Table of Contents

Foundations of Programming - Fundamentals

Library = Framework i.e. Huge amounts of prewritten tested code ready to be used.
Programmers know that no-programmers don't, is that it's not really about the language, it's about the library. Those are the skills to develop i.e. know not just the language, but also what you can do with it without writing it all yourself. ex. Java having prewritten code just for MIDI and music production.

Event driven:

HTML:

CSS: #box {width:100px; height:100px; background:red;}

JS: var box = document.getElementById("box");

```
box.onclick = function() {
    box.style.background='blue';
}
```

Programming Basics

Programming languages levels (Low (Closest to CPU and hardest to write) > High): Assembly > C > C++ > Objective-C > Java, C# > Ruby, Python > Javascript.

Every computer program is a series of instructions; very small and very specific ones. The art of programming is taking a large idea and breaking it into these instructions.

Core Programming Syntax

These instructions are called statemets and are like sentences in English.Syntax is to programming what grammar is to English.

Why isn't there just one programming language? Well, there is, and it's called machine language. That is the only language the computer (CPU) understands. It is impossible to write by a human. Writing a full program in machine code is like digging a tunnel with a spoon.

Programming languages serve to bridge the gap between human beings and computer hardware.

Whatever we write, regardless of language level, has to be converted to machine code before it can run.

Writing in a language requires 3 things:
1. How to write it i.e. where to actually type.
2. How will the source code be converted to machine code.
3. How to execute the program.

1. Code is written in plain text editors, programmer text editors and IDEs.
2. Source code is converted in machine code in 2 ways, either by compiling or interpreting it.
- Compiling: A program called compiler converts the source code into machine code in a separate file called an executable. This way, the source code can't be seen. Compilers are downloadable but are often built in into IDEs. ex. C, C++, Objective C.
- Interpreting: The code is run line by line i.e. processed on the spot and the soruce code is needed. ex. PHP, Javascript. The conversion to machine code is done by the web browser in the case for Javascript. OS runs the web browser, which runs the JS.
- There is a 3rd way called JIT compilation which is compiling the source into an intermidiate language (IL or bytecode) just a step before machine code, which can then be compiled fully depending on the PC the end user has. ex. Java, C#, VB.NET, Python.

Scripting languages are more limited programming languages that are embedded inside another program. ex. ActionSript > FLASH, VBScript > MS Office, Javascript > Web Browser.

Variables and Data Types

Data is stored in variables in orded for it to be used in other lines i.e. the computer doesn't know unless you write the statement again.

Variables are containers for data. We grab a piece of computer memory and we give it a name.

Strongly typed language = Variables must have a type. ex. integer, string.
Weakly typed language = No need of type.

All the same: a = a + 1; a += 1; a++; This is called incrementation.

Writing conditional code

Parantheses (), Brackets [], Braces {}

After each switch case, there needs to be a break;

Modular code

Large amounts of code are broken into smaller reusable blocks called functions. A function is simply the idea of taking a block of code, one line or 100 lines, wrapping it up and giving it a name so it can be called later as one thing. If it's in a function, it won't run unless you call it.

Recursion is a function calling itself.

function (a, b) { var result = a + b; alert(result);} the a, b in the () are parameters. function (5, 10) the 5, 10 are arguments.

Returning a value is done like so:

```
function addTwoNumbers(a,b) {
        var result = a + b;
        return result; // This saves the value and jumps out of the code.
}

var x = addTwoNumbers(5,10); // The var x is assigned the value of the result of the function.
alert(x); // This displays the returned result.
```

Iteration writing loops

The main issue with loops is not when to loop, but when to stop.

```
a = 1;

while (a < 10) {
        alert(a);
        a++; // This increments a which prevents an infinite loop.
}
```

```
var amount = 0;

// Create the index
i = 1;

// Check condition
While (i<10) {
        amount = amount + 100; // This is the same as amount += 100;
        // Increment index
        i++;
}

alert(amount);
```

A successful loop needs 3 things: Creating an index, checking a condition and incrementing the index.

The for loop has all that in one statement.

```
for (i = 1; i<10; i++) {}
```

Do While Loop

```
var a = 1;

do {
        // your code
        a++;
} while (a<10);
```

The difference here is that the code will always be executed once before the condition is looked at.

******* More about strings

String methods ex. string.length, methods (in this case) are functions that belong to the string.

Collections

Arrays are objects.

Methods are functions that belong to an object.

Associative array = dictionary = map = table (This is not a database) ex. Normal array [alabama, alaska, arizona] indexes [0,1,2]. Associative array [alabama, alaska, arizona] indexes [AL, AK, AZ]

Programming style

Variables and functions should be named by the camelCase convetion. ex. foo, fooBar.

Functions should be defined before they are called.

Input and output

DOM = Document Object Model i.e. Document = Whole website, Object = Elements ex. h1, ul, div id, Model = Agreed upon set of terms.
var headline = document.getElementById("mainHeading");
headline.innerHTML = "Wow, a new headline.";

Debugging

Bugs usually come down to syntax or logic errors.

Object orientation

Class is a blueprint, a definition. It describes what something is, but isn't the thing itself.
Classes describe 2 things: Class Person with 1. Attributes i.e. variables i.e. properties (name, height, weight) and 2. behaviour i.e. functions i.e. methods (walk, talk, jump).

Objects are the thing itself. Object is created from the class. Objects are createad by using "new" ex. var today = new Date();

Encapsulation = The act of classes containing properties and methods all in one.

Memory management

Pointer = A variable whose value is the memory address of another variable.

Assembly, C = Manual; C++, Objective-C = Referece countring; Java, C#, VB.NET = Garbage collection; Javascript = Automatic;

Multitasking = Running more than 1 programs at the same time.
Multithreading = Doing more than 1 thing in a program at the same time.

Exploring the languages

Java:
- Enormous library called "Java Class Library" i.e. prewritten code that you don't have to write.
- High level strongly typed language with garbage collection.
- Hybrid compilation model (Neither compiled nor interpreted), compiles into bytecode (instead of machine code). In order for the bytecode to be compiled into machine code, Java uses the Java Virtual Machine or JVM. This allows for the code to be run on multiple platforms, because it is compiled by the JVM specifically for the place we want to run it on.
- IDE: Eclipse, NetBeans

```
// Java example code
class HelloWorldApp {
	// another main method!
	public static void main(String[] args) {
		int myInt = 55;
		System.out.println("Hello, world!");
	}
}
```

C#:
- Enormous library called ".NET Framework" i.e. prewritten code.
- High level strongly typed language with garbage collection.
- The same as Java, it uses a hybrid compilation model. It compiles halfway into "Microsoft Intermediate Language" and can then be distributed to different PCs with different CPUs which then take the last step and compile it into their specific machine code. Like Java using JVM, .NET languages use ".NET Runtime" to do the last step compilation.
- Website development is lumped under the term ASP.NET which is simply a phrase saying that you are building a dynamic website using Microsoft technology (it is not a separate language), almost always C# or VB.NET.
- C# is the closes looking language to Java. Everything is a class / object.
- IDE: Visual Studio (Includes a text editor, compiler, debugger. http://microsoft.com/express

```
// C# example code
// Hello1.cs
public class Hello1
{
	public static void Main()
	{
		System.Console.WriteLine("Hello, world!");
	}
}
```

```
public class Animal
{
	public static void Eat()
	{
		System.Console.WriteLine("I am eating.");
	}

	public void Run()
	{
		System.Console.WriteLine("I am running.");
	}

public virtual void Sleep() {}

}

public class Dog: Animal
{
  public override void Sleep()
  {
	System.Console.WriteLine("I am sleeping in a house.");
  }
}
```

Scopes:
- Public (moze da se povika od bilo koja klasa)
- Protected (metodot moze da se povika samo od klasi sto nasleduvaat od Hello1 i Hello1)
- Private (moze da se povika samo vo ramki na Hello1).

Static:
- Ako go ima, metodot se povikuva bez da se instancira objekt od klasata. Animal.Eat(); (Eat e static funkcija)
- Ako go nema, mora da se instancira objekt od klasata pa da se povika metodot. Animal Mitre = new Animal(); Mitre.Run(); (Run ne e static funkcija)

Returns:
- Void (Metodot vrsi nekoja rabota, ama ne vrakja rezultat.)
- int, string, float, double, bool, Animal (vrakja objekt od tipot Animal)... (Ovie vrakjaat toa sto pisuva)

Virtual:
- Ako ima, metodot samo se deklarira vo glavnata klasa, a go specificiras vo klasite sto nasleduvaat.
- Ako go nema, go nema.

```
// VB.NET example code
Module Module1
	Sub Main()
		System.Console.WriteLine("Hello, world!")
	End Sub
End Module
```

```
public class Animal
{
	public static void Eat()
	{
		System.Console.WriteLine("I am eating.");
	}

	public void Run()
	{
		System.Console.WriteLine("I am running.");
	}

public virtual void Sleep() {}

}

public class Dog: Animal
{
   public override void Sleep()
   {
	System.Console.WriteLine("I am sleeping in a house.");
   }
}
```

Scopes:
- Public (moze da se povika od bilo koja klasa)
- Protected (metodot moze da se povika samo od klasi sto nasleduvaat od Hello1 i Hello1)
- Private (moze da se povika samo vo ramki na Hello1).

Static:
- Ako go ima, metodot se povikuva bez da se instancira objekt od klasata. Animal.Eat(); (Eat e static funkcija)
- Ako go nema, mora da se instancira objekt od klasata pa da se povika metodot. Animal Mitre = new Animal(); Mitre.Run(); (Run ne e static funkcija)

Returns / Return Type:
- Void (Metodot vrsi nekoja rabota, ama ne vrakja rezultat.)
- int, string, float, double, bool, Animal (vrakja objekt od tipot Animal)... (Ovie vrakjaat toa sto pisuva)

Virtual:
- Ako ima, metodot samo se deklarira vo glavnata klasa, a go specificiras vo klasite sto nasleduvaat.
- Ako go nema, go nema.

General

Library = Framework i.e. Huge amounts of prewritten tested code ready to be used.

C#:
- Enormous library called ".NET Framework" i.e. prewritten code.
- High level strongly typed language with garbage collection.
- The same as Java, it uses a hybrid compilation model. It compiles halfway into "Microsoft Intermediate Language" and can then be distributed to different PCs with different CPUs which then take the last step and compile it into their specific machine code. Like Java using JVM, .NET languages use ".NET Runtime" to do the last step compilation.
- Website development is lumped under the term ASP.NET which is simply a phrase saying that you are building a dynamic website using Microsoft technology (it is not a separate language), almost always C# or VB.NET.
- C# is the closes looking language to Java. Everything is a class / object.
- IDE: Visual Studio (Includes a text editor, compiler, debugger. http://microsoft.com/express

Can be cross-platform: iOS (Xamarin.iOS), Android (Xamarin.Android), Linux (Mono)

Solution = A collection of projects.fo
Projects have:
- Properties.
- References. Here you define which libraries are used.
- App.config. ex. Here you can configure which runtime and .NET verion the application will use.
- Program.cs. These are the actual programs, and as sub items, there are the classes and methods for that program.

Startup Project = Which project will run first during debug.

Debugging: Step into = Execute the code one line at a time.

- YOU CAN'T WRITE A FUNCTION INSIDE A FUNCTION. YOU CAN ONLY CALL FUNCTIONS INSIDE FUNCTIONS.

Data Structures - Types

Built in / Intrinsic / Simple Types (Because they simply store a value in the memory): Numeric, Character, Boolean.
Custom / Complex Types (Because they can encapsule attributes and functionalities): Structures, Classes, Interfaces, Enumerations.

C# uses a common type system i.e. each type can either be a value type or a reference type (pointer). The common type also supports inheritance i.e. Parent / Child classes.

Value types derive from System.ValueType, they are: Built in types, Structures, Enumerations. They are stored on the stack and are passed by value i.e. a copy of the value is passed rather than the data itself, so the data remains unchanged.

Reference types are: Classes, Interfaces. They are stored on the heap and are passed by reference i.e. via memory address which allows for original data modification.

All types derive from System.Object.

Nice explanation for this shit:

Say I want to share a web page with you.

If I tell you the URL, I'm passing by reference. You can use that URL to see the same web page I can see. If that page is changed, we both see the changes. If you delete the URL, all you're doing is destroying your reference to that page - you're not deleting the actual page itself.

If I print out the page and give you the printout, I'm passing by value. Your page is a disconnected copy of the original. You won't see any subsequent changes, and any changes that you make (e.g. scribbling on your printout) will not show up on the original page. If you destroy the printout, you have actually destroyed your copy of the object - but the original web page remains intact.

Passing by pointer is passing by reference - in the example above, the URL is a pointer to the resource. If you're talking about languages like C, where you can have pointers to other pointers... well, that's like me writing the URL on a post-it note, sticking it on the fridge, and telling you to go and look on the fridge - I'm giving you a pointer ("look on the fridge") to a pointer ("www.stackoverflow.com") to the actual thing you want.

Looping Structures for, foreach, while, do while

```
//for loop
for(int counter = 0; counter < 10; counter++) {
	Console.Writeline(counter);
}
----------------------------------------------------------------
//foreach loop. Much better for collections than for loops.
int [] arrInts = new int [] {3, 5, 6, 23, 95, 45, 32};
for each (int item in arrInts) {
	Console.Writeline(item);
}
----------------------------------------------------------------
//while loop
int sentinel = 0;
while (sentinel < 0) {
	Console.Writeline(sentinel);
	sentinel++; //The same as sentinel = sentinel +1 or sentinel += 1.
}
----------------------------------------------------------------
//do while loop
int sentinel = 10;
do {
	Console.Writeline(sentinel); // 10 will get displayed despite the while condition not being met.
	sentinel++;
} while (sentinel < 10);
```

Decision Structures if, if.. else if, switch

```
//if statement
bool result = true;
if (result) {
	Console.Writeline("Result was true!");
} else {
	Console.Writeline("Result was false!");
}
-------------------------------------------------------------
//if... else if
int value = 0;
if (value == 0;) {
	Console.Writeline("Value is 0");
} else if (value == 1) {
	Console.Writeline("Value is not 0");
} else {
	Console.Writeline("Value is something else");
}
-------------------------------------------------------------
//switch statement
int value = 0;
switch (value) {
	case 0:
		Console.Writeline("Value is 0");
		break;
	case 1:
		Console.Writeline("Value is 1");
		break;
	default:
		Console.Writeline("Value is something else");
		break;
}
```

Variables

Variables and Functions = camelCase, Constants = ALL UPPERCASE

```
int age = 0; // Integer Variable
const int NUM_MONTHS = 12; // Constant

struct Person {
	int age;
	string firstName;
	string lastName;
}
```

Functions

Functions should focus on a single task. This makes debugging easier and makes it easier to sculpt the function.
***Functions are declared outside of main, and are later called in main. "static void Main" is the entry point in a program because it's the first invoked function.
NEVER TRUST USER INPUT.

Non Returning Function

```
// Non returning function.
Concatenate("First ", "Last");

static void Concatenate (string first, string last) {
        string whole = first + last;
        Console.WriteLine(whole); // This should also be a separate function ex. displayString(whole); in order to follow the single task per function rule.
}
```

Returning Function (has to have the returning data type declared)

```
// Returning function.
string word; // This declares the variable word of string data type.
word = Concatenate("First ", "Last"); // This assigns the function to the variable word which in turn assigns the returned value to it.

// The upper can also be written in one line.
string word = Concatenate("First ", "Last");
Console.WriteLine(whole); // This displays the result of the function i.e. the returned value.

// This is also valid.
Console.WriteLine(Concatenate("First ", "Last"));

static string Concatenate (string first, string last) {
        string whole = first + last;
        return whole; // This returns the value in order for it to be assigned to a variable, in this case the variable word of string data type.
}
```

Object Oriented Programming

Class = A blueprint for building a house.
Object = The house.

You can't live in the bluepring of a house, but you can live in a house created by the blueprint.
You live in an instance of a house created by the blueprint.

Values cannot be assigned to a class. You create an instane of a class called object in order to do that.

Classes are containers for the attributes and behaviour of the objects. Classes in C# can only have 1 parent class.
All classes inherit from System.Object.

When classes are created, it is important to provide some control over the data that gets assigned to the members. We don't want illegal assignments such as negative ages. In order to control this, the scope private is used which means only code within the class has the ability to change the variables. This is called encapsulation.

To create a class in C#: Right click on the project / Add / Class / Name the class / Click Add. You will get this:

```
namespace ProjectName
{
        class ClassName
        {

        }
}
```

```
//Class example for Animal.cs
namespace ProjectName
{
        class Animal
        {
                private string type;
                private string color;
                private string weight;
                private string height;
                private int age;
                private in NumOfLegs;

                public void move() // The methods are public in order to be available for calling.
                {
                }

                public void makeNoise()
                {
                }
        }
}
```

```
//Program.cs; We can use the class here.

namespace ProgramName
{
	class Program
	{
		static void Main(string[] args)
		{
			// 1st Animal = Type of the variable.
			// new Animal = Variable name (instance / object name).
			// new = Instantiation keyword.
			// 2nd Animal = Class name.

			Animal newAnimal = new Animal(); // I want to create a variable "newAnimal" that will store a type of "Animal" in the memory.
								//"new Animal();" is called the constructor.
			Animal.move();
		}
	}
}
```

Encapsulation

First pillar of Object Oriented Programming.

When classes are created, it is important to provide some control over the data that gets assigned to the members. We don't want illegal assignments such as negative ages. In order to control this, the scope private is used which means only code within the class has the ability to change the variables. This is called encapsulation. Encapsulate = Hide implementation. Ex. Stereo vs the electronics. The electronics are encapsulated by the stereo and the user doesn't need to know how it workds internally. The user simply has to use the properties ex. The volume knob.

```
//Class example for Animal.cs
namespace ProjectName
{
	class Animal
	{
		private int age; // This cannot be assigned in Main because of the private scope. Member variables are not capitalized.

		public int Age // This on the other hand can. This is a property and is capitalized. It is "int" because it returns an integer.
		{
			get { return this.age; } // This returns the age from Main. "this" means me, my specific intance. It can work without "this". It is used to 		// specify that it is the age from this class and not some inherited one.
			set
			{
				if ( value < 0)
				{
					Console.WriteLine("Age cannot be less than 0.");
				}
				else
				{
					this.age = value; // Value is a special keyword used in public properties to indicate the incident of the value passed in by
	// the user of our code. Keyword used to bring the age value into the age property.
				}
			}
		}
	}
}
```

```
//Program.cs
namespace ProgramName
{
	class Program
	{
		static void Main(string[] args)
		{
			Animal Dog = new Animal(); // "new Animal();" is called the constructor.
			Dog.Age = 12; // Negative values are prevented by the if statement in the Animal
class.
		}
	}
}
```

Inheritance Second pillar of Object Oriented Programming. All classes inherit from System.Object

```
// Dog.cs // This is a class file.

class Dog : Animal // This means that dog is a sub-class of animal and that it inherits animal's properties
and methods.
{
}

// Program.cs

Dog Spot = new Dog; // This can be created because of the Animal inheritance.
Dog.move(); // Same.
```

Polymorphism Third pillar of Object Oriented Programming.
The ability to modify i.e. add properties and methods to sub-classes.

```
// Dog.cs
class Dog : Animal // Dog inherits many properties from Animal.
{
	public string name; // This property is specific to dogs only.
	public string owner; // This property is specific to dogs only.

	public override void move ()
	{
		Console.WriteLine("Running");
	}
}

// Animal.cs
public virtual move () // The addition of the word virtual adds the possibility of overriding the method with a
new sub-class method.
{
	Console.WriteLine("Moved")
}

public virtual makeNoise()
{
	Console.WriteLine("Made noise")
}

// Program.cs
Dog Spot = new Dog();
Dog.Age = 5; // This can be assigned because Dog inherits from Animal.
Dog.move(); // This will return "Running" because the method in Animal.cs was overriden by the method in
Dog.cs
Dog.makeNoise(); // This will return "Made noise" because there is no overriding method.
```

Namespaces

Logical containers for classes. Namespaces provide a notional separation for classes, class libraries provide a physical separation (in windows think a standalone dll). Class libraries are useful for when you want to wrap up functionality that can be shared with other projects.

Libraries can be organized inside with namespaces.

Namespaces help control the scope of the classes, prevent duplicate class names when using multiple vendor code.

C# provides the using directive to help shorten namespace.class.method typing in code. Ex. Instead of writing: System.Collections.Generic.Dictionary... one can type using System.Collections.Generic onace at the top, and then simply write Dictionary in the code.

Renaming a namespace: Right click on the namespace / Click refactor / Rename / Choose name and which instances to be changed.

Object Browser Window

This is used for searching prewritten classes i.e. methods in the .NET library so that you don't have to write them. The location is View / Object browser. Scope can be chosen as well as the libraries. The classes are on the left, and the methods and properties on the right.

Classes revision:

```
// Class creation
namespace Cars
{
    class Car
    {
        private string make; // Moze i vaka namesto dolnoto: "public string make {get; set;}" taka sto namesto myCar.Make ke bide myCar.make.
        private string model;
        private string color;

        public string Make
        {
            get { return this.make; }
            set { this.make = value; }
        }

        public string Model
        {
            get { return this.model; }
            set { this.model = value; }
        }

        public string Color
        {
            get { return this.color; }
            set { this.color = value; }
        }

        public void Drive() {
            Console.WriteLine("Driving");
        }
```

```
        public void Stop()
        {
            Console.WriteLine("Stopping");
        }
    }
}

// Class invoking
namespace Challenge_3
{
    class Program
    {
        static void Main(string[] args)
        {
            Car newCar = new Car();
            newCar.Make = "Ferrari";
            newCar.Model = "F50";
            newCar.Color = "Red";

            newCar.Drive();
            newCar.Stop();
        }
    }
}
```

Exception Handling

- An exception is an unexpected event in the code. Exception handling is done by wrapping code in protective blocks which will monitor for exceptions to happen.
- Exceptions are passed up the call stack until a handling routine is found or the program crashes.
- Use the general exception to catch specific exceptions which can later be handled specifically for each exception.

```
try { function }
catch { exception }
finally { function fix }
---------- Challenge -----------------------------------------------
int intValue = 32;
object objectValue = intValue;
string strValue;
// string errorMessage; Ova ne mora radi dole.

try { strValue = (string)objectValue; }

catch (Exception e)
{
            // errorMessage = e.Message. Ova ne mora zatoa sto direktno e err.Message a ne (errorMessage).
            Console.WriteLine(e.Message); // The error is: Unable to cast object of type 'System.Int32' to type 'System.String'.
}
-------------------------------------------------------------
Overflow exception = A number too big for the chosen data type.
Divide by Zero exception = Divide by zero duh.
```

The throw method allows for custom error messages as well as throwing (redirecting) the error to an exception handling class that logs error messages.

Resource Management

Stack vs Heap: http://youtu.be/_8-ht2AKyH4

Memory is allocated for every type.
Garbage collectors deal with reference types.
Value types are stored on the stack and they go in and out of scope automatically.
Reference types are stored on the heap and they remain in scope until no longer needed.
C / C++ have no auto garbage collectors and the programmers are responsible for releasing resources.

Garbage collections does a performance heat, but it does so periodically when the system is idle i.e. it doesn't run all the time. It does trigger when the heap is full.

Destructors

We generate class files to form the definition of the objects that will be used in the code. When an object is instantiated, a constructor is used to build the object i.e. to initialize instance variables, setting values for the member variables.

A destructor is used for tearing down objects and is the opposite of a constructor.

```
Class Dog
{
        public Dog(); // Constructor
        { // Initialization statements }

        ~Dog(); // Destructor
        { // Cleanup statements }
}
```

A destructor can only be used for classes and not for structures. This is because classes are reference types and structures are value types.
There can be only one destructor and it cannot be overloaded or inherited. It also takes no parameters or modifiers.
The destructor cannot be called because it is automatically invoked by the garbage collector.

Extensible Types

Generics

Used for creating classes and methods decoupled from data types, which allows code to be reused with any data type i.e. makes the code type independent.

In other words, instead of writing many functions to compare values for each data type, you write one generic comparison function, and later specify which data type to be used when the function is called.

```
public static bool areEqual<T>(T Value1, T Value2) // T is a dayta type placeholder. It can be anything, not just T.
{
        return Value1.Equals(Value2);
}

areEqual<string>("A", "A");
areEqual<int>(10, 10);
```

This way, you avoid writing separate functions for comparing ints and strings.

Boxing = Converting value types into reference types.

Collections

Non Generic Collections: ArrayList, Stack, Queue, SortedList
Generic Collections: Stack<int> intStack = new Stack<int>(); // The same can be done for the other types.

SortedList<int, string> list = new SortedList<int, string>();
list.Add(1, "one");
list.Add(2, "two");
list.Add(3, "three");

Foundations of Programming - Object Oriented Design

2 Mistakes are common:
1. Not using paper for planning and going straight to coding. This can lead to having to re-do weeks of work just because of a poor plan.
2. Thinking that all the rules (jargon below) will limit creativity. The opposite is actually true. These are less of a rule and more of an idea; a paradigm.

OOP Jargon: Abstraction, Polymorphism, Inheritance, Encapsulation, Composition, Association, Aggregation, Constructors, Destructors, Cardinality, Singleton, Chain-of-

Responsibility, Class-Responsibility-Collaboration...

To write any piece of software, 3 things are needed.
1. Analysis: Understand the problem.
2. Design: Plan the solution.
3. Programming: Build it.

There is no one or right way to make software from start to finish. There are many different methodologies that provide results. Here are some of the formal ones:
- SCRUM
- Extreme Programming (XP)
- SSADM
- Unified Process
- Agile Unified Process
- Feature-Driven Development (FDD)
- Cleanroom
- Adaptive Software Development
- Crystal Clear
- Behaviour-Driven Development
- Raid Application Development

Waterfall vs Agile / Iterative approach of design:
Waterfall is a structured step by step process where everything is anticipated and linear. This would work for building a bridge, not so much for software. New problems arise, bugs happen, people change their minds... All these things make the waterfall approach inefficient. That's why the agile / iterative approach is used; it allows constant code revision and is good enough to move forward. It doesnt have to be perfect.

Core Concepts of Object Orientation

First there were procedural languages: Assembly, C, Cobol, Fortran. The programs were written as one long piece of code. They soon started to be difficult to manage.

Object oriented programming is the solution to the problem. The one long piece of code is now divided into separate self contained objects, almost like having several mini programs each representing a different part. Each object contains its own data and logic while communicating between themselves.

There are alternatives to OOP, but they have very specific areas they cover, ex. Prolog is a logic programming language and Haskell which is functional programming language. They are heavily used in science, but are inferior for practical use.

Objects

If everything is revolving around object orientation and everything is an object... What is an object? Object orientation was invented solely for the purpose of making thinking about programming closer to the real world.What is an object in programming is the same as asking what is an object in the real world? Pretty much everything.

An object has 3 things ex. Coffee mug:
1. Identity i.e. Uniqueness. Each mug exists separately even though it can be the same as another one.
2. Attributes i.e. Characteristics / Properties. These describe the state of an object and are independent from other. The mug can be full or empty. It can also be black or white at the same time. An object can have many attributes.
3. Behaviour i.e. Methods.

Objects in programming are self contained i.e. the have a separate identity from other objects.

Ex. Bank Account Object:

Attributes:
balance: $500
number: A7652

Behaviour:
deposit()
withdraw()

Real world objects are physical things. In programming, they can be abstract. Ex. Date, Time, Bank Account. You cannot see nor touch them, but they exist. Also, in programming they can be even more abstract ex. Button vs an Array. The button can be seen.

Objects are not always physical or visible items.

But how can you tell what can be an object, aside from obvious ones like car, employee...?

Guidelines:
1. Is the word a noun?
2. Can "the" be put in front of it? The time, the date, the event...

Ok, but how are they made? Say hello to classes.

Classes

Objects and classes go hand in hand because they are created by them.
A class describes what an object will be, but it isn't the object itself. A class is a blueprint for a house, while the object is the house itself. One blueprint can be used for building 1000+ houses.

The class comes first and it is what we are writing. They describe what would an object be and what can it do.

What is a class?
1. (Type) name: What is it? Employee, Bank Account, Event...
2. (Properties) attributes: What desribes it? Height, Color, FileType...
3. (Methods) behavior: What can it do? Play, Open, Search, Save, Delete, Close...

Each object is an instance of a class. Creating objects = Instantiation.

Ex. Class:

Type: BankAccount

Properties:
accountNumber
balance
dateOpened
accountType

Methods:
open()
close()
deposit()
withdraw()

Most OOP languages provide many pre-written generic classes at minimum: strings, dates, collections, file I/O, networking + Many more. These are gathered in frameworks i.e. libraries like: Java Class Library, .NET Framework BCL, C++ Standard Library, Python Standard Library...

The 4 Fundamental Ideas in OOP - APIE:

Abstraction, Polymorphism, Inheritance, Encapsulation

These are the 4 ideas to keep in mind when creating classes: APIE - Abstraction, Polymorphism, Inheritance, Encapsulation.

These 4 words may sound intimidating, but chances are, you use them daily in normal conversations; just under different names.

Abstraction

It is at the heart of OOP. Focuses on the essentials of being.

If you are asked to think of a table, you can imagine it without knowing the material, color, size, number of legs, shape... You just know what the idea of a table is i.e. the abstraction of a table. This means you have seen enough real tables to abstract the idea of what a table means.

Abstraction means that we focus on the essential qualities of something, rather than one specific example. It also means that we discard what is irrelevant and unimportant. A table can have a height and width, but it is unlikely to have an engine size or flavour.

Abstraction means that we can have an idea or a concept that is completely separate from any specific instance. Instead of creating separate classes for each bank account, we create a blueprint for all bank accounts.

It becomes a question of not "What does a bank account class look like?", but rather "What should a bank account class look like?"; for this specific program at this time. The focus is on the bare essentials.

Encapsultion

You hide everything about an object except what is absolutelly necessary to expose.

Think of a capsule which not only keeps things together, but also protects them.

Encapsulation is the bundling of the attributes (properties) and behaviours (methods) in the same unit (class), while restricting access to the inner workings of that class or any objects based on that class.

An object should not reveal anything about itself except what is absolutely necessary for other parts of the application to work.

We don't want some other part of the application to be able to reach in the bank account class and change the account balance of an object without first going through the methods which can control things. To prevent this, we can hide the attribute and control the access so that it is only accessable from inside the object itself via the methods. This way, other parts of the applications can change the attributes but it would still be controlled by the methods and it could not be changes otherwise from the outside.

Think of the term "blackboxing", which is hiding the inner workings of a device while only providing a few public pieces for input and output.

We don't care what's inside a phone as long as we have dials to press and be able to call someone.

The most common question is: "I'm writing these classes, so why would I hide my own code from myself?" A: It's not about being secretive. It's about reducing dependencies between different parts of the application. A change in one place should not cascade down and require multiple changes elsewhere. If the attribute is accessed from within i.e. it's hidden from the outside, a change would only have to be made on the method accessing it rather than everything needing it from the outside.

How much should you hide? - As much as possible!

The effort put into abstracting and encapsulating our classes will be very useful when creating other classes with inheritance.

Inheritance

A great form of code reuse. Also, the foundation for polymorphism.

Inheritance is a great form of code reuse. Instead of writing a new class from scratch, we can base it on an existing one.

Ex. Customer inherits from Person. Person = Parent/Superclass, Customer = Child/Subclass.

Polymorphism

Lets you do the right thing at the right time.

Ex. 1 + 2 vs "Hello" + "World". Addition vs Concatenation. They both share the + sign, but behave differently. The first is 3, the second is HelloWorld.

Ex. Parent: BankAccount. Child: CheckingAccount, SavingsAccount, InvestmentAccount. They all inherit from BankAccount, but some need to behave differently.
The BankAccount has withdraw(). The InvestmentAccount inherits the method but it has to be changed so when you withdraw from the account without a 30 day notice you get a penalty. That change is called "Overriding the method of the parent class". You inherit the useful, and override the things that need change.

The beauty of this is that the withdraw() method can now be called without the need for a different function name as well as without knowing which class the object was instantiated from.

OOP Analysis and Design

The whole point of this is to determine which classes are needed and what do they do.

5 step OOP Analysis and Design Process:

1. Gather "Requirements". What does the app need to do i.e. what problem is it trying to solve. It should be very specific.
2. "Describe" the app. How would people use the app? Describe this via use cases and user stories. Sometimes the UI is essential, and sometimes it's a distraction.
3. "Identify" the main objects. What does the application revolve around? You pick these from the user stories. These almost always become the classes.
4. Describe the "Interactions". What happens? A customer opens a bank account. A spaceship explodes when it touches an asteroid. We use a sequence diagram for order.
5. Create a "Class Diagram". Visual representation of the needed classes. Here you get specific about OOP Concepts i.e. AEIP (APIE).

The goal is to get the class diagram. This process is not done only once. It is constantly revisited for refinement.

Requirements

CORE: Functional Requirements: What does it do? ex. Application must allow user to search by customer's last name, telephone number or order number.

Non-Functional Requirements: What else? ex. System must respond to searches within 2 sec.
- Help. What kind of documentation needs to be provided?
- Legal. Are there any laws to comply to? Who knows these laws?
- Performance. Response time? How many people can use it at once?
- Support. What happens if there is a problem at 2 am on a sunday?
- Security. It can be functional or non-functional depending on the app.

This should be used in every case, regardless of it being for a client or self.

A team may think this can be skipped because they know all the requirements, but the problem is that they all have semi-formed ideas about what the application COULD DO. The goal is to WRITE DOWN what the application MUST DO. You can't design half a feature.

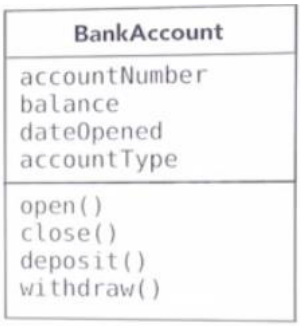
The bigger the application and organization, the more formal the process needs to be.

One common formal requirements approach is FURPS / FURPS+. This is a checklist, not instructions. Focus on MUST HAVE, instead of NICE TO HAVE.
- Functional. App features.
- Usability. Help, documentation, tutorials...
- Reliability. Disaster recovery, acceptable fail rate...
- Performance. Availability, capacity, resources...
- Supportability. Maintenance, scalability.

+
- Design. Must be an iPhone app, must use relational databases.
- Implementation. Which technologies? Languages?
- Interface. Not UI, but rather the need to interface with an external systems.
- Physical. Needs to run on a device with a camera, must ship with 50GB DVDs.

UML: Unified Modelling Language

It is a graphical notation, not a programming language. Diagrams for OOP design.
It is more useful to know a little than a lot of UML. UML can become the main focus, which is a mistake.

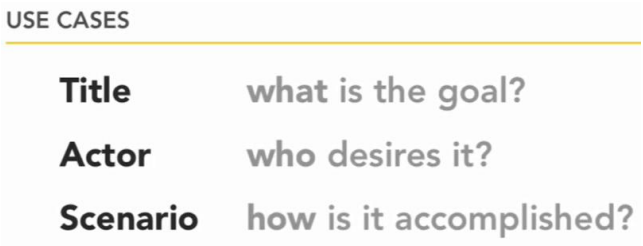


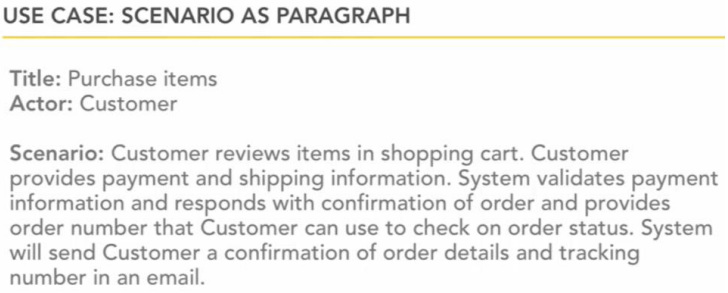
Class Diagram

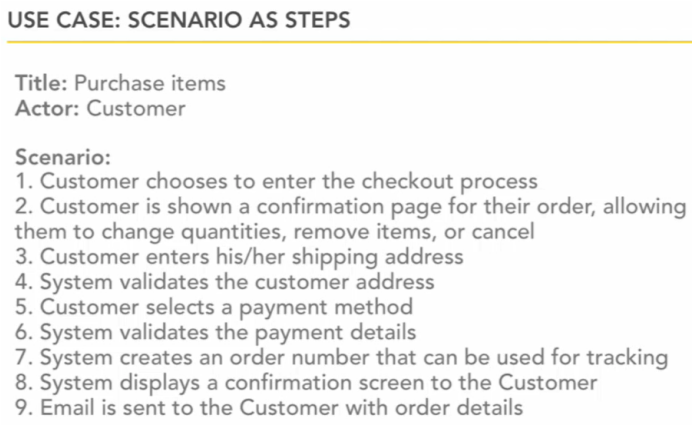
Use Cases

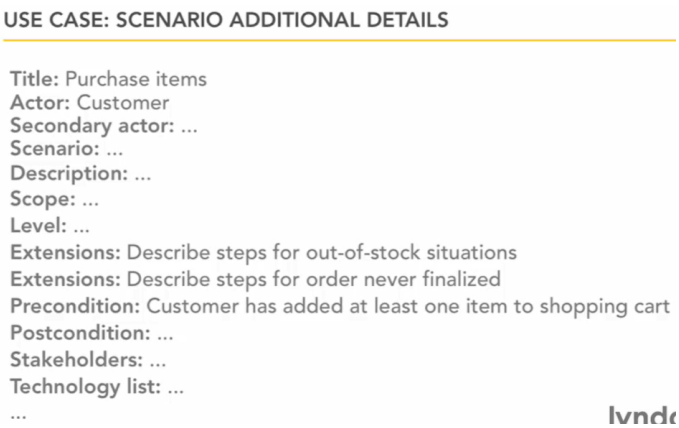
Here we focus on the user instead of the features of the program. How does the user accomplish something?

Informal vs Fully dressed use cases. The latter can hinder process for small projects, but is necessary for major global ones.









1. Title. What is the goal? Short phrase, active verb. ex. Create new page, Purchase items, Register new member...

2. Actor. Who desires it? Instead of simple user, use Customer, Member, Administrator... It doesn't have to be a human. It can be a system. A simple game can have just a user, whereas a corporate application can have multiple actors with different job titles and departments. Also within that same application, some processes require special actors regardless of job titles. ex. Requester and Approval for an Expense Approval System.

3. Scenario. How is it accomplished? Paragraphs vs Lists (Step by step guides). Preconditions and extensions can be defined. ex. Precondition: Customer must select one item. ex. If the item is out of stock. The focus should be on specific actions. ex. Log into system. Why does one log into? The user logs in to do something, not just to log in. Log in is too broad and simply a step towards the important goals i.e. the do something. ex. Purchase something, Create new document.
Write a sunny day scenario first where everything works in order. Later add the scenarios for everything that can go wrong from the actions side, not the technical. ex. Item is out of stock vs .NET is outdated.

Use ACTIVE voice. To the point without too many details.

BAD: The system is provided with the payment information and shipping information by the Customer.
GOOD: Customer provides payment and shipping information.

BAD: The system connects to the external payment processor over HTTPS and uses JSON to submit the provided payment information to be validated, then waits for a delegated callback response.
GOOD: System validates payment information.

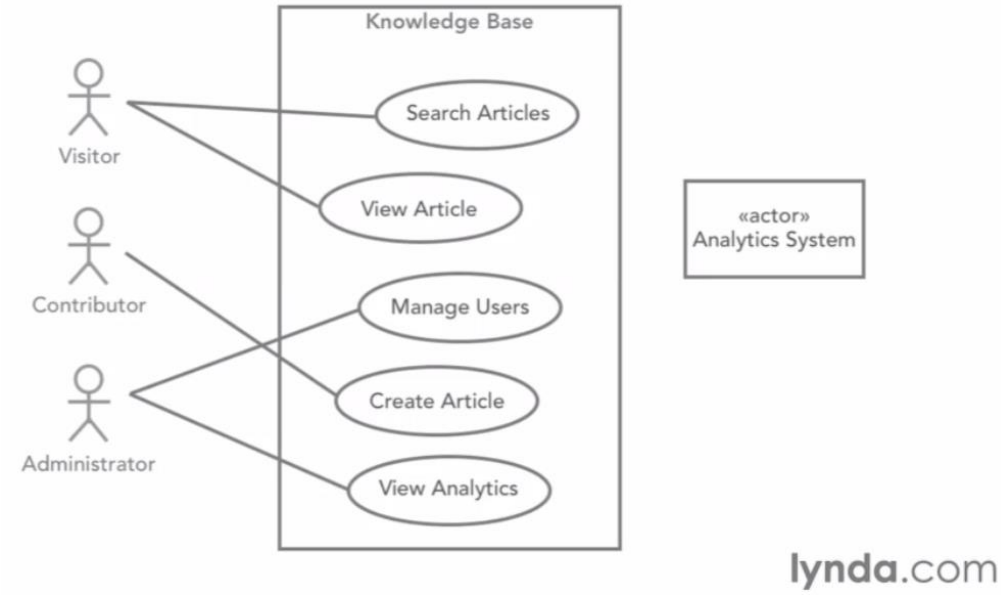
FOCUS ON INTENTION!

Ex. Provide steps without mentioning clicks, pages, mouse, buttons etc... Just the pure intent.

Questions that can provide unforseen actors and scenarios.
- Who does system administration tasks?
- Who manages users and security?
- What happens if the system fails?
- Is anyone looking at performance metrics and logs?

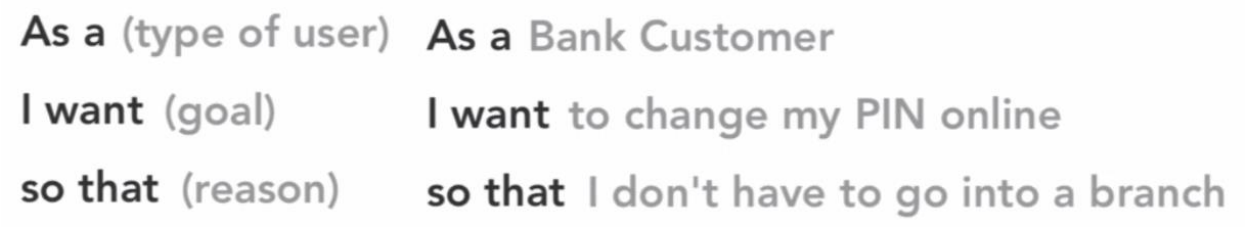
Use Case Diagram

It is not a diagram of a single use case, but rather a diagram of multiple use cases and actors at the same time. It provides an overview of how they interact and it is not a replacement for written use cases.



User Stories

They are simpler and shorter than use cases. It describes a scenario from a user perspective with the focus on their goal, rather than the system. They are usually 1 or 2 sentences long, in comparison to use cases which can be pages long.



Differences between Use Case and User Story

USER STORIES	USE CASES
short - one index card	long - a document
one goal, no details	multiple goals and details
informal	casual to (very) formal
"placeholder for conversation"	"record of conversation"

Domain Modelling

Creating a conceptual model = Identifying objects from use cases and stories.

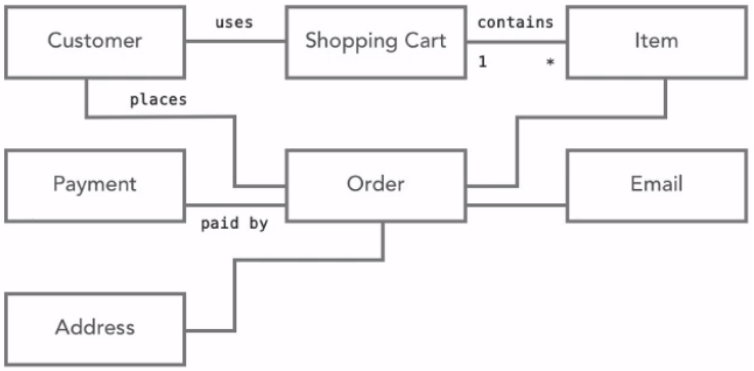
Use Case Scenario: Customer confirms items in shopping cart. Customer provides payment and address to process sale. System validates payment and responds by confirming order, and provides order number that Customer can use to check on order status. System will send Customer a copy of order details by email.

Noun List

Customer
Item
Shopping Cart
Payment
Address
~~Sale~~
Order
~~Order Number~~
~~Order Status~~
~~Order Details~~
Email
~~System~~

Conceptual Object Model

The goal here is to make a conceptual object model, not a database model. This is purely for planning purposes, not execution.

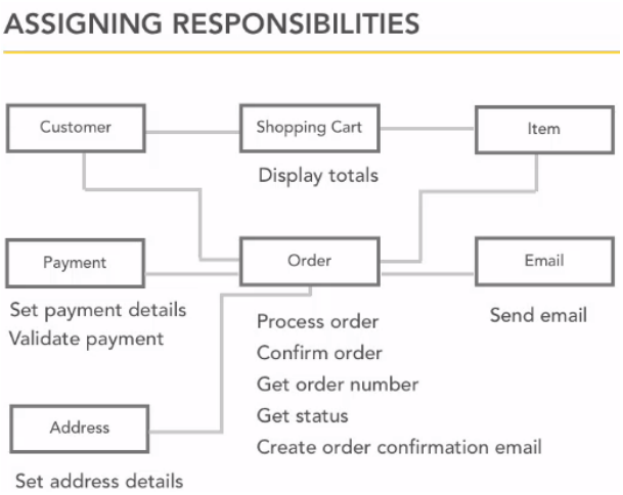


Look for the verbs. An object should be responsible for itself. Don't confuse the actors with this. They initiate the behavior that lives in the objects.

IDENTIFYING RESPONSIBILITIES

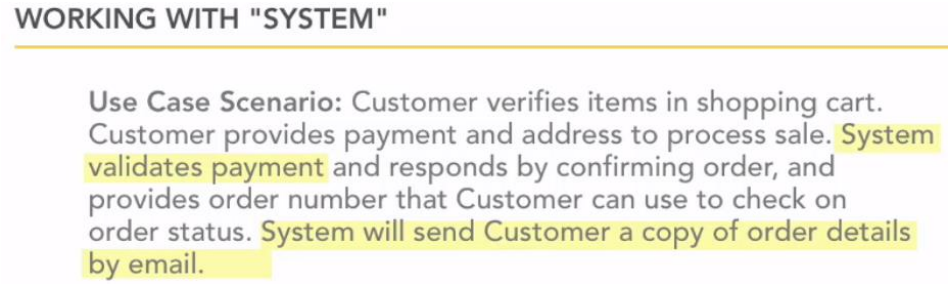
Use Case Scenario: Customer verifies items in shopping cart. Customer provides payment and address to process sale. System validates payment and responds by confirming order, and provides order number that Customer can use to check on order status. System will send Customer a copy of order details by email.

Verify items	Confirm order
Provide payment and address	Provide order number
Process sale	Check order status
Validate payment	Send order details email



It may sound like many of these should be under the customer, but keep in mind that the customer is the initiator.

It's a common mistake for people new in OOP to give way too much behavior to actors.



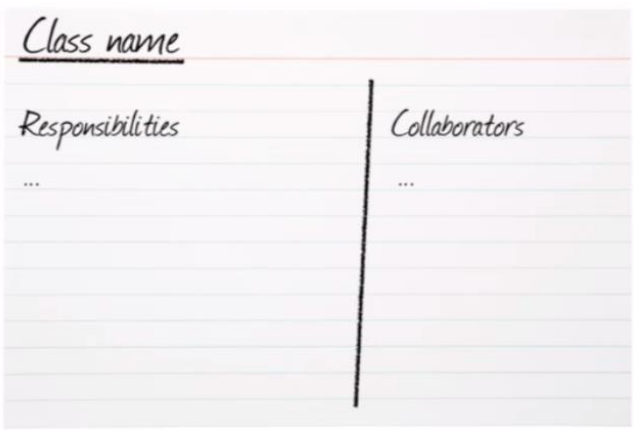
This can lead to people creating a "System" object and putting a lot of behavior in it. This is a big mistake. "System" is just a placeholder word for the specific object that has not yet been identified. It should be read as "SOME PART of the system does something…".

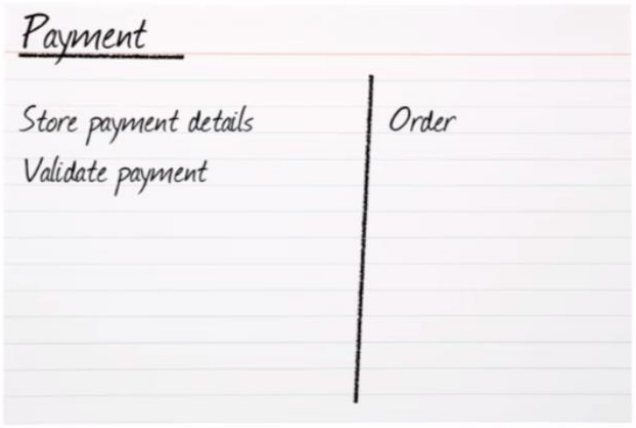
Responsibilities should be distributed between objects, not stored in one master object. This is a sign that you are thinking procedurally, not OOP.

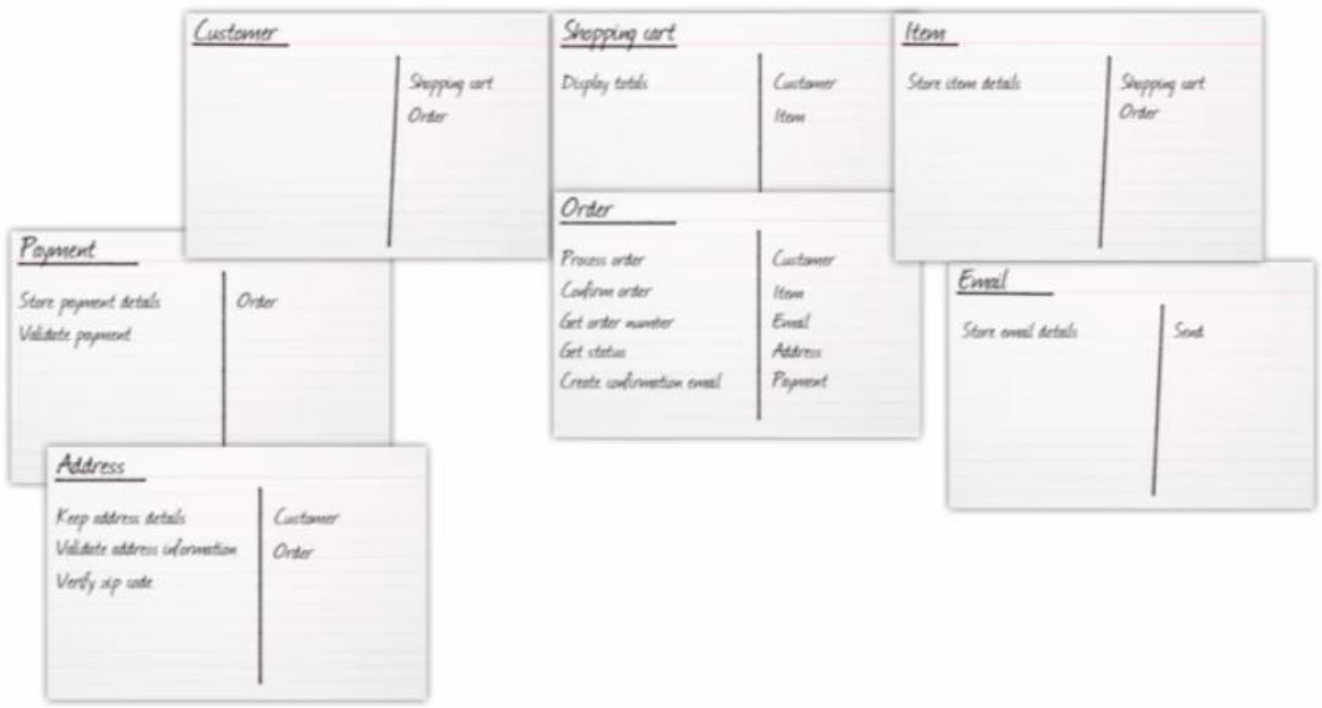
CRC – Class Responsibility Collaboration

1 CRC card = 1 Class

Avoid using software for this stage. There is a great advantage in the physicality of index cards.







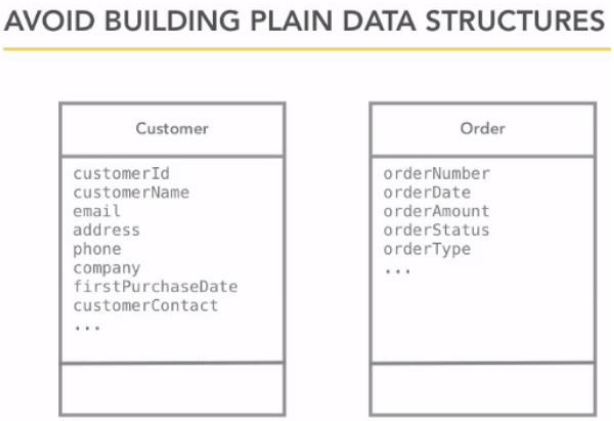
Creating Classes

Classes are named in singular with uppercase first letter.

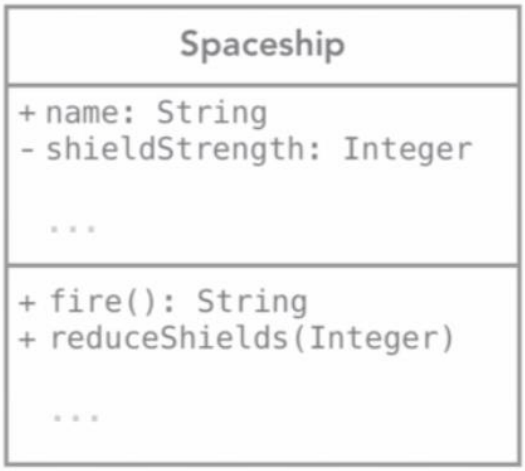
: Data Types, + = Public,- = Private

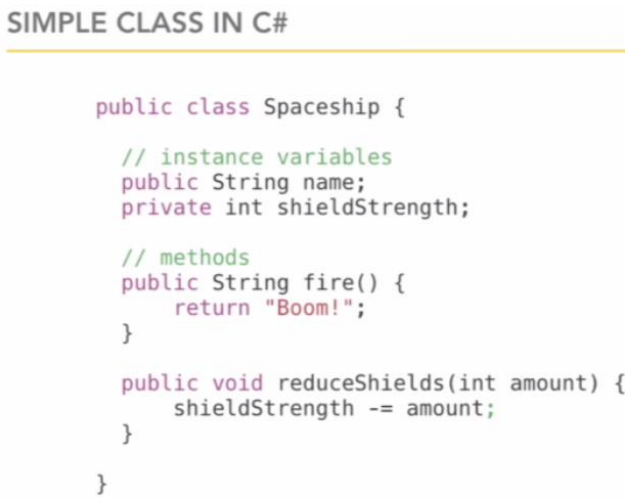
Everything should be as private as possible.

The initial focus should be on behavior / responsibilities, not properties. If the class diagrams have a ton of properties and no methods, it is a sign of wrong priorities.



Converting Class Diagrams to Code





Object lifetime

INSTANTIATION

Java	`Customer fred = new Customer();`
C#	`Customer fred = new Customer();`
VB.NET	`Dim fred As New Customer`
Ruby	`fred = Customer.new`
C++	`Customer *fred = new Customer();`
Objective-C	`Customer *fred = [[Customer alloc] init];`

If you want something else to happen i.e. take part in the instantiation of an object, a constructor is used.

A constructor is a special method that exists to construct objects. A constructor can make sure that any variables belonging to an object are immediately set to the right values as soon as the object is created.

The constructor is used to be able to set dynamic values to objects upon their creation. If you do not use a constructor you are only able to set initiated values to a specific value.

Constructors are an OOP concept and are not available in procedural languages.

No Constructor

With Constructor

In UML, if there is a method with the same name as a class, it means that it's a constructor.

With 2 Constructors

```
public class Spaceship {

  // instance variables
  public String name;
  private int shieldStrength;

  // constructor method
  public Spaceship() {
      name = "Unnamed ship";
      shieldStrength = 100;
  }
  // overloaded constructor
  public Spaceship(String n) {
      name = n;
      shieldStrength = 200;
  }
  // other methods omitted

}
```

```
Spaceship excelsior =
    new Spaceship("Excelsior 2");
```

object: excelsior

```
name: Excelsior 2
shieldStrength: 200
```

Overloaded constructors allow flexibility and information can be passed in when creating an object.

Static Variables and Methods

Static Variables and Methods are the same across all objects. This is used in order to access the variables from one place rather than individually for all created objects.

Static variables can be accessed ONLY with static methods.

Variables are called with the objects name whereas the static ones are called with the class name.

Ex. joeAcct.accountNumber vs SavingsAccount.interestRate

In UML they are denoted with underlined names.

Identifying Inheritance Situations

Are there shared attributes and behaviors between our objects?

INHERITANCE DESCRIBES AN "IS A" RELATIONSHIP

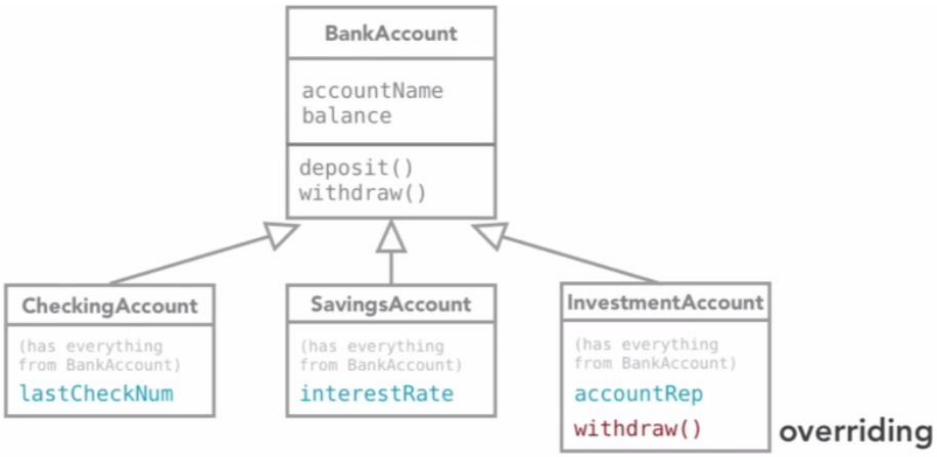
A car is a vehicle.
A bus is a vehicle.
A car is a bus.

An employee is a person.
A customer is a person.
A customer is a shopping cart.

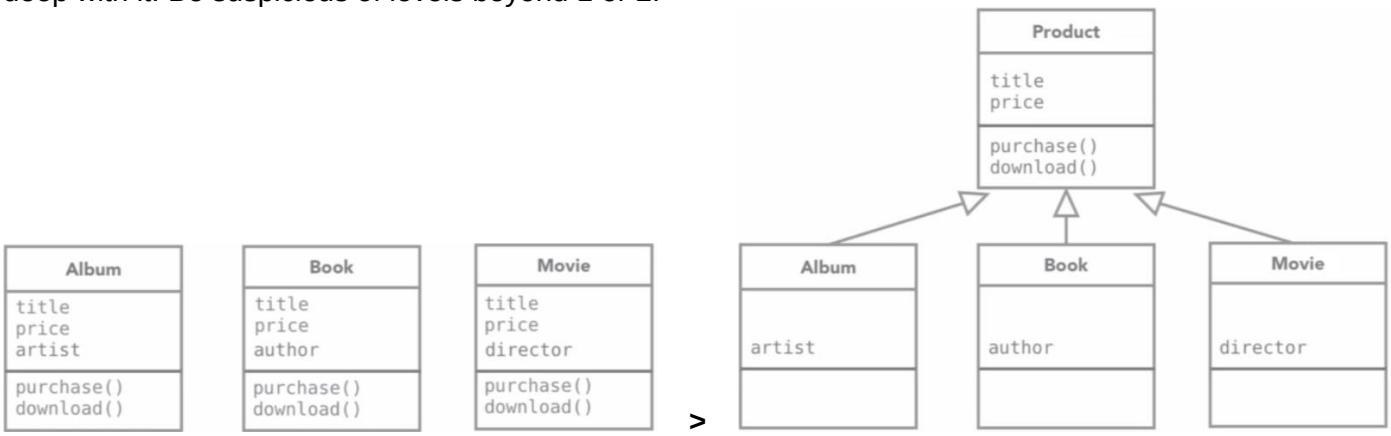
A checking account is a kind of bank account.
A savings account is a type of bank account.

A Bentley Continental GT is a car is a vehicle.

A Pomeranian is a dog is a mammal is an animal.



Don't go actively looking for inheritance. It should come naturally and be obvious. It is a mistake to go too deep with it. Be suspicious of levels beyond 1 or 2.



Using Inheritance

```
Java        public class Album extends Product {  ...
C#          public class Album : Product {  ...
VB.NET      Public Class Album
               Inherits Product ...
Ruby        class Album < Product   ...
C++         class Album : public Product {  ...
Objective-C @interface Album : Product {  ...
```

CALLING A METHOD IN THE SUPER / PARENT / BASE CLASS

```
Java        super.doSomething();
C#          base.doSomething();
VB.NET      MyBase.doSomething()
Ruby        super do_something
Objective-C [super someMethod];
C++         NamedBaseClass::doSomething();
```

Abstract classes are never instantiated and exist purely to be inherited from.

Some languages (Java, C#, C++) have the ability to mark a class as **abstract** when created, which means it won't allow to be instantiated as well as the requirement for the child objects to inherit from it before instantiation.

Interfaces

It is created like a class but it has no properties and behavior. It is not allowed to create functionalities in them. If we create a new class and we decide to implement and interface, it is like signing a contract. We are promising to create methods with particular names. If we don't, we will get a compiling error.

DEFINING INTERFACE CONTRACTS

```
interface Printable {

   // method signatures
   void print();
   void printToPDF(String filename);

}
```

IMPLEMENTING INTERFACES

```
class MyClass implements Printable {

    // method bodies
    public void print() {
        // provide implementation
    }

    public void printToPDF(String filename) {
        // provide implementation
    }

   // additional functionality...

}
```

The benefit of this is that we can have many different classes implementing the same interface, and other parts of the app can use objects with interfaces withoug knowing anything about how the work.

Many programmers prefer using interfaces over inheritance. This way the programmer is free to choose how to implement the methods, rather can be provided with them.

"Program to an interface, not an implementation" – Design Patterns, 1995

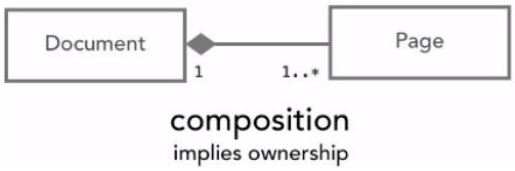
Aggregation and Composition

AGGREGATION DESCRIBES A "HAS A" RELATIONSHIP

A customer has a address.
A car has a engine.
A bank has many bank accounts.
A university has many students.

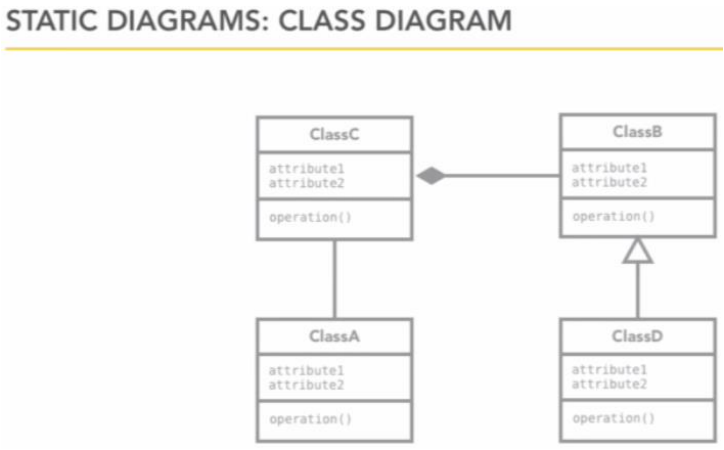
Composition is Aggregation with the difference that Composition implies ownership i.e. when he owning object is destroyed, so are the contained ones.

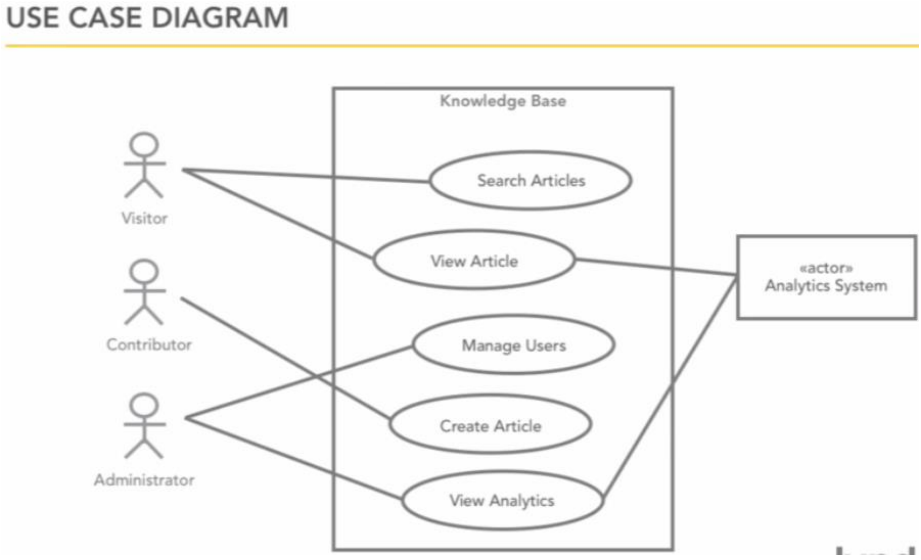
aggregation



composition
implies ownership

Advanced Concepts

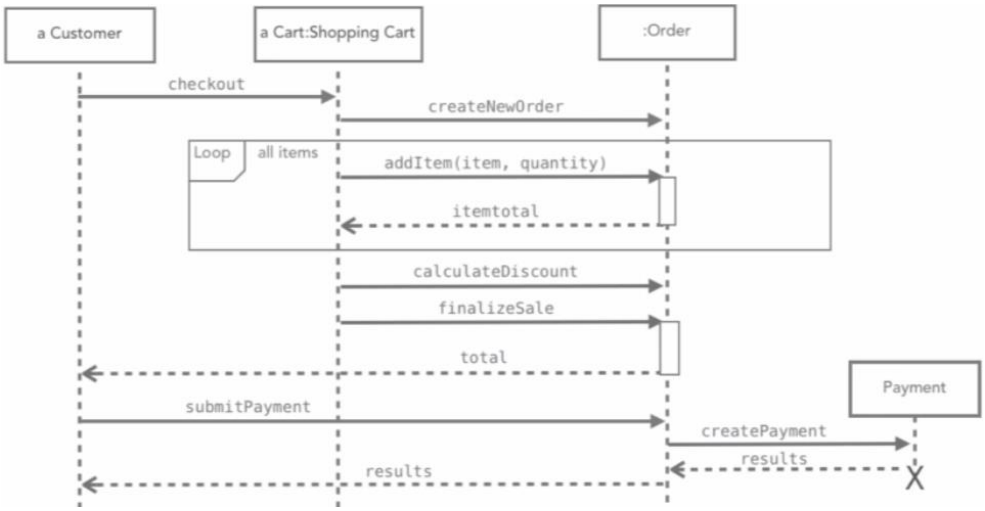




These are structural i.e. static diagrams. They are great for providing an overview of the classes with their compositions and actors. But, they are not good for representing the lifetime of objects. For this we use behavioral / dynamic diagrams. They describe how objects change and communicate, and the most common one is the sequence diagram.

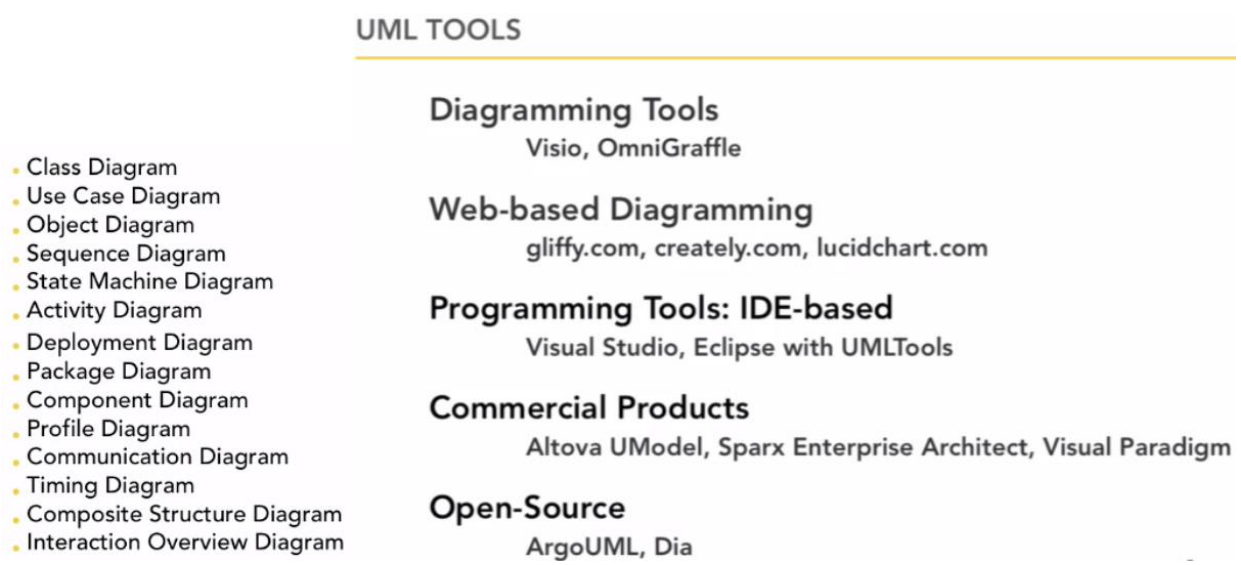
Sequence Diagrams

These don't describe systems, but rather particular parts of them. These are good for sketching ideas that are not completely clear and are not supposed to be used for every single process. These are simple planning tools.



UML Diagrams

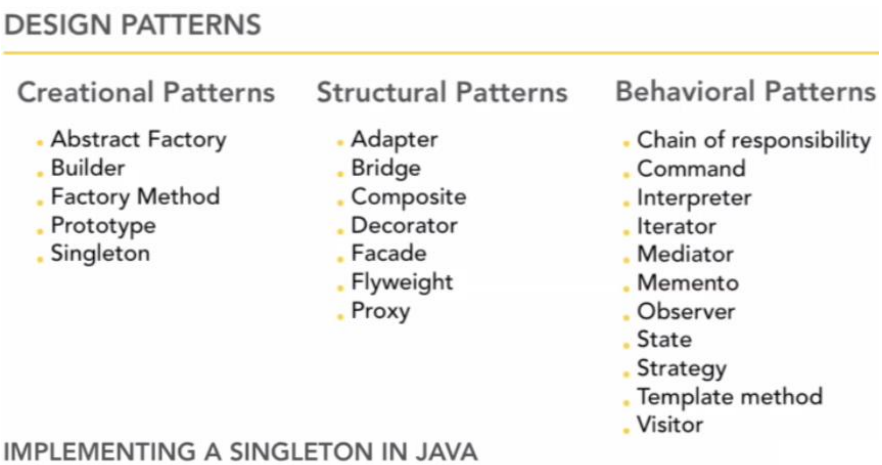
The first 4 are most common. It's always about selecting the right diagram for the right need. Their use should be driven by a business problem. You shouldn't be asking the question "Where can I write some sequence diagrams", but simply realizing where would one be useful when thinking about a situation that isn't clear.



Design Patterns

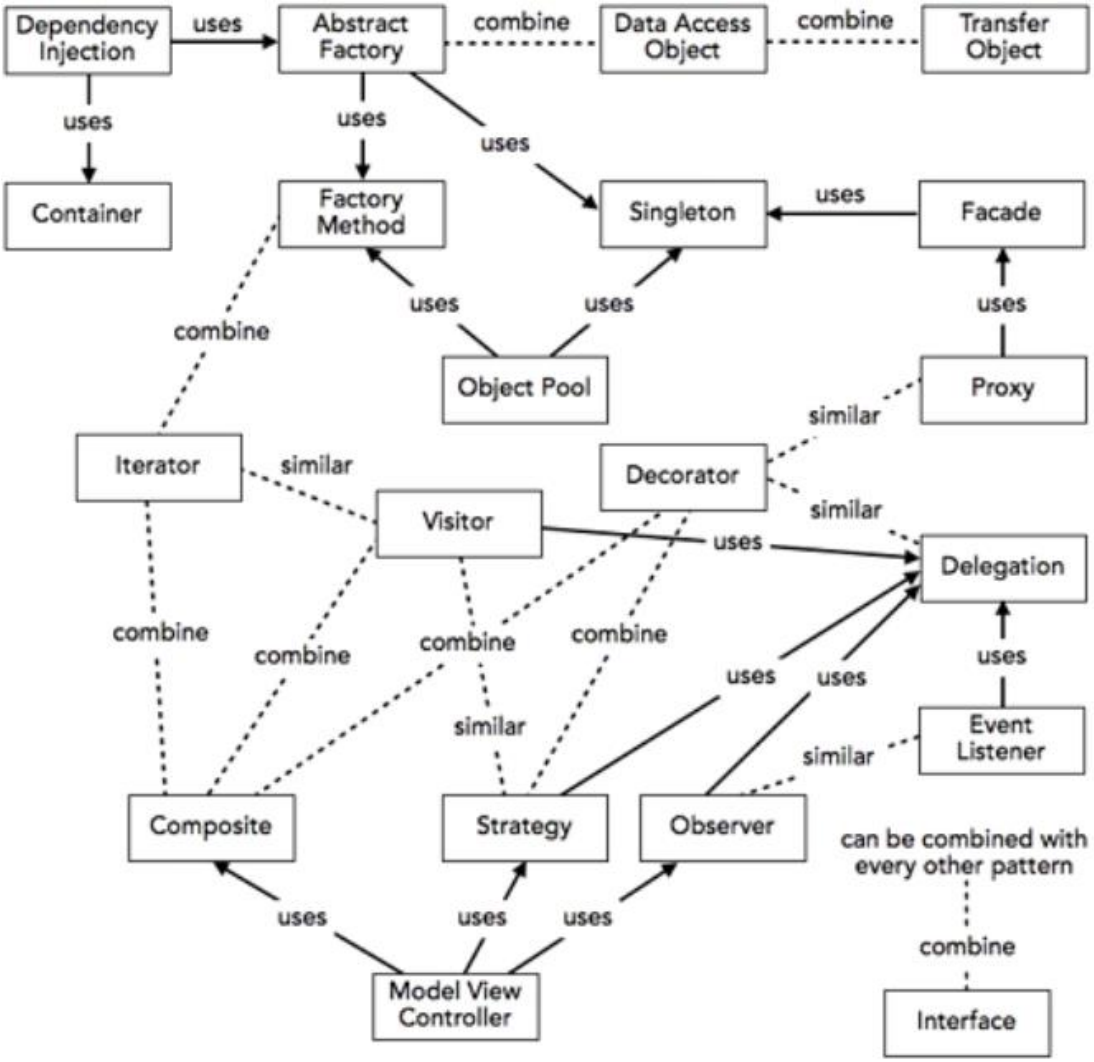
These are abstract ways to organize your programming. Think of design patterns as suggestions / best practices on how to organize classes and objects to accomplish a task. They are best practices for commong programming problems.

