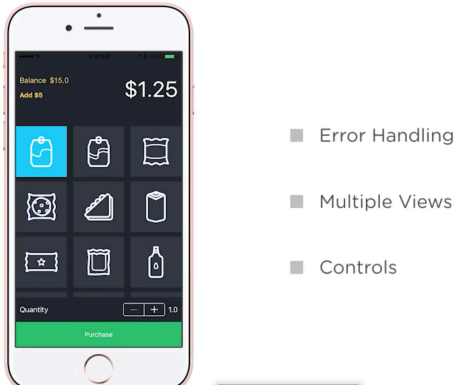
team treehouse: building a vending machine app in Swift 2.0

introduction

- the **main focus** of this **course** is **error handling**
- how to **introduce multiple views**
- how to work with **controls** that allow us to enter *various information*



concrete types and property lists

- **value types** that is a *struct* or *enum* are **things**
  * while **reference types** or **classes do things**
- the **compiler** can **verify** that the **enum value** you are using is **correctly defined**
  * it **cannot do this** with a **string**

property lists and XML

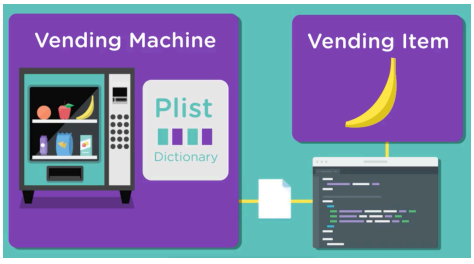
- a **property list** is a *file* that **organises data** into **named values** and **lists of values**

    * much like the dictionaries and arrays that we're worked with before
- **P-lists** give you the **means** to **produce structured data** in a **lightweight** and **efficient format** that is **easy to access** and **store**
    * *P-lists* are *used* in both *iOS* and *Mac development* and are traditionally *used* to **store user preferences**
        ** but we can *use it* to *store* any *sort of simple structured data*
- *under the hood*, **p-lists convert** the **structured data** that you see here in front of you to **XML** or **Extensible Markup Language**, which is a *language* for **marking up text** in a **way** that **organises** the **data** for a **computer** to **read easily**

XML: Extensible Markup Language

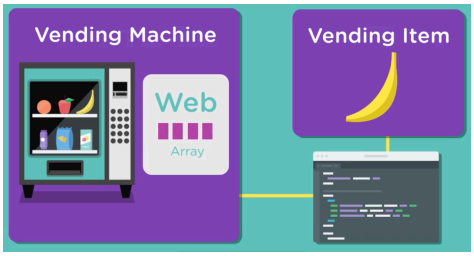
Set of rules for encoding documents in both machine and human readable format

- **property lists** are *meant to be* **lightweight** and **language independent**
- if we were to get the *P-list* from a *server*, both an iOS app and an Android app **can work** with the **same data source** to **convert** it to the **specific type** it needs
- **XML** is a very **common format** for **unpacking** and **storing data**
- when we **design** an **interface builder**, all our **storyboard files** are **stored** in the **XML** format as well in what is called a **NIV file**
- your **data** should **never** be **tightly coupled** to your **model**
    * for example, in our case, we have a *vending machine* and *vending item* as our **main models**
    * the **data** is **coming from** some **external P-list**
    * we could write the **code** to **retrieve** the **contents** of the **P-list**



- **convert** it into an **internal structure**, **obtain** the **data** we **want** and **create** the **instances** of **vending item** to **use** all **inside** of the **vending machine class**
    * this would work okay but let's say in the future we have a *new data source*, **the web**

    ** the **means** to **obtain data** would be **hard-coded** into the *vending machine class* but since **our data source** is now **completely different**
    ** we would need to **rewrite** some of the **internals** of the *vending machine class* to get *things to work*
    ** the **data** from the **web** may also be **stored** in **different data format**



