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: <div id="box"></div>

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.

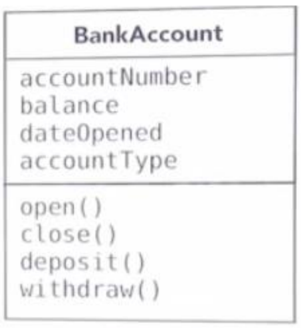
One common formal requirements approach is FURPS / FURPS+. This is a checklist, not instructions. Focus on MUST HAVE, instead of NICE TO HAVE.
- Functional. App features.
- Usability. Help, documentation, tutorials...
- Reliability. Disaster recovery, acceptable fail rate...
- Performance. Availability,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

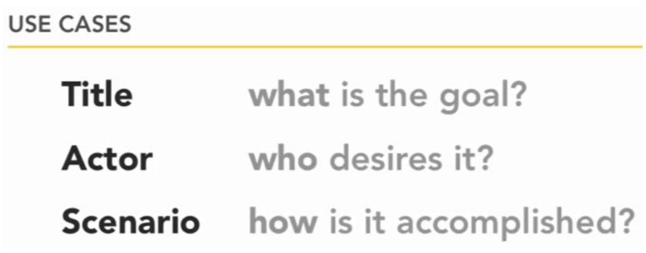
Class Diagram



Use Cases

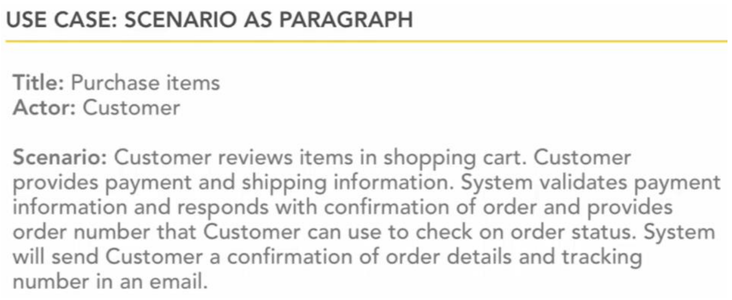
Here we focus on the user instead of the features of the program. How does the user accomplish something?

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



USE CASES

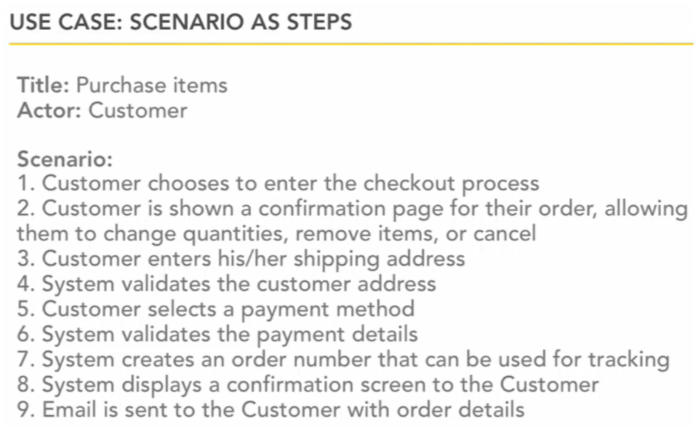
Title	what is the goal?
Actor	who desires it?
Scenario	how is it accomplished?



USE CASE: SCENARIO AS PARAGRAPH

Title: Purchase items
Actor: Customer

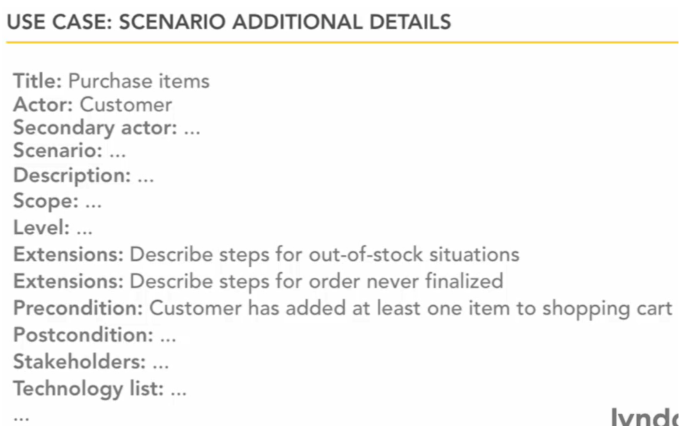
Scenario: Customer reviews items in shopping cart. Customer provides payment and shipping information. System validates payment information and responds with confirmation of order and provides order number that Customer can use to check on order status. System will send Customer a confirmation of order details and tracking number in an email.



USE CASE: SCENARIO AS STEPS

Title: Purchase items
Actor: Customer

Scenario:
1. Customer chooses to enter the checkout process
2. Customer is shown a confirmation page for their order, allowing them to change quantities, remove items, or cancel
3. Customer enters his/her shipping address
4. System validates the customer address
5. Customer selects a payment method
6. System validates the payment details
7. System creates an order number that can be used for tracking
8. System displays a confirmation screen to the Customer
9. Email is sent to the Customer with order details



USE CASE: SCENARIO ADDITIONAL DETAILS

Title: Purchase items
Actor: Customer
Secondary actor: ...
Scenario: ...
Description: ...
Scope: ...
Level: ...
Extensions: Describe steps for out-of-stock situations
Extensions: Describe steps for order never finalized
Precondition: Customer has added at least one item to shopping cart
Postcondition: ...
Stakeholders: ...
Technology list: ...
...

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.

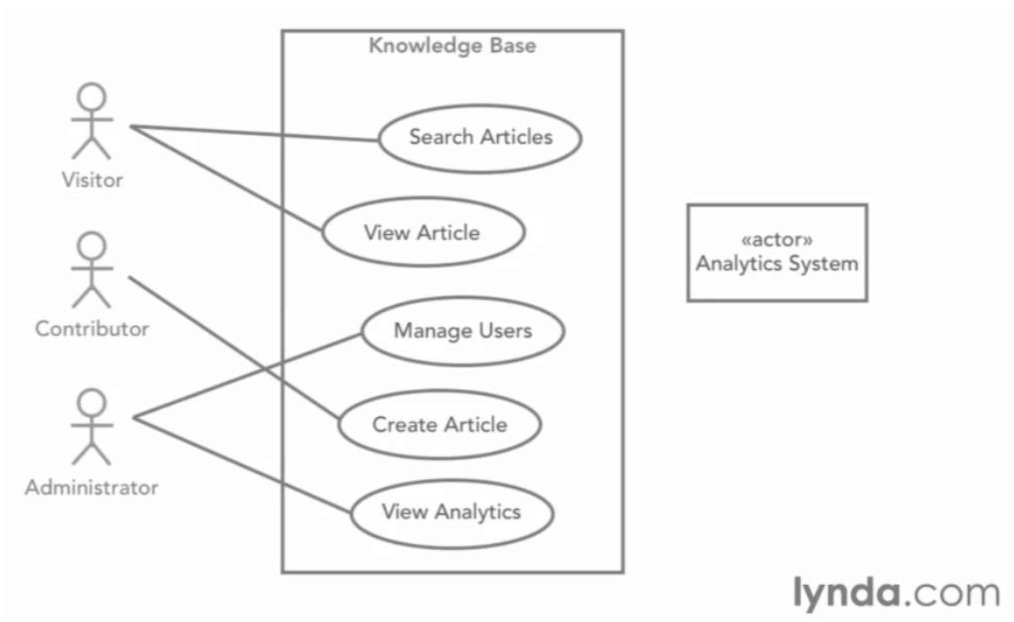
FOCUS ON INTENTION!

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?

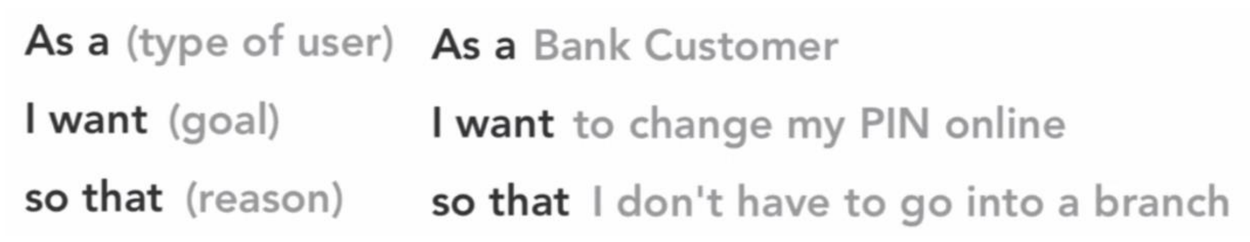
Use Case Diagram

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.



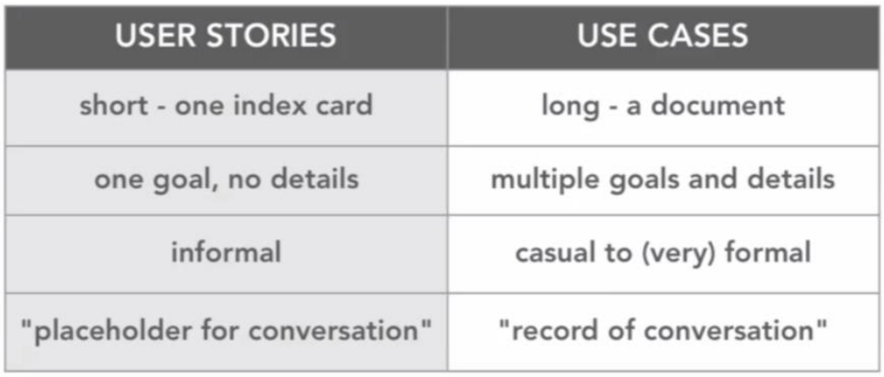
User Stories

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.



As a (type of user)	**As a** Bank Customer
I want (goal)	**I want** to change my PIN online
so that (reason)	**so that** I don't have to go into a branch

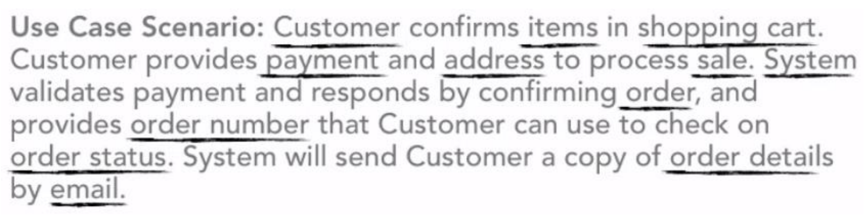
Differences between Use Case and User Story



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"

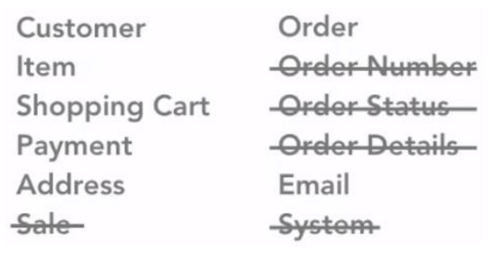
Domain Modelling

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



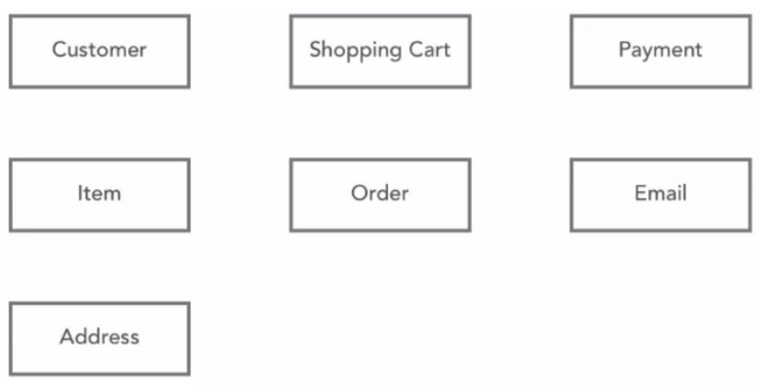
Use Case Scenario: Customer confirms items in shopping cart. Customer provides payment and address to process sale. System validates payment and responds by confirming order, and provides order number that Customer can use to check on order status. System will send Customer a copy of order details by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

Noun List

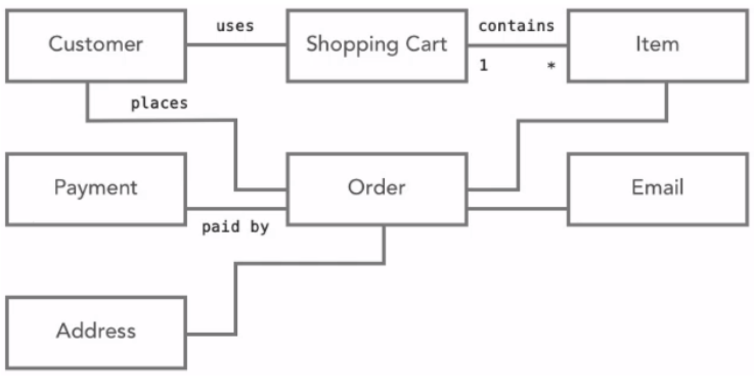


Customer	Order
Item	~~Order Number~~
Shopping Cart	~~Order Status~~
Payment	~~Order Details~~
Address	Email
~~Sale~~	~~System~~

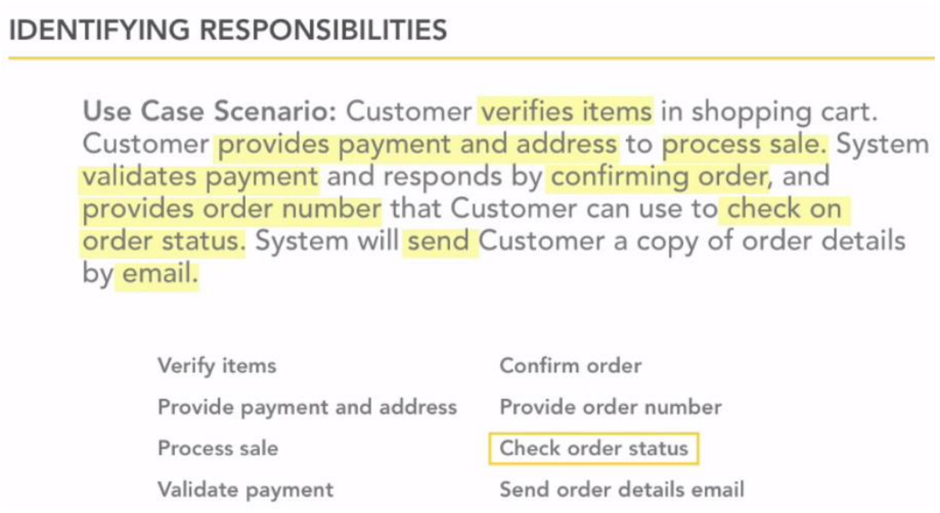
Conceptual Object Model



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



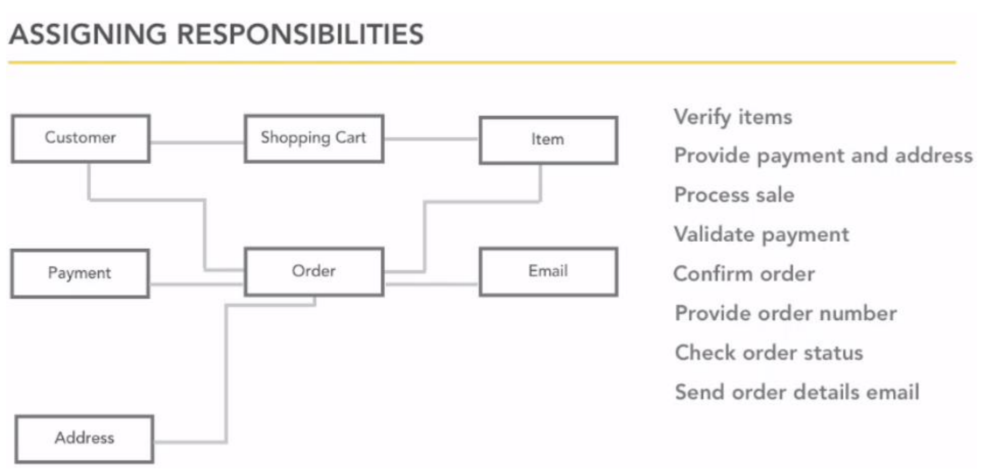
Look for the verbs. An object should be responsible for itself. Don't confuse the actors with this. They initiate the behavior that lives in the objects.

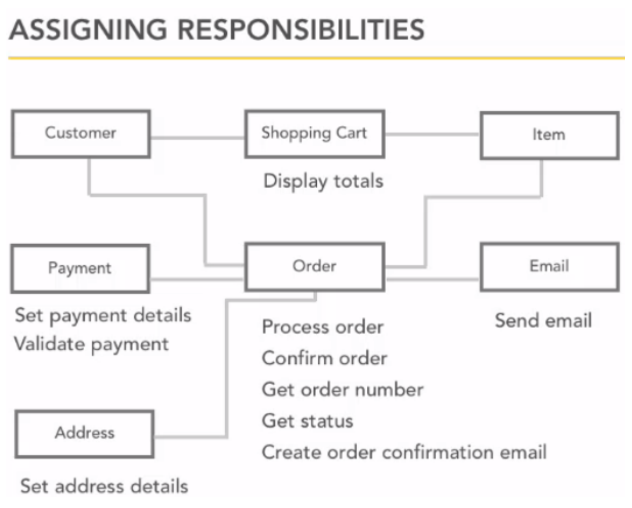


IDENTIFYING RESPONSIBILITIES

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.

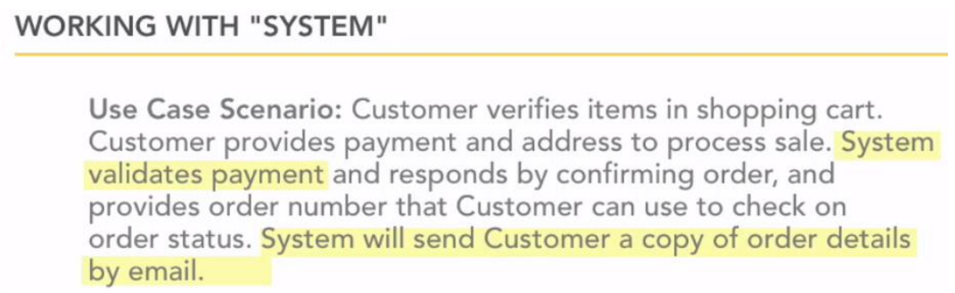
- Verify items
- Provide payment and address
- Process sale
- Validate payment
- Confirm order
- Provide order number
- Check order status
- Send order details email





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.



WORKING WITH "SYSTEM"

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.

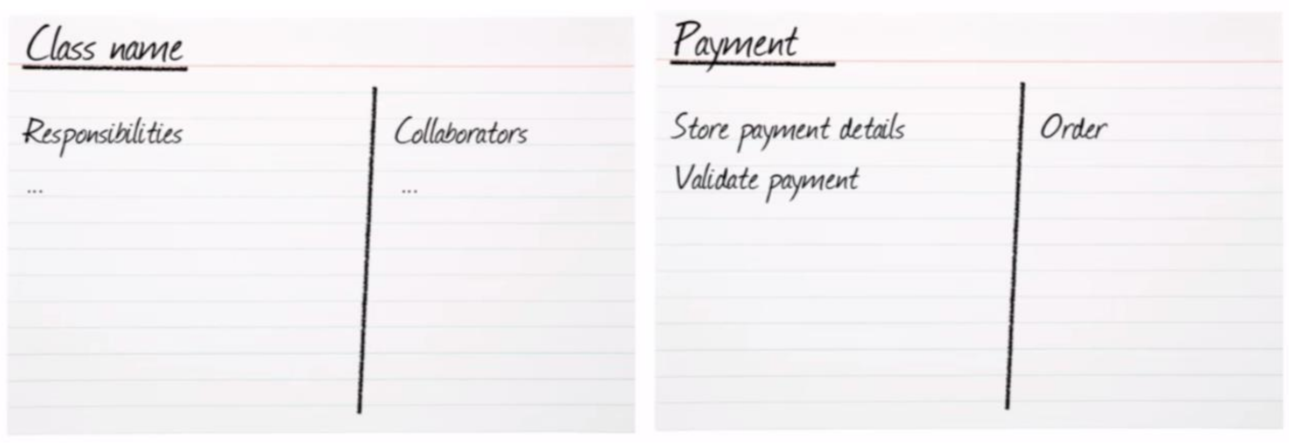
This can lead to people creating a "System" object and putting a lot of behavior in it. This is a big mistake. "System" is just a placeholder word for the specific object that has not yet been identified. It should be read as "SOME PART of the system does something…".

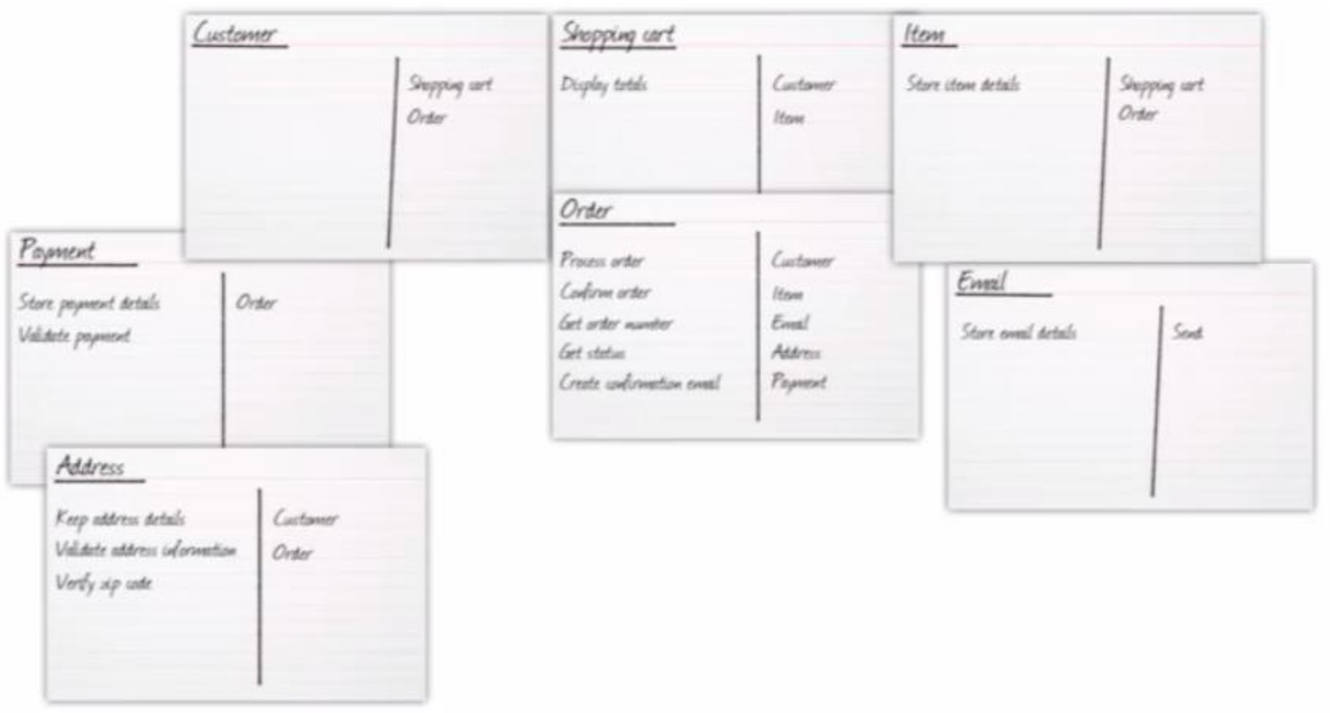
Responsibilities should be distributed between objects, not stored in one master object. This is a sign that you are thinking procedurally, not OOP.

CRC – Class Responsibility Collaboration

1 CRC card = 1 Class

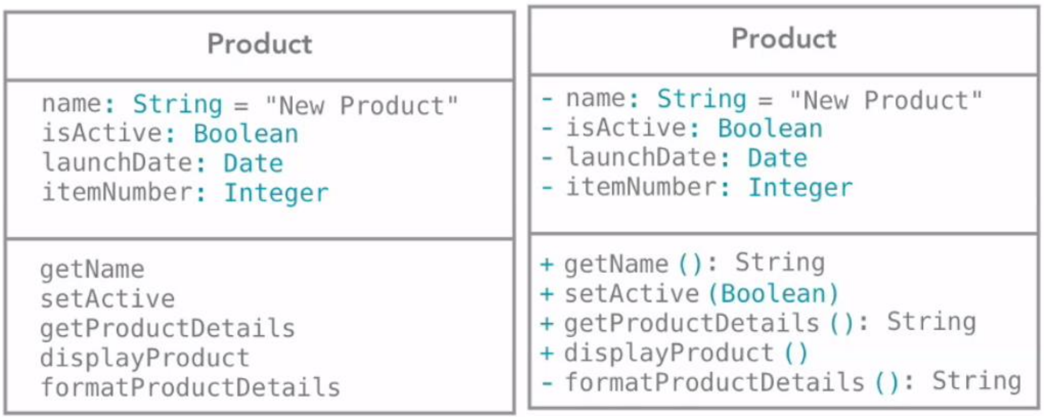
Avoid using software for this stage. There is a great advantage in the physicality of index cards.





Creating Classes

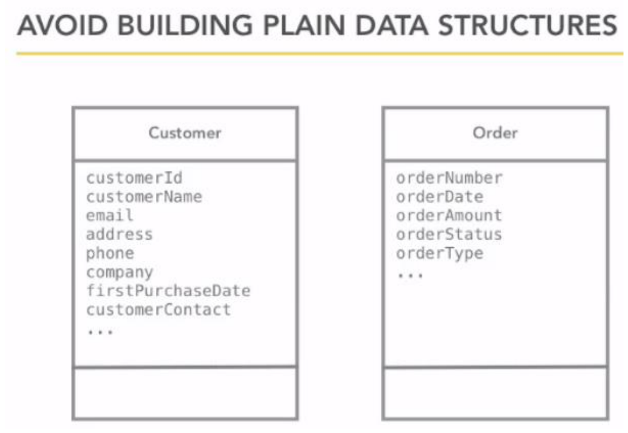
Classes are named in singular with uppercase first letter.



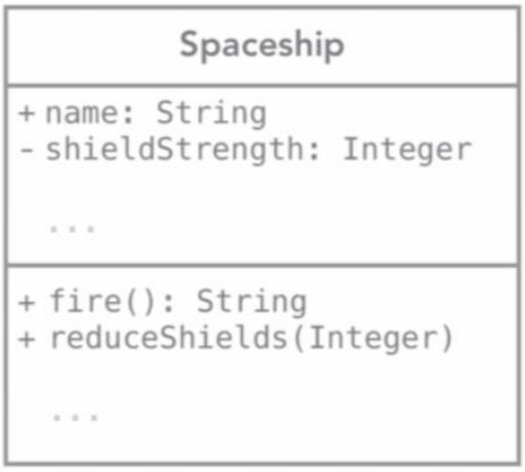
: Data Types, + = Public,- = Private

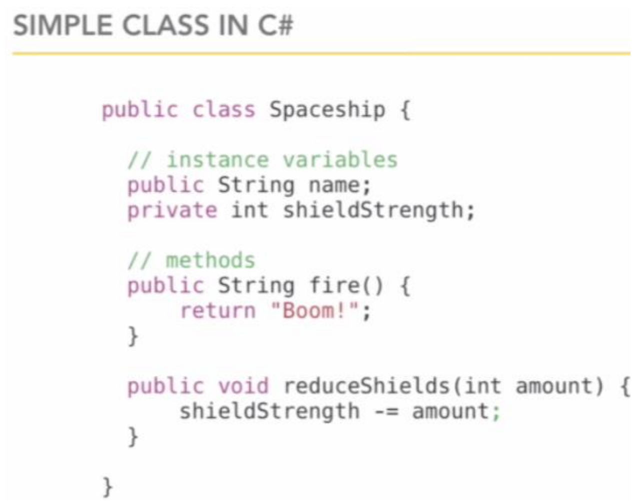
Everything should be as private as possible.

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.



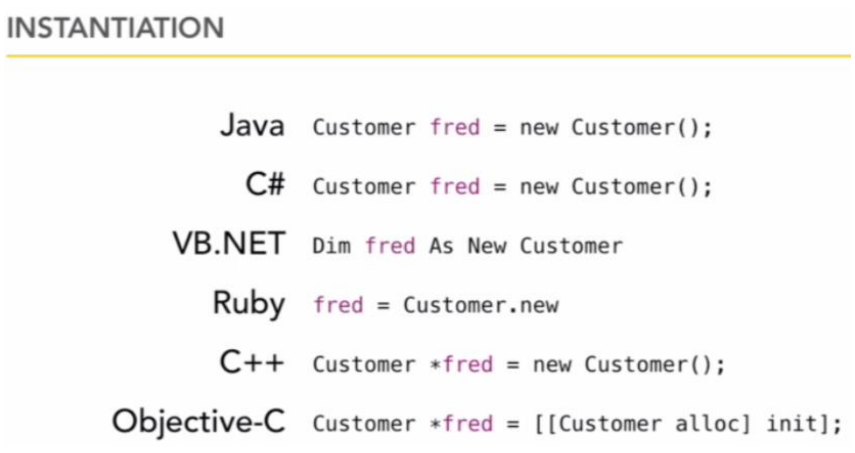
Converting Class Diagrams to Code





```
public class Spaceship {

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

   // methods
   public String fire() {
       return "Boom!";
   }

   public void reduceShields(int amount) {
       shieldStrength -= amount;
   }

}
```

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];`

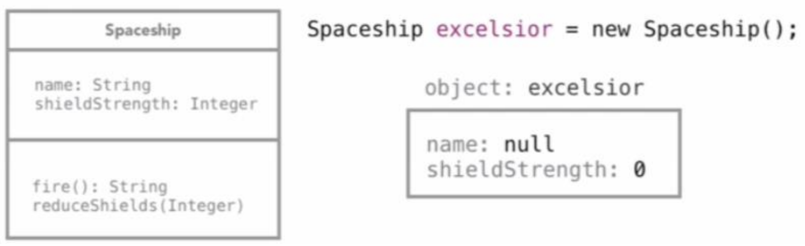
If you want something else to happen i.e. take part in the instantiation of an object, a constructor is used.

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.

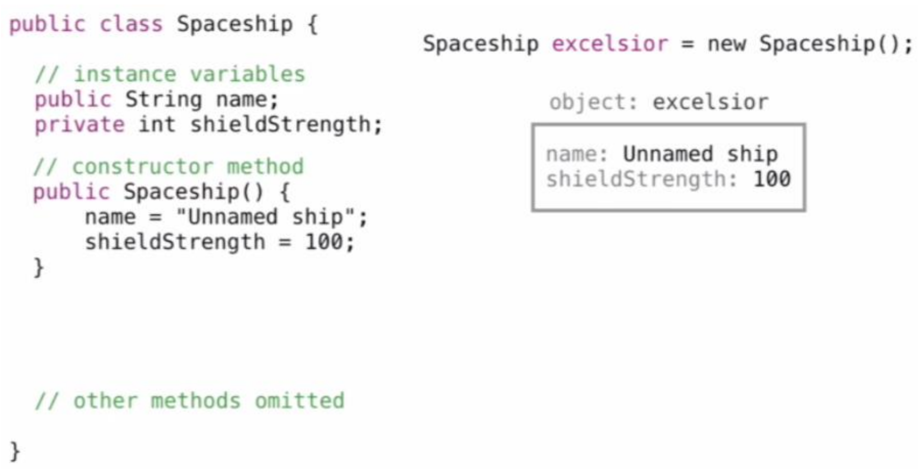
Constructors are an OOP concept and are not available in procedural languages.

No Constructor



```
Spaceship excelsior = new Spaceship();
```

With Constructor

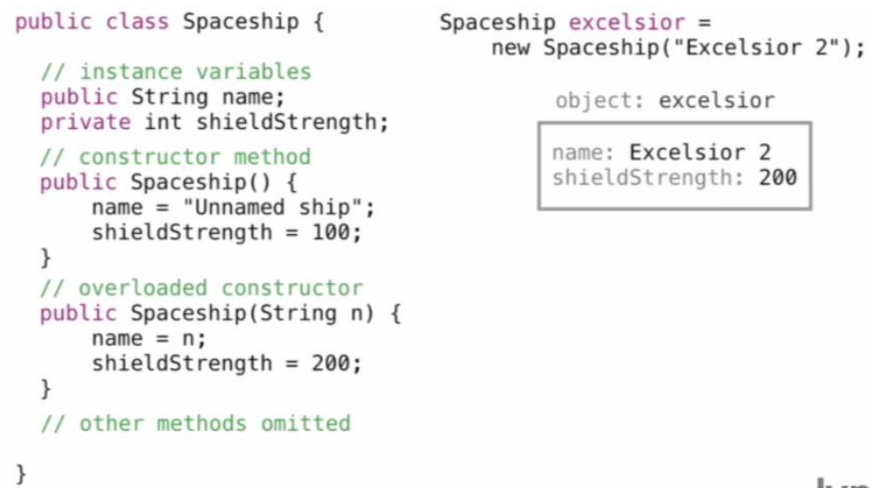


```
public class Spaceship {

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

  // constructor method
  public Spaceship() {
      name = "Unnamed ship";
      shieldStrength = 100;
  }



  // other methods omitted

}
```

```
Spaceship excelsior = new Spaceship();
```

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.