The below is based on a 90's book called "Design Patterns" written by GoF i.e. The Gang of Four. "Elements of reusable object oriented code".



IMPLEMENTING A SINGLETON IN JAVA

```
public class MySingleton {
    // placeholder for current singleton object
    private static MySingleton __me = null;
    // private constructor - now no other object can instantiate
    private MySingleton()  { }
    // this is how you ask for the singleton
    public static MySingleton getInstance() {
        // do I exist?
        if ( __me == null ) {
                // if not, instantiate and store
                __me = new MySingleton();
        }
        return MySingleton;
    }

   // additional functionality
   public someMethod()  { //... }
}
```


Dependency Injection
uses
Abstract Factory
combine
Data Access Object
combine
Transfer Object
uses
uses
uses
Container
Factory Method
Singleton
uses
Facade
uses
uses
uses
combine
Object Pool
Proxy
similar
Iterator
similar
Decorator
Visitor
similar
uses
Delegation
combine
combine
combine
combine
uses
uses
uses
similar
Event Listener
similar
Composite
Strategy
Observer
can be combined with every other pattern
uses
uses
uses
combine
Model View Controller
Interface

Design Principles

Syntax rules are easy, because the program will not compile or it will crash if not done correctly. The difficult part is the OOP design itself because there are no enforced rules.

If inheritance is not used for classes duplicating 90% od the same data, nothing will happen because it would work just fine.

If every member of every class is made public i.e violate encapsulation, nothing will happen.

If every single concept is dumped in one huge class like a procedural language, again it would work.

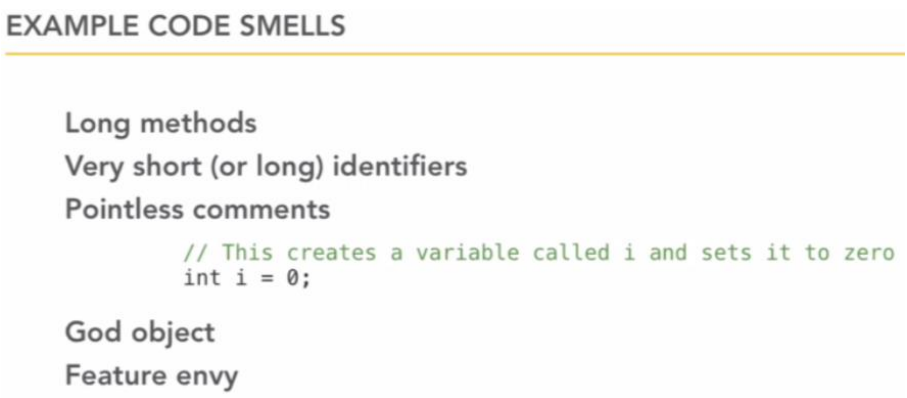
This creates a nightmare for debugging and adding new features. A single change can fracture the whole application.

There are no warnings for bad design. Good OOP practices are not automatically imposed, it is up to the programmer.

General software development principles:

DRY: Don't repeat yourself. // Don't copy past blocks of code.

YAGNI: You aint gonna need it. // Don't write features that are not being used at the moment, i.e. they may be useful in the future. This creates feature creep and code bloat.



Feature envy: A class that inherits everything and has nothing specific in it i.e. why does it exist?

SOLID Prinsiples

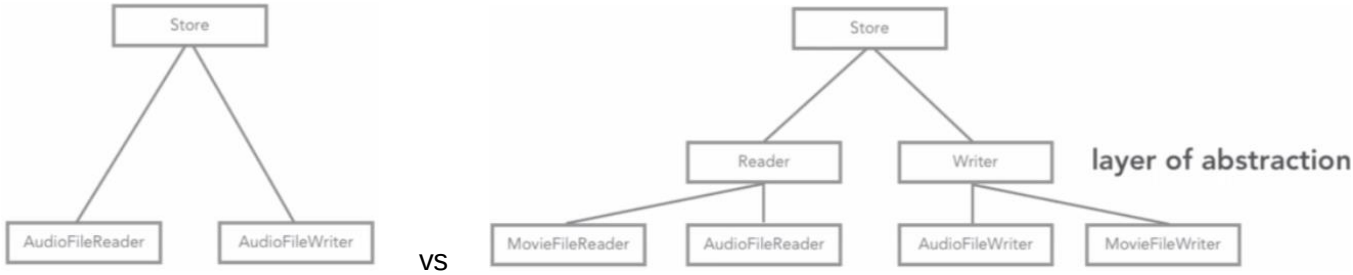
S – Single responsibility: An object should have one reason to exist, and that reason entirely encapsulated in one class. It can have many behaviors, but all of them should fall under that reason to exist. This is against the God Object.

O – Open/Close: Open for extension, but closed for modification. If a child class is added, the new behavior should go under the child class by overriding, not by modifying the parent class.

L – Liskov substitution: Derived classes must be substitutable for their base classes. Child classes should be able to be treated like their parent classes.

I – Interface segregation: Multiple specific interfaces are better than one general purpose interface. Interfaces should be as small as possible. Interface = List of methods to be implemented.

D – Dependency inversion: Depend on abstractions, not on concretions. Layers of abstraction should be added in order to make room for code extension, but be careful not to violate the YAGNI rule. Look at the picture below.



GRASP Principles

General Responsibility Assignment Software Patterns

- Creator
- Controller
- Pure Fabrication
- Information Expert
- High Cohesion
- Indirection
- Low Coupling
- Polymorphism
- Protected Variations

Creator – Who is responsible for creating an object? Questions to be asked for finding out. Does one object contain another (Composition)? Does one object very closely use another? Will one object know enough to make another object? If yes, then that is a creator object.

Controller – Don't connect UI elements directly to business objects. If there is a business class and an UI, we don't want high coupling and low cohesion between them. They should not be tied together i.e. the UI knowing about the business objects and vice versa.

The solution is a new "middle-man" class called the controller with the sole purpose of going between them. This way we don't expect the business object to update the actual screen, nor do we expect the UI elements to talk directly to the business objects. This is the basic idea behind the programming patterns **MVC** and **MVVM**.

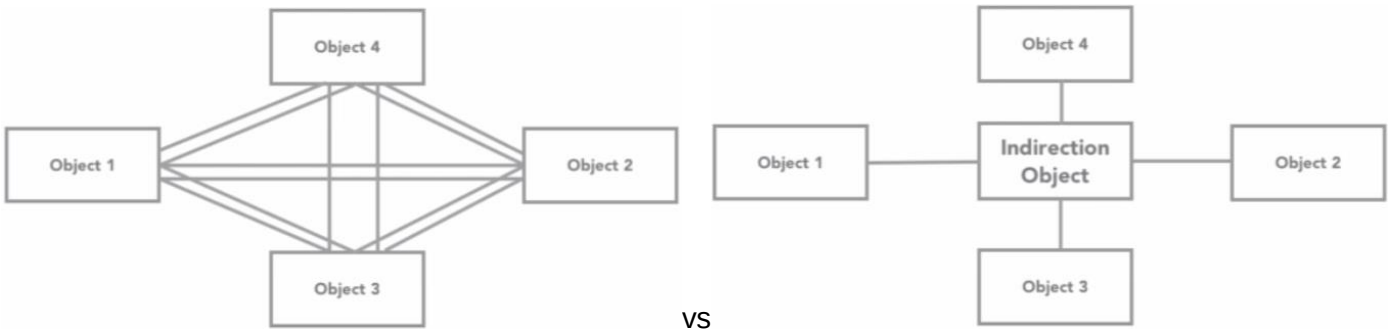
Pure Fabrication – What if something needs to exist in the application that doesn't announce itself as an obvious class or real world object? What if a behavior does not fit any existing class? When the behavior does not belong anywhere else, create a new class. There is nothing wrong with making a class just for pure functionality.

Information Expert – Assign the responsibility to the class that has the information needed to fulfill it. A class should be responsible for itself. Ex. If there are three object: Customer, Shopping Cart and Item; and the customer wants to know the total of the items in the cart. Where would this go? Nothing stops us from putting this inside the customer, but really, it's the shopping cart that knows the most and should be responsible.

Low Coupling – (Coupling: The level of dependencies between objects) Reducing the amount of required connections between objects. As many as necessary, but as few as possible.

High Cohesion – (Cohesion: The level that a class contains focused, related behaviors) A measure of how focused the internal behavior of a class is. High cohesion is when all the internal behaviors are related to the single responsibility. God objects have low cohesion.

Indirection – To reduce coupling, introduce an intermediate object.



Polymorphism – Automatically correct behavior based on type. We don't want conditional logic that checks for particular types.

Protected Variations – Protect the system from changes and variations by designing it so that there is as little impact as possible by them. Do this by implementing everything learned. Ex.

- Encapsulation with private data.
- Interfaces for enforcing formality, but not specific behavior.
- Child classes should always work when treated as their parent classes.

Conclusion

Language	Inheritance	Typing	Call to super	Private Methods	Abstract Classes	Interfaces
Java	Single	static	super	Yes	Yes	Yes
C#	Single	static	base	Yes	Yes	Yes
VB.NET	Single	static	MyBase	Yes	Yes	Yes
Objective-C	Single	static/ dynamic	super	No	No	Protocols
C++	Multiple	static	name of class::	Yes	Yes	Abstract Class
Ruby	Mix-ins	dynamic	super	Yes	n/a	n/a
JavaScript	Prototype	dynamic	n/a	Yes	n/a	n/a

Suggested reading:

- Software Requirements – Karl Wiegers. Good for consultants that need to enter an unknown area.
- Writing Effective Use Cases – Alistair Cockburn.
- User Stories Applied – Mike Cohen.
- UML Distilled – Martin Fowler.
- Refactoring – Martin Fowler.
- Design Patterns C++ – Gang of Four
- Head First: Design Patterns Java – O'Reilly.

Build software, make mistakes. Don't focus on learning at the expense of practice. The more you learn the more you realize there is to learn. You'll never feel fully prepared for a project.

Javascript

Programming - Basically telling a computer what to do and how to do it.

Source File - This is a document written in a programming language that tells the computer what you want it to do.

Programming Language - This is a language that (usually) resembles a mixture of English and math. It is both simple and strict enough for a compiler to understand.

Compiler - This translates a programming language that you can understand into a language the computer can understand, you can call it Computerese.

Library - A collection of useful code that has already been translated into Computerese that you can use in the programs you write.

.NET Platform - A large collection of tools, languages and libraries for writing programs with a heavy emphasis on productivity.

Sure, there's a lot more to it than that. You could tell them about IL and JIT compiling or garbage collection but these details aren't very relevant to a non-programmer.

****
********** JAVASCRIPT
****
****
For loop = When you know when to stop looping.
While loop = When you dont know when to stop looping. Used with true or false instead of numbers. Condition to be checked should be defined outside the loop. After reaching true, condition should be set to false to avoid infinite loop. increment is done inside the loop.

As we mentioned, for loops are great for doing the same task over and over when you know ahead of time how many times you'll have to repeat the loop. On the other hand, while loops are ideal when you have to loop, but you don't know ahead of time how many times you'll need to loop. However, you can combine a while loop with a counter variable to do the same kind of work a for loop does. In these cases, it's often a matter of preference.

```
// FOR loop

for (i=0; i<6; i++) {

	console.log(i);

}



// WHILE loop

var i = 0;

while (i<6) {

	console.log(i);

	i++;

}
```

```
// DO WHILE loop

var condition = false;

i = 99;

do {

        console.log(i)

} while (condition);
```

DO WHILE
Sometimes you want to make sure your loop runs at least one time no matter what. When this is the case, you want a modified while loop called a do/while loop.
This loop says: "Hey! Do this thing one time, then check the condition to see if we should keep looping."
After that, it's just like a normal while: the loop will continue so long as the condition being evaluated is true.

IF vs DO WHILE = IF runs once if something is true, DO WHILE runs continuously if something is true.

****
Let's go back to the analogy of computer languages being like regular spoken languages. In English, you have nouns (which you can think of as "things") and verbs (which you can think of as "actions"). Until now, our nouns (data, such as numbers, strings, or variables) and verbs (functions) have been separate.

No longer!

Using objects, we can put our information and the functions that use that information in the same place.

You can also think of objects as combinations of key-value pairs (like arrays), only their keys don't have to be numbers like 0, 1, or 2: they can be strings and variables.

An object is like an array in this way, except its keys can be variables and strings, not just numbers.

Objects are just collections of information (keys and values) between curly braces, like this:

```
var myObject = {
   key: value,
   key: value,
   key: value
};
```

Objects allow us to represent in code real world things and entities (such as a person or bank account). We do this by storing all relevant information in one place—an object.

How do we create an object? Like declaring a variable, or defining a function, we use var, followed by the name of the object and an equals sign. Each object then:

starts with {
has information inside
ends with };

Each piece of information we include in an object is known as a property. Think of a property like a category label that belongs to some object. When creating an object, each property has a name, followed by : and then the value of that property. For example, if we want Bob's object to show he is 34, we'd type in age: 34.

age is the property, and 34 is the value of this property. When we have more than one property, they are separated by commas. The last property does not end with a comma.

There are two ways to create an object: using object literal notation (which is what you just did) and using the object constructor.

Literal notation is just creating an object with curly braces, like this:

```
var myObj = { // THIS IS ALSO CALLED AN ASSOCIATIVE ARRAY
    type: 'fancy',
    disposition: 'sunny'
};

var emptyObj = {};
```

When you use the constructor, the syntax looks like this:

```
var myObj = new Object();
```

This tells JavaScript: "I want you to make me a new thing, and I want that thing to be an Object.

You can add keys to your object after you've created it in two ways:

```
myObj["name"] = "Charlie";
myObj.name = "Charlie";
```

Both are correct, and the second is shorthand for the first. See how this is sort of similar to arrays?

http://stackoverflow.com/questions/4859800/should-i-be-using-object-literals-or-constructor-functions
OBJECTS: LITERAL NOTATION vs CONSTRUCTOR?
- Literal = Single instance
- Constructor = Multiple instances

The method we've used to create objects uses object literal notation—that is, creating a new object with { } and defining properties within the brackets.

Another way of creating objects without using the curly brackets { } is to use the keyword new. This is known as creating an object using a constructor.

The new keyword creates an empty object when followed by Object(). The general syntax is:

```
var objectName = new Object();
```

In the last section, we discussed properties. We can think of properties as variables associated with an object. Similarly, a method is just like a function associated with an object.

Methods serve several important purposes when it comes to objects.

They can be used to change object property values. The method setAge on line 4 allows us to update bob.age.
They can be used to make calculations based on object properties. Functions can only use parameters as an input, but methods can make calculations with object properties.

METHODS ARE FUNCTIONS ASSOCIATED WITH A PARTICULAR OBJECT.

```
// here is bob again, with his usual properties
var bob = new Object();
bob.name = "Bob Smith";
bob.age = 30;
```

```
// this time we have added a method, setAge
bob.setAge = function (newAge){
  bob.age = newAge;
};
// here we set bob's age to 40
bob.setAge(40);
// bob's feeling old.  Use our method to set bob's age to 20
--------------------------
// here we define our method using "this", before we even introduce bob
var setAge = function (newAge) {
  this.age = newAge;
};
// now we make bob
var bob = new Object();
bob.age = 30;
// and down here we just use the method we already made
bob.setAge = setAge; // THIS SETS THE FUNCTION TO BECOME A METHOD FOR BOB, I.E. A
PROPERTY OF BOB.

// change bob's age to 50 here
bob.setAge(50);
-----------------
CUSTOM CONSTRUCTOR

function Person(name,age) {
  this.name = name;
  this.age = age;
  this.species = "Homo Sapiens";
}

var sally = new Person("Sally Bowles", 39);
var holden = new Person("Holden Caulfield", 16);

console.log("sally's species is " + sally.species + " and she is " + sally.age);
console.log("holden's species is " + holden.species + " and he is " + holden.age);

----------------------------
ANOTHER CUSTOM CONSTRUCTOR WITH METHODS

function Rectangle(height, width) {
  this.height = height;
  this.width = width;

  this.calcArea = function() {
      return this.height * this.width;
  };
  // put our perimeter function here!
  this.calcPerimeter = function() {
      return 2*this.height + 2*this.width;
  };

}

var rex = new Rectangle(7,3);

var area = rex.calcArea();
var perimeter = rex.calcPerimeter();

console.log(area);
console.log(perimeter);
```

ANOTHER CUSTOM CONSTRUCTOR WITH METHODS

```
function Rabbit(adjective) {
    this.adjective = adjective;
    this.describeMyself = function() {
        console.log("I am a " + this.adjective + " rabbit");
    };
}

// now we can easily make all of our rabbits
var rabbit1 = new Rabbit("fluffy");
var rabbit2 = new Rabbit("happy");
var rabbit3 = new Rabbit("sleepy");

rabbit1.describeMyself();
rabbit2.describeMyself();
rabbit3.describeMyself();
```

ANOTHER CUSTOM CONSTRUCTOR WITH ARRAY

```
// Our person constructor
function Person (name, age) {
    this.name = name;
    this.age = age;
}

// Now we can make an array of people
var family = new Array();
family[0] = new Person("alice", 40);
family[1] = new Person("bob", 42);
family[2] = new Person("michelle", 8);
family[3] = new Person("timmy", 6);
// add the last family member, "timmy", who is 6 years old

console.log(family.length);
```

**** In addition to making arrays of Objects, we can use objects as parameters for functions as well.

```
// Our person constructor
function Person (name, age) {
    this.name = name;
    this.age = age;
}

// We can make a function which takes persons as arguments
// This one computes the difference in ages between two people
var ageDifference = function(person1, person2) {
    return person1.age - person2.age;
}

var alice = new Person("Alice", 30);
var billy = new Person("Billy", 25);

// get the difference in age between alice and billy using our function
var diff = ageDifference(alice, billy);
```

console.log(diff);

// Our person constructor
function Person (name, age) {
  this.name = name;
  this.age = age;
}

// We can make a function which takes persons as arguments
// This one computes the difference in ages between two people
var ageDifference = function(person1, person2) {
  return person1.age - person2.age;
};

// Make a new function, olderAge, to return the age of
// the older of two people
var olderAge = function(person1, person2) {
  if (person1.age > person2.age) return person1.age;
  else return person2.age;
}

// Let's bring back alice and billy to test our new function
var alice = new Person("Alice", 30);
var billy = new Person("Billy", 25);

console.log("The older person is " + olderAge(alice, billy));

@@

LITERAL NOTATION vs OBJECT CONSTRUCTOR vs CUSTOM CONSTRUCTOR COMPARISON

var moby_dick = { // LITERAL NOTATION DOES NOT BELONG TO BOOK CLASS
  pages: 300,
  author: "Herman Melville"
};

Book.moby_dick = { // LITERAL NOTATION DOES BELONGS TO BOOK CLASS
  pages: 300,
  author: "Herman Melville"
};

// 3 lines required to make harry_potter OBJECT CONSTRUCTOR DOES NOT BELONG TO BOOK CLASS
var harry_potter = new Object();
harry_potter.pages = 350;
harry_potter.author = "J.K. Rowling";

// A custom constructor for book CUSTOM CONSTRUCTOR
function Book (pages, author) {
  this.pages = pages;
  this.author = author;
}

// Use our new constructor to make the_hobbit in one line BELONGS TO BOOK CLASS
var the_hobbit = new Book(320, "J.R.R. Tolkien");

@@@

```
function Circle (radius) {
    this.radius = radius;
    this.area = function () {
        return Math.PI * this.radius * this.radius;

    };
    // define a perimeter method here
    this.perimeter = function () {
        return 2 * Math.PI * this.radius;
    }
};

var circle1 = new Circle(5);

console.log(circle1.area());
console.log(circle1.perimeter());
```

You can add objects directly to friends, like this:

```
var friends = {
    bill: {},
    steve: {}
};
```

Or with the bracket ([]) or dot(.) notation, like this:

```
friends[bill] = {};
friends.steve = {};
```

Or with Object constructors, like this:

```
var friends = new Object();

friends.bill = new Object();
friends.steve = new Object();
```

Notice that "bill" and "steve" are not capitalized!

```
var friends = new Object();

friends.bill = { // THIS BELONGS TO THE FRIENDS OBJECT
    firstName: "Bill",
    lastName: "Gates",
    number: "2222 3333"
};

var steve = new Object(); // THIS ONE DOES NOT
steve.firstName = "Steve";
steve.lastName = "Jobs";
```

```
steve.number = "4444 9999";
-----------------------------------------------

var friends = new Object();

friends.bill = {
    firstName: "Bill",
    lastName: "Gates",
    number: "2222 3333",
    address: ["Street", "1"]
};

friends.steve = {
    firstName: "Steve",
    lastName: "Jobs",
    number: "4444 9999",
    address: ["Avenue", "2"]
};

var list = function() {
    for (var friend in friends) {
        console.log(friend);
    }
}

var search = function(name) {
    for (var friend in friends) {
        if (friends[friend].firstName === name) {
            console.log(friends[friend]);
            return friends[friend];
        }
    }
}
-------------------------------------------------------------------------------
var bob = {
    firstName: "Bob",
    lastName: "Jones",
    phoneNumber: "(650) 777-7777",
    email: "bob.jones@example.com"
};

var mary = {
    firstName: "Mary",
    lastName: "Johnson",
    phoneNumber: "(650) 888-8888",
    email: "mary.johnson@example.com"
};

var contacts = [bob, mary];

function printPerson(person) {
    console.log(person.firstName + " " + person.lastName);
}

function list() {
            var contactsLength = contacts.length;
            for (var i = 0; i < contactsLength; i++) {
                        printPerson(contacts[i]);
            }
}
```

```
/*Create a search function
then call it passing "Jones"*/
var search = function(lastName) {
    for (i=0; i<contacts.length; i++) {
        if (contacts[i].lastName === lastName) {
            printPerson(contacts[i]);
        }
        else return(lastName + " was not found.");
    }
}

search("Jones");

var add = function(firstName, lastName, email, phoneNumber) {
    contacts[contacts.length] = {
    firstName: firstName,
    lastName: lastName,
    phoneNumber: phoneNumber,
    email: email
    }
};

add("John", "Smith", "john.smith@example.com", "222 333");


list(contacts);
```

****

Introducing Functions

Programming is similar to baking cakes. Seriously! Imagine you are trying to teach your friend Jane how to bake many different types of cakes.

Each cake takes in different ingredients (ie. inputs). But the 'bake' instructions are always the same. For example:

Pre-heat the oven at 300 degrees
Mix all the ingredients in a bowl
Put contents into oven for 30 mins
And the output will be a different cake each time.

It is tedious to have to repeat to Jane the same 'bake' instructions every time. What if we could just say 'bake' and Jane would know to execute those three steps? That is exactly what a function is!

****

```
var number = prompt("Enter a number:");


var isEven = function(number) {

if (number % 2 === 0) {

        return true;

}

        else {
```

```
        return false;
    }

};

isEven();

************************************************************

var userChoice = prompt("Do you choose rock, paper or scissors?");

var computerChoice = Math.random();

if (computerChoice < 0.34) {

        computerChoice = "rock";

}
else if(computerChoice <= 0.67) {

        computerChoice = "paper";

}
else {

        computerChoice = "scissors";

}
console.log("Computer: " + computerChoice);


var compare = function(choice1, choice2) {

        if (choice1 == choice2) {

                return("The result is a tie!");

        }

        else if (choice1 == "rock") {

                if (choice2 == "scissors") return("rock wins");

                else return("paper wins");

        }

        else if (choice1 == "paper") {

                if (choice2 == "scissors") return("scissors wins");

                else return("paper wins");

        }

        else if (choice1 == "scissors") {
```

```
            if (choice2 == "paper") return("scissors wins");

            else return("rock wins");

        }

}


console.log(compare(userChoice, computerChoice));

**********************************************************************************************

var card = ["2", "3", "4", "5", "6", "7", "8", "9", "T", "J", "Q", "K", "A"];
var suit = ["s", "c", "h", "d"];

var randomCard1 = Math.random();
if (randomCard1 < 1/13) randomCard1 = card[0];
else if (randomCard1 <= 2*1/13) randomCard1 = card[1];
else if (randomCard1 <= 3*1/13) randomCard1 = card[2];
else if (randomCard1 <= 4*1/13) randomCard1 = card[3];
else if (randomCard1 <= 5*1/13) randomCard1 = card[4];
else if (randomCard1 <= 6*1/13) randomCard1 = card[5];
else if (randomCard1 <= 7*1/13) randomCard1 = card[6];
else if (randomCard1 <= 8*1/13) randomCard1 = card[7];
else if (randomCard1 <= 9*1/13) randomCard1 = card[8];
else if (randomCard1 <= 10*1/13) randomCard1 = card[9];
else if (randomCard1 <= 11*1/13) randomCard1 = card[10];
else if (randomCard1 <= 12*1/13) randomCard1 = card[11];
else if (randomCard1 <= 1) randomCard1 = card[12];

var randomSuit1 = Math.random();
if (randomSuit1 < 1/4) randomSuit1 = suit[0];
else if (randomSuit1 <= 2*1/4) randomSuit1 = suit[1];
else if (randomSuit1 <= 3*1/4) randomSuit1 = suit[2];
else if (randomSuit1 <= 1) randomSuit1 = suit[3];

var randomCard2 = Math.random();
if (randomCard2 < 1/13) randomCard2 = card[0];
else if (randomCard2 <= 2*1/13) randomCard2 = card[1];
else if (randomCard2 <= 3*1/13) randomCard2 = card[2];
else if (randomCard2 <= 4*1/13) randomCard2 = card[3];
else if (randomCard2 <= 5*1/13) randomCard2 = card[4];
else if (randomCard2 <= 6*1/13) randomCard2 = card[5];
else if (randomCard2 <= 7*1/13) randomCard2 = card[6];
else if (randomCard2 <= 8*1/13) randomCard2 = card[7];
else if (randomCard2 <= 9*1/13) randomCard2 = card[8];
else if (randomCard2 <= 10*1/13) randomCard2 = card[9];
else if (randomCard2 <= 11*1/13) randomCard2 = card[10];
else if (randomCard2 <= 12*1/13) randomCard2 = card[11];
else if (randomCard2 <= 1) randomCard2 = card[12];

var randomSuit2 = Math.random();
if (randomSuit2 < 1/4) randomSuit2 = suit[0];
else if (randomSuit2 <= 2*1/4) randomSuit2 = suit[1];
else if (randomSuit2 <= 3*1/4) randomSuit2 = suit[2];
else if (randomSuit2 <= 1) randomSuit2 = suit[3];

var dealtHand = function(card1, suit1, card2, suit2) {
```

```
    if(card1!=card2 || (card1===card2 && suit1!=suit2))
        return(card1 + suit1 + card2 + suit2);
}

document.writeln(dealtHand(randomCard1, randomSuit1, randomCard2, randomSuit2));

**********************************************************************************************

var cities = ["Melbourne", "Amman", "Helsinki", "NYC", "Skopje"];


for (var i = 0; i < cities.length; i++) {

        console.log("I would like to visit " + cities[i]);

}
**********************************************************************************************
var count = 0;


var loop = function() {

        while(count<3) {

                console.log("I'm looping!");

                count++;
        }

};


loop();

**********************************************************************************************
How does this code work? Math.floor(Math.random() * 5 + 1);

1. First we use Math.random() to create a random number from 0 up to 1. For example, 0.5
2. Then we multiply by 5 to make the random number from 0 up to 5. For example, 0.5 * 5 = 2.5
3. Next we use Math.floor() to round down to a whole number. For example, Math.floor( 2.5 ) = 2
4. Finally we add 1 to change the range from between 0 and 4 to between 1 and 5 (up to and including 5)


**********************************************************************************************
for (i=1; i<21; i++) {
    if (i % 3 === 0 && i % 5 === 0) {
        console.log("FizzBuzz");
    }
    else if (i % 5 === 0) {
        console.log("Buzz");
    }
    else if (i % 3 === 0) {
        console.log("Fizz");
    }
    else console.log(i);
}

**********************************************************************************************
var answer = prompt("Do you like fishsticks?");
```

```
switch(answer) {
    case "yes":
        console.log("You are a gay fish!");
    break;
    case "no":
        console.log("Are you sure?");
    break;
    case "maybe":
        console.log("Yes you do!");
    break;
    default:
        console.log("Only a gay fish can't answer...");
}

*****************************************************************************************
var movie = prompt("Which movie?");

var getReview = function (movie) {
    switch(movie) {
        case "Toy Story 2":
        return("Great story. Mean prospector.");
        break;
        case "Finding Nemo":
        return("Cool animation, and funny turtles.");
        break;
        case "The Lion King":
        return("Great songs.");
        break;
        default: return("I don't know!");
    }
};

getReview();
```

jQuery

<script type='text/javascript' src='script.js'></script>

****
Next, we'll need to start up our jQuery magic using the $(document).ready(); syntax you've seen. It works like this:

- $() says, "hey, jQuery things are about to happen!"
- Putting document between the parentheses tells us that we're about to work our magic on the HTML document itself.
- .ready(); is a function, or basic action, in jQuery. It says "hey, I'm going to do stuff as soon as the HTML document is ready!"
- Whatever goes in .ready()'s parentheses is the jQuery event that occurs as soon as the HTML document is ready.

So,

$(document).ready(something);

says: "when the HTML document is ready, do something!" (We'll show you how to replace something with an action in the next exercise.)

Note that .ready(); ends with a semicolon. This tells jQuery that you're done giving it a command.

Functions are the basic unit of action in jQuery. The main entry point of most jQuery applications is a block of code that looks like this:

```
$(document).ready(function() {
    Do something
});
```
Let's go through it bit by bit.

- $(document) is a jQuery object. The $() is actually a function in disguise; it turns the document into a jQuery object.
- .ready() is a type of function; you can think of it as sort of a helper that runs the code inside its parentheses as soon as the HTML document is ready.
- function(){} is the action that .ready() will perform as soon as the HTML document is loaded. (In the above example, the Do something placeholder is where those actions would go.)

****
BE CAREFUL OF THE NESTING!!! inside .ready() a function is defined function(){}

```
function() {
    jQuery magic;
}

+

$(document).ready();

=

$(document).ready(function() {
    jQuery magic;
});
```

****
$p vs $('p')

$p = $ does nothing. The $ sign just says that in this variable $p there is a p i.e. paragraph HTML selector.
$('p') = $ here is actually a function which gets passed a 'p' parameter which is a paragraph selector.

****

```
$(document).ready(function(){
    $div = $("div");
    $div.fadeIn("slow");
});
```

****

```
$(document).ready((function){
    $("red","pink").fadeTo("fast",1);
});

// IS THE SAME AS

$(document).ready(
    (function){$("red","pink").fadeTo("fast",1);}
);
```

****

The this keyword refers to the jQuery object you're currently doing something with.

```
$(document).ready(function() {
    $('div').click(function() {
        $(this).fadeOut('slow');
    });
});
```

****

```
$(document).ready(function(){
    $(".pull-me").click(function(){
        $(".panel").slideToggle("slow");
    });
});

// IS THE SAME AS THIS

$(document).ready(
    function(){
        $(".pull-me").click(
            function(){
                $(".panel").slideToggle("slow");
            }
        );
    }
);
```

****

```
$(".info").append("<p>Stuff!</p>");

IS THE SAME AS

$('<p>Stuff!</p>').appendTo('.info');
```

****

```
$(document).ready(function(){
    // THIS IS ALSO VALID
```

```
    /*-$body = $("body");
    $paragraph = "<p>I am a paragraph</p>";
    $body.append($paragraph);*/

    $("body").append("<p>I am a paragraph</p>");
});
```

****

.empty() deletes an element's content and all its descendants. For instance, if you .empty() an 'ol', you'll also remove all its 'li's and their text.

.remove(), not only deletes an element's content, but deletes the element itself.

****

```
$(document).ready(function(){
    $("#text").addClass("highlighted");
});
```

****

jQuery includes a .toggleClass() function that does exactly this.
If the element it's called on has the class it receives as an input, .toggleClass() removes that class;
if the target element doesn't have that class, .toggleClass() adds it.

****

```
$("div").height("100px");
$("div").width("50px");
$("div").css("background-color","#008800");

$(document).ready(function(){
    $div = $("div");
    $div.height(200);
    $div.width(200);
    $div.css("border-radius", 10);
});
```

****

Finally, we can update the contents of our HTML elements—that is, the bit between the opening and closing tags—using the .html() and .val() functions. CONTENT

.html() can be used to set the contents of the first element match it finds. For instance,

```
$('div').html();
```

will get the HTML contents of the first div it finds, and

```
$('div').html("I love jQuery!");
```

will set the contents of the first div it finds to "I love jQuery!"

.val() is used to get the value of form elements. For example,

```
$('input:checkbox:checked').val();
```

would get the value of the first checked checkbox that jQuery finds.

****

You can set a variable equal to the contents of the input field using .val(), like so:

```
//Get the value from an input
var input = $('input[name=checkListItem]').val();
```

1. Our selector finds our specific input using a css selector on our checkListItem input
2. We call val() to get the value of the field

```
$(document).ready(function(){
    $button = $("#button");
```

```
    $button.click(function(){
        var toAdd = $("input[name=checkListItem]").val();
        $list = $(".list");
        $appendix = '<div class="item">' + toAdd + '</div>' // ' vs " is important here; also don't put it in () out of habit.
        $list.append($appendix);
    });
});
```

****

Remove What's Been Clicked

Great job! Finally, we want to be able to check items off our list.

You might think we could do this:

```
$('.item').click(function() {
    $(this).remove();
});
```

and that's not a bad idea. The problem is that it won't work—jQuery looks for all the .items when the DOM is loaded, so by the time your document is ready, it's already decided there are no .items to .remove(), and your code won't work.

CUSTOM HANDLER CREATOR

For this, we'll need a new event handler: .on(). You can think of .on() as a general handler that takes the event, its selector, and an action as inputs. The syntax looks like this:

```
$(document).on('event', 'selector', function() {
    Do something!
});
```

In this case, 'event' will be 'click', 'selector' will be '.item', and the thing we'll want to do is call .remove() on this.

****

```
$(document).ready(function(){
    $("#button").click(function(){
        var toAdd = $("input[name=checkListItem]").val();
        $(".list").append('<div class="item">' + toAdd + '</div>'); // ' vs " is important here; also don't put it in () out of habit.
    });
    $(document).on("click", ".item", function(){ // .on is a custom handler
        $(this).remove();    // notice the usage of this here.
    });
});
```

Javascript vs jQuery

Select:
document.getElementById("divId")
--
$("#divId");

Get value:
document.getElementById("divId").value;
--
$("#divId").val();

Change content:
document.getElementById("divId").innerHTML="content";
--
$("#divId").html("content");

Click:
```
document.getElementById("divId").onclick=function() {
	// Do something
}
--
$("#divId").click(function() {
	// Do something
});
```

AJAX (Not part of JS vs jQ): Needs <script type="text/javascript" src="http://code.jquery.com/jquery.min.js"></script>

```
$.get("test.php", function(data) {	// This gets the content of test.php and puts it in the variable called data.
	alert(data);			// get should not be used because there is no catch for lost connection. Ajax has it.
});
**********
$.ajax({				// This is the preferred "get" method
	url:"test.php"
}).done(function(data){
	//alert(data);
	$("#text").html(data);		// This is the same as .innerHTML for JS. This changes the content of <p> with the one from test.php
	//$$("#text").html(data);	// This appends the content instead of swapping it.
});
```

Data Structures

It is an intentional arrangement of data held in memory.

The more constraints are put, the faster and smaller the data structure will be. Flexibility introduces overhead.

Humans naturally think in data structures i.e. collections of information. Ex. Recipe, shopping list, telephone directory, dictionary…

The five requirements of any data structure

- How to **Access** (one item / all items)
- How to **Insert** (at end / at position)
- How to **Delete** (from end / from position)
- How to **Find** (if exists / what location)
- How to **Sort** (sort in place / created sorted version)

Simple Structures

A mathematical tuple is a grouped collection of elements.

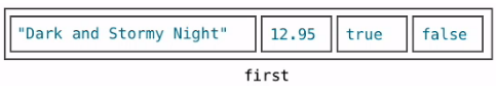
Struct = a very basic data structure.

Unorganized data vs a struct organization

```
// first book
string bookTitle = "Dark and Stormy Night";
double bookPrice = 12.95;
bool bookPublished = true;

// second book
string book2Title = "Stormy Returns";
double book2Price = 17.95;
bool book2Published = false;
bool book2isHardback = true;
```

vs

```
// define the struct
struct Book {
   string title;
   double price;
   bool isPublished;
   bool isHardback;
};

// create a variable with that struct type
Book first;
// set member variables
first.title = "Dark and Stormy Night";
first.price = 12.95;
first.isPublished = true;
first.isHardback = false;
```



first

Difference between structs and classes

struct	class
only data - no behavior	behavior and data
simple creation	explicit instantiation (new, alloc)
value types	reference types
no object-oriented features	polymorphism, inheritance, etc.
"Plain Old Data Structure" (PODS)	

Examples of PODS

```
struct Point {
    int x;
    int y;
};

Point startPosition;
startPosition.x = 50;
startPosition.y = 50;

Point finishPosition;
finishPosition.x = 500;
finishPosition.y = 100;

myObject.animate(startPosition,finishPosition);
```

```
struct Color {
    int red;
    int green;
    int blue;
    int alpha;
};

Color backgroundColor;
backgroundColor.red = 255;
backgroundColor.green = 0;
backgroundColor.blue = 0;
backgroundColor.alpha = 255;

myWindow.setBackground(backgroundColor);
```

Language support for structs

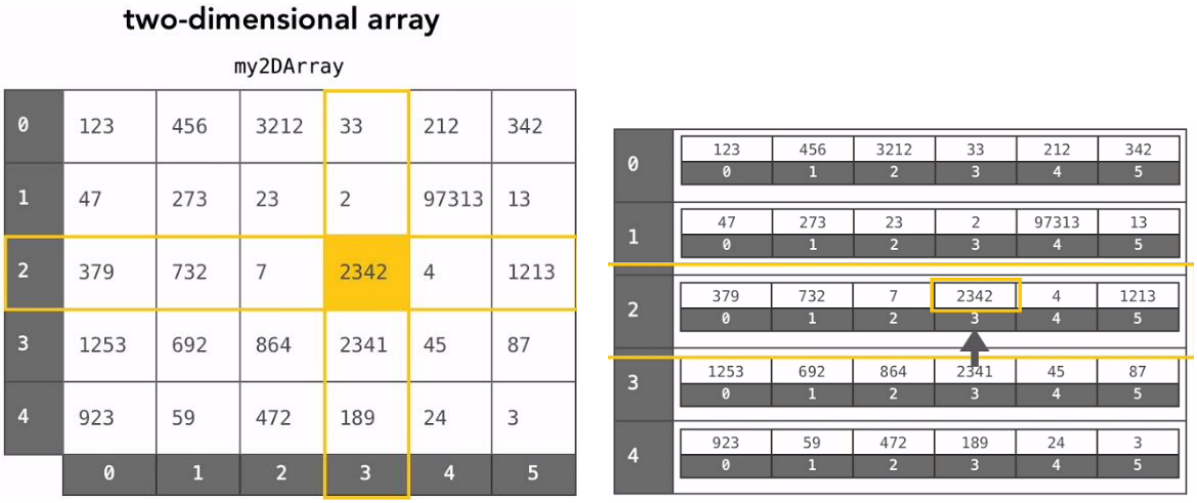
Objective-C	As in C, used in many Apple frameworks
C# / other .NET	Also allows basic behavior to be added
Java	Do not exist - closest equivalent is lightweight class
Python	Do not exist
Ruby	Exist, though implemented as lightweight class

Collections

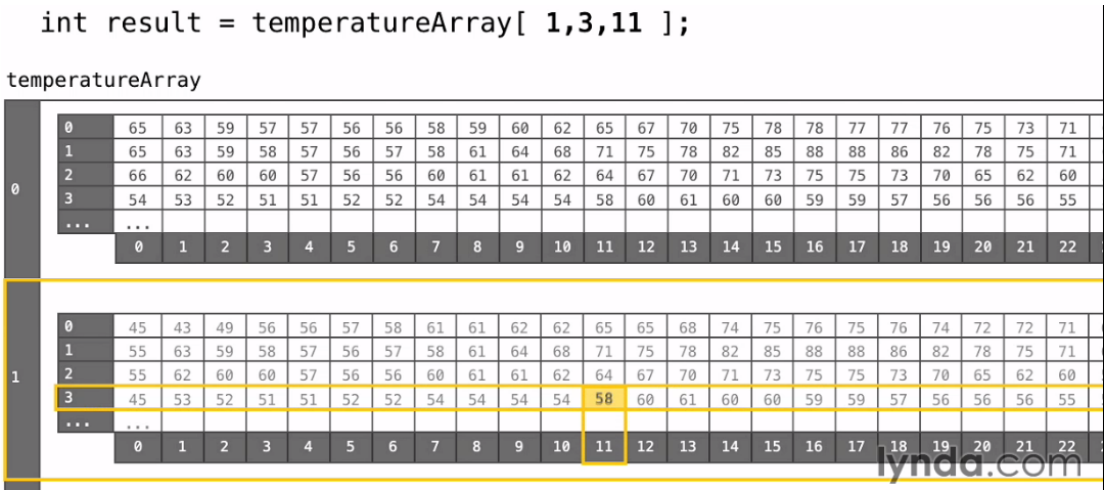
Arrays

The most common data structure. It as ordered collection of items.

Multi dimensional arrays are simply arrays of arrays.

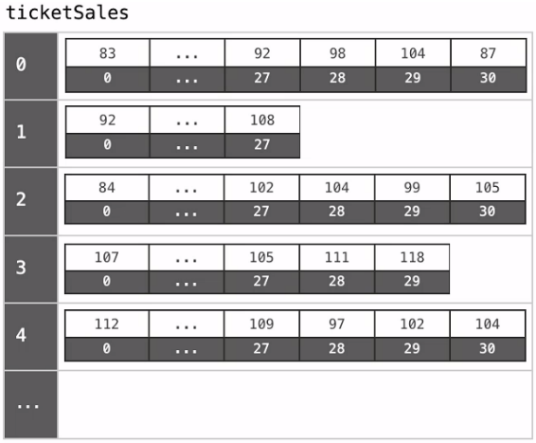


Three dimensional example: Track temperatures daily per hour per cities.



Jagged Arrays

Ex. Months have different number of days.
They are created by using logic as shown.




```
int[][] ticketSales = new int[12][]
for each month in ticketSales
    if april, june, september, november
       create array of 30 elements
    else if february and leap year
       create array of 29 elements
    else if february and not leap year
       create array of 28 elements
    else
       create array of 31 elements
    end if
    add array to ticketSales[month]
end for
```

Sorting

Always flinch at the thought of sorting. The operation is always resource intensive.

Make a difference between sorting and comparing.

Sorting is hard, comparing is easy.

```
myArray.sort() // need comparator / compare function
```

0	{id:XY100,lastname:Adams,firstname:Thomas,hiredate:4/11/2004,...}
1	{id:DE407,lastname:Smith,firstname:Grace,hiredate:2/28/2010,...}
2	{id:BC121,lastname:Smith,firstname:Sam,hiredate:12/1/1999,...}
3	{id:GH123,lastname:Thompson,firstname:Roy,hiredate:9/9/2011,...}
4	{id:MM004,lastname:Von Trapp,firstname:Florence,hiredate:7/1/2001,...}
5	{id:AB654,lastname:Zbigniew,firstname:Jane,hiredate:1/1/2001,...}

Comparator / Compare Function

```
PseudoCompare ( Employee a, Employee b)
    if a.lastname < b.lastname return -1 // less than
    if a.lastname > b.lastname return 1 // greater than
    if a.lastname == b.lastname
       if a.firstname < b.firstname return -1 // less than
       if a.firstname > b.firstname  return 1 // greater than
       if a.firstname == b.firstname return 0 // equal
    end if
end
```

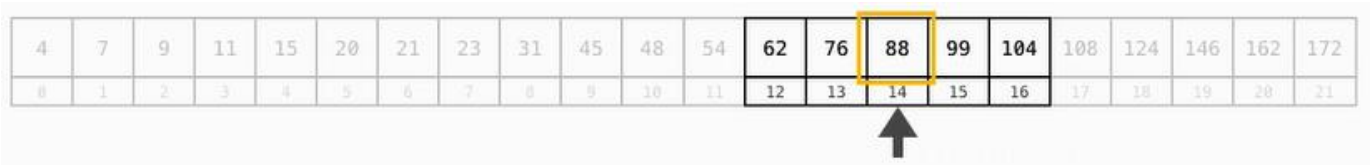
Searching: Linear vs Binary

Linear searching is also very resource intensive because it goes over every single member in an array. If the data structure is unordered, there might not be a faster solution than brute forcing it. If however, the data structure is ordered i.e. sorted, searching it becomes immensely more efficient.

```
// for simple yes/no result
if ( myArray.contains(99) ) {
    log("Yes, it exists.");
}
```

```
// for specific location
int result = myArray.indexOf(99);
if ( result != -1) {
   log("The value is located at position: " + result);
} else {
   log("The value was not found.");
}
```

Binary searching is an incredibly efficient search method (compared to the linear) and it has nothing to do with binary code i.e. 1 and 0. It simply means “in 2 pieces”. It works in a way where if the number wanted Is 99, the array length is determined and the search jumps to the midpoint. It then discards half of the array depending on where the number is. This division in half continues until the number is found or it finds out it doesn’t exist. The only problem with this is that **it only works on ordered data**.



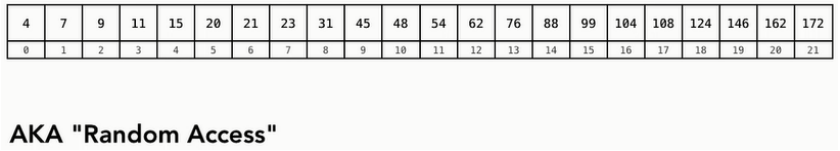
Lists

Are peculiar because they are all implemented differently in languages. A python list differs from a java list, while Objective-C and Ruby don’t have them.

Both arrays and lists are collections, used for grouping things under one name. But, they differ in the access i.e. arrays are direct access whereas lists are sequential. Access as in stored in memory.

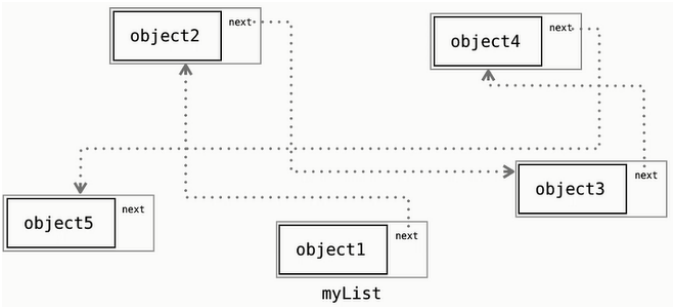
Arrays: Direct Access aka Random Access

The structure is stored in memory in one place meaning the objects are next to each other. The structure is based on a strict numeric index and all of the members are equally accessible regardless of the array size.



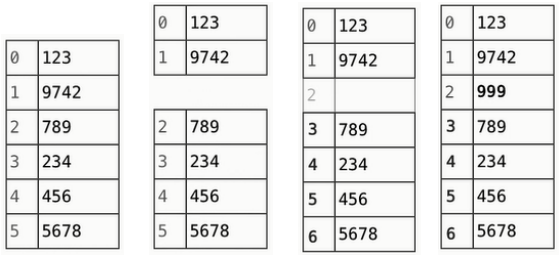
Lists: Sequential Access

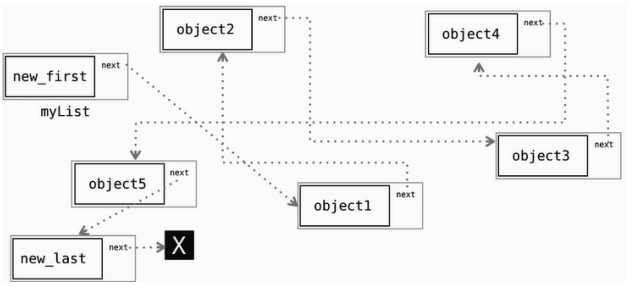
Unlike arrays, the objects are scattered in different memory locations. When accessing a list, the first node is given and each node contains the location of the next one. This is how every member is accessed via a sequence. It is not possible to jump to a specific object without accessing the first node and going through each one until the desired location is reached.



Why then use lists when arrays offer both direct and sequential access? The answer is adding and removing new elements.

Arrays require reallocation of the whole array in order to always keep a contigues area of memory. The bigger the array, the less efficient adding and removing new elements becomes.



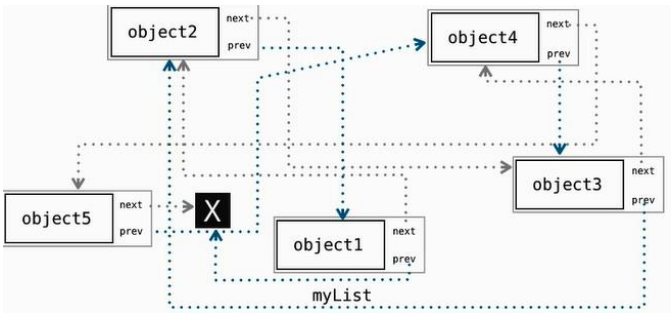


Lists (Singly Linked) on the other hand are as simple as adding a new node. Removal simply changes the next information in a node. These are fixed time operations and are size agnostic.

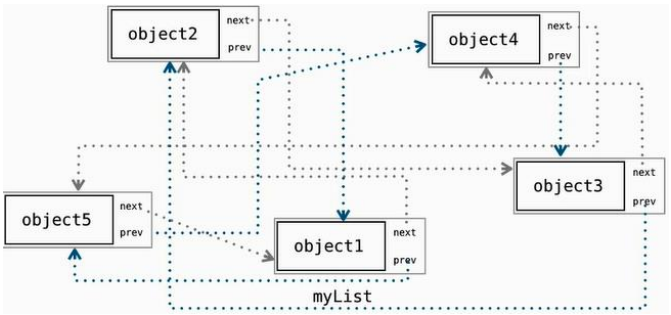
	Arrays	Linked Lists
Direct Access	**GOOD** fixed time O(1)	**POOR** linear time O(*n*)
Adding / Removing	**POOR** linear time O(*n*)	**GOOD** fixed time O(1)
Searching	O(*n*) linear search O(log *n*) binary search	O(*n*) - linear search

Doubly Linked Lists

Most of the lists used in programming are of this kind. The difference from singly liked lists is in the secondary information of **previous** in a node, as opposed to just **next**.



Null reference



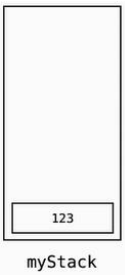
Circular

Linked List language support

Java	LinkedList in java.util
C#	LinkedList in System.Collections.Generic
Objective-C	n/a
Ruby	n/a
Python	n/a - "lists" are dynamic arrays, **not** linked lists
C++	std::list

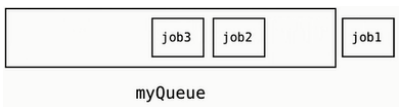
Stacks LIFO

```
myStack.push(444);
int topElement = myStack.pop();
        (444)
 int topValue = myStack.peek();
        (123)
```



Java	Stack (push / pop / peek)
C#	Stack (Push / Pop / Peek)
Python	use lists (append / pop)
Ruby	use Array (push / pop)
Objective-C	use NSMutableArray
C++	std::stack (push / pop)

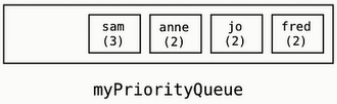
Queues FIFO



Java	LinkedList (add / remove)
C#	Queue (enqueue / dequeue)
Python	queue (put / get)
Ruby	use Array (push / shift)
Objective-C	NSMutableArray (addObject / removeObjectAtIndex:0)
C++	std::queue (push_back / pop_front)

Priority Queues

Typically requires a comparator or compare function



Java	PriorityQueue
C#	n/a
Python	n/a
Ruby	n/a
Objective-C	CFBinaryHeap
C++	std::priority_queue

Deques

Double-ended Queue

Caution! ***Deque*** vs ***Dequeue***

Java	LinkedList implements Deque
C#	n/a - use LinkedList for equivalent
Python	collections.deque
Ruby	n/a - use Array
Objective-C	n/a - use NSMutableArray
C++	std::deque

Hash-Based Data Structures

Associative Array

They cannot have duplicate keys.

Array

Associative Arrays

When sorted, the array indexes stay in place whereas the keys are paired with the values in an associative arrray i.e. they move together. The values can be objects:

AL	{name:Alabama,capital:Birmingham,pop:4833722,...}
AK	{name:Alaska,capital:Juneau,pop:735132,...}
AZ	{name:Arizona,capital:Phoenix,pop:6626624,...}

Associative Arrays - language support

Java	HashTable, HashMap, ConcurrentHashMap
Objective-C	NSDictionary, NSMutableDictionary
C#	Hashtable, StringDictionary, Dictionary<>, etc.
Ruby	Hash
Python	dict
C++	std::unordered_map

Hashing

Hashing is not encryption. Hashing functions are typically one-way i.e. not invertible and information is lost during the process. You can't turn ground beef into a cow. Ex:

Object 1 > Hash Function > 7146759310cegdc1
Object 2 > Hash Function > 542427436

Hashing Function - Example

```
Public Class Person {
    String firstname;                    Sam 19 1 13
    String lastname;                   Jones 10 15 14 5   (77)
    Date birthDate;              04/04/1990 04 04 1990  (1998)

    @Override                                   hash: 2075
    public int hashCode() {
       // code to add all numeric values
       // ...
       return hashvalue;
    }
}
```

Hashing rules

- Hashing should be deterministic under the same context
- Two objects that are ***equal*** should return the same hash
- But the same hash ***may*** also result from different objects

Hashing Collision

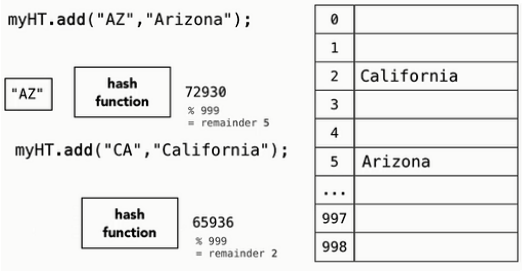
```
Sam          19 1 13
Jones        10 15 14 5   (77)
04/04/1990   04 04 1990   (1998)

hash: 2075

Fay          6 1 25
Adams        1 4 1 13 19  (70)
10/10/1985   10 10 1985   (2005)

hash: 2075
```

Why is it used? – The integer i.e. hash value can be used as a way to get to a certain location.

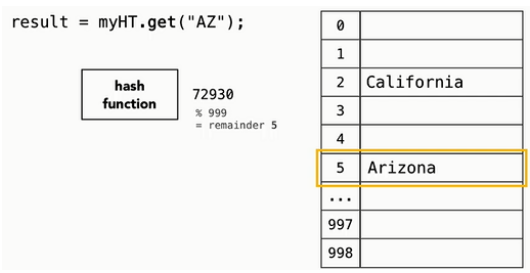
Hash Tables – Dictionaries

There is a huge speed advantage over arrays and linked lists in sorting, searching and adding or deleting new items.

Creating Hash Tables

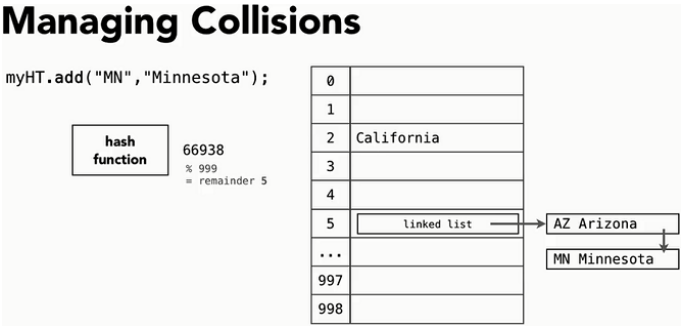


Searching



Searching is fast because instead of iterating through an array, the search value is hashed which gives the exact location of the item since it was put there in the first place based on the same hash.

Managing Collisions



This process of managing collisions is called Separate Chaining.

It works in a way where two hashes are the same, the second item is stored in the same location, but as a node, which meand that there is now a linked list (which is iterated) within the associative array i.e. hash table.

Default Hash Behavior

Java	`hashCode()`
Objective-C	`-hash`
C#	`GetHashCode()`
Ruby	`hash()`
Python	`hash()`
C++	`std::hash`

Hashing in Custom Classes

- Default equality behavior checks identity
- Can be overridden to check internal state
- If you redefine equality, redefine hashing

  If two objects are *equal* they must return the same hash
- This behavior is already provided for string objects

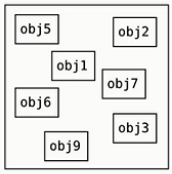
The default hash functionality will return an integer that is calculated from the id or the underlying memory address of that object. This means that you should get a different hash for every object you call this on.

Sets

Used for super fast lookup for the existence of objects in a collection. They function like hash tables but instead of storing items at a location based on a hashed key, the items are stored as hashes themselves. **They are only useful for checking membership, because no value is retrieved.**

- A set is an ***unordered*** collection of objects
- No index, sequence, or key
- No duplicates
- Fast lookup

```
mySet.contains(obj7); ✓
mySet.contains(obj5); ✓
mySet.contains(obj99); ✘
```



Set Implementation with Hash Table

```
if (mySet.contains(obj1)) {
```

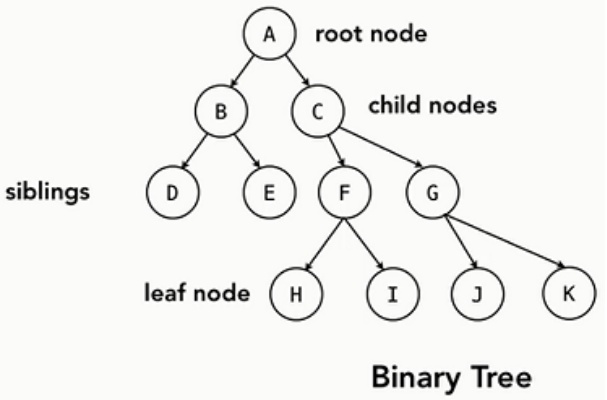
hash function → 3

0	obj2
1	
2	
3	obj1
4	
5	
...	

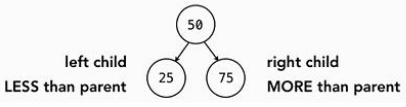
Java	`HashSet`
C#	`HashSet`
Python	`set / frozenset`
Ruby	`Set`
Objective-C	`NSSet, NSMutableSet`
C++	`std::set`

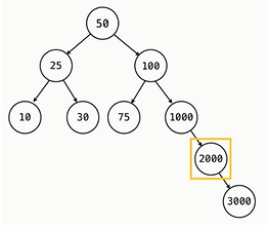
Trees

Binary Trees are allowed to have a maximum of 2 childs. Leaf nodes have no childs.



Binary Search Tree - Child Nodes





BST / Hash Table Comparison

Binary Search Tree	Hash Table
• Fast insertion, fast retrieval	• Fast insertion, fast retrieval
• Stays sorted - iterate elements in sequence	• Retrieval not in guaranteed order

Java	`TreeMap`
C#	`SortedDictionary`
Python	`n/a`
Ruby	`n/a`
Objective-C	`n/a`
C++	`std::set`

Heaps

Not to be confused with Heap memory allocation.

Items are added top to bottom, left to right. This is a balanced structure and no space is left unallocated. Items are swapped whenever needed to keep the balance.

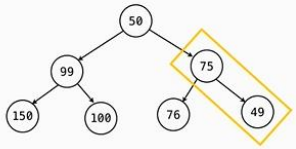
Min Heap or Max Heap?

Lowest (or highest) value at the top of the heap

- Min heap rule: a child must always be **greater than** its parent
- Max heap rule: a child must always be **less than** its parent

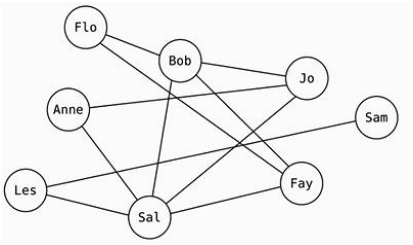
Java	`PriorityQueue`
C#	`n/a`
Python	`heapq`
Ruby	`n/a`
Objective-C	`CFBinaryHeap`
C++	`std::priority_queue`

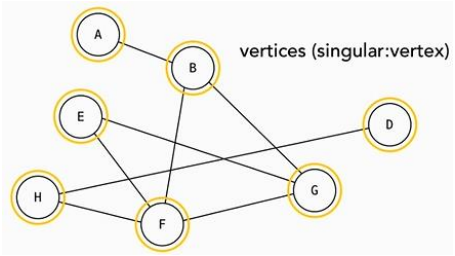
Min Heap: Example

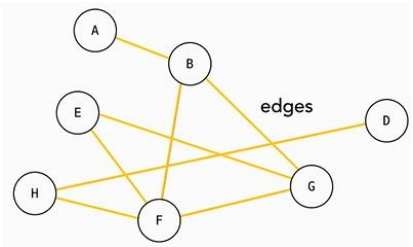


Graphs

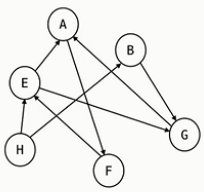
Useful for complex systems of interconnected points. Linked Lists, Trees and Heaps are all graph implementations. There is no support for generic graphs in any languages because they are too specific.



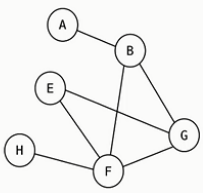




Directed / Undirected Graphs

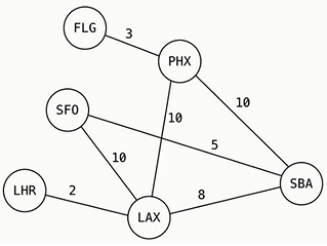


directed



undirected

Weighted Graphs



Summary

It's not about the data structure, it's about the data itself.
How much data is there? How much does it need to change? Does it need to be sorted? Does it need to be searched? These questions should determine the data structure.

Which data structure is the fastest? Fast to access? Fast to sort? Fast to search?

Arrays

- **Strengths**
  - Direct indexing
  - Easy to create and use
- **Weaknesses**
  - Sorting and searching
  - Inserting and deleting - particularly if not at start / end

Linked Lists

- **Strengths**
  - Inserting and deleting elements
  - Iterating through the collection
- **Weaknesses**
  - Direct access
  - Searching and sorting

Stacks and Queues

- **Strengths**
  - Designed for LIFO / FIFO
- **Should not be used for**
  - Direct access
  - Searching and sorting

Hash Tables

- **Strengths**
  - Speed of insertion and deletion
  - Speed of access
- **Weaknesses**
  - Some overhead
  - Retrieving in a sorted order
  - Searching for a specific value

Sets

- **Strengths**
  - Checking if an object is in a collection
  - Avoiding duplicates
- **Do not use for**
  - Direct access

Binary Search Trees

- **Strengths**
  - Speed of insertion and deletion
  - Speed of access
  - Maintaining sorted order
- **Weaknesses**
  - Some overhead

Fixed Structures are Faster / Smaller

Choose a fixed (immutable) version where possible

- If you need an immutable version to load, consider then copying to a mutable version for lookup

Foundations of Programming: Databases

Many business use spreadsheets in the beginning. Having data is not a good enough reason to have a database. Having data is not the problem. The problem is what comes next:

- Size. What happens when there are 2 million records?
- Ease of updating. Can 20 people work on the same file?
- Accuracy. What prevents me from putting invalid data in a spreadsheet?
- Security. Who can view? Who can edit? Who made changes?
- Redundancy. Duplicate values. Same product selling for different prices. Which spreadsheet is the valid one?
- Importance. A crash of a spreadsheet can lose you data. Can be corrupted etc.

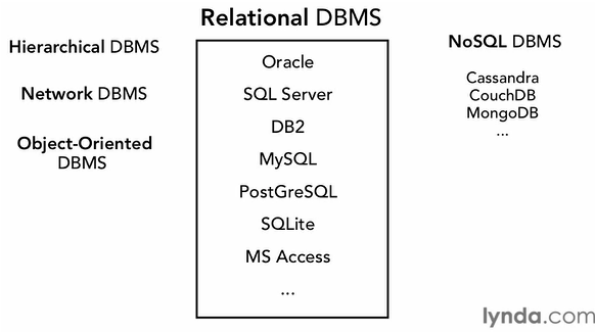
It's not about where to put the data, but rather to manage it effectively.

Database vs DBMS

- Oracle
- SQL Server
- MySQL
- PostgreSQL
- MongoDB

These are not databases. These are **DataBase Management Systems**. A DBMS is used to manage one or more databases. Many companies use multiple databases across different DBMSs.

Relational DBMSs are by far the most used ones, but there are some others.



Database Fundamentals

Tables

Databases are in fact collections of tables. Without them, a database would be an empty useless shell.

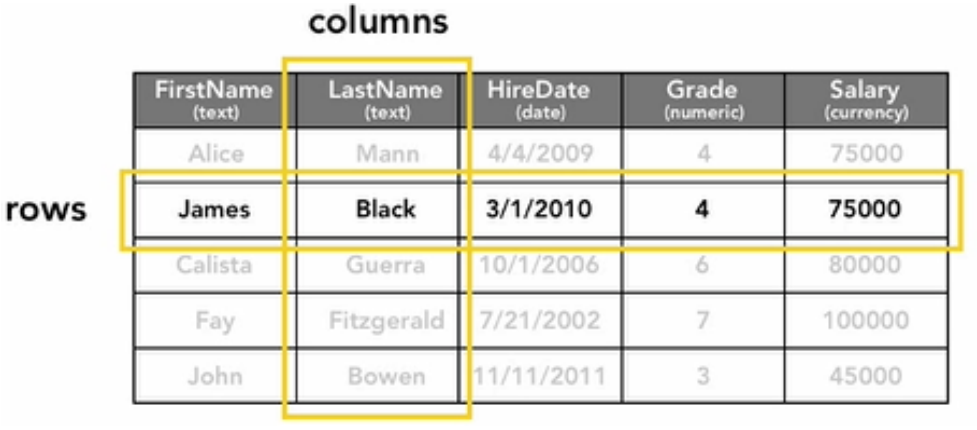
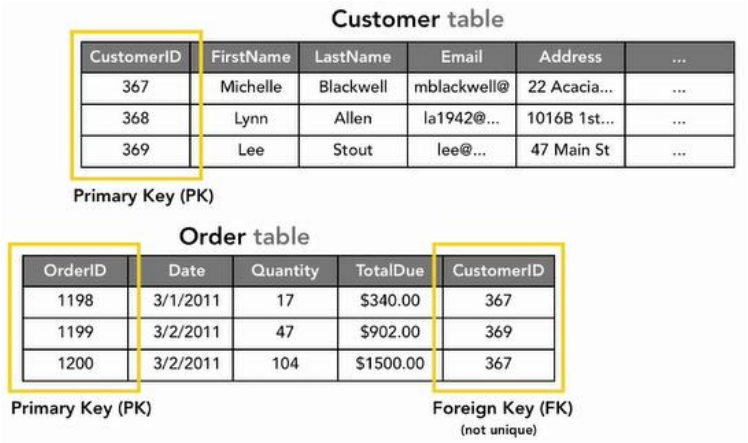
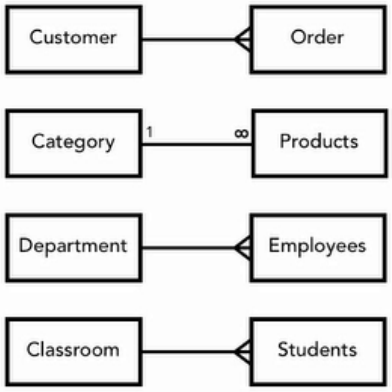


Table Relationships

One to many

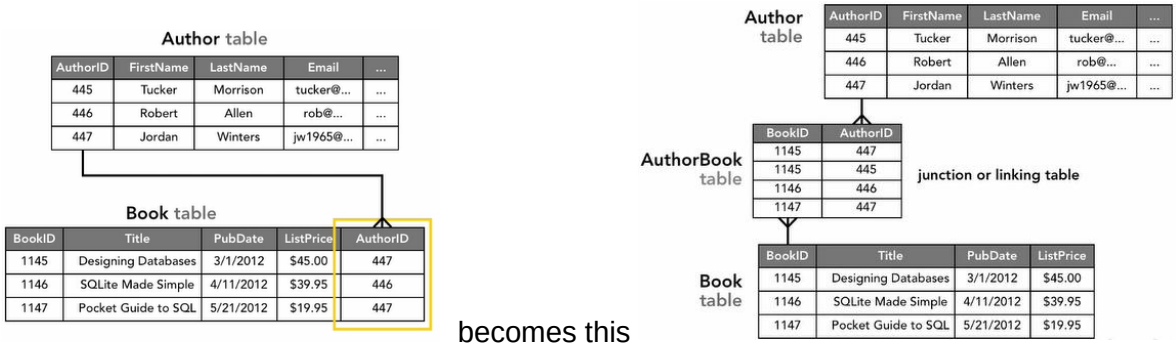




Many to many

- Not so obvious.
- Cannot be done directly.

Adding duplicate columns (AuthorID2) and multiple foreign keys (445, 446) is discouraged.



One to many

many to many

One to one relationships are very uncommon. If one row points to only one row, you might as well join the tables.

Transactions

Either both debit and credit happen at the same time or the transaction doesn't occur i.e. is reversed..

A transaction needs to be **ACID**:

- Atomic: It must happen completely or not at all, regardless of reason and if it's 2 steps or 20 steps.
- Consistent: The transaction must result in the rules set by the DBMS.
- Isolated: Data is locked during transactions, making it impossible to change.
- Durable: Changes made are permanent and reliable.

Introduction to SQL

It is a declarative query language, not a procedural, imperative language. In procedural languages you describe the steps whereas in a query language you describe the outcome.

Table

BookID	Title	PubDate	ListPrice
1145	Designing Databases	3/1/2012	$45.00
1146	SQLite Made Simple	4/11/2012	$39.95
1147	Pocket Guide to SQL	5/21/2012	$19.95
1148	DB Best Practices	5/22/2012	$48.00
1149	NoSQL Databases	5/26/2012	$35.00
1150	Fun with SQL	5/27/2012	$42.00
...	...	...	...

Procedural

```
for each b in Books
  if price > 40
    add b to expensive_books_array
  else
    ignore
  end
end

return expensive_books_array
```

Query

```
SELECT * FROM Books WHERE ListPrice > 40
```

BookID	Title	PubDate	ListPrice
1145	Designing Databases	3/1/2012	$45.00
1148	DB Best Practices	5/22/2012	$48.00
1150	Fun with SQL	5/27/2012	$42.00

SQL is used to CRUD i.e. Create, Read, Update and Delete.

Tables

Database Planning

Planning is crucial before doing anything. It is a very bad idea to begin with building immediately.

Building a database is like getting tattooed. You really want to get it right on the first try, because changes are painful.

Building a database is all about following a step by step methodology that was battle tested over 40 years. It is a very bad idea to get creative in this part.

Before doing anything, you need to ask these two question:

what's the point?

"It's a database to store product and order information."

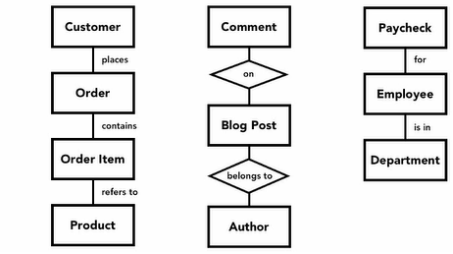
or...

"We help customers find books,
discover what others thought about them,
purchase and track their orders,
contribute their own reviews and opinions,
and learn about other products they might like
based on people with similar reading habits."

what do you have already?

- physical items
  - forms, order sheets, printouts, handouts
- people and expertise
- an existing "database"
  - spreadsheets, text files

entities



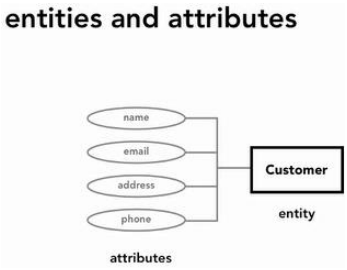
Separate tables are created for each object i.e. entity.

Tables should be named in singular form.

Although there is a crossover in design from OOP, the database is only concerned about saving the data and the relationship between the entities. Not the actual behavior.

The stage of early database design is called **Entity Relationship Modelling**. (Boxes and lines)

Identifying Columns and Data Types



Employee table

FirstName	character	not NULL	
LastName	character	not NULL	
DateHired	date	not NULL	default: today
SalaryGrade	integer	not NULL	
AddressLine1	character	not NULL	
AddressLine2	character	NULL	
City	character	not NULL	
State	character(2)	not NULL	
Zip	character	not NULL	
Email	character	not NULL	pattern match:email
Photo	binary	NULL	
(etc.)			

Columns can be named in many styles (FirstName, first_name, firstname, fName, strFirstName, firstName) Use: FirstName i.e. Uppercase each word.

Each column must have a defined data type and it should be known if it's required or not/ Not NULL means the column must contain data, whereas NULL means it could be left empty.

Flexibility is our friend in programming, but not in databases. Everything should be defined as precisely as possible in order to enforce rules which will keep the database relevant and clean of garbage.

