properties* and *instance methods* by using *dot notation with an instance*
- all of this is *common* between *two types of objects*
    * here's where things start to *diverge*
- classes support a mechanism called **inheritance**
    * where one class can be **based off another** and **use the same implementation**
        ** this leads to *great code reuse* and allows us to *extend objects* in ways that *narrowly scope their function*
        ** for example
            ** we don't need to build this one giant automobile class that has the properties and methods of buses, trucks, cars, motorbikes and so on



- instead we can create a base vehicle class and then subclass that to *create more specific types* like bus
    * by *adding properties* and *overriding methods*
- to subclass a base class or super class and create a new type
    * we declare a class like normal, and then add a colon and the name of the class we're inheriting from
- ```
  class Friend: Person {
  }
  ```
- when we *inherit from a super class*
    * we *need to ensure* that *all the stored properties*
        ** that is both for the *subclass* and *superclass*
            ** are *initialised* when *creating an instance*
- for now there's a *simple rule* that we're going to follow
    * we *provide values* for these *stored properties of a subclass*
        ** once the subclass is sorted
            ** we call **super.init** and *initialise any properties in the super class*
  ```

* once the *subclass* is *initialised, provide values for properties of the base class*