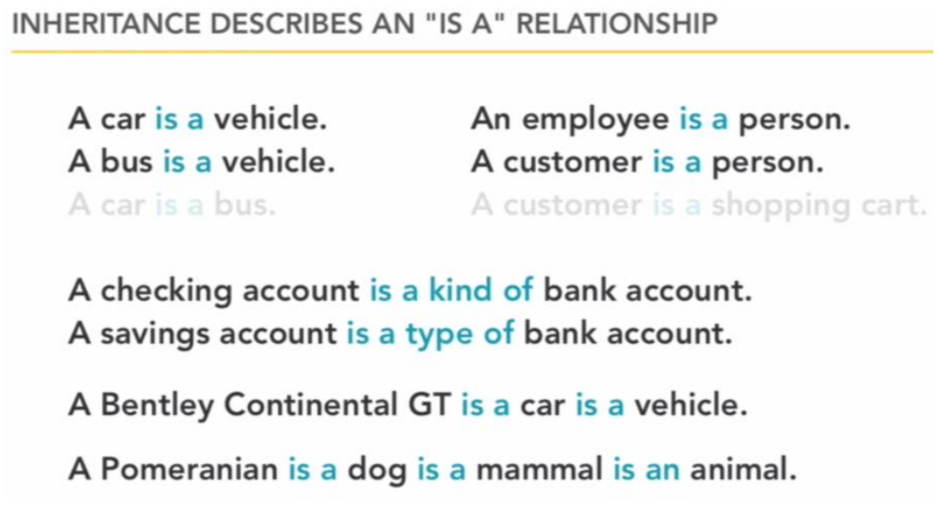
Variables are called with the objects name whereas the static ones are called with the class name.

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

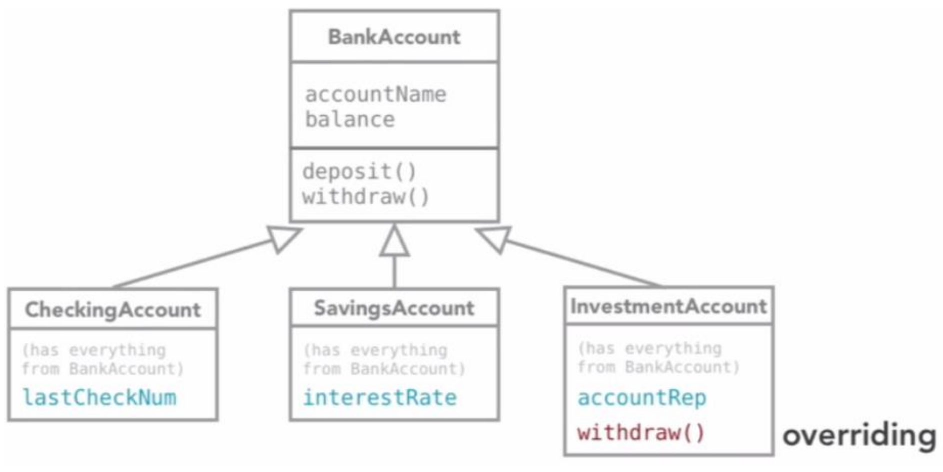
A car is a vehicle.
A bus is a vehicle.
A car is a bus.

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

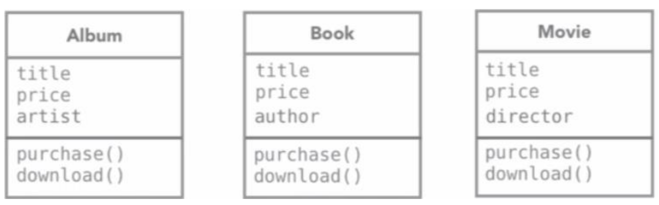
A checking account is a kind of bank account.
A savings account is a type of bank account.

A Bentley Continental GT is a car is a vehicle.

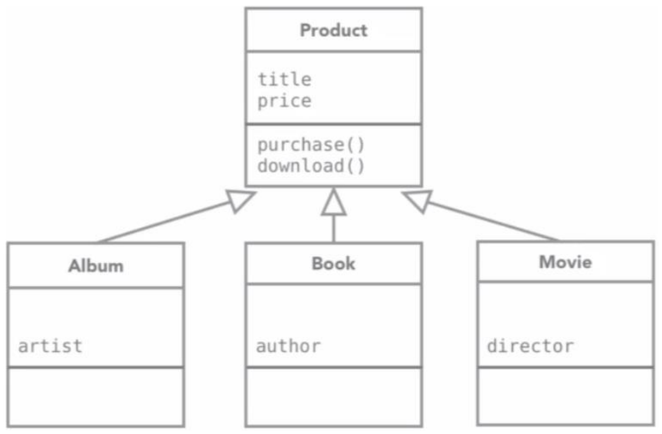
A Pomeranian is a dog is a mammal is an animal.



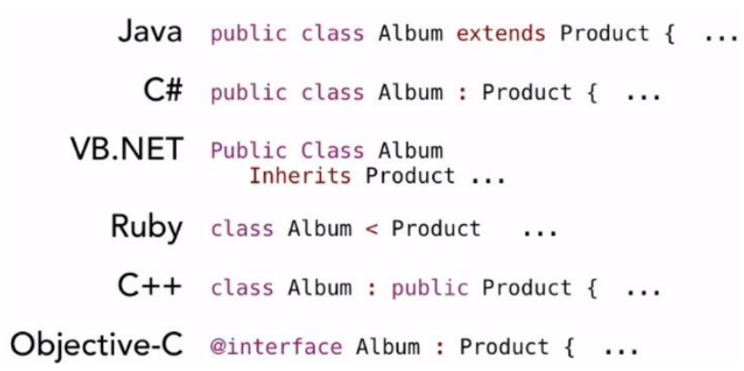
Don't go actively looking for inheritance. It should come naturally and be obvious. It is a mistake to go too deep with it. Be suspicious of levels beyond 1 or 2.



>



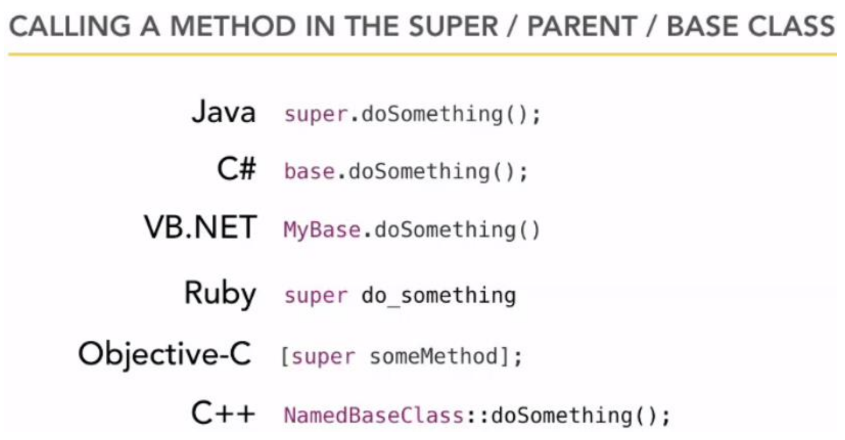
Using Inheritance



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

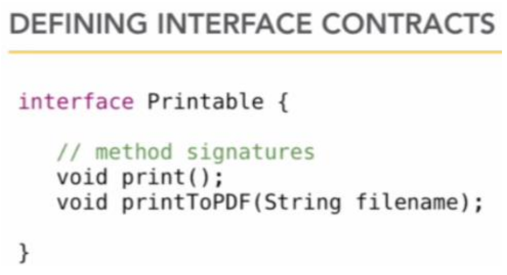
Java	`super.doSomething();`
C#	`base.doSomething();`
VB.NET	`MyBase.doSomething()`
Ruby	`super do_something`
Objective-C	`[super someMethod];`
C++	`NamedBaseClass::doSomething();`

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.

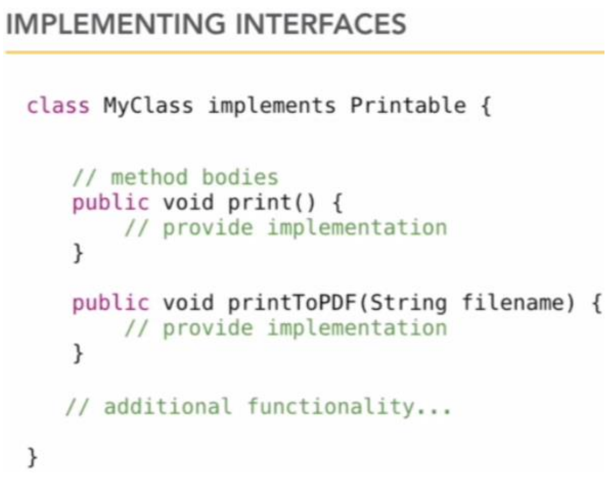
Interfaces

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

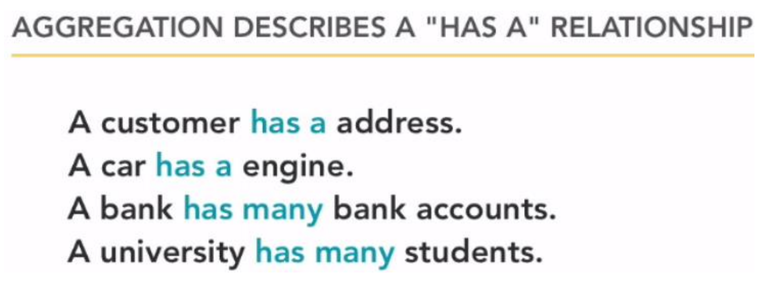
```
class MyClass implements Printable {

    // 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

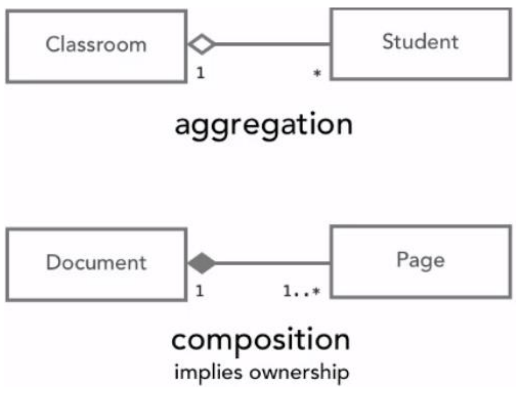
Aggregation and Composition



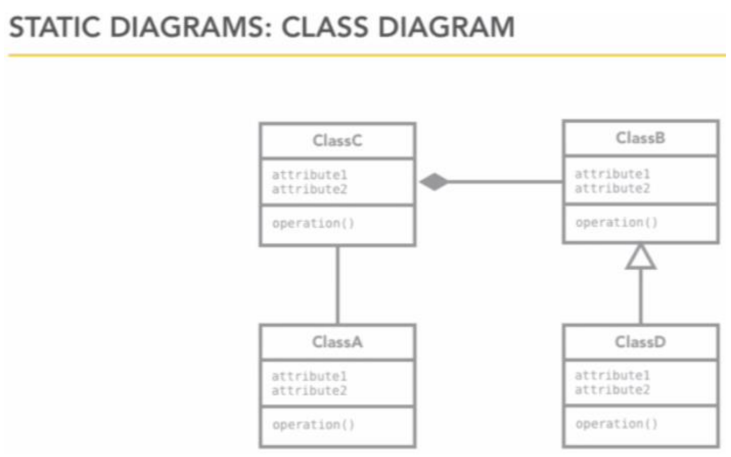
AGGREGATION DESCRIBES A "HAS A" RELATIONSHIP

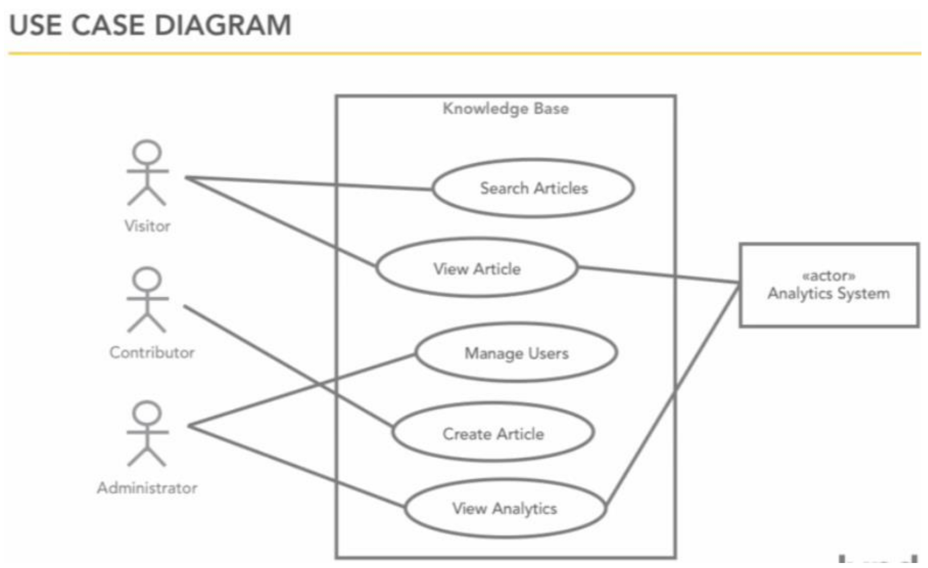
A customer has a address.
A car has a engine.
A bank has many bank accounts.
A university has many students.

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



Advanced Concepts

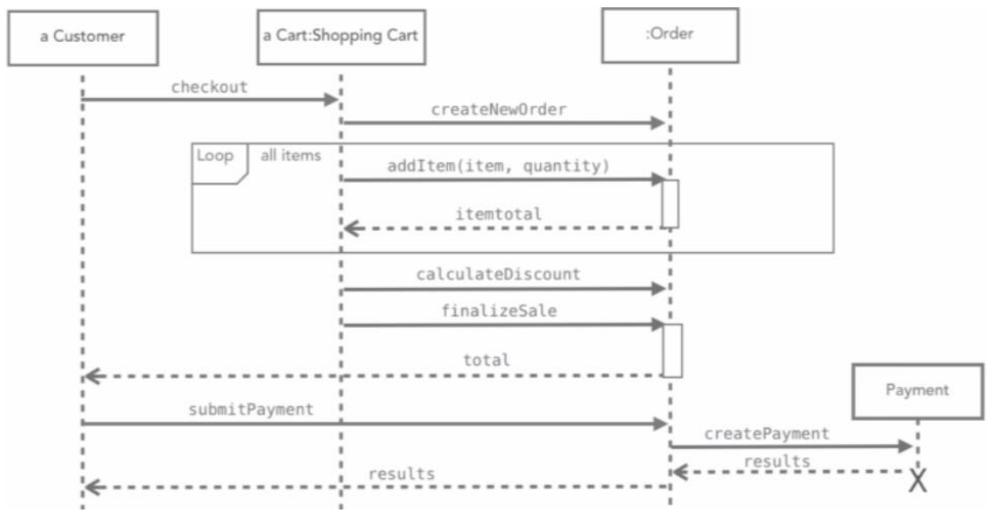




These are structural i.e. static diagrams. They are great for providing an overview of the classes with their compositions and actors. But, they are not good for representing the lifetime of objects. For this we use behavioral / dynamic diagrams. They describe how objects change and communicate, and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

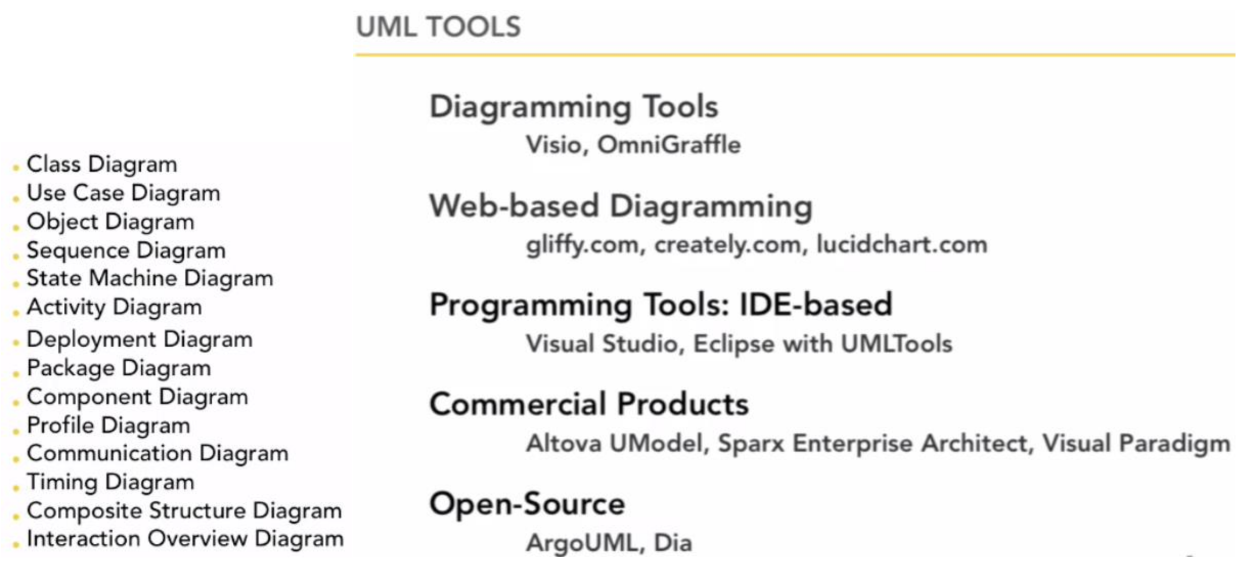
Sequence Diagrams

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.



UML TOOLS

- Class Diagram
- Use Case Diagram
- Object Diagram
- Sequence Diagram
- State Machine Diagram
- Activity Diagram
- Deployment Diagram
- Package Diagram
- Component Diagram
- Profile Diagram
- Communication Diagram
- Timing Diagram
- Composite Structure Diagram
- Interaction Overview Diagram

Diagramming Tools
Visio, OmniGraffle

Web-based Diagramming
gliffy.com, creately.com, lucidchart.com

Programming Tools: IDE-based
Visual Studio, Eclipse with UMLTools

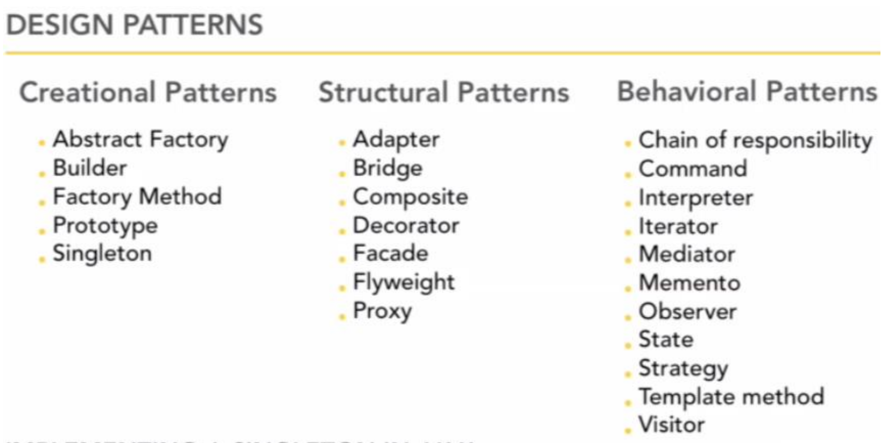
Commercial Products
Altova UModel, Sparx Enterprise Architect, Visual Paradigm

Open-Source
ArgoUML, Dia

Design Patterns

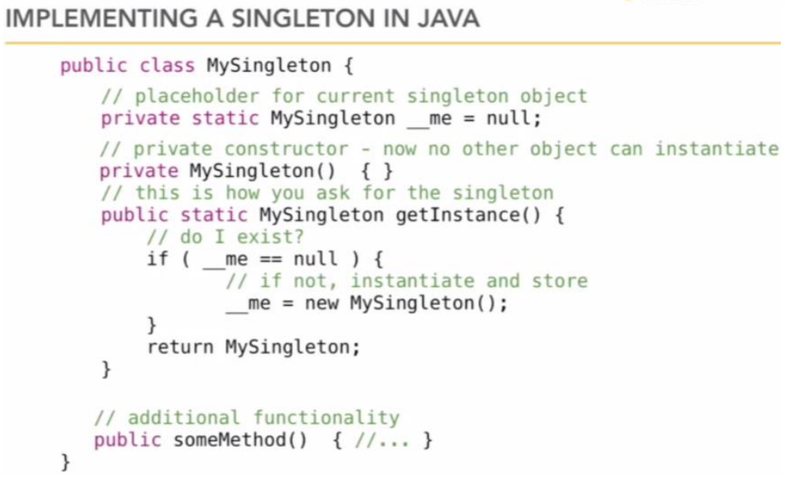
These are abstract ways to organize your programming. Think of design patterns as suggestions / best practices on how to organize classes and objects to accomplish a task. They are best practices for commong programming problems.

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”.



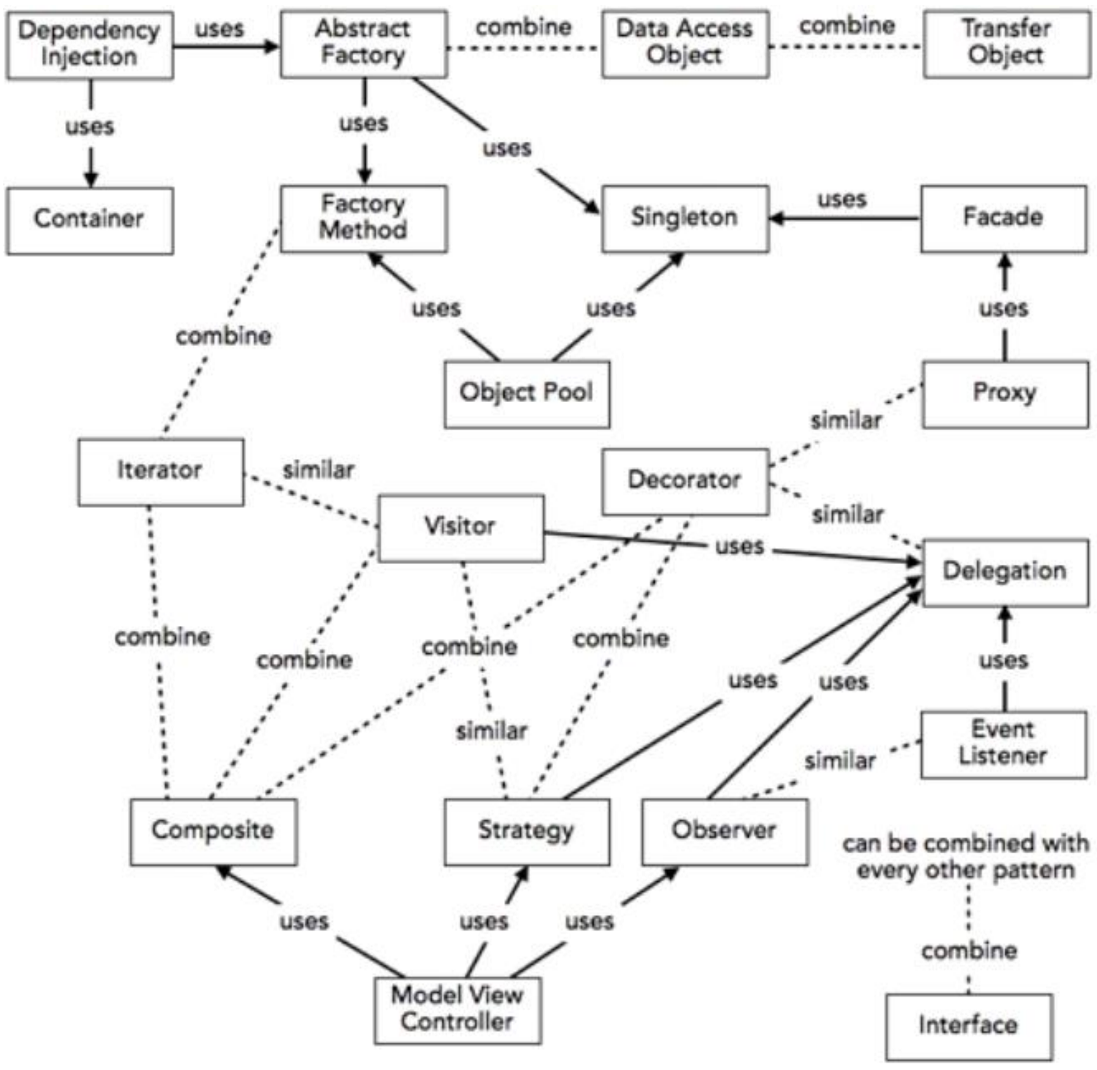
DESIGN PATTERNS

Creational Patterns	Structural Patterns	Behavioral Patterns
Abstract Factory	Adapter	Chain of responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template method
		Visitor



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
combine
uses
uses
uses
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.

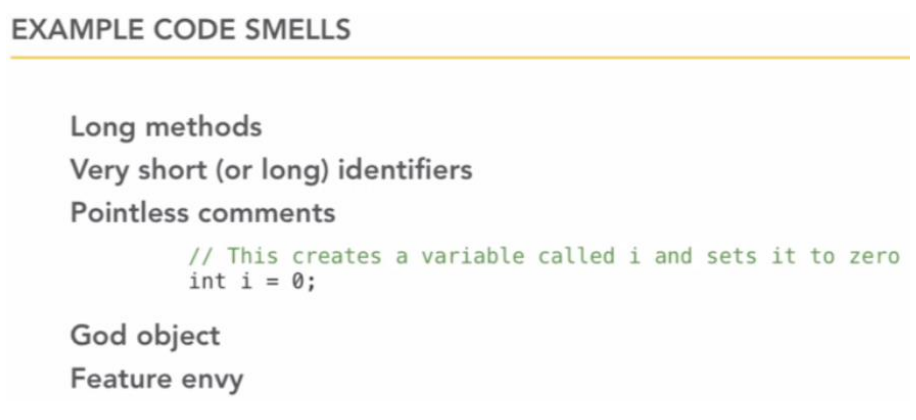
This creates a nightmare for debugging and adding new features. A single change can fracture the whole application.

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.



EXAMPLE CODE SMELLS

Long methods

Very short (or long) identifiers

Pointless comments

```
// This creates a variable called i and sets it to zero
int i = 0;
```

God object

Feature envy

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

SOLID Prinsiples

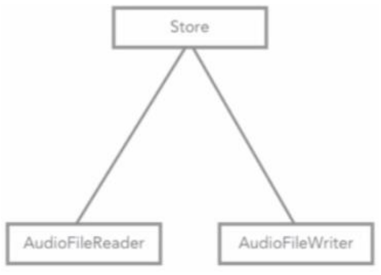
S – Single responsibility: An object should have one reason to exist, and that reason entirely encapsulated in one class. It can have many behaviors, but all of them should fall under that reason to exist. This is against the God Object.

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.

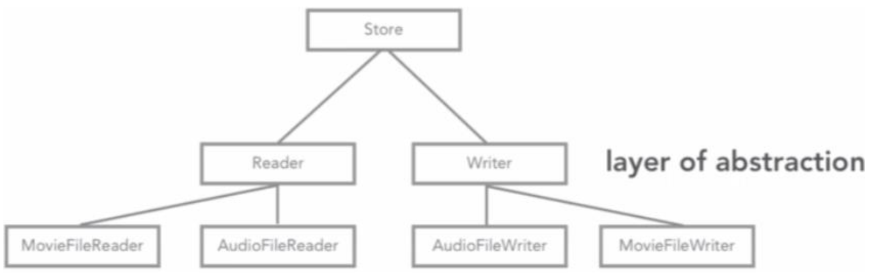
L – Liskov substitution: Derived classes must be substitutable for their base classes. Child classes should be able to be treated like their parent classes.

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.

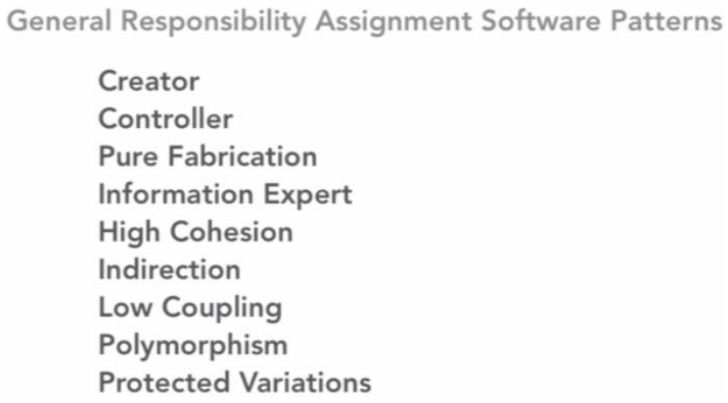
D – Dependency inversion: Depend on abstractions, not on concretions. Layers of abstraction should be added in order to make room for code extension, but be careful not to violate the YAGNI rule. Look at the picture below.



vs



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**.

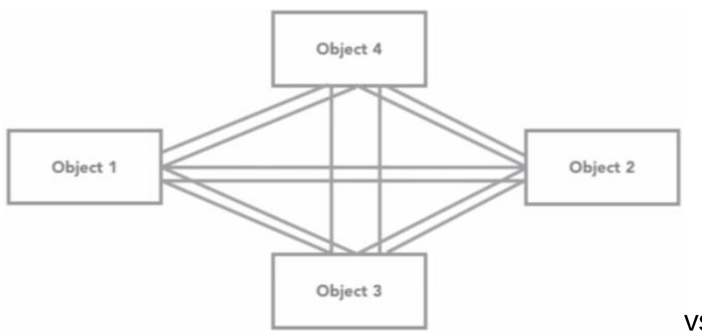
Pure Fabrication – What if something needs to exist in the application that doesn’t announce itself as an obvious class or real world object? What if a behavior does not fit any existing class? When the behavior does not belong anywhere else, create a new class. There is nothing wrong with making a class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

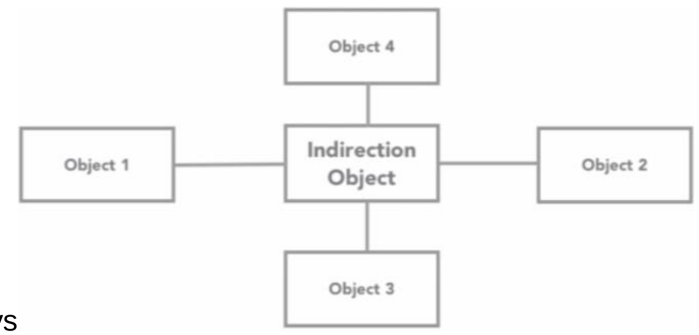
Low Coupling – (Coupling: The level of dependencies between objects) Reducing the amount of required connections between objects. As many as necessary, but as few as possible.

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.



vs

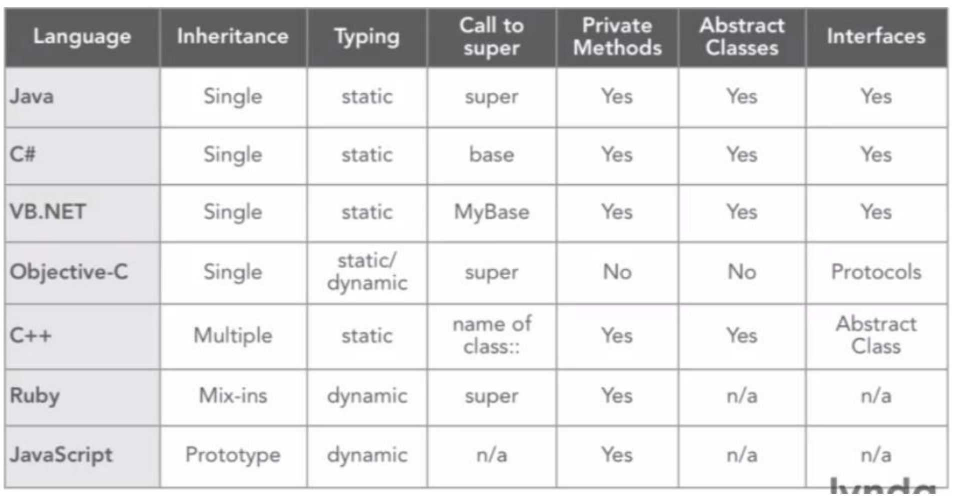


Polymorphism – Automatically correct behavior based on type. We don't want conditional logic that checks for particular types.

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.

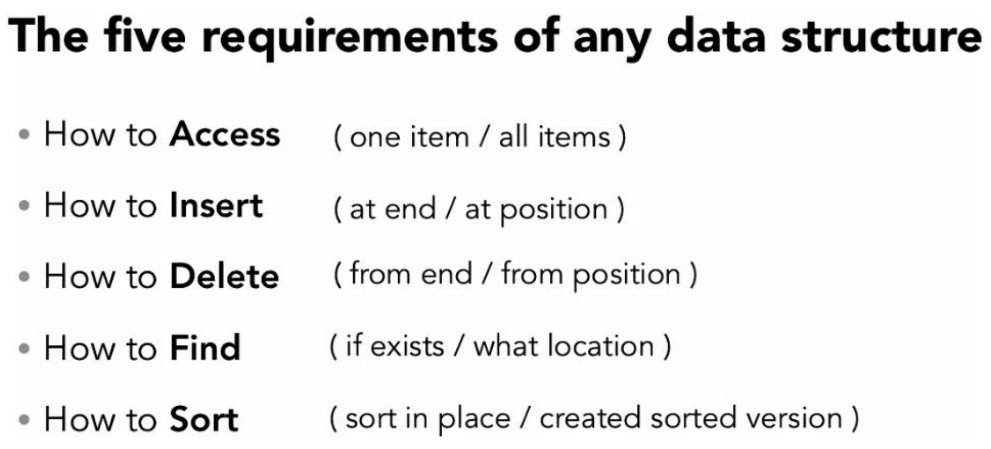
Build software, make mistakes. Don't focus on learning at the expense of practice. The more you learn the more you realize there is to learn. You'll never feel fully prepared for a project.

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

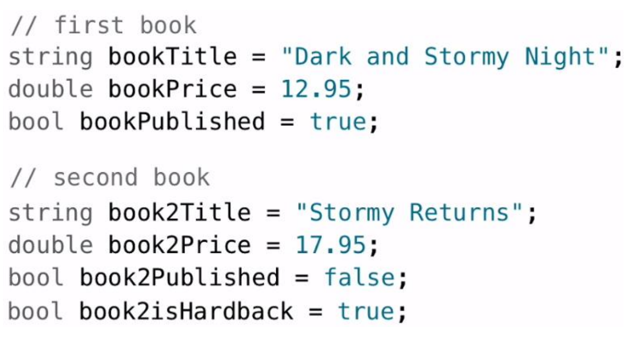
- How to **Access** (one item / all items)
- How to **Insert** (at end / at position)
- How to **Delete** (from end / from position)
- How to **Find** (if exists / what location)
- How to **Sort** (sort in place / created sorted version)

Simple Structures

A mathematical tuple is a grouped collection of elements.

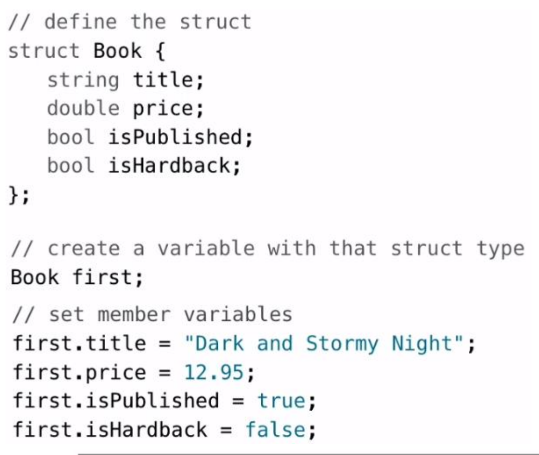
Struct = a very basic data structure.

Unorganized data vs a struct organization

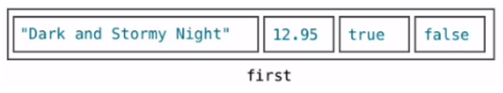


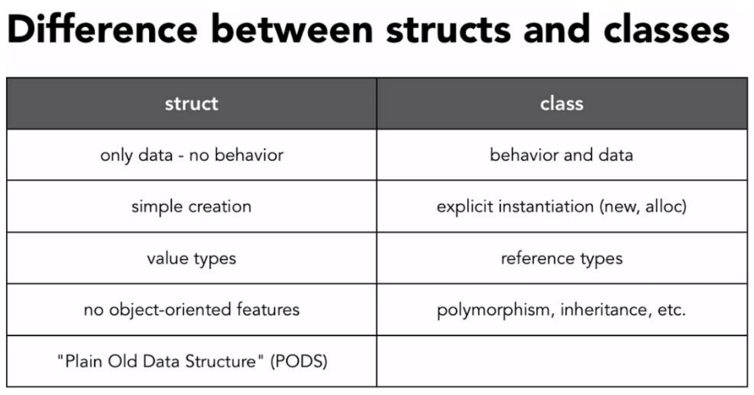
```
// first book
string bookTitle = "Dark and Stormy Night";
double bookPrice = 12.95;
bool bookPublished = true;