Primary Keys

A value that uniquely identifies an individual row. They are specified by integers called IDs which are auto-incremented. Sometimes they occur naturally in a table, and they are called "natural" keys.

integer, auto-increment

Customer

CustomerID	FirstName	LastName	Email	Address	...
1	Michelle	Blackwell	mblackwell@	22 Acacia...	...
2	Lynn	Allen	la1942@...	1016B 1st...	...

Primary Key (PK)

Book

ISBN	Title	ReleaseDate	ListPrice	...
1596717521	JavaScript Essential Training	6/1/2011	$149.95	
321158814	Building Rich Internet Apps	3/2/2003	$39.95	
765359146	Red	11/1/2008	$7.95	

Primary Key (PK)
(natural key)

Composite Key

Unique rows are identified by two keys, whereas each cannot do it on its own. Sure, primary keys can be manufactured via the auto-increment ID, but composite keys are still used.

Yearbook

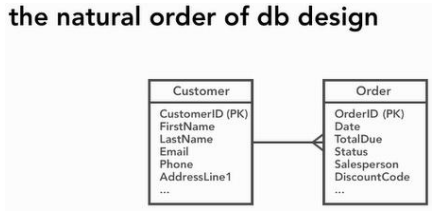
School	Year	ListPrice	PageCount	UnitsInStock	...
Orchard High	2010	$29.95	144	32	
Orchard High	2011	$34.95	132	14	
Lawstone Elementary	2010	$29.95	161	0	
Lawstone Elementary	2011	$29.95	155	38	
Lawstone Elementary	2012	$34.95	172	144	
...					

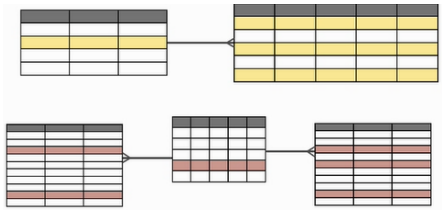
Primary Key (PK)
(composite key)

Schools and years are not unique by themselves, but when combined, given that a yearbook is published only once per year, they pose as a unique value. Orchard High 2010.

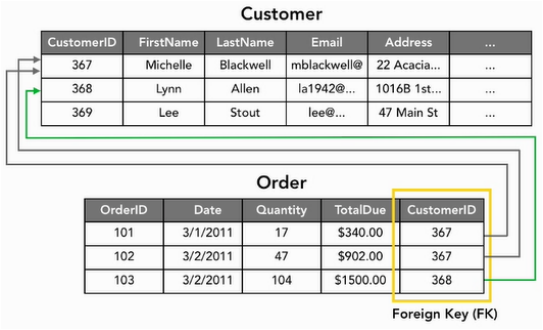
Relationships

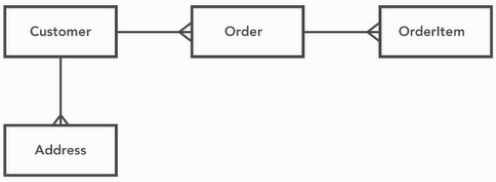
Look both ways (From each perspective) when trying to determine the relationship.

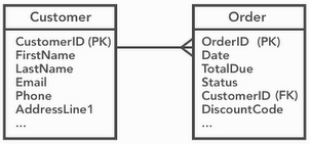




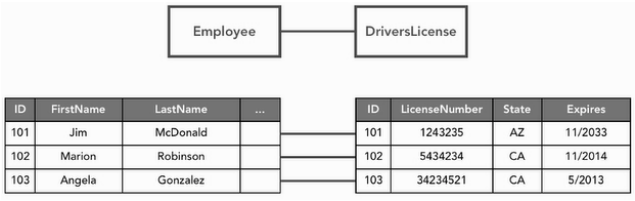
One-To-Many





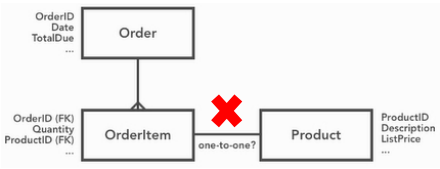


One-To-One

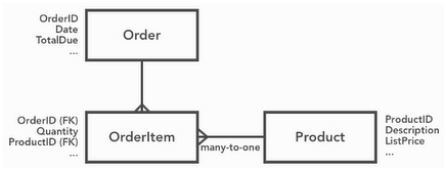


might as well

Sometimes you think you have a one-to-one relationship, but you don't. Look both ways (from each perspective) to avoid this.

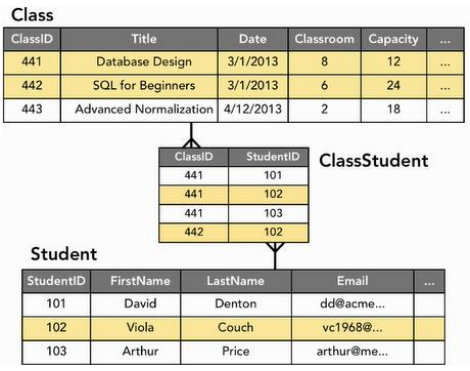


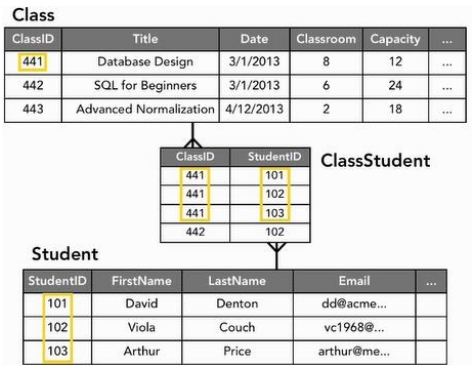
is in fact



Many-To-Many

Can't be represented directly.





Optimization

Normalization

Taking your database design through these 3 steps will vastly improve the quality of your data.

There are more than 3 Normal Forms, but usually 3 is the norm. There are more than 6.

First Normal Form (1NF)

Each of the columns and tables should contain one and only one value without it repeating.

Employee

EmployeeID	FirstName	LastName	Email	...	ComputerSerial
551	Les	Adams	ladams@	...	XP5435512
552	Jill	Baker	jbaker@	...	WA2324451
553	Stephen	Jackson	s.jackson@	...	BC32345412

What happens when an employee needs 2 computers?

Employee

EmployeeID	FirstName	LastName	Email	...	ComputerSerial
551	Les	Adams	ladams@	...	XP5435512 , XA5543231
552	Jill	Baker	jbaker@	...	WA2324451
553	Stephen	Jackson	s.jackson@	...	BC32345412 , ZZ87656, XX21312

Never do this!

Employee

EmployeeID	FirstName	LastName	Email	...	ComputerSerial	ComputerSerial2	ComputerSerial3
551	Les	Adams	ladams@	...	XP5435512	XA5543231	
552	Jill	Baker	jbaker@	...	WA2324451		
553	Stephen	Jackson	s.jackson@	...	BC32345412	ZZ87656	XX21312

This is better but still bad!

Employee

EmployeeID	FirstName	LastName	Email	...
551	Les	Adams	ladams@	...
552	Jill	Baker	jbaker@	...
553	Stephen	Jackson	s.jackson@	...

Computer

Serial	EmployeeID	Description	...
XP5435512	551	Dell Laptop	...
XA5543231	551	Apple MacBook	...
WA2324451	552	Acer Desktop	...
BC32345412	553	MacBook Pro	...
ZZ87656	553	HP Server	...
XX21312	553	iPad 3	...

Best solution.

Usually every 1NF problem is solved by creating a new table. One of the signs for the need is when columns start having the same name with a number differentiating them. Computer1, Computer2…

Second Normal Form (2NF)

Any non-key field should be dependent on the entire primary key i.e. **"Can I figure out any of the values in the row from just part of the composite key?". Only a problem when dealing with composite keys.**

Events

Course	Date	CourseTitle	Room	Capacity	Available	...
SQL101	3/1/2013	SQL Fundamentals	4A	12	4	
DB202	3/1/2013	Database Design	7B	14	7	
SQL101	4/14/2013	SQL Fundamentals	7B	14	10	
SQL101	5/28/2013	SQL Fundamentals	12A	8	8	
CS200	4/15/2012	C Programming	4A	12	11	

composite primary key

What happens if someone changes the Course ID, but not the title?

Events

Course	Date	Room	Capacity	Available	...
SQL101	3/1/2013	4A	12	4	
DB202	3/1/2013	7B	14	7	
SQL101	4/14/2013	7B	14	10	
SQL101	5/28/2013	12A	8	8	
CS200	4/15/2012	4A	12	11	

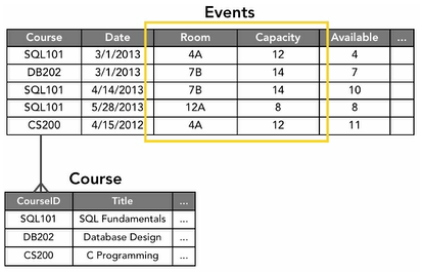
Course

CourseID	Title	...
SQL101	SQL Fundamentals	...
DB202	Database Design	...
CS200	C Programming	...

Solution!

Third Normal Form (3NF)

No non-key field is dependent on any other non-key field i.e. **“Can I figure out any of the values in this row from any of the other values?”.**.

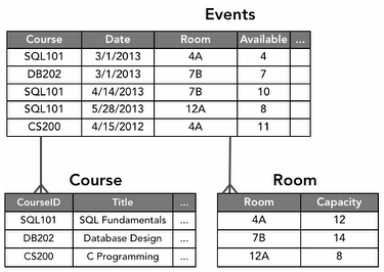


Events

Course	Date	Room	Capacity	Available	...
SQL101	3/1/2013	4A	12	4	
DB202	3/1/2013	7B	14	7	
SQL101	4/14/2013	7B	14	10	
SQL101	5/28/2013	12A	8	8	
CS200	4/15/2012	4A	12	11	

Course

CourseID	Title	...
SQL101	SQL Fundamentals	...
DB202	Database Design	...
CS200	C Programming	...



Events

Course	Date	Room	Available	...
SQL101	3/1/2013	4A	4	
DB202	3/1/2013	7B	7	
SQL101	4/14/2013	7B	10	
SQL101	5/28/2013	12A	8	
CS200	4/15/2012	4A	11	

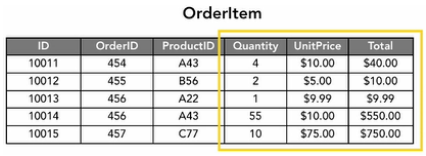
Course

CourseID	Title	...
SQL101	SQL Fundamentals	...
DB202	Database Design	...
CS200	C Programming	...

Room

Room	Capacity
4A	12
7B	14
12A	8

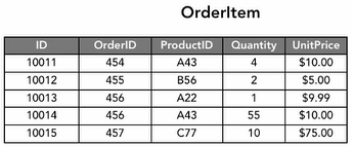
Same seats in a room.



OrderItem

ID	OrderID	ProductID	Quantity	UnitPrice	Total
10011	454	A43	4	$10.00	$40.00
10012	455	B56	2	$5.00	$10.00
10013	456	A22	1	$9.99	$9.99
10014	456	A43	55	$10.00	$550.00
10015	457	C77	10	$75.00	$750.00

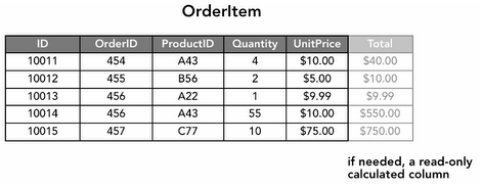
The total depends on the other values.



OrderItem

ID	OrderID	ProductID	Quantity	UnitPrice
10011	454	A43	4	$10.00
10012	455	B56	2	$5.00
10013	456	A22	1	$9.99
10014	456	A43	55	$10.00
10015	457	C77	10	$75.00

Remove the column



OrderItem

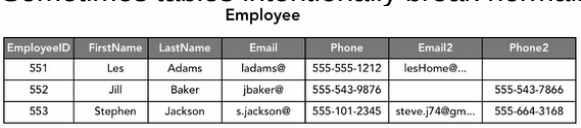
ID	OrderID	ProductID	Quantity	UnitPrice	Total
10011	454	A43	4	$10.00	$40.00
10012	455	B56	2	$5.00	$10.00
10013	456	A22	1	$9.99	$9.99
10014	456	A43	55	$10.00	$550.00
10015	457	C77	10	$75.00	$750.00

if needed, a read-only calculated column

Or make it a calculated read only column

Denormalization

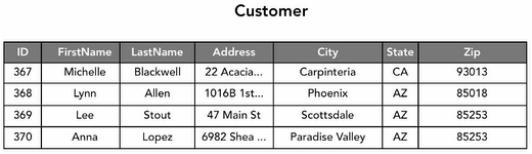
Sometimes tables intentionally break normalization, and some only seem like they do.



Employee

EmployeeID	FirstName	LastName	Email	Phone	Email2	Phone2
551	Les	Adams	ladams@	555-555-1212	lesHome@...	
552	Jill	Baker	jbaker@	555-543-9876		555-543-7866
553	Stephen	Jackson	s.jackson@	555-101-2345	steve.j74@gm...	555-664-3168

Creating a new table for phones and emails would complicate things needlessly.



Customer

ID	FirstName	LastName	Address	City	State	Zip
367	Michelle	Blackwell	22 Acacia...	Carpinteria	CA	93013
368	Lynn	Allen	1016B 1st...	Phoenix	AZ	85018
369	Lee	Stout	47 Main St	Scottsdale	AZ	85253
370	Anna	Lopez	6982 Shea ...	Paradise Valley	AZ	85253

The same goes for the area codes.

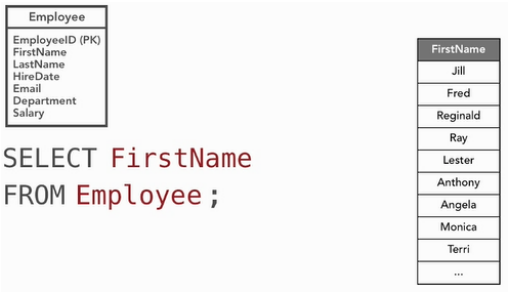
Querying

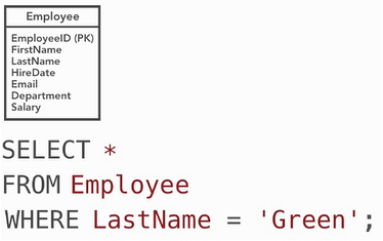
SELECT

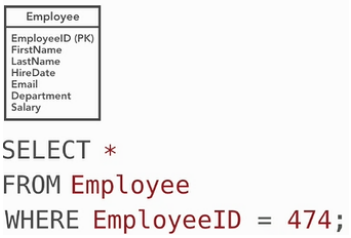
These SQL keywords handle 90% of needs. SQL is **NOT** case sensitive as well as whitespace. It is the convention for keywords to be UPPERCASE. It is not necessary for statements to end in (;), but it does provide extra clarity.

SELECT	GROUP BY	DELETE
FROM	JOIN	HAVING
WHERE	INSERT INTO	IN
ORDER BY	UPDATE	

SELECT columns **FROM** table **WHERE** condition





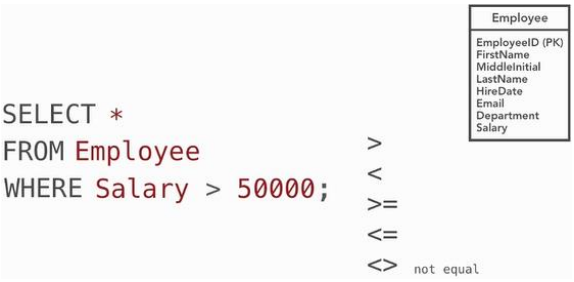


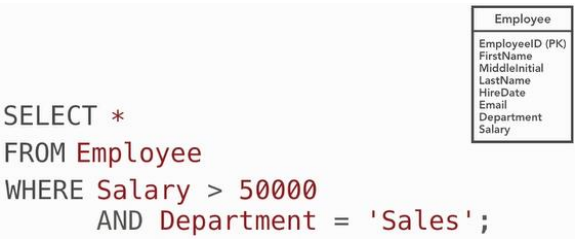
If the query needs to be of another database, without writing it in the scope:

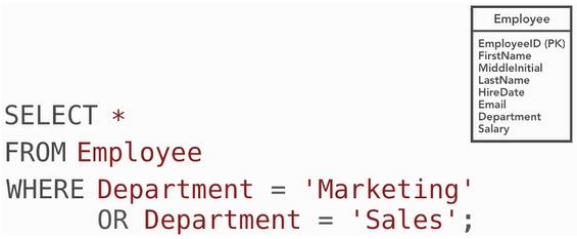
```
SELECT *
FROM HumanResources.Employee
WHERE Salary > 50000;
```

WHERE

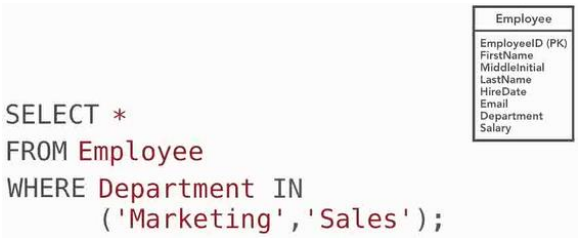
This is akin to an if statement in programming, where the result is a Boolean. String values must be in ‘single quotes’.

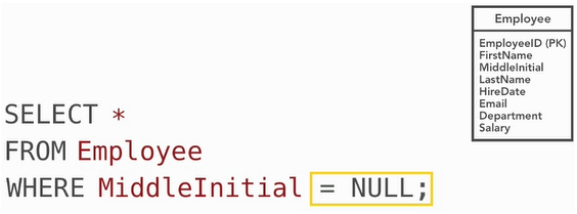




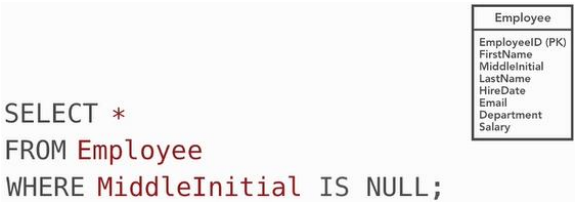


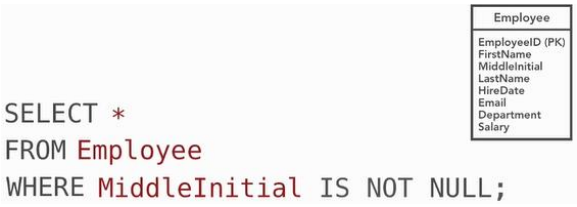
the same as





should be





Sorting

Ordering is **ascending** by **default** i.e. small to large.

```
SELECT Description,
       ListPrice, Color
FROM Product ;
```

Product
ProductID (PK)
Description
ListPrice
Color
Weight
Category
SKU
Manufacturer

Description	ListPrice	Color
Extender Cables	4.49	Black
Battery Charger	35.00	Black
Seat Cover	7.98	Red
Headphone Amp	420.00	Silver
...	...	...

```
SELECT Description,
       ListPrice, Color
FROM Product
ORDER BY ListPrice DESC ;
```

Product
ProductID (PK)
Description
ListPrice
Color
Weight
Category
SKU
Manufacturer

Description	ListPrice	Color
Premier Headphone Amp	699.00	Gold
Headphone Amp	420.00	Silver
Compressor Unit	399.00	Black
Adjustable LED Lamp	349.98	White
...	...	...

```
SELECT *
FROM Employee
WHERE Salary > 50000
ORDER BY LastName, FirstName;
```

Employee
EmployeeID (PK)
FirstName
LastName
HireDate
Email
Department
Salary
...

EmployeeID	FirstName	LastName	HireDate	...
489	Matilda	Aaron	1/14/2001	
24	Maria	Adams	9/24/1992	
551	Siobhan	Adams	6/23/2001	
439	Stephen	Adams	9/19/2000	
1008	Gertrude	Bailey	4/14/2008	
...				

Aggregate Functions

These do calculations on a set of data but return a single value.

```
SELECT *
FROM Employee
```

Employee
EmployeeID (PK)
FirstName
LastName
HireDate
Email
Department
Salary
...

EmployeeID	FirstName	LastName	...
2	Aaron	Cooper	...
4	Lou	Donoghue	
5	Alice	Bailey	
6	Oswald	Hall	
7	John	Velasquez	
...			

```
SELECT COUNT(*)
FROM Employee
```

result
547

Employee
EmployeeID (PK)
FirstName
LastName
HireDate
Email
Department
Salary
...

```
SELECT *
FROM Product
ORDER BY ListPrice DESC ;
```

Description	ListPrice	Color	...
Premier Headphone Amp	699.00	Gold	
Headphone Amp	420.00	Silver	
Compressor Unit	399.00	Black	
Adjustable LED Lamp	349.98	White	
...	...	...	

```
SELECT MAX(ListPrice)
FROM Product ;
```

result
699.00

```
SELECT AVG(ListPrice)
FROM Product ;
```

```
SELECT SUM(TotalDue)
FROM Order
WHERE CustomerID = 854;
```

result
2742.75

```
SELECT COUNT(*)
FROM Product
WHERE Color = 'Red'
```

result
276

```
SELECT COUNT(*) , Color
FROM Product
GROUP BY Color
```

result	Color
47	Black
32	Silver
8	White
86	Clear
...	...

Joining Tables

Employee

ID	FirstName	LastName	HireDate	DepartmentID	...
734	Aaron	Cooper	4/17/09	2	
735	Lou	Donoghue	5/22/05	4	
736	Alice	Bailey	9/1/99	(null)	
737	Oswald	Hall	3/19/11	5	
738	John	Velasquez	4/5/10	4	
...	...				

Department

DepartmentID	Name	Location	BudgetCode	...
1	Production	CA	A4	...
2	R&D	AZ	B17	
3	Marketing	CA	A7	
4	Sales	CA	A7	
5	PR	UK	C9	
...				

```
SELECT FirstName, LastName, HireDate,
            DepartmentID
FROM Employee
```

FirstName	LastName	HireDate	DepartmentID
Aaron	Cooper	4/17/09	2
Lou	Donoghue	5/22/05	4
Alice	Bailey	9/1/99	(null)
Oswald	Hall	3/19/11	5
John	Velasquez	4/5/10	4
...			

Employee.DepartmentID is used in order to differentiate which one is needed because there are two instances of DepartmentID.

INNER JOIN = default JOIN (The null values will not appear in the result as well as the rows linked to it.)

```
SELECT FirstName, LastName, HireDate,
    Employee.DepartmentID, Name, Location
FROM Employee JOIN Department
ON Employee.DepartmentID = Department.DepartmentID
```

FirstName	LastName	HireDate	Employee.DepartmentID	Name	Location
Aaron	Cooper	4/17/09	2	R&D	AZ
Lou	Donoghue	5/22/05	4	Sales	CA
Oswald	Hall	3/19/11	5	PR	UK
John	Velasquez	4/5/10	4	Sales	CA
...					

LEFT OUTER JOIN (LEFT means employee takes precedence because it is on the left side of the JOIN)

```
SELECT FirstName, LastName, HireDate,
    Employee.DepartmentID, Name, Location
FROM Employee LEFT OUTER JOIN Department
ON Employee.DepartmentID = Department.DepartmentID
```

FirstName	LastName	HireDate	Employee.DepartmentID	Name	Location
Aaron	Cooper	4/17/09	2	R&D	AZ
Lou	Donoghue	5/22/05	4	Sales	CA
Alice	Bailey	9/1/99	(null)	(null)	(null)
Oswald	Hall	3/19/11	5	PR	UK
John	Velasquez	4/5/10	4	Sales	CA
...					

RIGHT OUTER JOIN (RIGHT = Department is on the right of JOIN)

```
SELECT FirstName, LastName, HireDate,
    Employee.DepartmentID, Name, Location
FROM Employee RIGHT OUTER JOIN Department
ON Employee.DepartmentID = Department.DepartmentID
```

FirstName	LastName	HireDate	Employee.DepartmentID	Name	Location
(null)	(null)	(null)	(null)	Production	CA
Aaron	Cooper	4/17/09	2	R&D	AZ
Lou	Donoghue	5/22/05	4	Sales	CA
Oswald	Hall	3/19/11	5	PR	UK
John	Velasquez	4/5/10	4	Sales	CA
(null)	(null)	(null)	(null)	Marketing	CA

Insert, Update and Delete

Create	INSERT
Read	SELECT
Update	UPDATE
Delete	DELETE

INSERT

```
INSERT INTO table
    (column1,column2...)
    VALUES (value1, value2...)
```

Employee
EmployeeID (PK)
FirstName
LastName
HireDate
Email
Department
Salary

```
INSERT INTO Employee
    (FirstName, LastName, Department, Salary)
    VALUES ('Joe', 'Allen', 'Sales', 45000)
```

EmployeeID	FirstName	LastName	HireDate	Email	Department	Salary
734	Joe	Allen	3/12/13	(null)	Sales	45000
auto			default value	null		

UPDATE

It is imperative to use a WHERE clause, because without it, every single row would be updated!

```
UPDATE table
SET column = value
WHERE condition
```

EmployeeID	FirstName	LastName	HireDate	Email	Department	Salary
734	Joe	Allen	3/12/13	(null)	Sales	45000

```
UPDATE Employee
SET Email = 'joea@twotreesoliveoil.com'
WHERE EmployeeID = 734
```

EmployeeID	FirstName	LastName	HireDate	Email	Department	Salary
734	Joe	Allen	3/12/13	joea@twotreesoliveoil.com	Sales	45000

DELETE

Be CAREFUL with this one. Everyone has a horror story from an SQL DELETE statement written too casually.

```
DELETE FROM table
WHERE condition
```

EmployeeID	FirstName	LastName	HireDate	Email	Department	Salary
734	Joe	Allen	3/12/13	(null)	Sales	45000

```
DELETE FROM Employee
WHERE EmployeeID = 734
```

EmployeeID	FirstName	LastName	HireDate	Email	Department	Salary
734	Joe	Allen	3/12/13	(null)	Sales	45000

```
DELETE FROM Employee
```

This is very dangerous because it deletes everything without warning.

A good practice for DELETE and UPDATE statements is to first write them as a SELECT statement.

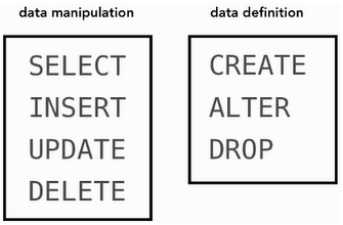
```
SELECT *
FROM Employee
WHERE EmployeeID = 734
```

EmployeeID	FirstName	LastName	HireDate	Email	Department	Salary
734	Joe	Allen	3/12/13	joea@twotreesoliveoil.com	Sales	45000

```
DELETE
FROM Employee
WHERE EmployeeID = 734
```

EmployeeID	FirstName	LastName	HireDate	Email	Department	Salary

Data Definition vs Manipulation



CREATE

```
CREATE   table
(column definitions)
```

```
CREATE  Employee
(EmployeeID     INTEGER   PRIMARY KEY,
 FirstName      VARCHAR(35) NOT NULL,
 LastName       VARCHAR(100) NOT NULL,
 Department     VARCHAR(30) NULL,
 Salary         INTEGER
);
```

ALTER

```
ALTER TABLE  table
```

```
ALTER TABLE Employee
ADD Email VARCHAR(100);
```

DROP

```
DROP TABLE  table
```

```
DROP TABLE Employee;
```

Be very careful with this one!

Developers are concerned with the first one as they would spend the bulk of the time doing data manipulation. The second one is more in the domain of database administrators. The third one as well, as it deals with permissions.

Indexing

Indexes are all about speed of access as they work as a book index i.e. they don't add any content, but rather help in finding it. They grow in importance as the database grows. They are very useful for columns which are constantly used. They are mostly relevant further down the road as an optimization tool.

On table creation, the primary key column is automatically declared as the clustered index, unless specified otherwise. It is usually the best option. Each table can only have one clustered index.

CustomerID	FirstName	LastName	Email	Address	...
551	Angela	Smith	angie@...	78 Privet Dr	...
561	Lee	Stout	lee@...	47 Main St	...
584	Michelle	Blackwell	mblackwell@	22 Acacia...	...
592	Lynn	Allen	la1942@...	1016B 1st...	...
...	...	...	...	...	...

clustered index

```
SELECT * FROM Customer WHERE CustomerID = 584;
```

This one is very fast because primary keys are already indexed and the search immediately jumps at the value.

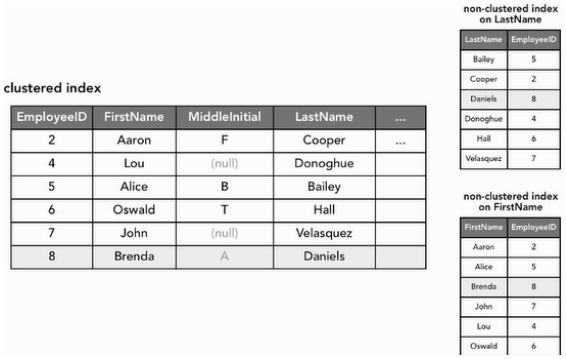
CustomerID	FirstName	LastName	Email	Address	...
551	Angela	Smith	angie@...	78 Privet Dr	...
561	Lee	Stout	lee@...	47 Main St	...
584	Michelle	Blackwell	mblackwell@	22 Acacia...	...
592	Lynn	Allen	la1942@...	1016B 1st...	...
...	...	...	...	...	...

clustered index

```
SELECT * FROM Customer WHERE LastName = 'Smith';
```

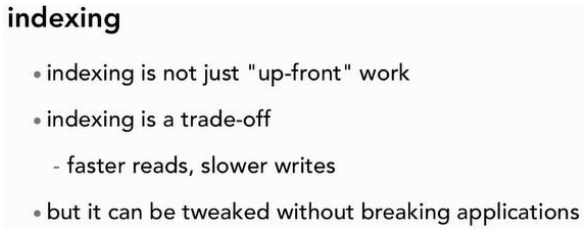
This one is significantly slower because the query has to go through every single row in order to find the value, since the column is not indexed.

The way fast searches are done is via non-clustered indexes which simply means that a column is picked to be sorted in order to be searchable faster.

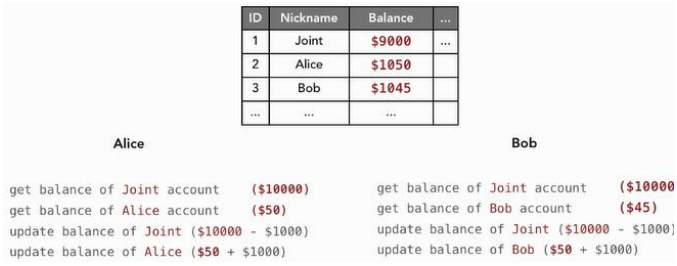


So why don't we just index every single column? This is a very bad idea, because then when inserting or updating data, what was once one operation, now becomes multiple ones for each column separately.

This slows the process significantly.This is why it is advised to use indexes only in high traffic columns.

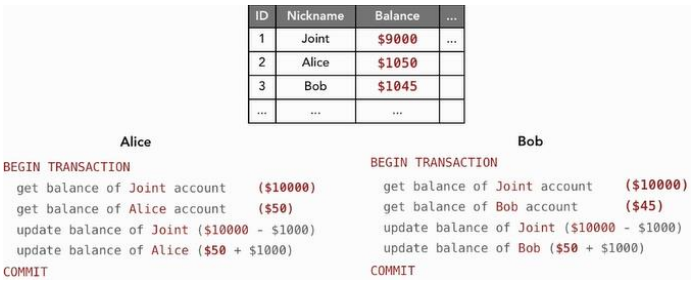


Conflict and Isolation



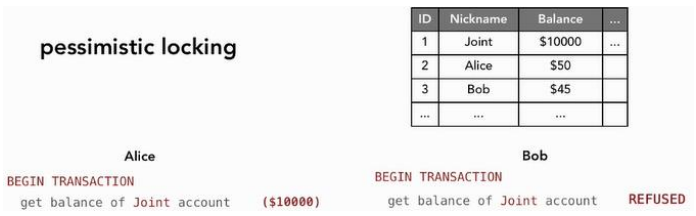
The balance should be 8000, not 9000. This problem occurs because both the transactions happened at the same time.

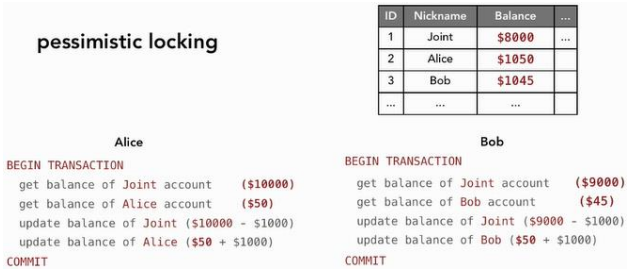
This is called a Race Condition Problem.

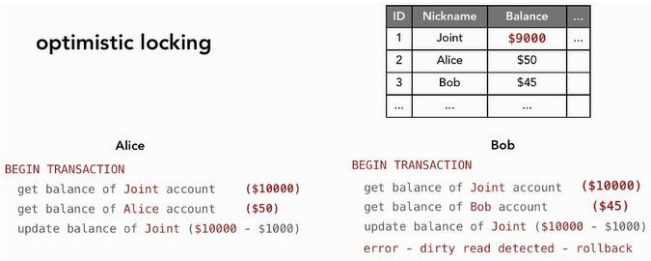


In order to deal with the problem, transactions are used. These see the actions as one thing rather than many separate ones. But, this is not enough because the content is not locked from editing.

Examples below for Pessimistic and Optimistic locking.







The difference from pessimistic locking is that the transaction is not locked outright, but rather it goes on until the field that was recently changed is ecnountered.

When this happens, a rollback occurs which goes at the start of the transaction.

Stored Procedures

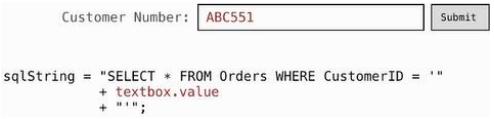
These are nothing but named chunks of SQL which are reusable, similar to functions in programming languages. Some organizations even force administrators to write ONLY stored procedures instead of plain SQL. This keeps the queries in the DBMS and it makes it easy to optimize.

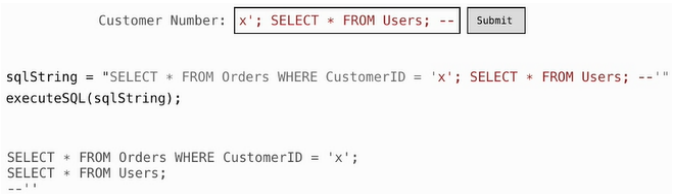
```
CREATE PROCEDURE HighlyPaid()
    SELECT * FROM Employee
    WHERE Salary > 50000
    ORDER BY LastName, FirstName
END;


CALL HighlyPaid();
```

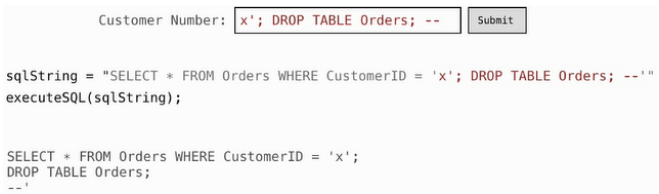
stored procedures can have parameters

```
CREATE PROCEDURE EmployeesInDept(IN dept VARCHAR(50))
    SELECT * FROM Employee
    WHERE Department = dept
    ORDER BY LastName, FirstName
END;


CALL EmployeesInDept('Accounting');
```

Stored procedures with parameters are a great way for preventing SQL injection attacks. This is true because stored procedures are inflexible and cannot be divided into separate statements.







The first statements returns no value. The second one is the injection. The third one acts as a comment due to the – before the ‘.

Database Options

Desktop Database Systems

Microsoft Access, FileMaker (Useful for internal use in small businesses, up to 10 users)

Used for simple tracking, such as contacts, events…

Relational DBMS

All DBMS's are based on the ideas of Edgard F. Codd in the 1970's.

DBMS's are not one thing. You install the database engine first, which lacks a UI, and then install an application for managing it. There may be separate apps for backing up data, reporting etc…

RDBMS

Name	Vendor	Released	AdminApplication	License
Oracle	Oracle	1979	Oracle SQL Developer	Commercial
DB2	IBM	1983	IBM Data Studio	Commercial
SQL Server	Microsoft	1989	SQL Server Management Studio	Commercial
MySQL	Oracle	1994	MySQL Workbench	Open Source
(etc.)	…	…	…	…

The pricing for commercial licenses varies greatly because of the specifics. There are many factors that can influence pricing such as number of users, number of CPUs, features selection…

In recent years, there has been growth in the hosted cloud based DBMS's. Microsoft Azure, Amazon RDS. You still need to design the database, but you don't need to worry about the hosting infrastructure.

Every DBMS has a free edition, which is usually called EXPRESS. They are limited in the amount of data that can be stored.

XML

XML database systems

Name	License	QueryLanguage
BaseX	Open Source	XQuery
Sedna	Open Source	XQuery
eXist	Open Source	XQuery

```
<?xml version="1.0"?>
<library>
   <course id="fop003">
      <author>Allardice, Simon</author>
      <title>Foundations of Programming: Databases</title>
      <genre>Developer</genre>
      <date_published>2013-01-30</date_published>
      <description>Getting started with databases and database technologies.</description>
   </course>
   <course id="java001">
      <author>Gassner, David</author>
      ...
```

RDBMS XML support

Name	XML Column Type?
Oracle	Yes
DB2	Yes
SQL Server	Yes
MySQL	No - store as text

example:

CourseID	Details
3	<?xml version="1.0"?><library><course id="fop003"><author>Allardice, Simon</author><title>Foundations of Programming: Databases</title>...

XML can be stored in relational databases. Some support having an XML column types, which means that xQuery can be used within the DBMS.

Object Oriented Database Systems

object-oriented database systems

Name
Objectivity/DB
Versant
VelocityDB

These solve the problem of objects in programming not mapping exatly with rows in a database table. These are popular in niche areas such as physics and engineering.

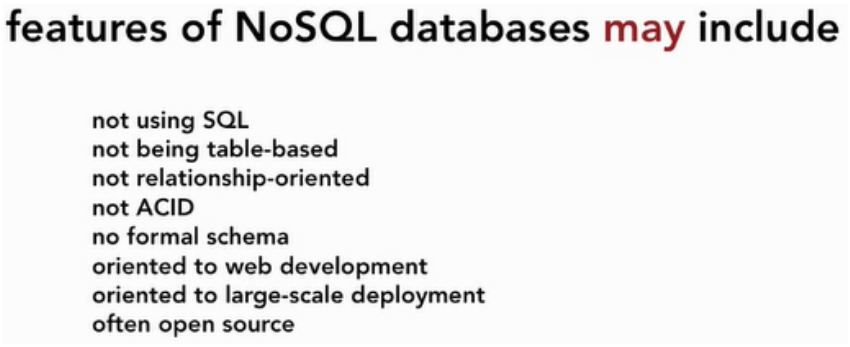
object-relational mapping (ORM)

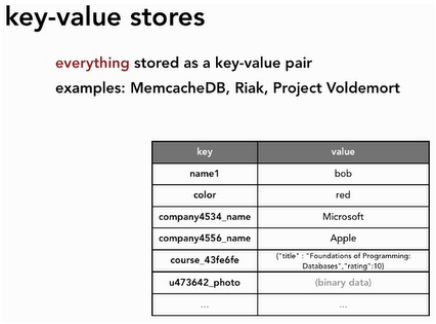
ORM Framework	Language
Hibernate	Java
Core Data	Objective-C
ActiveRecord	Ruby
NHibernate	C# / VB.NET

ORM's solve the same problems as OODBMS's. **C# .NET also uses the Entity Framework**. These simply create tables with columns based on objects automatically without the need of the user interfering.

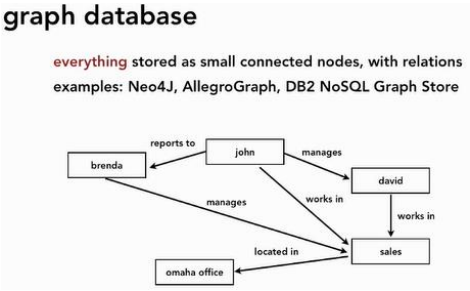
NoSQL Database Systems

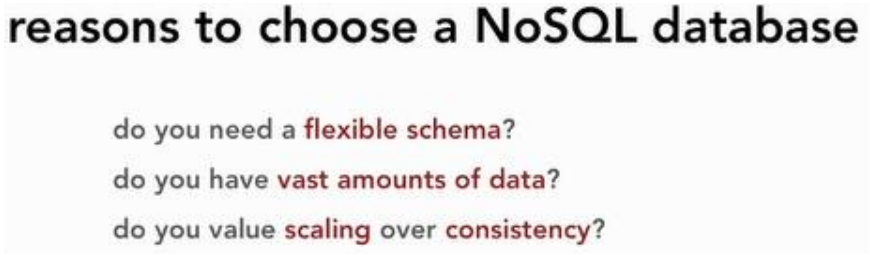
Means “Not only SQL”. These are not the new best way, but rather a solution to a new set of problems. These are commonly used for big data where hundreds of millions and billions of data is common. Also, these are useful when losing data is not detrimental. Ex. Click analytics vs Bank transfers. The latter cannot lose a single piece of data.





CouchDB, MongoDB





WPF for the Enterprise with C# and XAML

Panels

Grid – Rows and columns. Items are places with indexes.
Stack – Stacks items vertically or horizontally. Used for forms and labels.
Wrap – Items rearrange dynamically so that they fill up a row or a column.
Dock – Items can be docked on each side, with the first item being the one on top.
Canvas – Positions items absolutely via Top: Left:. Used for animations and games.

Controls

These can be grouped in:

- Text controls.
- Selection controls.
- List controls.
- Other controls.

Events

Event is the type of event (click, hover, change) and event handler is the code that executes.

All events handlers work in the same way with 2 passed parameters.

```
private void SaveButton_Click(object sender, RoutedEventArgs e) { }
private void Job_SelectionChanged(object sender, SelectionChangedEventArgs e) { }
```

Data Binding

It automates the connection between data and the view of the data. It significantly reduces the amount of code needed. It is the foundation of MVVM.

One Way Binding – Data is bound from its source i.e. the object that holds the data, to its target i.e. the object that displays the data.

Two Way Binding – The user is able to modify the data through the UI and have that data updated in the source. If the source changes while looking at the view, you want the view to be updated.

INPC (INotifyPropertyChanged) – It facilitates updating the view while the underlying data changes.

Element Binding – Instead of biding to a data source, the binding can be done to another element on the page.

Data Context – Binding happens between properties of the data source to the data target. Data context is the source itself i.e. the object whose properties you're binding from.

List Binding – Collections of data can also be bound, not just individual data. This can be bound to controls such as ListBoxes and ComboBoxes.

Data Templates – These tell which part of the data to be displayed.

Data Conversion – Sometimes, data does not display properly because the types are too different and the display is inappropriate. Data binding supports in WPF supports data conversion which happens at the source to the type expected at the target.

Data Validation – Many of the controls support data validation.

One Way Data Binding

In order to illustrate a binding, we need a binding source. Typically, that's some form of POCO i.e. Plain Old Class Object. In order to get the object, we need to declare a class and an instance of it.

A binding can point to a property, but nothing happens until it is specified which object the property belongs to. This is done by setting the data context and there are many ways to do this. The easiest way is:

```
// MainWindow.xaml.cs
Car car1 = new Car("Audi", 2008);
DataContext = car1;
```

```
// MainWindow.xaml
<TextBlock Text="{Binding name}"/>
```

INotifyPropertChanged

This is an interface that needs to be implemented and it requires **System.ComponentModel**. It also requires implementing an event of type **PropertyChangedEventHandler** like so:

public event PropertyChangedEventHandler propertyChanged;

PropertyChanged is called each time a property is being updated. To facilitate that, a helper method is used called **OnPropertyChanged** which uses **[CallerMemberName]** which requires using **System.Runtime.CompilerServices**. What this does is passing the name of the property that calls this method.

```
public event PropertyChangedEventHandler PropertyChanged;
  private void OnPropertyChanged(
      [CallerMemberName] string caller = "" )
  {
      if ( PropertyChanged != null )
      {
          PropertyChanged( this,
             new PropertyChangedEventArgs( caller ) );
      }
  }
```

In order for this to work, we can no longer use automatic properties because we need to call **OnPropertyChanged** method every time we call the setter.

```
public class Employee : INotifyPropertyChanged
{
   private string name;
   public string Name
   {
      get { return name; }
      set
      {
         name = value;
         OnPropertyChanged();
      }
   }
```

Two Way Data Binding

Two-way binding means that any data-related changes affecting the model are *immediately propagated* to the matching view(s), and that any changes made in the view(s) (say, by the user) are *immediately reflected* in the underlying model. When app data changes, so does the UI, and conversely.

It is activated by adding a mode with the selection TwoWay.

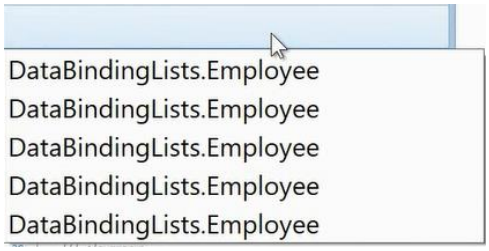
<TextBlock Text="{Binding name, mode=TwoWay }" />

Data Binding Lists

ObservableCollection = The view will be notified of any addition or removal in the collection.

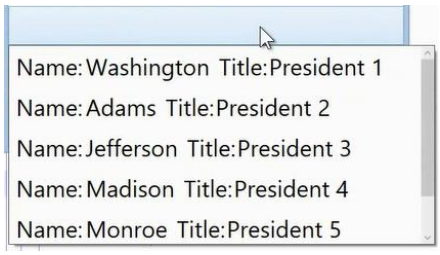
If only **ItemSource=”{Binding}”** is used, the result would display **DataBindingLists.Employee** for each object in the list. This happens because it is not specified what to be displayed about the objects. This is accomplished with a template.

```
<ComboBox Name="Presidents"
          ItemsSource="{Binding}"
          FontSize="30"
          Height="50"
          Width="550">

</ComboBox>
```



An **ItemTemplate** describes what each item should look like. Inside, we use a DataTemplate in which the XAML is defined for the look of the object which is repeated for each item.

```
<ComboBox Name="Presidents"
          ItemsSource="{Binding}"
          FontSize="30"
          Height="50"
          Width="550">
  <ComboBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal"
                  Margin="2">
        <TextBlock Text="Name:"
                   Margin="2" />
        <TextBlock Text="{Binding Name}"
                   Margin="2" />
        <TextBlock Text="Title:"
                   Margin="10,2,0,2" />
        <TextBlock Text="{Binding Title}"
                   Margin="2" />
      </StackPanel>

    </DataTemplate>
  </ComboBox.ItemTemplate>
```



Data Binding Elements

```
<StackPanel Orientation="Horizontal">
  <Slider Name="mySlider"
          Minimum="0"
          Maximum="100"
          Width="300" />
  <TextBlock Margin="5"
             Text="{Binding Value, ElementName=mySlider}" />
</StackPanel>
```

Asynchronous Programming

Great explanation: https://www.youtube.com/watch?v=MCW_eJA2FeY

This deals with the unresponsiveness of an application. This usually happens because of an infinite loop, a deadlock or using the UI thread for performing long operations. Ex. This code downloads a picture on click:

```
private void ButtonClick(object sender, EventArgs e)
{
    var client = new WebClient();
    var imageData = client.DownloadData("http://image-url");
    pictureBox.Image = Image.FromStream(new MemoryStream(imageData));
}
```

This may finish fast on a development PC, but when deployed, a real world slow internet connection will cause a delay until the picture is downloaded in which the UI is frozen.

A fix for this might use tasks and task continuations to offload long operations off the main thread. But, this causes another problem in which the download may trigger another action which would require a cascade of tasks and continuations for every single possibility, making the code hard to read.

To fix this, we use the C# **async / await** model:

```
private async void ButtonClick(object sender, EventArgs e)
{
    var client = new WebClient();
    var imageData = await client.DownloadDataTaskAsync("http://image-url");
    pictureBox.Image = Image.FromStream(new MemoryStream(imageData));
}
```

This causes the download to execute asynchronously, without blocking the UI. The **await** keyword ensures nothing happens before the called asynchronous method is finished. Both keywords – **async** and **await** – always work together i.e. **await** without **async** is not allowed.

In other words, async / await allows the user to build “normal” applications with straightforward logic and yet have them run asynchronously**. Write single threaded code which acts asynchronously**. When the await keyword is hit, it offloads the processing to another thread and it continues with the normal code.

If asynchronous programming is not used, an 8 core processor would act as 1 core because everything happens in the main thread, hence the freezing in a long operation.

Advanced Controls

- Tab Control
- Data Grid
- Tree View
- Status Bar
- Menus

Design Patterns

Design patterns are well tested solutions to common problems in software development.

They exists in order to make code change over time easier, by making it more flexible and maintainable.

Think of design patterns as guidelines for how to structure objects and their behavior to get a particular result.

The primary goal is to structure code in order to be flexible and resilient.

"I have a problem. When one of my object changes, I need to let all these other objects know. Is there a good way to do that?"

- This is a common problem
- There is a proven method to solve it: **The Observer Pattern**

Design patterns are also useful for communication. Example:

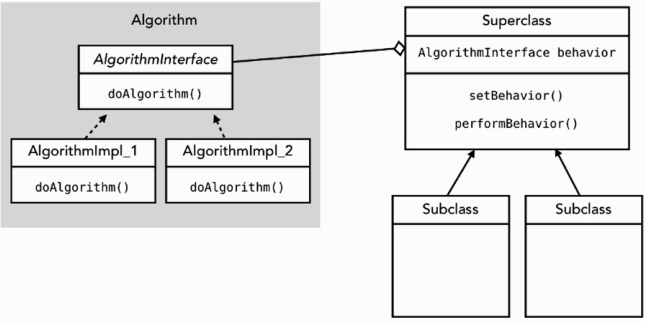
"First, register your object with mine, then implement an update method, then when it gets called, call my getValue method."

VS

"Just use the Observer Pattern"

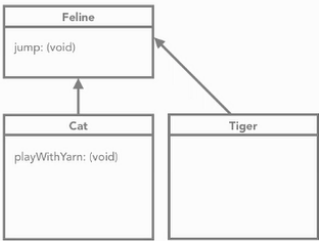
Strategy Pattern

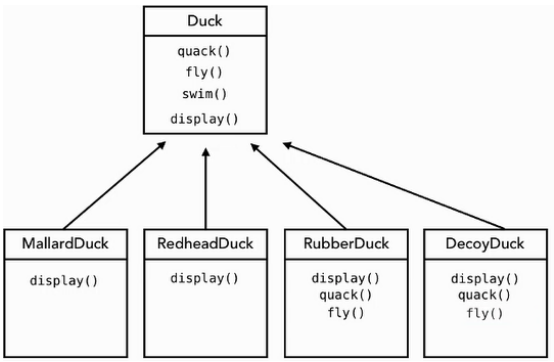
It is the combination of composition (interfaces) for behaviours that need more flexibility and inheritance for behaviours that don't need to change.



Inheritance

- Inheritance is a core principle of OO programming
- But we tend to overuse it
- Often results in design and code that is inflexible
- Let's look at an example: *a duck simulator*





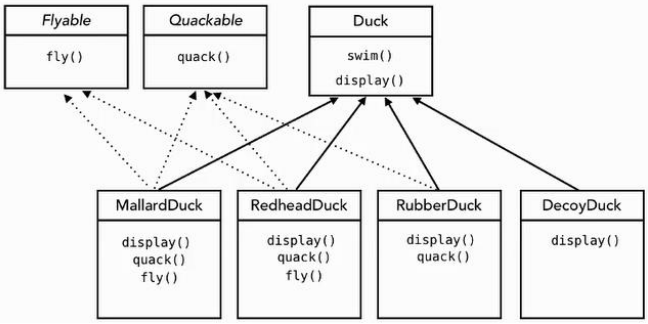
Problems with Our Design

- We have code duplicated across classes
- Hard to gain knowledge of all the ducks
- Changes can affect other ducks
- Runtime behavior changes are difficult

What About Using Interfaces?

- Allow different classes to share similarities
- Not all classes need to have the same behavior
- Let us try moving duck behaviors into interfaces

Implementing Ducks with Interfaces



Also Problematic

- Solves part of the problem, but...
- Absolutely destroys code reuse
- Becomes a maintenance nightmare
- Does not allow for runtime changes in behaviors other than flying or quacking

Reviewing Our Attempts

- **Tried inheritance: it did not work well**

  Behavior changed across subclasses, and not appropriate for all subclasses to have those behaviors
- **Tried interfaces: also did not work well**

  Sounded promising, but interfaces supply no implementation and destroyed reuse entirely
- **So, where do we go from here? Has OO failed us?**

Design Principle #1

Identify the aspects of your code that vary and separate them from what stays the same.

"Encapsulate What Varies"

- **If some aspect of code is changing,**

  That's a sign you should pull it out and separate it
- **By separating out the parts of your code that vary**

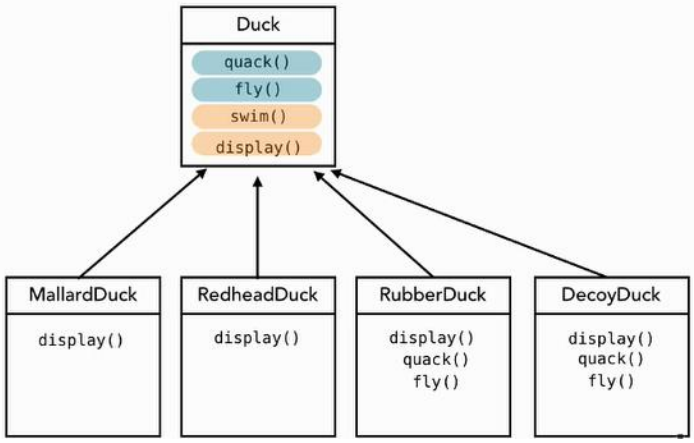
  You can extend or alter them without affecting the rest your code
- ***This principle is fundamental to almost every design pattern***

All patterns let some part of the code vary independently of the other parts

Fewer surprises from code changes and increased flexibility in your code

Change is the only constant in software desing.

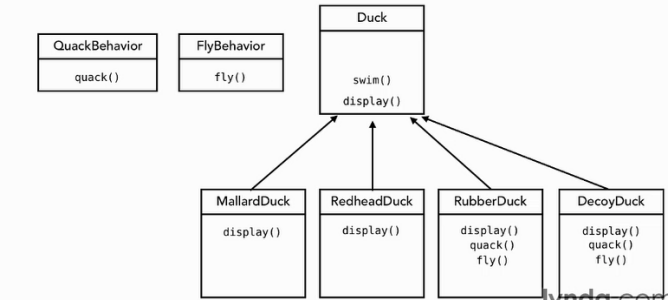
Identifying What Changes



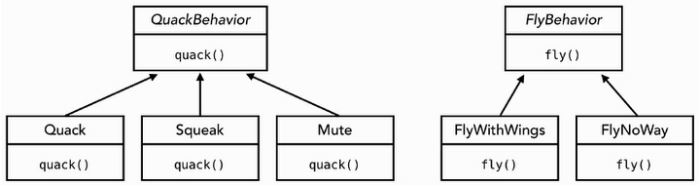
Design Principle #2

Program to an interface, not an implementation

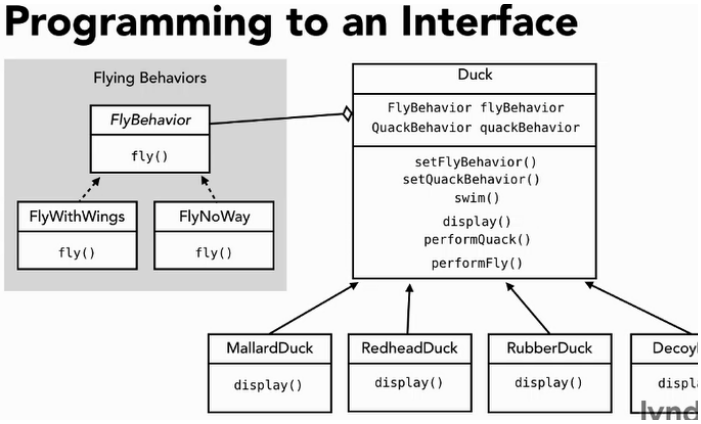
Identifying What Changes



Programming to an Interface



Programming to an Interface



Programming to an Interface with Ducks

- **Rather than relying on an implementation of behavior in our ducks**

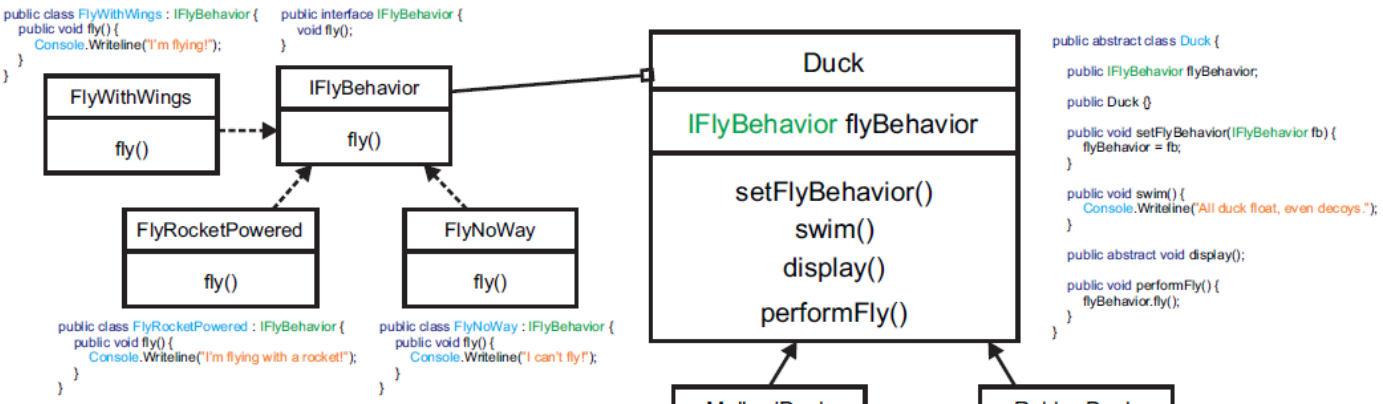
  We are relying on an interface
- **FlyBehavior and QuackBehavior are now interfaces**

  A *class* that implements a specific behavior

Programming to an Interface with Ducks

- This way we are no longer locked into specific implementations
- And ducks do not need to know details of how they implement the behaviors!

Strategy Pattern C#




```
public class FlyWithWings : IFlyBehavior {
    public void fly() {
        Console.Writeline("I'm flying!");
    }
}
```

```
public interface IFlyBehavior {
    void fly();
}
```

```
public class FlyRocketPowered : IFlyBehavior {
    public void fly() {
        Console.Writeline("I'm flying with a rocket!");
    }
}
```

```
public class FlyNoWay : IFlyBehavior {
    public void fly() {
        Console.Writeline("I can't fly!");
    }
}
```

```
public abstract class Duck {

    public IFlyBehavior flyBehavior;

    public Duck {}

    public void setFlyBehavior(IFlyBehavior fb) {
        flyBehavior = fb;
    }

    public void swim() {
        Console.Writeline("All duck float, even decoys.");
    }

    public abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }
}
```

```
public class MallardDuck : Duck {

    public MallardDuck() {
        flyBehavior = new FlyWithWings();
    }

    public override void display() {
        Console.Writeline("I'm a Mallard duck.");
    }
}
```

```
public class RubberDuck : Duck {

    public RubberDuck() {
        flyBehavior = new FlyNoWay();
    }

    public override void display() {
        Console.Writeline("I'm a rubber duckie.");
    }
}
```

Main

```
class Program {

    static void main(string[] args) {

    MallardDuck mallard = new MallardDuck();
    RubberDuck rubberDuckie = new RubberDuck();

    mallard.performFly();
    rubberDuckie.performFly();
    rubberDuckie.setFlyBehavir(new FlyRocketPowered());
    rubberDuckie.performFly();

    }
}
```

Output

```
I'm flying!
I can't fly!
I'm flying with a rocket!
```

Design Principle #3

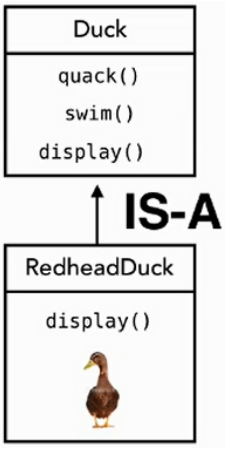
Favor composition over inheritance.

Inheritance = IS A relationship, Composition = HAS A relationship.

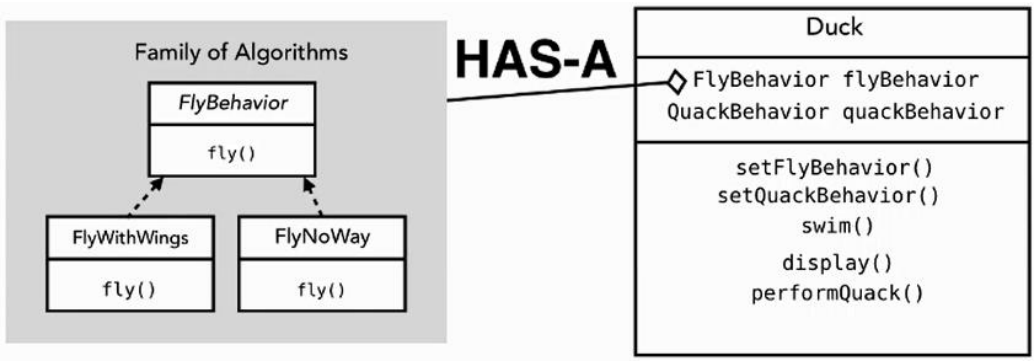
- **Note we're using a HAS-A relationship**

  Each duck *has* a FlyBehavior and Quack Behavior
- **Instead of inheriting behavior, we're composing it**

  A duck is composed with a fly and quack behavior
- **Important technique captured in design principle**



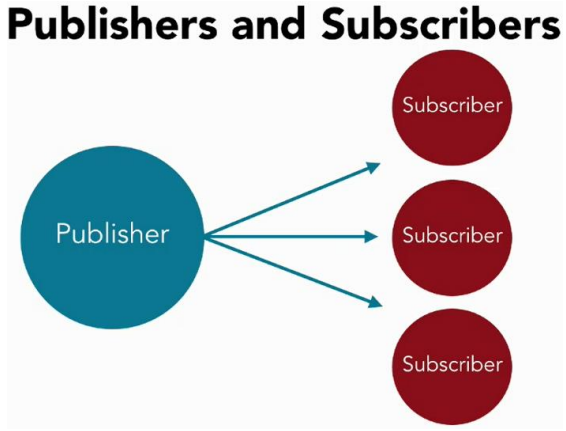
VS



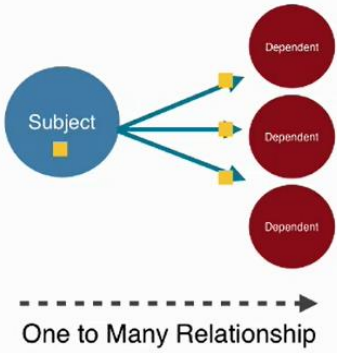
Observer Patter

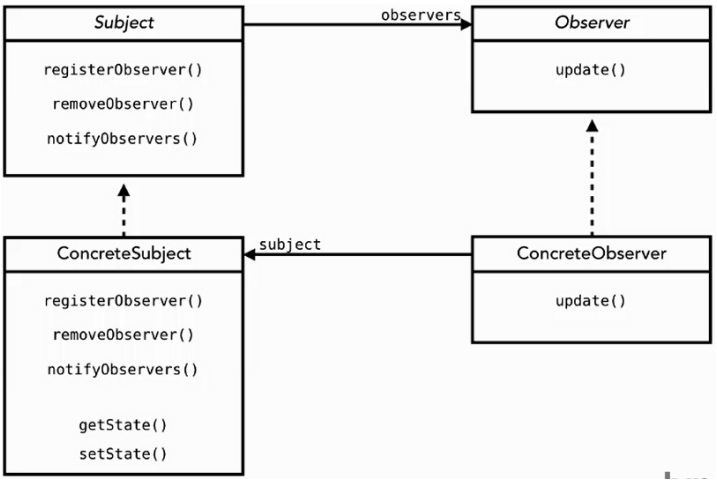
Example

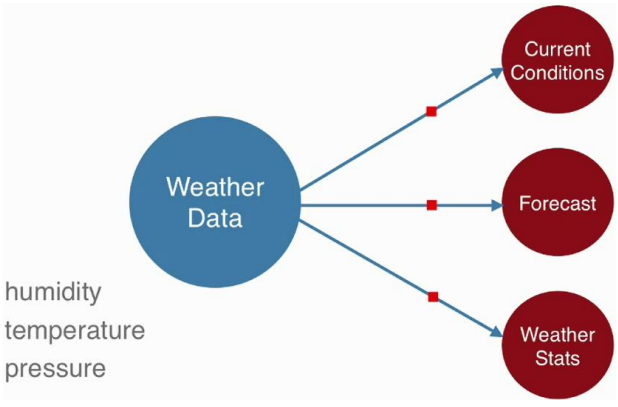
- **A publisher creates a new magazine and begins publishing issues**
  - You subscribe and receive issues as long as you stay subscribed
  - You can unsubscribe at any time
  - Others can also subscribe
  - If publisher ceases business, you stop receiving issues

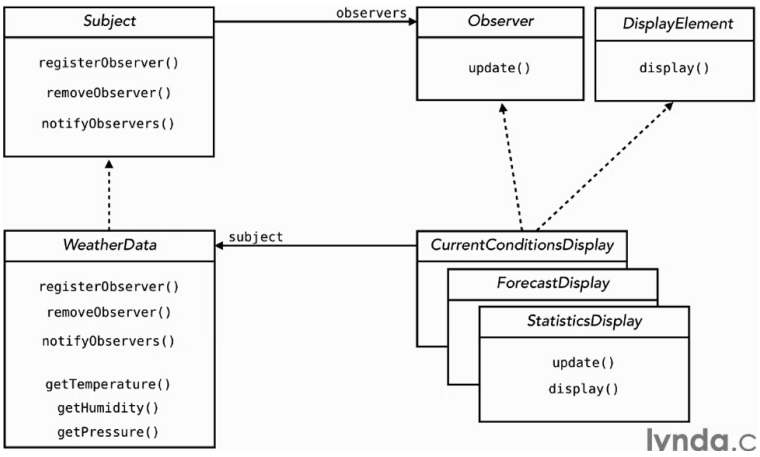


Defines a one-to-many relationship









WPF MVVM In Depth

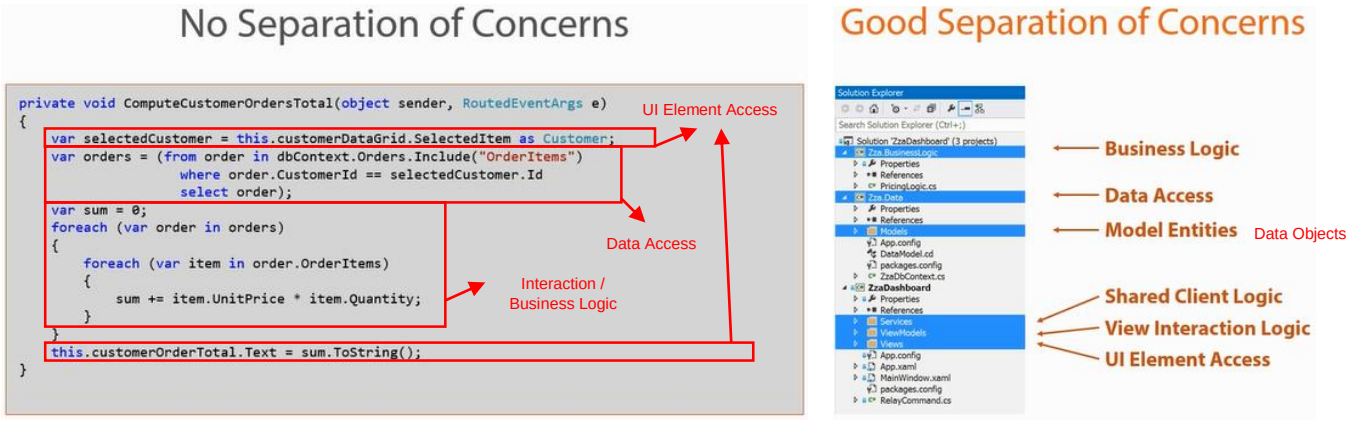
MVVM (Model View ViewModel) is all about organizing and structuring code in a way that leads to maintainable, testable and extensible applications.

Check out “Developing Extensible Software” by Miguel Castro, and “Extending XAML Applications with Behaviors” by Brian Noyes.

MVVM Fundamentals

Separation of Concerns

MVVM is mostly about trying to achieve good separation of concerns.



Related UI Separation Patterns

MVVM is an evolution of other UI separation patterns.

MVC (Model View Controller)

It dates back to early 1970’s. It is favored by modern web platforms. One of the main differences between MVVM and MVC is that there is a decoupled lifetime between the controller and the view. The Controller produces a View, but may not stick around after that, until a new request comes from the user.

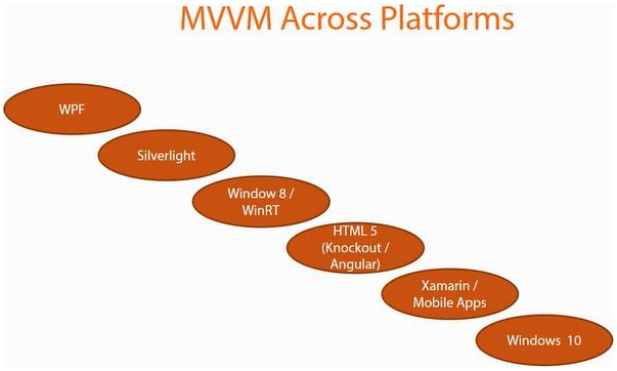
MVP (Model View Presenter)

It was introduced in the mid 2000’s as a result of the MVC not being the best fit for desktop applications, which required more stateful client views that stuck around in memory, as well as the supporting interaction logic. MVP is a more nuanced MVC pattern technically.

The difference between a Presenter and Controller is that the lifetimes of the Presenter and the View were coupled and they generally had a more ongoing conversation as the user interacted with the view. This was done mostly in the form of back and forth method calls between the two parts.

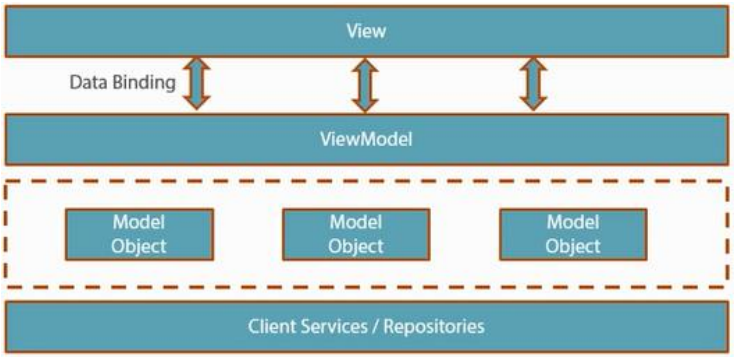
MVVM (Model View ViewModel)

It was first introduced along with WPF by Microsoft. MVVM is a nuance of MVP, where the explicit method calls between the View and its counterpart were replaced by two way data binding, flowing data and communications between the View and the ViewModel.



MVVM Responsibilities

MVVM is a layered architecture for the client side. The presentation layer is composed of the Views, the logic layer are the ViewModels and the persistence layer is the combination of the model objects and the client's services that produce and persist them through either direct data access in a two tier application or via service calls via n tier application.



Model Responsibilities

Contain the client data: The Model is the client side data model that supports the Views in the application. It's composed of objects with properties and backing member variables to hold the discrete pieces of data in memory.

Expose relationships between model objects: Some of those properties may reference other model objects, forming the object graph that is the object model as a whole.

Computed properties: Model Object may also expose computed properties. Properties whose value is computed based on the value of other properties in the model or information from the client execution context.

Raise change notifications (INotifyPropertyChanged.PropertyChanged): Because you will often be binding directly to model properties, model objects should raise property changed notifications, which for WPF data binding means implementing the **INotifyPropertyChanged** interface and firing the **PropertyChanged** event in the property set block.

Validation: Often you will embed validation information on the model objects, so it can work with the WPF data binding validation features through interfaces such as **INotifyDataErrorInfo / IDataErrorInfo**.

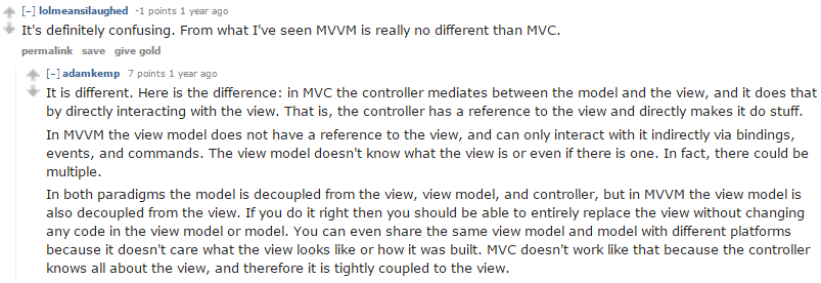
View Responsibilities

Structural definition of what the user sees on the screen: The goal of the View is to define the structure of what the user sees on the screen, which can be composed of static and dynamic parts. Static is the XAML hierarchy that defines the controls and their layout, of which the View is composed of. The dynamic part is any animation and state changes that are defined as part of the View.

GOAL: “No Code Behind”: There should be as least code as possible behind the View. It is impossible to have 0 code behind it. You would at least need the constructor and the call to initialize component, that trigger XAML parsing as the View is being constructed. The idea is to resist the urge to use event handling code, as well as interaction and data manipualtion.

Reality: Sometimes code behind is needed: Any code that is required to have a reference to a UI element is inherently View code. Ex. Animations expressed as code instead of XAML. Many controls have parts of their API that is not conducive to data binding, forcing you to code the behavior. The key concept is when using the MVVM pattern, you should always analyze the code you put in the code behind and see if there is any way to make it more declarative in XAML itself with mechanisms like data binding, commands or behavior, to dispatch calls into the ViewModel and put that logic there instead.

ViewModel Responsibilites



[–] **lolmeansilaughed** -1 points 1 year ago

It's definitely confusing. From what I've seen MVVM is really no different than MVC.