- and uses *arrays* instead, rather than *dictionaries* that a *P-list uses*, we will have to *change the internal structure of the vending machine* to *not use the inventory dictionary* because now *everything is different*
- the point here is that **hard coding** the **means** to **obtain** the **data inside the class** that **uses the data** is the **wrong approach**

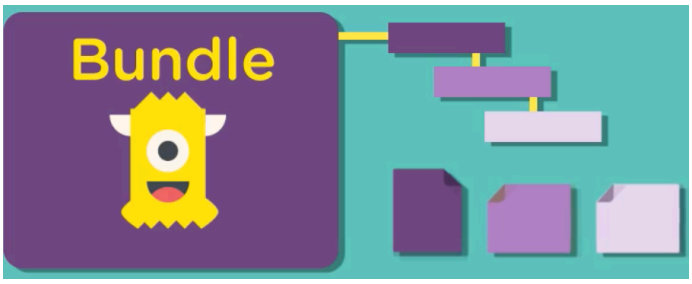
type methods

- an **instance method** is one that is **called** on an **instance** of a **particular type**
- a **type method** is **associated** with the **type** itself
    * the way you **indicate** that a **method** is a **type method** is by *adding the word* **static**, **if** it is **struct** and adding **class** if it is a **class**, in **front** of a **method signature**
- to **use** a **type method**, we **call** it on the **type** itself **rather than** an **instance of** the **type**

from property list to dictionary

- in *swift*, we may *not always know* the **type** of the **object** we are **working with**
    * to handle this, the language comes with **two type alieases** to **represent non-specific types**
    * a **type alias** is an **alternate name** for an **existing type**
- the *first type alias* is **AnyObject**

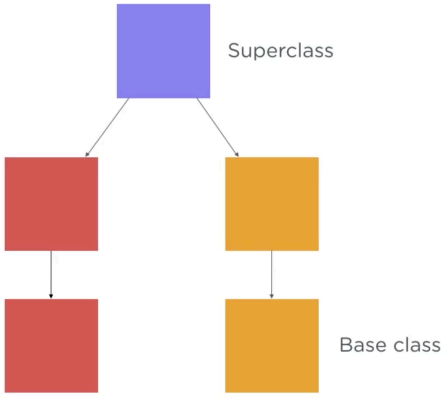
    * **AnyObject represents** an **instance** of **any class type**
- we also have a *type alias* called **Any**
    * **Any** is the most **generic reperesentation** of a *type in swift* and can **represent** an **instance** of **absolutely any type** in *swift* **including functions**
- when we **write** an **app** and **deliver** it to the **app store** it is **packaged** in what is known as a **bundle**
    * a **bundle** is a **directory** with a **standardised hierarchical structure** that **holds executable code** and the **resources used** by that **code**



- **bundles** are a great **abstraction** on **top of the files** in our **app** and it **provides** a really **easy way** to **interact with them**
    * since we're writing an app, it is more specifically known as an **application bundle**
- **NSBundle** can **query** our **bundle directly** and **locate** the **resources** that we **need**
- since a **bundle's structure** is **standardised** given the **right bundle**
- **NSBundle knows where to look** for **any resources** that we **need**
    * so we can **use NSBundle** to **get** a **path** to the **Plist** because it **knows** where **all those things exist**

downcasting

- it's called **downcasting** because if you *think* of the **object graph** as a **tree** with the **base type** *employee right at the top* and then *more specific types* as **nodes** *extending downwards* kind of like **branches** and **leaves**



- then when going from a *base* at the *top* to a *subclass down below*, we go **down the tree**
- **downcasting**, however **may not always succeed**
  * so the **type cast operator** comes in **two flavours**
    ** the **conditional form as?**
    ** and **forced form as!**
      ** *forced form* like the *forced unwrap operator should only be used* when you *know* the *downcast can succeed*
- **downcasting** is the **process** of **converting from** a **superclass** to a more **specific sub-class**