// 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;
```

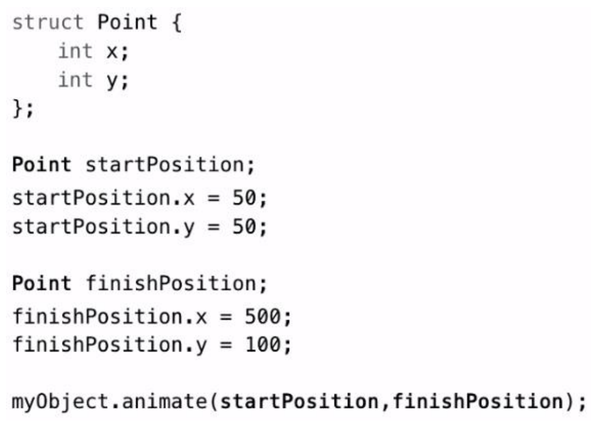




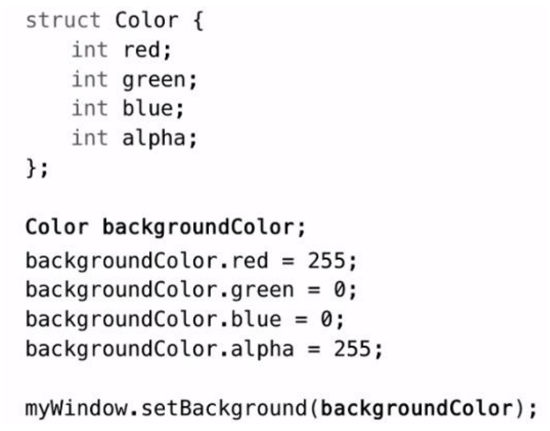
Difference between structs and classes

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)	

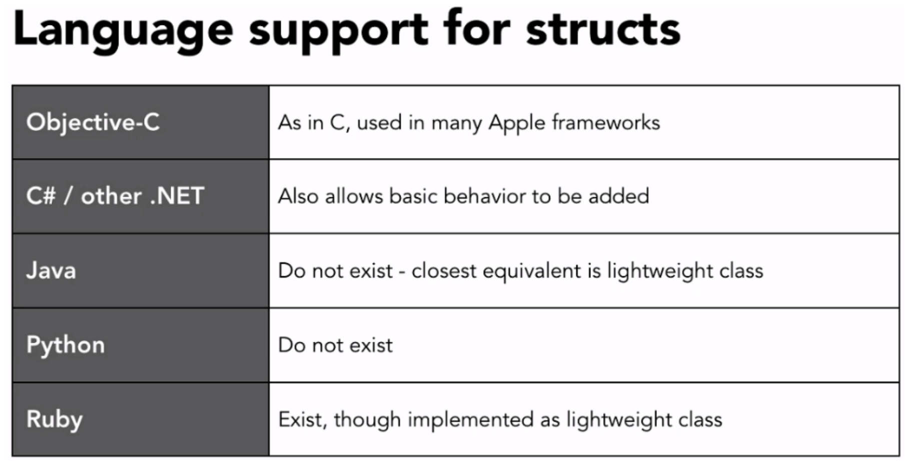
Examples of PODS



```
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

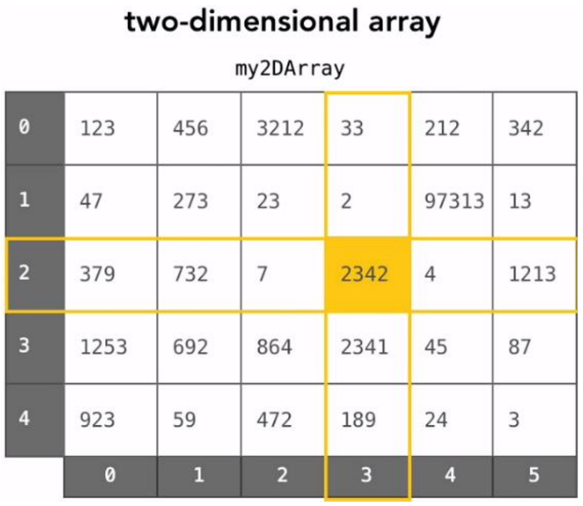
Objective-C	As in C, used in many Apple frameworks
C# / other .NET	Also allows basic behavior to be added
Java	Do not exist - closest equivalent is lightweight class
Python	Do not exist
Ruby	Exist, though implemented as lightweight class

Collections

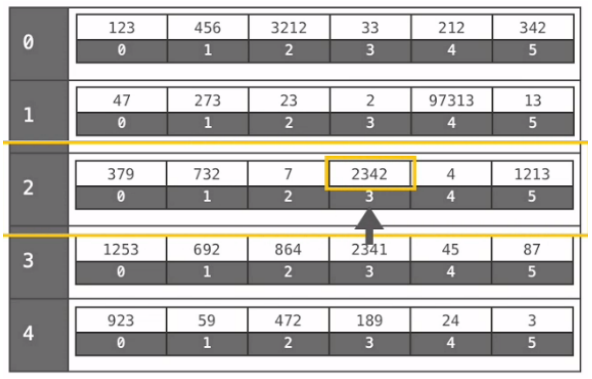
Arrays

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

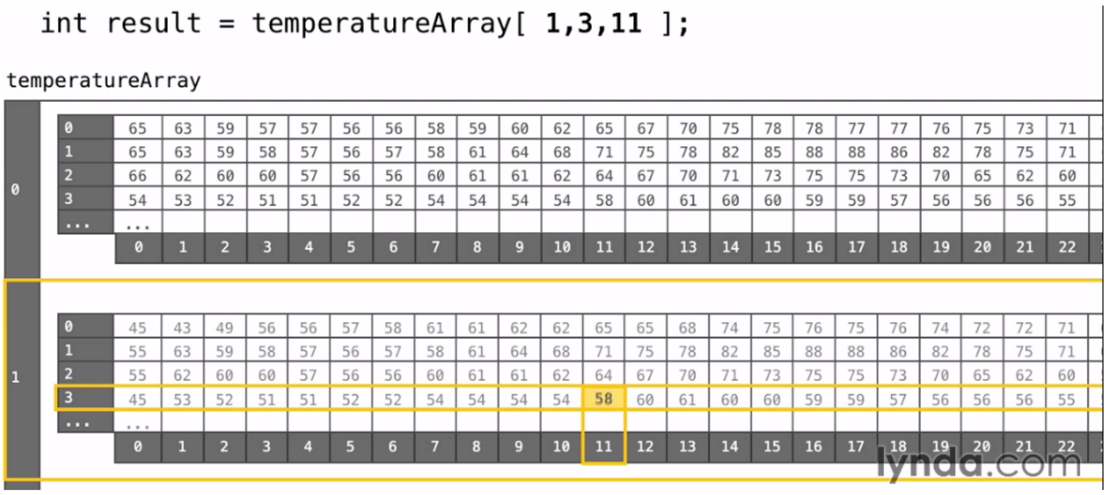
Multi dimensional arrays are simply arrays of arrays.



	0	1	2	3	4	5
0	123	456	3212	33	212	342
1	47	273	23	2	97313	13
2	379	732	7	2342	4	1213
3	1253	692	864	2341	45	87
4	923	59	472	189	24	3



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



```
int result = temperatureArray[ 1,3,11 ];
```

0

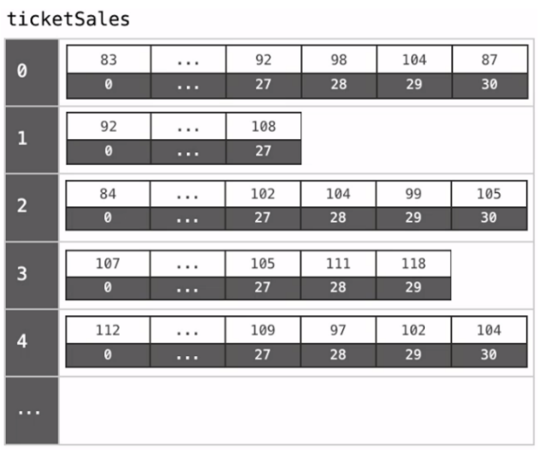
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
0	65	63	59	57	57	56	56	58	59	60	62	65	67	70	75	78	78	77	77	76	75	73	71
1	65	63	59	58	57	56	57	58	61	64	68	71	75	78	82	85	88	88	86	82	78	75	71
2	66	62	60	60	57	56	56	60	61	61	62	64	67	70	71	73	75	75	73	70	65	62	60
3	54	53	52	51	51	52	52	54	54	54	54	58	60	61	60	60	59	59	57	56	56	56	55
...	...																						

1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
0	45	43	49	56	56	57	58	61	61	62	62	65	65	68	74	75	76	75	76	74	72	72	71
1	55	63	59	58	57	56	57	58	61	64	68	71	75	78	82	85	88	88	86	82	78	75	71
2	55	62	60	60	57	56	56	60	61	61	62	64	67	70	71	73	75	75	73	70	65	62	60
3	45	53	52	51	51	52	52	54	54	54	54	58	60	61	60	60	59	59	57	56	56	56	55
...	...																						

Jagged Arrays

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



0	83 [0]	...	92 [27]	98 [28]	104 [29]	87 [30]
1	92 [0]	...	108 [27]			
2	84 [0]	...	102 [27]	104 [28]	99 [29]	105 [30]
3	107 [0]	...	105 [27]	111 [28]	118 [29]	
4	112 [0]	...	109 [27]	97 [28]	102 [29]	104 [30]
...						

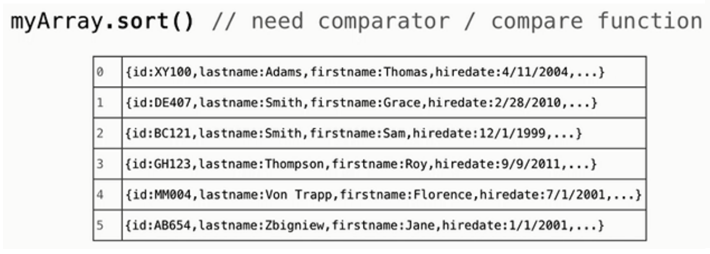
```
int[][] ticketSales = new int[12][]
for each month in ticketSales
    if april, june, september, november
       create array of 30 elements
    else if february and leap year
       create array of 29 elements
    else if february and not leap year
       create array of 28 elements
    else
  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

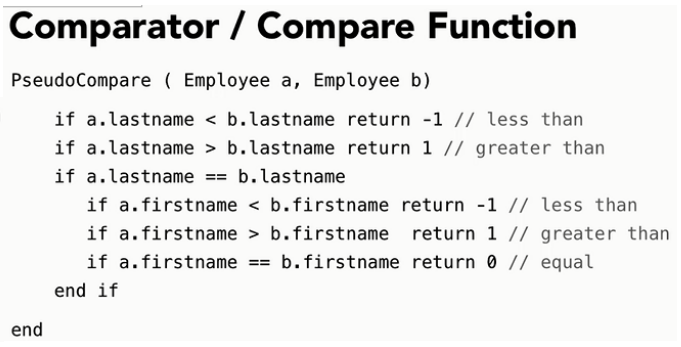
Make a difference between sorting and comparing.

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,...}

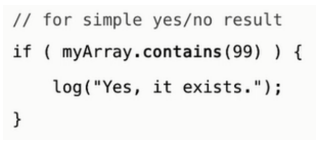


Comparator / Compare Function

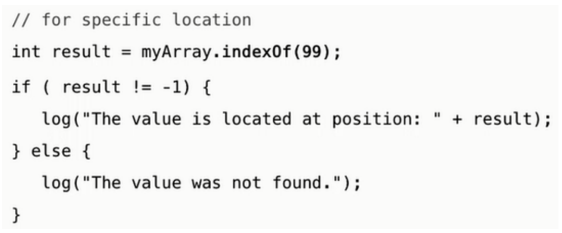
```
PseudoCompare ( Employee a, Employee b)

  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.

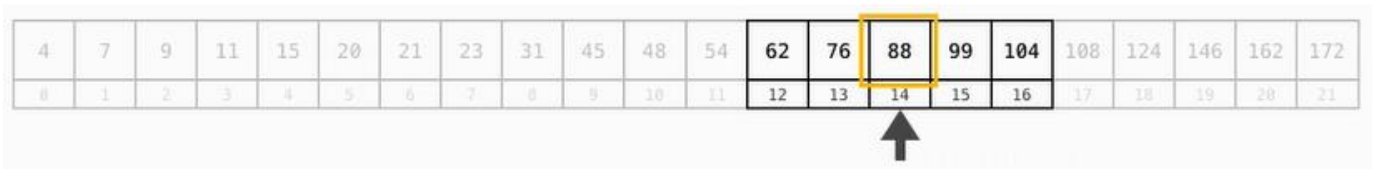


```
// for simple yes/no result
if ( myArray.contains(99) ) {
    log("Yes, it exists.");
}
```



```
// 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**.



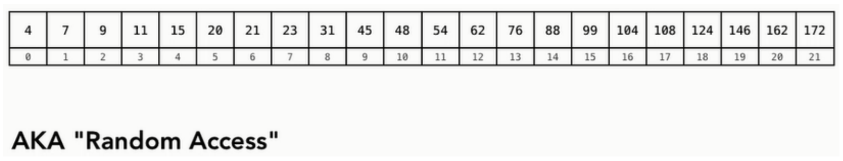
Lists

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.

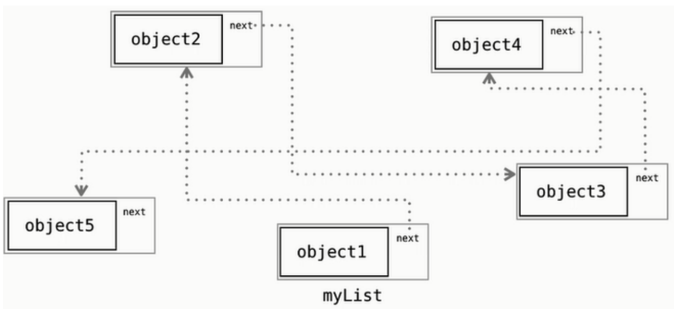


4	7	9	11	15	20	21	23	31	45	48	54	62	76	88	99	104	108	124	146	162	172
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

AKA "Random Access"

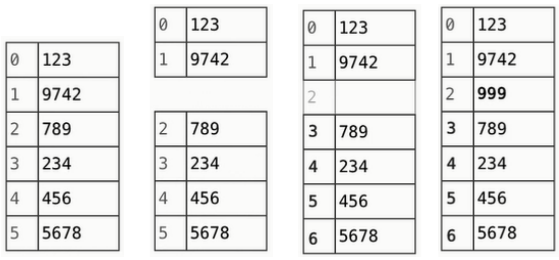
Lists: Sequential Access

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.

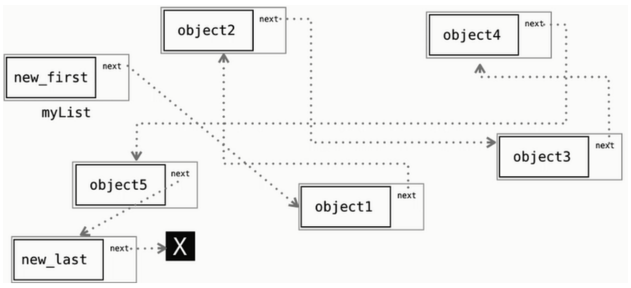


0	123
1	9742
2	789
3	234
4	456
5	5678

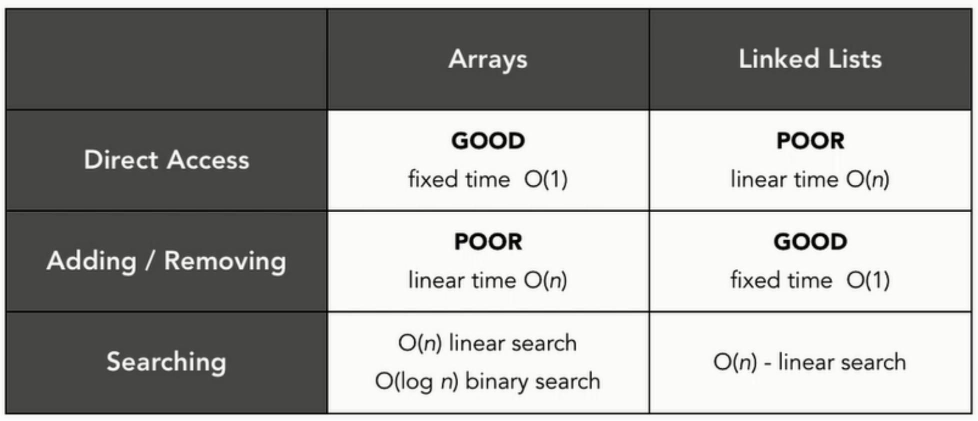
0	123
1	9742
2	789
3	234
4	456
5	5678

0	123
1	9742
2	
3	789
4	234
5	456
6	5678

0	123
1	9742
2	**999**
3	789
4	234
5	456
6	5678



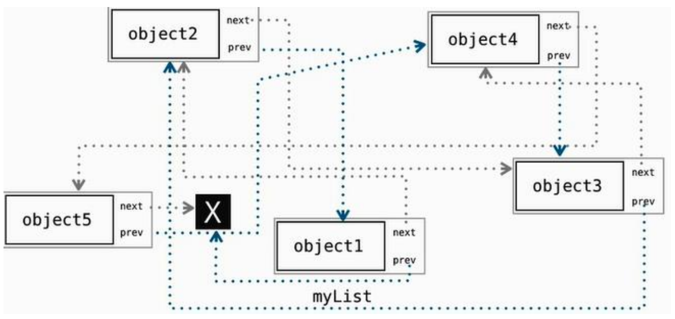
Lists (Singly Linked) on the other hand are as simple as adding a new node. Removal simply changes the next information in a node. These are fixed time operations and are size agnostic.



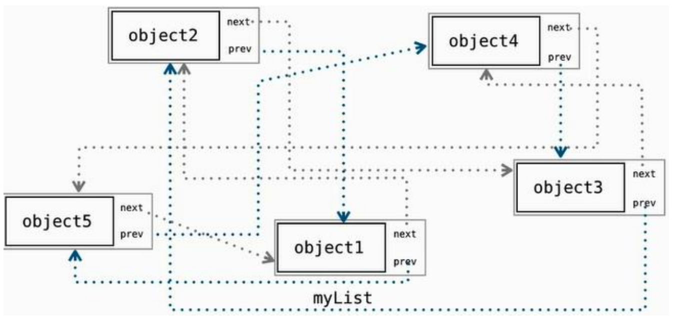
	Arrays	Linked Lists
Direct Access	**GOOD** fixed time O(1)	**POOR** linear time O(*n*)
Adding / Removing	**POOR** linear time O(*n*)	**GOOD** fixed time O(1)
Searching	O(*n*) linear search O(log *n*) binary search	O(*n*) - linear search

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

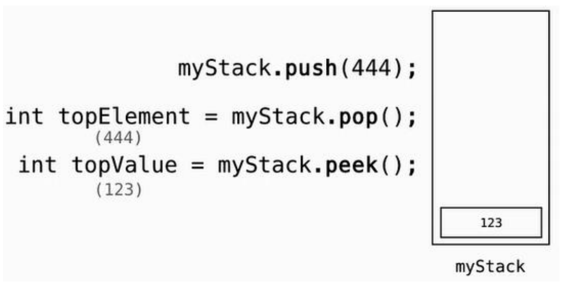


Circular

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`

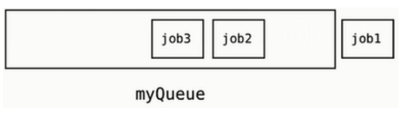
Stacks LIFO



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

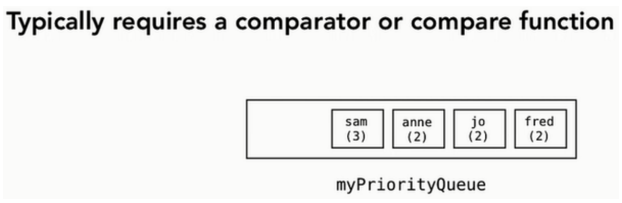
Java	`Stack      (push / pop / peek)`
C#	`Stack      (Push / Pop / Peek)`
Python	`use lists  (append / pop)`
Ruby	`use Array  (push / pop)`
Objective-C	`use NSMutableArray`
C++	`std::stack (push / pop)`

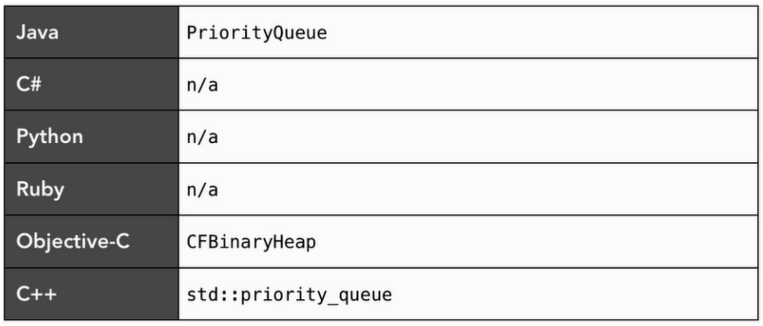
Queues FIFO



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)`

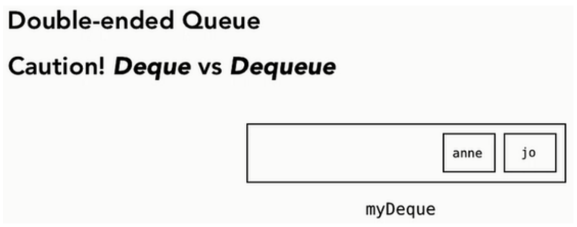
Priority Queues


Typically requires a comparator or compare function

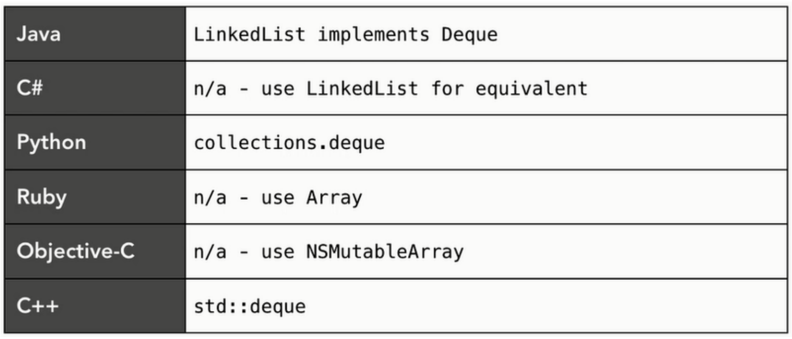


Java	PriorityQueue
C#	n/a
Python	n/a
Ruby	n/a
Objective-C	CFBinaryHeap
C++	std::priority_queue

Deques


Double-ended Queue

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

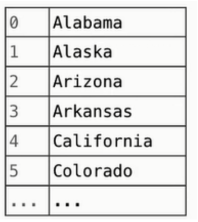


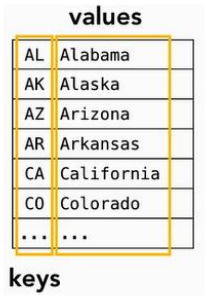
Java	LinkedList implements Deque
C#	n/a - use LinkedList for equivalent
Python	collections.deque
Ruby	n/a - use Array
Objective-C	n/a - use NSMutableArray
C++	std::deque

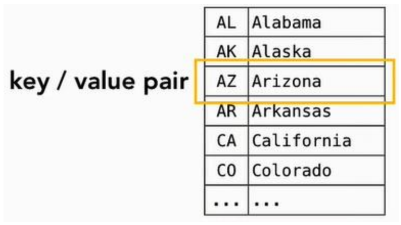
Hash-Based Data Structures

Associative Array

They cannot have duplicate keys.



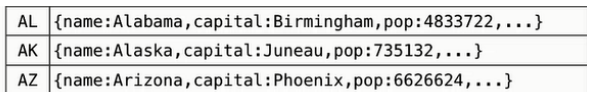




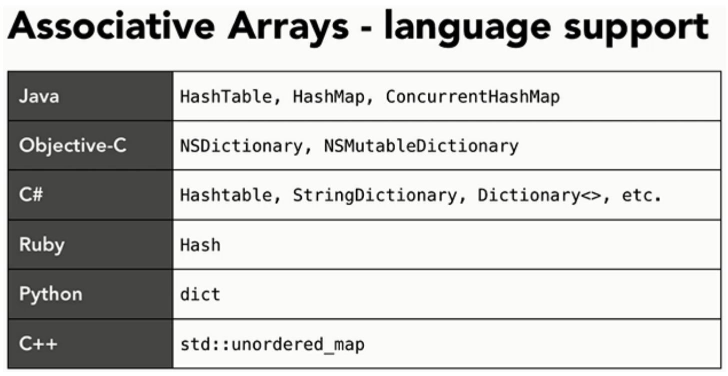
Array

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,...}



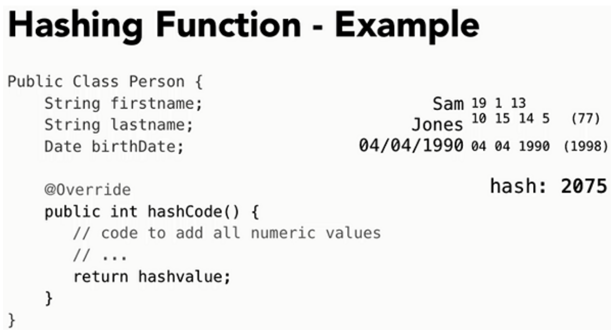
Associative Arrays - language support

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:

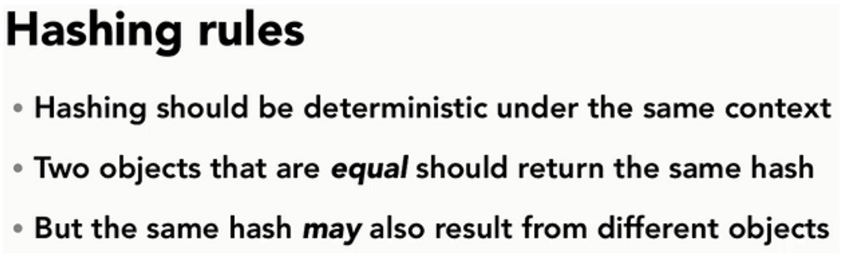
Object 1 > Hash Function > 7146759310cegdc1
Object 2 > Hash Function > 542427436



Hashing Function - Example

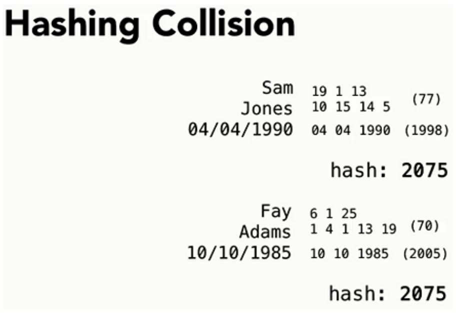
```
Public Class Person {
    String firstname;
    String lastname;
    Date birthDate;

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

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

hash: 2075
```



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



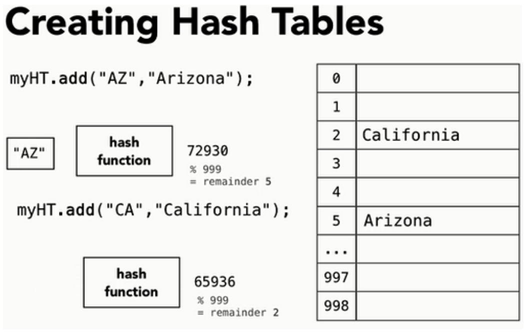
Hashing Collision

```
       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.

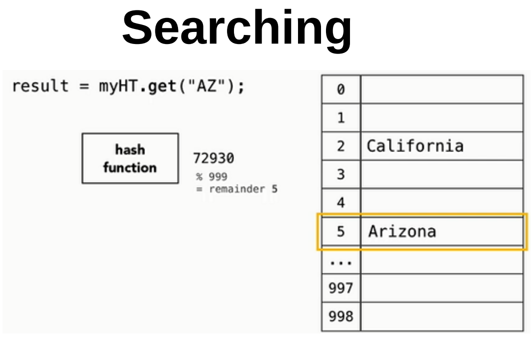
Hash Tables – Dictionaries

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



Creating Hash Tables

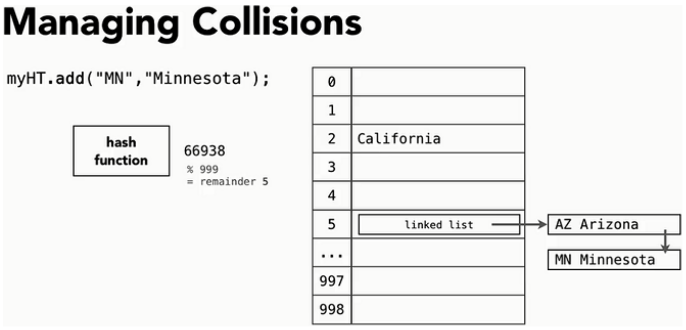
```
myHT.add("AZ","Arizona");
myHT.add("CA","California");
```



Searching

```
result = myHT.get("AZ");
```

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.

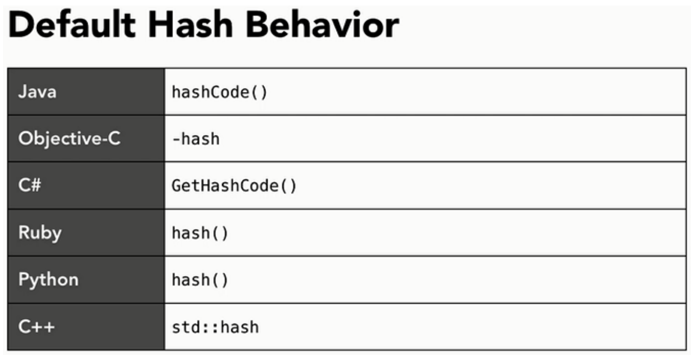


Managing Collisions

```
myHT.add("MN","Minnesota");
```

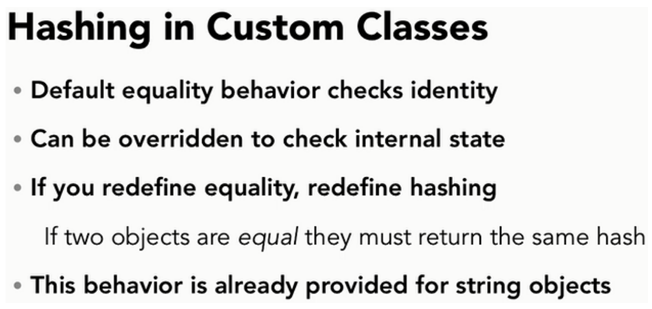
This process of managing collisions is called Separate Chaining.

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

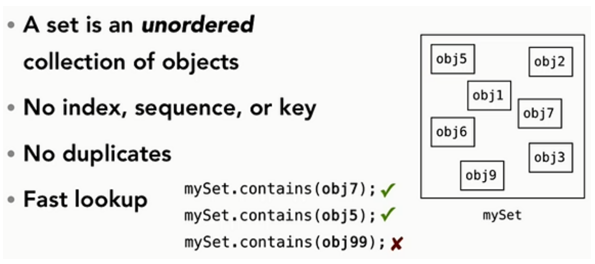
- **Default equality behavior checks identity**
- **Can be overridden to check internal state**
- **If you redefine equality, redefine hashing**

  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.