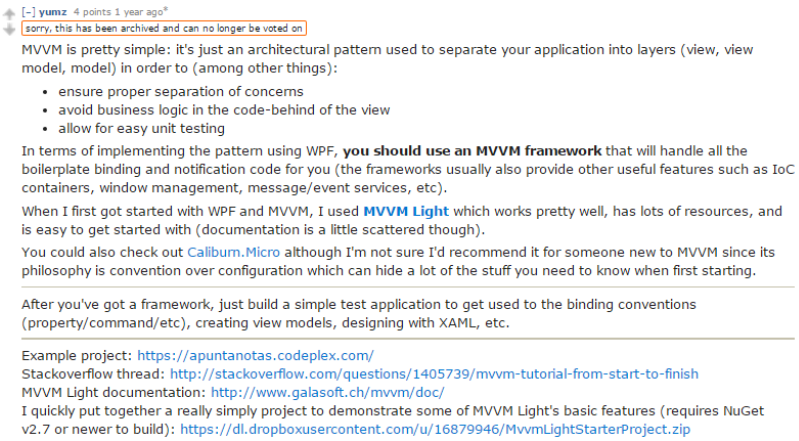
permalink save give gold

[–] **adamkemp** 7 points 1 year ago

It is different. Here is the difference: in MVC the controller mediates between the model and the view, and it does that by directly interacting with the view. That is, the controller has a reference to the view and directly makes it do stuff.

In MVVM the view model does not have a reference to the view, and can only interact with it indirectly via bindings, events, and commands. The view model doesn't know what the view is or even if there is one. In fact, there could be multiple.

In both paradigms the model is decoupled from the view, view model, and controller, but in MVVM the view model is also decoupled from the view. If you do it right then you should be able to entirely replace the view without changing any code in the view model or model. You can even share the same view model and model with different platforms because it doesn't care what the view looks like or how it was built. MVC doesn't work like that because the controller knows all about the view, and therefore it is tightly coupled to the view.



[–] **yumz** 4 points 1 year ago*

sorry, this has been archived and can no longer be voted on

MVVM is pretty simple: it's just an architectural pattern used to separate your application into layers (view, view model, model) in order to (among other things):

- ensure proper separation of concerns
- avoid business logic in the code-behind of the view
- allow for easy unit testing

In terms of implementing the pattern using WPF, **you should use an MVVM framework** that will handle all the boilerplate binding and notification code for you (the frameworks usually also provide other useful features such as IoC containers, window management, message/event services, etc).

When I first got started with WPF and MVVM, I used **MVVM Light** which works pretty well, has lots of resources, and is easy to get started with (documentation is a little scattered though).

You could also check out Caliburn.Micro although I'm not sure I'd recommend it for someone new to MVVM since its philosophy is convention over configuration which can hide a lot of the stuff you need to know when first starting.

After you've got a framework, just build a simple test application to get used to the binding conventions (property/command/etc), creating view models, designing with XAML, etc.

Example project: https://apuntanotas.codeplex.com/
Stackoverflow thread: http://stackoverflow.com/questions/1405739/mvvm-tutorial-from-start-to-finish
MVVM Light documentation: http://www.galasoft.ch/mvvm/doc/
I quickly put together a really simply project to demonstrate some of MVVM Light's basic features (requires NuGet v2.7 or newer to build): https://dl.dropboxusercontent.com/u/16879946/MvvmLightStarterProject.zip

"Dependency Injection" is a 25-dollar term for a 5-cent concept. Dependency injection means giving an object its instance variables.

Most of the Views are User Controlsl.

WPF MVVM Step by Step by .NET Interview Preparation videos on Youtube

.dll = class

MVVM Obsertvation

A framework is actually not necessary to implement MVVM and you should seriously consider whether using one is right for your WPF application or not. Many applications do not need much of the features the frameworks provide. However, there are two common classes that all MVVM frameworks contain. These are ViewModelBase and RelayCommand. Though some frameworks may give them different names or implement them slightly differently, they all have these classes. For example, MVVM Foundation names the ViewModelBase differently. It is called ObservableObject, which is more appropriate because it is incorrect to assume that all objects that implement INotifyPropertyChanged are going to be ViewModel objects.

Instead of installing and using an MVVM framework, you could simply include these classes in your application, as these are all you need to implement MVVM.

- ObservableObject
- RelayCommand

Bonus: Async / Await

While these two classes are enough, you may want to investigate how different MVVM Frameworks implement and what else they implement and why. You may find that another feature implemented is exactly what you need in your application and knowing about it could save you time.

Events and Delegates

Events

They are a mechanism for communication between objects i.e. when something happens in an objects, it notifies another object about that. This helps with designing loosely coupled applications. Ex.

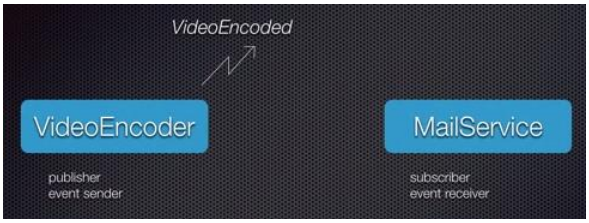
```
public class VideoEncoder
{
    public void Encode(Video video)
    {
        // Encoding logic
        // …

        _mailService.Send(new Mail());

        _messageService.Send(new Text());
    }
}
```

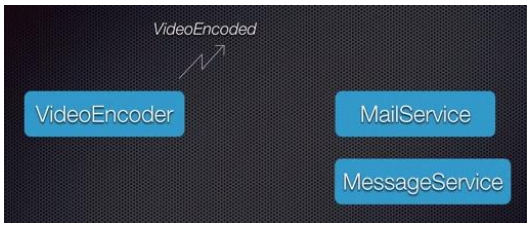
This is a video encoding class with a method that encodes a passed in video.

After it does the encoding, it sends an email and text to the owner of the video.

The problem here is that the adding of services changes the method which means the class has to be recompiled, as well as all the places using it.

This problem of tight coupling can be solved with an event called **VideoEncoded**. The interesting thing here is that the **VideoEncoder** knows nothing about the **MailService**. This makes it very easy to extend the application by adding other notification services like **MessageService** without altering the **VideoEncoder**, which leads to recompiling and possibly bugs.






```
public class VideoEncoder
{
    public void Encode(Video video)
    {
        // Encoding logic
        // …

        OnVideoEncoded();
    }
}
```

OnVideoEncoded() notifies the subscribers.

Any amount of subscribers as well as notification services can be added in this class without changing any code.

How does the VideoEncoder notify the subscribers? By **invoking a method (EventHandler) in the subscriber**. By how does it know which method to invoke? This is done **via a contract (a method with a specific signature called a Delegate) between publishers and subscribers**.

```
public void OnVideoEncoded(object source, EventArgs e)
{
}
```

This is an **EventHandler**.

This is a typical implementation of a method in the subscriber which is called an **EventHandler**. It is a method that is called by the publisher when the event is raised.

There needs to be a method like that in both **MailService** and **MessageService**.

The **VideoEncoder** knows nothing of them and it needs to invoke the **EventHandlers** in them.

This is done via **Delegates**.

Delegates

How do we tell VideoEncoder what method to call? With a delegate. Delegates are an agreement / contract between publishers and subscribers. They also determine the signature of the event handler method in subscriber.

To give a class the ability to publish an event, 3 steps are needed.

1. Define a delegate. A contract between the publisher and subscriber. A delegate determines the signature of the method in the subscriber that will be called when the publisher (VideoEncoder) publishes an event.
2. Define an event based on the delegate.
3. Raise (publish) the event.

Angular.js

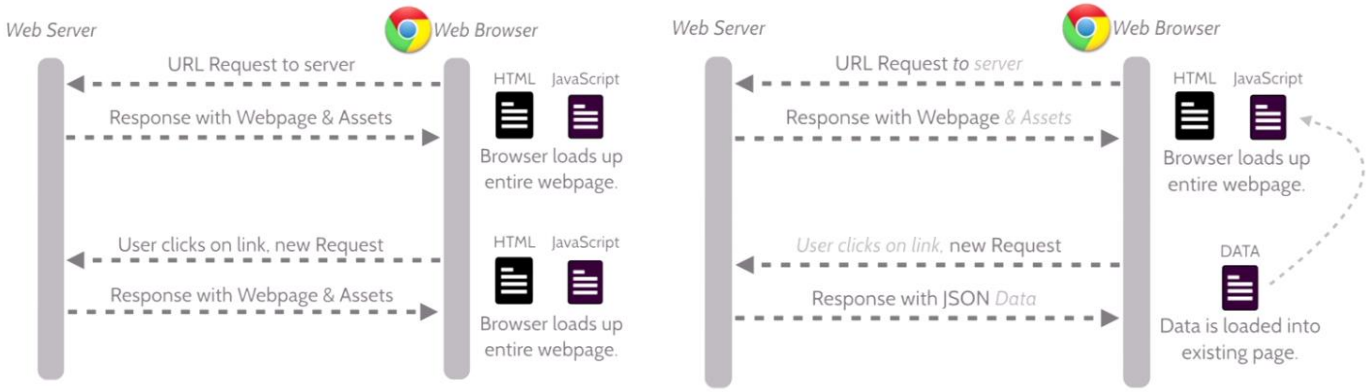
Overview

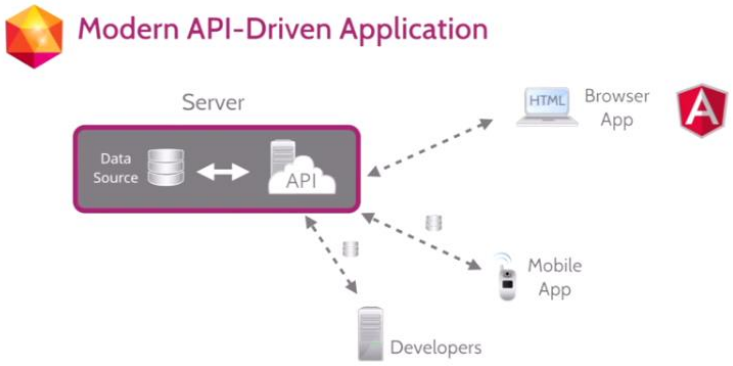
Code School Course

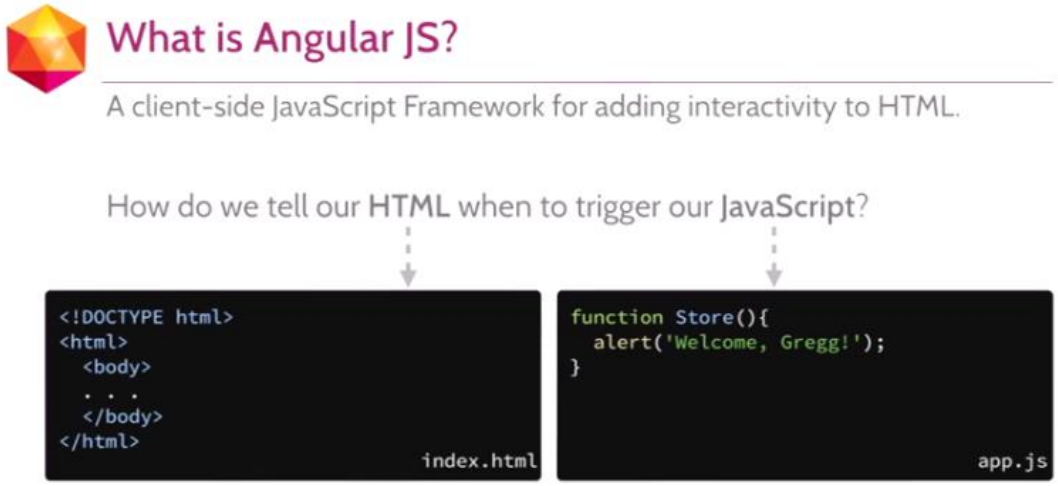
Testing

BDD = Behavior Driven Development
TDD = Test Driven Development

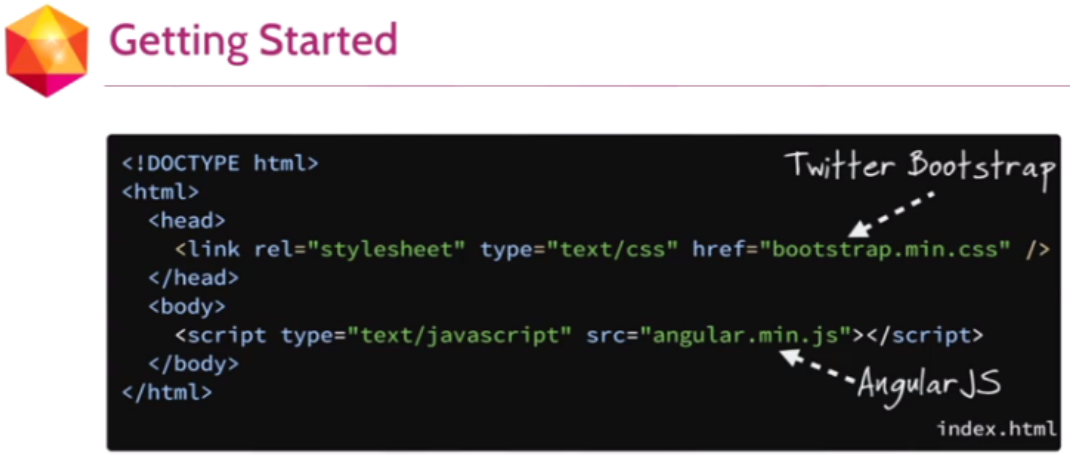
Traditional vs Responsive website



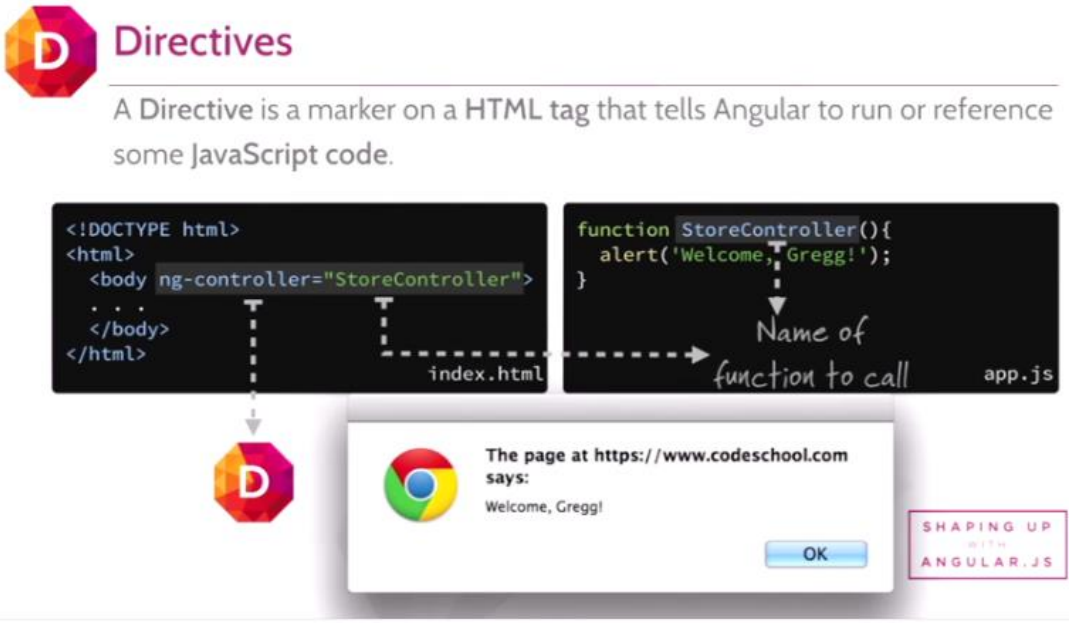




Installing



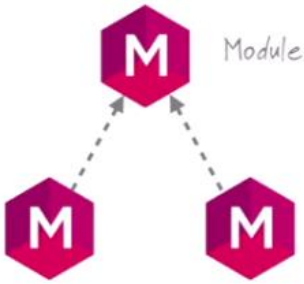
Directives



Modules

Modules

- Where we write pieces of our Angular application.
- Makes our code more maintainable, testable, and readable.
- Where we define dependencies for our app.



SHAPING UP WITH ANGULAR.JS

Creating Our First Module

```
var app = angular.module('store', [ ]);
```

AngularJS — Application Name — Dependencies

Other libraries we might need. We have none... for now...

Including Our Module

```
<!DOCTYPE html>
<html ng-app="store">
  <head>
    <link rel="stylesheet" type="text/css" href="bootstrap.min.css" />
  </head>
  <body>
    <script type="text/javascript" src="angular.min.js"></script>
    <script type="text/javascript" src="app.js"></script>
  </body>
</html>
```

Run this module when the document loads.

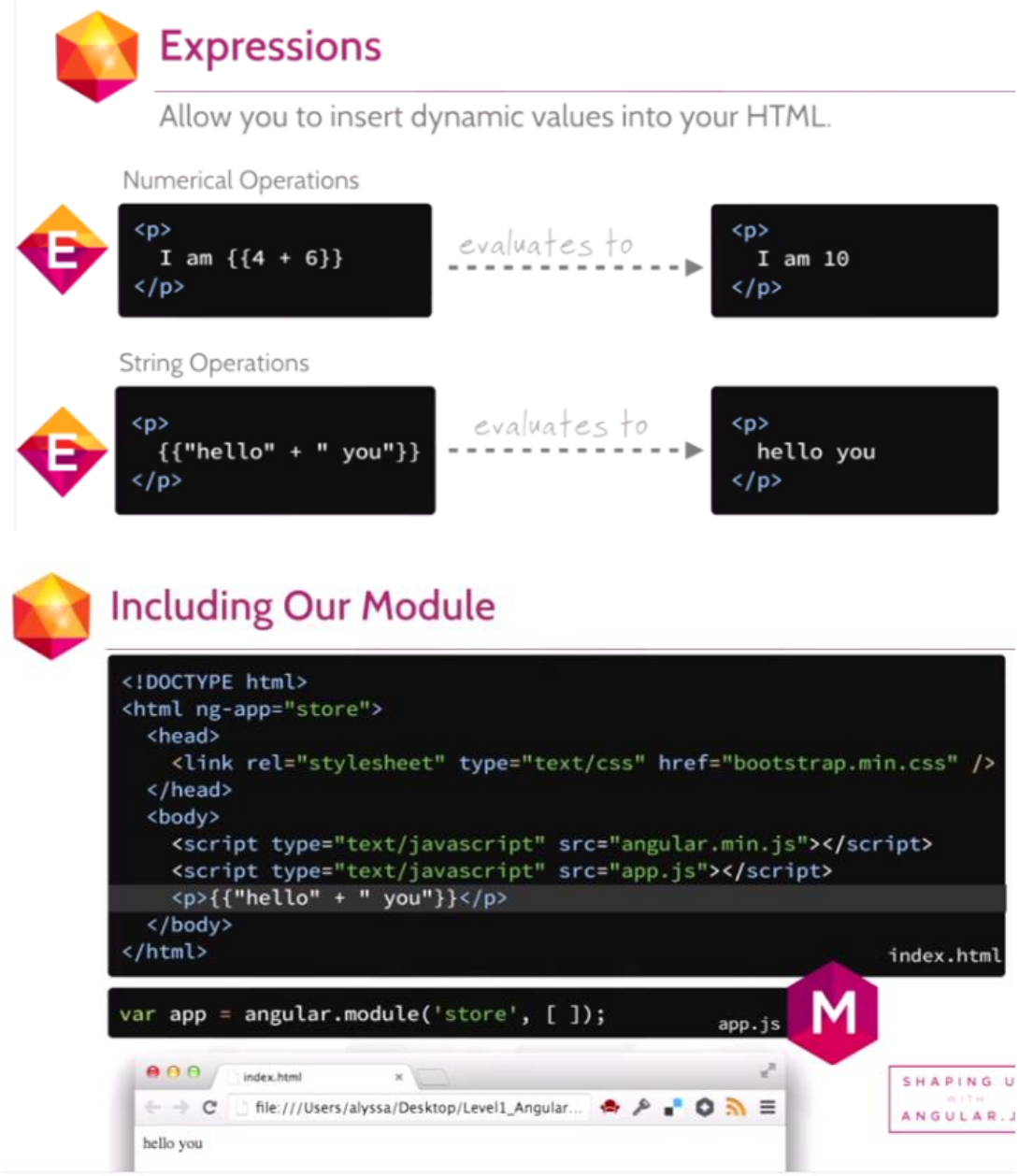
index.html

```
var app = angular.module('store', [ ]);
```

app.js

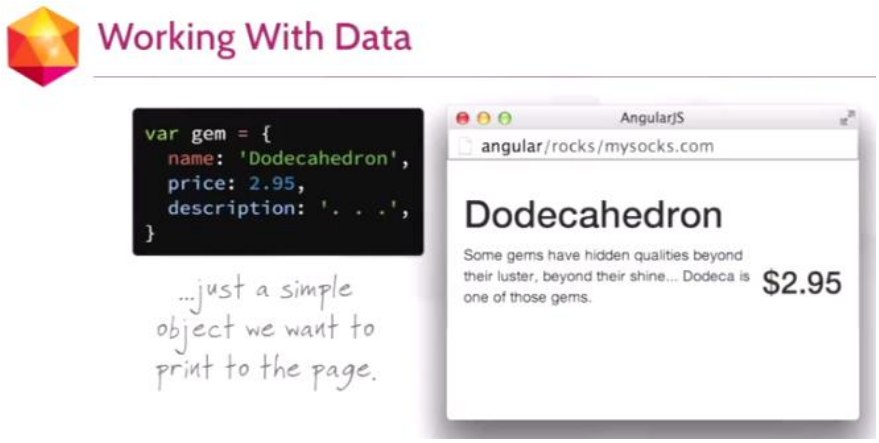
SHAPING UP WITH ANGULAR.JS

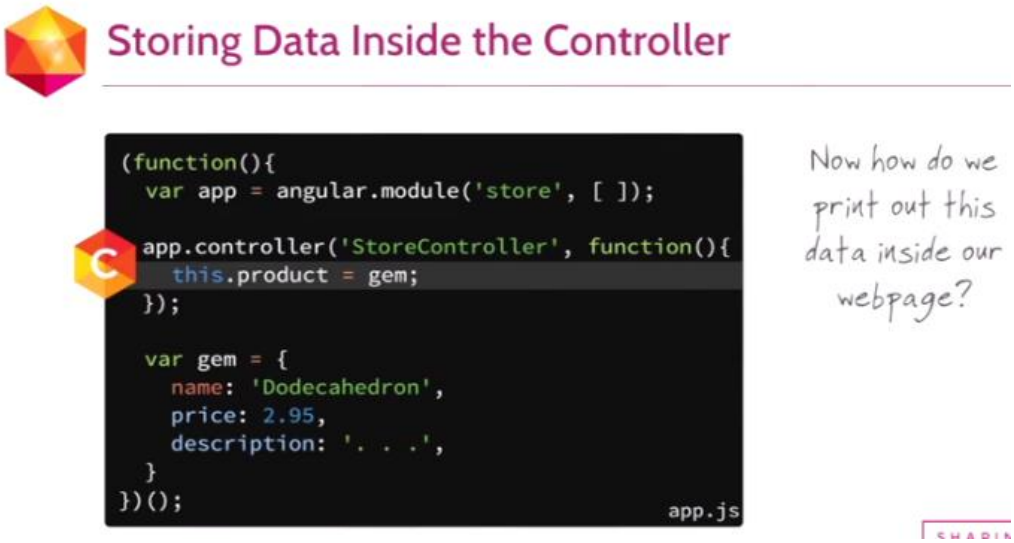
Expressions



Controllers

Controllers = Get data onto the page





Our Current HTML

```
<!DOCTYPE html>
<html ng-app="store">
  <head>
    <link rel="stylesheet" type="text/css" href="bootstrap.min.css" />
  </head>
  <body>
    <div>
      <h1> Product Name </h1>
      <h2> $Product Price </h2>
      <p> Product Description </p>
    </div>
    <script type="text/javascript" src="angular.min.js"></script>
    <script type="text/javascript" src="app.js"></script>
  </body>
</html>
```

Lets load our data into this part of the page.

index.html

Attaching the Controller

```
<body>
  <div>
    <h1> Product Name </h1>
    <h2> $Product Price </h2>
    <p> Product Description </p>
  </div>
  <script type="text/javascript" src="angular.min.js"></script>
  <script type="text/javascript" src="app.js"></script>
</body>
```

index.html

```
(function(){
  var app = angular.module('store', [ ]);

  app.controller('StoreController', function(){
    this.product = gem;
  });
  . . .
})();
```

app.js

Attaching the Controller

D Directive Controller name Alias

```
<body>
  <div ng-controller="StoreController as store">
    <h1> Product Name </h1>
    <h2> $Product Price </h2>
    <p> Product Description </p>
  </div>
  <script type="text/javascript" src="angular.min.js"></script>
  <script type="text/javascript" src="app.js"></script>
</body>
```

index.html

```
(function(){
  var app = angular.module('store', [ ]);

  app.controller('StoreController', function(){
    this.product = gem;
  });
  . . .
})();
```

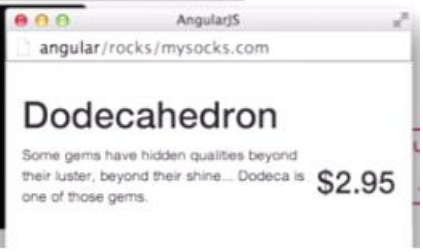
app.js

Displaying Our First Product

```
<body>
  <div ng-controller="StoreController as store">
     <h1> {{store.product.name}} </h1>
     <h2> ${{store.product.price}} </h2>
     <p> {{store.product.description}} </p>
  </div>
  <script type="text/javascript" src="angular.min.js"></script>
  <script type="text/javascript" src="app.js"></script>
</body>
```

index.html

```
(function(){
  var app = angular.module('store', [ ]);

  app.controller('StoreController', function(){
    this.product = gem;
  });
  . . .
})();
```



Scope

Understanding Scope

```
<body>
  <div ng-controller="StoreController as store">
     <h1> {{store.product.name}} </h1>
     <h2> ${{store.product.price}} </h2>
     <p> {{store.product.description}} </p>
  </div>
  {{store.product.name}}
  <script type="text/javascript" src="angular.min.js"></script>
  <script type="text/javascript" src="app.js"></script>
</body>
```

The scope of the Controller is only inside here...

Would never print a value!

Built-in Directives

Adding A Button

```
<body ng-controller="StoreController as store">
  <div>
    <h1> {{store.product.name}} </h1>
    <h2> ${{store.product.price}} </h2>
    <p> {{store.product.description}} </p>
    <button> Add to Cart </button>
  </div>
  <script type="text/javascript" src="angular.min.js"></script>
  <script type="text/javascript" src="app.js"></script>
</body>
```

Directives to the rescue!

How can we only show this button...

index.html

```
var gem = {
  name: 'Dodecahedron',
  price: 2.95,
  description: '. . .',
  canPurchase: false
}
```

...when this is true?

SHAPING UP WITH ANGULAR.JS

NgShow Directive

```
<body ng-controller="StoreController as store">
  <div>
    <h1> {{store.product.name}} </h1>
    <h2> ${{store.product.price}} </h2>
    <p> {{store.product.description}} </p>
    <button ng-show="store.product.canPurchase"> Add to Cart </button>
  </div>
  <script type="text/javascript" src="angular.min.js"></script>
  <script type="text/javascript" src="app.js"></script>
</body>
```

index.html

```
var gem = {
  name: 'Dodecahedron',
  price: 2.95,
  description: '. . .',
  canPurchase: false
}
```

Will only show the element if the value of the Expression is **true**.

SHAPING UP WITH ANGULAR.JS

NgHide Directive

```
<body ng-controller="StoreController as store">
  <div ng-show="!store.product.soldOut">
    <h1> {{store.product.name}} </h1>
    <h2> ${{store.product.price}} </h2>
    <p> {{store.product.description}} </p>
    <button ng-show="store.product.canPurchase"> Add to Cart </button>
  </div>
  <script type="text/javascript" src="angular.min.js"></script>
  <script type="text/javascript" src="app.js"></script>
</body>
```

This is awkward and a good example to use ng-hide!

index.html

```
var gem = {
  name: 'Dodecahedron',
  price: 2.95,
  description: '. . .',
  canPurchase: true,
  soldOut: true,
}
```

If the product is sold out we want to hide it.

NgHide Directive

```
<body ng-controller="StoreController as store">
  <div ng-hide="store.product.soldOut">
    <h1> {{store.product.name}} </h1>
    <h2> ${{store.product.price}} </h2>
    <p> {{store.product.description}} </p>
    <button ng-show="store.product.canPurchase"> Add to Cart </button>
  </div>
  <script type="text/javascript" src="angular.min.js"></script>
  <script type="text/javascript" src="app.js"></script>
</body>
```

Much better!

index.html

```
var gem = {
  name: 'Dodecahedron',
  price: 2.95,
  description: '. . .',
  canPurchase: true,
  soldOut: true,
}
```

If the product is sold out we want to hide it.

Multiple Products

```
app.controller('StoreController', function(){
  this.products = gems;
});
```

So we have multiple products...

```
var gems = [
  {
    name: "Dodecahedron",
    price: 2.95,
    description: ". . .",
    canPurchase: true,
  },
  {
    name: "Pentagonal Gem",
    price: 5.95,
    description: ". . .",
    canPurchase: false,
  }…
];
```

Now we have an array...

Maybe a Directive?

How might we display all these products in our template?

app.js

Working with An Array

```
<body ng-controller="StoreController as store">
  <div>
    <h1> {{store.products[0].name}} </h1>
    <h2> ${{store.products[0].price}} </h2>
    <p> {{store.products[0].description}} </p>
    <button ng-show="store.products[0].canPurchase">
      Add to Cart</button>
  </div>
  <div>
    <h1> {{store.products[1].name}} </h1>
    <h2> ${{store.products[1].price}} </h2>
    <p> {{store.products[1].description}} </p>
    <button ng-show="store.products[1].canPurchase">
      Add to Cart</button>
  </div>
  . . .
</body>
```

Why repeat yourself?

Why repeat yourself?

That works...

Why... You get it.

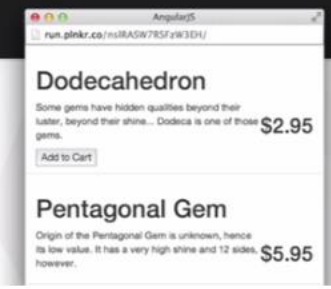
index.html

Working with An Array

```
<body ng-controller="StoreController as store">
  <div ng-repeat="product in store.products">
    <h1> {{product.name}} </h1>
    <h2> ${{product.price}} </h2>
    <p> {{product.description}} </p>
    <button ng-show="product.canPurchase">
      Add to Cart</button>
  </div>
  . . .
</body>
```

Repeat this section for each product.

index.html



What We Have Learned So Far

- **D** Directives – HTML annotations that trigger Javascript behaviors
- **M** Modules – Where our application components live
- **C** Controllers – Where we add application behavior
- **E** Expressions – How values get displayed within the page

Directives We Know & Love

ng-app – attach the Application Module to the page

```
<html ng-app="store">
```

ng-controller – attach a Controller function to the page

```
<body ng-controller="StoreController as store">
```

ng-show / ng-hide – display a section based on an Expression

```
<h1 ng-show="name"> Hello, {{name}}! </h1>
```

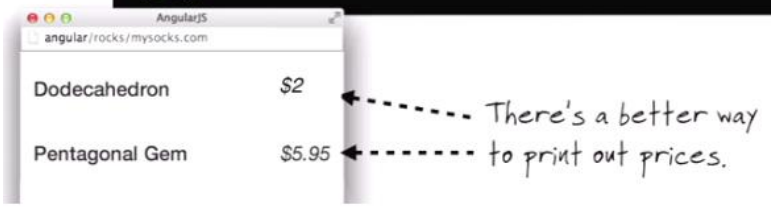
ng-repeat – repeat a section for each item in an Array

```
<li ng-repeat="product in store.products"> {{product.name}} </li>
```

Our Current Code

```
<body ng-controller="StoreController as store">
  <ul class="list-group">
    <li class="list-group-item" ng-repeat="product in store.products">
      <h3>
        {{product.name}}
        <em class="pull-right">${{product.price}}</em>
      </h3>
    </li>
  </ul>
</body>
```

index.html

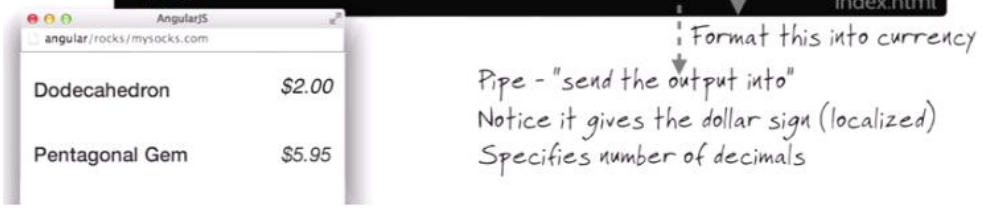


Filters

Our First Filter

```
<body ng-controller="StoreController as store">
  <ul class="list-group">
    <li class="list-group-item" ng-repeat="product in store.products">
      <h3>
        {{product.name}}
        <em class="pull-right">{{product.price | currency }}</em>
      </h3>
    </li>
  </ul>
</body>
```

index.html



Formatting with Filters

*Our Recipe `{{ data* | filter:options* }}`

date

`{{'1388123412323' | date:'MM/dd/yyyy @ h:mma'}}` 12/27/2013 @ 12:50AM

uppercase & lowercase

`{{'octagon gem' | uppercase}}` OCTAGON GEM

limitTo

`{{'My Description' | limitTo:8}}` My Descr

```
<li ng-repeat="product in store.products | limitTo:3">
```

Will list products by descending price.

orderBy

```
<li ng-repeat="product in store.products | orderBy:'-price'">
```

Without the - products would list in ascending order.

Adding an Image Array to our Product Array

```
var gems = [
  { name: 'Dodecahedron Gem',
    price: 2.95,
    description: '. . .',
    images: [
      {
        full: 'dodecahedron-01-full.jpg',
        thumb: 'dodecahedron-01-thumb.jpg'
      },
      {
        full: "dodecahedron-02-full.jpg",
        . . .
```

Our New Array

Image Object

app.js

To display the first image in a product: `{{product.images[0].full}}`

Using ng-src for Images

Using Angular Expressions inside a **src** attribute causes an error!

```
<img src="{{product.images[0].full}}"/>
```

...the browser tries to load the image *before* the Expression evaluates.

```
<body ng-controller="StoreController as store">
  <ul class="list-group">
    <li class="list-group-item" ng-repeat="product in store.products">
      <h3>
        {{product.name}}
        <em class="pull-right">{{product.price | currency}}</em>
        <img ng-src="{{product.images[0].full}}"/>
      </h3>
    </li>
  </ul>
</body>
```

NG-SOURCE to the rescue!

index.html

A Simple Set of Tabs

```
<section>
  <ul class="nav nav-pills">
    <li> <a href>Description</a> </li>
    <li> <a href>Specifications</a> </li>
    <li> <a href>Reviews</a> </li>
  </ul>
</section>
```

index.html

SHAPING UP WITH ANGULAR...

Introducing a new Directive!

```
<section>
  <ul class="nav nav-pills">
    <li> <a href ng-click="tab = 1">Description</a> </li>
    <li> <a href ng-click="tab = 2">Specifications</a> </li>
    <li> <a href ng-click="tab = 3">Reviews</a> </li>
  </ul>
  {{tab}}
</section>
```

index.html

Assigning a value to tab.

For now just print this value to the screen.

SHAPING UP

Introducing a new Directive!

Whoa, it's dynamic and stuff...

When `ng-click` changes the value of `tab` ...

... the `{{tab}}` expression automatically gets updated!

Expressions define a 2-way Data Binding ...

this means Expressions are re-evaluated when a property changes.

Let's add the tab content panels

```
<------------ tabs are up here...
. . .
<div class="panel">
  <h4>Description</h4>
  <p>{{product.description}}</p>
</div>
<div class="panel">
  <h4>Specifications</h4>
  <blockquote>None yet</blockquote>
</div>
<div class="panel">
  <h4>Reviews</h4>
  <blockquote>None yet</blockquote>
</div>
```

How do we make the tabs trigger the panel to show?

Let's add the tab content panels

```
<div class="panel" ng-show="tab === 1">
  <h4>Description</h4>
  <p>{{product.description}}</p>
</div>
<div class="panel" ng-show="tab === 2">
  <h4>Specifications</h4>
  <blockquote>None yet</blockquote>
</div>
<div class="panel" ng-show="tab === 3">
  <h4>Reviews</h4>
  <blockquote>None yet</blockquote>
</div>
```

show the panel if tab is the right number

Now when a tab is selected it will show the appropriate panel!

Setting the Initial Value

`ng-init` allows us to evaluate an expression in the current scope.

```
<section ng-init="tab = 1">
  <ul class="nav nav-pills">
    <li> <a href ng-click="tab = 1">Description</a> </li>
    <li> <a href ng-click="tab = 2">Specifications</a> </li>
    <li> <a href ng-click="tab = 3">Reviews</a> </li>
  </ul>
  . . .
```

index.html

The ng-class directive

```
<section ng-init="tab = 1">
  <ul class="nav nav-pills">
    <li ng-class="{ active:tab === 1 }">
      <a href ng-click="tab = 1">Description</a>
    </li>
    <li ng-class="{ active:tab === 2 }">
      <a href ng-click="tab = 2">Specifications</a>
    </li>
    <li ng-class="{ active:tab === 3 }">
      <a href ng-click="tab = 3">Reviews</a>
    </li>
  </ul>
  . . .
```

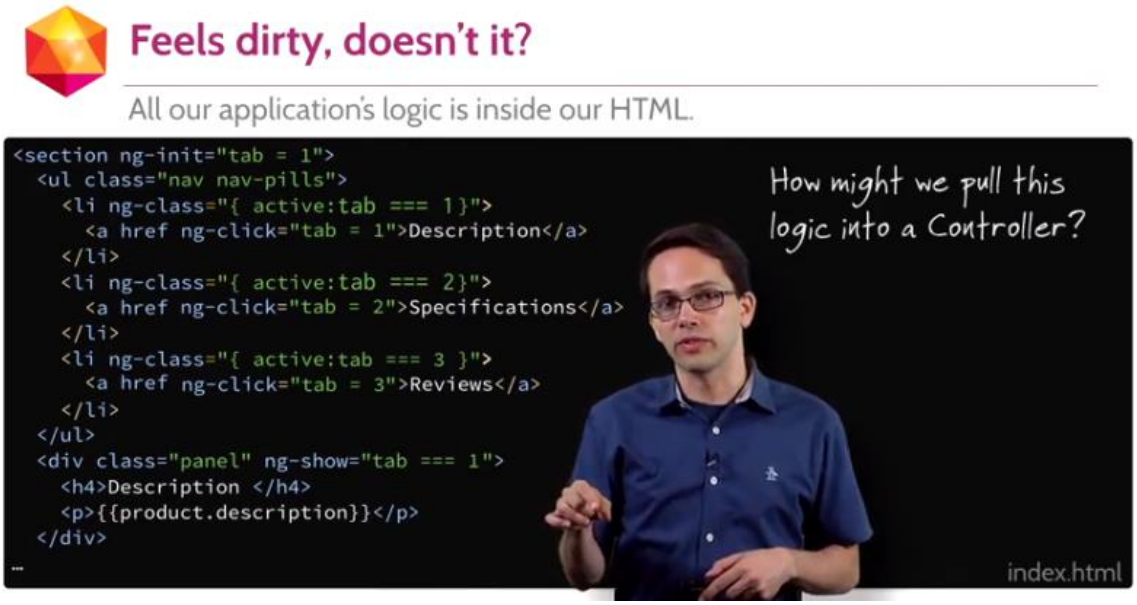
index.html

Expression to evaluate
If true, set class to "active", otherwise nothing.

Name of the class to set.

SHAPING UP WITH ANGULAR.JS

Decoupling

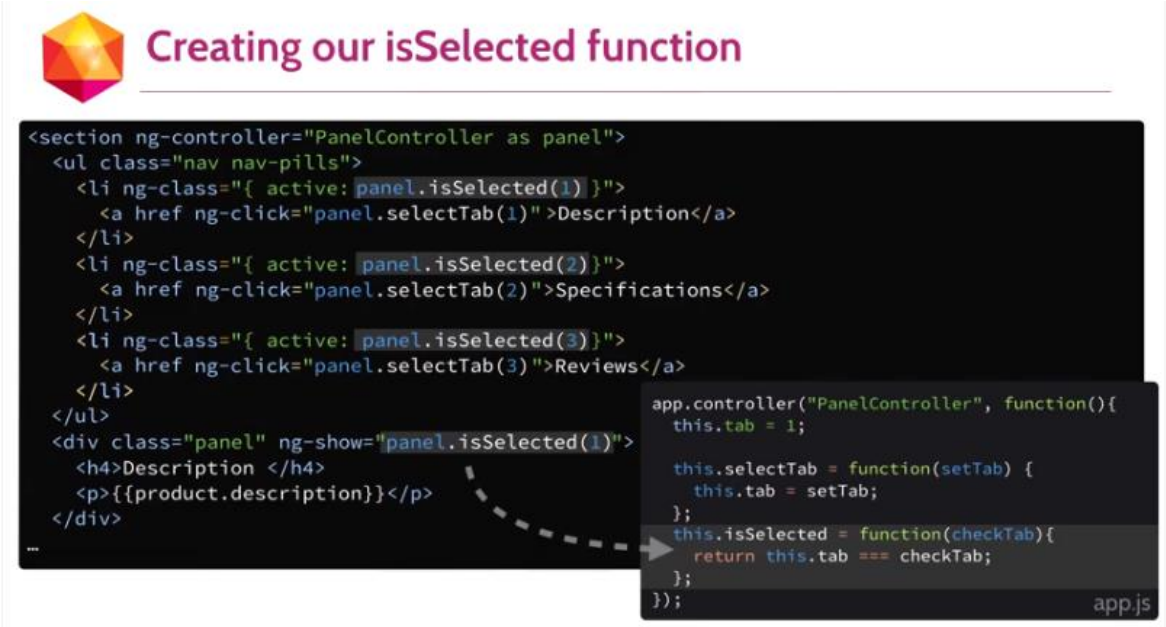

Feels dirty, doesn't it?
All our application's logic is inside our HTML.
<section ng-init="tab = 1">
<ul class="nav nav-pills">
<li ng-class="{ active:tab === 1}">
<a href ng-click="tab = 1">Description

<li ng-class="{ active:tab === 2}">
<a href ng-click="tab = 2">Specifications

<li ng-class="{ active:tab === 3 }">
<a href ng-click="tab = 3">Reviews

<h4>Description </h4>
{{product.description}}

How might we pull this logic into a Controller?
index.html


Creating our isSelected function
<section ng-controller="PanelController as panel">
<ul class="nav nav-pills">
<li ng-class="{ active: panel.isSelected(1) }">
<a href ng-click="panel.selectTab(1)">Description

<li ng-class="{ active: panel.isSelected(2)}">
<a href ng-click="panel.selectTab(2)">Specifications

<li ng-class="{ active: panel.isSelected(3)}">
<a href ng-click="panel.selectTab(3)">Reviews

<h4>Description </h4>
{{product.description}}

app.controller("PanelController", function(){
this.tab = 1;
this.selectTab = function(setTab) {
this.tab = setTab;
};
this.isSelected = function(checkTab){
return this.tab === checkTab;
};
});
app.js

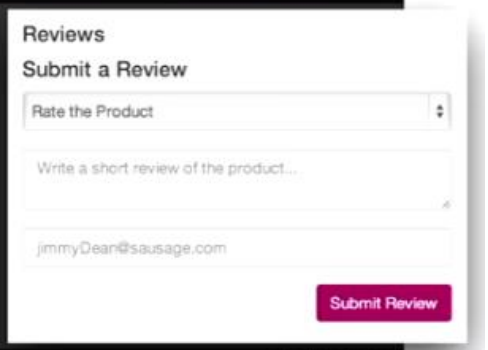
Looping Over Reviews in our Tab

```
<li class="list-group-item" ng-repeat="product in store.products">
  . . .
  <div class="panel" ng-show="panel.isSelected(3)">
    <h4> Reviews </h4>

    <blockquote ng-repeat="review in product.reviews">
      <b>Stars: {{review.stars}}</b>
      {{review.body}}
      <cite>by: {{review.author}}</cite>
    </blockquote>
  </div>
```

index.html

Writing out our Review Form

```
<h4> Reviews </h4>

<blockquote ng-repeat="review in product.reviews">...</blockquote>

<form name="reviewForm">
  <select>
    <option value="1">1 star</option>
    <option value="2">2 stars</option>
    . . .
  </select>
  <textarea></textarea>
  <label>by:</label>
  <input type="email" />
  <input type="submit" value="Submit" />
</form>
```



index.html

With Live Preview

```
<form name="reviewForm">
  <blockquote>
    <b>Stars: {{review.stars}}</b>
    {{review.body}}
    <cite>by: {{review.author}}</cite>
  </blockquote>
  <select>
    <option value="1">1 star</option>
    <option value="2">2 stars</option>
    . . .
  </select>
  <textarea></textarea>
  <label>by:</label>
  <input type="email" />
  <input type="submit" value="Submit" />
</form>
```

How do we bind this review object to the form?

index.html

Introducing ng-model

```
<form name="reviewForm">
  <blockquote>
    <b>Stars: {{review.stars}}</b>
    {{review.body}}
    <cite>by: {{review.author}}</cite>
  </blockquote>
  <select ng-model="review.stars">
    <option value="1">1 star</option>
    <option value="2">2 stars</option>
    . . .
  </select>
  <textarea ng-model="review.body"></textarea>
  <label>by:</label>
  <input ng-model="review.author" type="email" />
  <input type="submit" value="Submit" />
</form>
```

index.html

ng-model binds the form element value to the property

Two More Binding Examples

With a Checkbox

```
<input ng-model="review.terms" type="checkbox" /> I agree to the terms
```

Sets value to true or false

With Radio Buttons

```
What color would you like?

<input ng-model="review.color" type="radio" value="red" /> Red
<input ng-model="review.color" type="radio" value="blue" /> Blue
<input ng-model="review.color" type="radio" value="green" /> Green
```

Sets the proper value based on which is selected

We need to Initialize the Review

```
<form name="reviewForm" >
  <blockquote>
  <b>Stars: {{review.stars}}</b>
  {{review.body}}
  <cite>by: {{review.author}}</cite>
  </blockquote>

  <select ng-model="review.stars">
    <option value="1">1 star</option>
    <option value="2">2 stars</option>
    . . .
  </select>
  <textarea ng-model="review.body"></textarea>
  <label>by:</label>
  <input ng-model="review.author" type="email" />
  <input type="submit" value="Submit" />
</form>
```

index.html

We could do ng-init, but we're better off creating a controller.

Using the reviewCtrl.review

```
app.controller("ReviewController", function(){
  this.review = {};
});
```
app.js

```
<form name="reviewForm" ng-controller="ReviewController as reviewCtrl">
  <blockquote>
  <b>Stars: {{reviewCtrl.review.stars}}</b>
  {{reviewCtrl.review.body}}
  <cite>by: {{reviewCtrl.review.author}}</cite>
  </blockquote>

<select ng-model="reviewCtrl.review.stars">
  <option value="1">1 star</option>
  <option value="2">2 stars</option>
  . . .
</select>
<textarea ng-model="reviewCtrl.review.body"></textarea>
```
index.html

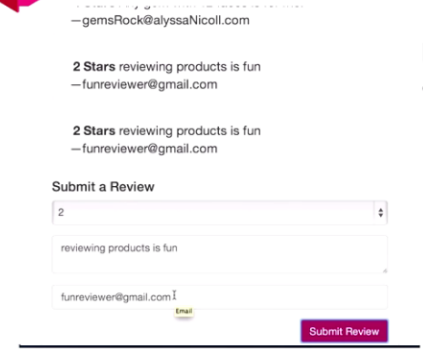
Using ng-submit to make the Form Work

```
app.controller("ReviewController", function(){
  this.review = {};

  this.addReview = function(product) {
    product.reviews.push(this.review);
  };
});
```
Push review onto this product's reviews array.

app.js

```
<form name="reviewForm" ng-controller="ReviewController as reviewCtrl"
                        ng-submit="reviewCtrl.addReview(product)"
  <blockquote>
  <b>Stars: {{reviewCtrl.review.stars}}</b>
  {{reviewCtrl.review.body}}
  <cite>by: {{reviewCtrl.review.author}}</cite>
  </blockquote>
```
index.html

Now with Reviews!



Review gets added, but the form still has all previous values!

Resetting the Form on Submit

```
app.controller("ReviewController", function(){
  this.review = {};

  this.addReview = function(product) {
    product.reviews.push(this.review);
    this.review = {};
  };
});
```
Clear out the review, so the form will reset.

app.js

```
<form name="reviewForm" ng-controller="ReviewController as reviewCtrl"
                        ng-submit="reviewCtrl.addReview(product)">
  <blockquote>
  <b>Stars: {{reviewCtrl.review.stars}}</b>
  {{reviewCtrl.review.body}}
  <cite>by: {{reviewCtrl.review.author}}</cite>
</blockquote>
```
index.html

Now with validation

Turn Off Default HTML Validation

```
<form name="reviewForm" ng-controller="ReviewController as reviewCtrl"
                        ng-submit="reviewCtrl.addReview(product)" novalidate>
  <select ng-model="reviewCtrl.review.stars" required>
    <option value="1">1 star</option>
    ...
  </select>

  <textarea name="body" ng-model="reviewCtrl.review.body" required></textarea>
  <label>by:</label>
  <input name="author" ng-model="reviewCtrl.review.author" type="email" required/>

  <div> reviewForm is {{reviewForm.$valid}} </div>
  <input type="submit" value="Submit" />
</form>
```

Mark Required Fields

Print Forms Validity

index.html

Preventing the Submit

```
<form name="reviewForm" ng-controller="ReviewController as reviewCtrl"
                        ng-submit="reviewCtrl.addReview(product)" novalidate>
```

index.html

We only want this method to be called if reviewForm.$valid is true.

Preventing the Submit

```
<form name="reviewForm" ng-controller="ReviewController as reviewCtrl"
    ng-submit="reviewForm.$valid && reviewCtrl.addReview(product)" novalidate>
```

index.html

If valid is false, then addReview is never called.

The Input Classes

index.html

```
<input name="author" ng-model="reviewCtrl.review.author" type="email" required />
```

Source before typing email

```
<input name="author" . . . class="ng-pristine ng-invalid">
```

Source after typing, with invalid email

```
<input name="author". . . class="ng-dirty ng-invalid">
```

Source after typing, with valid email

```
<input name="author" . . . class="ng-dirty ng-valid">
```

So, lets highlight the form field using classes after we start typing, `ng-dirty` showing if a field is valid or invalid. `ng-valid` `ng-invalid`

The classes

```
<input name="author" ng-model="reviewCtrl.review.author" type="email" required />
```
index.html

```
.ng-invalid.ng-dirty {
  border-color: #FA787E;
}

.ng-valid.ng-dirty {
  border-color: #78FA89;
}
```
Red border for invalid

Green border for valid

style.css

HTML5-based type validations

Web forms usually have rules around valid input:

- Angular JS has built-in validations for common input types:

```
<input type="email" name="email">
```

```
<input type="url" name="homepage">
```

```
<input type="number" name="quantity">
```

Can also define min & max with numbers

```
min=1 max=10
```

Decluttering our Code

```
<ul class="list-group">
  <li class="list-group-item" ng-repeat="product in store.products">
    <h3>
      {{product.name}}
      <em class="pull-right">${{product.price}}</em>
    </h3>
    <section ng-controller="PanelController as panel">
    . . .
```
index.html

We're going to have multiple pages that want to reuse this HTML snippet.

How do we eliminate this duplication?

SHAPING U

Using ng-include for Templates

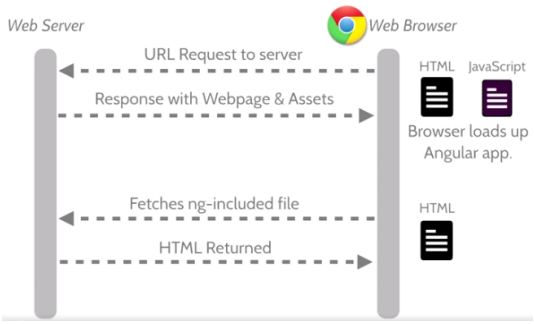
```
<ul class="list-group">
  <li class="list-group-item" ng-repeat="product in store.products">
    <h3 ng-include="'product-title.html'">
    </h3>
    <section ng-controller="PanelController as panel">
```
index.html

--->name of file to include

`ng-include` is expecting a variable with the name of the file to include.
To pass the name directly as a string, use single quotes (`'...'`)

```
{{product.name}}
<em class="pull-right">${{product.price}}</em>
```
product-title.html

```
<h3 ng-include="'product-title.html'" class="ng-scope">
  <span class="ng-scope ng-binding">Awesome Multi-touch Keyboard</span>
  <em class="pull-right ng-scope ng-binding">$250.00</em>
</h3>
```
generated html



Creating our First Custom Directive

Using ng-include...

```
<h3 ng-include="'product-title.html'"></h3>
```
index.html

Custom Directive

```
<product-title></product-title>
```
index.html

Our old code and our custom Directive will do the same thing... with some additional code.

Why Directives?

Directives allow you to write HTML that expresses the behavior of your application.

```
<aside class="col-sm-3">
  <book-cover></book-cover>
  <h4><book-rating></book-rating></h4>
</aside>

<div class="col-sm-9">
  <h3><book-title></book-title></h3>

  <book-authors></book-authors>

  <book-review-text></book-review-text>

  <book-genres></book-genres>
</div>
```

Writing Custom Directives

Template-expanding Directives are the simplest:

- define a custom tag or attribute that is expanded or replaced
- can include Controller logic, if needed

Directives can also be used for:

- Expressing complex UI
- Calling events and registering event handlers
- Reusing common components

How to Build Custom Directives

```
<product-title></product-title>
```
index.html

```
app.directive('productTitle', function(){
  return {
    A configuration object defining how your directive will work
  };
});
```
app.js

How to Build Custom Directives

```
<product-title></product-title>
```
index.html

dash in HTML translates to ... **camelCase** in JavaScript

```
app.directive('productTitle', function(){
  return {
    restrict: 'E',
    templateUrl: 'product-title.html'
  };
});
```
Type of Directive (E for Element)

Url of a template

generates into app.js

```
<h3>
  {{product.name}}
  <em class="pull-right">$250.00</em>
</h3>
```
index.html

Attribute vs Element Directives

Element Directive

```
<product-title></product-title>
```
index.html

Notice we're not using a self-closing tag... `<product-title/>`

...some browsers don't like self-closing tags.

Attribute Directive

```
<h3 product-title></h3>
```
index.html

Use Element Directives for UI widgets and Attribute Directives for mixin behaviors... like a tooltip.

Defining an Attribute Directive

```
<h3 product-title></h3>
```
index.html

```
app.directive('productTitle', function(){
  return {
    restrict: 'A',
    templateUrl: 'product-title.html'
  };
});
```
Type of Directive (A for Attribute)

generates into app.js

```
<h3>
  {{product.name}}
  <em class="pull-right">$250.00</em>
</h3>
```
index.html

Directives allow you to write better HTML

When you think of a dynamic web application, do you think you'll be able to understand the functionality just by looking at the HTML?

No, right?

When you're writing an Angular JS application, you should be able to understand the behavior and intent from just the HTML.

And you're likely using custom directives to write expressive HTML.

SHA

Reviewing our Directive

Template-Expanding Directives

```
<h3>
  {{product.name}}
  <em class="pull-right">${{product.price}}</em>
</h3>
```

index.html

An Attribute Directive

```
<h3 product-title></h3>
```

An Element Directive

```
<h3> <product-title></product-title> </h3>
```

What if we *need* a Controller?

```
<section ng-controller="PanelController as panels">
 <ul class="nav nav-pills"> . . . </ul>
 <div class="panel" ng-show="panels.isSelected(1)"> . . . </div>
 <div class="panel" ng-show="panels.isSelected(2)"> . . . </div>
 <div class="panel" ng-show="panels.isSelected(3)"> . . . </div>
</section>
```

Directive?

index.html

First, extract the template...

```
<h3> <product-title> </h3>
<product-panels ng-controller="PanelController as panels">
  . . .
</product-panels>
```

index.html

```
<section>
<ul class="nav nav-pills"> . . . </ul>
<div class="panel" ng-show="panels.isSelected(1)"> . . . </div>
<div class="panel" ng-show="panels.isSelected(2)"> . . . </div>
<div class="panel" ng-show="panels.isSelected(3)"> . . . </div>
</section>
```

product-panels.html

Now write the Directive ...

```
<product-panels ng-controller="PanelController as panels">
  . . .
</product-panels>
```

index.html

```
app.directive('productPanels', function(){
  return {
    restrict: 'E',
    templateUrl: 'product-panels.html'
  };
});
```

app.js

What about the Controller?

```
<product-panels ng-controller="PanelController as panels">
  . . .
</product-panels>
```
index.html

```
app.directive('productPanels', function(){
  return {
    restrict: 'E',
    templateUrl: 'product-panels.html'
  };
});
app.controller('PanelController', function(){
  . . .
});
```
app.js

First we need to move the functionality inside the directive

Moving the Controller Inside

```
<product-panels  ng-controller="PanelController as panels" >
  . . .
</product-panels>
```
x.html

Next, move the alias inside

```
app.directive('productPanels', function(){
  return {
    restrict: 'E',
    templateUrl: 'product-panels.html',
    controller:function(){
      . . .
    }
  };
});
```

Need to Specify the Alias

```
<product-panels>
  . . .
</product-panels>
```
index.html

```
app.directive('productPanels', function(){
  return {
    restrict: 'E',
    templateUrl: 'product-panels.html',
    controller:function(){
      . . .
    },
    controllerAs: 'panels'
  };
});
```
Now it works, using panels as our Controller Alias.

app.js

Starting to get a bit cluttered?

```
(function(){
  var app = angular.module('store', []);

  app.controller('StoreController', function(){ . . . });

  app.directive('productTitle', function(){ . . . });

  app.directive('productGallery', function(){ . . . });

  app.directive('productPanels', function(){ . . . });
  . . .
})();
```

app.js

Can we refactor these out?

Make a *new* Module

```
(function(){
  var app = angular.module('store', []);

  app.controller('StoreController', function(){ . . . });
  . . .
})();
```

app.js

Define a new module just for Product stuff...

Module Name

Different closure means different app variable.

```
(function(){
  var app = angular.module('store-products', [ ]);
  app.directive('productTitle', function(){ . . . });

  app.directive('productGallery', function(){ . . . });

  app.directive('productPanels', function(){ . . . });
})();
```

products.js

Add it to the dependencies ...

store depends on store-products

```
(function(){
  var app = angular.module('store', ['store-products']);

  app.controller('StoreController', function(){ . . . });
  . . .
})();
```

app.js

Module Name

```
(function(){
  var app = angular.module('store-products', [ ]);
  app.directive('productTitle', function(){ . . . });

  app.directive('productGallery', function(){ . . . });

  app.directive('productPanels', function(){ . . . });
})();
```

products.js

We'll also need to include the file

```
<!DOCTYPE html>
<html ng-app="store">
  <head> . . . </head>
  <body ng-controller="StoreController as store">
    . . .
    <script src="angular.js"></script>
    <script src="app.js"></script>
    <script src="products.js"></script>
  </body>
</html>
```

index.html

How should I organize my application Modules?

Best to split Modules around functionality:

- `app.js` – top-level module attached via `ng-app`
- `products.js` – all the functionality for products and only products

Does this feel strange?

```
(function(){
  var app = angular.module('store', [ 'store-products' ]);

  app.controller('StoreController', function(){
    this.products = [
      { name: '. . .', price: 1.99, . . . },
      { name: '. . .', price: 1.99, . . . },
      { name: '. . .', price: 1.99, . . . },
      . . .
    ];
    . . .
  });
})();
```

What is all this data doing here?

app.js

Where can we put it?
Shouldn't we fetch this from an API?

How do we get that data?

```
(function(){
  var app = angular.module('store', [ 'store-products' ]);

  app.controller('StoreController', function(){
    this.products = ???;
    . . .
  });
})();
```

How do we fetch our products from an API?

app.js

http://api.example.com/products.json

```
[
  { name: '. . .', price: 1.99, . . . },
  { name: '. . .', price: 1.99, . . . },
  { name: '. . .', price: 1.99, . . . },
  . . .
]
```

We need a Service!

Services give your Controller additional functionality, like ...

- Fetching JSON data from a web service with `$http`
- Logging messages to the JavaScript console with `$log`
- Filtering an array with `$filter`

All built-in Services start with a $ sign ...

Introducing the $http Service!

The $http Service is how we make an async request to a server ...

- By using $http as a function with an options object:

```
$http({ method: 'GET', url: '/products.json' });
```

- Or using one of the shortcut methods:

```
$http.get('/products.json', { apiKey: 'myApiKey' });
```

- Both return a Promise object with .success() and .error()
- If we tell $http to fetch JSON, the result will be automatically decoded into JavaScript objects and arrays

How does a Controller use a Service like $http?

Use this funky array syntax:

```
app.controller('SomeController', [ '$http', function($http){
} ]);
```

Service name

Service name as an argument

Dependency Injection!

```
app.controller('SomeController', [ '$http', '$log', function($http, $log){
} ]);
```

If you needed more than one

When Angular is Loaded Services are Registered

```
app.controller('SomeController', [ '$http', '$log', function($http, $log){
} ]);
```

A Controller is Initialized

```
app.controller('SomeController', [ '$http', '$log', function($http, $log){
} ]);
```

Then When the Controller runs ...

```
app.controller('SomeController', [ '$http', '$log', function($http, $log){
} ]);
```

Time for your injection!

```
(function(){
  var app = angular.module('store', [ 'store-products' ]);

  app.controller('StoreController', [ '$http',function($http){
    this.products = ???;
  }]);
})();
```

StoreController needs the $http Service...

...so $http gets injected as an argument!

app.js

Now what?

Let's use our Service!

```
(function(){
  var app = angular.module('store', [ 'store-products' ]);

  app.controller('StoreController', [ '$http',function($http){
    this.products = ???;

    $http.get('/products.json')
  }]);
})();
```

Fetch the contents of products.json...

app.js

Our Service will Return Data

```
(function(){
  var app = angular.module('store', [ 'store-products' ]);

  app.controller('StoreController', [ '$http',function($http){
    this.products = ???;

    $http.get('/products.json').success(function(data){
      ??? = data;
    });
  }]);
})();
```

What do we assign data to, though...?

$http returns a Promise, so success() gets the data...

app.js

Storing the Data for use in our Page

```
(function(){
  var app = angular.module('store', [ 'store-products' ]);

  app.controller('StoreController', [ '$http',function($http){
    var store = this;

    $http.get('/products.json').success(function(data){
      store.products = data;
    });
  }]);
})();
```

We need to store what this is ...

... and now we have somewhere to put our data!

app.js

But the page might look funny until the data loads.

Initialize Products to be a Blank Array

```
(function(){
  var app = angular.module('store', [ 'store-products' ]);

  app.controller('StoreController', [ '$http',function($http){
    var store = this;
    store.products = [ ];

    $http.get('/products.json').success(function(data){
      store.products = data;
    });
  }]);
})();
```

We need to initialize products to an empty array, since the page will render before our data returns from our get request.

app.js

Additional $http functionality

In addition to `get()` requests, `$http` can `post()`, `put()`, `delete()`...

```
$http.post('/path/to/resource.json', { param: 'value' });
```

```
$http.delete('/path/to/resource.json');
```

...or any other HTTP method by using a `config` object:

```
$http({ method: 'OPTIONS', url: '/path/to/resource.json' });
```

```
$http({ method: 'PATCH', url: '/path/to/resource.json' });
```

```
$http({ method: 'TRACE', url: '/path/to/resource.json' });
```

Laravel

Composer

Composer is a dependency manager which allows us to reuse any kind of code. Instead of reinventing the wheel, we can instead download popular packages. In fact, Laravel uses dozens of said packages.

Angular / Slim API

It’s an Immediately-Invoked Function Expression, or IIFE for short. It executes immediately after it’s created.

It has nothing to do with any event-handler for any events (such as document.onload).
The first pair of parentheses (function(){...}) turns the code within (in this case, a function) into an expression, and the second pair of parentheses (function(){...})() calls the function that results from that evaluated expression.

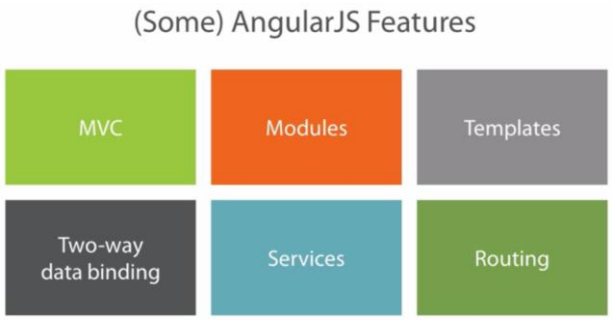
This pattern is often used when trying to avoid polluting the global namespace, because all the variables used inside the IIFE (like in any other normal function) are not visible outside its scope.
This is why, maybe, you confused this construction with an event-handler for window.onload, because it’s often used as this:

```
(function(){
    // all your code here
    var foo = function() {};
    window.onload = foo;
    // ...
})();
// foo is unreachable here (it’s undefined)
```

This is used in Angular.

You should, however, conform to modern practices in using angular 1.4. Notably, check out John Papa's Style Guide as a starting point for learning about that, and prefer using custom directives over ng-controller declarations, and using the Controller As syntax. Those two practices alone will prevent a lot of your headaches if you feel the need to port from 1.x to 2.0 when it comes out.

Building a Site in Angular and PHP



IIFE

Immediately Invoked Function Expression. The code runs instantly and all the variables live within this scope i.e. they are not global.

```
(function() {
        //Code goes here.
})();
```

Controller

JavaScript (app.js)

```
app.controller('CountryController', function () {
```

HTML

```
<div ng-controller="CountryController as countryCtrl">
```

Alias

You should look at **countryCtrl** as an instance of the **CountryController** class.

Databinding/Expressions

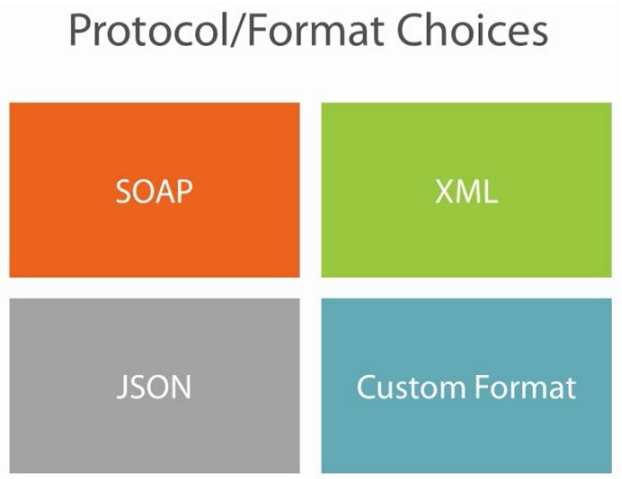
JavaScript (app.js)

```
this.countries =
    {
        name: 'Germany', code: 'de', states: [{ name: 'Bavaria' }, { name: 'Berlin' }]
    };
```

HTML

```
{{ countryCtrl.countries.name }}
```

We bind the **countryCtrl** **instance**, then to the **countries** **property**, and then to the **name** **sub-property**.



Creating JSON

```php
$data = array('one' => 1, 'two' => 2);
echo json_encode($data);
```

Parsing JSON

```php
$json = '{"one":1,"two":2}';
var_dump(json_decode($json));
```

Calling the service from Angular

HTTP Request

```
$http({
    method: 'GET', url: 'myService.php'
})
```

This works by using the “promise” concept which means that there is a window between making the request and getting the results that can be used to do something. A promise returns an object before returning the results. With this, we can use a method called .success which defines what to do if the request works.

Processing the Result

```
.success(function(data) {

})
```

Creating an Angular Service

Service Setup

```
app.factory('myService', function($http) {
    return { myMethod: function() {} }
});
```

Service Consumption

```
app.controller('MyController', function(myService) {
   myService.myMethod();
});
```

Two way data binding

Assigning a Model in Markup

```
<input type="text" name="state" ng-model="countryCtrl.newState">
<a href>Add state {{countryCtrl.newState}}</a>
```

Data Access in the Controller

```
this.newState = "";
```

Event Handling Directives

Custom Directives

Reusable HTML tags that reference some code as well as templates.

templateUrl	controller controllerAs	restrict

Defining a Directive

```
app.directive('stateView', function() {
    return {
        restrict: 'E',
        templateUrl: 'state-view.html',
        controller: function() {
        },
        controllerAs: 'stateCtrl'
    }
});
```

Using a Directive

```
<state-view></state-view>
```

Routing

Built-in Router	ui-router	Component Router (1.5+/2)
https://docs.angularjs.org/api/ngRoute	https://github.com/angular-ui/ui-router	https://github.com/angular/router

Using the Built-In Router

Configuring Routes

```
app.config(function($routeProvider) {
    $routeProvider.when('/states/:countryCode', {
        templateUrl: 'state-view.html',
        controller: function($routeParams) {
        },
        controllerAs: 'stateCtrl'
    })
});
```

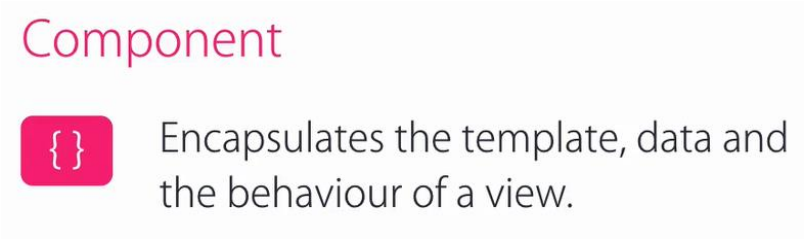
Link to Route

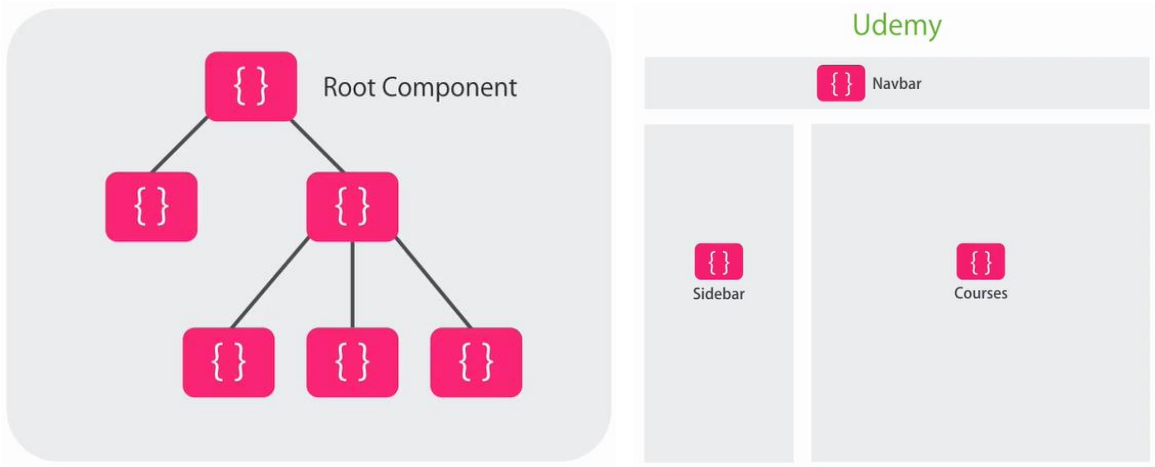
```
<a href ng-href="#/states/{{ c.code }}"></a>
```

Angular 2 – Mosh Hammedani

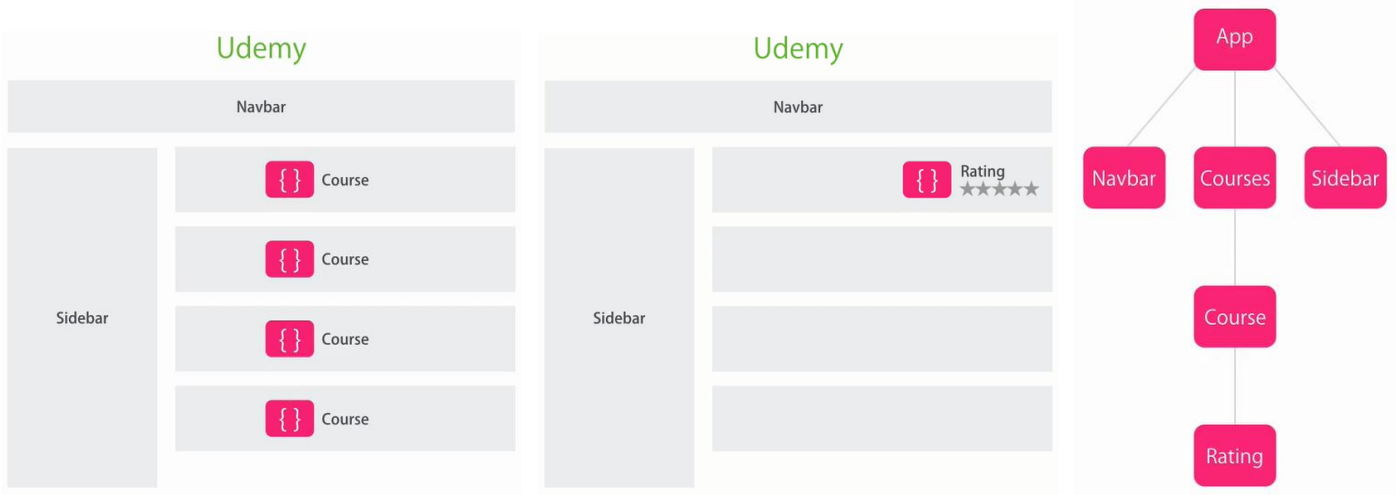
Architecture of Angular 2 Apps

Components





Each component can be comprised of multiple components and so on.



The benefit of this is that the application can be split into multiple smaller parts which can then be reused in many places, even in totally different projects.

