team treehouse: protocols

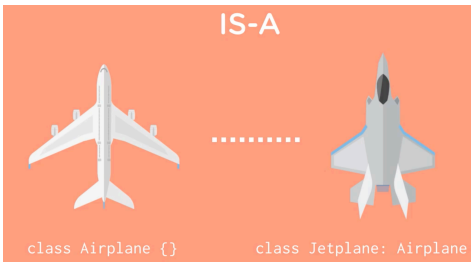
what is a protocol?

- on **other languages protocols** are **called interfaces** but the **concept** is the **same**
  * they **define** a **blueprint of methods**, **properties** and **other requirements** that **suit** a **particular task** or **piece of functionality**
  * it **allows us** to **define** a **set of expected behaviours**
- we **define protocols** in much the *same way* as we *define classes, structs* or *enumerations*
- **protocol** serves as a **blueprint** for **some functionality** or **behaviour**
  * this means that the **protocol** *only describes* what an **implementation** of this **functionality looks like**
    ** unlike a class or a struct, it **doesn't actually provide an implementation**
- the **advantage** of **specifying the protocol** is that it **provides** an **expectation of certain attributes or behaviours**
- a **computed property** lets you **create a property** that **determines its value through some computation**, hence the name *computed property*
  * you **cannot assign it** a **value directly**
- **protocols** are **first class citizens**
- designers of swift, call **swift** a **protocol oriented language**
- a **protocol** is a **contract** of **methods**, **properties** and **other requirements**
- **protocols** are **fully fledges types** in **swift**

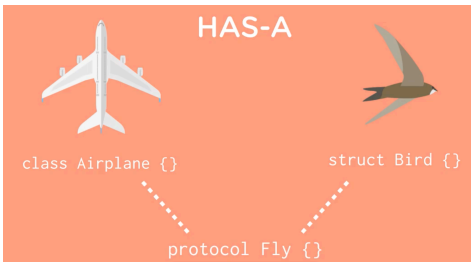
protocol inheritance

- *creating objects* through *composition* that is *using protocols* to *implement functionality* offers us a *certain level* of *flexibility* that *inheritance simply doesn't*
- oftentimes it can be *hard* to *determine when* we need *composion over inheritance*
- a *simple way* to *think about this* is to *ask if the relationship between* the *objects* is an **is a** or **has a relationshipt**

- let's say we have a base class called airplane that modelled an airplane
  * now we want to model a jet plane
    ** well a jet plane is a type of airplane so in this case **jetplane inherits from airplane**
    ** this is an **IS-A type** of **relationship** and **inheritance** is **best suited** as a **design pattern**



- if a **class wants to model** the **exact same behaviour** and **attributes** of **another class**, and perhaps to **add to it**, then we **can use inheritance**
  * in the example a *jetplane will do everything an airplane can do plus some more cool stuff*, so we use **inheritance**
- on the flip side, you you **only want** to **model** a **particular aspect**, a **limited subset** of the **behaviour**, then **composition** is **better**
- let's say we're modelling a bird
  * a bird is *not an airplane* but it it *has a feature* that the *airplane does*
    ** they can *both fly*
  * in a **HAS-A relationsip**, it makes sense to **extract** that **common behaviour** and **create a specific protocol** for it
    ** create a *fly protocol* that *both* the *airplane* and *bird* **conform to**



- **protocols** can **inherit from other protocols**
- **unlike classes** which **can only inherit** from a **single base class**
    * **protocols** can **inherit** from **multiple protocols**

swift's standard library protocols

- the **swift standard library** *contains* **55 primary protocols** and they *can be grouped* into **3 secions**

1. Can Do
2. Is A
3. Can Be

Can Do

- **Can Do** are used to **represent behaviour** where an **object can do something**
    * for example to compare one object to another object of the same type, we can *encapsulate this behaviour* is a **protocol** called **equatable**
- you can notice that all of these protocols have one thing in common, all of them have the suffix **able**
    * this is a **convention** the *swift developers* have *followed*
    * if **your protocols model behaviour** about **something your objects can do**
        ** they can be **compared**, they can be **hashed**, they can be **equated**, then you **add the able suffix**

Is A

- if **your object is a type of another object**, then it **should inherit from it**
  * **Is A** naming simply **indicates** that the **protocol models** a **concrete type**
  * *swift follows* a **convention** of *adding the suffix* **Type** to these kinds of protocols i.e.
    ** CollectionType
    ** IntegerType
      ** they kind of **model the identity of an object**

Can Be

- **Can Be protocols model behaviour** where **one type can be converted** to **another type**
  * some of these types in the swift standard library include
    ** FloatLiteralConvertible
    ** ArrayLiteralConvertible
    ** CustomStringConvertible
      ** with CustomStringConvertible you take any type that conforms and convert it to a *string representation*
    ** *objects* that *conform* to these *protocols* can also be **initialised** with the **literal value specified** in the **protocol**
- **Can Be protocols** that **convert** from **one type** to **another** follow the **convention** of having the *suffix*, **convertible**

protocol oriented programming

- **protocol oriented programming** simply means **carefully defining** the **interfaces** to **your objects**
- it means **preferring composition over inheritance** to **create flexible objects**
- you **want** to **always think small** when it comes to **programming**
  * *small protocols*, *small objects*, *small view controllers* and *small models*
    ** **each type** or **each object** should **do very little**

recap

- **protocols** are **useful** for *one main reason*
  * they **allow us** to **encapsulate common behaviour without** having to **resort** to an **inheritance structure**
- a *good way to* **distinguish** when you *need to use* a **protocol** is to **identify** whether the **relationshio** is an **Is-A** or a **Has-A** *relationship*
- **protocols** are **fully fledged types**
  * meaning we can **specify** a **protocol** whenever we *typically* **specify a type**
    ** that is the type of a
      ** stored property
      ** constant
      ** variable
      ** function
      ** parameter
      ** return type
- **protocols** can **inherit** from **other protocols**
- by **conforming** to **individual protocols**, we can *mix and match* the **behaviours** in our **final types**