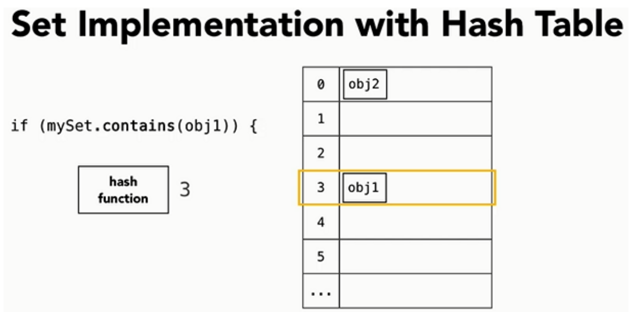
Sets

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.**



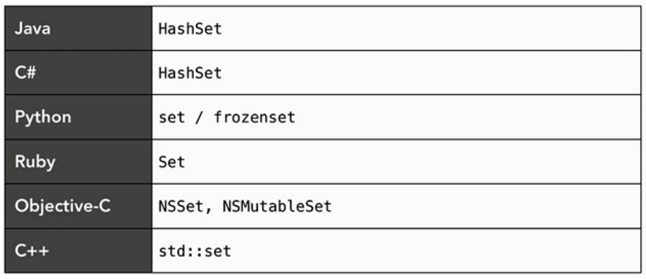
- A set is an ***unordered*** collection of objects
- No index, sequence, or key
- No duplicates
- Fast lookup

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



Set Implementation with Hash Table

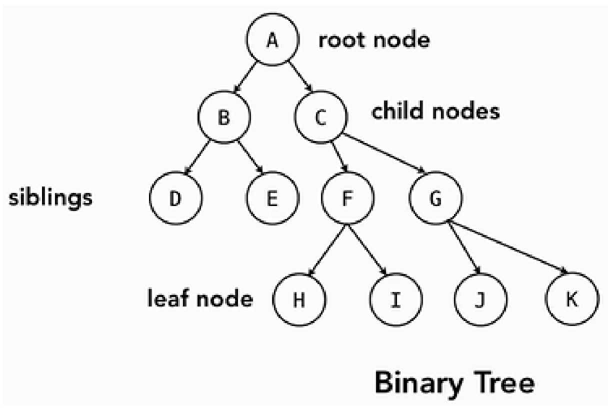
```
if (mySet.contains(obj1)) {
```

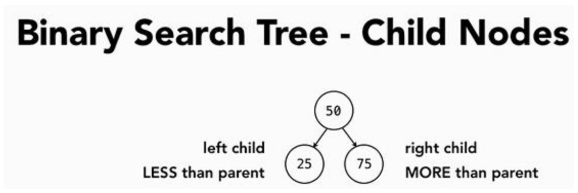


Java	`HashSet`
C#	`HashSet`
Python	`set / frozenset`
Ruby	`Set`
Objective-C	`NSSet, NSMutableSet`
C++	`std::set`

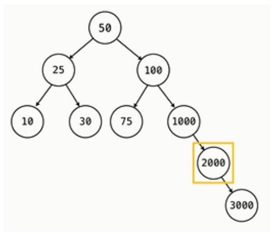
Trees

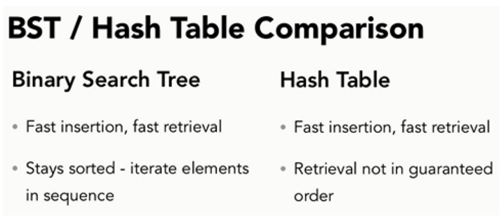
Binary Trees are allowed to have a maximum of 2 childs. Leaf nodes have no childs.





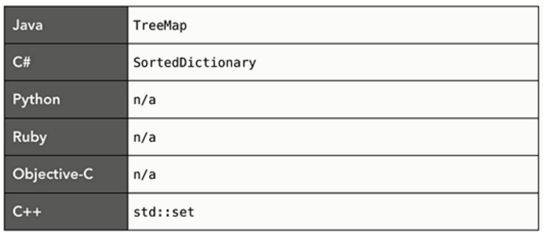
Binary Search Tree - Child Nodes





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

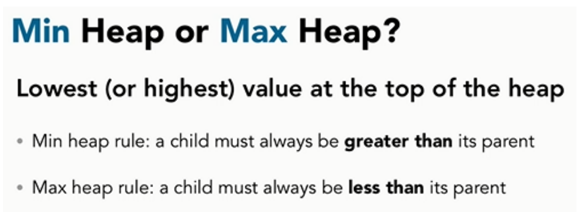


Java	TreeMap
C#	SortedDictionary
Python	n/a
Ruby	n/a
Objective-C	n/a
C++	std::set

Heaps

Not to be confused with Heap memory allocation.

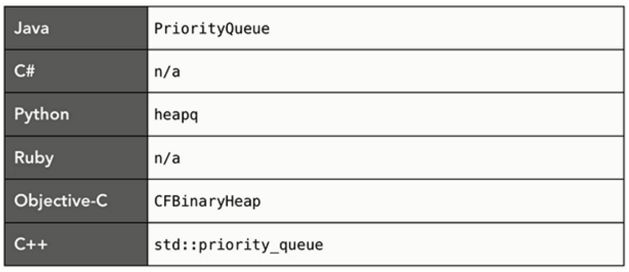
Items are added top to bottom, left to right. This is a balanced structure and no space is left unallocated. Items are swapped whenever needed to keep the balance.



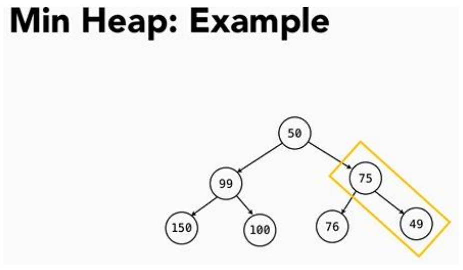
Min Heap or Max Heap?

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

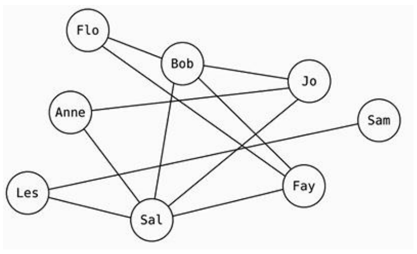


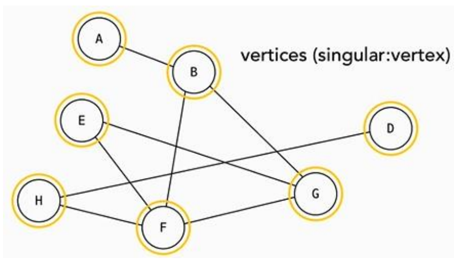
Java	PriorityQueue
C#	n/a
Python	heapq
Ruby	n/a
Objective-C	CFBinaryHeap
C++	std::priority_queue

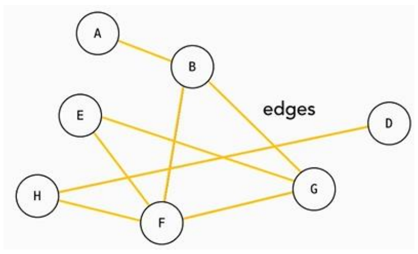


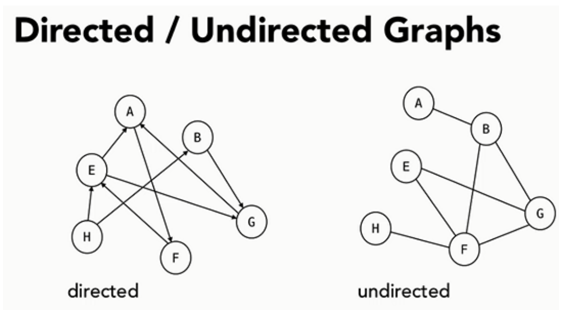
Graphs

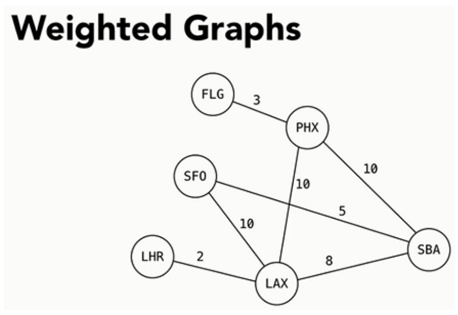
Useful for complex systems of interconnected points. Linked Lists, Trees and Heaps are all graph implementations. There is no support for generic graphs in any languages because they are too specific.







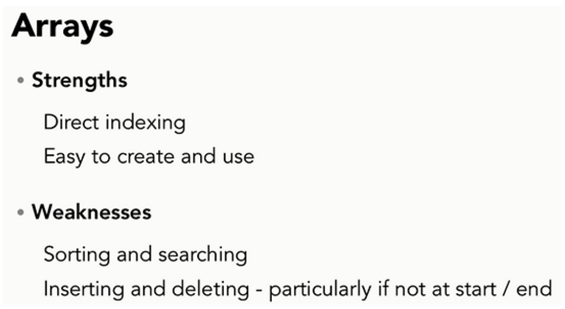




Summary

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?



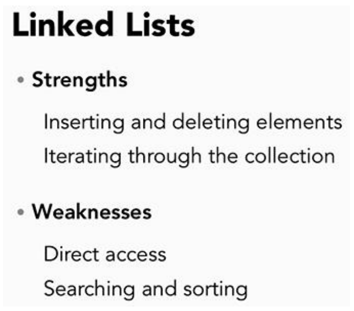
Arrays

- **Strengths**

  Direct indexing
  Easy to create and use

- **Weaknesses**

  Sorting and searching
  Inserting and deleting - particularly if not at start / end



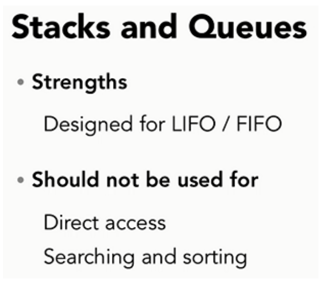
Linked Lists

- **Strengths**

  Inserting and deleting elements
  Iterating through the collection

- **Weaknesses**

  Direct access
  Searching and sorting



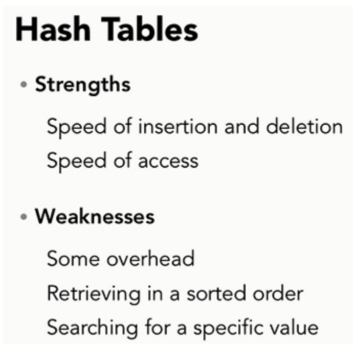
Stacks and Queues

- **Strengths**

  Designed for LIFO / FIFO

- **Should not be used for**

  Direct access
  Searching and sorting



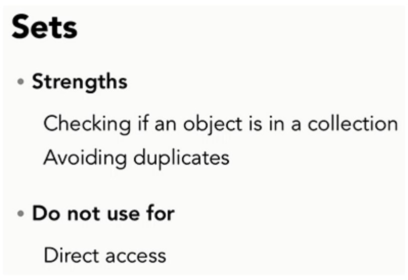
Hash Tables

- **Strengths**

  Speed of insertion and deletion
  Speed of access

- **Weaknesses**

  Some overhead
  Retrieving in a sorted order
  Searching for a specific value



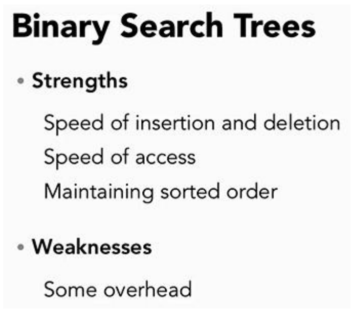
Sets

- **Strengths**

  Checking if an object is in a collection
  Avoiding duplicates

- **Do not use for**

  Direct access



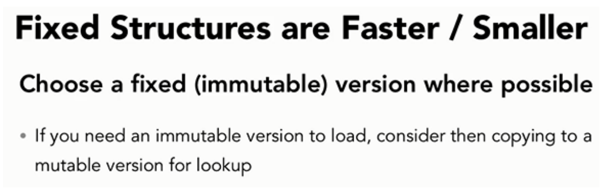
Binary Search Trees

- **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.

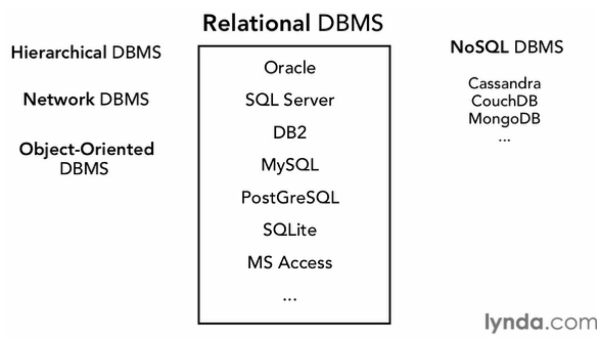
It's not about where to put the data, but rather to manage it effectively.

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.

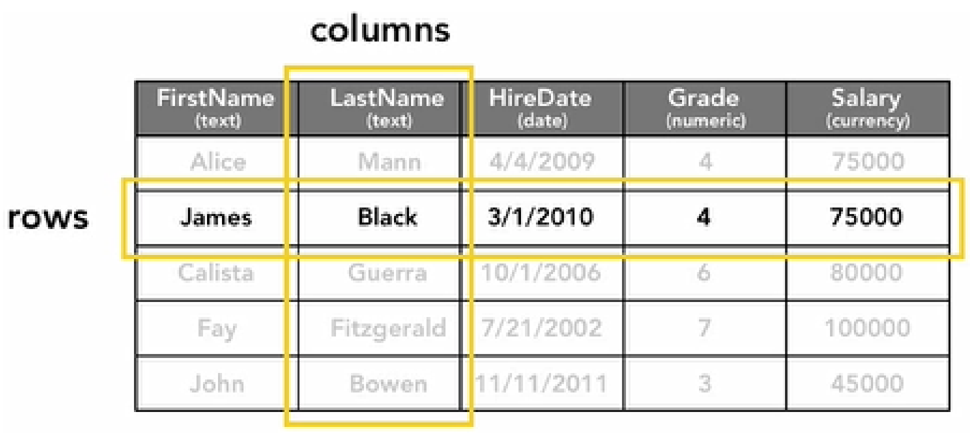
Relational DBMSs are by far the most used ones, but there are some others.



Database Fundamentals

Tables

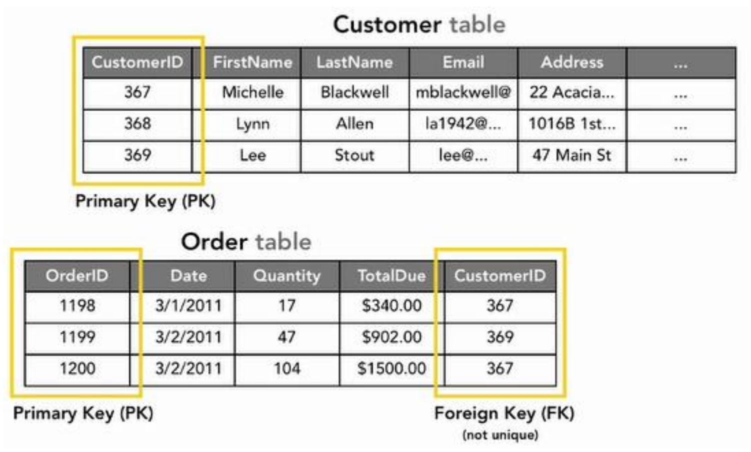
Databases are in fact collections of tables. Without them, a database would be an empty useless shell.


columns
rows

FirstName (text)	LastName (text)	HireDate (date)	Grade (numeric)	Salary (currency)
Alice	Mann	4/4/2009	4	75000
James	Black	3/1/2010	4	75000
Calista	Guerra	10/1/2006	6	80000
Fay	Fitzgerald	7/21/2002	7	100000
John	Bowen	11/11/2011	3	45000

Table Relationships

One to many


Customer table

CustomerID	FirstName	LastName	Email	Address	...
367	Michelle	Blackwell	mblackwell@	22 Acacia...	...
368	Lynn	Allen	la1942@...	1016B 1st...	...
369	Lee	Stout	lee@...	47 Main St	...

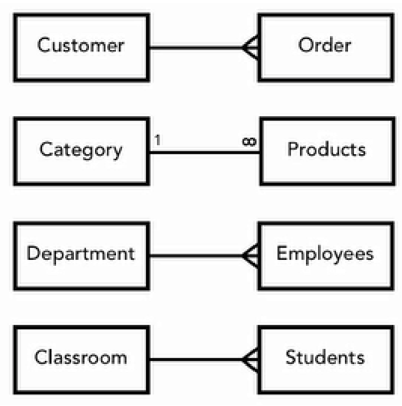
Primary Key (PK)

Order table

OrderID	Date	Quantity	TotalDue	CustomerID
1198	3/1/2011	17	$340.00	367
1199	3/2/2011	47	$902.00	369
1200	3/2/2011	104	$1500.00	367

Primary Key (PK)

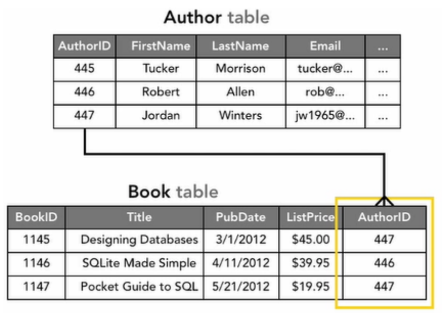
Foreign Key (FK) (not unique)



Many to many

- Not so obvious.
- Cannot be done directly.

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

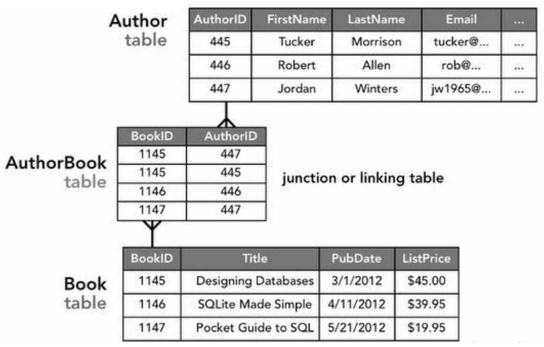

Author table

AuthorID	FirstName	LastName	Email	...
445	Tucker	Morrison	tucker@...	...
446	Robert	Allen	rob@...	...
447	Jordan	Winters	jw1965@...	...

Book table

BookID	Title	PubDate	ListPrice	AuthorID
1145	Designing Databases	3/1/2012	$45.00	447
1146	SQLite Made Simple	4/11/2012	$39.95	446
1147	Pocket Guide to SQL	5/21/2012	$19.95	447

becomes this


Author table

AuthorID	FirstName	LastName	Email	...
445	Tucker	Morrison	tucker@...	...
446	Robert	Allen	rob@...	...
447	Jordan	Winters	jw1965@...	...

AuthorBook table

BookID	AuthorID
1145	447
1145	445
1146	446
1147	447

junction or linking table

Book 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

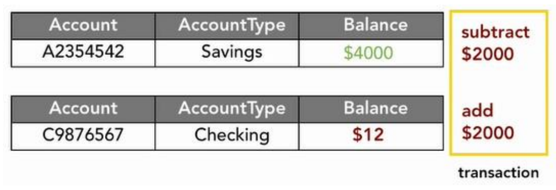
One to many

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..



Account	AccountType	Balance
A2354542	Savings	$4000

subtract $2000

Account	AccountType	Balance
C9876567	Checking	$12

add $2000

transaction

Debit

Credit

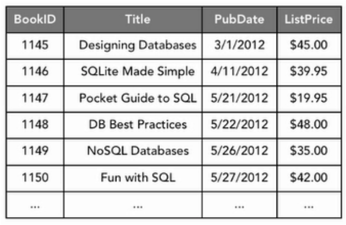
A transaction needs to be **ACID**:

- 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

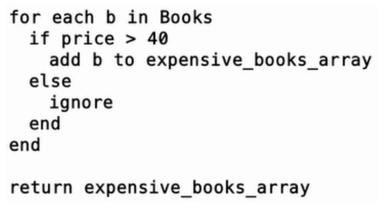
It is a declarative query language, not a procedural, imperative language. In procedural languages you describe the steps whereas in a query language you describe the outcome.

Table



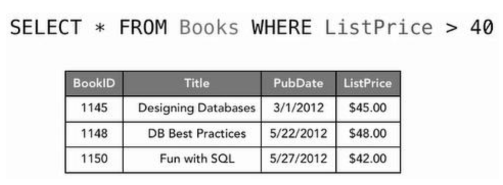
BookID	Title	PubDate	ListPrice
1145	Designing Databases	3/1/2012	$45.00
1146	SQLite Made Simple	4/11/2012	$39.95
1147	Pocket Guide to SQL	5/21/2012	$19.95
1148	DB Best Practices	5/22/2012	$48.00
1149	NoSQL Databases	5/26/2012	$35.00
1150	Fun with SQL	5/27/2012	$42.00
...	...	...	...

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

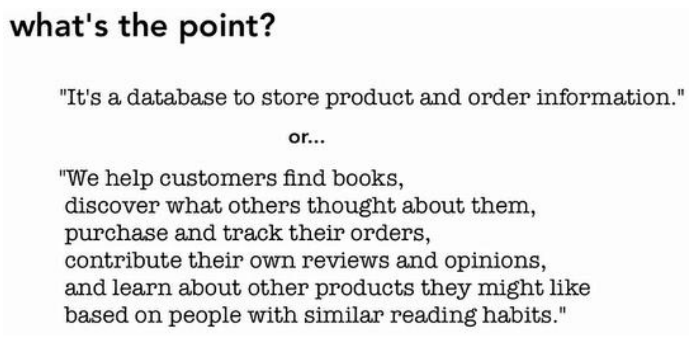
Database Planning

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:

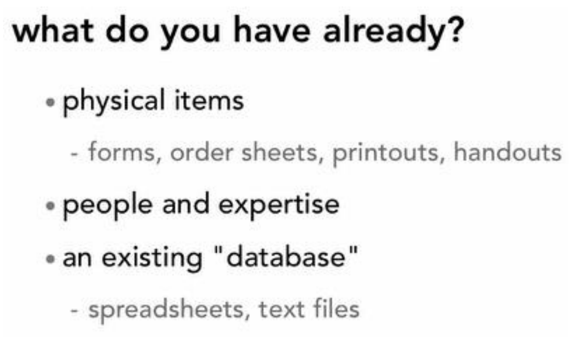


what's the point?

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

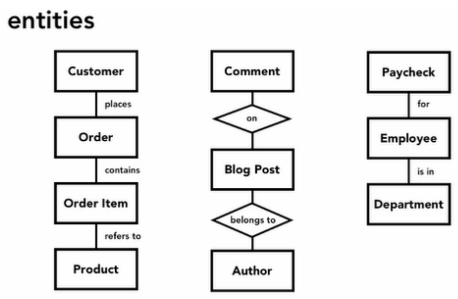
or...

"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



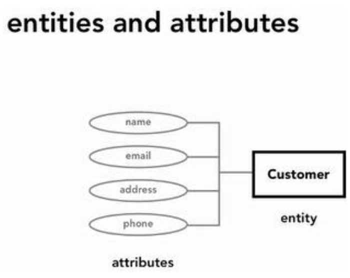
Separate tables are created for each object i.e. entity.

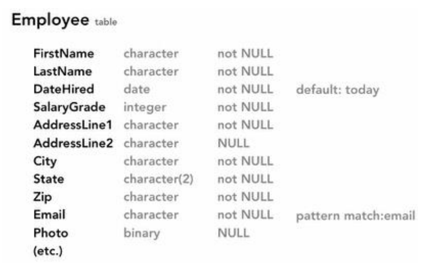
Tables should be named in singular form.

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.)			

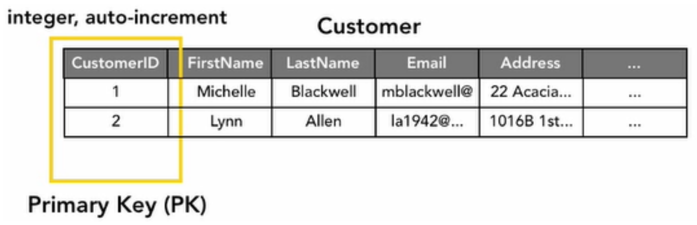
Columns can be named in many styles (FirstName, first_name, firstname, fName, strFirstName, firstName) Use: FirstName i.e. Uppercase each word.

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.

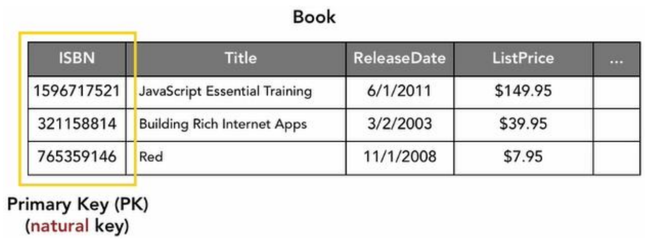
Primary Keys

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
Primary Key (PK)

Customer

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

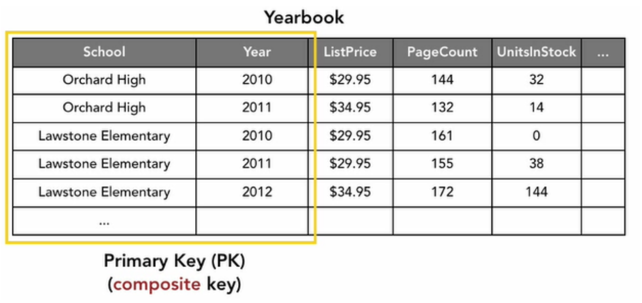

Primary Key (PK)
(natural key)

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	

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.


Primary Key (PK)
(composite key)

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	
...					

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

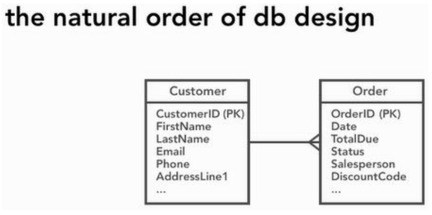
Look both ways (From each perspective) when trying to determine the relationship.

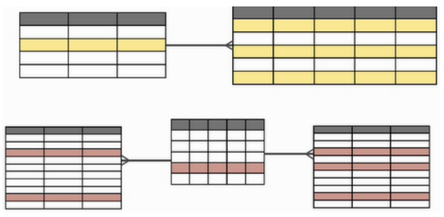
relationship (cardinality) options

One-To-One

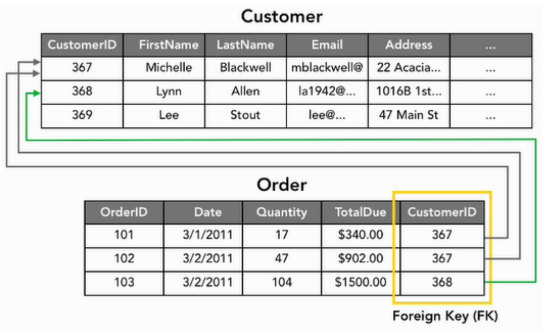
One-To-Many

Many-To-Many





One-To-Many

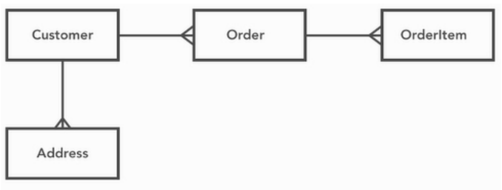

Customer

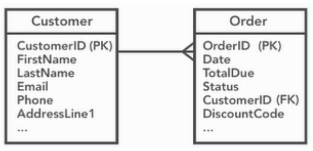
CustomerID	FirstName	LastName	Email	Address	...
367	Michelle	Blackwell	mblackwell@	22 Acacia...	...
368	Lynn	Allen	la1942@...	1016B 1st...	...
369	Lee	Stout	lee@...	47 Main St	...

Order

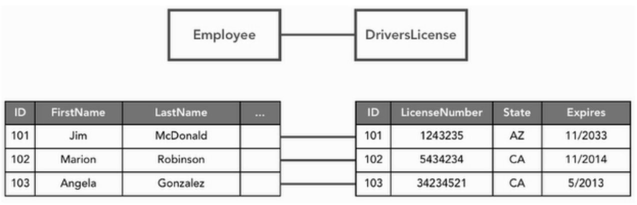
OrderID	Date	Quantity	TotalDue	CustomerID
101	3/1/2011	17	$340.00	367
102	3/2/2011	47	$902.00	367
103	3/2/2011	104	$1500.00	368

Foreign Key (FK)





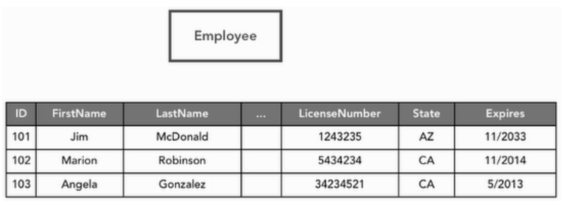
One-To-One


Employee
DriversLicense

ID	FirstName	LastName	...
101	Jim	McDonald	
102	Marion	Robinson	
103	Angela	Gonzalez	

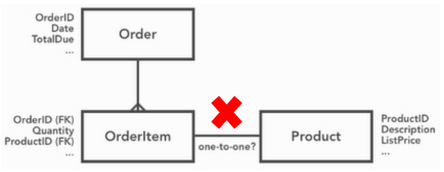
ID	LicenseNumber	State	Expires
101	1243235	AZ	11/2033
102	5434234	CA	11/2014
103	34234521	CA	5/2013

might as well

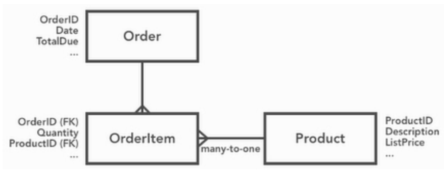


ID	FirstName	LastName	...	LicenseNumber	State	Expires
101	Jim	McDonald		1243235	AZ	11/2033
102	Marion	Robinson		5434234	CA	11/2014
103	Angela	Gonzalez		34234521	CA	5/2013

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

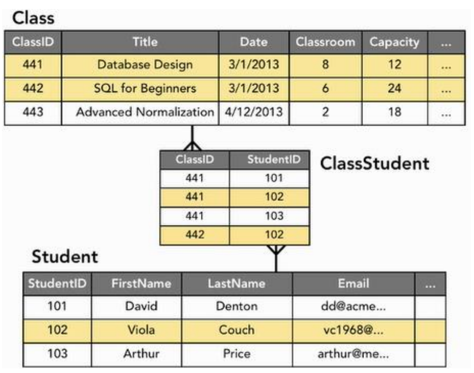


is in fact



Many-To-Many

Can't be represented directly.


Class

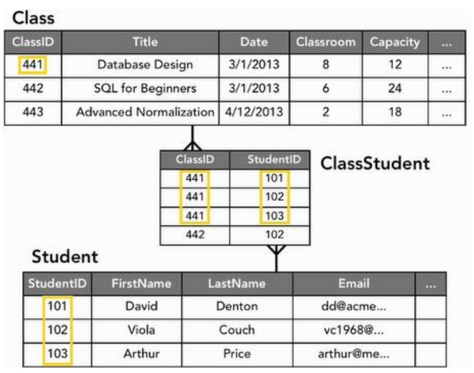
ClassID	Title	Date	Classroom	Capacity	...
441	Database Design	3/1/2013	8	12	...
442	SQL for Beginners	3/1/2013	6	24	...
443	Advanced Normalization	4/12/2013	2	18	...

ClassStudent

ClassID	StudentID
441	101
441	102
441	103
442	102

Student

StudentID	FirstName	LastName	Email	...
101	David	Denton	dd@acme...	
102	Viola	Couch	vc1968@...	
103	Arthur	Price	arthur@me...	


Class

ClassID	Title	Date	Classroom	Capacity	...
441	Database Design	3/1/2013	8	12	...
442	SQL for Beginners	3/1/2013	6	24	...
443	Advanced Normalization	4/12/2013	2	18	...

ClassStudent

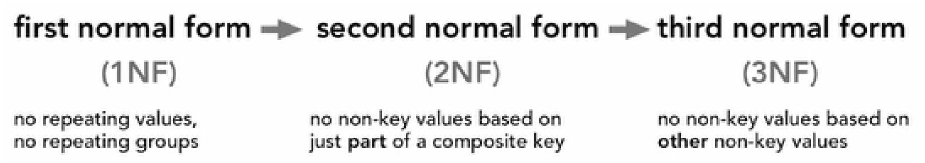
ClassID	StudentID
441	101
441	102
441	103
442	102

Student

StudentID	FirstName	LastName	Email	...
101	David	Denton	dd@acme...	
102	Viola	Couch	vc1968@...	
103	Arthur	Price	arthur@me...	

Optimization

Normalization



first normal form → second normal form → third normal form

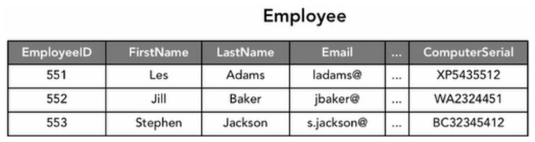
(1NF)	(2NF)	(3NF)
no repeating values, no repeating groups	no non-key values based on just part of a composite key	no non-key values based on other non-key values

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)

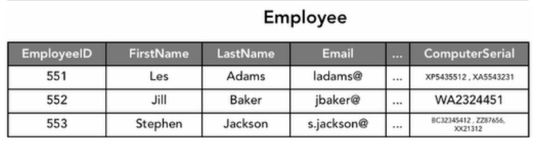
Each of the columns and tables should contain one and only one value without it repeating.



Employee

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

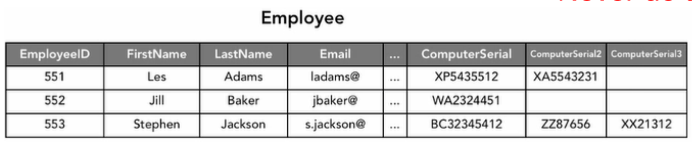
What happens when an employee needs 2 computers?