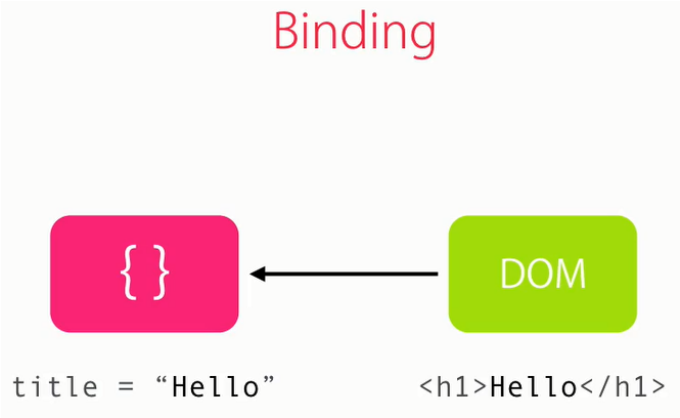
```
export class RatingComponent {

    averageRating: number;

    setRating(value) {
        …
    }
}
```

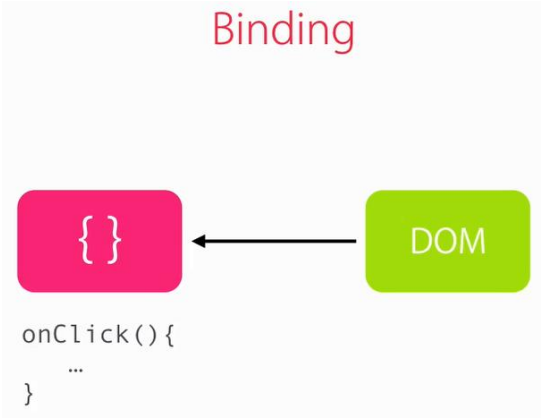
A component is nothing but a plain TypeScript class. The properties hold the data for the view whie the methods implement the behavior of the view; like what happens when a button is clicked.

The components are COMPLETELY decoupled from the DOM. Instead of modifying DOM elements with each change, binding is used. In the view, we bind to the methods and properties of our component. To handle an event raised from a DOM element by a click, we bind that element to a method in our component.

The reason for the decoupling is for making the components unit testable.

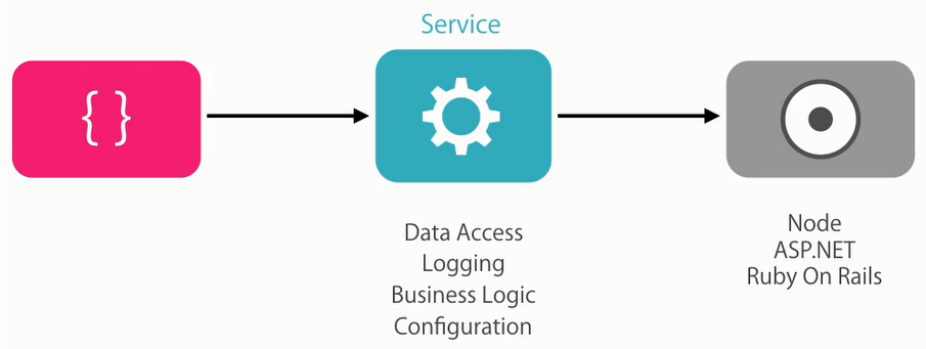
```
$("#title").text("Hello World");
```





Sometimes our components need to talk to back-end APIs to get or save data. To have good separation of concerns in our application, we delegate any logic that is not related to the view to a **service**.

A service is just a plain class that encapsulates any non UI logic, like making HTTP calls, logging, business roles etc.



Routers

As the user navigates from one page to another, it will figure out based on changes in the URL what component to present to the user.

Directives

Similar to components, we use directives to work with the DOM. The directive, unlike a component, doesn't have the template or HTML markup for a view. They are often used to add behavior to existing DOM elements.

Ex. We can use a directive to make a text-box automatically grow when it is focused. Angular has a bunch of built-in directives for commont tasks such as adding or removing DOM elements, repeating them. We can also create our own custom directives.

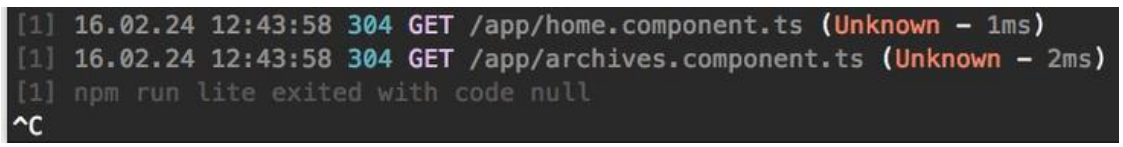
Node Install

Navigate to the application directory and install node (dependencies) by using **npm install**. In the project in package.json under scripts there are some custom node commands. The **start** command is a shortcut for **concurrently** running two commands.

- **npm run tsc:w** – Runs the TypeScript compiler into watch mode.
- **npm run lite** – Runs the lite webserver.

To **run the server**, **navigate to the directory where node was installed** (use **cd.. cd folder_name** and **dir**) and run the **npm start** command. After that, go to **localhost:3000** to view the website.

If the server terminates on its own, run this command **^C** and start it up again with **npm start**.



TypeScript

```
courses.component.ts app
1
2 export class CoursesComponent {
3
4 }
```

In TypeScript, each file is considered a module. In each module, we export one or more things, like a class, a function or a variable. The **export** keyword makes them available to other modules in the application. Later we can **import** said classes as needed. To make a class a component, we need to add a decorator.

Decorators / Annotations

```
app.component.ts app
1 import {Component} from 'angular2/core';
2
3 @Component({
4     selector: 'my-app',
5     template: '<h1>My First Angular 2 App</h1>'
6 })
7 export class AppComponent { }
```

This is nothing but a TypeScript class decorated with a component decorator. This decorator/annotation adds meta data to the class. All components are basically classes with such decorators. A decorator is the same as an attribute in C# and annotation in Java.

```
courses.component.ts app
1 import {Component} from 'angular2/core'
2
3 @Component({
4     selector: 'courses',
5     template: '<h2>Courses</h2>'
6 })
7 export class CoursesComponent {
8
9 }
```

The decorator “component” is a function that needs to be imported from the angular2

module. All decorators must be prefixed with an @ sign and then called. The function takes an object for which attributes are selected.

Selector: it specifies a CSS selector for a host HTML element. When angular sees an element that matches the selector, it will create an instance of the component in the element i.e. an element with the tag “courses”.

Template: It specifies the HTML that will be inserted into the DOM when the component’s view is rendered. We can write the template inline or put it in a separate file.

```
app.component.ts app
1 import {Component} from 'angular2/core';
2 import {CoursesComponent} from './courses.component'
3
4 @Component({
5     selector: 'my-app',
6     template: '<h1>My First Angular 2 App</h1><courses></courses>',
7     directives: [CoursesComponent]
8 })
9 export class AppComponent { }
```

Inside the directives array, we need to input any component or directives used in the template of the component. Ex. **<courses></courses>** is defined in **CoursesComponent**.

```
courses.component.ts app
1   import {Component} from 'angular2/core'
2
3   @Component({
4       selector: 'courses',
5       template: '<h2>Courses</h2>'
6   })
7
8   export class CoursesComponent {
9
10  }
```

A component encapsulates the data and logic behind a view. We can define properties in our component and display them in the template.

```
export class CoursesComponent {
    title: string = "The title of courses page";
}
```

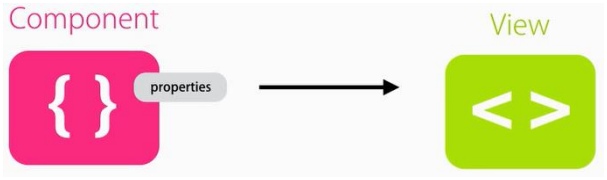
In TypeScript, unlike JavaScript, we can set the type of our variables. In some cases, we don't need to because it can be inferred from the value.

Interpolation

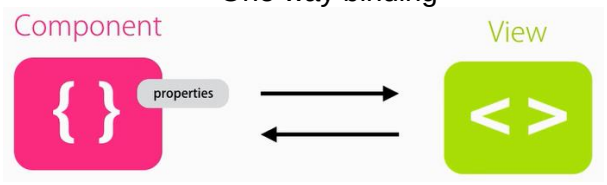
Rendering the value of a property by using {{ }}. If the value of the property in the component changes, the view will be automatically refreshed. This is called **one way binding**. There is also **two way binding**, which is used in forms i.e. when something is typed in an input form, the value is updated automatically.

courses.component.ts app

```
import {Component} from 'angular2/core'

@Component({
    selector: 'courses',
    template: `
        <h2>Courses</h2>
        {{ title }}
        `
})
export class CoursesComponent {
    title = "The title of courses page";
}
```



One way binding



Two way binding

Looping over a list

courses.component.ts app

```
import {Component} from 'angular2/core'

@Component({
    selector: 'courses',
    template: `
        <h2>Courses</h2>
        {{ title }}
        <ul>
            <li *ngFor="#course of courses">
            {{ course }}
            </li>
        </ul>
        `
})
export class CoursesComponent {
    title = "The title of courses page";
    courses = ["Course1", "Course2", "Course3"];
}
```

***ngFor="#instance of object"**

{{ instance }}

***ngFor** is a directive and it's similar to a **foreach** loop

Courses is the object we are iterating i.e. the property (list) in our component.

#course is a way to declare a local variable in our template.

Every iteration will hold one course at a time.

That course can be displayed by {{ course }}

Services

In a real application, the data comes from a server rather than a hardcoded list in the component. Any logic that is not about a view should be encapsulated in a separate class which we call a service. A service should be named like **serviceName.service.ts** and it is a simple class.

This service returns a list and it simulates a response from a RESTful API.

course.service.ts app

```
1
2 export class CourseService {
3     getCourses() : string[] {
4         return ["Course1", "Course2", "Course3"];
5     }
6 }
```

courses.component.ts app

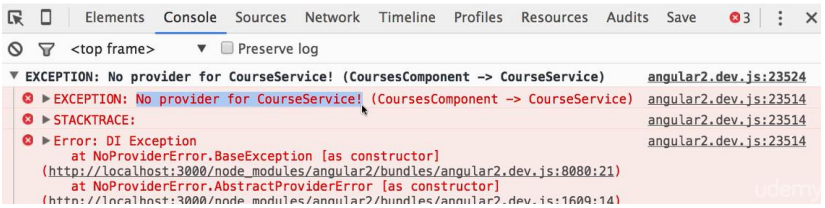
```
1 import {Component} from 'angular2/core'
2 import {CourseService} from './course.service'
```

```
4 @Component({
5     selector: 'courses',
6     template: `
7         <h2>Courses</h2>
8         {{ title }}
9         <ul>
10            <li *ngFor="#course of courses">
11            {{ course }}
12            </li>
13        </ul>
14        `,
15    providers: [CourseService]
16 })
```

```
17 export class CoursesComponent {
18     title = "The title of courses page";
19     courses;
20
21     constructor(courseService: CourseService){
22         this.courses = courseService.getCourses();
23     }
24 }
```

The service needs to be imported first. **./** means start from the current directory.

Providers: [CourseService] solves a common "**No provider for**" problem in Angular 2 apps.



This means that CoursesComponent has a dependency on CourseService, but Angular doesn't know how to create the service. "Providers" handles the dependency injection.

```
constructor(){
    new CourseService()
}
```

This causes tight coupling between CoursesComponent and CourseService. There are 2 problems with this.

If we create a custom constructor that takes 2 parameters, we would have to modify the instantiation of the service in every compontent that uses it.

Also, we won't be able to isolate the component and unit test it because we don't want a running server with a RESTful API. We want to use a mock service that simulates an API.

That's why we pass the service as a parameter in the constructor. **courseService** of type **CourseSevice** camelCase vs PascalCase

Directives

This directive enlarges an input form on focus, and reverts it back on blur (defocus).

We always start with exporting a class **AutoGrowDirective** and then we decorate it with a directive decorator **@Directive** after we import the component **import {Directive}** from Angular core.

auto-grow.directive.ts app

```
import {Directive, ElementRef, Renderer} from 'angular2/core'

@Directive({
    selector: '[autoGrow]',
    host: {
        '(focus)': 'onFocus()',
        '(blur)': 'onBlur()'
    }
})

export class AutoGrowDirective {

    constructor(private el: ElementRef, private renderer: Renderer){
    }

    onFocus(){
        this.renderer.setElementStyle(this.el, 'width', '200');
    }

    onBlur(){
        this.renderer.setElementStyle(this.el, 'width', '120');
    }
}
```

courses.component.ts app

```
import {Component} from 'angular2/core'
import {CourseService} from './course.service'
import {AutoGrowDirective} from './auto-grow.directive'

@Component({
    selector: 'courses',
    template: `
        <h2>Courses</h2>
        {{title}}
        <input type="text" autoGrow />
        <ul>
            <li *ngFor="#course of courses">
                {{ course }}
            </li>
        </ul>
        `,
        providers: [CourseService],
        directives: [AutoGrowDirective]
})

export class CoursesComponent {
    title = "The title of courses page";
    courses;

    constructor(courseService: CourseService){
        this.courses = courseService.getCourses();
    }
}
```

In the selector (CSS selector for the host element), square brackets are used for referring to an attribute. When Angular is scanning a template, if it finds an element that matches the CSS selector, it will apply the directive to it. Ex. The elemtn <input> has the attribute autoGrow in it.

```
host: {
    '(focus)': 'onFocus()',
    '(blur)': 'onBlur()'
}
```

We use host to subscribe to events raised from this element, in this case, focus and blur. We are binding the onFocus method in our Directve class to the focus event.

In the Directives methods, we need to access the host elements. To do that, we import 2 services from Angular; **ElementRef** for host **element access** and a service used to **modify the element** called **Renderer**.

Angular will automatcally inject instances of ElementRef and Renderer from the constructor into the Directive class i.e. dependency injection.

```
_el: ElementRef;

constructor(el: ElementRef, renderer: Renderer){
    this._el = el;
}
```

We then need the el and rendeder objects in the onFocus and onBlur methods.

One way is to declare two private fields in the class and initialize them in the constructor

Or, use this much simpler and cleaner way by using the **private** keyword.

```
constructor(private el: ElementRef, private renderer: Renderer){
}
```

In the component, where we use the directive, we need to declare the element in the template `<input type="text" autoGrow />`. But, nothing happens until we teach angular how to use it with `directives: [AutoGrowDirective]` after we import it with `import {AutoGrowDirective} from './auto-grow.directive'`

Property Binding

app.component.ts app

```
1  import {Component} from 'angular2/core';
2
3  @Component({
4      selector: 'my-app',
5      template: `
6          <h1>{{ title }}</h1>
7          <img src="{{ imageUrl }}" />
8      `
9  })
10 export class AppComponent {
11     title = "Angular App";
12     imageUrl = "http://lorempixel.com/400/200/"
13 }
```

Interpolation – The use of {{ }} to display properties of a component in the view. This is syntactical sugar.

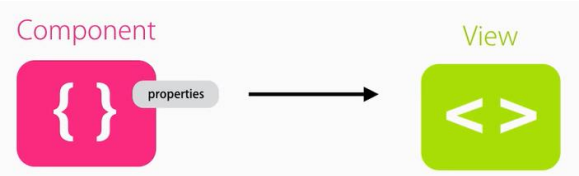
Behind the scene, when Angular compiles the template, the interpolations are translated into property bindings.

```
<img src="{{ imageUrl }}" />
<img [src]="imageUrl" />
<img bind-src="imageUrl" />
```

These are basically the same. For property binding, use [ ] brackets i.e. the second one.

```
<h1>{{ title }}</h1>
<h1 [textContent]="title"></h1>
```

Use interplation when inserting dynamic text between elements. textContent is a property of a DOM element. It's not an HTML attribute.



Property bindings work in one way. From component to DOM. If the property in the component changes, Angular will update the DOM. But any changes in the DOM are not reflected in the component.

Class Binding

Adds a CSS class based on a criterium. Ex.

```
<button class=”btn btn-primary”>Submit</button>

export class AppComponent {
        isActive = true;
}
```

```
<button class=”btn btn-primary”
[class.active]=”isActive”>Submit</button>

export class AppComponent {
        isActive = true;
}
```

Becomes

What this does is, if the condition in the component is true, the active class is added to the button, resulting with this in the view: <button class=”btn btn-primary active”>Submit</button>

favorite.component.ts app

```
import {Component} from 'angular2/core';

@Component({
    selector: 'favorite',
    template: `
    <i
        class="glyphicon"
        [class.glyphicon-star-empty]="!isFavori
        [class.glyphicon-star]="isFavorite"
        (click)="onClick()">
    </i>
    `
})
export class FavoriteComponent {
    isFavorite = false;

    onClick(){
        this.isFavorite = !this.isFavorite;
    }
}
```

This is the Chapter 3 exercise where a favorite star is to be made which can be toggled on and off.

The way this works is by decaring a base CSS class glyphicon on which we append the empty or full classes depending on the clicks/

When the start is clicked, the event calls the onClick() method which reverses the isFavorite boolean variable. Depending on this, the CSS class is appended to the base one.

Style Binding

This works the same way like class binding.

```
<button
        class=”btn btn-primary”
        [style.backgroundColor]=” isActive ? ‘blue’ : ‘gray’ ”>
                Submit
</button>

export class AppComponent {
        isActive = true;
}
```

The ternary operator is evaluated and it colors the button blue if true, and gray if false. Results in <button class=”btn btn-primary” style=”background-color: blue”>Submit</button>

Event Binding

We use it for handling events raised from the DOM, such as keystrokes, mouse movements, clicks etc. Similar to property binding, we have we have 2 syntaxes for event binding. We can use paranthesis or a prefix. Both of these are same.

```
<button (click)="onClick()">Submit</button>
<button on-click="onClick()">Submit</button>
```

We use [ ] for property binding, and ( ) for event binding.

app.component.ts app

```
import {Component} from 'angular2/core';

@Component({
    selector: 'my-app',
    template: `
        <button (click)="onClick()">Submit</button>
        <button on-click="onClick()">Submit</button>
    `
})
export class AppComponent {

    onClick(){
        console.log("Clicked");
    }
}
```

When we click the button, the onClick() method in the component is called, and the message is logged.

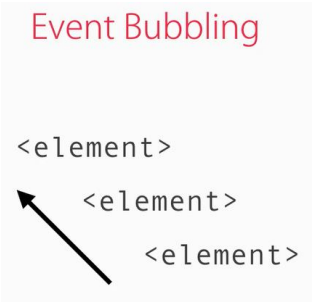
Sometimes, in the event handler onClick(), we may need to get access to the event that was raised. Ex. With mouse movements, the event object will tell us the x and y position. Or we can use the event object to get access to the DOM element that raised the event. To get access to the event, you add the event argument to onClick($event) and when subscribing, we need to pass it here (click)="onClick($event)".

```
10 export class AppComponent {
11
12     onClick($event){
13         console.log("Clicked", $event);
14     }
15 }
```

```
Elements  Console  Sources  Netw
<top frame>  Preserve log
Angular 2 is running in the development mo
production mode.
Clicked ▼MouseEvent {isTrusted: true}
    altKey: false
    bubbles: true
    button: 0
    buttons: 0
    cancelBubble: false
    cancelable: true
    clientX: 78
```

When the button is clicked, we get a mouse event.

```
Elements  Console  Sources  Network  Timeline
top frame>  Preserve log
    screenY: 242
    shiftKey: false
  ▶ sourceCapabilities: InputDeviceCapabilities
  ▶ srcElement: button
  ▼ target: button
      accessKey: ""
    ▶ attributes: NamedNodeMap
      autofocus: false
      baseURI: "http://localhost:3000/#"
      childElementCount: 0
```



The target property represents the DOM event that raised this event. In this case, it's a button.

All the DOM events bubble up the DOM tree, unless a handler along the way prevents further bubbling. This is a standard propagating mechanism. Ex:

```
3  @Component({
4      selector: 'my-app',
5      template: `
6          <div (click)="onDivClick()">
7              <button (click)="onClick($event)">Submit</button>
8          </div>
9      `
10 })
11 export class AppComponent {
12
13     onDivClick() {
14         console.log("Handled by Div");
15     }
16
17     onClick($event){
18         console.log("Clicked", $event);
19     }
20 }
```

When the button is clicked, the click event is raised from the button and it gets handled by the button itself. It will then bubble up to the containing element, in this case, the div.

It then gets handled by the other method, onDivClick(). If this div was contained by another element and that element was also interested in the click event, it would get notified as well. When the button is clicked, we get this:

```
Clicked ▶MouseEvent {isTrusted: true}
Handled by Div
```

To stop this bubbling, we add $event.stopPropagation(); in the method we want to be the last. If we apply this in onClick(), the handler in the div will not get called and we get this `Clicked ▶MouseEvent {isTrusted: true}`

Two-way Binding

The reality is that at the core of Angular, there is no two-way binding. There is only one-way binding that simulates two-way.

A directive called ngModel is used to create two-way binding between a DOM property and a component propery. ngModel replaces the following:

```
app.component.ts app
1  import {Component} from 'angular2/core';
2
3  @Component({
4      selector: 'my-app',
5      template: `
6          <input type="text" />
7      `
8  })
9  export class AppComponent {
10     title = "Angular App";
11 }
```

Let's say that we want to bind the input element's value property to the title property in our component. We can write this **<input type="text" [value]="title" />**, which is one direction of the binding from the component to the DOM. If the title property in the component changes, the DOM will be updated.

To implement the other direction, we use event binding. We are going to handle the input event of the input field **<input type="text" [value]="title" (input)="" />**, an event raised every time a character is typed. In the binding expression, we can either call a method to update the title in the component, or we can write the expression inline. This is a bad practice, because we want to encapsulate as much logic as possible in our component so we can easily unit test the logic. We write it here just for a demonstration.

We set up the **title property** of our component to be equal to the **value** of the **target element** (that raised the event) of the **event object** i.e. the value of the input field.

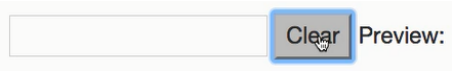
<input type="text" [value]="title" (input)="title = $event.target.value" />

We can test this with interpolation like so: **Preview {{ title }}**, and the result would be:

Every update in the input field i.e. DOM is reflected in the component. To check that the binding works the otner way from component to DOM, we can implement a button that clears the field.

<input type="button" (click)="title = ' ' " value="Clear" />

after click



However, the code is very cumbersome. This is where the ngModel directive comes into play.

```
app.component.ts app
1  rt {Component} from 'angular2/core';
2
3  ponent({
4  selector: 'my-app',
5  template: `
6      <input type="text" [value]="title" (input)="title = $event.target.value"
7      <input type="button" (click)="title = ''" value="Clear" />
8      Preview: {{ title }}
9  `
10
11 rt class AppComponent {
12 title = "Angular App";
```

Reminder: A directive is a class that allows us to control or extend the behaviour of the DOM.

ngModel

ngModel is a combination of property and event binding, and it is implemented like so:

<input type="text" [(ngMode)]="title" />

These two are identical.

```
<input type="text" [value]="title" (input)="title = $event.target.value"
<input type="text" [(ngModel)]="title" />
```

The ngModel directive applies property binding like **[value]="title"**
and event binding like **(input)="title = $event.target.value"**

An alternative to **[(ngMode)]="title"** is **bindon-ngModel ="title"** i.e. **bind (property)** and **on (event)** bidning.

As stated previously, there is no real two-way binding. Only a simulation. Data always flow one way.

Component API

Using the favorite star as an example, it always starts empty. What if we previously marked it as favorite? How do we keep what happened?

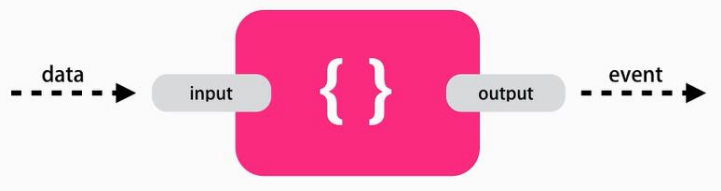
Component API

```
<favorite
    [isFavorite]="..."
    (change)="...">
</favorite>
```

The consumer of the favorite component, the app component, needs to use property binding to bind the isFavorite property of the component to some data. It would also be nice to use event binding to subscribe to events raised from this component.

Ex. When the user clicks the icon, the favorite component can raise a change event and app component will be notified.

To achieve this, we need to define a Public API for our component. We can mark few or more properties as input and output properties. These roperties would be visible from the outside and be available for property or event binding.



Button's API

```
<button
    [value]="..."
    (click)="...">
</favorite>
```

As an analogy, think of a DOM element like button. We can bind the value proerty of a button with a property in our component. Value here is an example of an **input property**. We can use it to pass data to our button.

Button also have events like click, that we can bind to methods in our components. Click is an example of an **output property**.

The combination of input and output properties, form the public API of a button. We want to declare a similar public API for a custom coponent.

Input Properties

The whole purpose of this is to make the properties visible to other other components. Ex. isFavorite is only visible in the FavoriteComponent and not in the AppComponent i.e. the component consuming it.

So, we need to declare such properties as input properties which is kind of like exposing them to the outside. We do this with an @Input decorator (it's a functions, so we call it), which needs to be imported.

This makes a property public, making it a part of the component's public API.

favorite.component.ts app

```
import {Component, Input} from 'angular2/core';

@Component({
    selector: 'favorite',
    template: `
    <i
        class="glyphicon"
        [class.glyphicon-star-empty]="!isFavorite"
        [class.glyphicon-star]="isFavorite"
        (click)="onClick()">
    </i>
    `
})
export class FavoriteComponent {
    @Input() isFavorite = false;

    onClick(){
        this.isFavorite = !this.isFavorite;
    }
}
```

favorite.component.ts app

```
import {Component, Input} from 'angular2/core';

@Component({
    selector: 'favorite',
    template: `
    <i
        class="glyphicon"
        [class.glyphicon-star-empty]="!isFavorite"
        [class.glyphicon-star]="isFavorite"
        (click)="onClick()">
    </i>
    `,
    inputs: ['isFavorite:is-favorite']
})
export class FavoriteComponent {
    @Input('is-favorite') isFavorite = false;

    onClick(){
        this.isFavorite = !this.isFavorite;
    }
}
```

We can give the public properties different names by passing a string argument in the @Imput function.

When we go in the component that consumes the original component, we can bind to the property made public and set its value.

Ex: Let's imagine that the AppComponent uses a service to get a post from a server. The post has a title and a property determining if the current user marked it as a favorite. We bind the element property with the component property i.e. The isFavorite is set to true from the "response" and that makes the CSS class have glyphicon-star appended to it, rather than the empty one.

app.component.ts app

```
import {Component} from 'angular2/core';
import {FavoriteComponent} from './favorite.component'

@Component({
    selector: 'my-app',
    template: `
        <favorite [isFavorite]="post.isFavorite"></favorite>
    `,
    directives: [FavoriteComponent]
})
export class AppComponent {
    post = {
        title: "Title",
        isFavorite: true
    }
}
```

app.component.ts app

```
import {Component} from 'angular2/core';
import {FavoriteComponent} from './favorite.component'

@Component({
    selector: 'my-app',
    template: `
        <favorite [is-favorite]="post.isFavorite"></favorite>
    `,
    directives: [FavoriteComponent]
})
export class AppComponent {
    post = {
        title: "Title",
        isFavorite: true
    }
}
```

To set the inputs, we can also use the inputs array `inputs: ['isFavorite:is-favorite']`, but this is not optimal. With this, we don't need to use the @Imput decorator i.e. we don't need to import it. Also, all of the inputs would be listed in one place. However, if during refactoring, some of the properties in the component are renamed, we would have to go back and rename them there as well.

Output Properties

The whole purpose of this is to make the events visible to other other components, so they can subscribe to them. If an event is raised in one component, a method can be called in another.

Ex. Everytime the favorite star is clicked, we want the component to raise a custom event. The consumer might be interested in the change and may want to do something (like calling a server to update data).

To raise a custom event, we first need to declare a property of type EventEmitter. The name of the property should be the name of the event we want published Ex. change.

To make it visible to the outside, we need to use @Output after we import it.

favorite.component.ts app

```
import {Component, Input, Output, EventEmitter} from 'angular2/core';

@Component({
    selector: 'favorite',
    template: `
    <i
        class="glyphicon"
        [class.glyphicon-star-empty]="!isFavorite"
        [class.glyphicon-star]="isFavorite"
        (click)="onClick()">
    </i>
    `
})
export class FavoriteComponent {
    @Input() isFavorite = false;

    @Output() change = new EventEmitter();

    onClick(){
        this.isFavorite = !this.isFavorite;
        this.change.emit({ newValue: this.isFavorite });
    }
}
```

We need to publish an event when onClick() is called and isFavorite is changed. We do that by emitting an object (along with any additional data) via our EventEmitter i.e. this.change with .emit. Here, we need to pass an obejct with the current value of isFavorite.

app.component.ts app

```
import {Component} from 'angular2/core';
import {FavoriteComponent} from './favorite.component'

@Component({
    selector: 'my-app',
    template: `
        <favorite [isFavorite]="post.isFavorite" (change)="onFavoriteChange(
    `,
    directives: [FavoriteComponent]
})
export class AppComponent {
    post = {
        title: "Title",
        isFavorite: true
    }

    onFavoriteChange($event){
        console.log($event);
    }
}
```

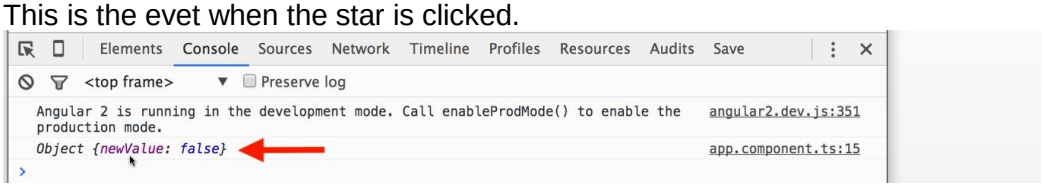
Now lets say the app component (the one consuming the favorite component) is interested in this event.

We can use event binding to handle this event i.e. **(change)=" "**.

In the binding expression, we make a call to a method in our component i.e. **onFavoriteChange($event)**.

We use $event to pass the additional data.

This is the evet when the star is clicked.



Templates

If a template is small, it is better to do it inline. The larger it gets, the better the idea to move it in a separate file. Doing it inline makes the component self sufficient and there is no need of declaring locations. The convention would be favorite.template.html

Also, note that this separation results in an extra HTTP call, which is don't only once the first time and it is cached for later use.

Both template and templateUrl can be used, but not both.

favorite.component.ts app

```
import {Component, Input, Output, EventEmitter} from 'angular2/core';

@Component({
    selector: 'favorite',
    template: `
    <i
        class="glyphicon"
        [class.glyphicon-star-empty]="!isFavorite"
        [class.glyphicon-star]="isFavorite"
        (click)="onClick()">
    </i>
    `
})
```

Becomes this:

• favorite.component.ts app

```
import {Component, Input, Output, EventEmitter} from 'angular2/core';

@Component({
    selector: 'favorite',
    templateUrl: 'app/favorite.template.html'
})
```

favorite.template.html app

```
    <i
        class="glyphicon"
        [class.glyphicon-star-empty]="!isFavorite"
        [class.glyphicon-star]="isFavorite"
        (click)="onClick()">
    </i>
```

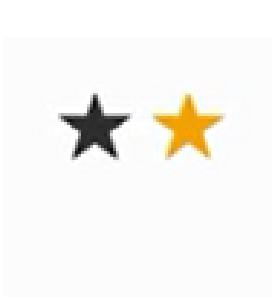
Styles

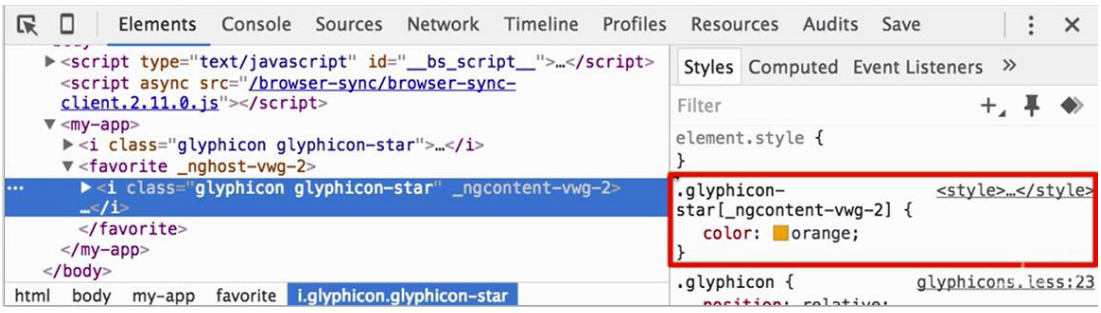
Another useful filed in the component meta data is styles or stylesUrl, used for CSS. They are declared similarly to templates. The interesting thing is that these styles are scoped to this component and won't leak outside. Using external styles generates an HTTP request so it is good to do them inline if they are not large.

```
@Component({
    selector: 'favorite',
    templateUrl: 'app/favorite.template.html',
    styles: [`
        .glyphicon-star {
            color: orange;
        }
    `]
})
```

app.component.ts app

```
import {Component} from 'angular2/core';
import {FavoriteComponent} from './favorite.component'

@Component({
    selector: 'my-app',
    template: `
        <i class="glyphicon glyphicon-star"></i>
        <favorite [isFavorite]="post.isFavorite"></favorite>
    `,
    directives: [FavoriteComponent]
})
```





The way styles don't leak is by Angular adding attributes to the element and the styles would only be applied to the elements containing the attribute. It is a random name and it changes every time.

Upvote / Downvote Exercise

The goals is to make a voting system where only one vote could be cast and the type of vote should be highlighted.

The votes and vote status should be preset in the app component via JSON API response simulation. Also, the act of voting should raise a custom event.

The Vote Component

vote.component.ts app

```
import {Component, Input, Output, EventEmitter} from 'angular2/core'

@Component({
    selector: 'vote',
    template: `
    <div style="width: 20px; font-size:30px; cursor: hand;">

        <span class="glyphicon glyphicon-menu-up"
            (click)="upvote()"
            [class.highlight]="myVote > 0"
        ></span>

        <span>{{voteCount}}</span>

        <span class="glyphicon glyphicon-menu-down"
            (click)="downvote()"
            [class.highlight]="myVote < 0"
        ></span

    </div>
    `,
    styles:[`
        .highlight{color: orange;}
    `]
})
```

```
export class VoteComponent {

    @Input() voteCount = 10;
    @Input() myVote = 0;

    @Output() vote = new EventEmitter();

    value = 1;

    upvote(){
        if(this.myVote < 1) {
            this.myVote += this.value;
            this.voteCount += this.value;
        }
        this.vote.emit({ voteStatus: this.myVote});
    }

    downvote(){
        if(this.myVote > -1) {
            this.myVote -= this.value;
            this.voteCount -= this.value;
        }
        this.vote.emit({ voteStatus: this.myVote});
    }
}
```

The App Component

app.component.ts

```
import {Component} from 'angular2/core'
import {VoteComponent} from './vote.component'


@Component({
    selector: 'my-app',
    template: `
        <vote
            [voteCount]="vote.voteCount"
            [myVote]="vote.myVote"
            (vote)="onVote($event)"
        ></vote>
    `,
    directives: [VoteComponent]
})
export class AppComponent {
    vote = {
        voteCount: 30,
        myVote: 1
    }

    onVote($event){
        console.log($event);
    }
}
```

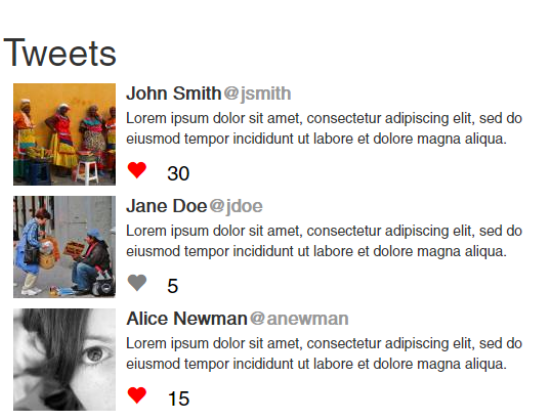
Mosh's Way

```
<div class="voter">
    <i
        class="glyphicon glyphicon-menu-up vote-button"
        [class.highlighted]="myVote == 1"
        (click)="upVote()"></i>

    <span class="vote-count">{{ voteCount + myVote }}</span>

    <i
        class="glyphicon glyphicon-menu-down vote-button"
        [class.highlighted]="myVote == -1"
        (click)="downVote()"></i>
</div>
```

voter.component.ts app

```
export class VoterComponent {
    @Input() voteCount = 0;
    @Input() myVote = 0;

    @Output() vote = new EventEmitter();

    upVote(){
        if (this.myVote == 1)
            return;

        this.myVote++;

        this.vote.emit({ myVote: this.myVote });
    }

    downVote(){
        if (this.myVote == -1)
            return;

        this.myVote--;
```

Tweets Exercise



The goal was to create a feed of tweets which were obtained from a service. The app component uses a tweet component which uses a favorite component.

The new thing here is the binding of the nested components i.e. how to reach the favorite component in the tweet component and give it a state with the information from the service.

We bind the data input to the tweet data from service. Then we bind the favorite input to the data.

app.component.ts app

```
import {Component} from 'angular2/core';
import {TweetComponent} from './tweets/components/tweet.component'
import {TweetService} from './tweets/services/tweet.service'

@Component({
    selector: 'my-app',
    directives: [TweetComponent],
    providers: [TweetService],
    template: `
        <h1>Tweets</h1>
        <div *ngFor="#tweet of tweets">
            <tweet [data]="tweet"></tweet>
        </div>
    `
})
export class AppComponent {
    tweets;
    constructor(tweetService:TweetService){
        this.tweets = tweetService.getTweets();
        console.log(this.tweets);
    }
}
```

tweet.service.ts app\tweets\services

```
export class TweetService{
    getTweets(){
        return [
            {
                image: "http://lorempixel.com/100/100/people/?3",
                author: "John Smith",
                handle: "@jsmith",
                status: "Lorem ipsum dolor sit amet, consectetur ad
                et dolore magna aliqua.",
                isLiked: true,
                favorites: 30
            },
            {
                image: "http://lorempixel.com/100/100/people/?9",
                author: "Jane Doe",
                handle: "@jdoe",
                status: "Lorem ipsum dolor sit amet, consectetur ad
                et dolore magna aliqua.",
                isLiked: false,
                favorites: 5
            },
            {
                image: "http://lorempixel.com/100/100/people/?7",
                author: "Alice Newman",
                handle: "@anewman",
                status: "Lorem ipsum dolor sit amet, consectetur ad
                et dolore magna aliqua.",
                isLiked: true,
                favorites: 15
            }
        ];
    }
}
```

tweet.component.ts app\tweets\components

```
import {Component, Input} from 'angular2/core'
import {FavoriteComponent} from './favorite.component'

@Component({
    selector: 'tweet',
    directives: [FavoriteComponent],
    template: `
        <div class="media tweet">
            <div class="media-left">
                <a href="#"><img class="media-object" src="{{data.image}}" alt="..."></a>
            </div>
            <div class="media-body">
                <h4 class="media-heading author">{{data.author}}</h4>
                <h4 class="media-heading handle">{{data.handle}}</h4>
                <p>{{data.status}}</p>
                <favorite
                    [isLiked]="data.isLiked"
                    [favorites]="data.favorites"
                ></favorite>
            </div>
        </div>
    `,
    styles: [`
        .author {font-weight:bold; float: left;} .handle {font-weight: bold; color: #999;}
        .tweet {margin: 10px; width:500px;}
    `]
})

export class TweetComponent{
    constructor(){
        console.log(this.data);
    }
    @Input() data;
}
```

favorite.component.ts app\tweets\components

```
import {Component, Input} from 'angular2/core'

@Component({
    selector: 'favorite',
    template: `
        <span class="glyphicon glyphicon-heart hand"
            [class.unfavorited]="!isLiked"
            [class.favorited]="isLiked"
            (click)="onClick()">
            <span class="text">{{ favorites }}</span>
        </span>
    `,
    styles:[`
        .favorited {color: red;}
        .unfavorited {color: gray;}
        .hand {cursor: hand; font-size: 20px;}
        .text {font-family:helvetica; font-size: 20px; color: black;}
    `]
})

export class FavoriteComponent {
    @Input() isLiked = false;
    @Input() favorites = 0;

    onClick(){
        if(this.isLiked==false){
            this.isLiked = !this.isLiked;
            this.favorites++;
        }else{
            this.isLiked = !this.isLiked;
            this.favorites--;
        }
    }
}
```

*ngIf

These two are identical in terms of hiding a div via an if statement. The result is the following.

You don't have any courses yet.

However, there is a subtle difference. *ngIf DOES NOT render the HTML in the DOM, whereas [hidden] renders it AND hides it.

This can impact the performance of the application significantly if there are a lot of elements.

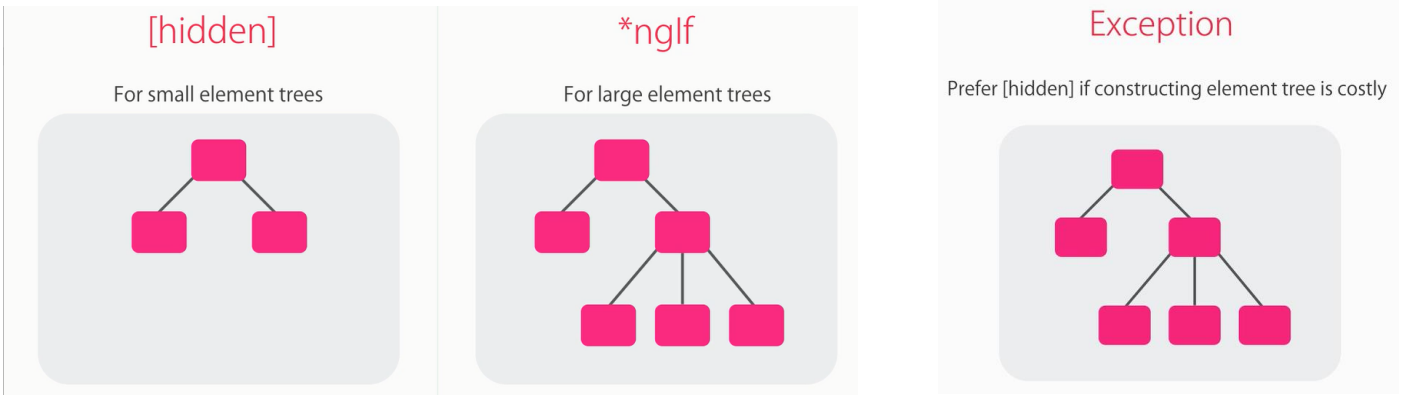
app.component.ts app

```
1  import {Component} from 'angular2/core';
2
3  @Component({
4      selector: 'my-app',
5      template: `
6          <div *ngIf="courses.length > 0">
7              List of courses
8          </div>
9          <div *ngIf="courses.length == 0">
10             You don't have any courses yet.
11         </div>
12     `
13 })
14 export class AppComponent {
15     courses = [];
16 }
```

app.component.ts app

```
1  import {Component} from 'angular2/core';
2
3  @Component({
4      selector: 'my-app',
5      template: `
6          <div [hidden]="courses.length == 0">
7              List of courses
8          </div>
9          <div [hidden]="courses.length > 0">
10             You don't have any courses yet.
11         </div>
12     `
13 })
14 export class AppComponent {
15     courses = [];
16 }
```

So, when should we use each?



*ngSwitch

We have two types of conditional statements. The angular equivalents are *ngIf and *ngSwitch

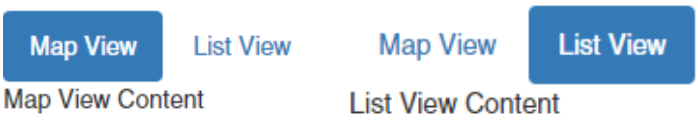
```
if (…) {}
else (…) {}
```

```
switch (…)
{
   case …:
}
```

ngSwitch

To display an element and its children from a set of possible elements, based on a condition.

Clicking on one of the “pills” i.e. menu items, renders a specific content.



The HTML5 template tags are not rendered until activated, which saves resources. The way this works is by binding the click event of the pills to set the property in the component to either map or list. After that, depending on the property, the active class is appended by using property binding.

To render the specific content, the ngSwitch directive is used. What this means is, we bind the switch property to the component property and in the case of viewMode = map, the map content is rendered and the same for list. This is a classic switch / case statement.

ng-switch.component.ts app\ng-switch

```
import {Component} from 'angular2/core'

@Component({
    selector: 'ng-switch',
    template: `
        <ul class="nav nav-pills">
            <li [class.active]=" viewMode == 'map' ">
                <a (click)=" viewMode = 'map' ">Map View</a>
            </li>
            <li [class.active]=" viewMode == 'list' ">
                <a (click)=" viewMode = 'list' ">List View</a>
            </li>
        </ul>
        <div [ngSwitch]="viewMode">
            <template [ngSwitchWhen]=" 'map' ">Map View Content</template>
            <template [ngSwitchWhen]=" 'list' ">List View Content</template>
        </div>
    `
})

export class SwitchComponent {
    viewMode = 'map';
}
```

app.component.ts

```
import {Component} from 'angular2/core';
import {SwitchComponent} from
'./ng-switch/ng-switch.component'

@Component({
    selector: 'my-app',
    directives: [SwitchComponent],
    template: `
        <h1>Sandbox 1</h1>
        <ng-switch></ng-switch>
    `
})
export class AppComponent { }
```

The click event binding could also call methods in the component. Ex. **(click)=”viewMode = ‘map’”** could be changed into **(click)=”mapViewMode()”** with **mapViewMode(){viewMode = ‘map’}**. This is great for unit testing.

*ngFor

Read the previous explanation of this. Most important thing to remember is that **"#course of courses"** is **"instance of object"**

Here a new thing is introduced, and that is indexing. This is nothing special as it only creates a new variable #i and it gives it the value of the index for the current item.

app.component.ts app

```
import {Component} from 'angular2/core';

@Component({
    selector: 'my-app',
    template: `
        <ul>
            <li *ngFor="#course of courses, #i = index">
                {{ i + 1 }} - {{ course }}
            </li>
        </ul>
    `
})
export class AppComponent {
    courses = ['Course 1', 'Course 2', 'Course 3'];
}
```

The Leading Asterisk *

The asterisk * is syntactic sugar for treating an element as a template. Under the hood, the asterisk * transforms the target element into a template element. Ex:

Using an asterisk *

```
<li *ngFor="#course of courses, #i = index">
    {{ i + 1 }} - {{ course }}
</li>
```

Not using one.

```
<template ngFor [ngForOf]="courses" #course #i=index>
    <li>{{ i + 1 }} - {{ course }}</li>
</template>
```

Pipes

It is a concept used to format data. There are built in ones like: Uppercase, Lowercase, Decimal, Currency, Date, JSON…and we can create custom ones.

```
export class AppComponent {
    course = {
        title: "Angular 2 for Beginners",
        rating: 4.9745,
        students: 5981,
        price: 99.95,
        releaseDate: new Date(2016, 3, 1)
    }
}
```

Before piping:

```
{{ course.title }}
<br/>
{{ course.students }}
<br/>
{{ course.rating }}
<br/>
{{ course.price }}
<br/>
{{ course.releaseDate }}
<br/>
{{ course }}
```

ANGULAR 2 FOR BEGINNERS
5981
4.9745
99.95
Fri Apr 01 2016 00:00:00 GMT+1100 (AEDT)
[object Object]

After piping:

```
{{ course.title | uppercase | lowercase }}
<br/>
{{ course.students | number }}
<br/>
{{ course.rating | number:'2.2-2' }}
<br/>
{{ course.price | currency:'AUD':true:'2.2-2' }}
<br/>
{{ course.releaseDate | date:'MMM yyyy' }}
<br/>
{{ course | json }}
```

angular 2 for beginners
5,981
04.97
A$99.95
Apr 2016
{ "title": "Angular 2 for Beginners", "rating": 4.9745, "students": 5981, "price": 99.95, "releaseDate": "2016-03-31T13:00:00.000Z" }

Creating Custom Pipes

As an example, we will create a pipe that displays an excerpt of some longer content.

app.component.ts app

```
import {Component} from 'angular2/core';

import {SummaryPipe} from './summary.pipe';

@Component({
    selector: 'my-app',
    template: `
        {{ post.title }}
        <br/>
        {{ post.body | summary }}
    `,
    pipes: [SummaryPipe]
})
```

```
export class AppComponent {
    post = {
        title: "Angular Tutorial for Beginners",
        body: `
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliqu
            interdum ut ex. Sed ullamcorper, leo nec maximus vestibulum, a
            pretium et vitae est.
        `
    }
}
```

To declare a pipe, we create a new file with the name.pipe.ts convention and we implement the PipeTransform interface.

PipeTransform is an interface which has the method tranform that takes in two arguments. The first one is for the actual content to be shortened and the second (string array) for any additional ones.

summary.pipe.ts app

```
import {Pipe, PipeTransform} from 'angular2/core';

@Pipe({ name: 'summary' })
export class SummaryPipe implements PipeTransform {
    transform(value: string, args: string[]) {
        if (value)
            return value.substring(0, 50) + "...";
    }
}
```

Angular Tutorial for Beginners
Lorem ipsum dolor sit amet, consectet...

To extend this pipe's functionality, we add more arguments to the transform method.

```
{{ post.body | summary:10 }}
```

summary.pipe.ts app

```
import {Pipe, PipeTransform} from 'angular2/core';

@Pipe({ name: 'summary' })
export class SummaryPipe implements PipeTransform {
    transform(value: string, args: string[]) {
        var limit = (args && args[0]) ? parseInt(args[0]) : 50;

        if (value)
            return value.substring(0, limit) + "...";
    }
}
```

The value of limit is set with a ternary operator where we first check if there is an argument (the number of characters i.e. :10 in the piping) and if that argument is in the 0 index of the args string array.

If true, convert the value at index 0 to an integer (because it is a string in a string array). If false, the default value is 50 characters.

ngClass

This is used for appending CSS classes depending on a logical expression.

This is one way to do it, useful for small CSS applications.

```
[class.glyphicon-star-empty]="!isFavorite"
[class.glyphicon-star]="isFavorite"
```

This is a second way, useful I situations with many CSS classes. Both of these examples are identical. We bind the ngClass directive to an expression, which is an object in this case.

The object can have one or more key value pairs. Each key represents a CSS class. If the value is true, the class will be added to the element. The keys i.e. glyphicon-star need to be enclosed in quotes because there are hyphens in the names, which is not allowed in property names in Javascript.

```
[ngClass]="{
    'glyphicon-star-empty': !isFavorite,
    'glyphicon-star': isFavorite
}"
```

ngStyle

It works the same way as ngClass. It is seen as an anti-pattern because you might as well use a class if there are many styles.

```
[style.backgroundColor]="canSave ? 'blue' : 'gray'"
[style.color]="canSave ? 'white' : 'black'"
[style.fontWeight]="canSave ? 'bold': 'normal'"
```

Using ngStyle, which is identical to the one above:

```
[ngStyle]="{
    backgroundColor: canSave ? 'blue' : 'gray',
    color: canSave ? 'white' : 'black',
    fontWeight: canSave ? 'bold': 'normal'
}"
```

Elvis Operator ?.

It is used to solve errors when accessing properties with null values.

- Title: Review applications
- Assigned to: Mosh

```
<ul>
    <li>Title: {{ task.title }}</li>
    <li>Assigned to: {{ task.assignee.name }}</li>
</ul>
```

```
12 export class AppComponent {
13     task = {
14         title: "Review applications",
15         assignee: {
16             name: "Mosh"
17         }
18     };
19 }
```

Using a null property results in an error.

```
12 export class AppComponent {
13     task = {
14         title: "Review applications",
15         assignee: null
16     };
17 }
```

```
▶EXCEPTION: TypeError: Cannot read property 'name' of null in [Assigned to: {{  angular2.dev.js:23514
task.assignee.name }} in AppComponent@3:16]
```

**
Assigned to: {{ task.asignee.name }}
**

We can use *ngIf to handle null values. This is not efficient because someone has to go over every property and make a mental image of everything that can be null, and apply the if statements.

**<li *ngIf=”task.asignee != null”>Assigned to: {{ task.asignee.name }}
**

Another way is to use a ternary operator. Using this is perfectly fine, but there is an even more elegant way.

**
Assigned to: {{ task.asignee != null ? task.asignee.name : “” }}
**

Elvis operator. It is in the blue color. That’s it. A simple question mark after the property. Good thing about this is that it can be used multiple times. This protects against possible null values.

**
Assigned to: {{ task.asignee?.name }}
**

**
Assigned to: {{ task.asignee?.role?.name }}
**

ngContent

If you are building reusable components, it's always good practice to prefix them (bs-panel).

Sometimes we need to insert content into our template of our component (BootstrapPanel) from the outside (AppComponent). This useful for supplying markup from a host component (AppComponent). To do this, we need to define the insertion points in the component (BootstrapPanel).

The insertion points are done with the ng-content directive and we identify it with a class. After that, we apply the classes to the elements containing the content, who also need to be wrapped in the proper selector (bs-panel).

app.component.ts app

```
import {Component} from 'angular2/core';
import {BootstrapPanel} from './bootstrap.panel.component';

@Component({
    selector: 'my-app',
    directives: [BootstrapPanel],
    template: `
        <bs-panel>
            <div class="heading">The Heading</div>
            <div class="body">This is the body!</div>
        </bs-panel>
    `
})
export class AppComponent {
}
```

bootstrap.panel.component.ts app

```
import {Component} from 'angular2/core';

@Component({
    selector: 'bs-panel',
    template: `
    <div class="panel panel-default">
        <div class="panel-heading">
            <ng-content select=".heading"></ng-content>
        </div>
        <div class="panel-body">
            <ng-content select=".body"></ng-content>
        </div>
    </div>
    `
})
export class BootstrapPanel {
}
```

The Heading

This is the body!

Challenge - Accordion

zippy-accordion.component.ts app\zippy-accordion

```
import {Component, Input} from 'angular2/core'

@Component({
    selector: 'zippy',
    template:`
        <div class="zippy col-md-12">
            <div class="zippy-title gray box" (click)="toggle()">
                {{title}}
                <div class="pull-right glyphicon"
                    [ngClass]="{
                        'glyphicon-chevron-down': !isExpanded,
                        'glyphicon-chevron-up': isExpanded
                    }">
                </div>
            </div>
            <div *ngIf="isExpanded" class="zippy-content light box">
                <ng-content></ng-content>
            </div>
        </div>
    `,styles: [`
        .box {padding: 5px;}
        .gray {background:#eee;}
        .light {background:#ccc;}
    `]
})

export class ZippyAccordionComponent{
    isExpanded = false;
    @Input() title: string;

    toggle(){
        this.isExpanded = !this.isExpanded;
    }
}
```

app.component.ts app

```
import {Component} from 'angular2/core';
import {ZippyAccordionComponent} from
'./zippy-accordion/zippy-accordion.component'

@Component({
    selector: 'my-app',
    directives: [ZippyAccordionComponent],
    template: `

        <div class="col-md-3">
            <zippy title="This is the title">
                This is the body
            </zippy>
            <zippy title="This is the title">
                This is the body
            </zippy>
        </div>
    `
})
export class AppComponent { }
```

This is the title

This is the body

This is the title

Zen Coding

Write code using CSS syntax, press TAB at the end and HTML markup is outputted. It's a great way of generating HTML without manually typing everything.

div.form-group + tab otuputs

. - specify classes
> - specify immediate children
+ - specify siblings
[ ] - specify attributes

SHIFT + ALT + F reformats the code in Visual Studio Code.

--

div.form-group>label+input.form-control[type='text']

outputs (TAB must be pressed at the end)

```
<div class="form-group">
        <label for=""></label>
        <input type="text" class="form-control"></div>
</div>
```

--

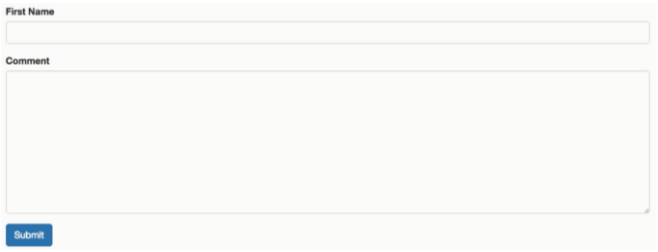
button.btn.btn-primary[type='submit']

```
<button class="btn btn-primary" type="submit"></button>
```

Basic Form

contact-form.component.html app\components

```
<form>
    <div class="form-group">
        <label for="firstName">First Name</label>
        <input id="firstName" type="text" class="form-control">
    </div>

    <div class="form-group">
        <label for="comment">Comment</label>
        <textarea id="comment" cols="30" rows="10" class="form-control"></textarea>
    </div>

    <button class="btn btn-primary" type="submit">Submit</button>
</form>
```



contact-form.component.ts app\components

```
import {Component} from 'angular2/core';

@Component({
    selector: 'contact-form',
    templateUrl: 'app/components/contact-form.component.html'
})
export class ContactFormComponent { }
```

app.component.ts app

```
import {Component} from 'angular2/core';
import {ContactFormComponent} from
'./components/contact-form.component'

@Component({
    selector: 'my-app',
    directives: [ContactFormComponent],
    template: `
        <contact-form></contact-form>
    `
})
export class AppComponent { }
```

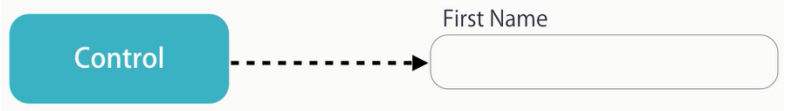
Control and Control Group

With these, we turn the basic forms into Angular forms. With this, we can tell if any of the inputs are changed, if they have errors, what values are storred etc.

Control

The control class represents a single input field in a form. It has properties which tell us:

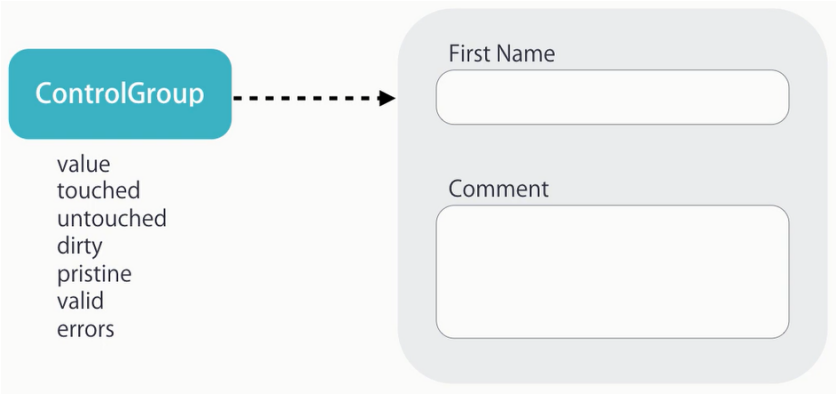
Value – The value in the field
Touched – Has it been interacted with
Untouched – Or not
Dirty – Has the value changed
Pristine – Or not
Valid – Is it valid
Errors – If not, which errors



Control Group

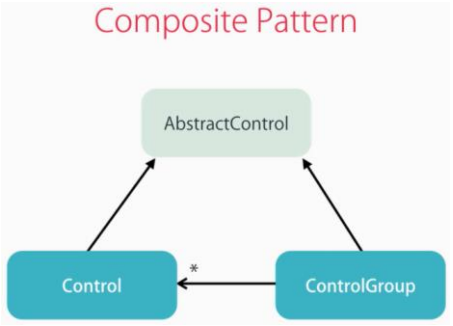
As the name suggests, this is a group of controls. Each form is a control group, as it contains at least one control.

All the properties available in the control class are available here too.



Both the **Control** and **ControlGroup** classes derive from another class called **AbstractControl**.

The model shown is the implementation of the composite pattern. **ControlGroup** is the composite, and **Control** is the part.



To implement a form in an Angular app, we need to create a **ControlGroup** object for the form and a **Control** object for each input field.

There are two ways to create these control objects. They can be created **Implicitly** by Angular, or we can **Explicitly** create them.

The difference between the two is that when we create them Explicitly, we have more control over the validation logic meaning we can implement a more complex validation logic. Also, we can unit test them. However, this requires more code than the implicit creation.

ngControl

A directive used to associate the input fields to the created control objects.

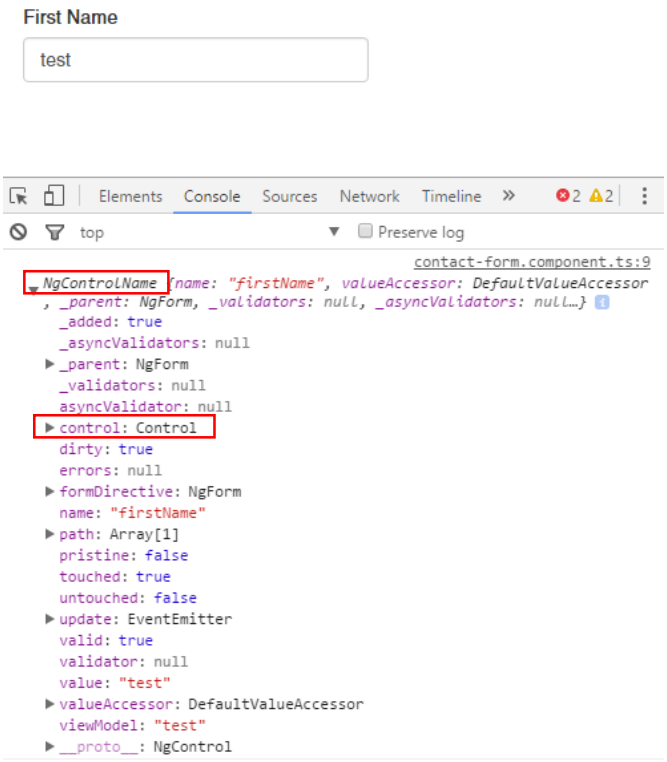
Implicit Creation

contact-form.component.html app\components

```
<form>
    <div class="form-group">
        <label for="firstName">First Name</label>
        <input
            ngControl="firstName"
            #firstName="ngForm"
            (change)="log(firstName)"
            id="firstName"
            type="text"
            class="form-control">
    </div>
</form>
```

contact-form.component.ts app\components

```
import {Component} from 'angular2/core';

@Component({
    selector: 'contact-form',
    templateUrl: 'app/components/contact-form.component.html'
})
export class ContactFormComponent {
    log(x){
        console.log(x);
    }
}
```



ngControl=”firstName”; firstName is the name of the control object behind the scenes i.e. associate this input field with the control object. Every control object has a name and it is a way to distinguish it from other objects. This is not to be confused with the id=”firstName” which is a DOM element.

This object is created automatically by Angular i.e. implicit creation.

We then bind the change event to a log method in the component. This will log any change the input field. We need to pass a reference in the log method of the directive ng-control=”firstName”, and to do this, we create a temporary local variable in the template and set it to the ng-control directive, like so #firstName=”ngForm”.

When Angular sees this temporary local variable on an input field set to ngForm, it automatocally applies that variable to the ngControl directive and it applies it to that field. Mosh said that they should have made it look like #firstName=”ngControl” but they decided to use ngForm.

This variable will hold a reference to the ngControl directive applied to the input field.

In the Google Chrome console, NgControlName is the ngControl object applied to the input element. Inside, there is a reference to a control object implicitly created by Angular. The properties displayed in NgControlName are computed properties meaning the real values reside in the control object within that the directive is referencing.

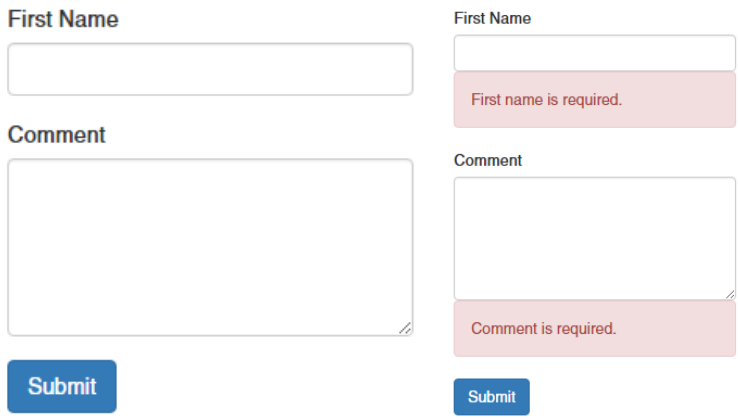
Validation Errors

In implicitly created controls, there are only 3 validation options available: required, minlength and maxlength. To enforce these, simply apply the HTML attribute.

In the binding expression for *ngIf we can use the temporary local variable, which is actually a reference to the ngControl directive applied to the input field. The directive has a property called valid which means we can dislay the error div by using *ngIf="!firstName.valid". In other words, when firstName.value is false, render the div.

contact-form.component.html app\components

```
<form>
    <div class="form-group">
        <label for="firstName">First Name</label>
        <input
            ngControl="firstName"
            #firstName="ngForm"
            id="firstName"
            type="text"
            class="form-control"
            required>
        <div
            *ngIf="firstName.touched && !firstName.valid"
            class="alert alert-danger">
            First name is required.
        </div>
    </div>
    <div class="form-group">
        <label for="comment">Comment</label>
        <textarea
            ngControl="comment"
            #comment="ngForm"
            id="comment"
            cols="30"
            rows="5"
            class="form-control"
            required>
        </textarea>
        <div
            *ngIf="comment.touched && !comment.valid"
            class="alert alert-danger">
            Comment is required.
        </div>
    </div>
</form>
```




```
▼ <form>
  ▼ <div class="form-group">
      <label for="firstName">First Name</label>
      <input class="form-control ng-touched ng-dirty ng-
      invalid" id="firstName" ngcontrol="firstName"
      required type="text">
      <!--template bindings={}-->
      <div class="alert alert-danger">
                First name is required.
          </div>
```

When the *ngIf is triggered, special classes are appended automatically by Angular to the input elements. Ex. firstName has the ng-touched (element was focused), ng-dirty (element was altered) and ng-invalid (element is empty) all added to it. These classes depend on the state of the input field.

styles.css app

```
.ng-touched.ng-invalid {
    border: 1px solid red;
}
```

What we can do now, is apply styles to these classes. In this case, we can give the input fields a red border whenever ng-invalid is appended. Ex: if the element **has both the ng-touched AND ng-invalid classes**, make the border red.

Specific Validation Errors

```
<div class="form-group">
    <label for="firstName">First Name</label>
    <input
        ngControl="firstName"
        #firstName="ngForm"
        id="firstName"
        type="text"
        class="form-control"
        required
        minlength="3">
    <div
        class="alert alert-danger"
        *ngIf="firstName.touched && firstName.errors.required">
        First name is required.
    </div>
    <div
        class="alert alert-danger"
        *ngIf="firstName.touched && firstName.errors.minlength">
        First name should be minimum 3 characters.</div>
</div>
```

What does **firstName.errors.minlength** mean? **firstName** is a reference to the **ngControl** directive applied to our input field.

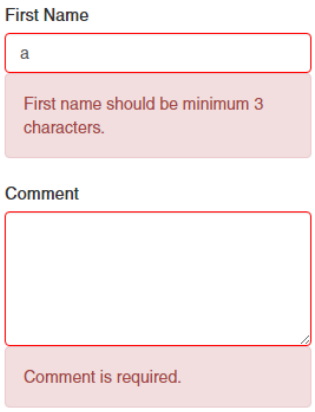
This directive has a property called **.errors**, which can be null if the input field has no errors.

Otherwise it's going to be an object with a separate key for each validation rule.

Here we are checking if the **firstName.errors** object has a key called **.required**.

This code is a bit messy as **firstName.touched** is duplicated, so a refactored code would look like this:

```
    class="form-control"
    required
    minlength="3">
<div *ngIf="firstName.touched && firstName.errors">
    <div
        class="alert alert-danger"
        *ngIf="firstName.errors.required">
        First name is required.
    </div>
    <div
        class="alert alert-danger"
        *ngIf="firstName.errors.minlength">
        First name should be minimum 3 characters.
    </div>
</div>
```



There is still one problem with this code, and that is the hardcoded value of 3 characters. If this number changes in the future, we would have to make the changes in two places. To render this dynamically, we use interpolation.

```
<div
    *ngIf="firstName.errors.minlength"
    class="alert alert-danger">
    First name should be minimum
    {{ firstName.errors.minlength.requiredLength }}
    characters.
</div>
```

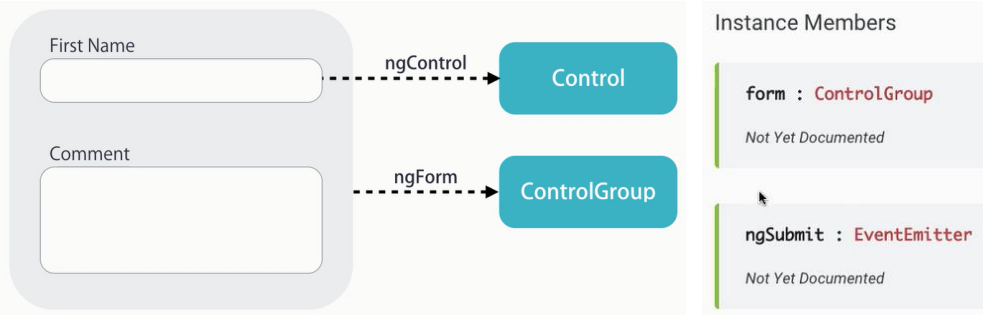
If we interpolate {{ firstName.errors | json }} we would get a JSON errors object like this:

{ "minlength": { "requiredLength": 3, "actualLength": 1 } }

minlength is the key, which is the validation that failed. Inside we have the value for the key. It is an object with two properties, **requredLength** and **actualLength**. With the above interpolation in the error div **{{ firstName.errors.minlength.requiredLength }}**, we access the requiredLength property to dynamically render the numer of required characters.

ngForm

To recap, **ngControl** is a directive that binds an input field to a **Control** object. **ngForm** is a similar directive which binds the whole form to a **ControlGroup** object. Unlike ngControl, we don't need to explicitly apply it on our form element. Angular automatically applies it whenever it encounters a form element.



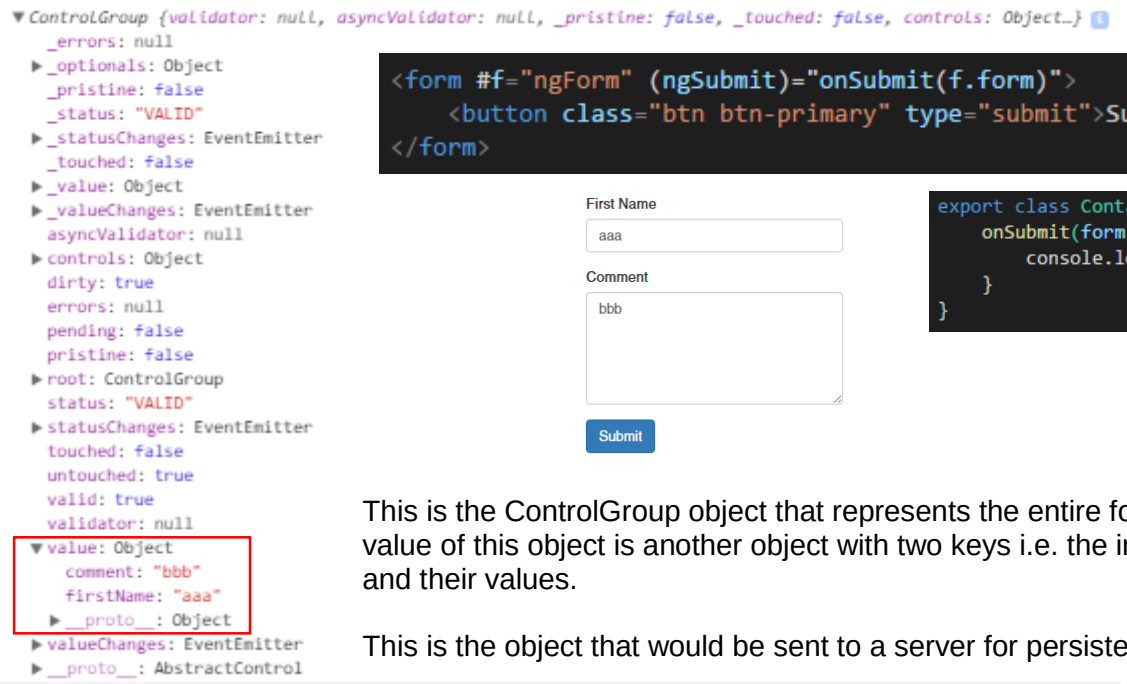
This directive has a property called **form** which is of type **ControlGroup**, so it binds our **form** element to a **Control** object under the hood.

It also has another property called **ngSubmit** which is an **EventEmitter**, used for publishing events from our components i.e. we can bind our ngSubmit to a method in our component, which is called when the user clicks the submit button.

In the form element, using event binding, we bind ngSubmit to a method in our component called onSubmit(). When we implement this method, we need to have access to the ControlGroup object that represents the entire form.

With that ControlGroup object, we can access each Control object, their state and their value, or we can access the state and value of the form as a whole.

We need to pass a reference of the **ControlGroup** object to our submit method. To do this, in our form element, we declare a temporary local variable **#f** and set it to **ngForm**. Angular sets the temporary local variable **#f** to the **ngForm** directive applied on the form element. This directive has a property called **form**, so in our submit method, we can pass **f.form**.



This is the ControlGroup object that represents the entire form. The value of this object is another object with two keys i.e. the input fields and their values.

This is the object that would be sent to a server for persistence.

Disable Submit Button

contact-form.component.html app\components

```
<form #f="ngForm" (ngSubmit)="onSubmit(f.form)">
    <button
        class="btn btn-primary"
        [disabled]="!f.valid"
        type="submit">
            Submit
    </button>
</form>
```

We bind the disabled property of the button to an expression. In this expression, we need to see if the form is valid or not.

We do this by using the temporary local variable #f that references the ngForm directive applied to the form element.

Subscribe Form Challenge

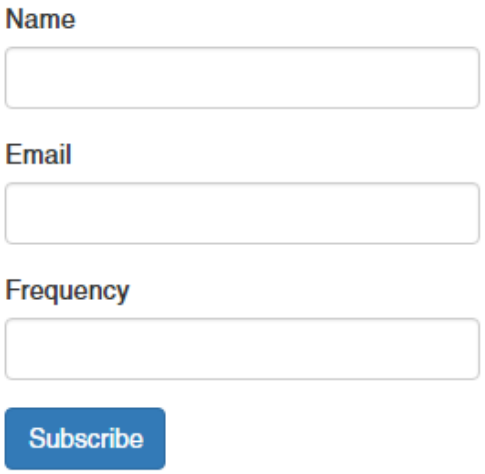
subscribe.component.html app\components

```
<form>
    <label>Name</label>
    <input>
    <label>Email</label>
    <input>
    <label>Frequency</label>
    <input>
    <button>Subscribe</button>
</form>
```

After Bootstrap

subscribe.component.html app\components

```
<form>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control">
    </div>
    <div class="form-group">
        <label>Email</label>
        <input class="form-control">
    </div>
    <div class="form-group">
        <label>Frequency</label>
        <input class="form-control">
    </div>
    <button class="btn btn-primary">
        Subscribe
    </button>
</form>
```



```
<form #f="ngForm" (ngSubmit)="onSubscribe(f.form)">
```

```
<div class="form-group">
    <label for="name">Name</label>
    <input
        ngControl="name"
        #name="ngForm"
        id="name"
        type="text"
        class="form-control"
        required
        minlength="3">
    <div *ngIf="name.touched && name.errors">
        <div
            class="alert alert-danger"
            *ngIf="name.errors.required">
            Name is required.
        </div>
        <div
            class="alert alert-danger"
            *ngIf="name.errors.minlength">
            Name should be minimum {{
            name.errors.minlength.requiredLength }}
            characters.
        </div>
    </div>
</div>
```

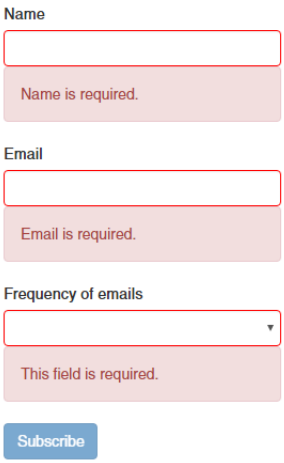
```
<div class="form-group">
    <label for="email">Email</label>
    <input
        ngControl="email"
        #email="ngForm"
        id="email"
        type="email"
        class="form-control"
        required>
    <div
        class="alert alert-danger"
        *ngIf="email.touched && !email.valid">
        Email is required.
    </div>
</div>
```

```
<div class="form-group">
    <label for="frequency">Frequency of
    emails</label>
    <select
        ngControl="frequency"
        #frequency="ngForm"
        id="frequency"
        class="form-control"
        required>
        <option value=""></option>
        <option *ngFor="#frequency of frequencies"
        value="{{ frequency.id }}">
            {{ frequency.label }}
        </option>
    </select>
    <div
        class="alert alert-danger"
        *ngIf="frequency.touched &&
        !frequency.valid">
        This field is required.
    </div>
</div>
```

```
    <button class="btn btn-primary" type="submit" [disabled]="!f.valid">Subscribe</button>
</form>
```

subscribe.component.ts app\components

```
1   import {Component} from 'angular2/core';
2
3   @Component({
4       selector: 'subscribe-form',
5       templateUrl: 'app/components/subscribe.component.html'
6   })
7   export class SubscribeFormComponent {
8       frequencies = [
9           { id: 1, label: 'Daily' },
10          { id: 2, label: 'Weekly' },
11          { id: 3, label: 'Monthly' }
12      ];
13
14      onSubscribe(form){
15          console.log(form.value);
16      }
17  }
```



PDO

- PDO and MySQLi objects connect to the database
- Use the object's methods to interact with the database
- Methods can take arguments
- Class constants are often used as arguments
- `PDO::FETCH_ASSOC`, `MYSQLI_ASSOC`

- Methods can return a new object
- `$result = $db->query($sql);`
- `$row = $result->fetch();`
- `echo $db->affected_rows;`

MySQLi

- MySQL is the dominant database used with PHP
- Has some MySQL features not supported by PDO
- MySQLi is compatible with MariaDB
- Original MySQL functions deprecated since PHP 5.5
- ~~`mysql_query()`, `mysql_fetch_assoc()`, etc~~

- Two interfaces: procedural and object-oriented
- No significant difference in performance
- Object-oriented interface is more concise
- Code is easier to read

Prepared Statements

- Template for SQL query that uses values from user input
- Placeholders for values stored in variables
- Prevents SQL injection
- More efficient when same query is reused
- Can bind results to named variables

Placeholders

- Can be used only for column values
- Cannot be used for column names or operators
- Non-numeric values are automatically wrapped in quotes

```
$sql = 'SELECT user_id, first_name, last_name
        FROM users
        WHERE username = ? AND password = ?';
```

```
$sql = 'SELECT user_id, first_name, last_name
        FROM users
        WHERE username = :username AND password = :pwd';
```

Using Prepared Statements

- Prepare and validate the SQL with placeholders
- Bind values to the placeholders
- Execute the statement
- Bind output values to variables (optional)
- Fetch the results

Transactions

- Set of SQL queries executed as a unit
- Operation is committed only if all parts succeed
- Transaction can be rolled back if an error occurs
- Particularly useful for financial transfers
- Prevents rows being modified by another connection

PDO Basics

DSN – Database Source Name

- Identifies which database to connect to
- Prefix followed by colon identifies PDO driver
- Name/value pairs separated by semicolons
- DSN format depends on the driver

- **MySQL**

```
$dsn = 'mysql:host=localhost;dbname=oophp';
$dsn = 'mysql:host=localhost;port=3307;dbname=oophp';
```

- **SQLite3**

```
$dsn = 'sqlite:/path/to/oophp.db';
```

- **MS SQL Server**

```
$dsn = 'sqlsrv:Server=localhost;Database=oophp';
```

Fetching Results (4 ways)

- `fetch()` gets the next row from a result set

```php
<?php
try {
    require_once '../../includes/pdo_connect.php';
    $sql = 'SELECT name, meaning, gender FROM names
            ORDER BY name';
    $result = $db->query($sql);
} catch (Exception $e) {
    $error = $e->getMessage();
}
```

```php
<?php while ($row = $result->fetch()) { ?>
<tr>
    <td><?php echo $row[0]; ?></td>
    <td><?php echo $row['meaning']; ?></td>
    <td><?php echo $row['gender']; ?></td>
</tr>
<?php } ?>
```

- `fetchAll()` creates an array containing all rows

```php
<?php
try {
    require_once '../../includes/pdo_connect.php';
    $sql = 'SELECT name, meaning, gender FROM names
            ORDER BY name';
    $result = $db->query($sql);
    $all = $result->fetchAll();
} catch (Exception $e) {
    $error = $e->getMessage();
}
```

```
Array
(
    [0] => Array
        (
            [name] => Alice
            [0] => Alice
            [meaning] => noble, light
            [1] => noble, light
            [gender] => girl
            [2] => girl
        )

    [1] => Array
        (
            [name] => Aubrey
            [0] => Aubrey
            [meaning] => ruler of elves
            [1] => ruler of elves
            [gender] => unisex
            [2] => unisex
        )
```

Use **fetchAll(PDO::FETCH_ASSOC)** to get just the column names.
Use **fetchAll(PDO::FETCH_NUM)** to get just the column numbers.

- `fetchColumn()` gets a single column from the next row

```php
<?php
try {
    require_once '../../includes/pdo_connect.php';
    $sql = 'SELECT name, meaning, gender FROM names
            ORDER BY name';
    $result = $db->query($sql);
    $all = $result->fetchAll();
} catch (Exception $e) {
    $error = $e->getMessage();
}
```

```php
<table>
    <tr>
        <th>Column</th>
    </tr>
    <?php while($col = $result->fetchColumn()) { ?>
    <tr>
        <td><?php echo $col; ?></td>
    </tr>
    <?php } ?>
</table>
```

- `fetchObject()` gets the next row as an object

Query vs Exec

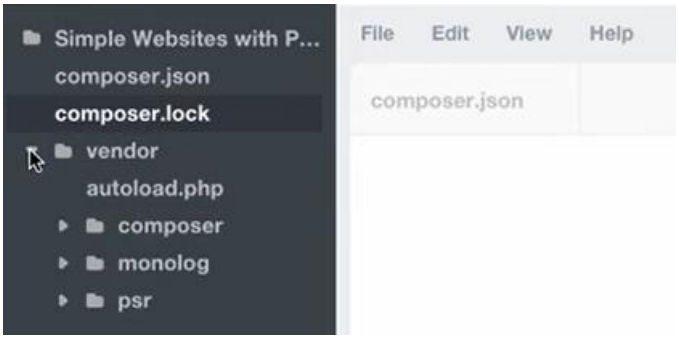
Which Should I Use?

- `query()`
  - Returns the result set for SELECT queries
  - Returns the SQL with INSERT, UPDATE, and DELETE
- `exec()`
  - Returns the number of rows affected
  - Better for INSERT, UPDATE, and DELETE

Slim Framework

Composer

- You have a project that depends on a number of libraries
- Some of those libraries depend on other libraries
- You declare the things you depend on
- Composer finds out which versions of which packages need to be installed, and installs them
- Downloads them into your project



Everything in the vendor folder is managed by composer.json

Autoload.php fixes the messy usage of include at the top of every page when doing OOP.

Namespacing

Without Namespacing

index.php

```php
<?php
require 'vendor/autoload.php';
date_default_timezone_set('America/New_York');

$log = new Monolog\Logger('name');
$log->pushHandler(new Monolog\Handler\StreamHandler('app.txt', Monolog\Logger::WARNING));

$log->addWarning('Foo');
echo 'Hello World!';
```

With Namespacing

index.php

```php
<?php
require 'vendor/autoload.php';
date_default_timezone_set('America/New_York');

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

$log = new Logger('name');
$log->pushHandler(new StreamHandler('app.txt', Logger::WARNING));

$log->addWarning('Foo');
echo 'Hello World!';
```

MVC

MVC

- model : contains database related code
- view : contains our user interaction code
- controller : links the view code to the model code

Slim

- Routing
- HTTP Request and Response
- Caching
- Middleware
- Sessions
- Overkill Military Grade Crypto

Apache Configuration

Ensure your .htaccess and index.php files are in the same public-accessible directory. The .htaccess file should contain this code:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^ index.php [QSA,L]
```

Make sure your Apache virtual host is configured with the AllowOverride option so that the .htaccess rewrite rules can be used:

```
AllowOverride All
```

Don’t forget to activate the rewrite module in the file httpd.conf.

Change this line : #LoadModule rewrite_module modules/mod_rewrite.so

To : LoadModule rewrite_module modules/mod_rewrite.so

Restart all services

RESTful Web API

The key principles of REST involve separating your API into logical resources. These resources are manipulated using HTTP requests where the method (GET, POST, PUT, PATCH, DELETE) has specific meaning.

Our HTTP verbs are POST, GET, PUT, and DELETE. (I think of them as mapping to the old metaphor of CRUD (Create-Read-Update-Delete).)

Use 2 base URLs per resource. In your URLs - nouns are good; verbs are bad.

Resource	POST create	GET read	PUT update	DELETE delete
/dogs	create a new dog	list dogs	**bulk update dogs**	delete all dogs
/dogs/1234	**error**	show Bo	if exists update Bo **if not error**	delete Bo

Once you have your resources defined, you need to identify what actions apply to them and how those would map to your API. RESTful principles provide strategies to handle CRUD actions using HTTP methods mapped as follows:

GET /tickets - Retrieves a list of tickets
GET /tickets/12 - Retrieves a specific ticket
POST /tickets - Creates a new ticket
PUT /tickets/12 - Updates ticket #12
PATCH /tickets/12 - Partially updates ticket #12
DELETE /tickets/12 - Deletes ticket #12

Should the endpoint name be singular or plural? The keep-it-simple rule applies here. Although your inner-grammatician will tell you it's wrong to describe a single instance of a resource using a plural, the pragmatic answer is to keep the URL format consistent and always use a plural. Not having to deal with odd pluralization (person/people, goose/geese) makes the life of the API consumer better and is easier for the API provider to implement (as most modern frameworks will natively handle /tickets and /tickets/12 under a common controller).

But how do you deal with relations? If a relation can only exist within another resource, RESTful principles provide useful guidance. Let's look at this with an example. A ticket in Enchant consists of a number of messages. These messages can be logically mapped to the /tickets endpoint as follows:

GET /tickets/12/messages - Retrieves list of messages for ticket #12
GET /tickets/12/messages/5 - Retrieves message #5 for ticket #12
POST /tickets/12/messages - Creates a new message in ticket #12
PUT /tickets/12/messages/5 - Updates message #5 for ticket #12
PATCH /tickets/12/messages/5 - Partially updates message #5 for ticket #12
DELETE /tickets/12/messages/5 - Deletes message #5 for ticket #12

What about actions that don't fit into the world of CRUD operations?

This is where things can get fuzzy. There are a number of approaches:

Restructure the action to appear like a field of a resource. This works if the action doesn't take parameters. For example an activate action could be mapped to a boolean activated field and updated via a PATCH to the resource.
Treat it like a sub-resource with RESTful principles. For example, GitHub's API lets you star a gist with PUT /gists/:id/star and unstar with DELETE /gists/:id/star.
Sometimes you really have no way to map the action to a sensible RESTful structure. For example, a multi-resource search doesn't really make sense to be applied to a specific resource's endpoint. In this case, /search would make the most sense even though it isn't a resource. This is OK - just do what's right from the perspective of the API consumer and make sure it's documented clearly to avoid confusion.

SSL everywhere - all the time

Always use SSL. No exceptions. Today, your web APIs can get accessed from anywhere there is internet (like libraries, coffee shops, airports among others). Not all of these are secure. Many don't encrypt communications at all, allowing for easy eavesdropping or impersonation if authentication credentials are hijacked.

Another advantage of always using SSL is that guaranteed encrypted communications simplifies authentication efforts - you can get away with simple access tokens instead of having to sign each API request.

One thing to watch out for is non-SSL access to API URLs. Do not redirect these to their SSL counterparts. Throw a hard error instead! The last thing you want is for poorly configured clients to send requests to an unencrypted endpoint, just to be silently redirected to the actual encrypted endpoint.

Result filtering, sorting & searching

It's best to keep the base resource URLs as lean as possible. Complex result filters, sorting requirements and advanced searching (when restricted to a single type of resource) can all be easily implemented as query parameters on top of the base URL. Let's look at these in more detail:

Filtering: Use a unique query parameter for each field that implements filtering. For example, when requesting a list of tickets from the /tickets endpoint, you may want to limit these to only those in the open state. This could be accomplished with a request like GET /tickets?state=open. Here, state is a query parameter that implements a filter.

Sorting: Similar to filtering, a generic parameter sort can be used to describe sorting rules. Accommodate complex sorting requirements by letting the sort parameter take in list of comma separated fields, each with a possible unary negative to imply descending sort order. Let's look at some examples:

GET /tickets?sort=-priority - Retrieves a list of tickets in descending order of priority
GET /tickets?sort=-priority,created_at - Retrieves a list of tickets in descending order of priority. Within a specific priority, older tickets are ordered first
Searching: Sometimes basic filters aren't enough and you need the power of full text search. Perhaps you're already using ElasticSearch or another Lucene based search technology. When full text search is used as a mechanism of retrieving resource instances for a specific type of resource, it can be exposed on the API as a query parameter on the resource's endpoint. Let's say q. Search queries should be passed straight to the search engine and API output should be in the same format as a normal list result.

Combining these together, we can build queries like:

GET /tickets?sort=-updated_at - Retrieve recently updated tickets
GET /tickets?state=closed&sort=-updated_at - Retrieve recently closed tickets

GET /tickets?q=return&state=open&sort=-priority,created_at - Retrieve the highest priority open tickets mentioning the word 'return'

JSON only responses

It's time to leave XML behind in APIs. It's verbose, it's hard to parse, it's hard to read, its data model isn't compatible with how most programming languages model data and its extendibility advantages are irrelevant when your output representation's primary needs are serialization from an internal representation.

snake_case vs camelCase for field names

If you're using JSON (JavaScript Object Notation) as your primary representation format, the "right" thing to do is to follow JavaScript naming conventions - and that means camelCase for field names! If you then go the route of building client libraries in various languages, it's best to use idiomatic naming conventions in them - camelCase for C# & Java, snake_case for python & ruby.

Food for thought: I've always felt that snake_case is easier to read than JavaScript's convention of camelCase. I just didn't have any evidence to back up my gut feelings, until now. Based on an eye tracking study on camelCase and snake_case (PDF) from 2010, snake_case is 20% easier to read than camelCase! That impact on readability would affect API explorability and examples in documentation.

Many popular JSON APIs use snake_case. I suspect this is due to serialization libraries following naming conventions of the underlying language they are using. Perhaps we need to have JSON serialization libraries handle naming convention transformations.

Pretty print by default & ensure gzip is supported

An API that provides white-space compressed output isn't very fun to look at from a browser. Although some sort of query parameter (like ?pretty=true) could be provided to enable pretty printing, an API that pretty prints by default is much more approachable. The cost of the extra data transfer is negligible, especially when you compare to the cost of not implementing gzip.

Consider some use cases: What if an API consumer is debugging and has their code print out data it received from the API - It will be readable by default. Or if the consumer grabbed the URL their code was generating and hit it directly from the browser - it will be readable by default. These are small things. Small things that make an API pleasant to use!

But what about all the extra data transfer?

Let's look at this with a real world example. I've pulled some data from GitHub's API, which uses pretty print by default. I'll also be doing some gzip comparisons:

$ curl https://api.github.com/users/veesahni > with-whitespace.txt
$ ruby -r json -e 'puts JSON JSON.parse(STDIN.read)' < with-whitespace.txt > without-whitespace.txt
$ gzip -c with-whitespace.txt > with-whitespace.txt.gz
$ gzip -c without-whitespace.txt > without-whitespace.txt.gz
The output files have the following sizes:

without-whitespace.txt - 1252 bytes
with-whitespace.txt - 1369 bytes
without-whitespace.txt.gz - 496 bytes
with-whitespace.txt.gz - 509 bytes
In this example, the whitespace increased the output size by 8.5% when gzip is not in play and 2.6% when gzip is in play. On the other hand, the act of gzipping in itself provided over 60% in bandwidth savings. Since the cost of pretty printing is relatively small, it's best to pretty print by default and ensure gzip compression is supported!

To further hammer in this point, Twitter found that there was an 80% savings (in some cases) when enabling gzip compression on their Streaming API. Stack Exchange went as far as to never return a response that's not compressed!

Authentication

A RESTful API should be stateless. This means that request authentication should not depend on cookies or sessions. Instead, each request should come with some sort authentication credentials.

By always using SSL, the authentication credentials can be simplified to a randomly generated access token that is delivered in the user name field of HTTP Basic Auth. The great thing about this is that it's completely browser explorable - the browser will just popup a prompt asking for credentials if it receives a 401 Unauthorized status code from the server.

However, this token-over-basic-auth method of authentication is only acceptable in cases where it's practical to have the user copy a token from an administration interface to the API consumer environment. In cases where this isn't possible, OAuth 2 should be used to provide secure token transfer to a third party. OAuth 2 uses Bearer tokens & also depends on SSL for its underlying transport encryption.

An API that needs to support JSONP will need a third method of authentication, as JSONP requests cannot send HTTP Basic Auth credentials or Bearer tokens. In this case, a special query parameter access_token can be used. Note: there is an inherent security issue in using a query parameter for the token as most web servers store query parameters in server logs.

For what it's worth, all three methods above are just ways to transport the token across the API boundary. The actual underlying token itself could be identical.

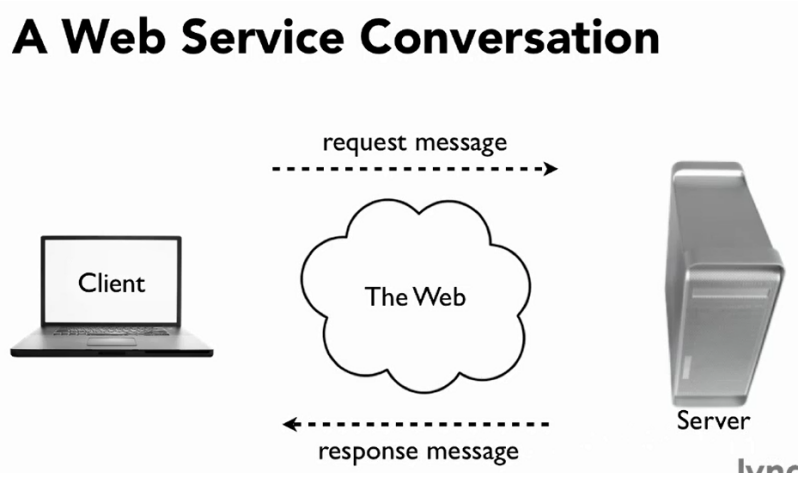
HTTP status codes

HTTP defines a bunch of meaningful status codes that can be returned from your API. These can be leveraged to help the API consumers route their responses accordingly. I've curated a short list of the ones that you definitely should be using:

200 OK - Response to a successful GET, PUT, PATCH or DELETE. Can also be used for a POST that doesn't result in a creation.
201 Created - Response to a POST that results in a creation. Should be combined with a Location header pointing to the location of the new resource
204 No Content - Response to a successful request that won't be returning a body (like a DELETE request)
304 Not Modified - Used when HTTP caching headers are in play
400 Bad Request - The request is malformed, such as if the body does not parse
401 Unauthorized - When no or invalid authentication details are provided. Also useful to trigger an auth popup if the API is used from a browser
403 Forbidden - When authentication succeeded but authenticated user doesn't have access to the resource
404 Not Found - When a non-existent resource is requested
405 Method Not Allowed - When an HTTP method is being requested that isn't allowed for the authenticated user
410 Gone - Indicates that the resource at this end point is no longer available. Useful as a blanket response for old API versions
415 Unsupported Media Type - If incorrect content type was provided as part of the request
422 Unprocessable Entity - Used for validation errors
429 Too Many Requests - When a request is rejected due to rate limiting

Web Services

A web service is a framework for a conversation between two computers.



R

R Statistics Essential Training - 26.09.2013 @ 5h 59m

You Don’t Know JS – Up & Going

Interpretation vs Compiling

Statements like a = b * 2 are helpful for developers when reading and writing, but are not actually in a form the computer can directly understand. So a special utility on the computer (either an interpreter or a compiler) is used to translate the code you write into commands a computer can understand.

For some computer languages, this translation of commands is typically done from top to bottom, line by line, every time the program is run, which is usually called interpreting the code.

For other languages, the translation is done ahead of time, called compiling the code, so when the program runs later, what's running is actually the already compiled computer instructions ready to go.

It's typically asserted that JavaScript is interpreted, because your JavaScript source code is processed each time it's run. But that's not entirely accurate. The JavaScript engine actually compiles the program on the fly and then immediately runs the compiled code.

Converting Between Types and Coercion

If you have a number but need to print it on the screen, you need to convert the value to a string, and in JavaScript this conversion is called "coercion." Similarly, if someone enters a series of numeric characters into a form on an ecommerce page, that's a string, but if you need to then use that value to do math operations, you need to coerce it to a number.

JavaScript provides several different facilities for forcibly coercing between types. For example:

```
var a = "42";
var b = Number( a );

console.log( a );   // "42"
console.log( b );   // 42
```

Using Number(..) (a built-in function) as shown is an explicit coercion from any other type to the number type. That should be pretty straightforward.

But a controversial topic is what happens when you try to compare two values that are not already of the same type, which would require implicit coercion.

When comparing the string "99.99" to the number 99.99, most people would agree they are equivalent. But they're not exactly the same, are they? It's the same value in two different representations, two different types. You could say they're "loosely equal," couldn't you?

To help you out in these common situations, JavaScript will sometimes kick in and implicitly coerce values to the matching types.

So if you use the == loose equals operator to make the comparison "99.99" == 99.99, JavaScript will convert the left-hand side "99.99" to its number equivalent 99.99. The comparison then becomes 99.99 == 99.99, which is of course true.

While designed to help you, implicit coercion can create confusion if you haven't taken the time to learn the rules that govern its behavior. Most JS developers never have, so the common feeling is that implicit coercion is confusing and harms programs with unexpected bugs, and should thus be avoided. It's even sometimes called a flaw in the design of the language.

However, implicit coercion is a mechanism that can be learned, and moreover should be learned by anyone wishing to take JavaScript programming seriously. Not only is it not confusing once you learn the rules, it can actually make your programs better! The effort is well worth it.

Comments

- Code without comments is suboptimal.
- Too many comments (one per line, for example) is probably a sign of poorly written code.
- Comments should explain why, not what. They can optionally explain how if that's particularly confusing.

State

State is tracking the changes to values as your program runs.

Blocks

```
var amount = 99.99;

// a general block
{
    amount = amount * 2;
    console.log( amount );  // 199.98
}
```

This kind of standalone { .. } general block is valid, but isn't as commonly seen in JS programs.

Typically, blocks are attached to some other control statement, such as an if statement (see "Conditionals") or a loop (see "Loops").

For example:

```
var amount = 99.99;

// is amount big enough?
if (amount > 10) {          // <-- block attached to `if`
    amount = amount * 2;
    console.log( amount );  // 199.98
}
```

We'll explain if statements in the next section, but as you can see, the { .. } block with its two statements is attached to if (amount > 10);

the statements inside the block will only be processed if the conditional passes.

Note: Unlike most other statements like console.log(amount);, a block statement does not need a semicolon (;) to conclude it.

Scope

If you ask the phone store employee for a phone model that her store doesn't carry, she will not be able to sell you the phone you want. She only has access to the phones in her store's inventory. You'll have to try another store to see if you can find the phone you're looking for.

Programming has a term for this concept: scope (technically called lexical scope). In JavaScript, each function gets its own scope. Scope is basically a collection of variables as well as the rules for how those variables are accessed by name. Only code inside that function can access that function's scoped variables.

Continues below… **V V V**

A variable name has to be unique within the same scope -- there can't be two different a variables sitting right next to each other. But the same variable name a could appear in different scopes.

```
function one() {
    // this `a` only belongs to the `one()`
function
    var a = 1;
    console.log( a );
}

function two() {
    // this `a` only belongs to the `two()`
function
    var a = 2;
    console.log( a );
}

one();      // 1
two();      // 2
```

Also, a scope can be nested inside another scope, just like if a clown at a birthday party blows up one balloon inside another balloon.

If one scope is nested inside another, code inside the innermost scope can access variables from either scope.

Consider:

```
function outer() {
    var a = 1;

    function inner() {
        var b = 2;

        // we can access both `a` and `b`
here
        console.log( a + b );   // 3
    }

    inner();

    // we can only access `a` here
    console.log( a );           // 1
}

outer();
```

Lexical scope rules say that code in one scope can access variables of either that scope or any scope outside of it.

So, code inside the inner() function has access to both variables a and b, but code in outer() has access only to a -- it cannot access b because that variable is only inside inner().

Recall this code snippet from earlier:

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // calculate the new amount with the tax
    amt = amt + (amt * TAX_RATE);

    // return the new amount
    return amt;
}
```

The TAX_RATE constant (variable) is accessible from inside the calculateFinalPurchaseAmount(..) function, even though we didn't pass it in, because of lexical scope.

Nested Scopes

```
function foo() {
    var a = 1;

    function bar() {
        var b = 2;

        function baz() {
            var c = 3;

            console.log( a, b, c ); // 1 2 3
        }

        baz();
        console.log( a, b );        // 1 2
    }

    bar();
    console.log( a );               // 1
}

foo();
```

In addition to creating declarations for variables at the function level, ES6 lets you declare variables to belong to individual blocks (pairs of { .. }), using the let keyword.

Besides some nuanced details, the scoping rules will behave roughly the same as we just saw with functions:

Because of using let instead of var, **b** will belong only to the **if statement** and thus not to the whole **foo()** function's scope. Similarly, **c** belongs only to the **while loop**.

Block scoping is very useful for managing your variable scopes in a more fine-grained fashion, which can make your code much easier to maintain over time.

Conditionals

In addition to the if statement we introduced briefly in Chapter 1, JavaScript provides a few other conditionals mechanisms that we should take a look at.

Sometimes you may find yourself writing a series of if..else..if statements like this:

```
if (a == 2) {
    // do something
}
else if (a == 10) {
    // do another thing
}
else if (a == 42) {
    // do yet another thing
}
else {
    // fallback to here
}
```

This structure works, but it's a little verbose because you need to specify the a test for each case.

Here's another option, the switch statement:

```
switch (a) {
    case 2:
        // do something
        break;
    case 10:
        // do another thing
        break;
    case 42:
        // do yet another thing
        break;
    default:
        // fallback to here
}
```

The **break** is important if you want only the statement(s) in one case to run. If you omit break from a case, and that case matches or runs, execution will continue with the next case's statements regardless of that case matching. This so called "fall through" is sometimes useful/desired:

Continues below… **V V V**

Another form of conditional in JavaScript is the "conditional operator," often called the "ternary operator." It's like a more concise form of a single if..else statement, such as:

```
var a = 42;

var b = (a > 41) ? "hello" : "world";
```

similar to:

```
if (a > 41) {
    b = "hello";
}
else {
    b = "world";
}
```

If the test expression (a > 41 here) evaluates as true, the first clause ("hello") results, otherwise the second clause ("world") results, and whatever the result is then gets assigned to b.

Strict Mode

ES5 added a "strict mode" to the language, which tightens the rules for certain behaviors. Generally, these restrictions are seen as keeping the code to a safer and more appropriate set of guidelines. Also, adhering to strict mode makes your code generally more optimizable by the engine. **Strict mode is a big win for code, and you should use it for all your programs.**

You can opt in to strict mode for an individual function, or an entire file, depending on where you put the strict mode pragma:

```
function foo() {
    "use strict";

    // this code is strict mode

    function bar() {
        // this code is strict mode
    }
}

// this code is not strict mode
```

Compare that to:

```
"use strict";

function foo() {
    // this code is strict mode

    function bar() {
        // this code is strict mode
    }
}

// this code is strict mode
```

One key difference (improvement!) with strict mode is disallowing the implicit auto-global variable declaration from omitting the var:

```
function foo() {
    "use strict";   // turn on strict mode
    a = 1;          // `var` missing, ReferenceError
}

foo();
```

If you turn on strict mode in your code, and you get errors, or code starts behaving buggy, your temptation might be to avoid strict mode. But that instinct would be a bad idea to indulge. If strict mode causes issues in your program, almost certainly it's a sign that you have things in your program you should fix.

Not only will strict mode keep your code to a safer path, and not only will it make your code more optimizable, but it also represents the future direction of the language. It'd be easier on you to get used to strict mode now than to keep putting it off -- it'll only get harder to convert later!

Functions As Values

So far, we've discussed functions as the primary mechanism of scope in JavaScript. You recall typical function declaration syntax as follows:

```
function foo() {
    // ..
}
```

Though it may not seem obvious from that syntax, foo is basically just a variable in the outer enclosing scope that's given a reference to the function being declared. That is, the function itself is a value, just like 42 or [1,2,3] would be.

This may sound like a strange concept at first, so take a moment to ponder it. Not only can you pass a value (argument) to a function, but a function itself can be a value that's assigned to variables, or passed to or returned from other functions.

As such, a function value should be thought of as an expression, much like any other value or expression.

```
var foo = function() {
    // ..
};

var x = function bar(){
    // ..
};
```

The first function expression assigned to the foo variable is called anonymous because it has no name.

The second function expression is named (bar), even as a reference to it is also assigned to the x variable. Named function expressions are generally more preferable, though anonymous function expressions are still extremely common.

Immediately Invoked Function Expressions (IIFEs)

In the previous snippet, neither of the function expressions are executed -- we could if we had included foo() or x(), for instance.

There's another way to execute a function expression, which is typically referred to as an immediately invoked function expression (IIFE):

```
(function IIFE(){

    console.log( "Hello!" );

})();

// "Hello!"
```

The outer (..) that surrounds the (function IIFE(){ .. }) function expression is just a nuance of JS grammar needed to prevent it from being treated as a normal function declaration.

The final () on the end of the expression -- the })(); line -- is what actually executes the function expression referenced immediately before it.

That may seem strange, but it's not as foreign as first glance. Consider the similarities between foo and IIFE here:

```
function foo() { .. }

// `foo` function reference
expression,
// then `()` executes it
foo();

// `IIFE` function expression,
// then `()` executes it
(function IIFE(){ .. })();
```

As you can see, listing the (function IIFE(){ .. }) before its executing () is essentially the same as including foo before its executing ();

in both cases, the function reference is executed with () immediately after it.

Continues below… **V V V**

Because an IIFE is just a function, and functions create variable scope, using an IIFE in this fashion is often used to declare variables that won't affect the surrounding code outside the IIFE:

```
var a = 42;

(function IIFE(){
    var a = 10;
    console.log( a );   // 10
})();

console.log( a );       // 42
```

IIFEs can also have return values:

```
var x = (function IIFE(){
    return 42;
})();

x;  // 42
```

The 42 value gets returned from the IIFE-named function being executed, and is then assigned to x.

Closure

Closure is one of the most important, and often least understood, concepts in JavaScript. I won't cover it in deep detail here, and instead refer you to the Scope & Closures title of this series. But I want to say a few things about it so you understand the general concept. It will be one of the most important techniques in your JS skillset.

You can think of closure as a way to "remember" and continue to access a function's scope (its variables) even once the function has finished running.

Consider:

```
function makeAdder(x) {
    // parameter `x` is an inner variable

    // inner function `add()` uses `x`, so
    // it has a "closure" over it
    function add(y) {
        return y + x;
    };

    return add;
}
```

The reference to the inner add(..) function that gets returned with each call to the outer makeAdder(..) is able to remember whatever x value was passed in to makeAdder(..). Now, let's use makeAdder(..):

```
// `plusOne` gets a reference to the inner `add(..)` function with closure over the `x`
parameter of the outer `makeAdder(..)`
var plusOne = makeAdder( 1 );

// `plusTen` gets a reference to the inner `add(..)` function with closure over the `x`
parameter of the outer `makeAdder(..)`
var plusTen = makeAdder( 10 );

plusOne( 3 );       // 4  <-- 1 + 3
plusOne( 41 );      // 42 <-- 1 + 41

plusTen( 13 );      // 23 <-- 10 + 13
```

More on how this code works:

1. When we call makeAdder(1), we get back a reference to its inner add(..) that remembers x as 1. We call this function reference plusOne(..).
2. When we call makeAdder(10), we get back another reference to its inner add(..) that remembers x as 10. We call this function reference plusTen(..).
3. When we call plusOne(3), it adds 3 (its inner y) to the 1 (remembered by x), and we get 4 as the result.
4. When we call plusTen(13), it adds 13 (its inner y) to the 10 (remembered by x), and we get 23 as the result.

Don't worry if this seems strange and confusing at first -- it can be! It'll take lots of practice to understand it fully.

But trust me, once you do, it's one of the most powerful and useful techniques in all of programming. It's definitely worth the effort to let your brain simmer on closures for a bit. In the next section, we'll get a little more practice with closure.

Closure - Stackoverflow Explanation

http://stackoverflow.com/questions/111102/how-do-javascript-closures-work

Whenever you see the function keyword within another function, the inner function has access to variables in the outer function i.e. it has closure over them.

```
function foo(x) {
  var tmp = 3;

  function bar(y) {
    console.log(x + y + (++tmp)); // will log 16
  }

  bar(10);
}

foo(2);
```

This will always log 16, because bar can access the x which was defined as an argument to foo, and it can also access tmp from foo.

That **is** a closure. A function doesn't have to return in order to be called a closure. **Simply accessing variables outside of your immediate lexical scope creates a closure.**

```
function foo(x) {
  var tmp = 3;

  return function (y) {
    console.log(x + y + (++tmp)); // will also log 16
  }
}

var bar = foo(2); // bar is now a closure.
bar(10);
```

The above function will also log 16, because bar can still refer to x and tmp, even though it is no longer directly inside the scope.

However, since tmp is still hanging around inside bar's closure, it is also being incremented. It will be incremented each time you call bar.

The simplest example of a closure is this:

```
var a = 10;
function test() {
  console.log(a); // will output 10
  console.log(b); // will output 6
}
var b = 6;
test();
```

When a JavaScript function is invoked, a new execution context is created. Together with the function arguments and the parent object, this execution context also receives all the variables declared outside of it (in the above example, both 'a' and 'b').

It is possible to create more than one closure function, either by returning a list of them or by setting them to global variables. All of these will refer to the **same** x and the same tmp, they don't make their own copies.

Here the number x is a literal number. As with other literals in JavaScript, when foo is called, the number x is **copied** into foo as its argument x.

On the other hand, JavaScript always uses references when dealing with objects. If say, you called foo with an object, the closure it returns will **reference** that original object!

```
function foo(x) {
  var tmp = 3;

  return function (y) {
    console.log(x + y + tmp);
    x.memb = x.memb ? x.memb + 1 : 1;
    console.log(x.memb);
  }
}

var age = new Number(2);
var bar = foo(age); // bar is now a closure referencing age.
bar(10);
```

As expected, each call to bar(10) will increment x.memb. What might not be expected, is that x is simply referring to the same object as the age variable! After a couple of calls to bar, age.memb will be 2! This referencing is the basis for memory leaks with HTML objects.

Another very interesting explanation.

I'm a big fan of analogy and metaphor when explaining difficult concepts, so let me try my hand with a story.

Once upon a time:

There was a princess...

```
function princess() {
```

She lived in a wonderful world full of adventures. She met her Prince Charming, rode around her world on a unicorn, battled dragons, encountered talking animals, and many other fantastical things.

```
    var adventures = [];

    function princeCharming() { /* ... */ }

    var unicorn = { /* ... */ },
        dragons = [ /* ... */ ],
        squirrel = "Hello!";
```

But she would always have to return back to her dull world of chores and grown-ups.

```
    return {
```

And she would often tell them of her latest amazing adventure as a princess.

```
        story: function() {
            return adventures[adventures.length - 1];
        }
    };
}
```

But all they would see is a little girl...

```
var littleGirl = princess();
```

...telling stories about magic and fantasy.

```
littleGirl.story();
```

And even though the grown-ups knew of real princesses, they would never believe in the unicorns or dragons because they could never see them. The grown-ups said that they only existed inside the little girl's imagination.

But we know the real truth; that the little girl with the princess inside...

...is really a princess with a little girl inside.

Comment: I love this explanation, truly. For those who read it and don't follow, the analogy is this: the princess() function is a complex scope containing private data. Outside the function, the private data can't be seen or accessed. The princess keeps the unicorns, dragons, adventures etc. in her imagination (private data) and the grown-ups can't see them for themselves. BUT the princess's imagination is captured in the closure for the story() function, which is the only interface the littleGirl instance exposes into the world of magic.

Modules

The most common usage of closure in JavaScript is the module pattern. Modules let you define private implementation details (variables, functions) that are hidden from the outside world, as well as a public API that is accessible from the outside.

```
function User(){
    var username, password;

    function doLogin(user,pw) {
        username = user;
        password = pw;

        // do the rest of the login work
    }

    var publicAPI = {
        login: doLogin
    };

    return publicAPI;
}

// create a `User` module instance
var fred = User();

fred.login( "fred", "12Battery34!" );
```

The User() function serves as an outer scope that holds the variables username and password, as well as the inner doLogin() function; these are all private inner details of this User module that cannot be accessed from the outside world.

Warning: We are not calling **new** User() here, on purpose, despite the fact that probably seems more common to most readers. User() is just a function, not a class to be instantiated, so it's just called normally. Using new would be inappropriate and actually waste resources.

Executing User() creates an instance of the User module -- a whole new scope is created, and thus a whole new copy of each of these inner variables/functions. We assign this instance to fred. If we run User() again, we'd get a new instance entirely separate from fred.

The inner doLogin() function has a closure over username and password, meaning it will retain its access to them even after the User() function finishes running.

publicAPI is an object with one property/method on it, login, which is a reference to the inner doLogin() function. **When we return publicAPI from User(), it becomes the instance we call fred**.

At this point, the outer User() function has finished executing. Normally, you'd think the inner variables like username and password have gone away. But here they have not, because there's a closure in the login() function keeping them alive.

That's why we can call fred.login(..) -- the same as calling the inner doLogin(..) -- and it can still access username and password inner variables.

There's a good chance that with just this brief glimpse at closure and the module pattern, some of it is still a bit confusing. That's OK! It takes some work to wrap your brain around it.

this Identifier

Another very commonly misunderstood concept in JavaScript is the this identifier. Again, there's a couple of chapters on it in the this & Object Prototypes title of this series, so here we'll just briefly introduce the concept.

While it may often seem that this is related to "object-oriented patterns," in JS this is a different mechanism.

If a function has a **this** reference inside it, that **this** reference **usually points to an object**. But which object it points to depends on how the function was called.

It's important to realize that **this** does not refer to the function itself, as is the most common misconception.

Here's a quick illustration:

```
function foo() {
    console.log( this.bar );
}

var bar = "global";

var obj1 = {
    bar: "obj1",
    foo: foo
};

var obj2 = {
    bar: "obj2"
};

// --------

foo();              // "global"
obj1.foo();         // "obj1"
foo.call( obj2 );   // "obj2"
new foo();          // undefined
```

There are four rules for how this gets set, and they're shown in those last four lines of that snippet.

1. foo() ends up setting this to the global object in non-strict mode -- in strict mode, this would be undefined and you'd get an error in accessing the bar property -- so "global" is the value found for this.bar.

2. obj1.foo() sets this to the obj1 object.

3. foo.call(obj2) sets this to the obj2 object.

4. new foo() sets this to a brand new empty object.

Bottom line: to understand what this points to, you have to examine how the function in question was called.

It will be one of those four ways just shown, and that will then answer what this is.

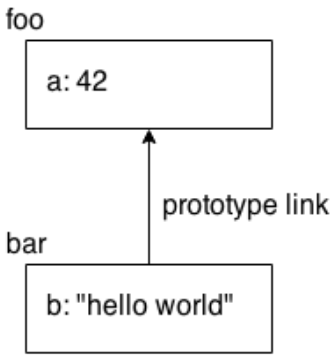
Prototypes

The prototype mechanism in JavaScript is quite complicated. We will only glance at it here. You will want to spend plenty of time reviewing Chapters 4-6 of the this & Object Prototypes title of this series for all the details.

When you reference a property on an object, if that property doesn't exist, JavaScript will automatically use that object's internal prototype reference to find another object to look for the property on. You could think of this almost as a fallback if the property is missing.

The internal prototype reference linkage from one object to its fallback happens at the time the object is created. The simplest way to illustrate it is with a built-in utility called Object.create(..).

```
var foo = {
    a: 42
};

// create `bar` and link it to `foo`
var bar = Object.create( foo );

bar.b = "hello world";

bar.b;      // "hello world"
bar.a;      // 42 <-- delegated to `foo`
```



The **a** property doesn't actually exist on the **bar** object, but because **bar** is prototype-linked to **foo**, JavaScript automatically falls back to looking for a on the **foo** object, where it's found.

This linkage may seem like a strange feature of the language. The most common way this feature is used -- and I would argue, abused -- is to try to emulate/fake a "class" mechanism with "inheritance."

But a more natural way of applying prototypes is a pattern called "behavior delegation," where you intentionally design your linked objects to be able to delegate from one to the other for parts of the needed behavior.

You Don’t Know JS - Scope & Closures

Encapsulation = Scope

One of the most fundamental paradigms of nearly all programming languages is the ability to store values in variables, and later retrieve or modify those values. In fact, the ability to store values and pull values out of variables is what gives a program state.

ReferenceError is Scope resolution-failure related, whereas TypeError implies that Scope resolution was successful, but that there was an illegal/impossible action attempted against the result.

Scope is the set of rules that determines where and how a variable (identifier) can be looked-up. This look-up may be for the purposes of assigning to the variable, which is an LHS (left-hand-side) reference, or it may be for the purposes of retrieving its value, which is an RHS (right-hand-side) reference.

LHS references result from assignment operations. Scope-related assignments can occur either with the = operator or by passing arguments to (assign to) function parameters.

The JavaScript Engine first compiles code before it executes, and in so doing, it splits up statements like var a = 2; into two separate steps:

First, var a to declare it in that Scope. This is performed at the beginning, before code execution.

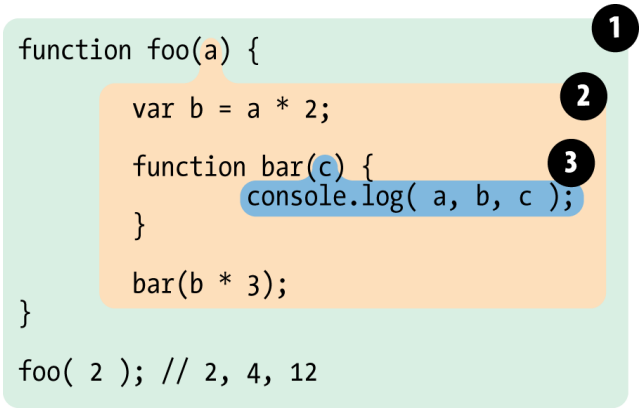
Later, a = 2 to look up the variable (LHS reference) and assign to it if found.

Both LHS and RHS reference look-ups start at the currently executing Scope, and if need be (that is, they don't find what they're looking for there), they work their way up the nested Scope, one scope (floor) at a time, looking for the identifier, until they get to the global (top floor) and stop, and either find it, or don't.

Unfulfilled RHS references result in ReferenceErrors being thrown. Unfulfilled LHS references result in an automatic, implicitly-created global of that name (if not in "Strict Mode" [^note-strictmode]), or a ReferenceError (if in "Strict Mode" [^note-strictmode]).

There are three nested scopes inherent in this code example. It may be helpful to think about these scopes as bubbles inside of each other.

```
function foo(a) {

    var b = a * 2;

    function bar(c) {
        console.log( a, b, c );
    }

    bar(b * 3);
}

foo( 2 ); // 2 4 12
```



Bubble 1 encompasses the global scope, and has just one identifier in it: foo.

Bubble 2 encompasses the scope of foo, which includes the three identifiers: a, bar and b.

Bubble 3 encompasses the scope of bar, and it includes just one identifier: c.

Scope bubbles are defined by where the blocks of scope are written, which one is nested inside the other, etc. In the next chapter, we'll discuss different units of scope, but for now, let's just assume that each function creates a new bubble of scope.

The bubble for bar is entirely contained within the bubble for foo, because (and only because) that's where we chose to define the function bar.

Notice that these nested bubbles are strictly nested. We're not talking about Venn diagrams where the bubbles can cross boundaries. In other words, no bubble for some function can simultaneously exist (partially) inside two other outer scope bubbles, just as no function can partially be inside each of two parent functions.

Had there been a c both inside of bar(..) and inside of foo(..), the console.log(..) statement would have found and used the one in bar(..), never getting to the one in foo(..).

No matter where a function is invoked from, or even how it is invoked, its lexical scope is only defined by where the function was declared.

Lex-Time and Avoiding eval() and with

Lexical scope means that scope is defined by author-time decisions of where functions are declared. The lexing phase of compilation is essentially able to know where and how all identifiers are declared, and thus predict how they will be looked-up during execution.

Two mechanisms in JavaScript can "cheat" lexical scope: eval(..) and with. The former can modify existing lexical scope (at runtime) by evaluating a string of "code" which has one or more declarations in it. The latter essentially creates a whole new lexical scope (again, at runtime) by treating an object reference as a "scope" and that object's properties as scoped identifiers.

The downside to these mechanisms is that it defeats the Engine's ability to perform compile-time optimizations regarding scope look-up, because the Engine has to assume pessimistically that such optimizations will be invalid. Code will run slower as a result of using either feature. Don't use them.

Hiding i.e. Scoping

```
function doSomething(a) {
    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

function doSomethingElse(a) {
    return a - 1;
}

var b;

doSomething( 2 ); // 15
```

In this snippet, the b variable and the doSomethingElse(..) function are likely "private" details of how doSomething(..) does its job.

Giving the enclosing scope "access" to b and doSomethingElse(..) is not only unnecessary but also possibly "dangerous", in that they may be used in unexpected ways, intentionally or not, and this may violate pre-condition assumptions of doSomething(..).

A more "proper" design would hide these private details inside the scope of doSomething(..), such as:

```
function doSomething(a) {
    function doSomethingElse(a) {
        return a - 1;
    }

    var b;

    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

doSomething( 2 ); // 15
```

Now, b and doSomethingElse(..) are not accessible to any outside influence, instead controlled only by doSomething(..). The functionality and end-result has not been affected, but the design keeps private details private, which is usually considered better software.

Collision Avoidance

Another benefit of "hiding" variables and functions inside a scope is to avoid unintended collision between two different identifiers with the same name but different intended usages. Collision results often in unexpected overwriting of values.

```
function foo() {
    function bar(a) {
        i = 3; // changing the `i` in the enclosing scope's for-loop
        console.log( a + i );
    }

    for (var i=0; i<10; i++) {
        bar( i * 2 ); // oops, infinite loop ahead!
    }
}

foo();
```

The i = 3 assignment inside of bar(..) overwrites, unexpectedly, the i that was declared in foo(..) at the for-loop. In this case, it will result in an infinite loop, because i is set to a fixed value of 3 and that will forever remain < 10.

The assignment inside bar(..) needs to declare a local variable to use, regardless of what identifier name is chosen. var i = 3; would fix the problem (and would create the previously mentioned "shadowed variable" declaration for i). An additional, not alternate, option is to pick another identifier name entirely, such as var j = 3;. But your software design may naturally call for the same identifier name, so utilizing scope to "hide" your inner declaration is your best/only option in that case.

Global "Namespaces"

A particularly strong example of (likely) variable collision occurs in the global scope. Multiple libraries loaded into your program can quite easily collide with each other if they don't properly hide their internal/private functions and variables.

Such libraries typically will create a single variable declaration, often an object, with a sufficiently unique name, in the global scope. This object is then used as a "namespace" for that library, where all specific

exposures of functionality are made as properties off that object (namespace), rather than as top-level lexically scoped identifiers themselves.

For example:

```
var MyReallyCoolLibrary = {
    awesome: "stuff",
    doSomething: function() {
        // ...
    },
    doAnotherThing: function() {
        // ...
    }
};
```

Function Declarations vs Expressions, Anonymous Functions, IIFEs and Callbacks

Functions As Scopes

We've seen that we can take any snippet of code and wrap a function around it, and that effectively "hides" any enclosed variable or function declarations from the outside scope inside that function's inner scope.

```
var a = 2;

function foo() { // <-- insert this

    var a = 3;
    console.log( a ); // 3

} // <-- and this
foo(); // <-- and this

console.log( a ); // 2
```

While this technique "works", it is not necessarily very ideal. There are a few problems it introduces. The first is that we have to declare a named-function foo(), which means that the identifier name foo itself "pollutes" the enclosing scope (global, in this case).

We also have to explicitly call the function by name (foo()) so that the wrapped code actually executes.

It would be more ideal if the function didn't need a name (or, rather, the name didn't pollute the enclosing scope), and if the function could automatically be executed.

Fortunately, JavaScript offers a solution to both problems.

```
var a = 2;

(function foo(){ // <-- insert this

    var a = 3;
    console.log( a ); // 3

})(); // <-- and this

console.log( a ); // 2
```

Let's break down what's happening here.

First, notice that the wrapping function statement starts with (function... as opposed to just function....

While this may seem like a minor detail, it's actually a major change. Instead of treating the function as a standard declaration, the function is treated as a function-expression.

Note: The easiest way to distinguish declaration vs. expression is the position of the word "function" in the statement (not just a line, but a distinct statement). If "function" is the very first thing in the statement, then it's a function declaration. Otherwise, it's a function expression.

The key difference we can observe here between a function declaration and a function expression relates to where its name is bound as an identifier.

Compare the previous two snippets. In the first snippet, the name foo is bound in the enclosing scope, and we call it directly with foo(). In the second snippet, the name foo is not bound in the enclosing scope, but instead is bound only inside of its own function.

In other words, (function foo(){ .. }) as an expression means the identifier foo is found only in the scope where the .. indicates, not in the outer scope. Hiding the name foo inside itself means it does not pollute the enclosing scope unnecessarily.

Anonymous vs. Named

You are probably most familiar with function expressions as callback parameters, such as:

```
setTimeout( function(){
    console.log("I waited 1 second!");
}, 1000 );
```

This is called an "anonymous function expression", because function()... has no name identifier on it. Function expressions can be anonymous, but function declarations cannot omit the name -- that would be illegal JS grammar.

Anonymous function expressions are quick and easy to type, and many libraries and tools tend to encourage this idiomatic style of code. However, they have several draw-backs to consider:

Anonymous functions have no useful name to display in stack traces, which can make debugging more difficult. Without a name, if the function needs to refer to itself, for recursion, etc., the deprecated arguments.callee reference is unfortunately required. Another example of needing to self-reference is when an event handler function wants to unbind itself after it fires.

Anonymous functions omit a name that is often helpful in providing more readable/understandable code. A descriptive name helps self-document the code in question.

Inline function expressions are powerful and useful -- the question of anonymous vs. named doesn't detract from that. **Providing a name for your function expression quite effectively addresses all these draw-backs, but has no tangible downsides. The best practice is to always name your function expressions:**

```
setTimeout( function timeoutHandler(){ // <-- Look, I have a name!
    console.log( "I waited 1 second!" );
}, 1000 );
```

Invoking Function Expressions Immediately

```
var a = 2;

(function foo(){

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

Now that we have a function as an expression by virtue of wrapping it in a ( ) pair, we can execute that function by adding another () on the end, like (function foo(){ .. })(). The first enclosing ( ) pair makes the function an expression, and the second () executes the function.

This pattern is so common, a few years ago the community agreed on a term for it: IIFE, which stands for Immediately Invoked Function Expression.

Of course, IIFE's don't need names, necessarily -- the most common form of IIFE is to use an anonymous function expression. While certainly less common, naming an IIFE has all the aforementioned benefits over anonymous function expressions, so it's a good practice to adopt.

```
var a = 2;

(function IIFE(){

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

There's a slight variation on the traditional IIFE form, which some prefer: (function(){ .. }()).

Look closely to see the difference. In the first form, the function expression is wrapped in ( ), and then the invoking () pair is on the outside right after it.

In the second form, the invoking () pair is moved to the inside of the outer ( ) wrapping pair.

These two forms are identical in functionality. It's purely a stylistic choice which you prefer.

Another variation on IIFE's which is quite common is to use the fact that they are, in fact, just function calls, and pass in argument(s).

```
var a = 2;

(function IIFE( global ){

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

})( window );

console.log( a ); // 2
```

We pass in the window object reference, but we name the parameter global, so that we have a clear stylistic delineation for global vs. non-global references.

Of course, you can pass in anything from an enclosing scope you want, and you can name the parameter(s) anything that suits you. This is mostly just stylistic choice.

Another application of this pattern addresses the (minor niche) concern that the default undefined identifier might have its value incorrectly overwritten, causing unexpected results. By naming a parameter undefined, but not passing any value for that argument, we can guarantee that the undefined identifier is in fact the undefined value in a block of code:

```
undefined = true; // setting a land-mine for other
code! avoid!

(function IIFE( undefined ){

    var a;
    if (a === undefined) {
        console.log( "Undefined is safe here!" );
    }

})();
```

Still another variation of the IIFE inverts the order of things, where the function to execute is given second, after the invocation and parameters to pass to it.

This pattern is used in the UMD (Universal Module Definition) project. Some people find it a little cleaner to understand, though it is slightly more verbose.

```
var a = 2;

(function IIFE( def ){
    def( window );
})(function def( global ){

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

});
```

The def function expression is defined in the second-half of the snippet, and then passed as a parameter (also called def) to the IIFE function defined in the first half of the snippet.

Finally, the parameter def (the function) is invoked, passing window in as the global parameter.

You Don’t Know JS - Async

The relationship between the now and later parts of your program is at the heart of asynchronous programming.

Despite clearly allowing asynchronous JS code (like the timeout we just looked at), up until recently (ES6), JavaScript itself has actually never had any direct notion of asynchrony built into it.

It is done via an event loop i.e. a continuously running loop represented by the while loop, and each iteration of this loop is called a "tick." For each tick, if an event is waiting on the queue, it's taken off and executed. These events are your function callbacks.

It's very common to conflate the terms "async" and "parallel," but they are actually quite different. Remember, async is about the gap between now and later. But parallel is about things being able to occur simultaneously.

Concurrency is when two or more "processes" are executing simultaneously over the same period, regardless of whether their individual constituent operations happen in parallel (at the same instant on separate processors or cores) or not. You can think of concurrency then as "process"-level (or task-level) parallelism, as opposed to operation-level parallelism (separate-processor threads).

The **event loop** queue is like an amusement park ride, where once you finish the ride, you have to go to the back of the line to ride again. But the **Job queue** is like finishing the ride, but then cutting in line and getting right back on.

A "callback," serves as the target for the event loop to "call back into" the program, whenever that item in the queue is processed.

As you no doubt have observed, callbacks are by far the most common way that asynchrony in JS programs is expressed and managed. Indeed, the callback is the most fundamental async pattern in the language.

```
// A
setTimeout( function(){
    // C
}, 1000 );
// B
```

Stop for a moment and ask yourself how you'd describe (to someone else less informed about how JS works) the way that program behaves. Go ahead, try it out loud. It's a good exercise that will help my next points make more sense.

Most readers just now probably thought or said something to the effect of: "Do A, then set up a timeout to wait 1,000 milliseconds, then once that fires, do C." How close was your rendition?

You might have caught yourself and self-edited to: "Do A, setup the timeout for 1,000 milliseconds, then do B, then after the timeout fires, do C." That's more accurate than the first version. Can you spot the difference?

Even though the second version is more accurate, both versions are deficient in explaining this code in a way that matches our brains to the code, and the code to the JS engine. The disconnect is both subtle and monumental, and is at the very heart of understanding the shortcomings of callbacks as async expression and management.

As soon as we introduce a single continuation (or several dozen as many programs do!) in the form of a callback function, we have allowed a divergence to form between how our brains work and the way the code will operate. Any time these two diverge (and this is by far not the only place that happens, as I'm sure you know!), we run into the inevitable fact that our code becomes harder to understand, reason about, debug, and maintain.

So if synchronous brain planning maps well to synchronous code statements, how well do our brains do at planning out asynchronous code?

It turns out that how we express asynchrony (with callbacks) in our code doesn't map very well at all to that synchronous brain planning behavior.

Can you actually imagine having a line of thinking that plans out your to-do errands like this?

"I need to go to the store, but on the way I'm sure I'll get a phone call, so 'Hi, Mom', and while she starts talking, I'll be looking up the store address on GPS, but that'll take a second to load, so I'll turn down the radio so I can hear Mom better, then I'll realize I forgot to put on a jacket and it's cold outside, but no matter, keep driving and talking to Mom, and then the seatbelt ding reminds me to buckle up, so 'Yes, Mom, I am wearing my seatbelt, I always do!'. Ah, finally the GPS got the directions, now..."

As ridiculous as that sounds as a formulation for how we plan our day out and think about what to do and in what order, nonetheless it's exactly how our brains operate at a functional level. Remember, that's not multitasking, it's just fast context switching.

The reason it's difficult for us as developers to write async evented code, especially when all we have is the callback to do it, is that stream of consciousness thinking/planning is unnatural for most of us.

We think in step-by-step terms, but the tools (callbacks) available to us in code are not expressed in a step-by-step fashion once we move from synchronous to asynchronous.

And that is why it's so hard to accurately author and reason about async JS code with callbacks: because it's not how our brain planning works.

"Inversion of control" is when you take part of your program and give over control of its execution to another third party.

Callback Hell Story

It might not be terribly obvious why this is such a big deal. Let me construct an exaggerated scenario to illustrate the hazards of trust at play.

Imagine you're a developer tasked with building out an ecommerce checkout system for a site that sells expensive TVs. You already have all the various pages of the checkout system built out just fine. On the last page, when the user clicks "confirm" to buy the TV, you need to call a third-party function (provided say by some analytics tracking company) so that the sale can be tracked.

You notice that they've provided what looks like an async tracking utility, probably for the sake of performance best practices, which means you need to pass in a callback function. In this continuation that you pass in, you will have the final code that charges the customer's credit card and displays the thank you page.

This code might look like:

```
analytics.trackPurchase( purchaseData, function(){
    chargeCreditCard();
    displayThankyouPage();
} );
```

Easy enough, right? You write the code, test it, everything works, and you deploy to production. Everyone's happy!

Six months go by and no issues. You've almost forgotten you even wrote that code. One morning, you're at a coffee shop before work, casually enjoying your latte, when you get a panicked call from your boss insisting you drop the coffee and rush into work right away.

When you arrive, you find out that a high-profile customer has had his credit card charged five times for the same TV, and he's understandably upset. Customer service has already issued an apology and processed a refund. But your boss demands to know how this could possibly have happened. "Don't we have tests for stuff like this!?"

You don't even remember the code you wrote. But you dig back in and start trying to find out what could have gone awry.

After digging through some logs, you come to the conclusion that the only explanation is that the analytics utility somehow, for some reason, called your callback five times instead of once. Nothing in their documentation mentions anything about this.

Frustrated, you contact customer support, who of course is as astonished as you are. They agree to escalate it to their developers, and promise to get back to you. The next day, you receive a lengthy email explaining what they found, which you promptly forward to your boss.

Apparently, the developers at the analytics company had been working on some experimental code that, under certain conditions, would retry the provided callback once per second, for five seconds, before failing with a timeout. They had never intended to push that into production, but somehow they did, and they're totally embarrassed and apologetic. They go into plenty of detail about how they've identified the breakdown and what they'll do to ensure it never happens again. Yadda, yadda.

What's next?

You talk it over with your boss, but he's not feeling particularly comfortable with the state of things. He insists, and you reluctantly agree, that you can't trust *them* anymore (that's what bit you), and that you'll need to figure out how to protect the checkout code from such a vulnerability again.

After some tinkering, you implement some simple ad hoc code like the following, which the team seems happy with:

```
var tracked = false;
analytics.trackPurchase( purchaseData,
function(){
    if (!tracked) {
        tracked = true;
        chargeCreditCard();
        displayThankyouPage();
    }
} );
```

Note: This should look familiar to you from Chapter 1, because we're essentially creating a latch to handle if there happen to be multiple concurrent invocations of our callback.

But then one of your QA engineers asks, "what happens if they never call the callback?" Oops. Neither of you had thought about that.

You begin to chase down the rabbit hole, and think of all the possible things that could go wrong with them calling your callback. Here's roughly the list you come up with of ways the analytics utility could misbehave:

- Call the callback too early (before it's been tracked)
- Call the callback too late (or never)
- Call the callback too few or too many times (like the problem you encountered!)
- Fail to pass along any necessary environment/parameters to your callback
- Swallow any errors/exceptions that may happen
- ...

That should feel like a troubling list, because it is. You're probably slowly starting to realize that you're going to have to invent an awful lot of ad hoc logic **in each and every single callback** that's passed to a utility you're not positive you can trust.
Now you realize a bit more completely just how hellish "callback hell" is.

Promises

Recall that we wrap up the continuation of our program in a callback function, and hand that callback over to another party (potentially even external code) and just cross our fingers that it will do the right thing with the invocation of the callback.

We do this because we want to say, "here's what happens later, after the current step finishes."

But what if we could uninvert that inversion of control? What if instead of handing the continuation of our program to another party, we could expect it to return us a capability to know when its task finishes, and then our code could decide what to do next?

This paradigm is called **Promises**.

…

Promises don't get rid of callbacks at all. They just change where the callback is passed to. Instead of passing a callback to foo(..), we get something (ostensibly a genuine Promise) back from foo(..), and we pass the callback to that something instead.

we can string multiple Promises together to represent a sequence of async steps.

The key to making this work is built on two behaviors intrinsic to Promises:

- Every time you call then(..) on a Promise, it creates and returns a new Promise, which we can chain with.
- Whatever value you return from the then(..) call's fulfillment callback (the first parameter) is automatically set as the fulfillment of the chained Promise (from the first point).

```
var p = Promise.resolve( 21 );

var p2 = p.then( function(v){
    console.log( v ); // 21
    // fulfill `p2` with value `42`
    return v * 2;
} );

// chain off `p2`
p2.then( function(v){
    console.log( v ); // 42
} );
```

By returning v * 2 (i.e., 42), we fulfill the p2 promise that the first then(..) call created and returned.

When p2'sthen(..) call runs, it's receiving the fulfillment from the return v * 2 statement.

Of course, p2.then(..) creates yet another promise, which we could have stored in a p3 variable.

But it's a little annoying to have to create an intermediate variable p2 (or p3, etc.). Thankfully, we can easily just chain these together:

```
var p = Promise.resolve( 21 );
p
.then( function(v){
    console.log( v ); // 21
    // fulfill the chained promise with value `42`
    return v * 2;
} )
// here's the chained promise
.then( function(v){
    console.log( v ); // 42
} );
```

Resolve and Reject

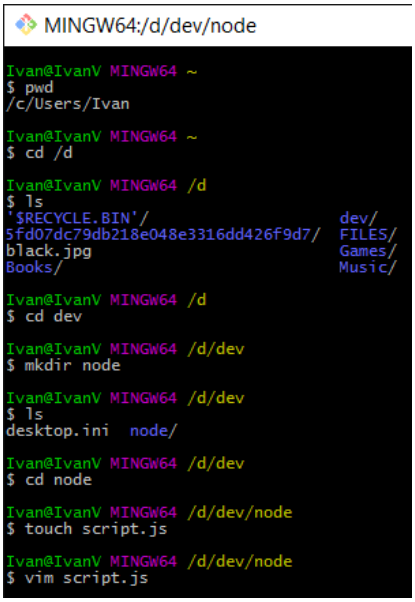
This is the naming convention for the two parameters a promise takes.

```
foo.then(
    function resolve(){
        // If successful, do this.
    },
    function rejecte(err){
        // If failed, do this.
    }
);
```

Node

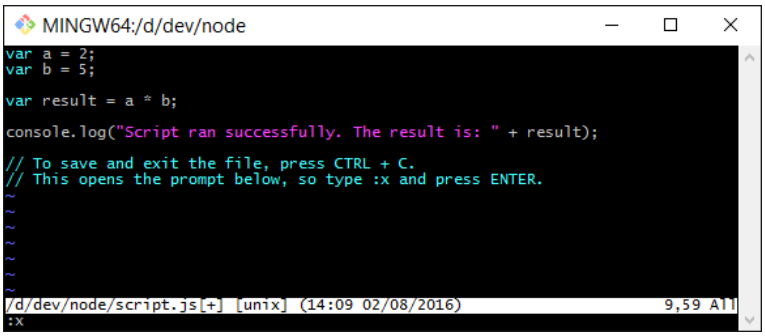
Node is isntalled on the computer and then ran in a terminal. Chrome uses the V8 engine (C++) to interpret Javascript in the browser, and Node uses the same engine for the same job on the desktop.

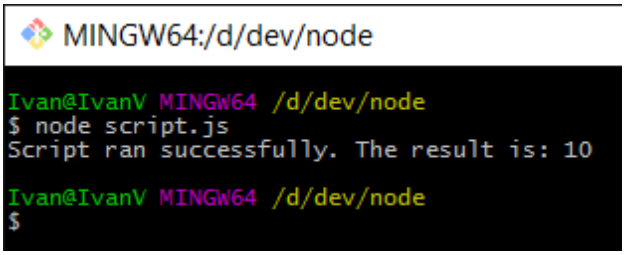
Example Workflow



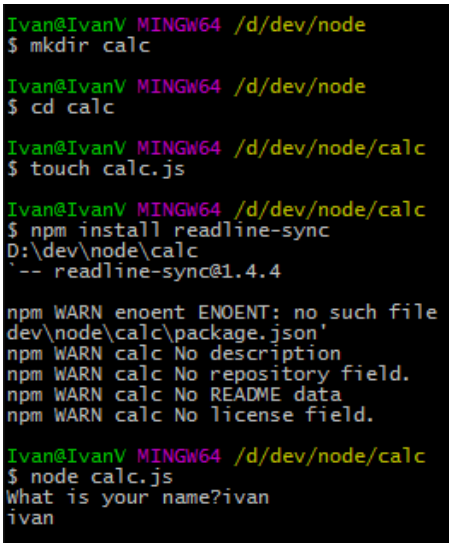
Using BASH for Windows (Unix commands):

1. **pwd** – Show current path.
2. **cd /d** – Change to the **D:** directory.
3. **ls** – Show directory (**D:**) content.
4. **cd dev** – Navigate inside the dev folder.
5. **mkdir node** – Create a **node** folder inside the current directory (**dev**).
6. **ls** – Show directory (**dev**) content.
7. **cd node** – Navigate inside the node folder.
8. **touch script.js** – Create a javascript file with the given name.
9. **vim script.js** – Open and edit in the VIM text editor.
10. Write a simple javascript program.
11. Press **CTRL + C** to go to the VIM prompt and type **:x** to save and exit the file.
12. **node script.js** – Run the javascript program inside the file.

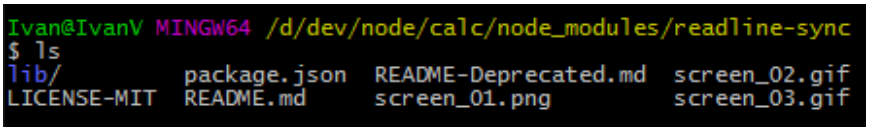




Install NPM Package



Simply navigate to the destination folder and type **npm install PACKAGE_NAME**. This will create a node_modules folder with the readline-sync package inside.



After that, the library can be used to write a script using a prompt as an input. Below is the example script and the result.

```
var readline = require("readline-sync");
var name = readline.question("What is your name?");
console.log(name);
```

Modules Export - Import

Anything can be exported as a module.

app.js

```
var print = require("./print");

print("Exported module works!");
```

print.js

```
var print = function (thingToPrint) {
    console.log(thingToPrint);
}

module.exports = print;
```

```
Ivan@IvanV MINGW64 /d/dev/node/udemy-react/node-modules
$ node app.js
Exported module works!
```

Calculator Module Example

app.js

```
var calculator = require("./calculator");

var a = 10;
var b = 5;

calculator.add(a, b);
calculator.subtract(a, b);
calculator.multiply(a, b);
calculator.divide(a, b);
```

calculator.js

```
module.exports = {
    add: add,
    subtract: subtract,
    multiply: multiply,
    divide: divide
}

function add(a, b) { console.log(a + b); }
function subtract(a, b) { console.log(a - b); }
function multiply(a, b) { console.log(a * b); }
function divide(a, b) { console.log(a / b); }
```

```
Ivan@IvanV MINGW64 /d/dev/node/calculator
$ node app.js
15
5
50
2
```

Importing Modules

require(“./Component.jsx”);

./ = Current folder.
../ - Up one folder.
../../ - Up two folders.

React - Udemy - React JS and Flux Web Development for Beginners

Required NPM Packages

Browserify – Allows the browser to use NPM modules by taking them all and putting them into one file, to be used by the Front-End.
Babel – Compiles ES6/JSX code into browser-readable Javascript ES5.
Babel-React-Preset – Tells **Babel** how to compile JSX.
Babelify – Allows you to use **Babel** with **Browserify**.
Watchify – Watches for changes and instantly starts compiling, avoiding having to type node compiling commands after each change.
React and **ReactDOM** – The actual libraries.

Replace Gulp/Grunt and Browserify with Webpack

React Skeleton/Seed Project

This is used in order to avoid having to set up the same things all over again for each new project.

- **npm install –g PACKAGE** – Installs it globally
- **npm install --save PACKAGE** – Installs it in the current directory and adds a dependency to **package.json** in the current **node project** created by **git init**.

mkdir react-skeleton
cd react-skeleton

git init
git remote add origin GITHUB_URL
touch README.md
git status
git add .
git status
git commit –m “initial commit”
git push origin master

npm install –g browserify
npm install –-save babelify
npm install –-save watchify
npm install –-save babel-preset-react
npm install –-save react
npm install –-save react-dom

Skeleton Structure

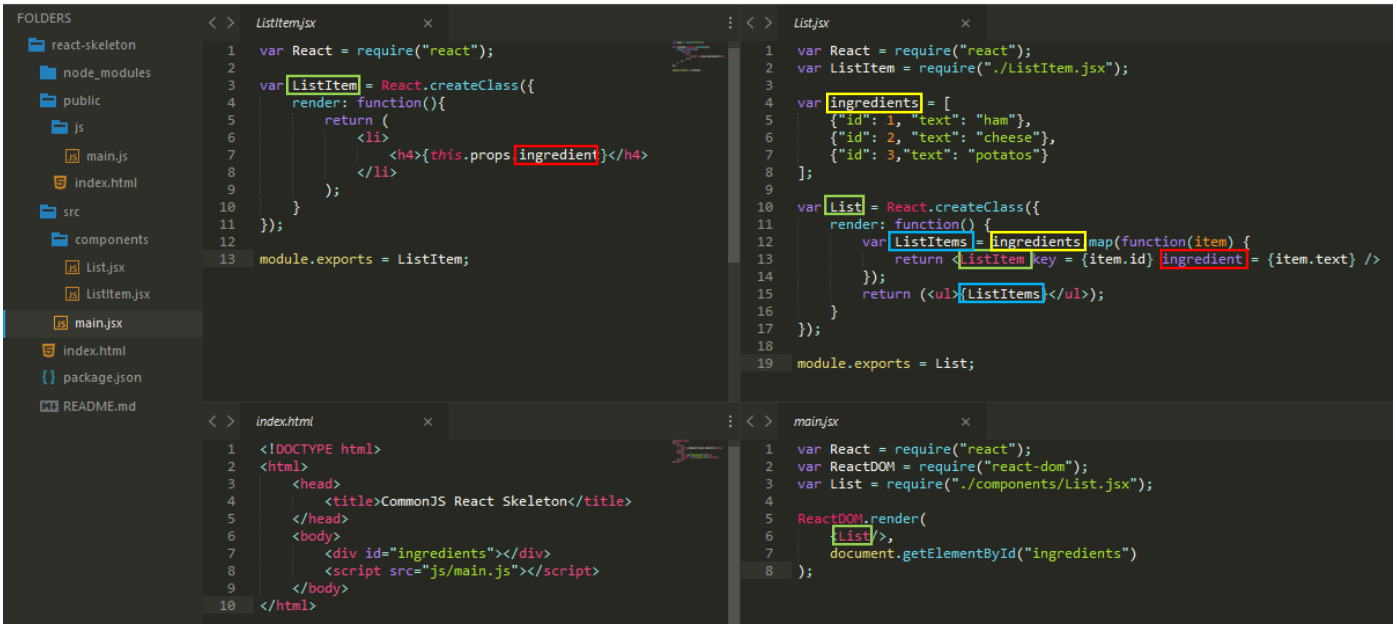
src – has the source code, JSX and JS files.

```
"scripts": {
  "start": "watchify src/main.jsx -v -t [ babelify --presets [ react ] ] -o public/js/main.js"
},
```

The **main.jsx** file (which has all the dependencies) is monitored by **Watchify** for any changes, transpiles them (ES6) with **Babelify**, into one **main.js** (usable ES5 Javascript) file placed in public, which is referenced in the public **index.html** file. We need to run **npm start** to actually start the compiling script.

public – has the actual code (compiled) to be used in the browser.