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

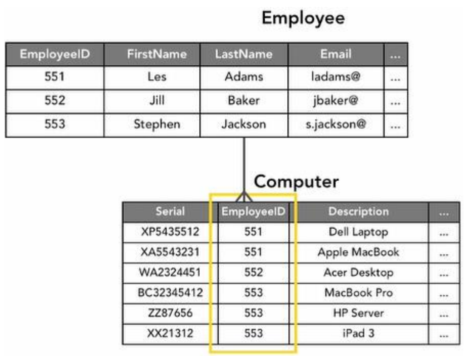
Never do this!



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

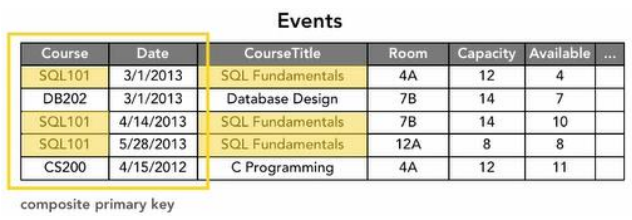
Serial	EmployeeID	Description	...
XP5435512	551	Dell Laptop	...
XA5543231	551	Apple MacBook	...
WA2324451	552	Acer Desktop	...
BC32345412	553	MacBook Pro	...
ZZ87656	553	HP Server	...
XX21312	553	iPad 3	...

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.**

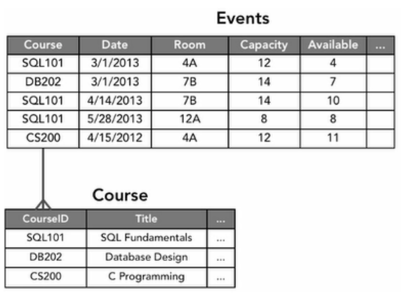


Events

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	

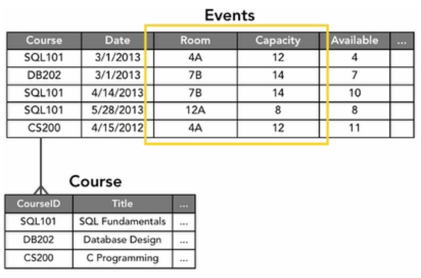
Course

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?”**..

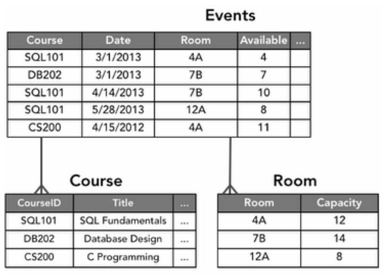


Events

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	

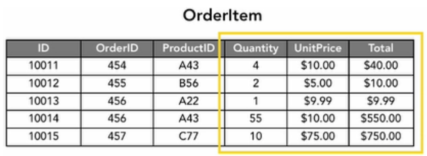
Course

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

Room

Room	Capacity
4A	12
7B	14
12A	8

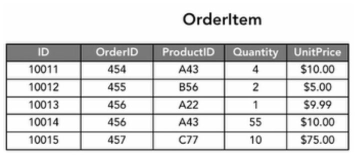
Same seats in a room.



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

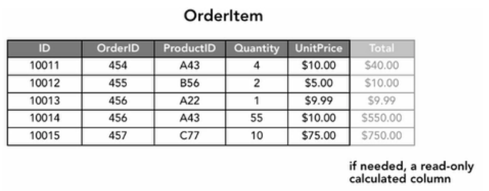
The total depends on the other values.



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

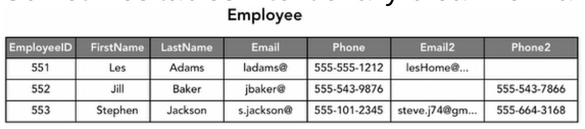
ID	OrderID	ProductID	Quantity	UnitPrice	Total
10011	454	A43	4	$10.00	$40.00
10012	455	B56	2	$5.00	$10.00
10013	456	A22	1	$9.99	$9.99
10014	456	A43	55	$10.00	$550.00
10015	457	C77	10	$75.00	$750.00

if needed, a read-only calculated column

Or make it a calculated read only column

Denormalization

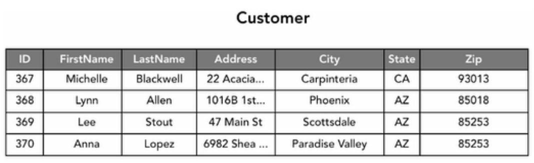
Sometimes tables intentionally break normalization, and some only seem like they do.



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.

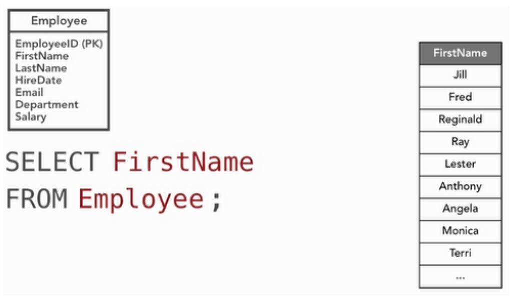
Querying

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	

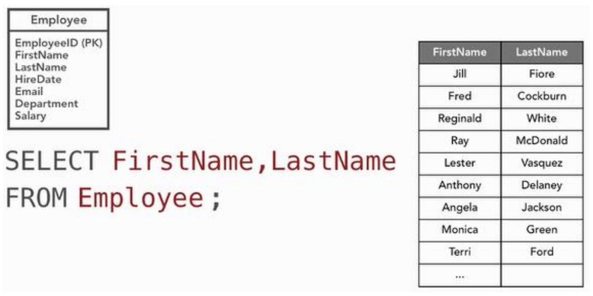
SELECT columns **FROM** table **WHERE** condition



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

```sql
SELECT FirstName
FROM Employee;
```

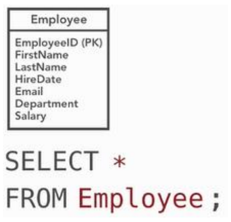
FirstName
Jill
Fred
Reginald
Ray
Lester
Anthony
Angela
Monica
Terri
...



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

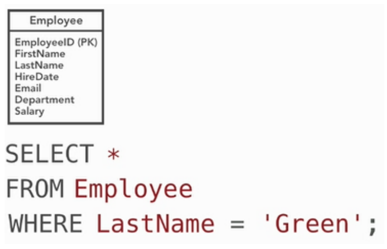
```sql
SELECT FirstName,LastName
FROM Employee;
```

FirstName	LastName
Jill	Fiore
Fred	Cockburn
Reginald	White
Ray	McDonald
Lester	Vasquez
Anthony	Delaney
Angela	Jackson
Monica	Green
Terri	Ford
...	



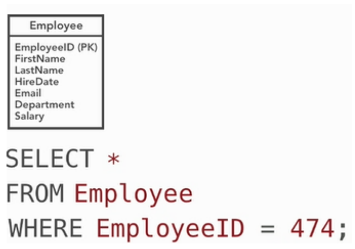
```
Employee
EmployeeID (PK)
FirstName
LastName
HireDate
Email
Department
Salary
```

```sql
SELECT *
FROM Employee;
```



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

```sql
SELECT *
FROM Employee
WHERE LastName = 'Green';
```



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

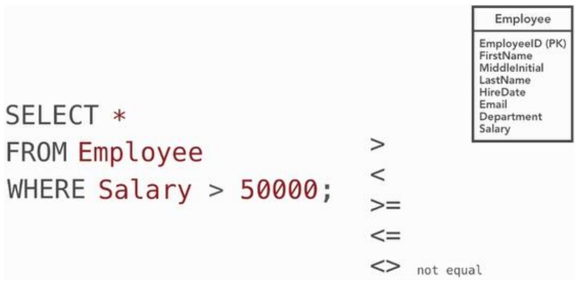
```sql
SELECT *
FROM Employee
WHERE EmployeeID = 474;
```

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

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

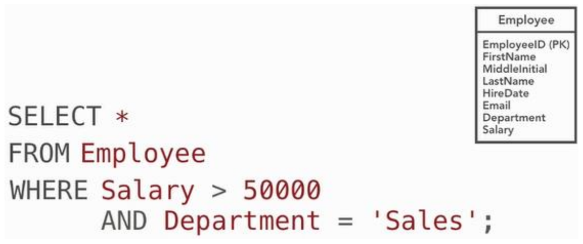
WHERE

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

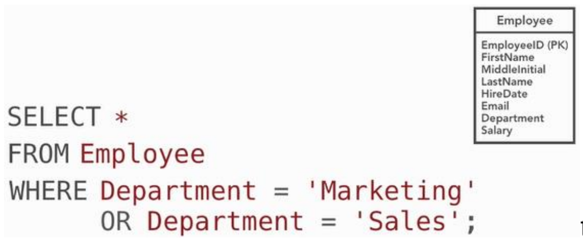


```
SELECT *
FROM Employee
WHERE Salary > 50000;
```

```
>
<
>=
<=
<>  not equal
```

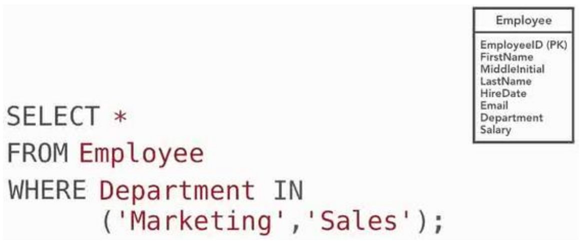


```
SELECT *
FROM Employee
WHERE Salary > 50000
      AND Department = 'Sales';
```

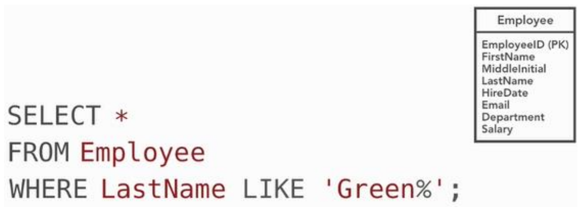


```
SELECT *
FROM Employee
WHERE Department = 'Marketing'
      OR Department = 'Sales';
```

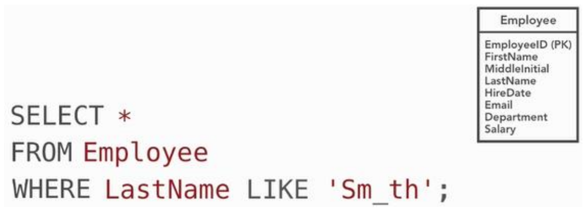
the same as



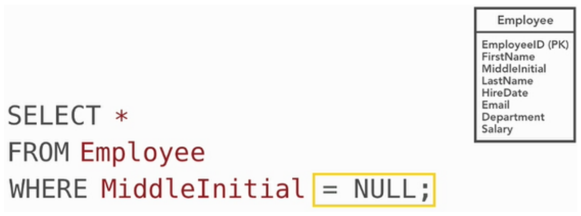
```
SELECT *
FROM Employee
WHERE Department IN
      ('Marketing','Sales');
```



```
SELECT *
FROM Employee
WHERE LastName LIKE 'Green%';
```

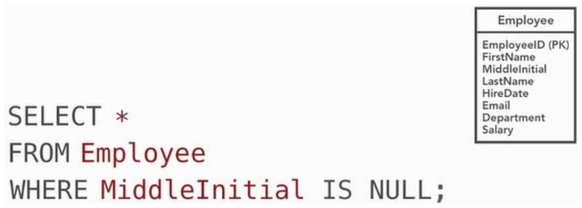


```
SELECT *
FROM Employee
WHERE LastName LIKE 'Sm_th';
```

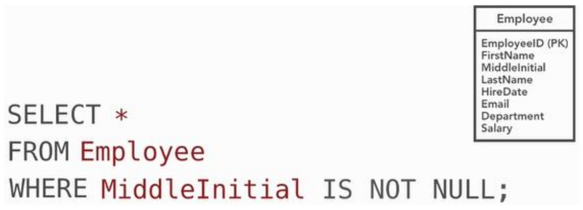


```
SELECT *
FROM Employee
WHERE MiddleInitial = NULL;
```

should be



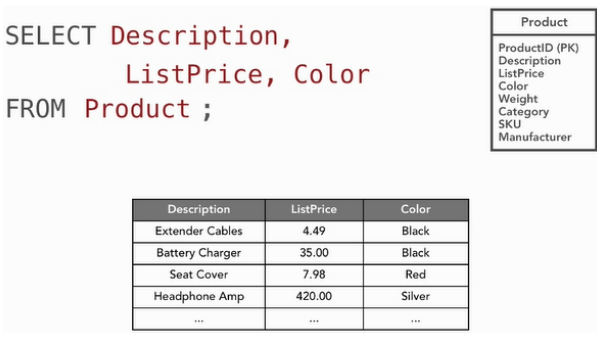
```
SELECT *
FROM Employee
WHERE MiddleInitial IS NULL;
```



```
SELECT *
FROM Employee
WHERE MiddleInitial IS NOT NULL;
```

Sorting

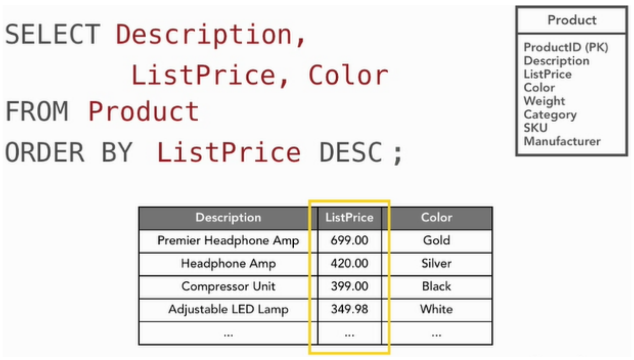
Ordering is **ascending** by **default** i.e. small to large.



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

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

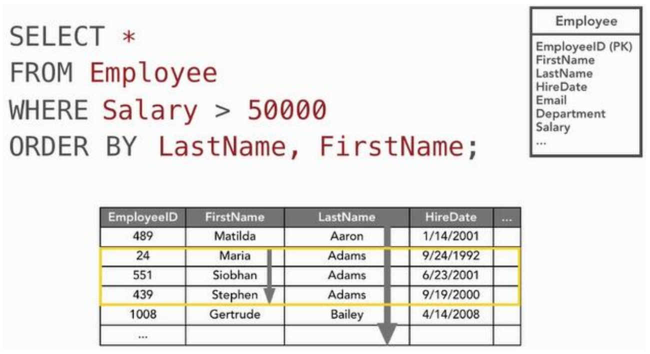
Description	ListPrice	Color
Extender Cables	4.49	Black
Battery Charger	35.00	Black
Seat Cover	7.98	Red
Headphone Amp	420.00	Silver
...	...	...



```
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
...	...	...



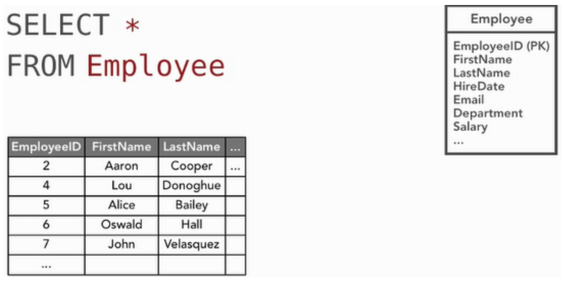
```
SELECT *
FROM Employee
WHERE Salary > 50000
ORDER BY LastName, FirstName;
```

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

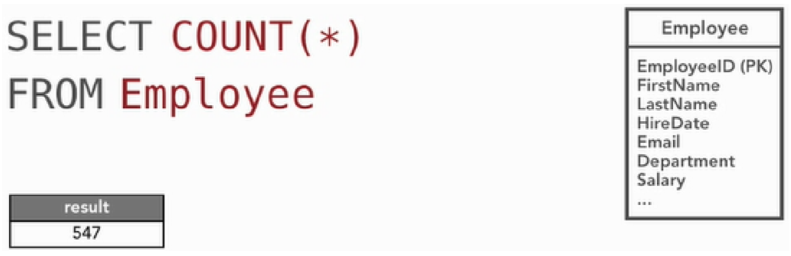
These do calculations on a set of data but return a single value.



```
SELECT *
FROM Employee
```

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

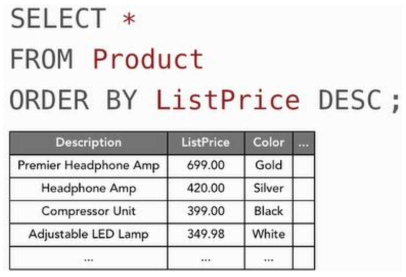
EmployeeID	FirstName	LastName	...
2	Aaron	Cooper	...
4	Lou	Donoghue	
5	Alice	Bailey	
6	Oswald	Hall	
7	John	Velasquez	
...			



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

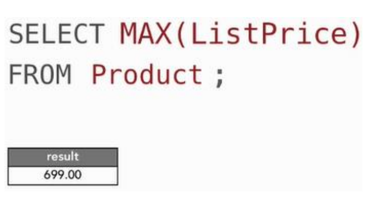
Employee
EmployeeID (PK)
FirstName
LastName
HireDate
Email
Department
Salary
...

result
547



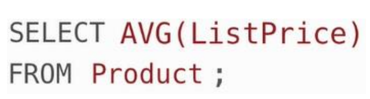
```
SELECT *
FROM Product
ORDER BY ListPrice DESC ;
```

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	
...	...	...	

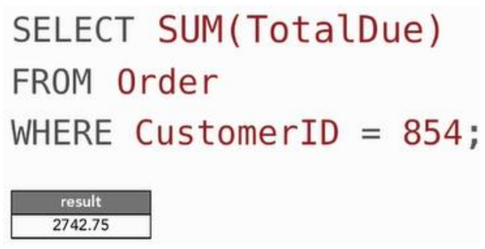


```
SELECT MAX(ListPrice)
FROM Product ;
```

result
699.00

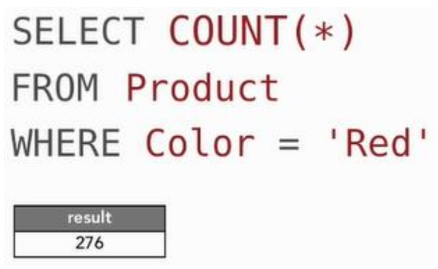


```
SELECT AVG(ListPrice)
FROM Product ;
```



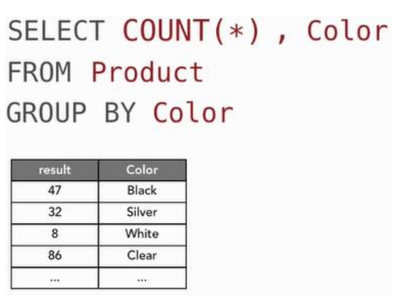
```
SELECT SUM(TotalDue)
FROM Order
WHERE CustomerID = 854;
```

result
2742.75



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

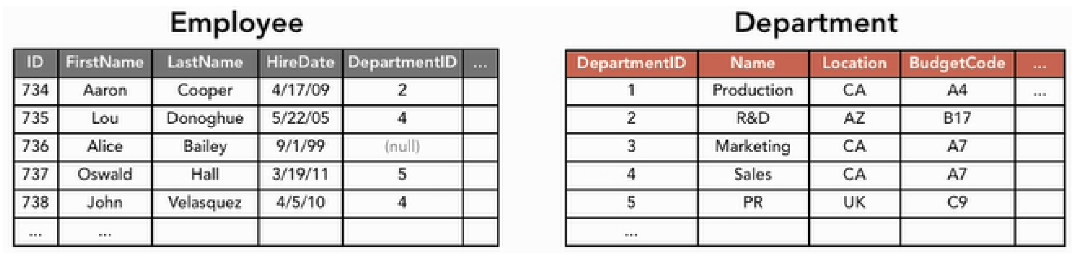
result
276



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

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

Joining Tables

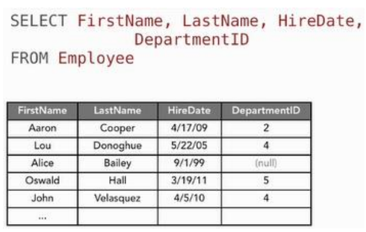


Employee

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	
...				

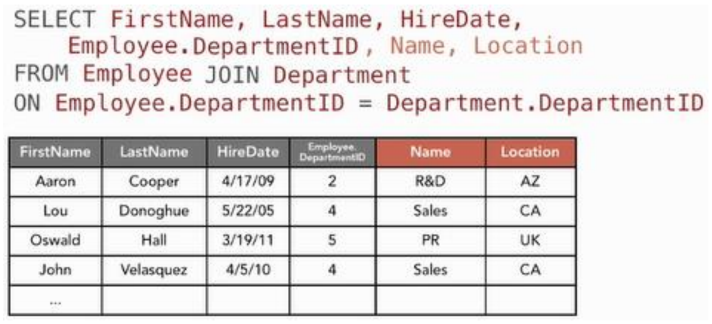


```
SELECT FirstName, LastName, HireDate,
                DepartmentID
FROM Employee
```

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.

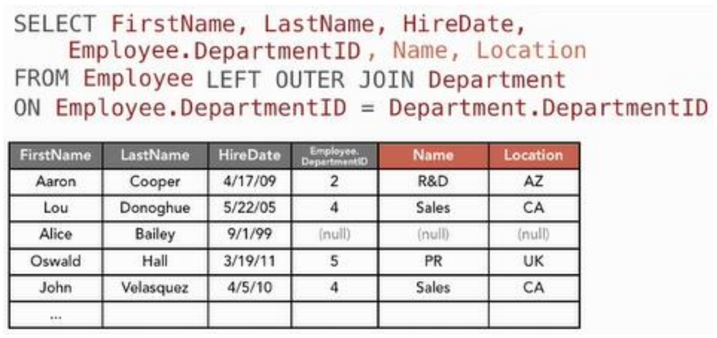
INNER JOIN = default JOIN (The null values will not appear in the result as well as the rows linked to it.)



```
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
...					

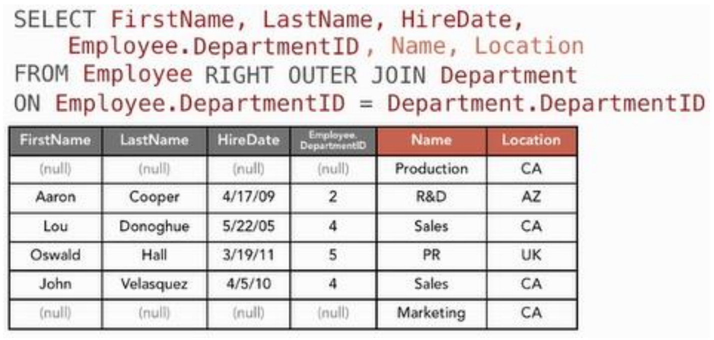
LEFT OUTER JOIN (LEFT means employee takes precedence because it is on the left side of the JOIN)



```
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
...					

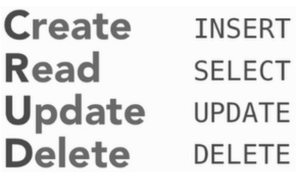
RIGHT OUTER JOIN (RIGHT = Department is on the right of JOIN)



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

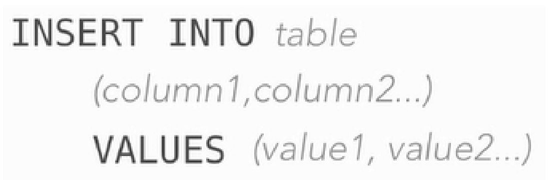
FirstName	LastName	HireDate	Employee.DepartmentID	Name	Location
(null)	(null)	(null)	(null)	Production	CA
Aaron	Cooper	4/17/09	2	R&D	AZ
Lou	Donoghue	5/22/05	4	Sales	CA
Oswald	Hall	3/19/11	5	PR	UK
John	Velasquez	4/5/10	4	Sales	CA
(null)	(null)	(null)	(null)	Marketing	CA

Insert, Update and Delete

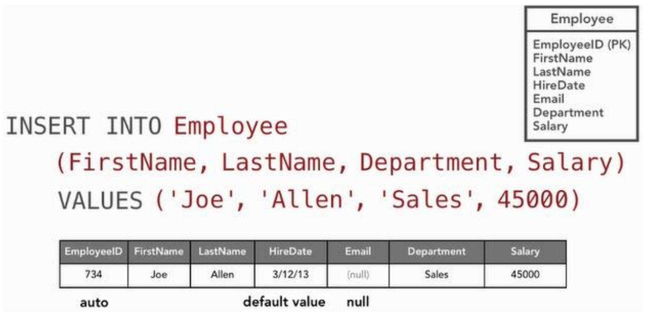


Create	INSERT
Read	SELECT
Update	UPDATE
Delete	DELETE

INSERT



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



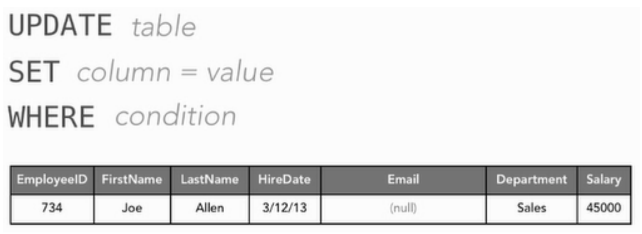
```
INSERT INTO Employee
    (FirstName, LastName, Department, Salary)
    VALUES ('Joe', 'Allen', 'Sales', 45000)
```

Employee
EmployeeID (PK) FirstName LastName HireDate Email Department Salary

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

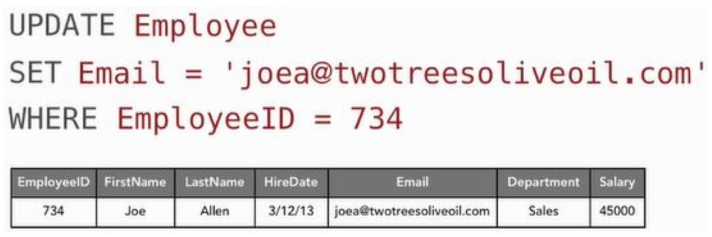
UPDATE

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

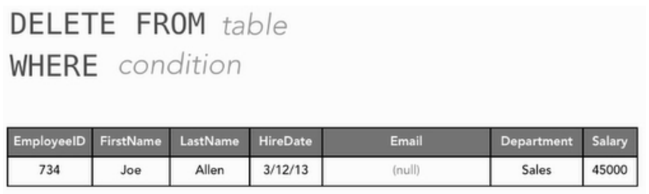


```
UPDATE Employee
SET Email = 'joea@twotreesoliveoil.com'
WHERE EmployeeID = 734
```

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

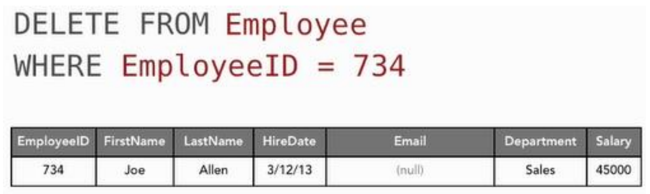
DELETE

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



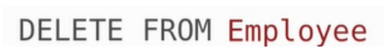
```
DELETE FROM table
WHERE condition
```

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



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

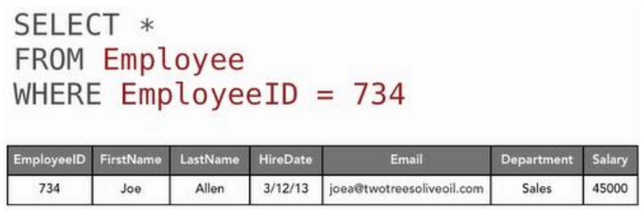
EmployeeID	FirstName	LastName	HireDate	Email	Department	Salary
734	Joe	Allen	3/12/13	(null)	Sales	45000



```
DELETE FROM Employee
```

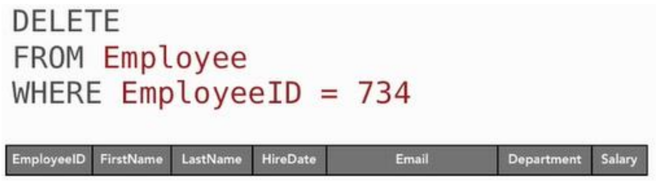
This is very dangerous because it deletes everything without warning.

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



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

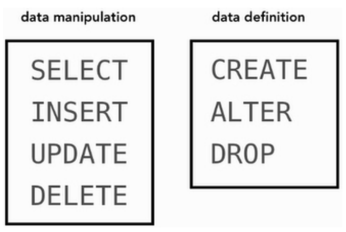
EmployeeID	FirstName	LastName	HireDate	Email	Department	Salary
734	Joe	Allen	3/12/13	joea@twotreesoliveoil.com	Sales	45000



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

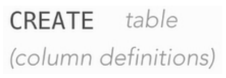
EmployeeID	FirstName	LastName	HireDate	Email	Department	Salary

Data Definition vs Manipulation

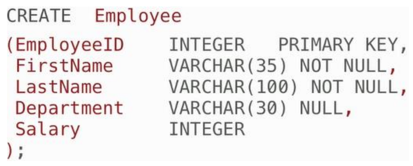

data manipulation
data definition

```
SELECT      CREATE
INSERT      ALTER
UPDATE      DROP
DELETE
```

CREATE

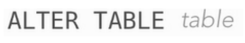


```
CREATE   table
(column definitions)
```

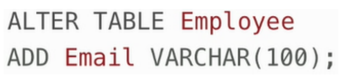


```
CREATE  Employee
(EmployeeID     INTEGER   PRIMARY KEY,
 FirstName      VARCHAR(35) NOT NULL,
 LastName       VARCHAR(100) NOT NULL,
 Department     VARCHAR(30) NULL,
 Salary         INTEGER
);
```

ALTER

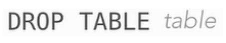


```
ALTER TABLE table
```

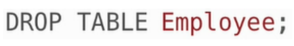


```
ALTER TABLE Employee
ADD Email VARCHAR(100);
```

DROP



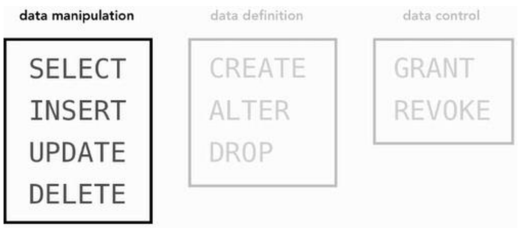
```
DROP TABLE table
```



```
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.

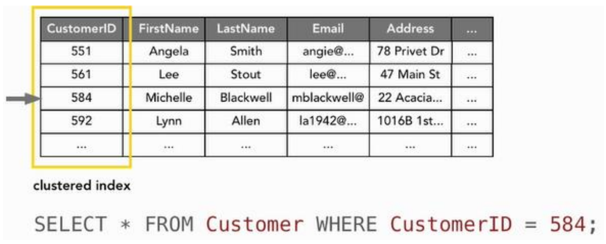

data manipulation
data definition
data control

```
SELECT      CREATE      GRANT
INSERT      ALTER       REVOKE
UPDATE      DROP
DELETE
```

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.

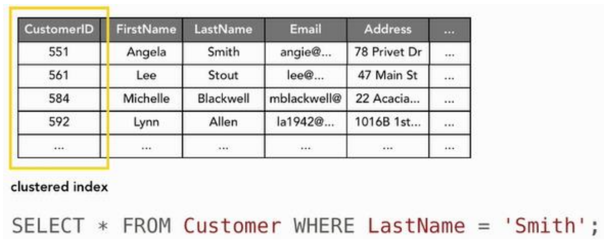


CustomerID	FirstName	LastName	Email	Address	...
551	Angela	Smith	angie@...	78 Privet Dr	...
561	Lee	Stout	lee@...	47 Main St	...
584	Michelle	Blackwell	mblackwell@	22 Acacia...	...
592	Lynn	Allen	la1942@...	1016B 1st...	...
...	...	...	...	...	...

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.

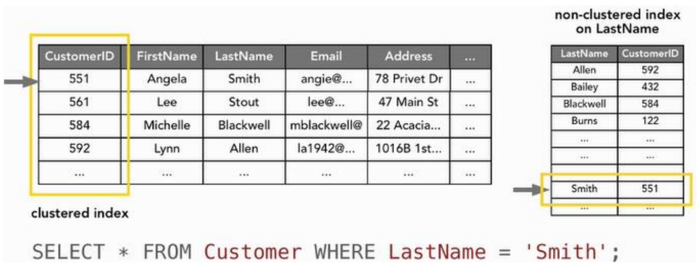


CustomerID	FirstName	LastName	Email	Address	...
551	Angela	Smith	angie@...	78 Privet Dr	...
561	Lee	Stout	lee@...	47 Main St	...
584	Michelle	Blackwell	mblackwell@	22 Acacia...	...
592	Lynn	Allen	la1942@...	1016B 1st...	...
...	...	...	...	...	...

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.



clustered index

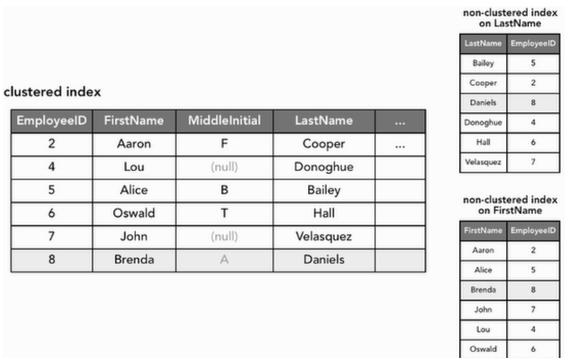
CustomerID	FirstName	LastName	Email	Address	...
551	Angela	Smith	angie@...	78 Privet Dr	...
561	Lee	Stout	lee@...	47 Main St	...
584	Michelle	Blackwell	mblackwell@	22 Acacia...	...
592	Lynn	Allen	la1942@...	1016B 1st...	...
...	...	...	...	...	...

non-clustered index on LastName

LastName	CustomerID
Allen	592
Bailey	432
Blackwell	584
Burns	122
...	...
...	...
Smith	551
...	...