Skeleton Example Components



1. The **div** with the **ingredients id** is grabbed and **List** is rendered inside it.
2. **List** is populated with **ListItems** which are created by **mapping** the **ingredients array**.
3. **ListItems** returns **ListItem elements** with the **key** and **ingredient properties**, by assigning them the **values** from the **ingredients array** via the **map function** by using **item** as an **argument**.
4. **ListItem** returns the **actual list items with the ingredients** which get their **values from the component above** i.e. **this.props.ingredient** gets it from **<ListItem key={item.id} ingredient={item.text} />**

Components

Guidelines

- The idea is to break down everything into the smallest components possible, while abstracting them to increase reusability. Ex. Christmas To Do list vs Generic To Do list (that can be anything)
- User interaction usually happens at the top level component, because that's where the data usually comes from.
- Data trickles down to the lowest component, top to bottom one-way binding. There is no two way binding.
- Always start coding from the lowest/smallest component up.
- Components have **properties** and **state**. **this.props** is READ ONLY i.e. data is passed down into them. **this.state** is used for data that can change (mutable data). **Never mutate properties.**
- Lists i.e. List Items MUST have unique identifiers, in order for the virtual DOM to work accurately.
- React handles the ***this*** keywords automatically, so there is no need for explicit binding. It can be put anywhere without the need of creating ***this = that*** or ***this = context*** etc.

Empty Component Template

```
var React = require("react");

var Component = React.createClass({
    render: function() {
        return ();
    }
});

module.exports = Component;
```

React.createClass({ })

The **createClass** method is used for declaring classes. It also takes an object as an argument which has the render property.

Render

Render handles JSX i.e. the mixture of HTML and Javascript. Anything between { } inside render is considered Javascript meaning any expression can be used. Ex. {this.props.item}

There is no need for the () after return. It's just there for the visual effect of separation. Also, only one element can be returned at a time, with infinite nestings.

this.props.key

props is an object containing properties named after the keys. Ex. **this.props.text** refers to the **property** with the **key** (there can also be a key named key) named **text** inside the object calling it i.e. **this**. The values are passed down from the component above with **<Component key={index} text={text} />** and then **this.props.text** referes to the value passed in **{text}.**

getInitialState

Every created Class (Component) in React automatically calls its property **getInitialState** only once upon creation, regardless if it's defined or not.

.setState

It is a method for the created classes (components) which takes an object with properties as an argument. Upon calling, it changes the values of the properties (state).

Ingredients List Application Exercise

Components:

1. The list Item. ex. Chicken
2. The list. ex. Chicken, Lettuce, Tomato
3. The input box and add button. (These can be separate, but no need to now)
4. The component containing them all and handling the data (manager)

Continued below…

1. main.jsx

```
var React = require("react");
var ReactDOM = require("react-dom");
var ListManager = require("./components/
    ListManager.jsx");

ReactDOM.render(
    <ListManager title="Ingredients" />,
    document.getElementById("ingredients")
);
```

2. ListManager.jsx

```
var React = require("react");
var List = require("./List.jsx");

var ListManager = React.createClass({
    getInitialState: function(){
        return {items: [], newItemText:""};
    },
    onChange: function(e){
        this.setState({newItemText: e.target.value});
    },
    handleSubmit: function(e){
        e.preventDefault();

        var currentItems = this.state.items;
        currentItems.push(this.state.newItemText);
        this.setState({items: currentItems, newItemText:""});
    },
    render: function(){
        return(
            <div>
                <h3>{this.props.title}</h3>
                <form onSubmit={this.handleSubmit}>
                    <input onChange={this.onChange} value={this.state.newItemText} />
                    <button>Add</button>
                </form>
                <List items={this.state.items} />
            </div>
        );
    }
});

module.exports = ListManager;
```

These just make the input box empty, preventing this

Defined or not, this method is auto invoked only once on class creation. Here, it sets the new state for rerendering.

Sets the state of newItemText on each keystroke, ready to be pushed onChange.

e.preventDefault() stops the actual form submission and does the specified instead.

Gets the current state of items, pushes the state of newItemText from the input box and finally sets the state of the items to include the pushed content while emptying the input box.

3. List.jsx

```
var React = require("react");
var ListItem = require("./ListItem.jsx");

var List = React.createClass({
    render: function(){

        var createItem = function(text, index){
            return <ListItem key={index + text} text={text} />;
        };

        return(<ul>{this.props.items.map(createItem)}</ul>);
    }
});

module.exports = List;
```

4. ListItem.jsx

```
var React = require("react");

var ListItem = React.createClass({
    render: function(){
        return(
            <li><h4>{this.props.text}</h4></li>
        );
    }
});

module.exports = ListItem;
```

item1item2item3 Add

- item1
- item1item2
- item1item2item3

Ingredients

Water Add

- Chicken
- Broccoli
- Rice

```
var React = require("react");
var List = require("./List.jsx");

var ListManager = React.createClass({
    //Called once in the component life cycle - an initializer
    getInitialState: function(){
        return {items: [], newItemText:""};
    },
    //Update the state property on each keystroke
    onChange: function(e){
        this.setState({newItemText: e.target.value});
    },
    //Stop the button from getting clicks since we use form onSubmit
    handleSubmit: function(e){
        e.preventDefault();

        //Grab the current list of items
        var currentItems = this.state.items;

        //Add the new item to the list
        currentItems.push(this.state.newItemText);

        //Update the main item list with the new list and clear the newItemText
        this.setState({items: currentItems, newItemText:""});
    },
    render: function(){
        //onChange is called with every keystroke so we can store the most recent data entered
        //value is what the user sees in the input box - we point this to newItemText so it
        //updates on every keystroke
        return(
            <div>
                <h3>{this.props.title}</h3>
                <form onSubmit={this.handleSubmit}>
                    <input onChange={this.onChange} value={this.state.newItemText} />
                    <button>Add</button>
                </form>
                <List items={this.state.items} />
            </div>
        );
    }
});

module.exports = ListManager;
```

Bootstrap/CSS for JSX

- DO NOT use CDNs for live websites, as it slows them down a lot.
- Always keep the scripts before the closing </body> tag for speed improvement.
- Everything should be inside of a container.
- NEVER EVER EVER put columns inside columns. All the columns should be inside a row. So it goes row columns row columns…
- All the classes should be set on divs. Avoid using them on other elements.
- CTRL + SHIFT + M opens the developer tool showing the layout for different devices.

Grid System

<div class=”col-xs-3 col-sm-6 col-lg-12”>

Makes the div take the full screen on large devices, half the screen on small ones and a third on extra small ones.

className

The ***class=” “*** keyword is conflicting in JSX, so we use ***className=” “*** instead.

Inline Style

The way to do this is to create an **object** with **parameters** replacing css keywords and insert that inside style={ }. The css keywords are written without the dash (-) character, as that would be invalid Javascript.

```
var divStyle={
   marginTop: 10
}

<div style={divStyle} className=”col-sm-12”></div>
```

React-Bootstrap-Seed

This is my own take on a seed file with small examples to get me started.

index.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>React Bootstrap Seed</title>
        <link rel="stylesheet" href="css/
        bootstrap.min.css">
        <link rel="stylesheet" href="css/
        bootstrap-theme.min.css">
    </head>
    <body>

        <div class="container">
            <div class="row">
                <div id="component1"></div>
                <div id="component2"></div>
                <div id="component3"></div>
            </div>
        </div>

        <script src="js/jquery-3.1.0.min.js"></script>
        <script src="js/bootstrap.min.js"></script>
        <script src="js/main.js"></script>
    </body>
</html>
```

main.jsx

```
var React = require("react");
var ReactDOM = require("react-dom");
var Component = require("./components/Component.jsx");

ReactDOM.render(
    <Component
        title="React Bootstrap Component 1"
        headingColor="red"
    />,
    document.getElementById("component1")
);

ReactDOM.render(
    <Component
        title="React Bootstrap Component 2"
        headingColor="green"
    />,
    document.getElementById("component2")
);

ReactDOM.render(
    <Component
        title="React Bootstrap Component 3"
        headingColor="blue"
    />,
    document.getElementById("component3")
);
```

Component.jsx

```
var React = require("react");

var Component = React.createClass({
    render: function(){

        var headingStyle = {}
        if(this.props.headingColor) {
            headingStyle.color = this.props.headingColor;
        }

        return(
            <div className="col-xs-12 col-sm-4">
                <h3 style={headingStyle}>
                    {this.props.title}
                </h3>
                <p>Lorem ipsum dolor sit amet,
                    consectetur adipiscing eit, sed do
                    eiusmod tempor incididunt ut labore
                    et dolore magna aliqua.</p>
            </div>
        );
    }
});

module.exports = Component;
```

Reusability

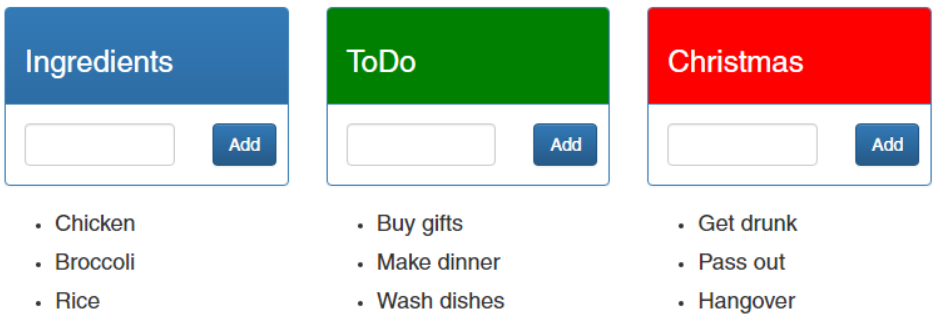
```
render: function(){
    //onChange is called with every keystroke so we can store the most recent data entered
    //value is what the user sees in the input box - we point this to newItemText so it
    //updates on every keystroke

    var divStyle = {
        marginTop: 10
    }

    var headingStyle = {

    }

    if(this.props.headingColor){
        headingStyle.background = this.props.headingColor;
    }

    return(
        <div style={divStyle} className="col-sm-4">
            <div className="panel panel-primary">
                <div style={headingStyle} className="panel-heading">
                    <h3>{this.props.title}</h3>
                </div>
                <div className="row panel-body">
                    <form onSubmit={this.handleSubmit}>
                        <div className="col-xs-8">
                            <input className="form-control" onChange={this.onChange} value={this.state.newItemText} />
                        </div>
                        <div className="col-xs-4">
                            <button className="btn btn-primary">Add</button>
                        </div>
                    </form>
                </div>
            </div>
            <List items={this.state.items} />
        </div>
    );
}
```

main.jsx

```
var React = require("react");
var ReactDOM = require("react-dom");
var ListManager = require("./components/ListManager.jsx");

ReactDOM.render(
    <ListManager title="Ingredients" />,
    document.getElementById("ingredients")
);

ReactDOM.render(
    <ListManager title="ToDo" headingColor="green" />,
    document.getElementById("todo")
);

ReactDOM.render(
    <ListManager title="Christmas" headingColor="red" />,
    document.getElementById("christmas")
);
```

A very cool thing is that components can be modified by using **this.props** for CSS. Ex. **headingStyle** is created empty for “instantiation” and via an if statement, we pass the colors if they are set.

Set the background property of the **headingStyle** object to be equal to a color via **this.props.headingColor** which gets the value from the upper component via attrbutes i.e. **headingColor=”color”.**



Thinking in React

READ THIS: https://facebook.github.io/react/docs/thinking-in-react.html

React is the V in MVC. It's all about creating highly reusable view components and the back-end is not important. It can be Firebase, your own HTTP requests, Flux architecture, Relay… There are a lot of ways to bring data to the app, but React doesn't care about that.

There is not right way to design the apps. This is just Facebooks recommendation. An app can be built by completely doing the UI first and then doing the components. It can be done complete component by component. Whatever makes the most sense, use that.

Build process

1. **Draw:** Create a mockup. Visualize the overall look of the app.
2. **Plan:** Break up the UI into as many REUSABLE components as possible.
3. **Design:** Build a static version of React by using only props and not state (reserved for interactivity).
4. **Interaction:** Add interactivity i.e. apply state.

React Developer Tools

Get the "**React Developer Tolls**" extension for Google Chrome. Make sure "Allow in incognito" and "Allow access to file URLs" are checked. The tools is mostly used for debugging.

In order for the tool to work, the project needs to be ran on a server. Simply opening the index.html file in the browser will not work. To make it work, "**http-server**" is needed for **node**. Install it with "**npm install –g http-sever**" an navigate to the project folder. Make sure that the index.html file is in the top level or in a public folder as it will be looked for automatically by http-sever.

After that, run the module with "**http-server –p 8.483**" where –p stands for port number and the port is anything you want. After that, go to **http:127.0.0.1:8483**, press **CTR + ALT + J** to open up the **Chrome Developer Toos** and navigate to the **React tab**.

Important Notes about React

Autobinding

When creating callbacks in Javascript, you usually need to explicitly bind a method to its instance such that the value of ***this*** is correct. With React, every method is automatically bound to its component instance (except when using ES6 class syntax). React caches the bound method such that it's extremely CPU and memory efficient. It's also less typing!

Multiple Components - Composability

So far, we've looked at how to write a single component to display data and handle user input. Next let's examine one of React's finest features: **Composability**.

Motivation: Separation of Concerns

By building modular components that reuse other components with well-defined interfaces, you get much of the same benefits that you get by using functions or classes. Specifically you can *separate the different concerns* of your app however you please simply by building new components. By building a custom component library for your application, you are expressing your UI in a way that best fits your domain.

The Virtual DOM

React is very fast because it never talks to the DOM directly. React maintains a fast in-memory representation of the DOM. **render()** methods actually return a **description** of the DOM, and React can compare this description with the in-memory representation to compute the fastest way to update the browser.

Additionally, React implements a full synthetic event system such that all event objects are guaranteed to conform to the W3C spec despite the browser quirks, and everything bubbles consistently and efficiently across browsers. You can even use some HTML5 events in IE8!

Most of the time you should stay within React's "faked browser" world since it's more performant and easier to reason about. However, sometimes you simply need to access the underlying API, perhaps to work with third-party library like a jQuery plugin. React provides escape hatches for you to use the underlying DOM API directly.

Reactive Updates

Open hello-react.html in a web browser and type your name into the text field. Notice that React is only changing the time string in the UI – any input you put in the text field remains, even though you haven't written any code to manage this behavior. React figures it out for you and does the right thing.

The way we are able to figure this out is that React does not manipulate the DOM unless it needs to. **It uses a fast, internal mock DOM to perform diffs and computes the most efficient DOM mutation for you.**

The inputs to this component are called **props** – short for "properties". They're passed as attributes in JSX syntax. You should think of these as immutable within the component, that is, **never write** to **this.props.**

What Shouldn't Go in State?

this.state should only contain the minimal amount of data needed to represent your UI's state. As such, it should not contain:

- **Computed data:** Don't worry about precomputing values based on state – It's easier to ensure that your UI is consistent if you do all computation within **render().** For example, if you have an array of list items in state and you want to render the count as a string, simply render **this.state.listItems.length + ' list items'** in your **render()** method rather than storing it on state.
- **React components:** Build them in **render()** based on underlying props and state.
- **Duplicated data from props:** Try to use props as the source of truth where possible. One valid use to store props in state is to be able to know its previous values, because props may change as the result of a parent component re-rendering.

Notes:

NEVER mutate **this.state** directly, as calling **setState()** afterwards may replace the mutation you made. Treat **this.state** as if it were immutable.

setState() does not immediately mutate **this.state** but creates a pending state transition. Accessing **this.state** after calling this method can potentially return the existing value.

There is no guarantee of synchronous operation of calls to **setState** and calls may be batched for performance ains.

setState() will always trigger a re-render unless conditional rendering logic is implemented in **shouldComponentUpdate().** If mutable objects are being used and the logic cannot be implemented in **shouldComponentUpdate()**, calling **setState()** only when the new state differs from the previous state will avoid unnecessary re-renders.

React Event System

What ths means is that React has its own implementation (because of the virtual DOM) for most of the regular HTML events. For the full list, go here: https://facebook.github.io/react/docs/events.html

HTML: <button **onclick=(onClickFunction)** >Click Me!</button>
JSX: <button **onClick={this.onClickFunction}** >Click Me!</button>

Routing

This requres 2 packages, **history** and **react-router**. We install them by running **npm install history@1.13.1 react-router@latest --save.** Multiple packages can be installed in one line and the –save is to include it in package.json.

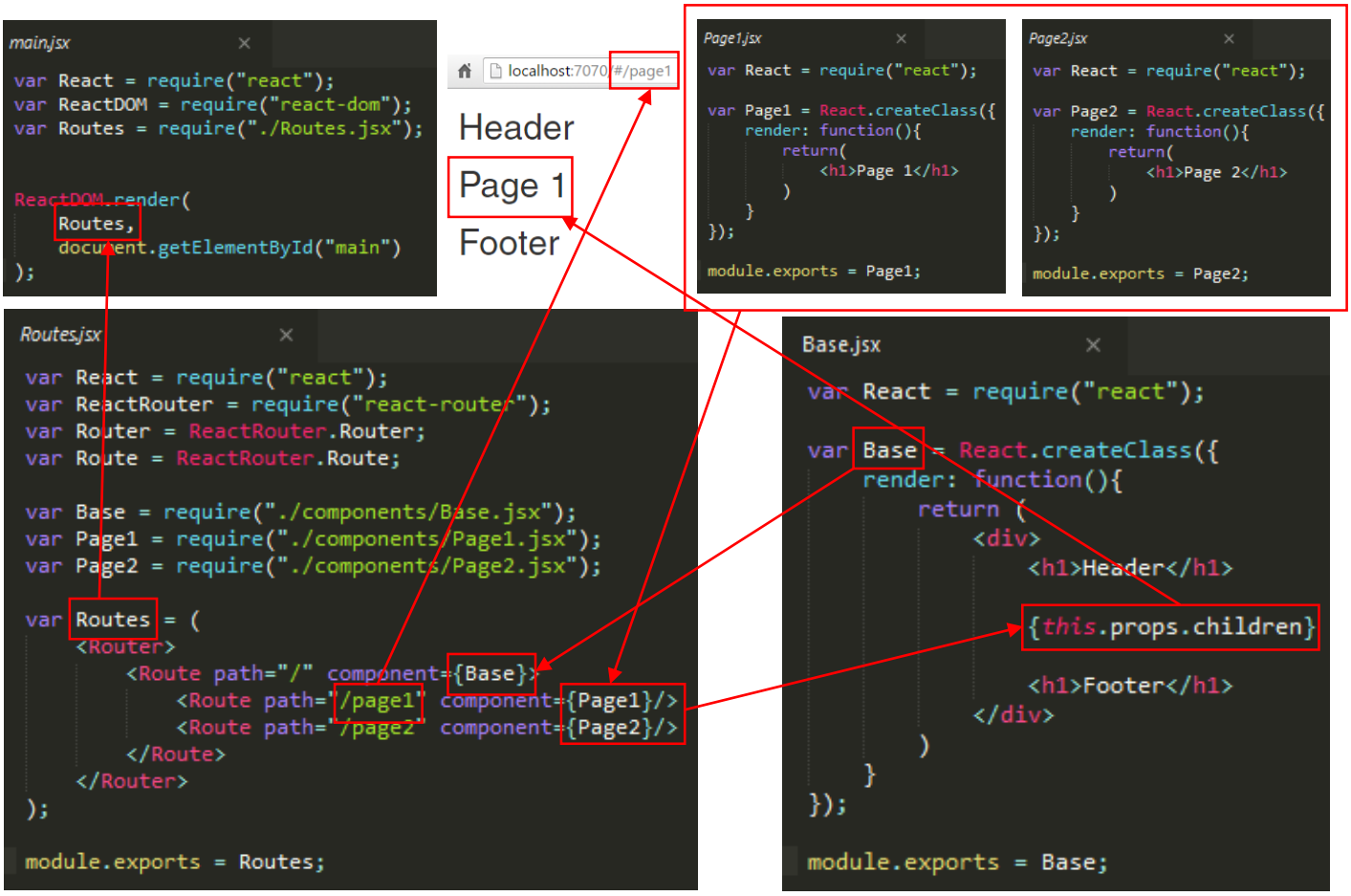
There are 3 main things:

- **Router** – Manages all the routes in the application.
- **Route** – Think of it as a single page.
- **Link -** These are the actual links for navigation, as **** doesn't work.

ReactRouter is the package, so we need to grab the Router and Route from it by using **ReactRouter.Router** and **ReactRouter.Route**. Usually, there is a page created called "base" with things to be included on every page. **this.props.children** is automatically populated with the components nested in the **Base** path in **Routes**.

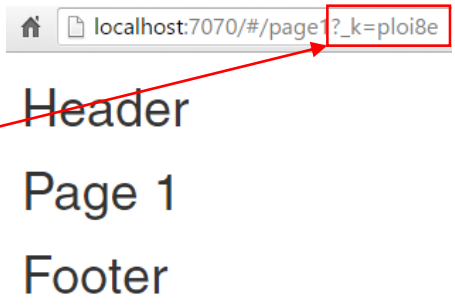
Routing works ONLY on a live server. It will not work on a local file system.

For this, we need to run the http-server module in node with **http-server –p 7070**. There must be an index.html file or a public folder. After that go to http://127.0.0.1:7070 or **localhost:7070**



Hash History

```
Routes.jsx
var React = require("react");
var ReactRouter = require("react-router");
var Router = ReactRouter.Router;
var Route = ReactRouter.Route;

var CreateHistory = require('history/lib/createHashHistory');

var History = new CreateHistory({
    queryKey: false
});

var Base = require("./components/Base.jsx");
var Page1 = require("./components/Page1.jsx");
var Page2 = require("./components/Page2.jsx");

var Routes = (
    <Router history={History}>
        <Route path="/" component={Base}>
            <Route path="/page1" component={Page1}/>
            <Route path="/page2" component={Page2}/>
        </Route>
    </Router>
);

module.exports = Routes;
```



This is used for supporting older browsers in the storing of state which the hash represents.

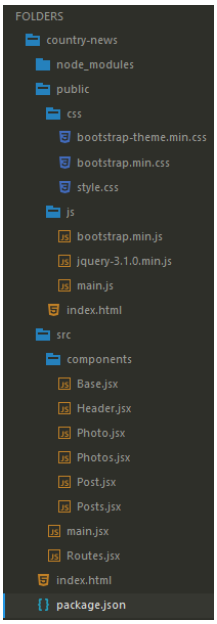
https://github.com/ReactTraining/history/blob/master/docs/HashHistoryCaveats.md

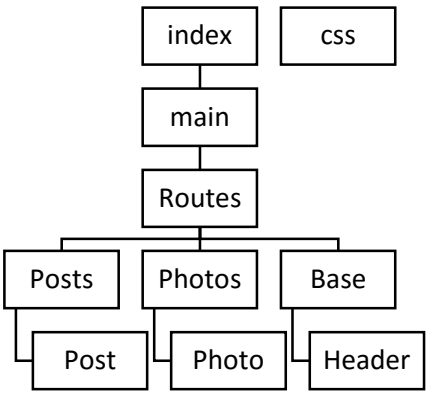
This can be removed by implementing the things circled in red, which modify the history module (a dependency of react-router).

Country News Exercise

Don't design the layout first and then try to make the components. It is much harder this way because of JSX vs CSS. Instead, build each component from start to finish starting from the smallest one.

1. Create the folder "country-news" and paste in the content from the react-bootstrap-seed.
2. Define all the components and break them down to the smallest parts.
3. Run npm start in one terminal for transpiling/debugging the app.
4. Run http-server –p 7070 in another terminal in order to create a local server for the app, so that the routings can work. They don't work on a local server.
5. Build component by component, finishing one completely before moving to the next one.
6. Use className for adding CSS from a .css file, rather than using style={styleName}.
7. Use lorepixel.com for randomly generating placeholder images.





index.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>Country News</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
        <link rel="stylesheet" href="css/bootstrap-theme.min.css">
        <link rel="stylesheet" href="css/style.css">
    </head>
    <body>
        <div style="height:500px; width:100%; height:100%; position:fixed; background:#ddd;">
            <div style="background:#3b5998; height:200px;"></div>
        </div>
        <div class="container" style="width:800px;">
            <div id="main"></div>
        </div>

        <script src="js/jquery-3.1.0.min.js"></script>
        <script src="js/bootstrap.min.js"></script>
        <script src="js/main.js"></script>
    </body>
</html>
```

main.jsx

```
var React = require("react");
var ReactDOM = require("react-dom");

var Posts = require("./components/Posts.jsx");
var Photos = require("./components/Photos.jsx");
var Header = require("./components/Header.jsx");

var Routes = require("./Routes.jsx");

ReactDOM.render(
    Routes,
    document.getElementById("main")
);
```

Routes.jsx

```
var React = require("react");
var ReactRouter = require("react-router");
var Router = ReactRouter.Router;
var Route = ReactRouter.Route;

var CreateHistory = require('history/lib/createHashHistory');

var History = new CreateHistory({
    queryKey: false
});

var Base = require("./components/Base.jsx");
var Posts = require("./components/Posts.jsx");
var Photos = require("./components/Photos.jsx");

var Routes = (
    <Router history={History}>
        <Route component={Base}>
            <Route path="/" component={Posts}></Route>
            <Route path="/photos" component={Photos}></Route>
        </Route>
    </Router>
);

module.exports = Routes;
```

Base.jsx

```
var React = require("react");
var Header = require("./Header.jsx");

var Base = React.createClass({
    render: function(){
        return(
            <div>
                <Header/>
                {this.props.children}
            </div>
        );
    }
});

module.exports = Base;
```

Header.jsx

```
var React = require("react");
var Link = require('react-router').Link

var Header = React.createClass({
    render: function(){
        return(
            <div>
                <div className="row">
                    <div className="col-xs-12">
                        <div className="row">
                            <div className="col-xs-7 header">
                                <h1>Country News</h1>
                                <h4>Top stories in my country</h4>
                            </div>
                            <div className="col-xs-1 social">
                                <a href={"https://www.facebook.com"} target={"_blank"}>
                                    <div className="row">
                                        <img className="col-xs-12" src={"Facebook Logo"}/>
                                    </div>
                                </a>
                            </div>
                        </div>
                    </div>
                </div>
                <div className="row">
                    <div className="col-xs-12">
                        <div className="col-xs-1 col-xs-offset-10">
                            <Link to="/">Posts</Link>
                        </div>
                        <div className="col-xs-1">
                            <Link to="/photos">Photos</Link>
                        </div>
                    </div>
                </div>
            </div>
        );
    }
});

module.exports = Header
```

Posts.jsx

```
var React = require("react");
var Post = require("./Post.jsx");

var data = [
    {
        title: "Lorem ipsum dolor sit amet 1",
        subtitle: "Lorem ipsum dolor sit amet 1",
        excerpt: "Lorem ipsum dolor sit amet, consectetur.",
        thumb: "http://lorempixel.com/400/200/nature/1",
        location: "Lorem, Ipsum",
        authorPic: "http://lorempixel.com/400/200/people/1"
    },
    {
        title: "Lorem ipsum dolor sit amet 2",
        subtitle: "Lorem ipsum dolor sit amet 2",
        excerpt: "Lorem ipsum dolor sit amet, consectetur.",
        thumb: "http://lorempixel.com/400/200/nature/2",
        location: "Lorem, Ipsum",
        authorPic: "http://lorempixel.com/400/200/people/2"
    }
]

var Posts = React.createClass({
    render: function(){
        var posts = data.map(function(post){
            return <Post
                        title={post.title}
                        subtitle={post.subtitle}
                        excerpt={post.excerpt}
                        thumb={post.thumb}
                        location={post.location}
                        authorPic={post.authorPic}
                    />
        });
        return (<div>{posts}</div>);
    }
});

module.exports = Posts;
```

Post.jsx

```
var React = require("react");

var Post = React.createClass({
    render: function(){

        return(
            <div className="row">
                <div className="col-xs-12 post">
                    <div className="row">
                        <div className="col-xs-2 author">
                            <div className="row">
                                <img
                                    className="col-xs-10 col-xs-offset-1"
                                    src={this.props.authorPic}
                                />
                            </div>
                        </div>
                        <div className="col-xs-10">
                            <h1>{this.props.title}</h1>
                            <h3>{this.props.subtitle}</h3>
                            <p>{this.props.excerpt}</p>
                            <img src={this.props.thumb} />
                            <p>{this.props.location}</p>
                        </div>
                    </div>
                </div>
            </div>
        );
    }
});

module.exports = Post;
```

Photos.jsx

```
var React = require("react");
var Photo = require("./Photo.jsx");

var Photos = React.createClass({
    render: function(){

        var pics = [];

        for (let i = 0; i < 30; i++) {
            pics.push(
                <Photo
                    url={"http://lorempixel.com/400/200/nature/" + i}
                />
            );
        }

        return (
            <div className="row">
                <div className="col-xs-12 photoGrid">
                    <div className="row">
                        {pics}
                    </div>
                </div>
            </div>
        );
    }
});

module.exports = Photos;
```

Photo.jsx

```
var React = require("react");

var Photo = React.createClass({
    render: function(){
        return (
            <img
                className="col-xs-3 photo"
                src={this.props.url}
            />
        );
    }
});

module.exports = Photo;
```

Forms

LeadCapture.jsx

```jsx
var React = require("react");
var EmailField = require("./EmailField.jsx");
var NameField = require("./NameField.jsx");

var LeadCapture = React.createClass({
    onSubmit: function(){
        if(!this.refs.fieldEmail.state.valid){
            alert("Need a valid Email!");
        } else {
            //Send request to email host or server.
            httpRequestBody = {
                email: this.refs.fieldEmail.state.value,
                name: this.refs.fieldName.state.value
            }

            this.refs.fieldName.clear();
            this.refs.fieldEmail.clear();
        }
    },
    render: function(){
        return(
            <div className="col-xs-3">
                <div className="panel panel-default">
                    <div className="panel-body">
                        <NameField type="First" ref="fieldName" />
                        <br/>
                        <EmailField ref="fieldEmail" />
                        <button className="btn btn-primary"onClick={this.onSubmit}>
                            Submit
                        </button>
                    </div>
                </div>
            </div>
        );
    }
});

module.exports = LeadCapture;
```

EmailField.jsx

```jsx
var React = require("react");
var validator = require("email-validator");

var EmailField = React.createClass({
    getInitialState: function(){
        return {valid: true, value: ""}
    },
    onChange: function(e){
        var val = e.target.value;

        if(!validator.validate(val)){
            this.setState({valid: false, value: val});
        } else {
            this.setState({valid: true, value: val});
        }
    },
    clear: function(){
        this.setState({valid: true, value: ""})
    },
    render: function(){
        var formClass = this.state.valid ? "form-group" : "form-group has-error"
        return(
            <div className={formClass}>
                <input
                    className="form-control"
                    onChange={this.onChange}
                    value={this.state.value}
                    placeholder="Email"
                />
            </div>
        );
    }
});

module.exports = EmailField;
```

NameField.jsx

```jsx
var React = require("react");

var NameField = React.createClass({
    getInitialState(){
        return {value: ""}
    },
    onChange: function(e){
        var val = e.target.value;
        this.setState({value: val})
    },
    clear: function(){
        this.setState({value: ""})
    },
    render: function(){
        return(
            <input
                className="form-control"
                placeholder={this.props.type + " Name"}
                onChange={this.onChange}
                value={this.state.value}
            />
        );
    }
});

module.exports = NameField;
```

Refs

Refs allow parent components to interact with child components. You gain access to a child component by adding **ref=" "** (name can be anything) to a component. Ex: <EmailComponent ref="emailComponent" />

After this, the state in the child component can be accessed, as well as invoking functions. Ex:

this.refs.emailComponent.state.value would return the email inputted in a field.
this.refs.emailComponent.functionName() will invoke the desired function.

Express

It's a framework that sits on top of Node in order to allow HTTP requests to be made and handled by Node.

Mark advises to avoid using fake data in the view, and rather use the actual backend servers. This would save a lot of refactoring in the future.

Simple Express Server

server.js

```
var express = require('express');
var bodyParser = require('body-parser');
var app = express();

//Allow all requests from all domains & localhost
app.all('/*', function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "X-Requested-With, Content-Type, Accept");
  res.header("Access-Control-Allow-Methods", "POST, GET");
  next();
});

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: false}));

var ingredients = [
    {
        "id": "234kjw",
        "text": "Eggs"
    }
];

app.get('/ingredients', function(req, res) {
    console.log("GET Fr SERVER");
    res.send(ingredients);
});

app.post('/ingredients', function(req, res) {
    var ingredient = req.body;
    console.log(req.body);
    ingredients.push(ingredient);
    res.status(200).send("Successfully posted ingredient");
});

app.listen(6069);
```

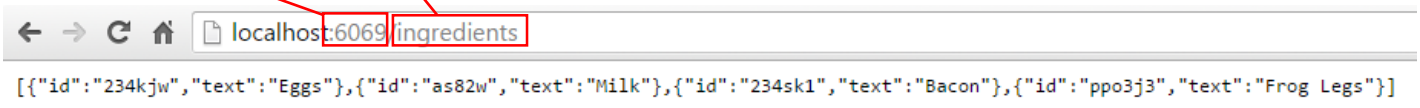
An Express server can run on just a couple of lines. These server is started normally with **node server.js** and it listens to the designated 6069 port.

body-parser is a package (was part of express, now standalone) which takes incoming requests and parses them into JSON.

app.use is middleware which processes requests before they get to the get post handlers.

app.listen(6069) turns on the server and listens to the port.

app.all res.header is used in order to be able to make the requests, because they are coming from an unknown client.



[{"id":"234kjw","text":"Eggs"},{"id":"as82w","text":"Milk"},{"id":"234sk1","text":"Bacon"},{"id":"ppo3j3","text":"Frog Legs"}]

If any changes are made in the server, it has to be restarted. In order to avoid doing this manually, we can use an npm package called **nodemon**. To use it, install it with **npm install –g nodemon** and instead of **node server.js** use **nodemon server.js**. This will run the server, watch for changes AND restart it after each change.

Web Requests HTTP

Clients sends requests, server sends responses. Both follow the HTTP protocol. All of this is baically sending data back and forth. Each time you open Chrome and visit Google, the browser makes an HTTP GET request and as a response, it gets the HTML.

To see this, you can use the linux terminal as a client and use **curl** with the full address to get the HTML. Ex. **curl** https://www.google.com

Both the browser and the terminal made the exact same request and got the exact same response. The difference is that the browser knows how to handle the HTML.

Postman

Postman is a Google Chrome extension for testing API servers with HTTP Requests. It simply displays JSON nicely, which does not look good in a terminal. There are some great APIs which server JSON for testing purposes, such as: Star Wars API, Pokemon API. Also, you can use the JSON Online Formatter for making raw JSON pretty.

CRUD

Create = POST
Read = GET
Update = PUT
Delete = DELETE

.bind

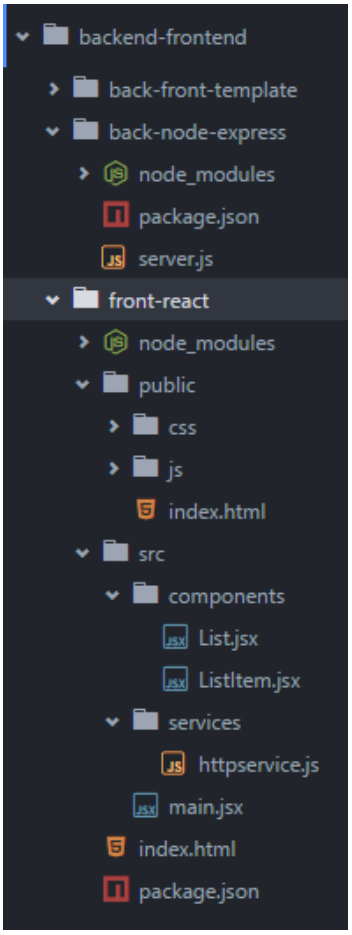
this refers to the call site. It defaults to the Window object. So in the WTF, the call site for the .getCar method is not marksGarage, but rather storeGetCarForLater, which defaults to the Window object. In this case, we use bind to use marksGarage as the call site specifically.

Fetch

It is a polyfill (a popular feature not present in the language, but provided by a third party) which servers as an easier way to make web requests and handle responses than using XMLHttpRequest (used in vanilla javascript). jQuery for example has a wrapper around this ugliness which provides a simpler API.

This is mainted by Github themselves and we can install this with: **npm install whatwg-fetch --save**

Node Express Server + React View



server.js

```
var express = require('express');
var bodyParser = require('body-parser');
var app = express();

//Allow all requests from all domains & localhost
app.all('/*', function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "X-Requested-With, Content-Type, Accept");
  res.header("Access-Control-Allow-Methods", "POST, GET");
  next();
});

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: false}));

var ingredients = [
    {"id": "234kjw", "text": "Eggs"},
    {"id": "as82w", "text": "Milk"},
    {"id": "234sk1", "text": "Bacon"},
    {"id": "ppo3j3", "text": "Frog Legs"}
];

app.get('/ingredients', function(req, res) {
    console.log("GET From SERVER");
    res.send(ingredients);
});

app.listen(6069);
```

HTTP calls should be in a data store, not the view.

List.jsx

```
var React = require("react");
var ListItem = require("./ListItem.jsx");
var HTTP = require("../services/httpservice");

var List = React.createClass({
    getInitialState: function(){
        return {ingredients: []};
    },
    componentWillMount: function(){
        HTTP.get("/ingredients")
        .then(function(data){
            console.log("Data: ", data);
            this.setState({ingredients: data});
        }.bind(this)); // binds it to the component, instead of the callback.
    },
    render: function(){
        var listItems = this.state.ingredients.map(function(item){
            return <ListItem key={item.id} ingredient={item.text} />;
        });

        return (<ul>{listItems}</ul>);
    }
});

module.exports = List;
```

httpservice.js

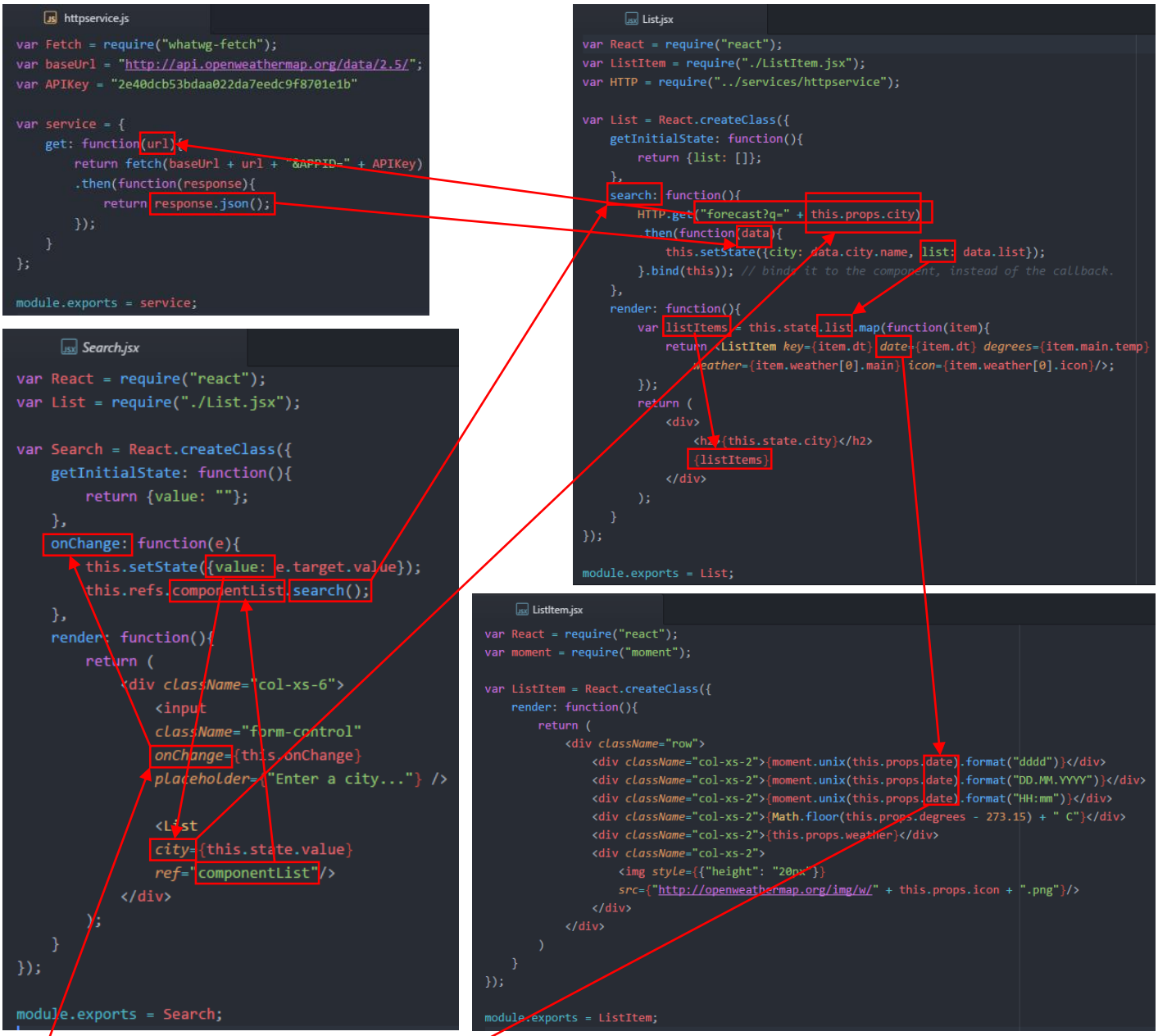
```
var Fetch = require("whatwg-fetch");
var baseUrl = "http://localhost:6069";

var service = {
    get: function(url){
        return fetch(baseUrl + url)
        .then(function(response){
            return response.json();
        });
    }
};

module.exports = service;
```

ListItem.jsx

```
var React = require("react");

var ListItem = React.createClass({
    render: function(){
        return (
            <li>
                <h4>{this.props.ingredient}</h4>
            </li>
        )
    }
});

module.exports = ListItem;
```

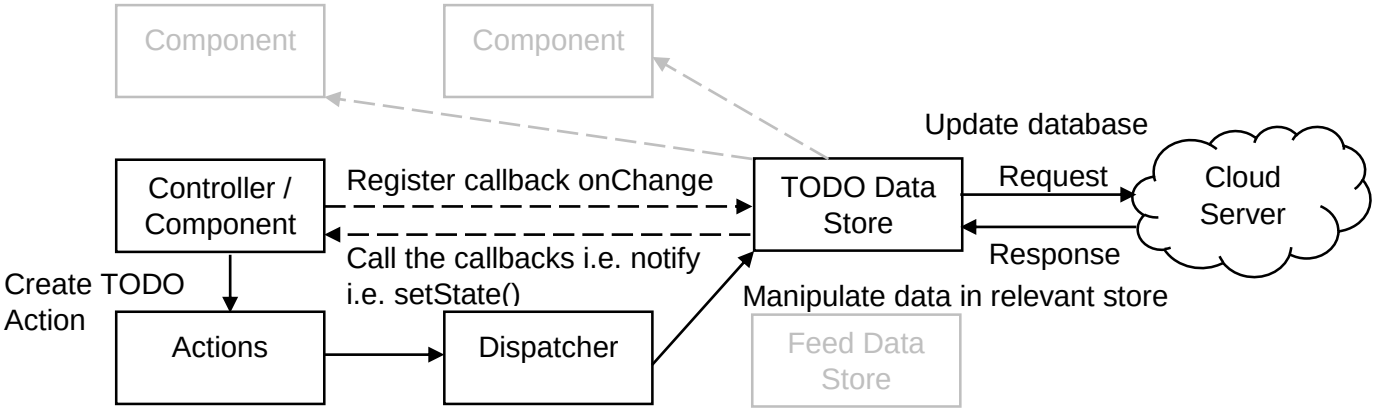
Weather App

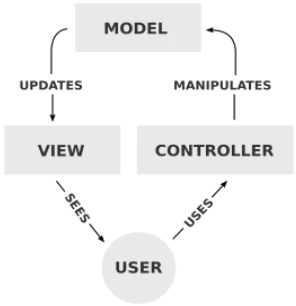
HTTP calls should be in a data store, not the view.



Flux

React is just the VIiew in the MVC pattern. There is no good architecture for managing data with the View. Flux is a pattern designed by Facebook to do just that. The main problem is the communication between components i.e. if one components updates the data, how will another know it?

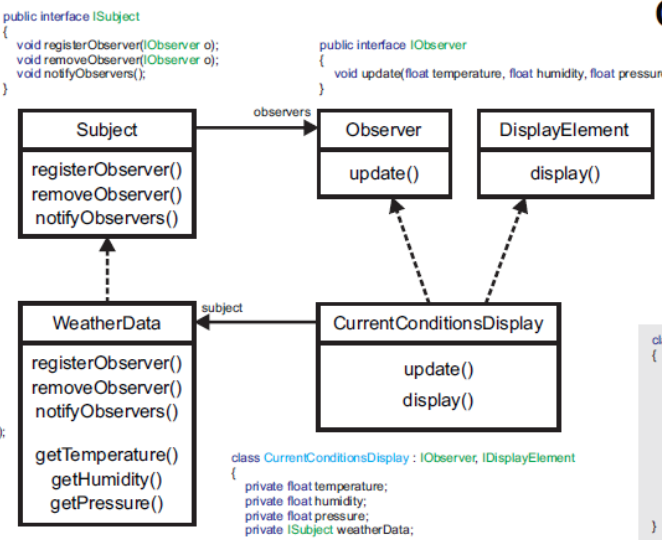




1. Controller registers with a data store in order to get notified when changes happen. It registers a function to be called.
2. The view components inside the controller create actions. ex. Click button to add new todo.
3. Actions talk to the dispatcher which notifies the relevant data stores.
4. The data stores process, and add or retrieve data from the server.
5. The data stores call the callbacks which notify the subscribed components of the changes and sets the new state.

Flux is pretty much the **Observer Pattern**.

Observer Pattern




```
public class WeatherData : ISubject
{
   private List<IObserver> observers;
   private float temperature;
   private float humidity;
   private float pressure;

   public WeatherData()
   {
      observers = new List<IObserver>();
   }

   public void registerObserver(IObserver o)
   {
      observers.Add(o);
   }

   public void removeObserver(IObserver o)
   {
      int i = observers.IndexOf(o);
      if (i >= 0)
      {
         observers.RemoveAt(i);
      }
   }

   public void notifyObservers()
   {
      for (int i = 0; i < observers.Count(); i++)
      {
         IObserver observer = (IObserver)observers[i];
         observer.update(temperature, humidity, pressure);
      }
   }

   public void measurementsChanged()
   {
      notifyObservers();
   }

   public void setMeasurements(float temperature, float humidity, float pressure)
   {
      this.temperature = temperature;
      this.humidity = humidity;
      this.pressure = pressure;
      measurementsChanged();
   }

   public float getTemperature()
   {
      return temperature;
   }

   public float getHumidity()
   {
      return humidity;
   }

   public float getPressure()
   {
      return pressure;
   }
}
```

```
public interface ISubject
{
   void registerObserver(IObserver o);
   void removeObserver(IObserver o);
   void notifyObservers();
}
```

```
public interface IObserver
{
   void update(float temperature, float humidity, float pressure);
}
```

```
public interface IDisplayElement
{
   void display();
}
```

```
class CurrentConditionsDisplay : IObserver, IDisplayElement
{
   private float temperature;
   private float humidity;
   private float pressure;
   private ISubject weatherData;

   public CurrentConditionsDisplay(ISubject weatherData)
   {
      this.weatherData = weatherData;
      weatherData.registerObserver(this);
   }

   public void update(float temperature, float humidity, float pressure)
   {
      this.temperature = temperature;
      this.humidity = humidity;
      this.pressure = pressure;
      display();
   }

   public void display()
   {
      Console.WriteLine("Current conditions: " + temperature
         + " F degrees, Humidity : " + humidity
         + " % and Pressure: " + pressure + " P.");
   }
}
```

Main

```
class Program
{
   static void Main(string[] args)
   {
      WeatherData weatherData = new WeatherData();

      CurrentConditionsDisplay currentDisplay
      = new CurrentConditionsDisplay(weatherData);

      weatherData.setMeasurements(80, 65, 30.4f);
      weatherData.setMeasurements(82, 70, 29.2f);
      weatherData.setMeasurements(78, 90, 27.7f);
   }
}
```

Output

```
Current conditions: 80 F degrees,
Humidity : 65 % and
Pressure: 30,4 P.

Current conditions: 82 F degrees,
Humidity : 70 % and
Pressure: 29,2 P.

Current conditions: 78 F degrees,
Humidity : 90 % and
Pressure: 27,7 P.
```

The concept "Flux" is simply:

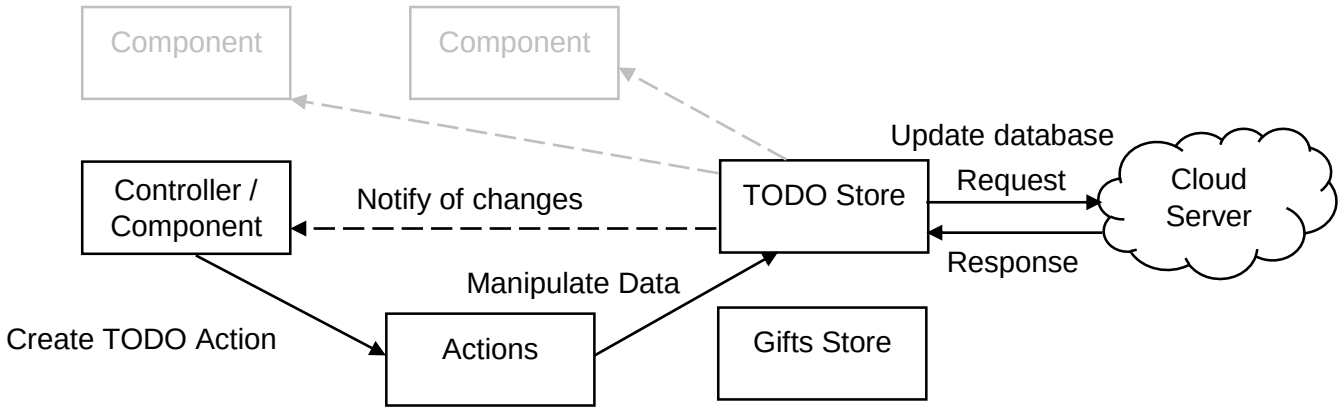
1. That your view triggers an event (say, after user types a name in a text field).
2. That event updates a model.
3. Then the model triggers an event.
4. And the view responds to that model's event by re-rendering with the latest data.

That's it.

source:
http://blog.andrewray.me/reactjs-for-stupid-people/
http://blog.andrewray.me/flux-for-stupid-people/
http://www.jackcallister.com/2015/02/26/the-flux-quick-start-guide.html

Reflux

It’s a library that makes the implementation of Flux much easier.



Redux

Functional Programming

Professor Frisby’s Mostly Adequate Guide to Functional Programming
https://drboolean.gitbooks.io/mostly-adequate-guide/content/

Elm

After installing Elm successfully, you will have the following command line tools available on your computer:

- elm-repl
- elm-reactor
- elm-make
- elm-package

Each one has a --help flag that will show more information.

elm-repl i.e. node

elm-repl lets you play with simple Elm expressions.

```
$ elm-repl

> 1 / 2
0.5 : Float
```

Note: elm-repl works by compiling code to JavaScript, so make sure you have Node.js installed. We use that to evaluate code.

elm-reactor i.e. npm start + http-server

elm-reactor helps you build Elm projects without messing with the command-line too much. You just run it at the root of your project, like this:

```
git clone https://github.com/evancz/elm-architecture-tutorial.git
cd elm-architecture-tutorial
elm-reactor
```

This starts a server at http://localhost:8000. You can navigate to any Elm file and see what it looks like. Try to check out examples/1-button.elm.

Notable flags:

- --port lets you pick something besides port 8000. So you can say elm-reactor --port=8123 to get things to run at http://localhost:8123.
- --address lets you replace localhost with some other address. For example, you may want to use elm-reactor --address=0.0.0.0 if you want to try out an Elm program on a mobile device through your local network.

elm-make i.e. npm start

elm-make builds Elm projects. It can compile Elm code to HTML or JavaScript. It is the most general way to compile Elm code, so if your project becomes too advanced for elm-reactor, you will want to start using elm-make directly.

Say you want to compile Main.elm to an HTML file named main.html. You would run this command:

```
elm-make Main.elm --output=main.html
```

Notable flags:

- --warn prints warnings to improve code quality

elm-package i.e. npm

elm-package downloads and publishes packages from our package catalog. As community members solve problems in a nice way, they share their code in the package catalog for anyone to use!

Say you want to use evancz/elm-http and NoRedInk/elm-decode-pipeline to make HTTP requests to a server and turn the resulting JSON into Elm values. You would say:

```
elm-package install evancz/elm-http
elm-package install NoRedInk/elm-decode-pipeline
```

This will add the dependencies to your elm-package.json file that describes your project. (Or create it if you do not have one yet!) More information about all this here!

Notable commands:

- install: install the dependencies in elm-package.json
- publish: publish your library to the Elm Package Catalog
- bump: bump version numbers based on API changes
- diff: get the difference between two APIs

Core Language

We will cover values, functions, lists, tuples, and records. These building blocks all correspond pretty closely with structures in languages like JavaScript, Python, and Java.

Functions

```
> import String
> String.length
<function> : String -> Int
```

The function String.length has type String -> Int. This means it must take in a String argument, and it will definitely return an integer result. So let's try giving it an argument:

A String -> Int function **must** get a String argument!

Here is the crazy secret though: this is how all functions are defined! You are just giving a name to an anonymous function. So when you see things like this:

```
> half n = n / 2
<function> : Float -> Float
```

You can think of it as a convenient shorthand for:

```
> half = \n -> n / 2
<function> : Float -> Float
```

This is true for all functions, no matter how many arguments they have. So now let's take that a step farther and think about what it means for functions with multiple arguments:

```
> divide x y = x / y
<function> : Float -> Float -> Float

> divide 3 2
1.5 : Float
```

That seems fine, but why are there two arrows in the type for divide?! **To start out, it is fine to think that "all the arguments are separated by arrows, and whatever is last is the result of the function"**. So divide takes two arguments and returns a Float.

To really understand why there are two arrows in the type of divide, it helps to convert the definition to use anonymous functions.

```
> divide x y = x / y
<function> : Float -> Float -> Float

> divide x = \y -> x / y
<function> : Float -> Float -> Float

> divide = \x -> (\y -> x / y)
<function> : Float -> Float -> Float
```

All of these are totally equivalent. We just moved the arguments over, turning them into anonymous functions one at a time. So when we run an expression like divide 3 2 we are actually doing a bunch of evaluation steps:

```
divide 3 2
(divide 3) 2                -- Step 1 - Add the implicit parentheses
((\x -> (\y -> x / y)) 3) 2 -- Step 2 - Expand `divide`
(\y -> 3 / y) 2             -- Step 3 - Replace x with 3
3 / 2                       -- Step 4 - Replace y with 2
1.5                         -- Step 5 - Do the math
```

After you expand divide, you actually provide the arguments one at a time. Replacing x and y are actually two different steps.

Let's break that down a bit more to see how the types work. In evaluation step #3 we saw the following function:

```
> (\y -> 3 / y)
<function> : Float -> Float
```

It is a Float -> Float function, just like half. Now in step #2 we saw a fancier function:

```
> (\x -> (\y -> x / y))
<function> : Float -> Float -> Float
```

Well, we are starting with \x -> ... so we know the type is going to be something like Float -> .... We also know that (\y -> x / y) has type Float -> Float.

So if you actually wrote down all the parentheses in the type, it would instead say Float -> (Float -> Float). You provide arguments one at a time. So when you replace x, the result is actually another function.

It is the same with all functions in Elm:

```
> import String
> String.repeat
<function> : Int -> String -> String
```

This is really Int -> (String -> String) because you are providing the arguments one at a time.

Because all functions in Elm work this way, you do not need to give all the arguments at once. It is possible to say things like this:

```
> divide 128
<function> : Float -> Float

> String.repeat 3
<function> : String -> String
```

This is called partial application. It lets us use the (|>) operator to chain functions together in a nice way, and it is why function types have so many arrows!

Type Annotations

So far we have just let Elm figure out the types, but it also lets you write a type annotation on the line above a definition if you want. So when you are writing code, you can say things like this:

```
half : Float -> Float
half n =
  n / 2

divide : Float -> Float -> Float
divide x y =
  x / y

askVegeta : Int -> String
askVegeta powerLevel =
  if powerLevel > 9000 then
    "It's over 9000!!!"

  else
    "It is " ++ toString powerLevel ++ "."
```

People can make mistakes in type annotations, so what happens if they say the wrong thing? Well, the compiler does not make mistakes, so it still figures out the type on its own. It then checks that your annotation matches the real answer. In other words, the compiler will always verify that all the annotations you add are correct.

Type Aliases

The whole point of type aliases is to make your type annotations easier to read.

As your programs get more complicated, you find yourself working with larger and more complex data. For example, maybe you are making twitter-for-dogs and you need to represent a user. And maybe you want a function that checks to see if a user has a bio or not. You might write a function like this:

```
hasBio : { name : String, bio : String, pic : String } -> Bool
hasBio user =
  String.length user.bio > 0
```

That type annotation is kind of a mess, and users do not even have that many details! Imagine if there were ten fields. Or if you had a function that took users as an argument and gave users as the result.

In cases like this, you should create a type alias for your data:

```
type alias User =
  { name : String
  , bio : String
  , pic : String
  }
```

This is saying, wherever you see User, replace it by all this other stuff. So now we can rewrite our hasBio function in a much nicer way:

```
hasBio : User -> Bool
hasBio user =
  String.length user.bio > 0
```

Looks way better! It is important to emphasize that these two definitions are exactly the same. We just made an alias so we can say the same thing in fewer key strokes.

So if we write a function to add a bio, it would be like this:

```
addBio : String -> User -> User
addBio bio user =
  { user | bio = bio }
```

Imagine what that type annotation would look like if we did not have the User type alias. Bad!

Type aliases are not just about cosmetics though. They can help you think more clearly. When writing Elm programs, it is often best to start with the type alias before writing a bunch of functions. I find it helps direct my progress in a way that ends up being more efficient overall.

Suddenly you know exactly what kind of data you are working with. If you need to add stuff to it, the compiler will tell you about any existing code that is affected by it. I think most experienced Elm folks use a similar process when working with records especially.

Note: When you create a type alias specifically for a record, it also generates a record constructor. So our User type alias will also generate this function:

```
User : String -> String -> String -> User
```

The arguments are in the order they appear in the type alias declaration. You may want to use this sometimes.

Everything in Elm is an expression. It doesn't do something, it returns something.

Combining values - records

```
> person = { name="Andy", age=21 }
{ name = "Andy", age = 21 } : { age : number, r

> { person | age=22 }
{ name = "Andy", age = 22 } : { name : String,
```

You can't modify a record, but you can return another version of a recors.

Functions

```
> square 3
9 : number

> gravity 10 0.1
9.019 : Float

> square (gravity 10 0.1)
81.342361 : Float
```

Parantheses are used for grouping things (like math), not for calling functions.

Functions - anonymous

```
> sos = \x y -> x^2 + y^2
  : number -> number -> number

> sos 3 4
25 : number
```

**** Start the anonymous function with x and y arguments.
-> The body of the function

Combining functions

```
square n = n^2

apply_twice f n = f (f n)

> apply_twice square 3
81 : number
```

Combining functions - currying

```
square_twice = apply_twice square

> square_twice 3
81 : number
```

apply_twice takes a function and a number as an argument. It returns a function applied twice, once on the number, and once on the result of the first function.

The second one is useful for mapping, as it takes only one argument and apply_twice takes two. It is syntactic sugar for square_twice **n** = apply_twice **square n**

Types - functions

```
square : Int -> Int
square n = n^2
```

Types - functions

```
gravity : Float -> Float -> Float
gravity v t = v - 9.81 * t
```

Types - functions

```
square : Int -> Int
square x = x^2

apply_twice : ( Int -> Int ) -> Int -> Int
apply_twice f n = f (f n)

square_twice : Int -> Int
square_twice = apply_twice square
```

Types - naming types

```
type alias Ducks = Int
type alias Elephants = Int

x : Ducks
x = 3

y : Elephants
y = 4

> x * y
12
```

Types - naming types

```
type alias Person =
    { name : String
    , age : Int
    , scores : List Int
    }

twice_age : Person -> Int
twice_age p = p.age * 2
```

Types - record specs

```
bad_commit : { foo | lines : Int } -> Bool
bad_commit c = c.lines > 100

> bad_commit { files=["a.cpp","b.elm"], lines=2
True
```

c.lines = 200 is used outside records. {c | lines = 200 } is used inside records. “.” and “|” are kinda the same. It is also used in Union Types to separate the possibilities.

Types - generic types

```
repeats : List a -> List a
repeats xs =
    case xs of
        h :: t ->
            h :: h :: (repeats t)

        [] ->
            []


> repeats [1,2,3]
[1,1,2,2,3,3]
```

Take a list of any type and return a list of any type. The second case is for an empty list. The first case separates the list in the first item h and the rest of the list t. The body then appends the first item h twice to the recursive function i.e. the same logic applied to the second item.

Here's the catchs. It doesn't apply recursion on a mutated list i.e. [1,1,2,3], but rather the second item in [1,2,3]. Otherwise it would be an infinite loop.

Combining types

```
type Drink = Coffee | Tea

enjoy : Drink -> String
enjoy drink =
    case drink of
        Coffee -> "whoosh"
        Tea -> "aaah"
```

We have a type Drink, and the values can be either **OF TYPE** Coffee OR Tea. Depending on the type, we then return a result. This is very poweful because we can pass in different types of coffee or tea.

Combining types

```
report : Status -> String
report status = case status of
    Available -> "is available now"
    Busy msg -> "is " ++ msg
    Away m -> "back in " ++ (toString m) ++ " m

> status b
"is in a meeting" : String

> status w
"back in 13 mins" : String
```

Html.App.beginnerProgram does the work of routing the messages that your view produces into this function.

Elm for Beginners

http://courses.knowthen.com/p/elm-for-beginners
https://github.com/knowthen/elm/blob/master/DEVSETUP.md

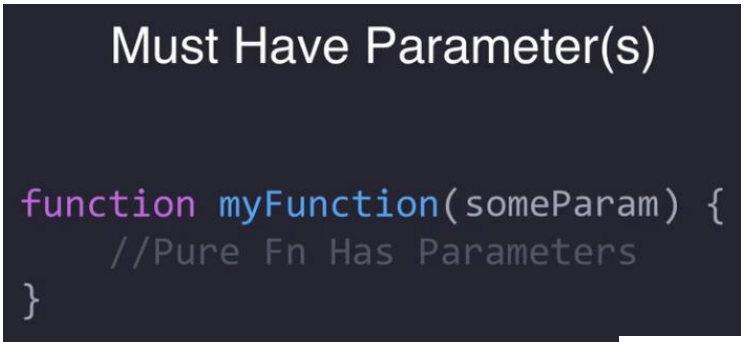
Development Environment Setup Steps

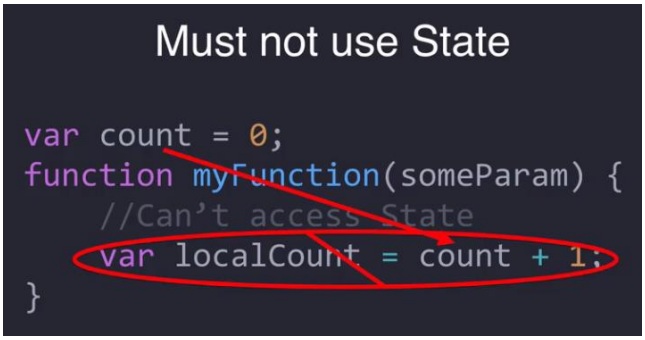
1. Install Recent Version of **Nodejs** by downloading and running the installer found at https://nodejs.org/en/ or use the Node Version Manager found at https://github.com/creationix/nvm
2. Install **Elm** by keying the command `npm install -g elm`
3. Install the atom editor located at https://atom.io/
4. Install the **language-elm** using the atom package manager (apm), by keying the command `apm install language-elm`
5. Install **elm-oracle** by keying `npm install -g elm-oracle`
6. Determine the path where the `elm-oracle` command was installed by keying `which elm-oracle` on mac and unix or **`where.exe elm-oracle`** on windows. Copy the entire path and file name to the clipboard, for use in the next step
7. Open up Atom, then open up the preferences/settings by pressing `CMD + ,` or via the menu. Click `packages` then filter by `elm`. Find the language-elm package and click the `settings` button. Set the `elm-oracle` setting, by pasting the value we copied in the prior step.
8. Download the current version of **elm-format** found at https://github.com/avh4/elm-format
9. Unzip the downloaded file and move the `elm-format` executable to a location in your PATH variable, such as `mv ~/Downloads/elm-format /usr/local/bin/elm-format`
10. Install the `elm-format` Atom Package *(Note: different from elm-format command)*, by keying `apm install elm-format`
11. Start Atom, Open up Settings `CMD + ,`, click `Packages`, filter by `elm`, then click on the `elm-format` package's `settings`button. Set the `elm-format` command path setting and veryify the `format on save` checkbox is checked.
12. Install atom linter by keying `apm install linter`
13. Install the elm linger by keying `apm install linter-elm-make`
14. Locate and copy the path and file for the `elm-make` command by keying the command `which elm-make` for mac or **`where.exe elm-make`** on windows.
15. Open the `linter-elm-make` settings page in atom as you did in steps 7 and 13, then click the settings button next to `linter-elm-make` and then set the `elm-make` setting to the copied value from the prior step.

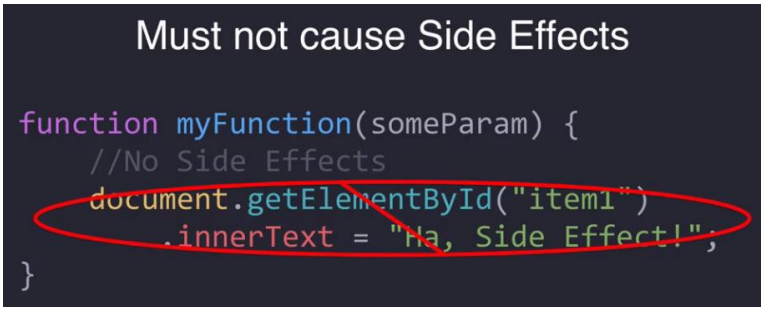
Your atom / elm dev environment should be good to go!

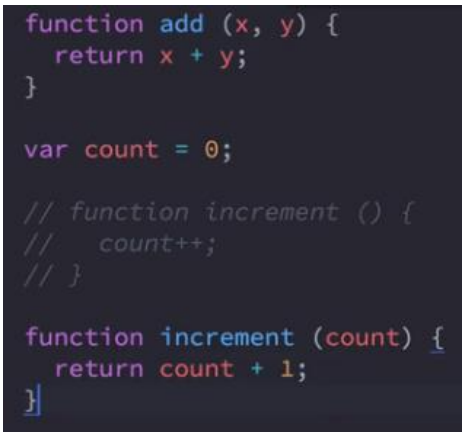
Functional Programming

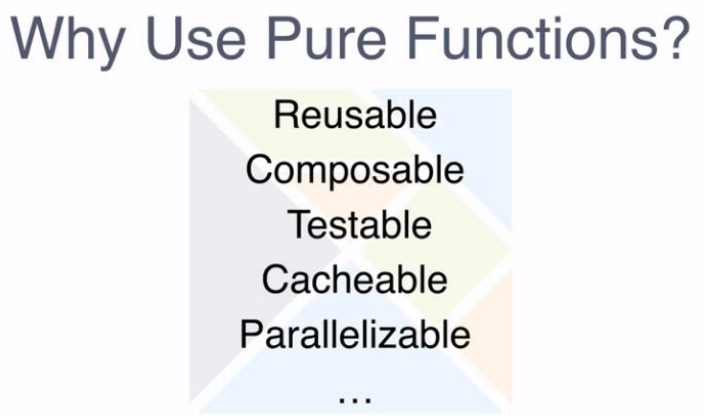
It's a style of programming where only **pure functions** are used i.e. functions that always **return a value** based on **input parameters** without causing **side effects**.





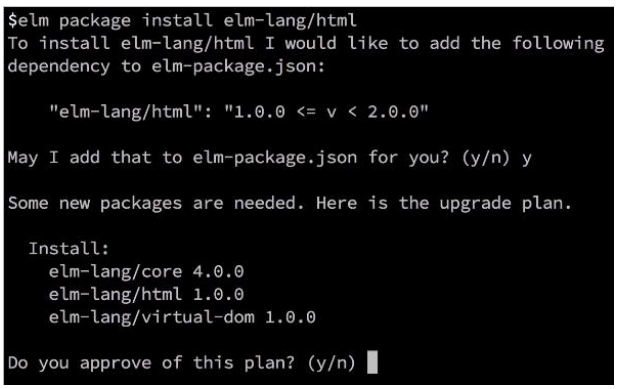






Installing Packages

The **entry point** for elm apps is the **main** function (technically a value, which we set). In order for the code to work, several core packages are required.



This is achieved by running **elm package install elm-lang/html**.

After that, we run **elm reactor** in order to compile the code to javascript and start a web server.

To use the packages, we need to import them in our program

Importing Packages / Modules

The examples bellow are equivalent. We use exposing in order to avoid typing more code.

```elm
import Html exposing (..)

main = text "Hello World"
```

```elm
import Html

main = Html.text "Hello World"
```

Functions

Everything is an expression and the return is implicit.

```elm
functionName param1 param2 =
    -- body
```

```elm
\param1 param2 -> -- Body
```

```elm
add a b = a + b

result = add 1 2

main = text (toString result)
```

Expanding a Function

These are all an equivalent way of adding 3 numbers, with a result of 6.

Normal

```elm
result = add 1 2 3
```

Recursion

```elm
result = add (add 1 2) 3
```

Partial Application and Currying with Named Functions

```elm
-- add : a -> b -> a + b
add3 = add 3
-- add3 : b -> 3 + b
result = add3 1
result == 4
```

Partial Application and Currying with Forward Pipe Operator

```
result = exp |> (val -> b)
result == b
```

The result from the left side becomes an argument on the right. This works exactly like Unix piping.

```
add 1 2 |> (b -> 3 + b)
```

```
result = add 1 2 |> add 3
```

The right side becomes **add left 3** i.e. **add 3 3**.

Partial Application and an Anonymous Expression

```
result = add 1 2 |> \a -> a + 3
```

Local Variables with Let-in block

```
someFunction param =
    let
        -- assignment here
    in
        -- expression here
```

```
counter = 0

increment cnt amt =
    let
        localCount = cnt
    in
        localCount + amt

result = increment counter 1

main = text (toString result)
```

This may seem verbose, but it saves a lot of debugging in the future by not having state side effects.

The increment function may seem like it mutates the counter variable, but in the actual function, the value passed in the first argument **cnt** is copied in a new value **localCount**. After that, we add the second argument to the new variable and return the result.

It might have been easier to use **result = counter + 1**, but this way, we have a function which can have anything passed in.

Functions Exercises

1. Create a function that uppercases names longer than 10 characters with this format: "JAMES MOORE - name length: 11" or "foo bar - name length: 7"

```elm
import Html exposing (..)
import String

display name =
    if String.length name > 10 then
        String.toUpper name
        ++ " - name length: "
        ++ (toString (String.length name))
    else
        name
        ++ " - name length: "
        ++ (toString (String.length name))

result = display "Ivan Veselinovski"

main = text result
```

My solution

```elm
import Html
import String

capitalize maxLength name =
    if String.length name > maxLength then
        String.toUpper name
    else
        name

main =
    let
        name =
            "James Moore"

        length =
            String.length name
    in
        (capitalize 10 name)
            ++ " - name length: "
            ++ (toString length)
            |> Html.text
```

Teacher

Infix Functions

```elm
(~+) a b = a + b + 0.1

result = 1 ~+ 2 -- 3.1
```

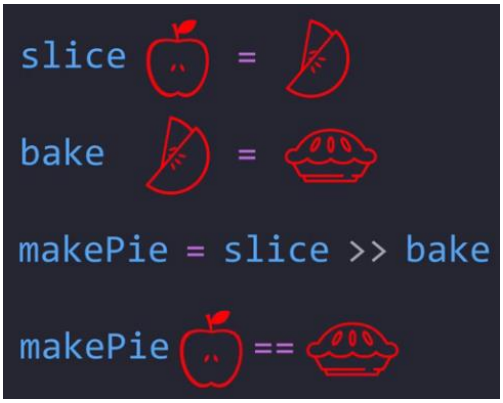
Elm automatically considers functions named with non-alphanumeric characters as infix functions.

Pretty much all the operators are made this way.

Ex. a function **add a b** can be callled with **1 `add` 2** instead of **add 1 2**.

Function Composition

Combine multiple functions into one.



Functions Exercises 2

1. Write an infix function named `~=` that takes two Strings and returns True when the first letter of each string is the same.

```
(~=) a b = String.left 1 a == String.left 1 b

result = "airplane" ~= "anchor"

main = Html.text (toString result)
```

My solution

```
(~=) a b =
    String.left 1 a == String.left 1 b

main =
    "James"
        ~= "Jim"
        |> toString
        |> Html.text
```

Teacher

3. Using function composition, create a function named `wordCount` that returns the number of words in a sentence.

```
granulize sentence = String.words sentence

numerify words = List.length words

wordCount = granulize >> numerify

main = wordCount "The quick brown fox jumps"
        |> toString
        |> Html.text
```

My solution

```
wordCount =
    String.words >> List.length

main =
    "Here is a test sentence"
        |> wordCount
        |> toString
        |> Html.text
```

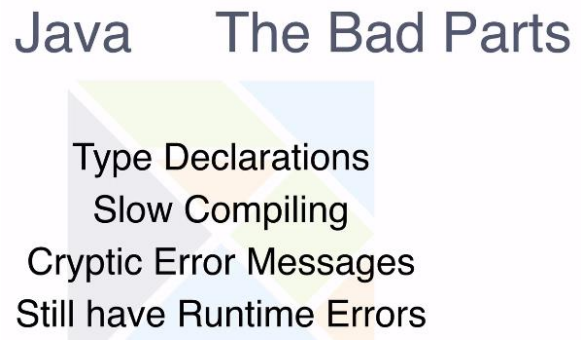
Teacher

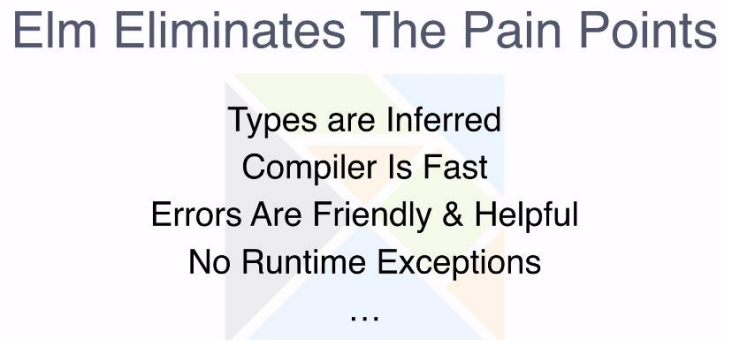
Static Type System

wordCount = String.words >> List.length

```
words : String -> List String

length : List a -> Int

wordCount : String -> Int
```

```
> import String
> wordCount = String.words >> List.length
<function> : String -> Int
> wordCount "Elm's Static Type System is Nice!"
6 : Int
```





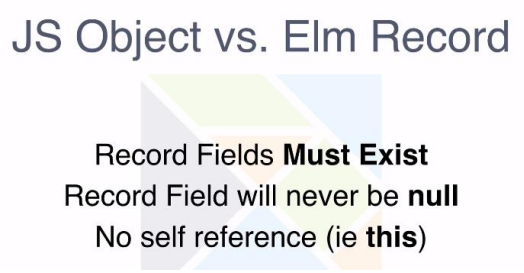
Elm Types

Primitive Types

String, Char, Bool, Number (Float, Int), List a, Touple, Records

```
// JavaScript
var person = {
    name: "James",
    age: 42
}
```

```
-- elm
person =
    { name = "James"
    , age = 42 }
```




```
> person = { name = "James", age = 42 }
{ name = "James", age = 42 } : { age : number, name : String }
> person.name
"James" : String
```

person.name is the same as **.name person**, which can be useful in certain cases.

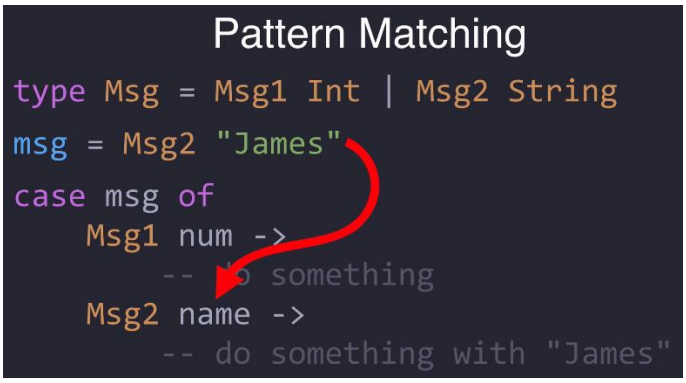
```
> .name person
"James" : String
> olderPerson = { person | age = 43 }
{ name = "James", age = 43 } : { name : String, age : number }
> person
{ name = "James", age = 42 } : { name : String, age : number }
> person == olderPerson
False : Bool
```

The pipe symbol is used to modify properties in a record. It returns a completely new record.

Union Types

```
type SomeType = Type1 | Type2 | Type3
```

```
> type Action = AddPlayer | Score
> action = AddPlayer
AddPlayer : Repl.Action
> type Action = AddPlayer String | Score Int Int
> AddPlayer
<function> : String -> Repl.Action
> Score
<function> : Int -> Int -> Repl.Action
> action = AddPlayer "James"
AddPlayer "James" : Repl.Action
> msg = AddPlayer "James"
AddPlayer "James" : Repl.Action
```




```
> case msg of \
|   AddPlayer name -> "Add player " ++ name \
|   Score id points -> "Player id " ++ (toString id) ++ " scored " ++ (toString points)
"Add player James" : String
```

Maybe vs null

Elm doesn't support null, and to handle the absence of a value, the union type maybe is used.

```
> first = List.head []
Nothing : Maybe.Maybe a
```

```
type Maybe a
    = Just a
    | Nothing
```

```
> case first of \
|   Just val -> val \
|   Nothing -> "Empty"
"Empty" : String
```

```
if(obj !== null){
    console.log(obj.doSomething())
}
```

Checking for null in Javascript.

Maybe.withDefault takes two parameters, the default value if false and the second if true.

```
> first = List.head []
Nothing : Maybe.Maybe a
> Maybe.withDefault "Some Default, because first is Nothing!" first
"Some Default, because first is Nothing!" : String
```

Handling Errors i.e. try/catch

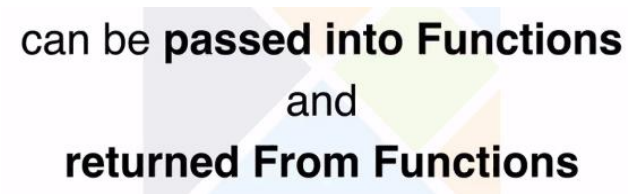
In elm, the Result union type is used, which has two possibilities. Ok if it succeeds, and Err if it fails.

```
type Result error value
    = Ok value
    | Err error
```

```
try {
    let person = JSON.parse(message);
} catch (e) {
    // handle exception
}
```

```
> import Date
> someDate = Date.fromString "01/01/1974"
Ok {} : Result.Result String Date.Date
> case someDate of \
|   Ok val -> "It was a legit date" \
|   Err err -> err
"It was a legit date" : String
> someDate = Date.fromString "Not a Date"
Err "unable to parse 'Not a Date' as a date" : Result.Result String Date.Date
```

Functions are Values




```
> maybeNum = Just 235
Just 235 : Maybe.Maybe number
> add1 a = a + 1
<function> : number -> number
> result = Maybe.map add1 maybeNum
Just 236 : Maybe.Maybe number
```

Maybe.map takes two parameters. The first one is a function, which will be applied to each element in the list passed as the second parameter.

Type Annotations

They describe the **INPUT** and **OUTPUT TYPES** for functions and, **TYPE** for constants. It is not obligatory to define the types as elm automatically infers them. However, it is highly advised to use them.

Not only is the code more readable, but sometimes we want to make the types more restrictive than the iferred ones. Ex. The compiler might infer a number, but we may want int.

```
add : number -> number -> number
add a b =
    a + b
```

```
mass : Int
mass =
    235
```

They help us write more reusable functions.

Let's say we want to apply a 20% discount on each purchase over 5 items.

```
cart =
    [ { name = "Lemon", price = 0.5, qty = 1, discounted = False }
    , { name = "Apple", price = 1.0, qty = 5, discounted = False }
    , { name = "Pear", price = 1.25, qty = 10, discounted = False }
    ]
```

```
fiveOrMoreDiscount item =
    if item.qty >= 5 then
        { item
            | price = item.price * 0.8
            , discounted = True
        }
    else
        item


newCart =
    List.map fiveOrMoreDiscount cart
```

This codes works great, but what happens if we want to offer 30% for purchases over 10 items as well???

Sure, we can write another function tenOrMoreDiscount, but that violates the DRY principle.

With currying and composition, we can do this easily.

```
discount minQty discPct item =
    if not item.discounted && item.qty >= minQty then
        { item
            | price = item.price * (1.0 - discPct)
            , discounted = True
        }
    else
        item
newCart =
    List.map ((discount 10 0.3) >> (discount 5 0.2)) cart
```

Type Aliases

```
cart : List { name : String, price : Float, qty : number, discounted : Bc
cart =
    [ { name = "Lemon", price = 0.5, qty = 1, discounted = False }
    , { name = "Apple", price = 1.0, qty = 5, discounted = False }
    , { name = "Pear", price = 1.25, qty = 10, discounted = False }
    ]
```

```
type alias Item =
    { name : String
    , price : Float
    , qty : Int
    , discounted : Bool
    }
```

```
cart : List Item
cart =
    [ { name = "Lemon", price = 0.5, qty = 1, discounted = False }
    , { name = "Apple", price = 1.0, qty = 5, discounted = False }
    , { name = "Pear", price = 1.25, qty = 10, discounted = False }
    ]
```

Type Exercises

Write the logic necessary to set the `freeQty` of each record using the following logic: Purchases of 5 or more receive 1 free. Purchases of 10 or more receive 3 free. Create a type alias for the cart records and add the appropriate type annotations to all values and functions.

```
type alias Item = {
    name : String,
    qty : Int,
    freeQty : Int
}

cart : List Item
cart =
    [ { name = "Lemon", qty = 1, freeQty = 0 }
    , { name = "Apple", qty = 5, freeQty = 0 }
    , { name = "Pear", qty = 10, freeQty = 0 }
    ]

freeItems : Int -> Int -> Item -> Item
freeItems minQty free item =
    if item.qty >= minQty then
        {item
            | freeQty = free}
    else
        item

result : List Item
result = List.map ((freeItems 5 1) >> (freeItems 10 3)) cart

main : Html.Html msg
main = result
        |> toString
        |> Html.text
```

My solution

```
cart =
    [ { name = "Lemon", qty = 1, freeQty = 0 }
    , { name = "Apple", qty = 5, freeQty = 0 }
    , { name = "Pear", qty = 10, freeQty = 0 }
    ]

free minQty freeQty item =
    if item.freeQty == 0 && minQty > 0 then
        { item | freeQty = item.qty // minQty * freeQty }
    else
        item

main =
    List.map ((free 10 3) >> (free 5 1)) cart
        |> toString
        |> Html.text
```

Teacher

Input Fields

```
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (..)
import Html.App as App
```

This handles all the dependecies requred for The Elm Architecture.

```
main = App.beginnerProgram
    { model = model, view = view, update = update }
```

beginnerProgram connects all the pieces behind the scenes. It's a light version of **Program**.

```
model = { content = "" }
```

We use a record to define our model.

```
type Msg = Change String

update msg model =
  case msg of
    Change newContent ->
      { model | content = newContent }
```

The update functions listens for a message and if the type is Change, the model is updated with the new value passed in from the input field

The view updated the DOM automatically.

```
view model =
  div []
    [ input [ onInput Change ] []
    , div [] [ text model.content ]
    ]
```

On each keystroke an event is fired which dispatches the **Change** message which activates the **update** function.

Parts of an Elm App

Model

```
type alias Model = Int

initModel : Model
initModel = 0
```

It's a container that holds the raw data your application needs. It can be a primitve type (Int, String) or a Record. We are describing what the model looks like and what the initial value is.

Update

It's responsible for updating the model i.e. state. It has two concerns:

1. What things can happen in the app?
2. When they happen, how should the model change?

```
-- what actions are possible?
type Msg = AddCalorie | Clear

update : Msg -> Model -> Model
update msg model =
    -- TODO: update Model
```

```
type Msg = AddCalorie | Clear

update msg model =
    case msg of
        AddCalorie -> model + 1
        Clear -> initModel
```

These concerns are handled by two things:

1. A union type representing all the possible things that can happen.
2. An **update** function that takes a **message** (One of the defined union types) and a **model** (current state of the app. It returns and updated model.

View

A functions that takes a model and generates Html.

```
-- What should the view be
-- considering the model?
view : Model -> Html Msg
view model =
    -- TODO: Create a view
```

```
-- Consistent Signature
tagName -- div, p, button etc…
    : List (Attribute msg)-- id, class…
    -> List (Html msg)-- text, div, p…
    -> Html msg
```

```
-- Elm                      <!-- Html  -->
div [] []                   <div></div>
p [ class "story" ]         <p class="story">
    [ text "Once" ]         Once</p>
```

```
view : Model -> Html Msg
view model =
    div []
        [ h3 []
            [ text ("Total Calories: "
                ++ (toString model)) ]
        , button
            [ type' "button"
            , onClick AddCalorie ]
            [ text "Add" ]
        , button
            [ type' "button"
            , onClick Clear ]
            [ text "Clear" ]
    ]
```

App

```
main : Program Never
main = App.beginnerProgram { model = initModel, update = update, view = view }
```

We set **main** to be equal to the value returned by calling the **App.beginnerProgram function** which takes a record as a parameter where model = initModel value, update = update function and view = view function.

Connecting them with Html.App.beginnerProgram

We do this by using the **beginnerProgram** funciton and passing the **initial model**, the **view** and the **update** function. It links them all together into a working app.

Html.App takes the responsibility of maintaining state, listening for messages, and calling the update and view sections as appropriate.

```
main = Html.beginnerProgram { model = model, view = view, update = update }
```

```
beginnerProgram : { model : model, view : model -> Html msg, update : msg -> model -> model }
    -> Program Never
```

If you are using **elm-reactor** you will just see everything on screen. If you are using **elm-make** it will be generating HTML files that you can open in any browser.

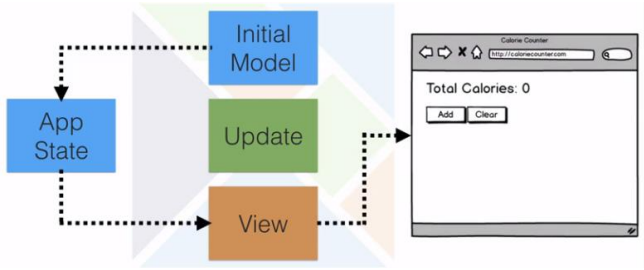
Create a Program that specifies how your whole app should work. **The essense of every Elm program is:**

- Model — a nice representation of your application state.
- View — a function that turns models into HTML.
- Update — given a user-input messages and the model, produce a new model.

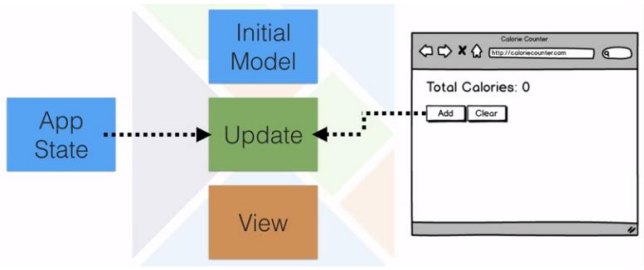
Between these three things, you have everything you need!

1. When the user clicks on a button, it produces a message.
2. That message is piped into the update function, producing a new model.
3. We use the view function to show the new model on screen.
4. And then we just repeat this forever!

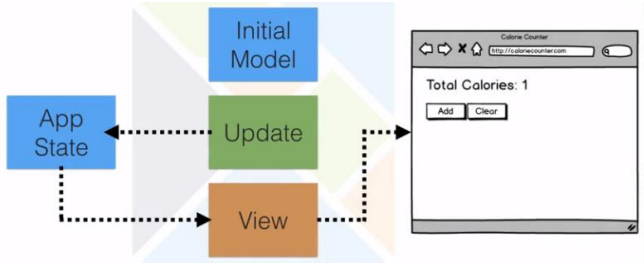
App Flow



When the app first loads, the **initialModel** sets the current state of the app. Then the view function is called with the current state i.e. **model** and it generates a view showing the initialModel values.



When one of the buttons is clicked, the message it generates is passed into the **update** function along with the current state of the app i.e. **model**, and a new model is generated i.e. returned.



The **model** returned from the update funciton is stored as the new application state i.e. **model** and then the **view** function is called with the updated model which displays the new values.

Application state is handled by the Html.App package.

Simple App Exercise – Calorie Counter

```
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (..)
import Html.App as App
import String exposing (..)
```

```
type alias Model = { calories : Int, input : Int }

initModel : Model
initModel = { calories = 0, input = 0 }
```

```
update : Msg -> Model -> Model
update msg model =
    case msg of
        AddCalorie ->
            { model
                | calories = model.calories + model.input
                , input = 0 }

        Input input ->
            case String.toInt input of
                Ok input -> { model | input = input }
                Err err -> { model | input = 0 }

        Clear -> initModel
```

```
view : Model -> Html Msg
view model =
    div []
        [ h3 []
            [ text ("Total Calories: " ++ ( toString model.calories ) ) ]
        , input
            [ placeholder "Insert Calories..."
            , onInput Input
            , value
                (if model.input == 0 then "" else toString model.input ) ] []
        , button
            [ type' "button"
            , onClick AddCalorie ]
            [ text "Add" ]
        , button
            [ type' "button"
            , onClick Clear ]
            [ text "Clear" ]
        , p [] [ text ( toString model )]
    ]
```

```
main : Program Never
main = App.beginnerProgram { model = initModel, update = update, view = view }
```

Elm Build Process

Elm make

https://github.com/elm-lang/elm-make

Before compiling, **elm-make** will ask to install the latest dependencies. This will create a **elm-package.json** file as well as an **elm-stuff** directory with all the packages inside. This is pretty much like using **npm install** which creates a **node_modules** directory based on the **package.json** file.

elm-make Main.elm --output=main.**html**

This will create an HTML file with all the JS inside a script tag in the head.

elm-make Main.elm --output=main.**js**

This will create a JS file with just Javascript inside.

The second method is better because the file can simply be added to a custom HTML which will not be overwritten in the build process.

A task runner can be used to automate this process along with starting a server.

Elm seed

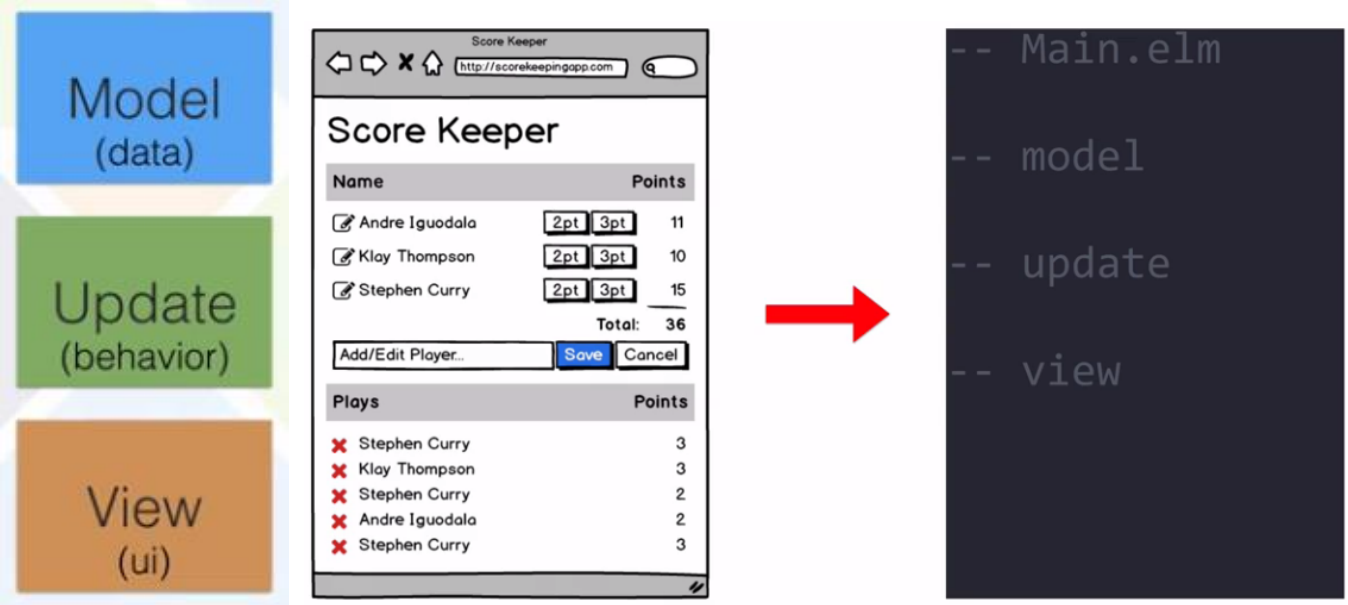
To simplify the starting process, we can create a boilerplate HTML and CSS file which would be reused in different projects. The only thing needed is to add the compiled JS file into the HTML one.

```
<!doctype html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Basketball Score Keeper</title>
  <link rel="stylesheet" href="app.css">
</head>

<body>
  <div id="app"></div>
  <script src="bundle.js"></script>
  <script>
    var app = Elm.Main.embed(document.getElementById("app"));
  </script>
</body>

</html>
```

For a nicer structure, we can move all the Elm components into a src directory (by modifying the elm-package.json file with source-directories: [“src”]) and use the following command to compile in another directory:

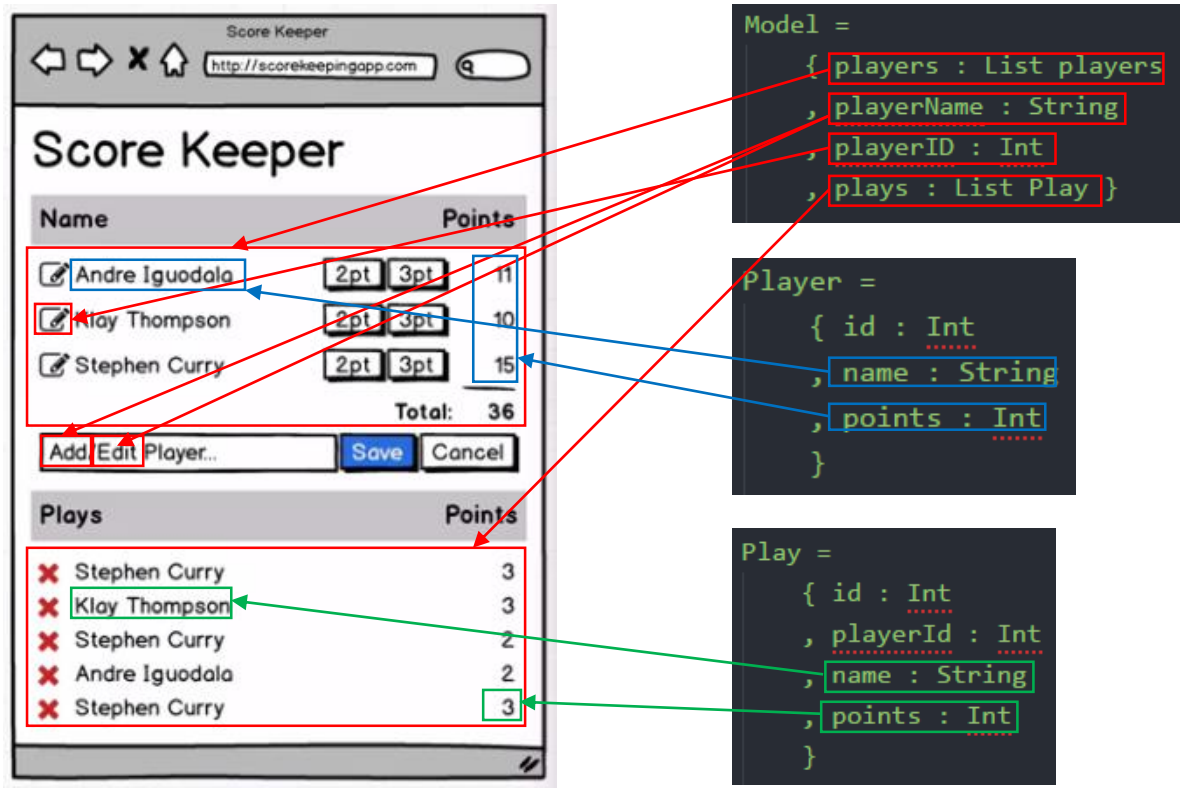
elm-make Main.elm --output=js/main.**js**

Score Keeper App

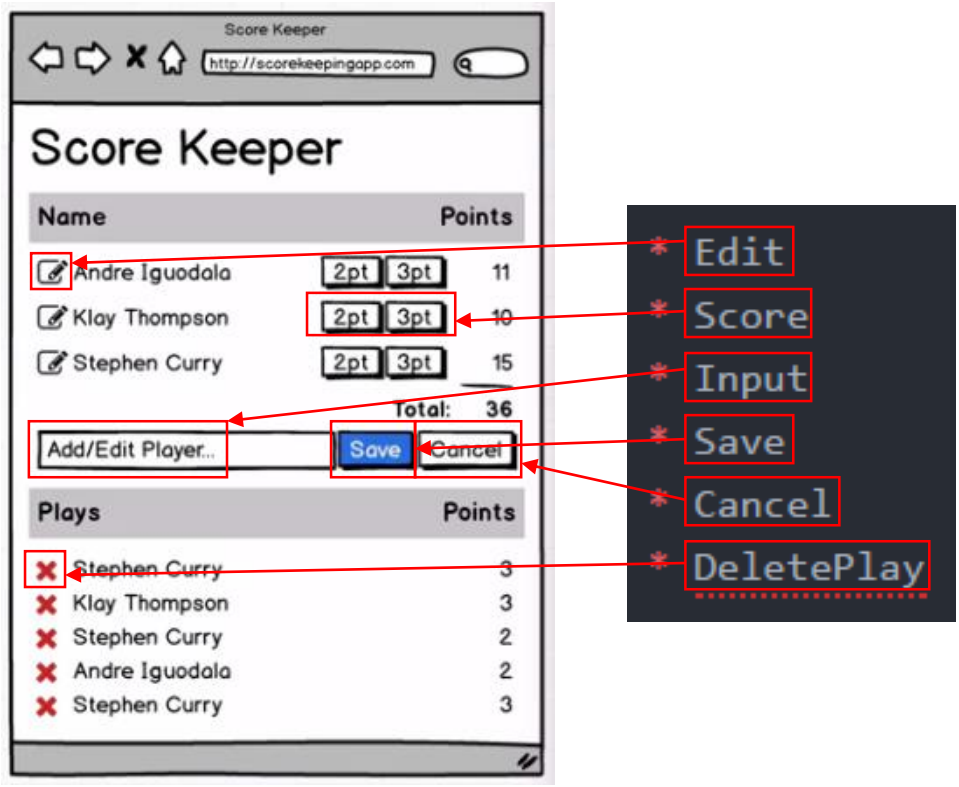


Planning

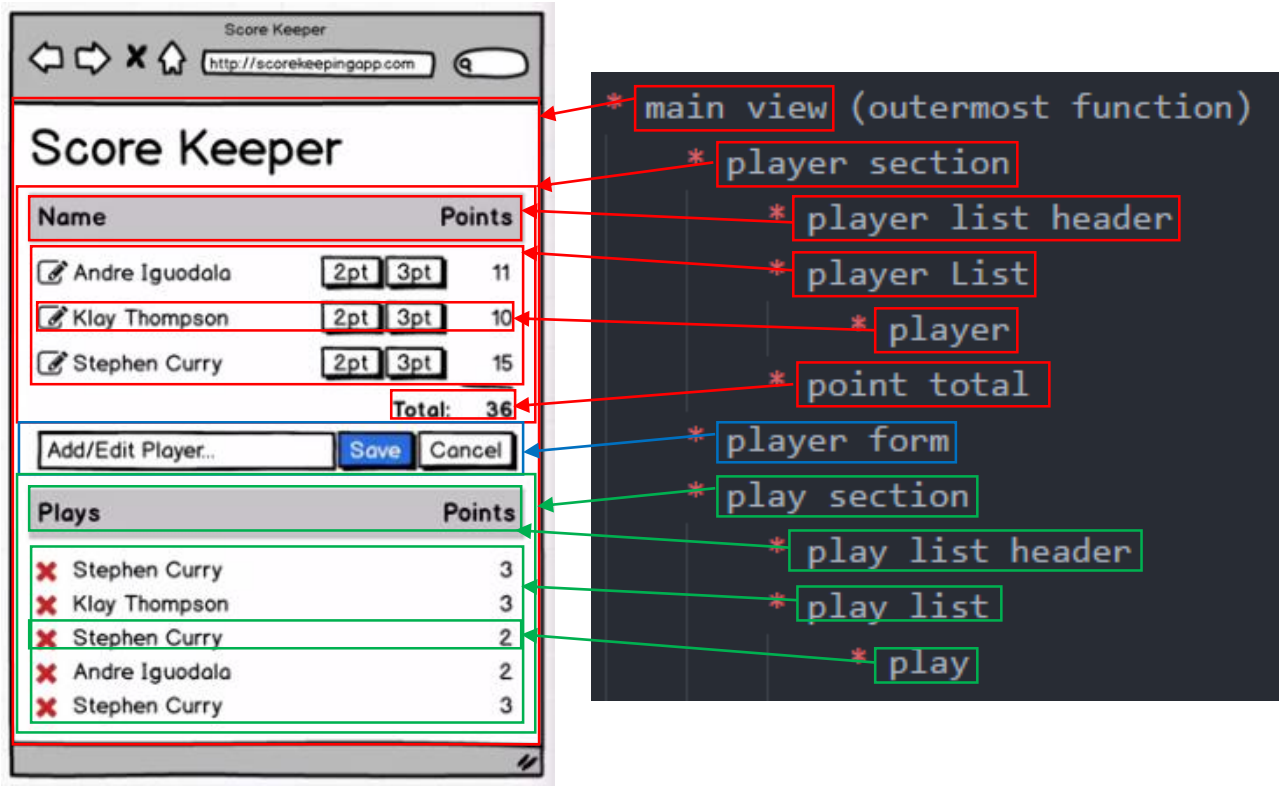
Model



Update


Score Keeper
http://scorekeepingapp.com
Score Keeper
Name
Points
Andre Iguodala
2pt
3pt
11
Klay Thompson
2pt
3pt
10
Stephen Curry
2pt
3pt
15
Total:
36
Add/Edit Player...
Save
Cancel
Plays
Points
Stephen Curry
3
Klay Thompson
3
Stephen Curry
2
Andre Iguodala
2
Stephen Curry
3
Edit
Score
Input
Save
Cancel
DeletePlay

View


Score Keeper
http://scorekeepingapp.com
Score Keeper
Name
Points
Andre Iguodala
2pt
3pt
11
Klay Thompson
2pt
3pt
10
Stephen Curry
2pt
3pt
15
Total:
36
Add/Edit Player...
Save
Cancel
Plays
Points
Stephen Curry
3
Klay Thompson
3
Stephen Curry
2
Andre Iguodala
2
Stephen Curry
3
main view (outermost function)
player section
player list header
player List
player
point total
player form
play section
play list header
play list
play

Coding

It’s best if we break down the application into small pieces and get them working one by one. We always start by defining the model.

Elm Beyond the Basics

Review

Model

```
-- model
type alias Model =
    { username : String
    , password : String
    }
initModel : Model
initModel =
    { username = ""
    , password = ""
    }
```

Update

```
-- update
type Msg
    = UsernameInput String
    | ButtonPress

update msg model =
    case msg of
        ButtonPress ->
            ...
```

View

```
-- view
view model =
    div [] [
        ...
    ]
```

Simple Login Module

Username | Password | Login

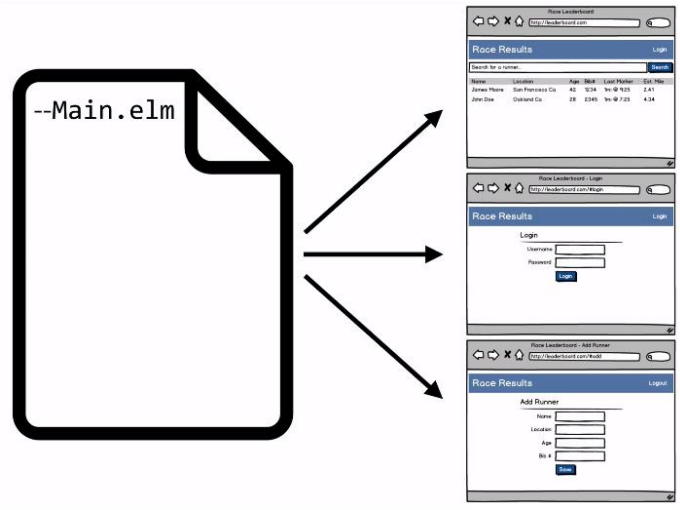
```
type alias Model =
    { username : String
    , password : String
    }

initModel : Model
initModel =
    { username = ""
    , password = ""
    }
```

```
type Msg
    = UsernameInput String
    | PasswordInput String

update : Msg -> Model -> Model
update msg model =
    case msg of
        UsernameInput username ->
            { model | username = username }
        PasswordInput password ->
            { model | password = password }
```

```
view : Model -> Html Msg
view model =
    div []
        [ Html.form []
            [ input
                [ type' "text"
                , onInput UsernameInput
                , placeholder "Username"
                ]
                []
            , input
                [ type' "text"
                , onInput PasswordInput
                , placeholder "Password"
                ]
                []
            , button [ type' "submit" ]
                [ text "Login" ]
            ]
        ]
```

```
main : Program Never
main = App.beginnerProgram { model = initModel, update = update, view = view }
```

The Elm Architecture

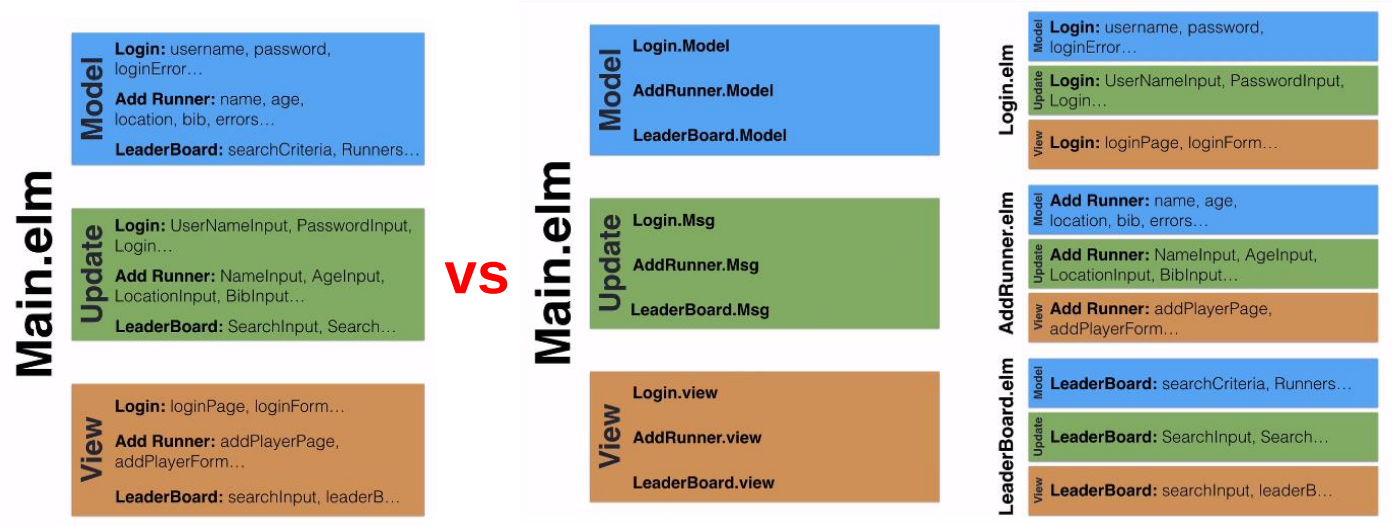


It's totally possible to wrote an application with three pages in a single module, but that is not a good idea because the code would get ugly pretty fast.

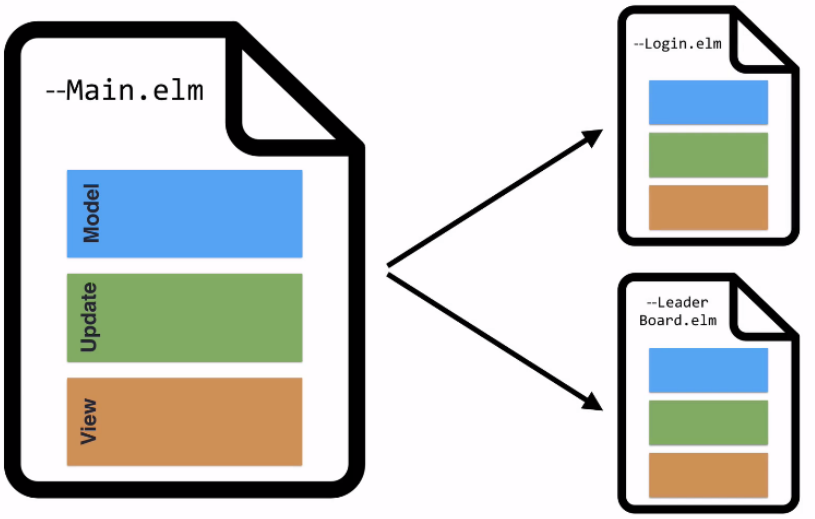
This is where The Elm Architecture really shines.

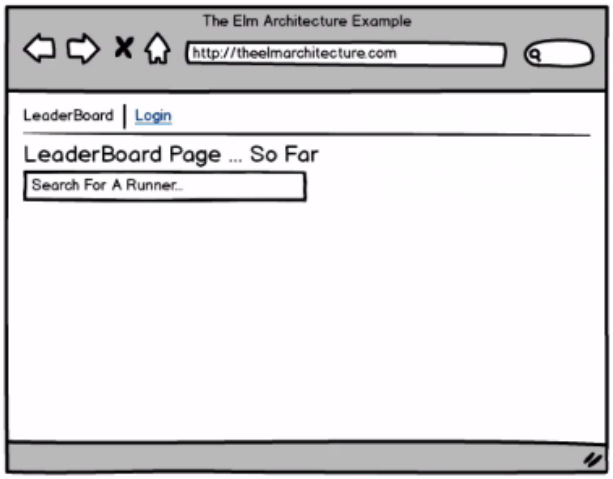
The application is divided into logical modules which are imported and tied together in the Main module.

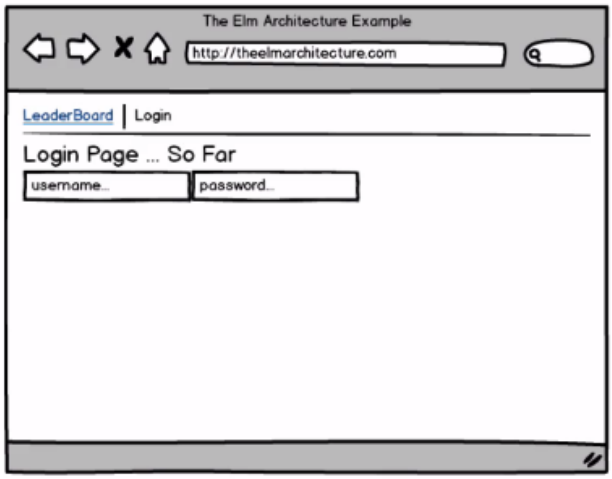
TEA is a nice pattern that can be used all over the place… But should it? In order to avoid over-engineering things, the pattern should be used only when absolutely needed. **Start with a single Main module and later if needed, refactor the code into separate modules**, which is easy to do.



Example







Before

Both modules have the import and main parts when standalone, but they are removed for export purposes.

```
import Html exposing (..)
import Html.Events exposing (..)
import Html.Attributes exposing (..)
import Html.App as App
```

```
main : Program Never
main =
    App.beginnerProgram
        { model = initModel
        , update = update
        , view = view
        }
```

```
module Login exposing (..)

type alias Model =
    { username : String
    , password : String
    }

initModel : Model
initModel =
    { username = ""
    , password = ""
    }

type Msg
    = UsernameInput String
    | PasswordInput String

update : Msg -> Model -> Model
update msg model =
    case msg of
        UsernameInput username ->
            { model | username = username }
        PasswordInput password ->
            { model | password = password }

view : Model -> Html Msg
view model =
    div []
        [ h3 [] [text "Login Page"]
        , Html.form []
            [ input
                [ type' "text"
                , onInput UsernameInput
                , placeholder "Username"
                ]
                []
            , input
                [ type' "text"
                , onInput PasswordInput
                , placeholder "Password"
                ]
                []
            , button [ type' "submit" ]
                [ text "Login" ]
            ]
        ]
```

```
module LeaderBoard exposing (..)

type alias Model =
    { runners : List Runner
    , query : String
    }

type alias Runner =
    { id : Int
    , name : String
    , location : String
    }

initModel : Model
initModel =
    { runners = []
    , query = ""
    }

type Msg
    = QueryInput String

update : Msg -> Model -> Model
update msg model =
    case msg of
        QueryInput query ->
            { model | query = query }

view : Model -> Html Msg
view model =
    div []
        [ h3 [] [ text "Leaderboard Page" ]
        , input
            [ type' "text"
            , onInput QueryInput
            , value model.query
            , placeholder "Search runner"
            ]
            []
        , hr [] []
        , h4 [] [ text "Leaderboard Model:" ]
        , p [] [ text <| toString model ]
        ]
```

After

```
module Main exposing (..)

import Html exposing (..)
import Html.Events exposing (..)
import Html.Attributes exposing (..)
import Html.App as App

import Login
import LeaderBoard
```

A message lbMsg originated from some interaction on the LeaderBoardPage. The Main module doesn't know much about the LeaderBoard messages, except that they exist. LeaderBoard module's.update function is fully aware of the different types of messages within its module and it knows how to update the model based on them. **We're just calling the LeaderBoards module's update function and we provide the correct parameters (lbMsg and model.leaderBoard)**, and in the end store the potentially updated leaderboard model into the corresponding leaderboard field (constant).

```
type alias Model =
    { page : Page
    , leaderBoard : LeaderBoard.Model
    , login : Login.Model
    }

initModel : Model
initModel =
    { page = LeaderBoardPage
    , leaderBoard = LeaderBoard.initModel
    , login = Login.initModel
    }

type Page
    = LeaderBoardPage
    | LoginPage
```

```
type Msg
    = ChangePage Page
    | LeaderBoardMsg LeaderBoard.Msg
    | LoginMsg Login.Msg

update : Msg -> Model -> Model
update msg model =
    case msg of
        ChangePage page ->
            { model | page = page }

        LeaderBoardMsg lbMsg ->
            { model | leaderBoard = LeaderBoard.update lbMsg model.leaderBoard }

        LoginMsg loginMsg ->
            { model | login = Login.update loginMsg model.login }
```

```
update : Msg -> Model -> Model
update msg model =
    case msg of
        QueryInput query ->
            { model | query = query }
```

```
view : Model -> Html Msg
view model =
    let
        page =
            case model.page of
                LeaderBoardPage ->
                    App.map LeaderBoardMsg
                        (LeaderBoard.view model.leaderBoard)

                LoginPage ->
                    App.map LoginMsg
                        (Login.view model.login)
    in
        div []
            [ a
                [ href "#"
                , onClick (ChangePage LeaderBoardPage)
                ]
                [ text "LeaderBoard" ]
            , span [] [ text " | " ]
            , a
                [ href "#"
                , onClick (ChangePage LoginPage)
                ]
                [ text "Login" ]
            ]
        ]
```

```
main : Program Never
main =
    App.beginnerProgram
        { model = initModel
        , view = view
        , update = update
        }
```

```
LeaderBoardPage ->
    LeaderBoard.view model.leaderBoard
```

We are transforming all the LeaderBoard messages into Main messages.

This is done because we are calling the view function in that module (LeaderBoard), which returns messages that won't compile since they are different from the ones outputted in this module (Main).

It is further explained on the next page.

Fixing The Different Message Types?

Hint:

How do we transform…

```
List a   ->  List b
Maybe a  ->  Maybe b
```

Answer:

Map

```
Html a   ->  Html b
Map : ( a -> b) -> Html a -> Html b
```

We need to transform / map from Html capable of generating LeaderBoard Messages into Html capable of generating Main Messages.

```
Html LB.Msg  ->  Html Main.Msg

Map : ( LB.Msg -> Main.Msg) ->
  Html LB.Msg -> Html Main.Msg
```

```
map : (a -> msg) -> Html a -> Html msg
```

Map takes 2 parameters. The first one is a transformation function which takes one parameter of type a (Html) and it returns a message. The second parameter is Html capable of generating messages of type a.

This is the same type the transformation function takes as a parameter. The map function ultimately returns an Html capable of generating a slightly different type of messages.

```
view : Model -> Html Msg
view model =
    let
        page =
            case model.page of
                LeaderBoardPage ->
                    LeaderBoard.view model.leaderBoard
```

```
view : Model -> Html Msg
view model =
    let
        page =
            case model.page of
                LeaderBoardPage ->
                    App.map LeaderBoardMsg
                        (LeaderBoard.view model.leaderBoard)
```

Simply put, we are calling the view function inside the LeaderBoard module and we pass it the leaderBoard model as a parameter. However, this willl not compile becase the view function returns messages from that module, while we are calling the function inside the Main module.

In order to fix this, we must change those messages into Main messages.

Effects i.e. Talking to Severs

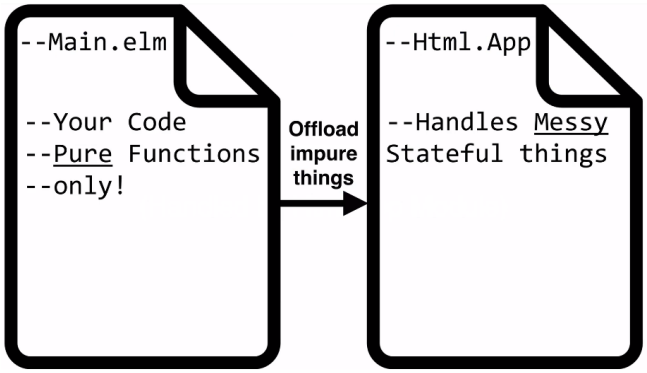
Effects are "**Talking to the Outside World**" i.e.:

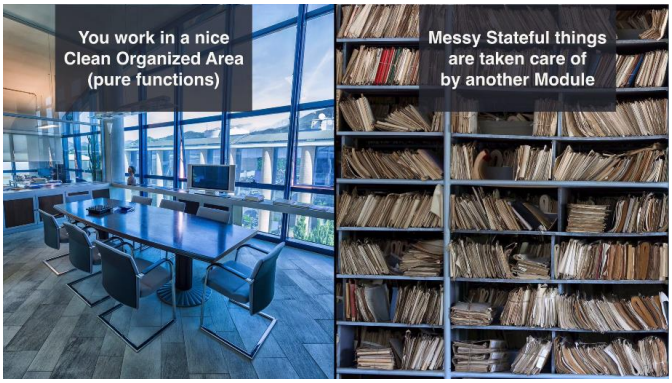
- Websockets
- Http Requests
- Local Storage

The problem is, in Elm, you can only write pure functions, but you need to do impure things…

Handling effects is similar to handling state (**offloading the responsibility of maintaining state to the Html.App module**).

We tightly control where the complicated and messy parts of our app happen, leaving our app cleaner and easier to work with.





Maintaining State in Elm

```
update : Msg -> Model -> Model
update msg model =
    case msg of
      -- change model as needed

...

main : Program Never
main =
    Html.App.beginnerProgram
        { update = update
        ...
        }
```

We write an update function which can be seen as a set of rules for how the model should change when things happen in the app.

We then pass the update function (rules) to the thing maintaining the state, the **Html.App module**, which uses the update function as needed to update the app's state.

Effects are handled in a very similar way…

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        AddRunner runner ->
            let
                -- pseudo code
                command = "Add This runner ..."
            in
                ( model, command )
```

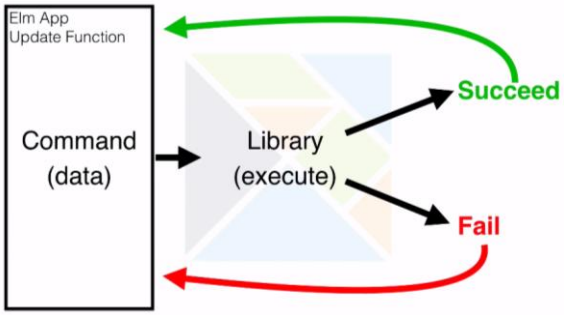
Let's say our app needs to talk to a server to add something in a database. What if we describe what we want done as a piece of data or a command? And then we return that command from the function.

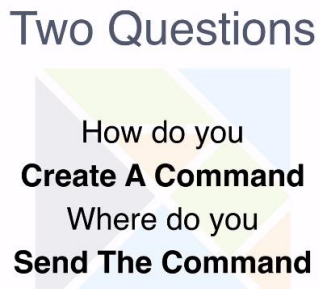
Doing this allows us to have purity in the functions because we just return data. However, another module can receive the command and act on it i.e. do the impure things.

Analogy

A mob boss wants to do some dirty work. Instead of doing it himself (messing up his house and dirtying his hands), he gives commands to someone and the work happenes in some remote place. Note: Just because a command was issued, doesn't mean it'll happen. The command needs to be sent to the right person and the person will either succeed or fail.

The flow of issuing a command that succeeds or fails is exactly the flow for many effects in Elm.





Http Requests

How do we turn this basic app (**Left**) into one that can do Http requests (**Right**)?

```elm
module Main exposing(..)

import Html exposing (..)
import Html.App as App

type alias Model = String

initModel : Model
initModel = "Finding a joke..."

type Msg
    = Joke String

update : Msg -> Model -> Model
update msg model =
    case msg of
        Joke joke -> joke

view : Model -> Html Msg
view model =
    div [] [ text model ]

main : Program Never
main =
    App.beginnerProgram
        { model = initModel
        , update = update
        , view = view
        }
```

```elm
module Main exposing(..)

import Html exposing (..)
import Html.App as App
import Http
import Task

randomJoke : Cmd Msg
randomJoke =
    let
        url = "http://api.icndb.com/jokes/random"
        task = Http.getString url
        cmd = Task.perform Fail Joke task
    in
        cmd

type alias Model = String

initModel : Model
initModel = "Finding a joke..."

init : ( Model, Cmd Msg )
init = ( initModel, randomJoke )

type Msg = Joke String | Fail Http.Error

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Joke joke -> ( joke, Cmd.none )
        Fail error -> ( ( toString error ), Cmd.none )

view : Model -> Html Msg
view model =
    div [] [ text model ]

subscriptions : Model -> Sub Msg
subscriptions model = Sub.none

main : Program Never
main =
    App.program
        { init = init
        , update = update
        , view = view
        , subscriptions = subscriptions
        }
```

Commands

We do this by creating commands (which is just data) and sending them to a place where the actual effects can happen.

How do we **Create a Command** and **Where Do We Send it**?

Before anything, we need to install the necessary package by using elm package install **evancz/elm-http** and import it.

After that, we create a constant randomJoke which is a **command capable of generating messages**.

```
randomJoke : Cmd Msg
```

similar to

```
view : Model -> Html Msg
```

Both can generate messages.

For the actual Http request, we will use the getString function which takes a URL (API) as a parameter (string) and returns a **Task** that can FAIL with an Error of an Http.Error type, or SUCCEED with a certain type, in this case a string. Please note that this example omitts decoding JSON and returns it as a String.

```
getString : String -> Task Error String
```

A Task is an alias type which comes from the Task module in the core package.

We can think of Tasks as something used to describe asynchronous operations that can fail.

Both of these are equal.

```
randomJoke : Cmd Msg
randomJoke =
    let
        url = "http://api.icndb.com/jokes/random"
        task = Http.getString url
        cmd = Task.perform Fail Joke task
    in
        cmd
```

```
randomJoke : Cmd Msg
randomJoke = Task.perform Fail Joke (Http.getString "http://api.icndb.com/jokes/random")
```

How does our app know when user interactions happen? Via messages generated from the Html and sent to the update function.

```
view : Model -> Html Msg
view model =
    div [] [
        button [ onClick ClickMsg ] []
        input [ onInput InputMsg ] []
    ]
```

```
update : Msg -> Model -> Model
update msg model =
    case msg of
        ClickMsg ->
            ...
        InputMsg input ->
            ...
```

The same message mechanism is used to communicate an Http failure or success.

```
randomJoke : Cmd Msg
randomJoke =
    let
        url =
            "http://..."

        task =
            Http.getString url
```

```
update : Msg -> Model -> Model
update msg model =
    case msg of
        HttpFailure err ->
            ...
        HttpSuccess response ->
            ...
```

How do we translate Task failures and successes into corresponding fail / success messages that our app can deal with?

This is done by **transforming our Task into a Command** which is done **by calling the Task module's perform function**.

A **command** is **just data** that **describes** what we **want to do** and how to **deal with** possible **outcomes**.

```
perform : (x -> msg) -> (a -> msg) -> Task x a -> Cmd msg
```

```
randomJoke : Cmd Msg
randomJoke = Task.perform Fail Joke (Http.getString "http://api.icndb.com/jokes/random")
```

The perform function takes three parameters.

1. The first parameter is the function that takes the ERROR value from the task if an error occurred and returns a message our app understands.
2. The second parameter is the function that takes the SUCCESS value from the task if it's successful and returns a message our app understands.
3. The third parameter is the task itself i.e. the Http request.

We can think of a command as a data structure that contains a task that can either fail and succeed, and application specific messages used to communicate failure and success back to the app.

First parameter is a fail message sent to the update function. Second one is a success message sent to the update function. Both messages are the result of a function that reads the response from the Http request. The third parameter is the Http request that returns a response (fail or success).

```
type Msg = Joke String | Fail Http.Error

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Joke joke -> ( joke, Cmd.none )
        Fail error -> ( ( toString error ), Cmd.none )
```

We can think of commands as a list of household chores that a parent might make for their child. The commands are chores like clean room, wash dishes… But if the list of chores is never passed along to the child, the chores would never get done.

Similarly, the **randomJoke command** (which is just a constant) **will never be executed** unless we **pass it along** something that **can execute it**.

To use commands, we need to switch from using the **beginnerProgram** function to the **program** function.

Program

The program function takes a record parameter like the beginnerProgram one, with a slight difference.

init

The value assigned to the **init** field should be a touple with the **initialModel** as the **first value**, and the **second value** is **one or more commands** that should be run when the **app first loads**. This is similar to React's **componentDidMount**. We can use the randomJoke command to make a request and get a joke when the app loads without any user interaction.

update

While update still takes a message and a model as the first two parameters, the return type is a bit different, as it now includes commands. this is the answer to the "How do we hand our commands to something that can run them?".

This is done by simply specifying the commands we like executed in the init field and as part of the return value of the update function.

When the Http.App module receives commands, it executes the commands and uses the success and error message types, we specified earlier with the task module's perform function, to notify our app of the task's outcome, so we can deal with it.

Refresh (New Joke) Button

It's logical to do this by adding a button with **randomJoke onClick**. However, this doesn't work as the return type is a command, and view returns Html. So, we need to create a new message called **NewJoke** in which we issue the command.

```
type Msg = Joke String | Fail Http.Error | NewJoke

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Joke joke -> ( joke, Cmd.none )
        Fail error -> ( ( toString error ), Cmd.none )
        NewJoke -> ("Fetching new joke...", randomJoke)

view : Model -> Html Msg
view model =
    div []
        [  button [ onClick NewJoke ] [ text "New Joke" ]
        , hr [] []
        , text model
        ]
```

Decoding JSON

How do we turn a JSON response into an Elm value?

```
// some JSON
{
    "name" : "James",
    "age" : 42,
}
```

```
// JavaScript
let person = JSON.parse(someJson);

person.name === "James";
person.age === 42;
```

```
-- pseudocode
{-
   Expect:
       "name" that's a "String"
       "age" that's an "Int"
-}
decoder : Decode Person
...

{-
    Check Expectations
-}
personResult : Result Error Person
personResult =
    decodeString decoder someJson
```

Elm is a bit more verbose as decoding JSON requires describing what the expected properties and their types are by using a **Decoder**…

And later checking if the JSON conforms to the expectations by calling a function **decodeString** and passing the decoder and the JSON.

If the JSON contains the properties and values we expect, the JSON is transformed into an Elm value (Person) we can use.

If it doesn't match, we get an error.

Expectation

```
at : List String -> Decoder a -> Decoder a
```

```
import Json.Decode exposing (..)

json : String
json =
    """
{
    "type": "success",
    "value":{
        "id":496,
        "joke":"Chuck Norris joke...",
        "categories":[
            "nerdy"
        ]
    }
}
"""
```

```
decoder : Decoder String
decoder = at ["value", "joke"] string

jokeResult : Result String String
jokeResult = decodeString decoder json

main : Html msg
main =
    case jokeResult of
        Ok joke -> text joke
        Err err -> text err
```

For simplification, we simulate a JSON response as a string. To start, we first import the Json.Decode module and we describe the decoder.

The decoder will be a decoder of strings i.e. we are decoding the JSON into a simple string or more specifically, we are decoding the joke property in the JSON.

To create the decoder, we will use the **at** function in the Json.Decode module which lets us access a nested field in a JSON value. **at** takes a **list of strings** and a **decoder of type a**. The list of strings is a **"path"** to the value we want decoded.

The at function also takes a type specific decoder as the last parameter which simply says that the value found at the path's end should be a whatever type, in this case a string. The string decoder is part of the Json.Decode module.

The decoder simply says: **"You should find a property at value > joke that's a string."**

The next step is the actual decoding

Decoding

We set a constant jokeResult to be equal to the value returned by calling decodeString (Json.Decode). decodeString takes a decoder and a value (JSON) to decode.

Since decoding can fail, the type of the joke constant is the Result type where the error is a string and the result is a string.

Shorter Way

Decoding HTTP Response (Long Way)

```
decoder : Decode String
...

jokeResult : Result Error Person
jokeResult =
    decodeString decoder httpJsonResponse
```

Instead of creating a decoder and calling decodeString, we can use a shorter and simpler way by using the **Http module's get function**.

```
get : Decoder value -> String -> Task Error value
```

get takes two parameters. A **decoder** and a **URL to make the get request**. The **actual decoding** step is **handled by the get function** so we **don't need to call getString** ourselves.

```
randomJoke : Cmd Msg
randomJoke =
    let
        url = "http://api.icndb.com/jokes/random"
        task = Http.get (at [ "value", "joke" ] string) url
        cmd = Task.perform Fail Joke task
    in
        cmd
```

```
randomJoke : Cmd Msg
randomJoke =
    let
        url = "http://api.icndb.com/jokes/random"
        task = Http.getString url
        cmd = Task.perform Fail Joke task
    in
        cmd
```

The one on the left gets the desired property, while the one on the right simply gets the JSON as a string.

1. Make an Http request at the given url.
2. Look for the nested property value > joke.
3. Attempt to decode the joke property as a string.

```
randomJoke : Cmd Msg
randomJoke = Task.perform Fail Joke (Http.get (at [ "value", "joke" ] string) "http://api.icndb.com/jokes/random")
```

Decoding Multiple Properties

What if we wanted to decode the joke id, joke content and category?

To decode the Response into a more advanced Type, we need to **create a new decoder for the Type**.

Let's say we want to decode the response into a record. First, we need to create an alias for the record we want to decode the response into.

Then we need to decode it with one of two ways because the at function is not enough by itself. The first one is by using objectX depending on the number of properties (X). It is cumbersome and it doesn't handle properties that may not have a value.

```
object3
    :  (a -> b -> c -> value)
    -> Decoder a
    -> Decoder b
    -> Decoder c
    -> Decoder value
```

object3 takes 4 parameters.

1. First one is a function that takes 3 parameters and returns the type we want the decoder to create (Response).
2. The next 3 parameters are decoders for the 3 JSON properties we want decoded.

Ex. Decoder a or ("id" := int) means "Find a property named id which should be an integer".

```
import Json.Decode exposing (..)

type alias Response =
    { id : Int
    , joke : String
    , categories : List String
    }

responseDecoder : Decoder Response
responseDecoder =
    object3 Response
        ("id" := int)
        ("joke" := string)
        ("categories" := list string)
        |> at ["value"]

randomJoke : Cmd Msg
randomJoke =
    let
        url = "http://api.icndb.com/jokes/random"
        task = Http.get responseDecoder url
        cmd = Task.perform Fail Joke task
    in
        cmd
```

After the decoder decodes each of the properties represented by a, b and c, the decoded values are passed into the function we provided as the first parameter i.e. the Response record. Whatever type that function returns, is the decoded value.

(:=) is an infix function which takes two parameters. A string to identify the specific property to decode on the left, and a decoder for that property's type on the right.

In the end, we need to make the overall decoder use the nested property (they are all under "value") by passing it as the second parameter in the **at** function. "If you look at the value property, you should find an object with these 3 properties".

The last thing needed is to change the message type from Joke String into Joke Response. This way we can access the values in the records, just like accessing an object in Javascript.

```
type Msg = Joke Response | Fail Http.Error | NewJoke

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Joke response -> ( (toString response.id) ++ " - " ++ response.joke, Cmd.none )
        Fail error -> ( ( toString error ), Cmd.none )
        NewJoke -> ("Fetching new joke...", randomJoke)
```

Better Way to Decode Multiple Properties

```elm
import Json.Decode exposing (..)
import Json.Decode.Pipeline exposing (decode, required, optional)

type alias Response =
    { id : Int
    , joke : String
    , categories : List String
    }

responseDecoder : Decoder Response
responseDecoder =
    decode Response
        |> required "id" int
        |> required "joke" string
        |> optional "categories" (list string) []
        |> at ["value"]

randomJoke : Cmd Msg
randomJoke =
    let
        url = "http://api.icndb.com/jokes/random"
        task = Http.get responseDecoder url
        cmd = Task.perform Fail Joke task
    in
        cmd
```

This one is useful if the shape of the response is unpredictable, meaning if properties can be missing. Also, it is useful if we want to hardcode a value in the decoded record.

For this we use the community package from NoRedInk by installing it with **elm package install NoRedInk/elm-decode-pipeline**.

We pipe the return value of the decoder into the required function which takes the property name (“id”) and a decoder for the property (int decoder).

The optional functions need an additional parameter for the default value when properties are not available.

Aside from that, everything is the same as using object3 i.e. using the at function for the location of the properties.

Navigation

The problem with the previous navigation is that the URL is unchanged regardless of the displayed page.

Withour URL updates, we can't:
- Bookmark a page.
- Share a URL.
- Refreshing takes you to a default page, instead of the same page.

For this we use the Navigation package which allows us to do things with the URL.

It's helpful to think of the browser's location input box as just a form input field that might be placed on the page. **From the app's perspective, it's just a text input that the app needs access to**. The app wants to know about changes and also make changes.

How will the app know when something changes? How can the app make changes?

Before URL Navigation

```elm
module Main exposing (..)

import Html exposing (..)
import Html.Events exposing (..)
import Html.App as App

type Page
    = LeaderBoard
    | AddRunner
    | Login
    | NotFound

type alias Model =
    { page : Page }

initModel : Model
initModel =
    { page = LeaderBoard }

init : (Model, Cmd Msg)
init = (initModel, Cmd.none)
```

```elm
type Msg
    = Navigate Page

update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
    case msg of
        Navigate page ->
            ( { model | page = page }, Cmd.none )
```

```elm
menu : Model -> Html Msg
menu model =
    header []
        [ a [ onClick (Navigate LeaderBoard) ]
            [ text "Leaderboard" ]
        , text " | "
        , a [ onClick (Navigate AddRunner) ]
            [ text "Add Runner" ]
        , text " | "
        , a [ onClick (Navigate Login) ]
            [ text "Login" ]
        ]

viewPage : String -> Html Msg
viewPage pageDescription =
    div []
        [ h3 [] [ text pageDescription ]
        , p [] [ text <| "TODO: make " ++ pageDescription ]
        ]
```

```elm
view : Model -> Html Msg
view model =
    let
        page =
            case model.page of
                LeaderBoard -> viewPage "Leaderboard Page"
                AddRunner -> viewPage "Add Runner Page"
                Login -> viewPage "Login Page"
                NotFound -> viewPage "Page Not Found"
    in
        div []
            [ menu model
            , hr [] []
            , page
            ]
```

```elm
subscriptions : Model -> Sub Msg
subscriptions model = Sub.none

main : Program Never
main =
    App.program
        { init = init
        , update = update
        , view = view
        , subscriptions = subscriptions
        }
```

Navigaton After

The Navigation package has a program function which can be used instead of the Html.App program one.

```elm
program
    :  Parser data
    -> { init : data -> (model, Cmd msg), update : msg -> model -> (model, Cmd msg), urlUpdate : data -> model -> (model, Cmd msg)
    -> Program Never
```

The new program function provides what is needed to deal with URL changes. How is it different from Html.App?

The first thing is the first parameter Parser of data. The parser takes the raw location information from the browser and transforms it into a useful data type of our choosing.

The second difference is the type of the init field. Instead of being a constant, it is now a function that takes the value returned by the parser as a single parameter. The reason init is now a function is that when our page first loads, our app needs to know what the starting URL is so it can set the initialModel page value. In other words, it navigates our app to the correct starting page.

The third difference is the urlUpdate field which needs to be set to a function whose first parameter is a type variable named data i.e. the return from the parser. The second parameter is the model. The return value is a touple of model and command.

urlUpdate gets called everytime the URL changes

To start, we need to install the navigation package with **elm package install elm-lang/navigation**.

After that, we set the program to Navigation.program instead of App.Program and we create a navigation parser.

To create the navigation parser, we need to use package's makeParser function, passing it a function (locationParser) that takes a location record and returns a value represented by the type variable a. (whatever type we want) We decide exactly what this function should return.

The value returned from locationParser is passed along to the init function when the page first loads, and the urlUpdate function everytime the URL changes.

```
makeParser : (Location -> a) -> Parser a
```

```
type alias Location =
    { href : String
    , host : String
    , hostname : String
    , protocol : String
    , origin : String
    , port_ : String
    , pathname : String
    , search : String
    , hash : String
    , username : String
    , password : String
    }
```

```
locationParser : Location -> Page
locationParser location =
    case location.hash of
        "#" -> LeaderBoard
        "#add" -> AddRunner
        "#login" -> Login
        _ -> NotFound

main : Program Never
main =
    Navigation.program (makeParser locationParser)
        { init = init
        , update = update
        , view = view
        , subscriptions = subscriptions
        , urlUpdate = urlUpdate
        }
```

The hash in our app is the part of the URL after the # (pound) sign.

At this point the locationParser has transformed the URL change into a Page value.

```elm
initModel : Model
initModel =
    { page = LeaderBoard }

init : (Model, Cmd Msg)
init = (initModel, Cmd.none)
```

```elm
initModel : Page -> Model
initModel page =
    { page = page }

init : Page -> (Model, Cmd Msg)
init = (initModel page, Cmd.none)
```

```elm
type Msg
    = Navigate Page

update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
    case msg of
        Navigate page ->
            ( { model | page = page }, Cmd.none )
```

```elm
toHash : Page -> String
toHash page =
    case page of
        LeaderBoard -> "#"
        AddRunner -> "#add"
        Login -> "#login"
        NotFound -> "#notfound"


update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
    case msg of
        Navigate page ->
            ( model , newUrl (toHash page) )

urlUpdate : Page -> Model -> (Model, Cmd Msg)
urlUpdate page model =
    ( { model | page = page }, Cmd.none )
```

```elm
main : Program Never
main =
    App.program
        { init = init
        , update = update
        , view = view
        , subscriptions = subscriptions
        }
```

```elm
locationParser : Location -> Page
locationParser location =
    case location.hash of
        "#" -> LeaderBoard
        "#add" -> AddRunner
        "#login" -> Login
        _ -> NotFound

main : Program Never
main =
    Navigation.program (makeParser locationParser)
        { init = init
        , update = update
        , view = view
        , subscriptions = subscriptions
        , urlUpdate = urlUpdate
        }
```

Websockets

Ports

Useful when Javascript libraries are needed that don’t exist in Elm, or when migrating from other frameworks like React and Angular.

Modifying Lists

There are two ways to add items in a list We can use **append** or **cons**. Appends adds the item at the end of the list, whilst cons adds it at the beginning. Cons is much more efficient as it doesn't have to traverse the whole list.

```
-- append ++
newPlayers = players ++ [player]
-- cons ::
newPlayers = player :: players
```

We can think of lists (arrays) as a series of linked items i.e. a series of cons operators.

```
-- linked
myList = [el1, el2, el3]
-- Equivalent Syntax
myList =  el1 :: el2 :: el3 :: []
```

Append

It has to traverse the whole list in order to find the end and insert a cons operation before the empty array. This is rather inneficient as it it still a cons operation.

```
-- Append
newList =  myList ++ [el4]
-- Append
newList =  el1 :: el2 :: el3 :: el4 :: []
```

Cons

This is substantially faster as it just adds the new item at the beginning of the list. Even better, the first item can be accessed with the head operator, which is similira to array.length – 1, but much more efficient as it doesn't have to evaluate the size of the array.

```
-- Cons
newList = el4 :: myList
-- Cons
newList = [el4, el1, el2, el3]
```

Map in Elm

List.map takes a functions as the first parameter and a list as the second. It outputs a new list containing the modified values.

```
map : (a -> b) -> List a -> List b

newList = List.map times2 [ 1, 2, 3 ]

times2 num = num * 2

newList = [ 2, 4, 6 ]
```

Map in Javascript

Map returns an array, with the results from the function called on each item from the original array.

map.js

```
var array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

console.log(array);

var squared = array.map(function(item){
    return item * item;
});

console.log(squared);
```

```
Ivan@IvanV MINGW64 /d/dev/map
$ node map.js
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
[ 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 ]
```

It can also be done like this: **var squared = array.map(Math.sqrt);**

Webpack

LearnCode.academy Youtube

https://www.youtube.com/watch?v=9kJVYpOqcVU

It is the module loader/build system of choice for React developers.

What it does is the same as **Browserify** (putting everything in one file and compiling it to be used by a browser) and **Gulp/Grunt** (minifying code etc.). So instead of using combinations, we can just use **Webpack**.

```
untitled  scripts.js  module1.js  module2.js  scripts.min.js  index.html  webpack.config.js
1  var debug = process.env.NODE_ENV !== "production";
2  var webpack = require('webpack');
3
4  module.exports = {
5    context: __dirname,
6    devtool: debug ? "inline-sourcemap" : null,
7    entry: "./js/scripts.js",
8    output: {
9      path: __dirname + "/js",
10     filename: "scripts.min.js"
11   },
12   plugins: debug ? [] : [
13     new webpack.optimize.DedupePlugin(),
14     new webpack.optimize.OccurenceOrderPlugin(),
15     new webpack.optimize.UglifyJsPlugin({ mangle: false, sourcemap: false }),
16   ],
17 };
```

Explanations

webpack.config.js

var debug: Is our Node environment in production? If yes, we will run all the minification plugins, and not have a devtool i.e. null. If it's in debug mode there will be no plugins used and we will run source mapping (helps with console logging).
context: __dirname = The directory we are currently in. If we had an app or src directory, we would do __dirname+"/app".
entry: This is where all the dependencies end up (require).
output: The file from entry, after it gathers all the dependencies in one file, it will be saved in the path directory under the filename specified, after it's processed by all the plugins in production mode.

```
(v5.4.0) ~/code/sandbox/module-loaders
o → webpack
Hash: abc9360b94b79352af52
Version: webpack 1.12.10
Time: 64ms
        Asset     Size  Chunks             Chunk Names
scripts.min.js  4.54 kB       0  [emitted]  main
   [0] ./js/scripts.js 50 bytes {0} [built]
   [1] ./js/module1.js 42 bytes {0} [built]
   [2] ./js/module2.js 42 bytes {0} [built]

(v5.4.0) ~/code/sandbox/module-loaders
o →
```

```
(v5.4.0) ~/code/sandbox/module-loaders
o → NODE_ENV=production webpack
Hash: abc9360b94b79352af52
Version: webpack 1.12.10
Time: 95ms
        Asset     Size  Chunks             Chunk Names
scripts.min.js  677 bytes       0  [emitted]  main
   [0] ./js/scripts.js 50 bytes {0} [built]
   [1] ./js/module1.js 42 bytes {0} [built]
   [2] ./js/module2.js 42 bytes {0} [built]

(v5.4.0) ~/code/sandbox/module-loaders
o →
```

The first one runs webpack in **debug mode** and returns a source mapped file **scripts.min.js (4,5KB)**.The second command runs it **in production mode** with **NODE_ENV=production webpack** and returns a minified file **scripts.min.js (677 bytes)**.

Another cool thing that webpack does is running localized environments. Ex. using jQuery in ONLY one module and nowhere else i.e. jQuery code will not work outside the module.

Webpack Basics - Lynda

Linux

Commands

man *command* – Explains the command.
apt-cache search *name* – Searches the package manager for a specific program.
sudo apt-get install *name* – Installs the specified package.
sudo dpkg –i *filename.deb* – Installs a downloaded .deb file.

dpkg is a backend for **apt-get** which is a backend for **aptitude** (GUI).

GIT

https://www.youtube.com/watch?v=0fKg7e37bQE

For it to work in Windows, you need to install it first. Get it from here: https://git-scm.com/download/win

Most Common Windows / Unix Commands

Action	Windows	Unix
Navigate outside	cd..	cd ..
Navigate to	cd folder-name	cd folder-name
List directory content	dir	ls
Show path		pwd
Change partition	D:	/d
Clear screen		clear

Most Common GIT Commands

Action	GIT Commands
Show all the commands i.e. help	git
Copy (pull) repository in current folder from Github	git clone URL
Show differences between local folder and Github	git status
Add a file to be tracked	git add FILENAME
Add everything to be tracked	git add . / git add -A
Save changes locally with a message	git commit –m “The message”
Upload (push) the changes to Github	git push
Download the latest changes	git pull
Exit after committing without using -m	ESC + :wq

Omitting –m while committing opens up a multiline message which can be exited by ESC + :wq

git config user.name NAME
git config user.email EMAIL

Path to github repositories: /d/work/acme/github

Common Workflow

Updating

1. Navigate to directory.
2. git status
3. git add . OR git add FILENAME
4. git commit –m “message”
5. git push

Uploading existing project first time

https://help.github.com/articles/adding-an-existing-project-to-github-using-the-command-line/

1. Create repository on github without README.md to avoid errors.
2. Navigate to the local project.
3. git init
4. git add .
5. git commit –m “First commit”
6. git remote add origin REPOSITORY_URL
7. git push origin master

SSH

SSH, useful for creating secure tunnels over unsecured networks allowing secure networking. Like a VPN, just more hands on and flexible.

It’s a better way of putting files on a server over FTP.

Sublime IDE - Text Editor

Installing Themes

https://scotch.io/bar-talk/best-sublime-text-3-themes-of-2015-and-2016

To install themes, just use package control. So the process would be:

1. Install **Package Control** by running a script from the website inside the Sublime console.
2. CTRL + SHIFT + P
3. Look for **Package Control: Install Package** and click it.
4. Search for the theme inside and hit enter.
5. Set the theme in **Preferences -> Settings – User** by editing the JSON property called theme