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

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.



clustered index

EmployeeID	FirstName	MiddleInitial	LastName	...
2	Aaron	F	Cooper	...
4	Lou	(null)	Donoghue	
5	Alice	B	Bailey	
6	Oswald	T	Hall	
7	John	(null)	Velasquez	
8	Brenda	A	Daniels	

non-clustered index on LastName

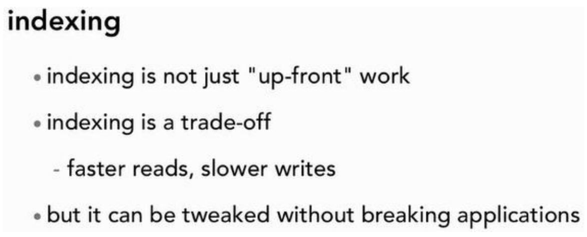
LastName	EmployeeID
Bailey	5
Cooper	2
Daniels	8
Donoghue	4
Hall	6
Velasquez	7

non-clustered index on FirstName

FirstName	EmployeeID
Aaron	2
Alice	5
Brenda	8
John	7
Lou	4
Oswald	6

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.

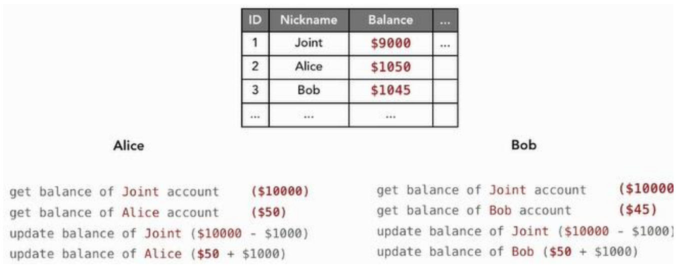
This slows the process significantly.This is why it is advised to use indexes only in high traffic columns.



indexing

- indexing is not just "up-front" work
- indexing is a trade-off
  - faster reads, slower writes
- but it can be tweaked without breaking applications

Conflict and Isolation



ID	Nickname	Balance	...
1	Joint	$9000	...
2	Alice	$1050	
3	Bob	$1045	
...	...	...	

Alice

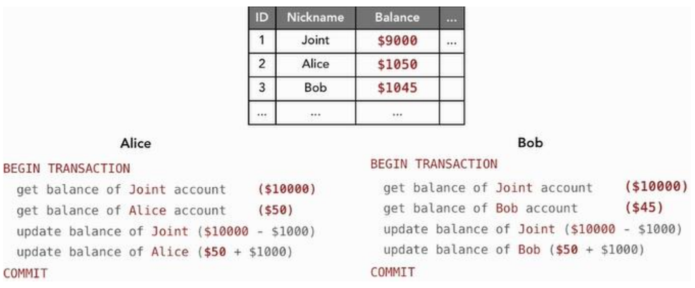
```
get balance of Joint account      ($10000)
get balance of Alice account      ($50)
update balance of Joint ($10000 - $1000)
update balance of Alice ($50 + $1000)
```

Bob

```
get balance of Joint account      ($10000)
get balance of Bob account        ($45)
update balance of Joint ($10000 - $1000)
update balance of Bob ($50 + $1000)
```

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.



ID	Nickname	Balance	...
1	Joint	$9000	...
2	Alice	$1050	
3	Bob	$1045	
...	...	...	

Alice

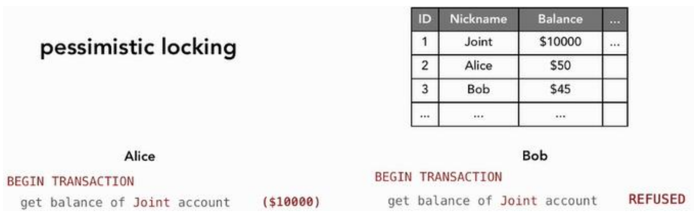
```
BEGIN TRANSACTION
  get balance of Joint account      ($10000)
  get balance of Alice account      ($50)
  update balance of Joint ($10000 - $1000)
  update balance of Alice ($50 + $1000)
COMMIT
```

Bob

```
BEGIN TRANSACTION
  get balance of Joint account      ($10000)
  get balance of Bob account        ($45)
  update balance of Joint ($10000 - $1000)
  update balance of Bob ($50 + $1000)
COMMIT
```

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.



pessimistic locking

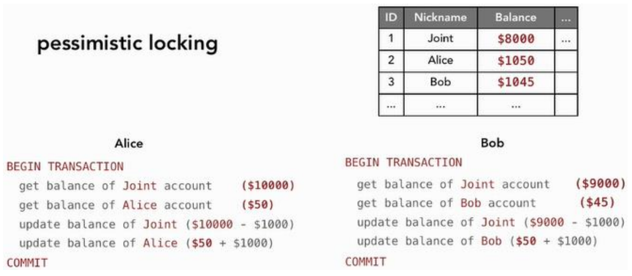
ID	Nickname	Balance	...
1	Joint	$10000	...
2	Alice	$50	
3	Bob	$45	
...	...	...	

Alice

```
BEGIN TRANSACTION
  get balance of Joint account      ($10000)
```

Bob

```
BEGIN TRANSACTION
  get balance of Joint account      REFUSED
```



pessimistic locking

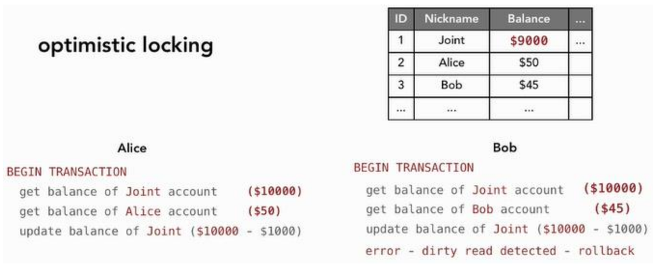
ID	Nickname	Balance	...
1	Joint	$8000	...
2	Alice	$1050	
3	Bob	$1045	
...	...	...	

Alice

```
BEGIN TRANSACTION
  get balance of Joint account      ($10000)
  get balance of Alice account      ($50)
  update balance of Joint ($10000 - $1000)
  update balance of Alice ($50 + $1000)
COMMIT
```

Bob

```
BEGIN TRANSACTION
  get balance of Joint account      ($9000)
  get balance of Bob account        ($45)
  update balance of Joint ($9000 - $1000)
  update balance of Bob ($50 + $1000)
COMMIT
```


optimistic locking

ID	Nickname	Balance	...
1	Joint	$9000	...
2	Alice	$50	
3	Bob	$45	
...	...	...	

Alice

```
BEGIN TRANSACTION
  get balance of Joint account     ($10000)
  get balance of Alice account     ($50)
  update balance of Joint ($10000 - $1000)
```

Bob

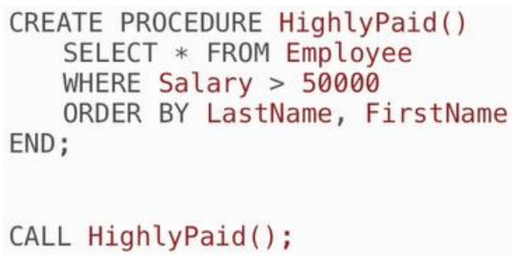
```
BEGIN TRANSACTION
  get balance of Joint account   ($10000)
  get balance of Bob account      ($45)
  update balance of Joint ($10000 - $1000)
  error - dirty read detected - rollback
```

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.

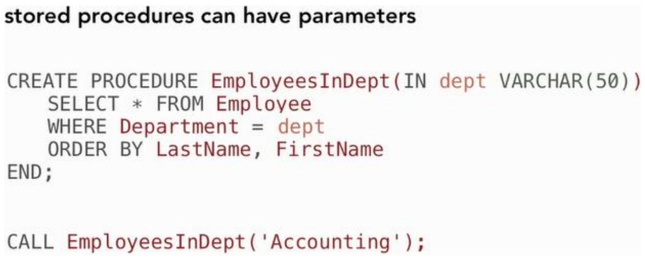
When this happens, a rollback occurs which goes at the start of the transaction.

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.

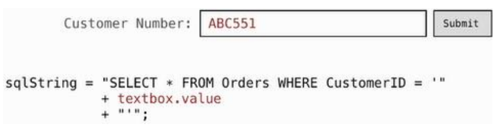


```
CREATE PROCEDURE HighlyPaid()
    SELECT * FROM Employee
    WHERE Salary > 50000
    ORDER BY LastName, FirstName
END;


CALL HighlyPaid();
```

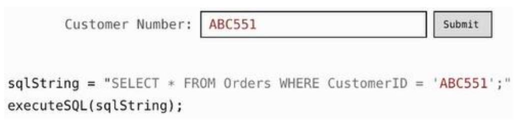

stored procedures can have parameters

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


CALL EmployeesInDept('Accounting');
```

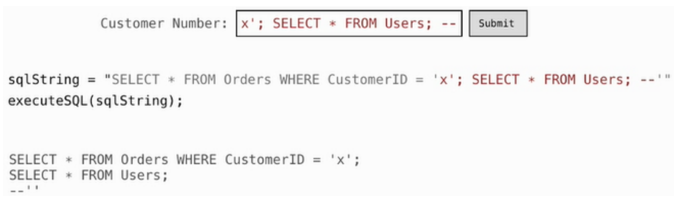
Stored procedures with parameters are a great way for preventing SQL injection attacks. This is true because stored procedures are inflexible and cannot be divided into separate statements.



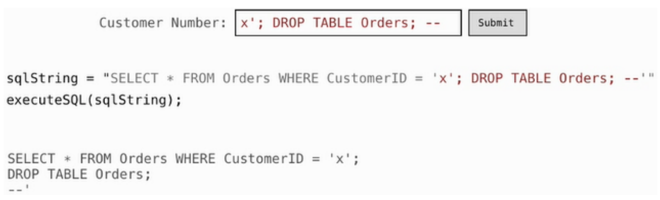
```
sqlString = "SELECT * FROM Orders WHERE CustomerID = '"
            + textbox.value
            + "'";
```



```
sqlString = "SELECT * FROM Orders WHERE CustomerID = 'ABC551';"
executeSQL(sqlString);
```



```
sqlString = "SELECT * FROM Orders WHERE CustomerID = 'x'; SELECT * FROM Users; --'"
executeSQL(sqlString);


SELECT * FROM Orders WHERE CustomerID = 'x';
SELECT * FROM Users;
--''
```



```
sqlString = "SELECT * FROM Orders WHERE CustomerID = 'x'; DROP TABLE Orders; --'"
executeSQL(sqlString);


SELECT * FROM Orders WHERE CustomerID = 'x';
DROP TABLE Orders;
--'
```

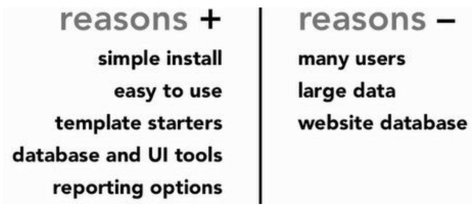
The first statements returns no value. The second one is the injection. The third one acts as a comment due to the – before the ‘.

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…

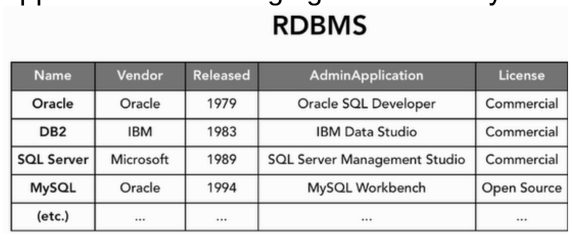


reasons +	reasons –
simple install	many users
easy to use	large data
template starters	website database
database and UI tools	
reporting options	

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

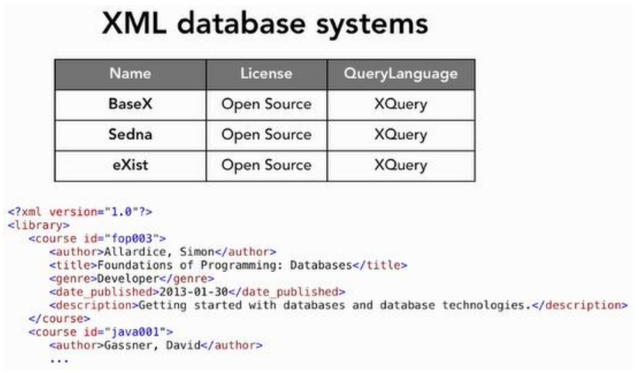
Name	Vendor	Released	AdminApplication	License
Oracle	Oracle	1979	Oracle SQL Developer	Commercial
DB2	IBM	1983	IBM Data Studio	Commercial
SQL Server	Microsoft	1989	SQL Server Management Studio	Commercial
MySQL	Oracle	1994	MySQL Workbench	Open Source
(etc.)	...	...	...	...

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.

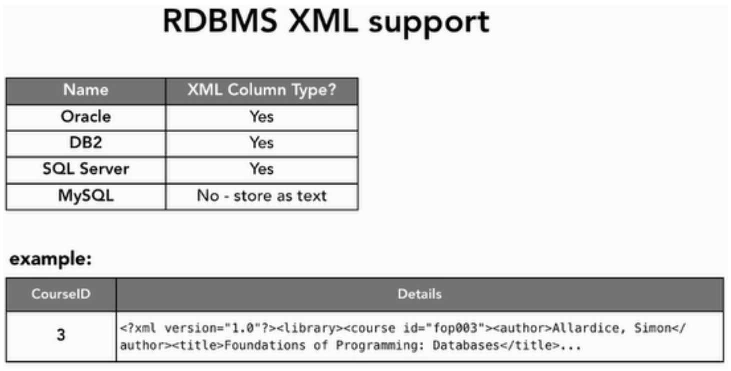
XML



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

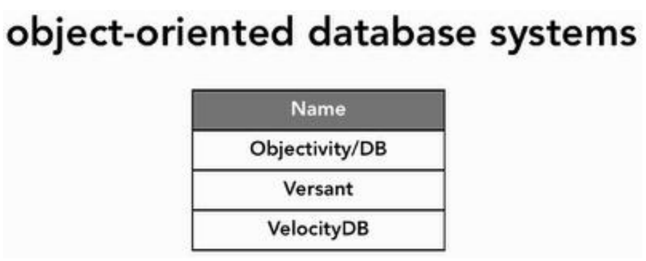
Name	XML Column Type?
Oracle	Yes
DB2	Yes
SQL Server	Yes
MySQL	No - store as text

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.

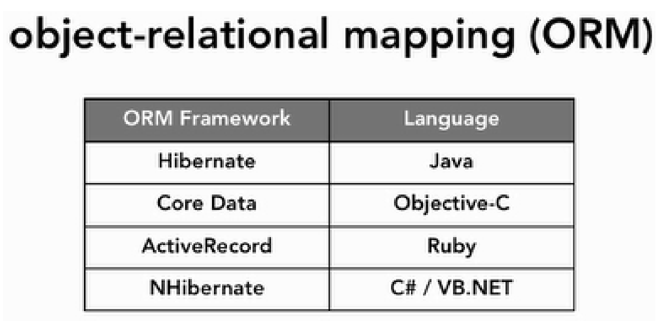
Object Oriented Database Systems



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.



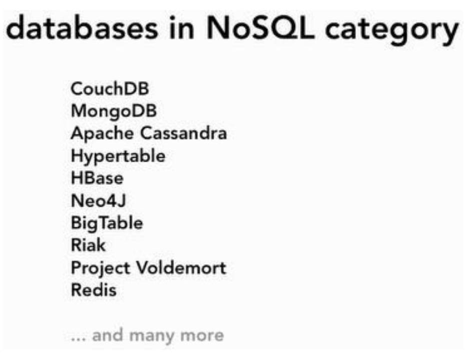
object-relational mapping (ORM)

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

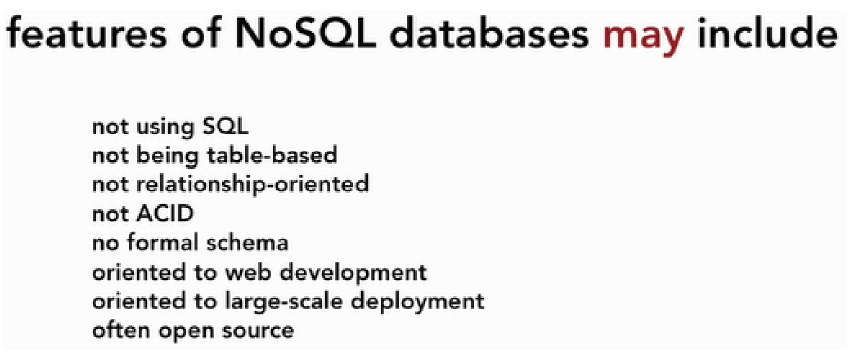
Means “Not only SQL”. These are not the new best way, but rather a solution to a new set of problems. These are commonly used for big data where hundreds of millions and billions of data is common. Also, these are useful when losing data is not detrimental. Ex. Click analytics vs Bank transfers. The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



databases in NoSQL category

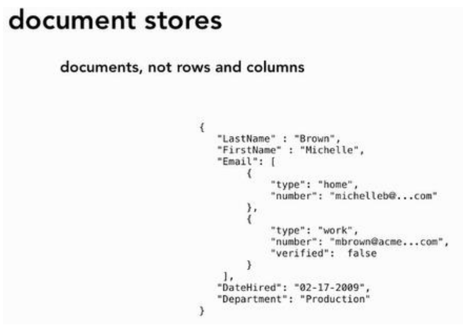
CouchDB
MongoDB
Apache Cassandra
Hypertable
HBase
Neo4J
BigTable
Riak
Project Voldemort
Redis

... and many more



features of NoSQL databases may include

not using SQL
not being table-based
not relationship-oriented
not ACID
no formal schema
oriented to web development
oriented to large-scale deployment
often open source

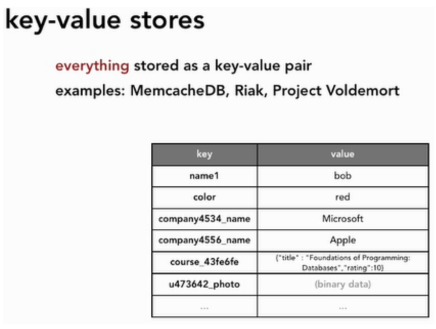


document stores

documents, not rows and columns

```
{
   "LastName" : "Brown",
   "FirstName" : "Michelle",
   "Email": [
         {
             "type": "home",
             "number": "michelleb@...com"
         },
         {
             "type": "work",
             "number": "mbrown@acme...com",
             "verified":  false
         }
    ],
   "DateHired": "02-17-2009",
   "Department": "Production"
}
```

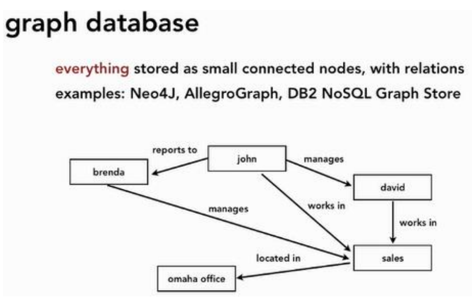
CouchDB, MongoDB



key-value stores

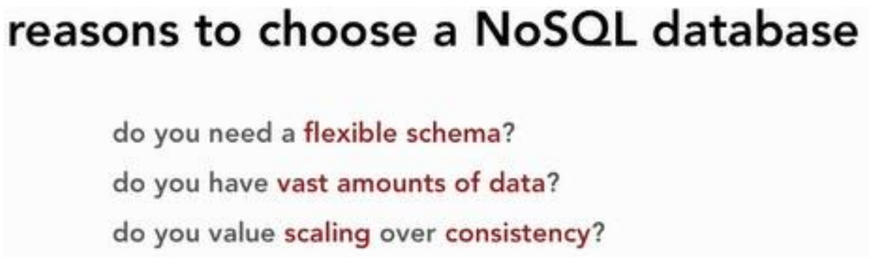
everything stored as a key-value pair
examples: MemcacheDB, Riak, Project Voldemort

key	value
name1	bob
color	red
company4534_name	Microsoft
company4556_name	Apple
course_43fe6fe	{"title" : "Foundations of Programming: Databases","rating":10}
u473642_photo	(binary data)
...	...



graph database

everything stored as small connected nodes, with relations
examples: Neo4J, AllegroGraph, DB2 NoSQL Graph Store



reasons to choose a NoSQL database

do you need a flexible schema?
do you have vast amounts of data?
do you value scaling over consistency?

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;
```

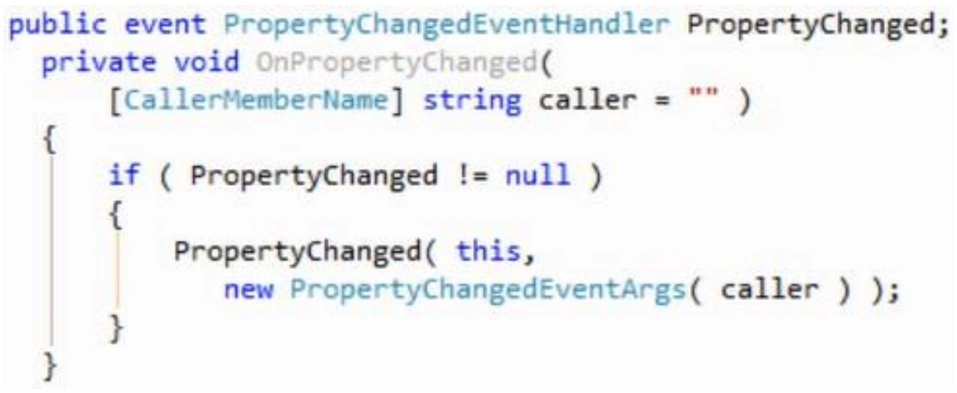
```
// MainWindow.xaml
<TextBlock Text="{Binding name}"/>
```

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:

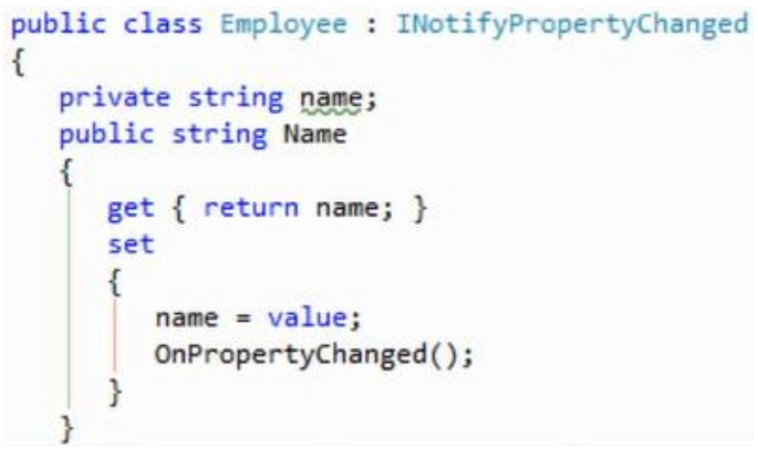
public event PropertyChangedEventHandler propertyChanged;

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.

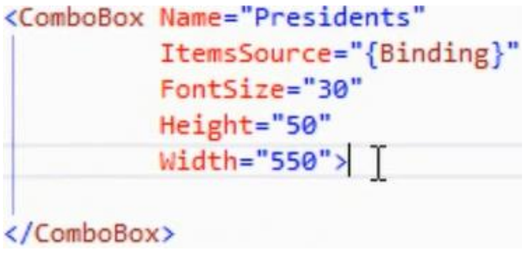
It is activated by adding a mode with the selection TwoWay.

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

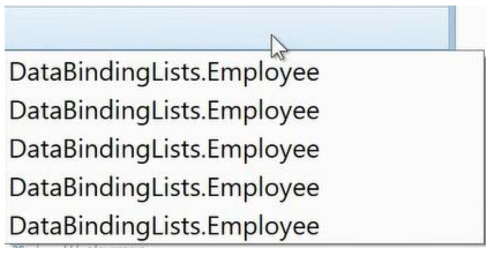
Data Binding Lists

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.

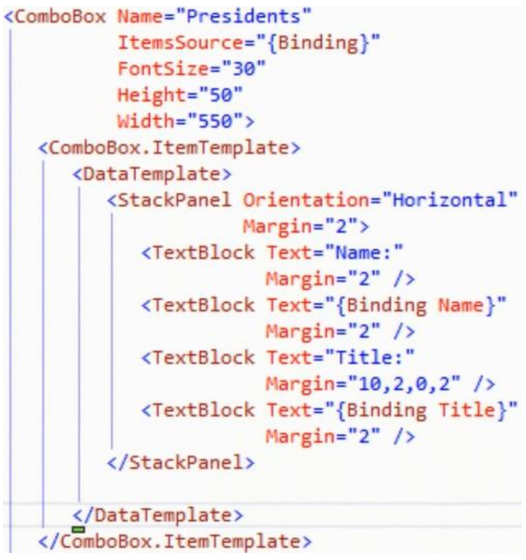


```
<ComboBox Name="Presidents"
          ItemsSource="{Binding}"
          FontSize="30"
          Height="50"
          Width="550">

</ComboBox>
```

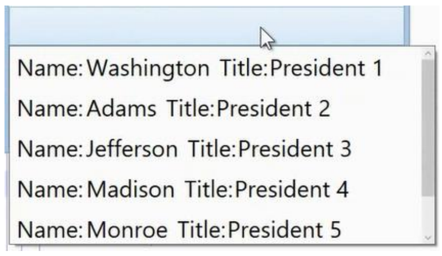


```
DataBindingLists.Employee
DataBindingLists.Employee
DataBindingLists.Employee
DataBindingLists.Employee
DataBindingLists.Employee
```

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.

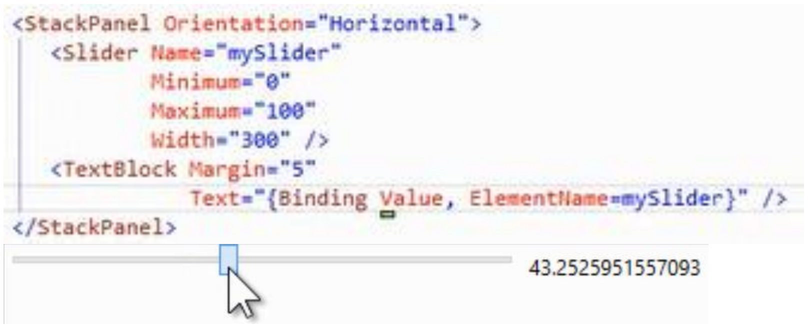


```
<ComboBox Name="Presidents"
          ItemsSource="{Binding}"
          FontSize="30"
          Height="50"
          Width="550">
   <ComboBox.ItemTemplate>
      <DataTemplate>
         <StackPanel Orientation="Horizontal"
                     Margin="2">
            <TextBlock Text="Name:"
  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



```
Name: Washington  Title: President 1
Name: Adams  Title: President 2
Name: Jefferson  Title: President 3
Name: Madison  Title: President 4
Name: Monroe  Title: President 5
```

Data Binding Elements



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

```
43.2525951557093
```

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:

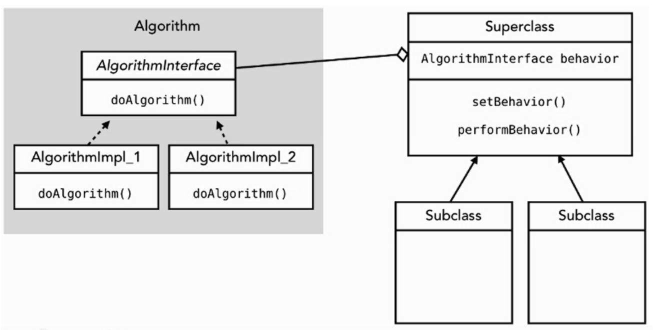
"First, register your object with mine, then implement an update method, then when it gets called, call my getValue method."

VS

"Just use the Observer Pattern"

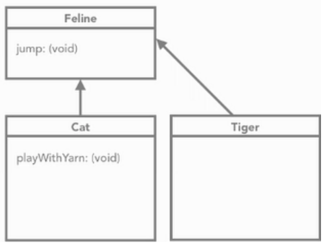
Strategy Pattern

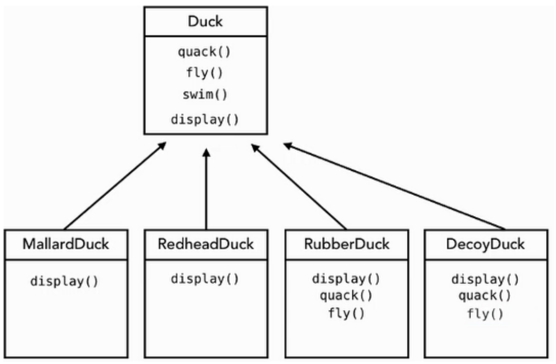
It is the combination of composition (interfaces) for behaviours that need more flexibility and inheritance for behaviours that don't need to change.



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*



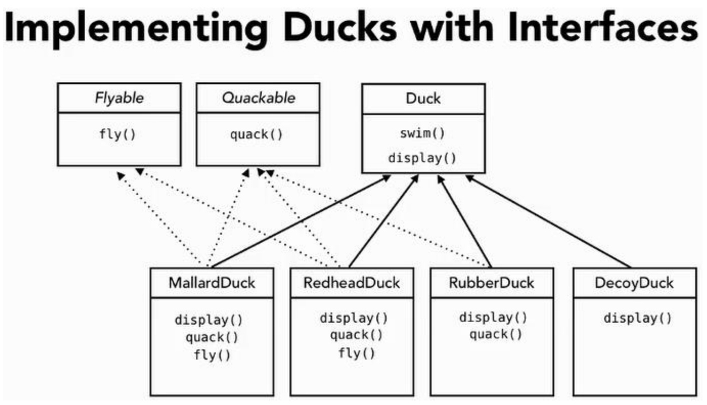


Problems with Our Design

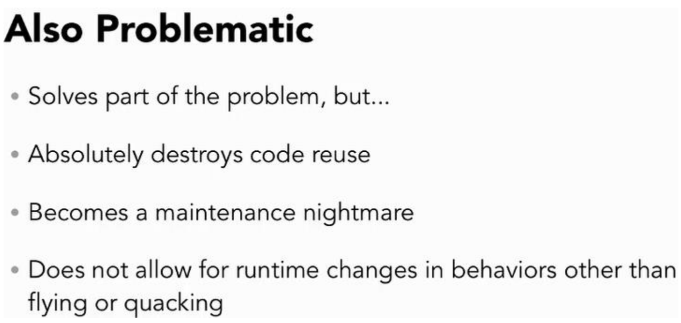
- We have code duplicated across classes
- Hard to gain knowledge of all the ducks
- Changes can affect other ducks
- Runtime behavior changes are difficult

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

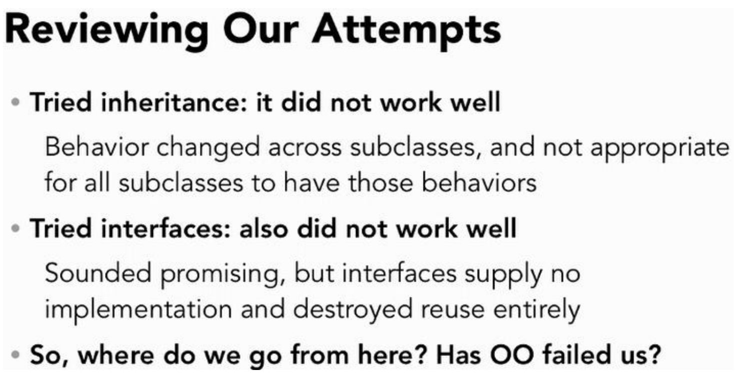


Implementing Ducks with Interfaces



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

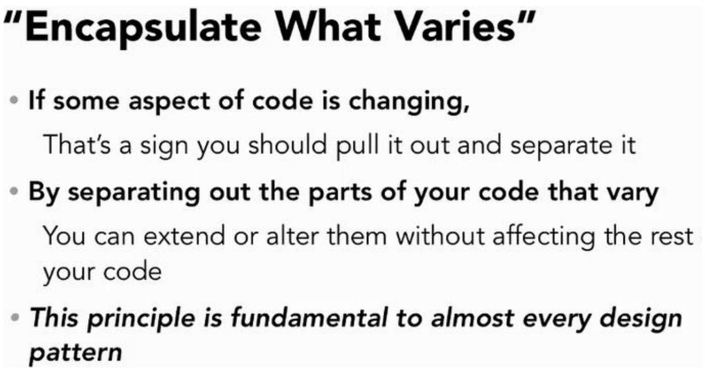
- **Tried inheritance: it did not work well**

  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.

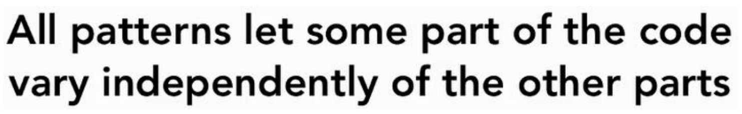


"Encapsulate What Varies"

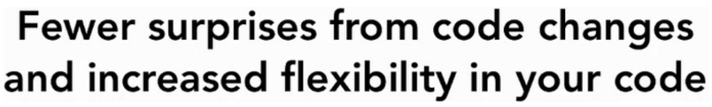
- **If some aspect of code is changing,**

  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***

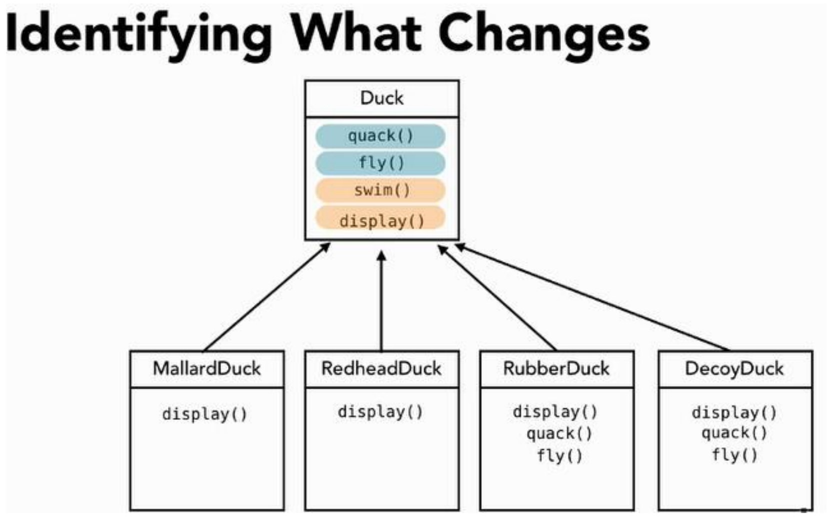


All patterns let some part of the code vary independently of the other parts



Fewer surprises from code changes and increased flexibility in your code

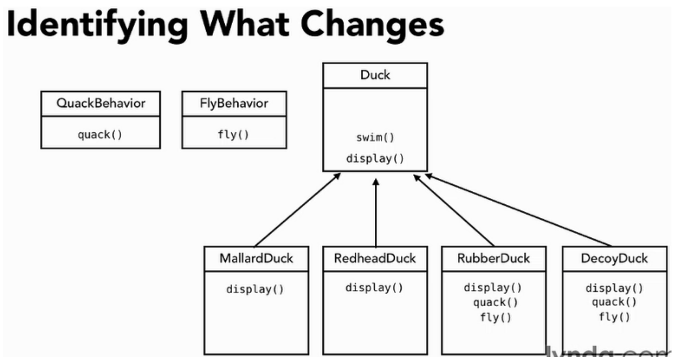
Change is the only constant in software desing.

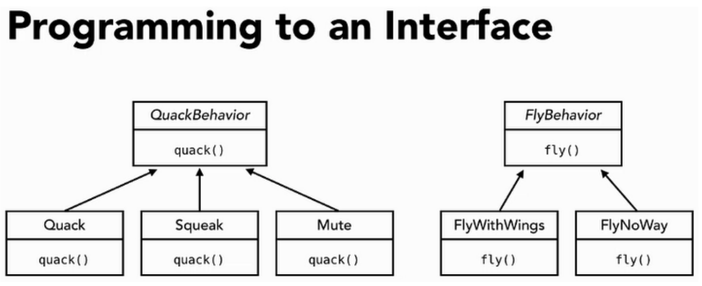


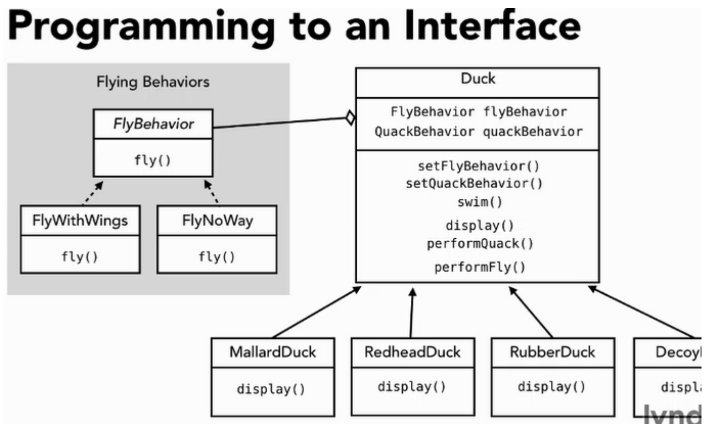
Identifying What Changes

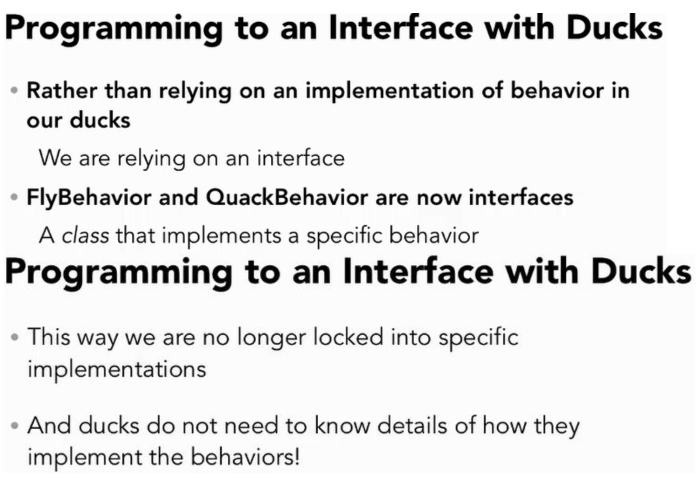
Design Principle #2

Program to an interface, not an implementation








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

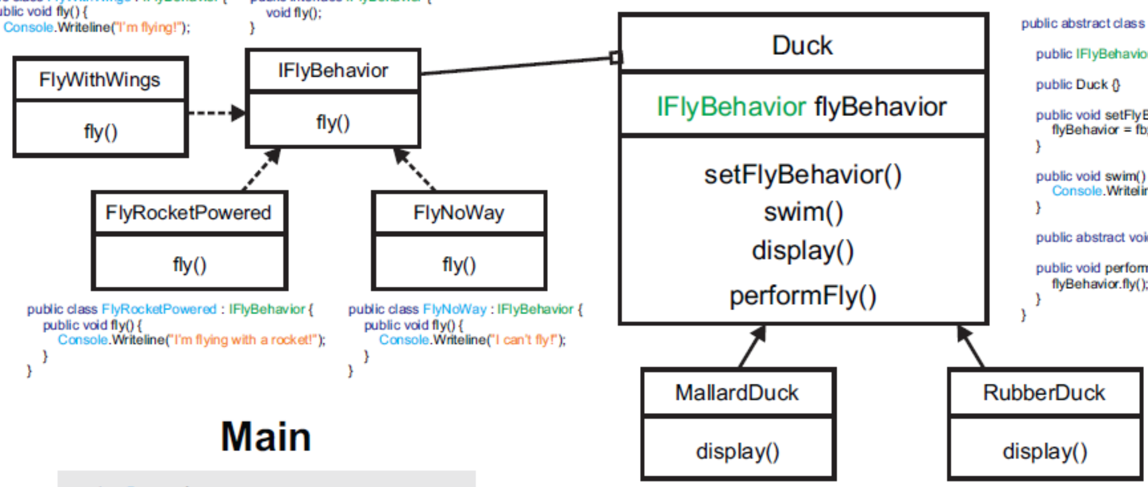
- This way we are no longer locked into specific implementations

- 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 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 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 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 MallarDuck();
    RubberDuck rubberDuckie = new RubberDuck();

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

    }
}
```

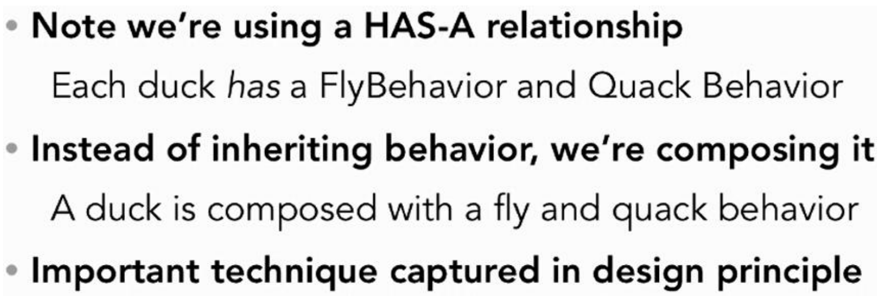
Output

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

Design Principle #3

Favor composition over inheritance.

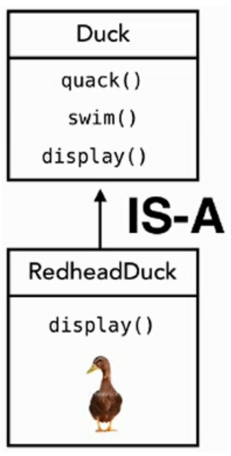
Inheritance = IS A relationship, Composition = HAS A relationship.



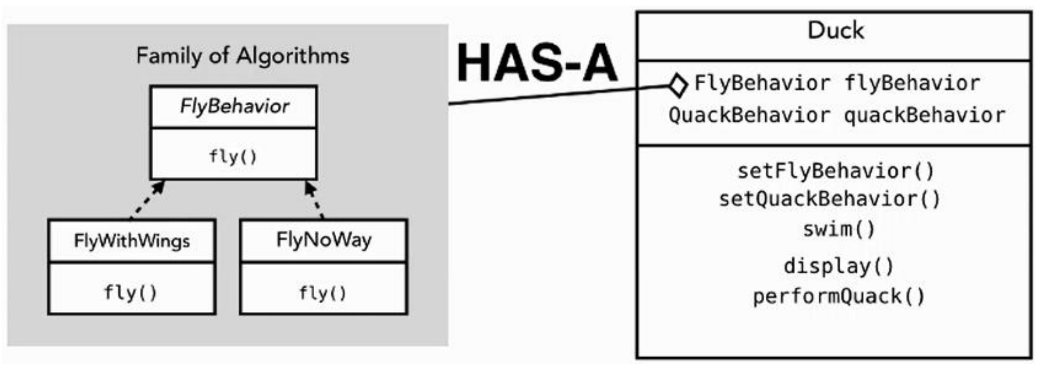
- **Note we're using a HAS-A relationship**

  Each duck *has* a FlyBehavior and Quack Behavior
- **Instead of inheriting behavior, we're composing it**

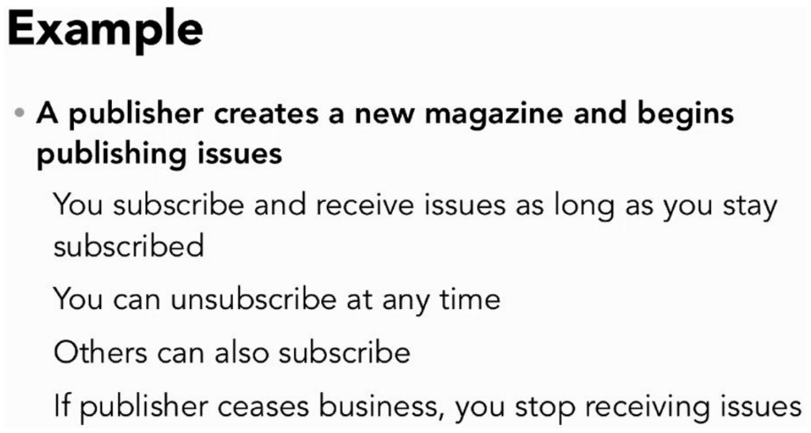
  A duck is composed with a fly and quack behavior
- **Important technique captured in design principle**



VS

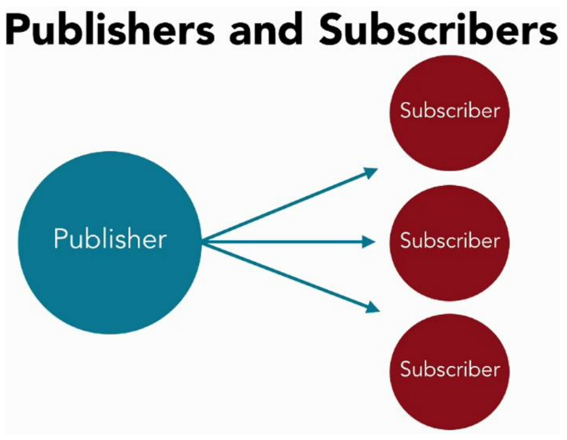


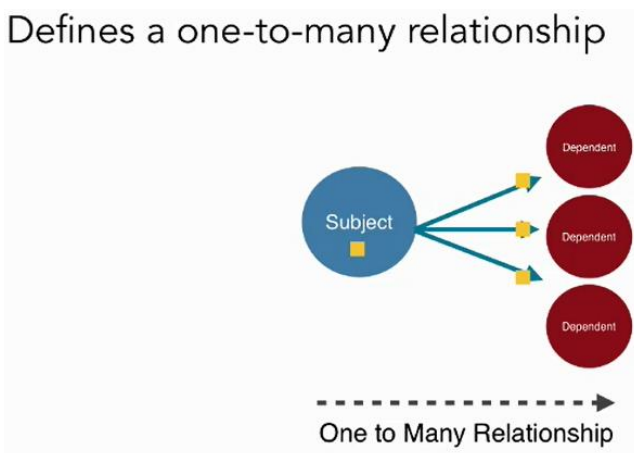
Observer Patter

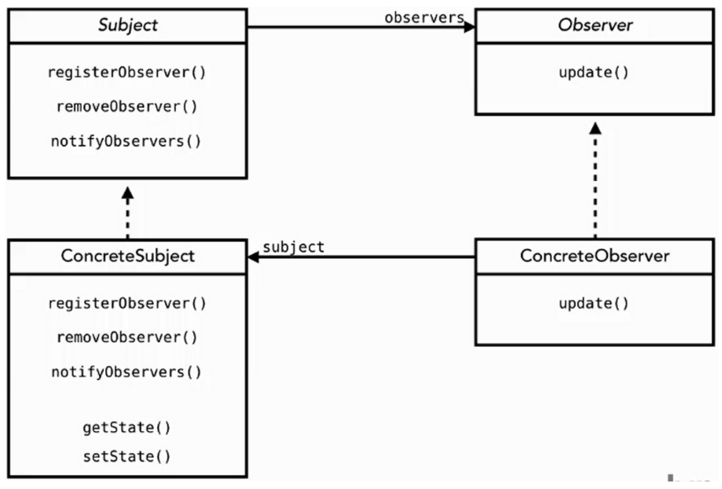


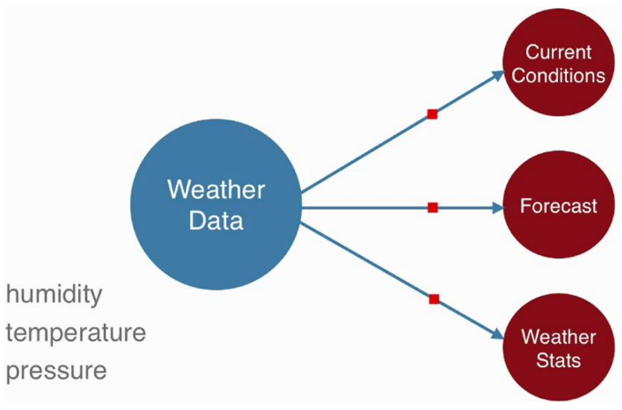
Example

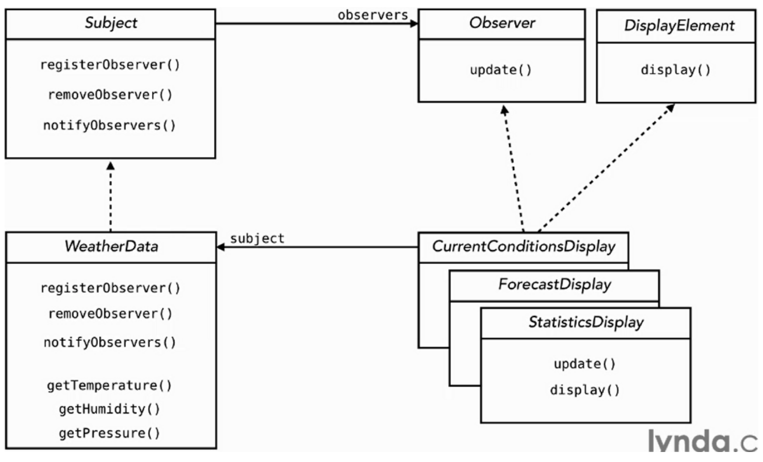
- **A publisher creates a new magazine and begins publishing issues**
  - You subscribe and receive issues as long as you stay subscribed
  - You can unsubscribe at any time
  - Others can also subscribe
  - If publisher ceases business, you stop receiving issues











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.

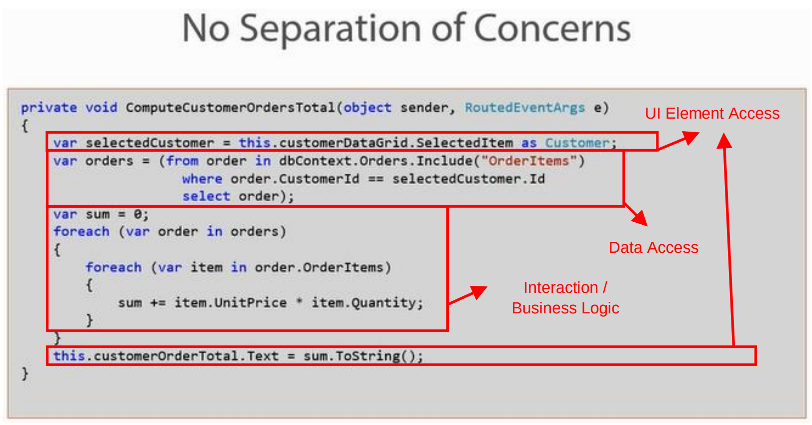
Check out “Developing Extensible Software” by Miguel Castro, and “Extending XAML Applications with Behaviors” by Brian Noyes.

MVVM Fundamentals

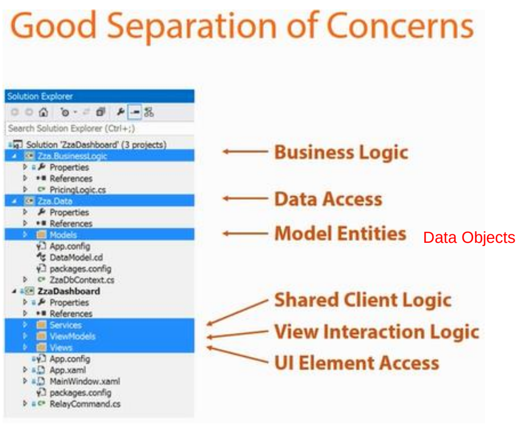
Separation of Concerns

MVVM is mostly about trying to achieve good separation of concerns.

Good Separation of Concerns
A bit more work to put things away
Makes getting dressed easy



```
private void ComputeCustomerOrdersTotal(object sender, RoutedEventArgs e)
{
    var selectedCustomer = this.customerDataGrid.SelectedItem as Customer;
    var orders = (from order in dbContext.Orders.Include("OrderItems")
                  where order.CustomerId == selectedCustomer.Id
                  select order);
    var sum = 0;
    foreach (var order in orders)
    {
        foreach (var item in order.OrderItems)
        {
            sum += item.UnitPrice * item.Quantity;
        }
    }
    this.customerOrderTotal.Text = sum.ToString();
}
```



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.

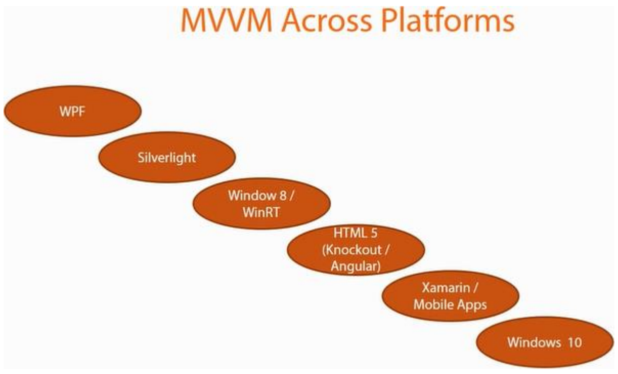
MVP (Model View Presenter)

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.

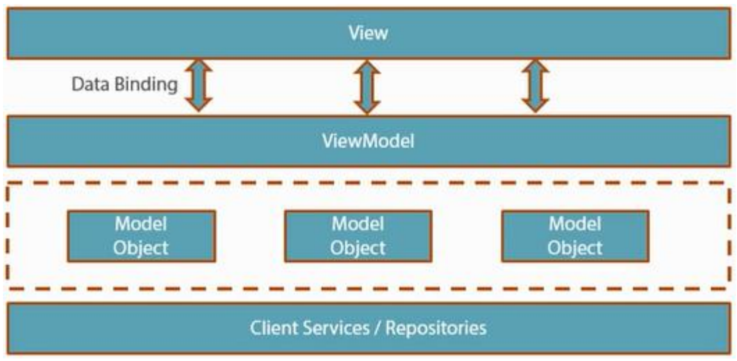
MVVM (Model View ViewModel)

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**.

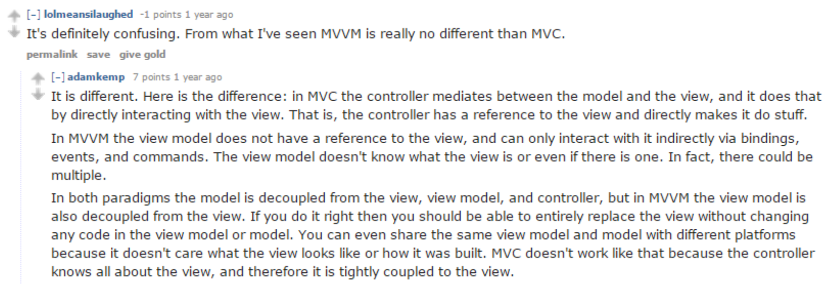
View Responsibilities

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.

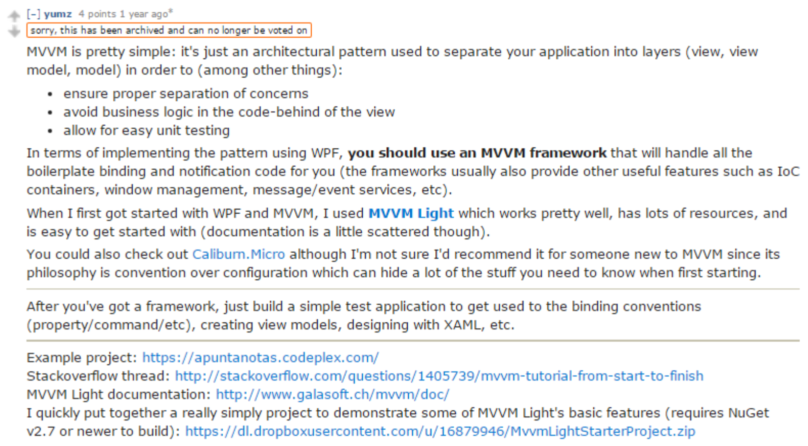
permalink save give gold

[–] **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.

.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

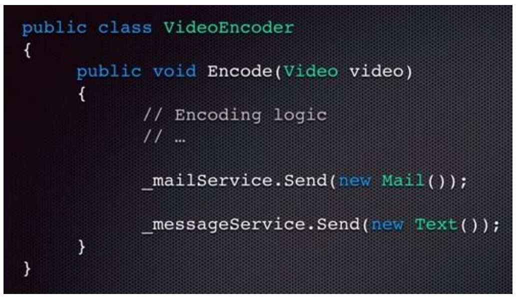
Bonus: Async / Await

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.



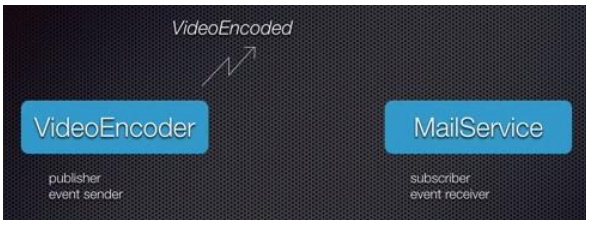
```
public class VideoEncoder
{
        public void Encode(Video video)
        {
                // Encoding logic
                // …

                _mailService.Send(new Mail());

                _messageService.Send(new Text());
        }
}
```

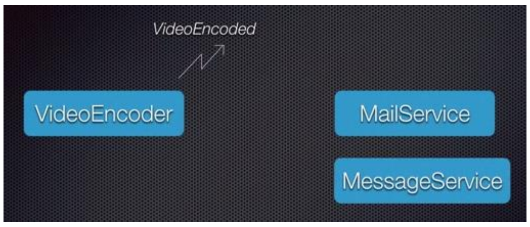
This is a video encoding class with a method that encodes a passed in video.

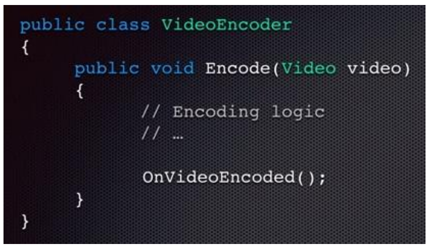
After it does the encoding, it sends an email and text to the owner of the video.

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.





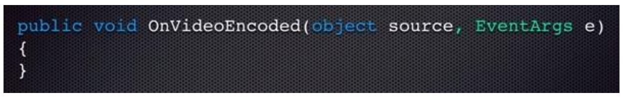


```
public class VideoEncoder
{
        public void Encode(Video video)
        {
                // Encoding logic
                // …

                OnVideoEncoded();
        }
}
```

OnVideoEncoded() notifies the subscribers.

Any amount of subscribers as well as notification services can be added in this class without changing any code.

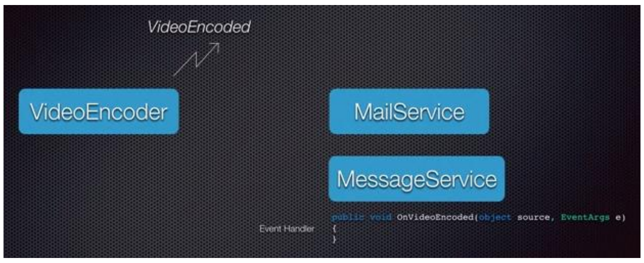
How does the VideoEncoder notify the subscribers? By **invoking a method (EventHandler) in the subscriber**. By how does it know which method to invoke? This is done **via a contract (a method with a specific signature called a Delegate) between publishers and subscribers**.



```
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.



```
public void OnVideoEncoded(object source, EventArgs e)
{
}
```

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.

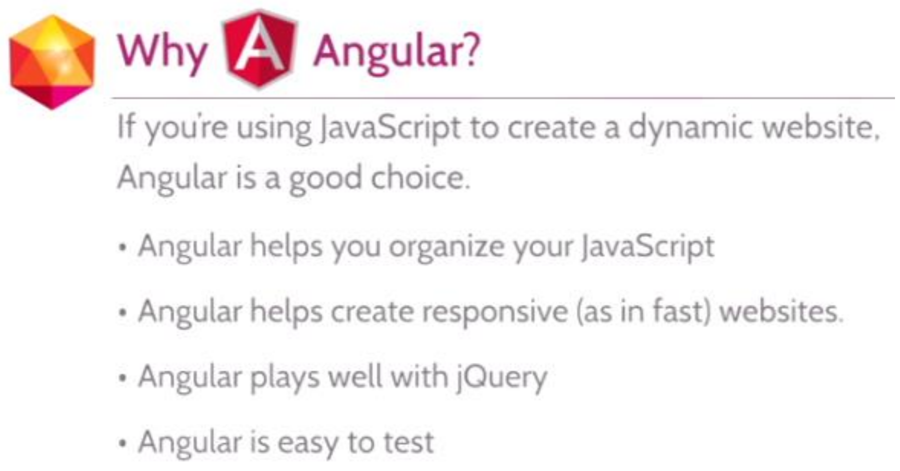
Angular.js

Overview

Code School Course

Testing

BDD = Behavior Driven Development
TDD = Test Driven Development

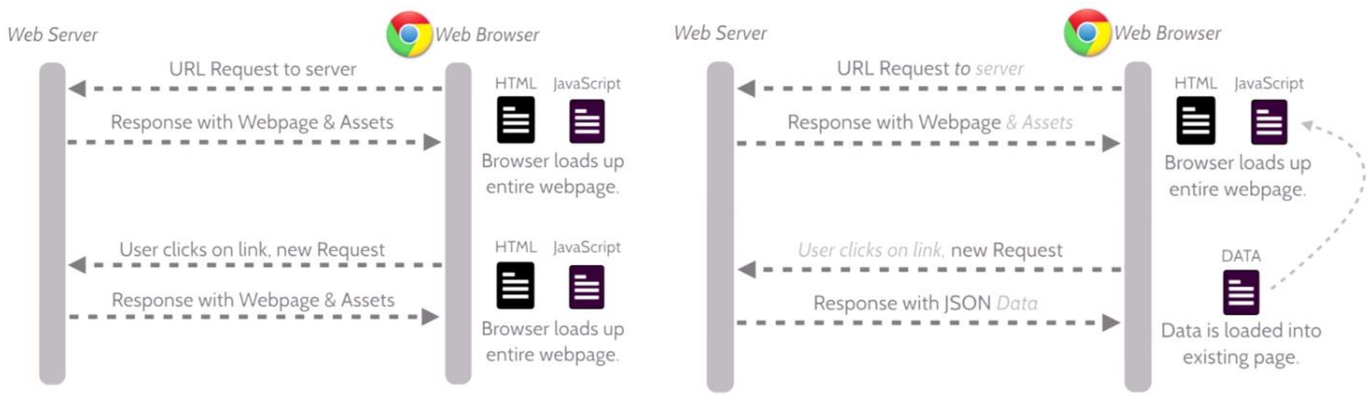


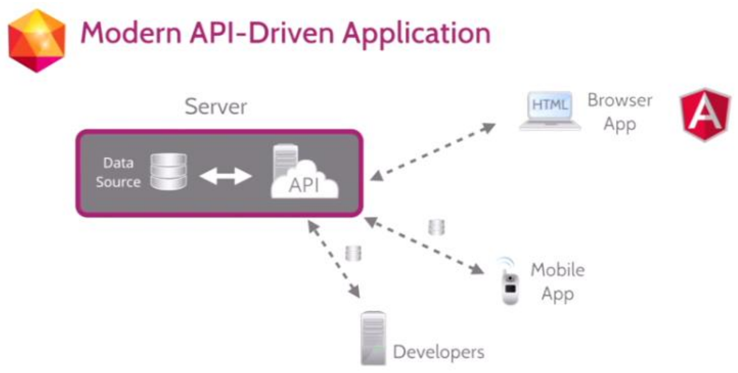
Why Angular?

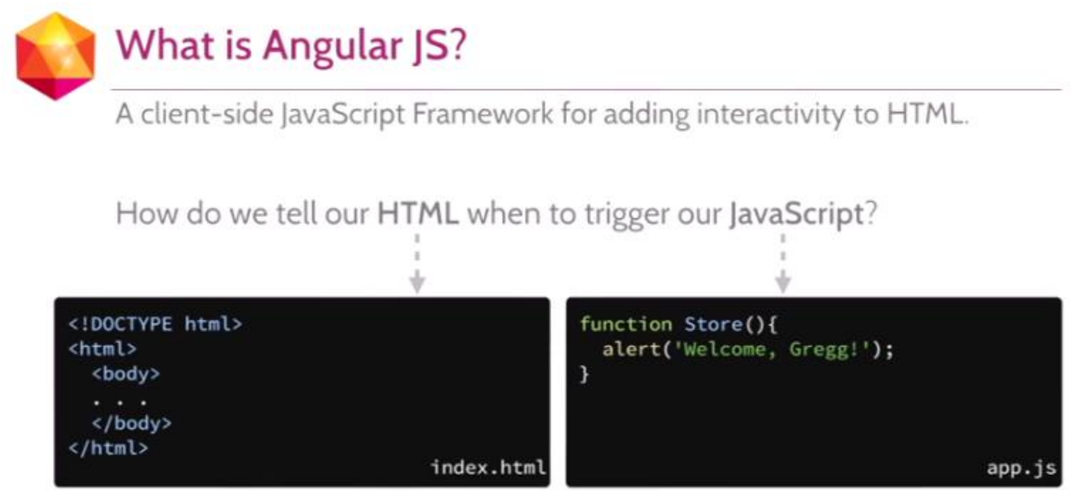
If you're using JavaScript to create a dynamic website, Angular is a good choice.

- Angular helps you organize your JavaScript
- Angular helps create responsive (as in fast) websites.
- Angular plays well with jQuery
- Angular is easy to test

Traditional vs Responsive website


Web Server
Web Browser
URL Request to server
Response with Webpage & Assets
HTML
JavaScript
Browser loads up entire webpage.
User clicks on link, new Request
Response with Webpage & Assets
HTML
JavaScript
Browser loads up entire webpage.
Web Server
Web Browser
URL Request to server
Response with Webpage & Assets
HTML
JavaScript
Browser loads up entire webpage.
User clicks on link, new Request
Response with JSON Data
DATA
Data is loaded into existing page.





What is Angular JS?

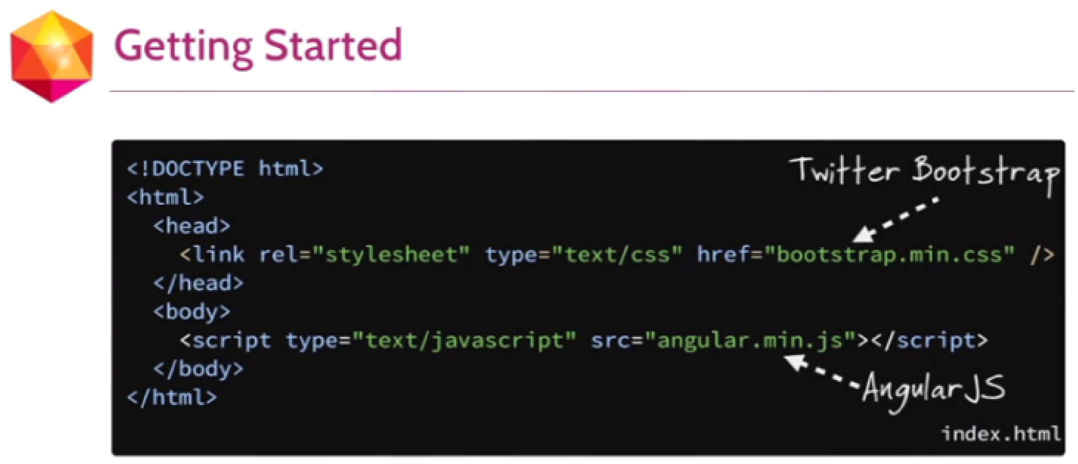
A client-side JavaScript Framework for adding interactivity to HTML.

How do we tell our **HTML** when to trigger our **JavaScript**?

```
<!DOCTYPE html>
<html>
  <body>
   . . .
  </body>
</html>
```
index.html

```
function Store(){
  alert('Welcome, Gregg!');
}
```
app.js

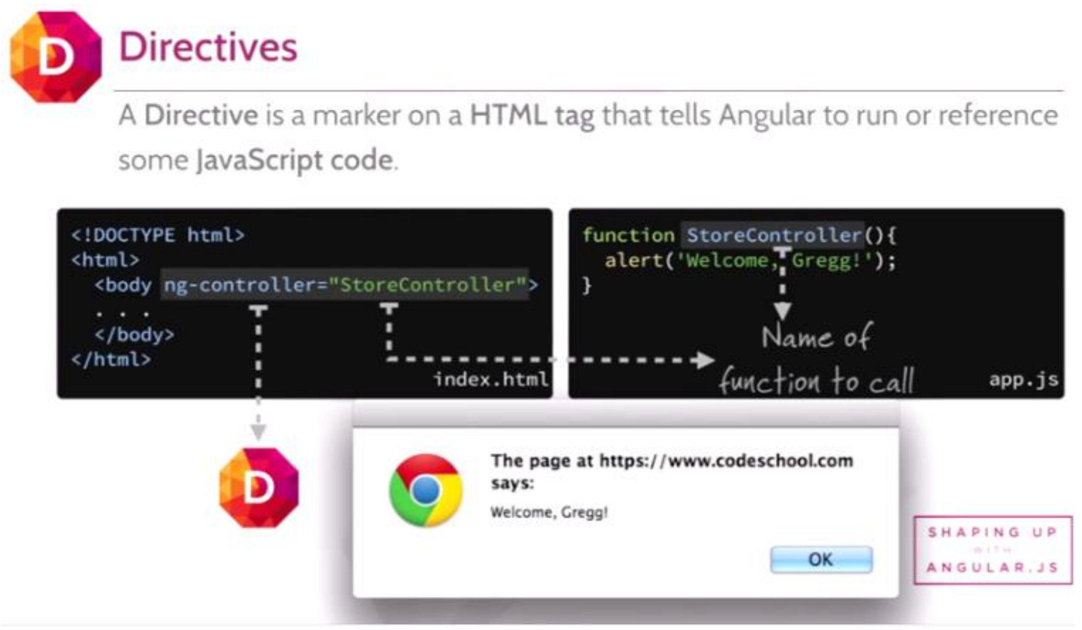
Installing



Getting Started

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="bootstrap.min.css" />
  </head>
  <body>
    <script type="text/javascript" src="angular.min.js"></script>
  </body>
</html>
```
index.html

Directives



Directives

A **Directive** is a marker on a **HTML tag** that tells Angular to run or reference some **JavaScript code**.

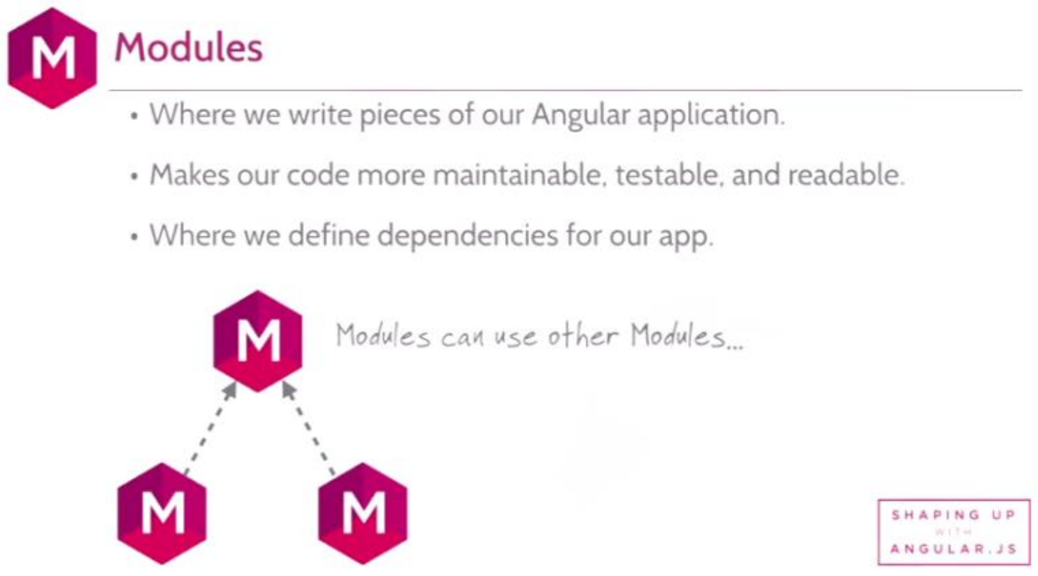
```
<!DOCTYPE html>
<html>
  <body ng-controller="StoreController">
   . . .
  </body>
</html>
```
index.html

```
function StoreController(){
  alert('Welcome, Gregg!');
}
```
app.js

The page at https://www.codeschool.com says:

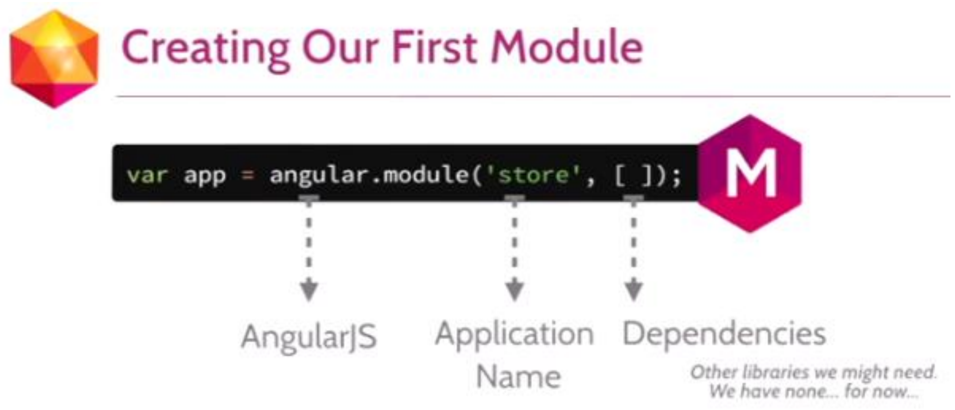
Welcome, Gregg!

Modules



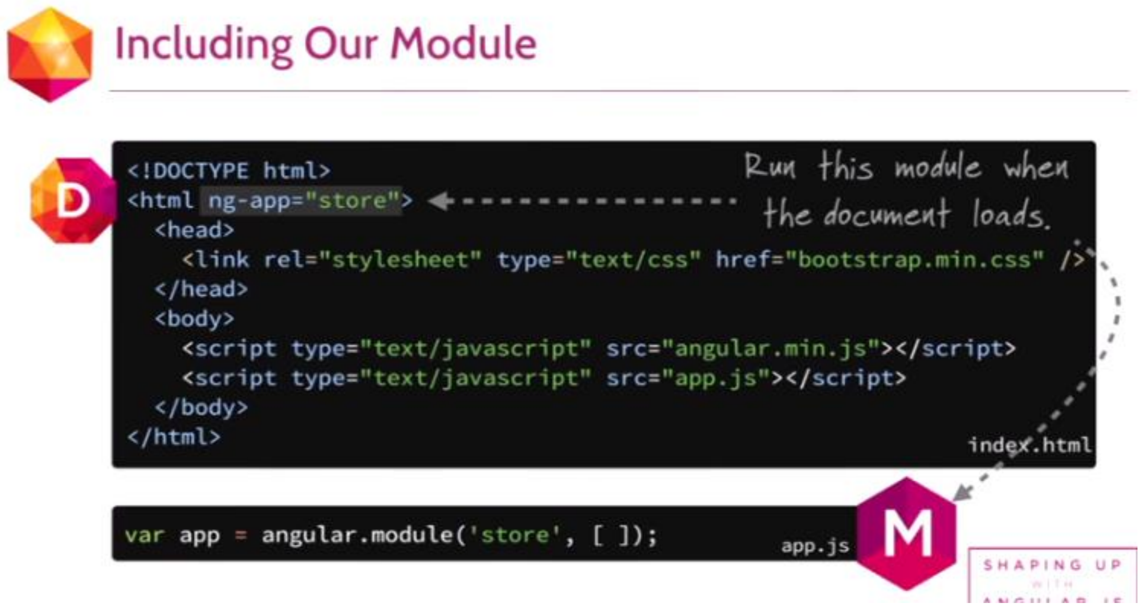
Modules

- Where we write pieces of our Angular application.
- Makes our code more maintainable, testable, and readable.
- Where we define dependencies for our app.



Creating Our First Module

```
var app = angular.module('store', [ ]);
```



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>
```

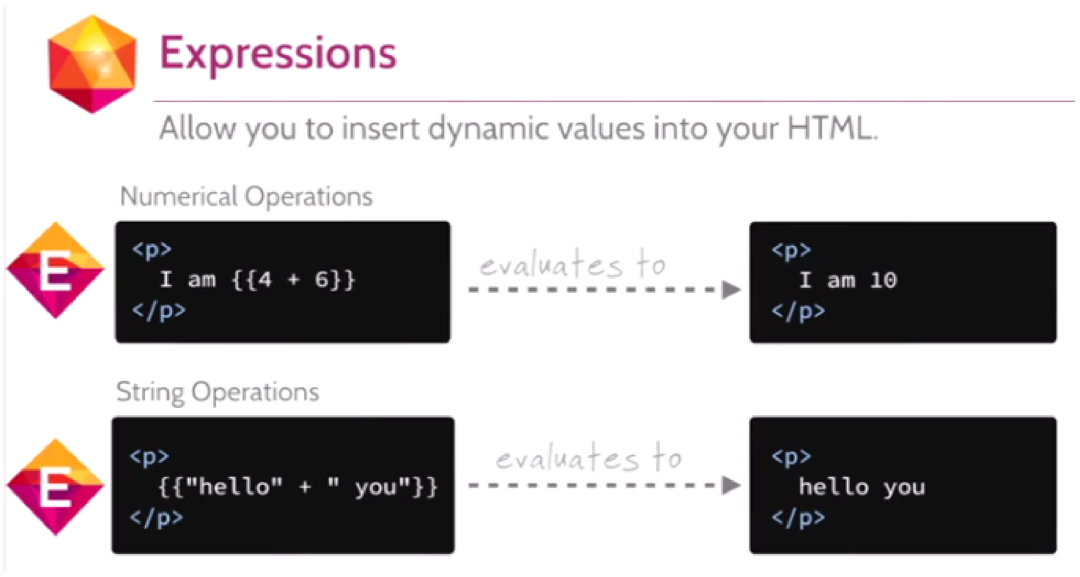
index.html

```
var app = angular.module('store', [ ]);
```

app.js

D
Run this module when the document loads.
M
SHAPING UP WITH ANGULAR.JS

Expressions



Expressions

Allow you to insert dynamic values into your HTML.

Numerical Operations

```
<p>
  I am {{4 + 6}}
</p>
```

evaluates to

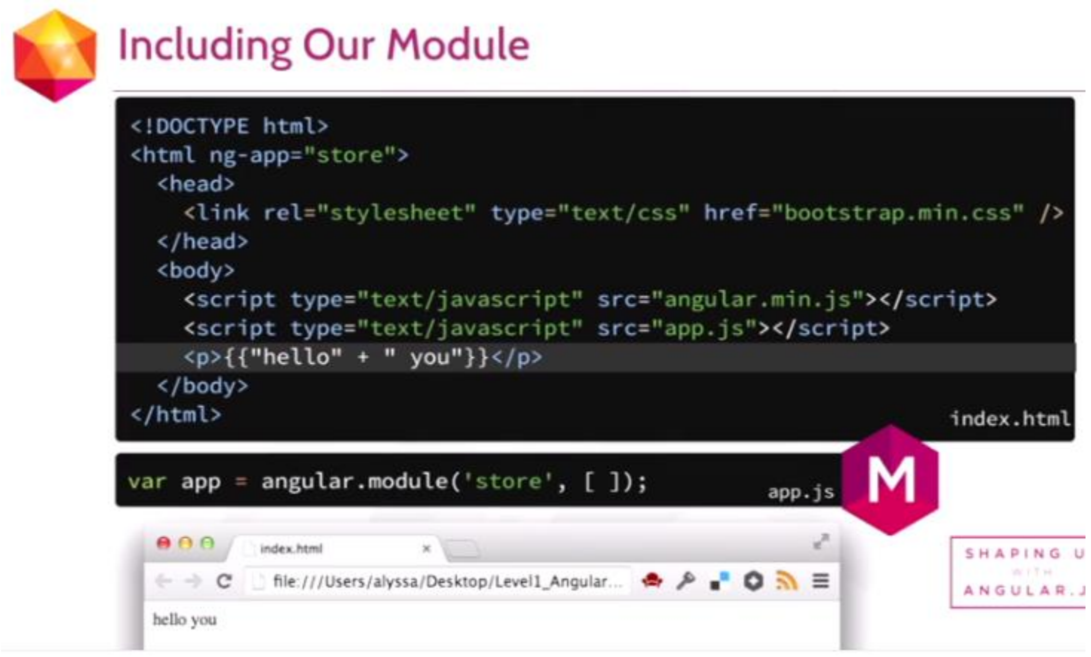
```
<p>
  I am 10
</p>
```

String Operations

```
<p>
  {{"hello" + " you"}}
</p>
```

evaluates to

```
<p>
  hello you
</p>
```



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>
    <p>{{"hello" + " you"}}</p>
  </body>
</html>
```

index.html

```
var app = angular.module('store', [ ]);
```

app.js

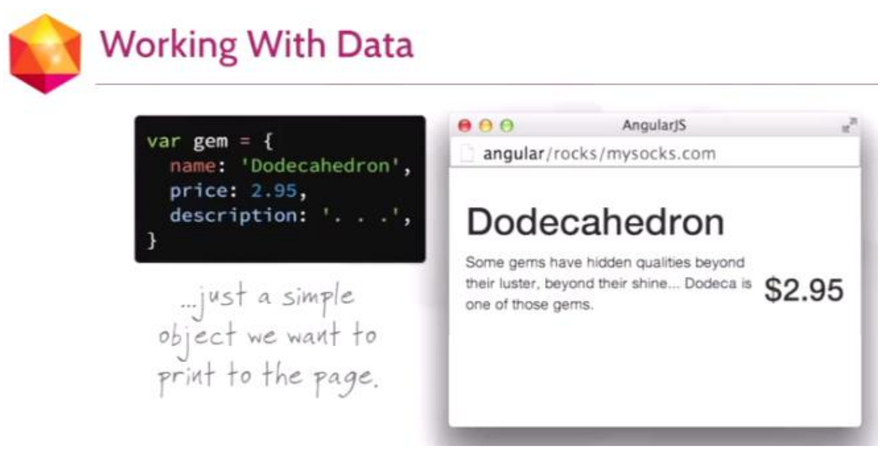
index.html

file:///Users/alyssa/Desktop/Level1_Angular...

hello you

Controllers

Controllers = Get data onto the page



Working With Data

```
var gem = {
  name: 'Dodecahedron',
  price: 2.95,
  description: '. . .',
}
```

...just a simple object we want to print to the page.

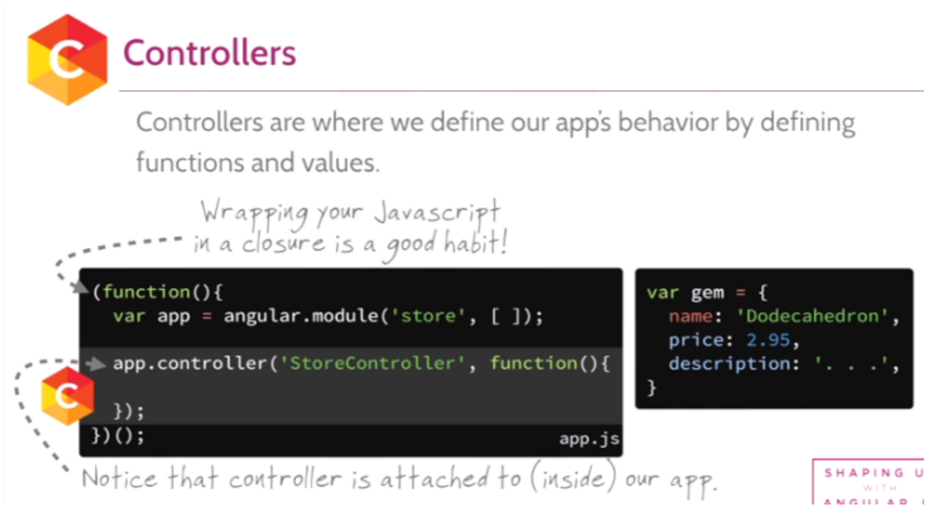
AngularJS

angular/rocks/mysocks.com

Dodecahedron

Some gems have hidden qualities beyond their luster, beyond their shine... Dodeca is one of those gems.

$2.95



Controllers

Controllers are where we define our app's behavior by defining functions and values.

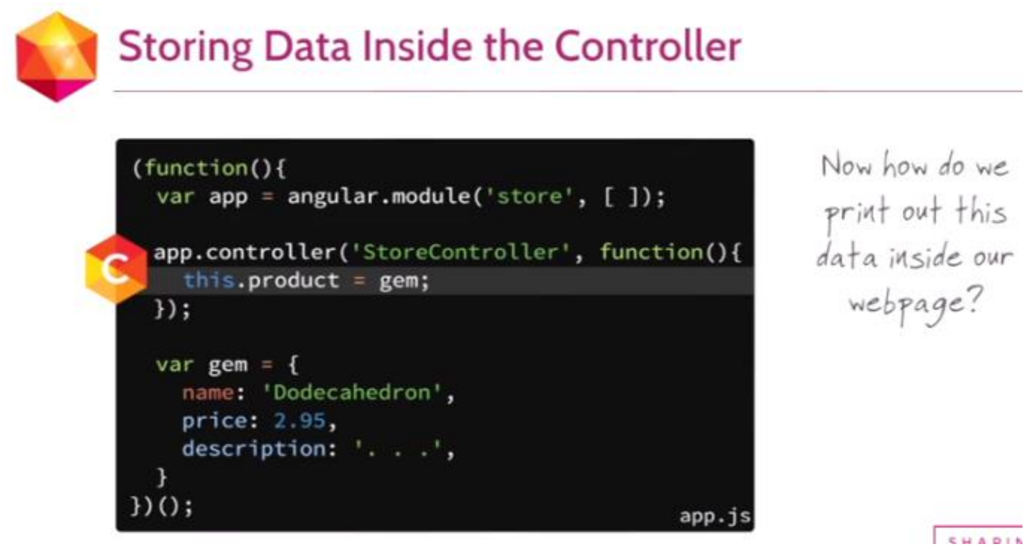
Wrapping your Javascript in a closure is a good habit!

```
(function(){
  var app = angular.module('store', [ ]);

  app.controller('StoreController', function(){

  });
})();
```

app.js

```
var gem = {
  name: 'Dodecahedron',
  price: 2.95,
  description: '. . .',
}
```

Notice that controller is attached to (inside) our app.



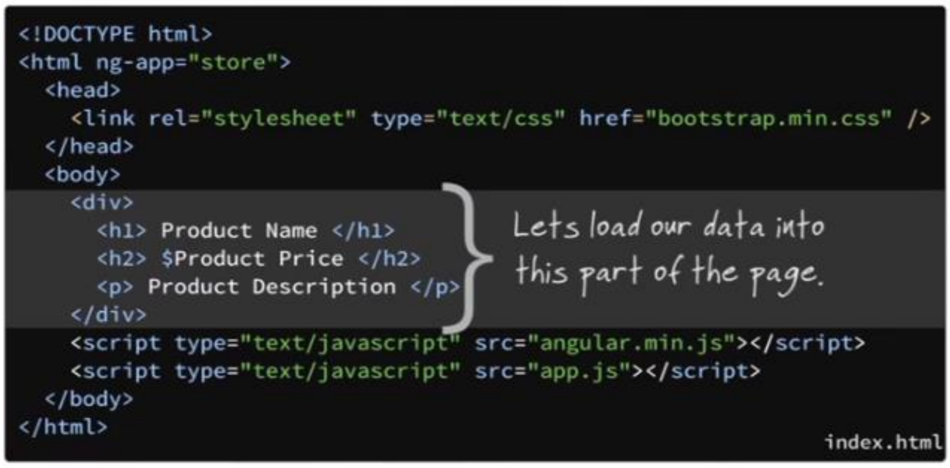
Storing Data Inside the Controller

```
(function(){
  var app = angular.module('store', [ ]);

  app.controller('StoreController', function(){
    this.product = gem;
  });

  var gem = {
    name: 'Dodecahedron',
    price: 2.95,
    description: '. . .',
  }
})();
```

app.js

Now how do we print out this data inside our webpage?

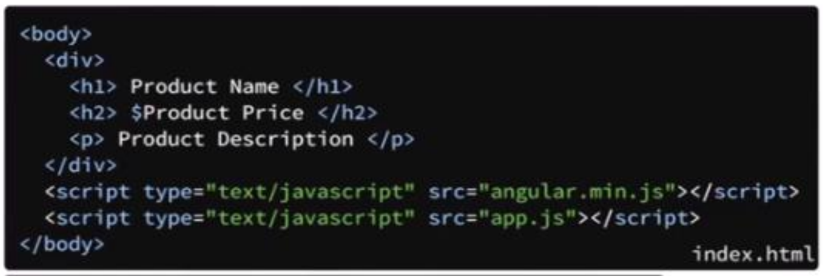
Our Current HTML



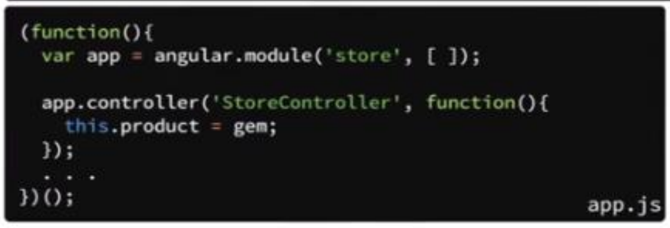
```html
<!DOCTYPE html>
<html ng-app="store">
  <head>
    <link rel="stylesheet" type="text/css" href="bootstrap.min.css" />
  </head>
  <body>
    <div>
      <h1> Product Name </h1>
      <h2> $Product Price </h2>
      <p> Product Description </p>
    </div>
    <script type="text/javascript" src="angular.min.js"></script>
    <script type="text/javascript" src="app.js"></script>
  </body>
</html>
```
index.html

Lets load our data into this part of the page.

Attaching the Controller

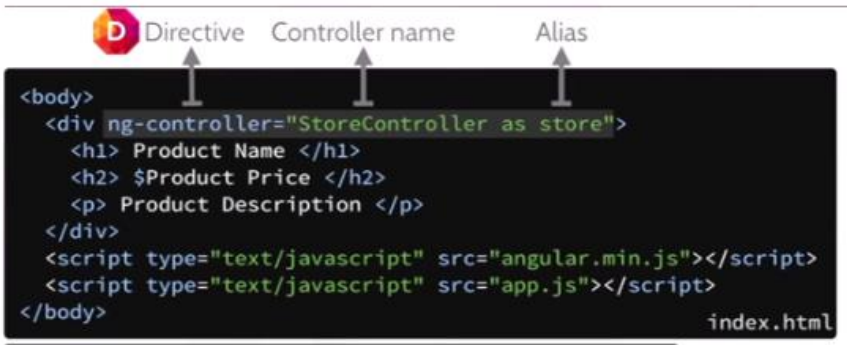


```html
<body>
  <div>
    <h1> Product Name </h1>
    <h2> $Product Price </h2>
    <p> Product Description </p>
  </div>
  <script type="text/javascript" src="angular.min.js"></script>
  <script type="text/javascript" src="app.js"></script>
</body>
```
index.html

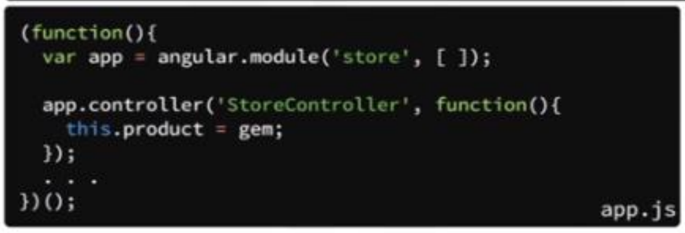


```javascript
(function(){
  var app = angular.module('store', [ ]);

  app.controller('StoreController', function(){
    this.product = gem;
  });
  . . .
})();
```
app.js

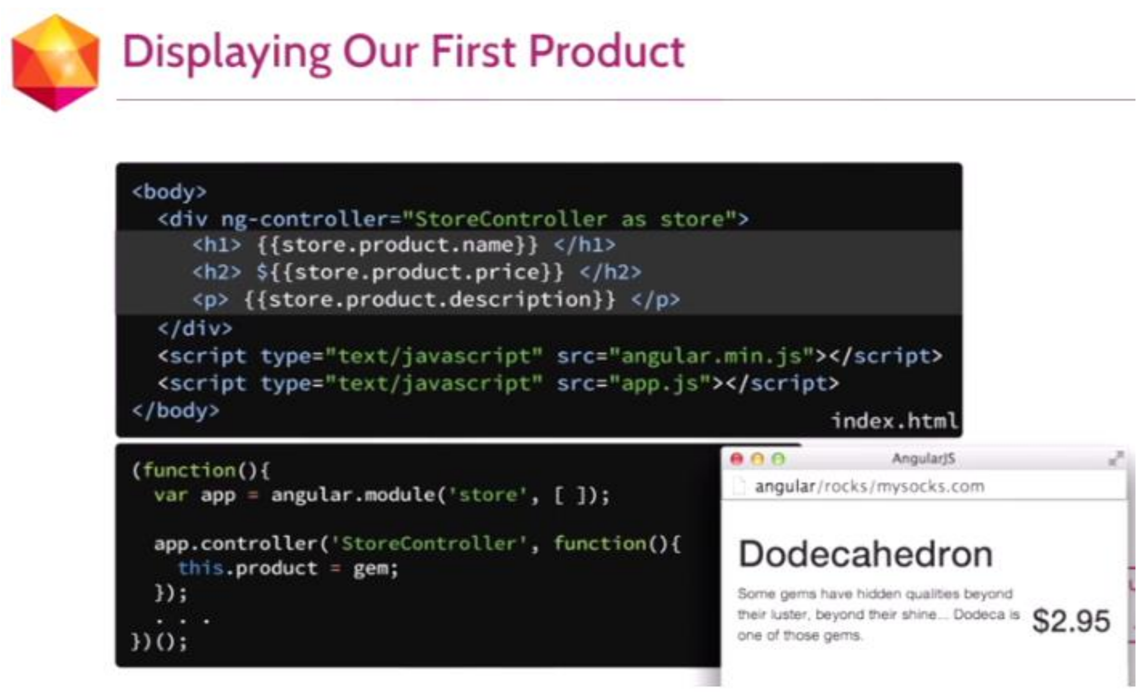
Attaching the Controller



```html
<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



```javascript
(function(){
  var app = angular.module('store', [ ]);

  app.controller('StoreController', function(){
    this.product = gem;
  });
  . . .
})();
```
app.js



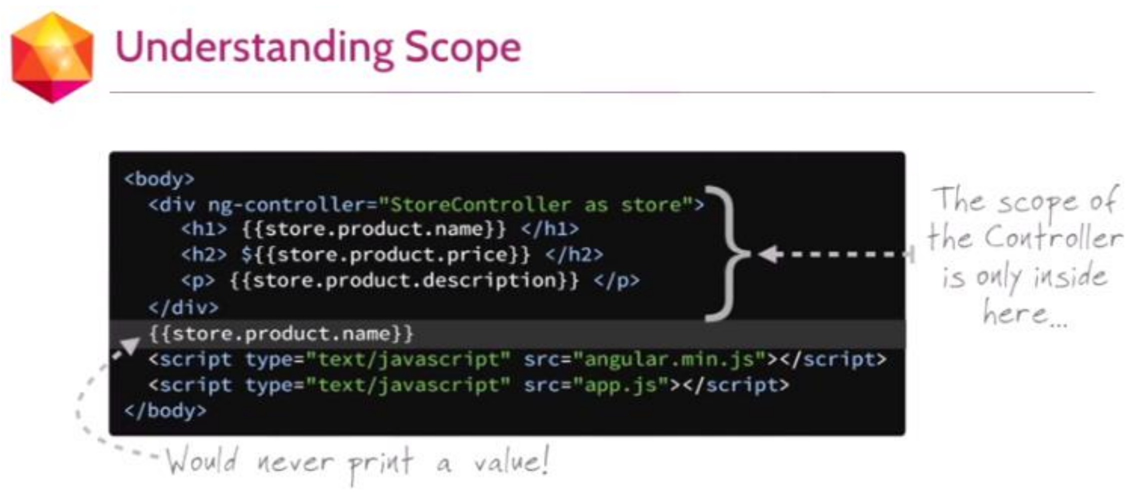
Displaying Our First Product

```
<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;
  });
  . . .
})();
```

AngularJS
angular/rocks/mysocks.com
Dodecahedron
Some gems have hidden qualities beyond their luster, beyond their shine... Dodeca is one of those gems.
$2.95

Scope



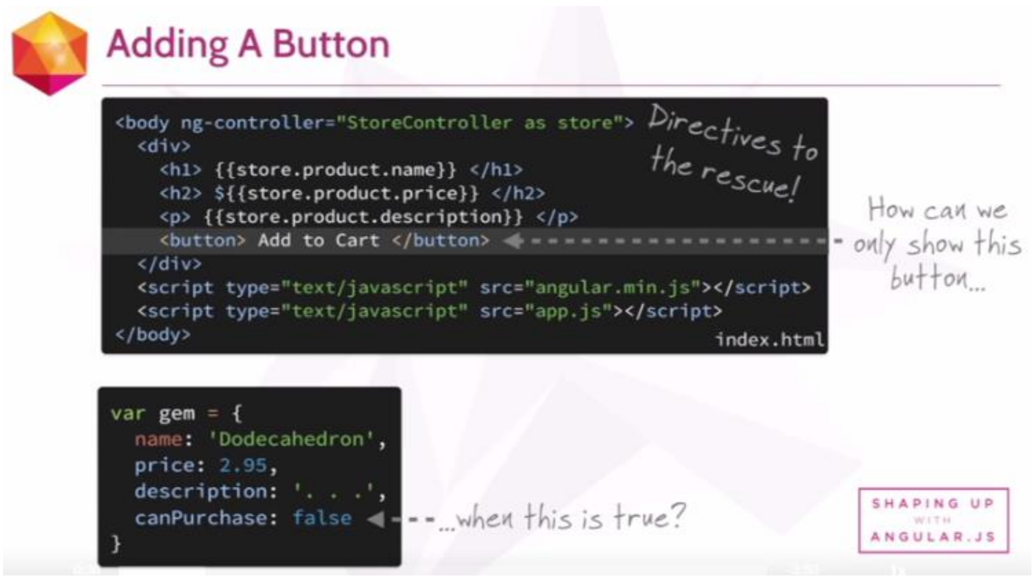
Understanding Scope

```
<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

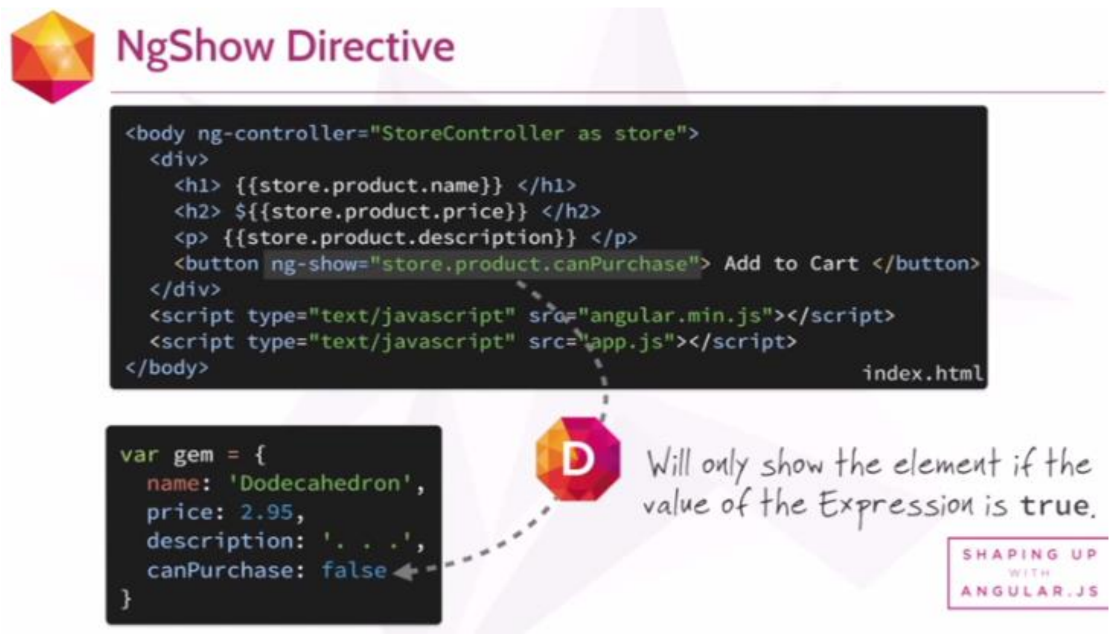
```
<body ng-controller="StoreController as store">
  <div>
    <h1> {{store.product.name}} </h1>
    <h2> ${{store.product.price}} </h2>
    <p> {{store.product.description}} </p>
    <button> Add to Cart </button>
  </div>
  <script type="text/javascript" src="angular.min.js"></script>
  <script type="text/javascript" src="app.js"></script>
</body>
```
index.html

Directives to the rescue!

How can we only show this button...

```
var gem = {
  name: 'Dodecahedron',
  price: 2.95,
  description: '. . .',
  canPurchase: false
}
```

...when this is true?

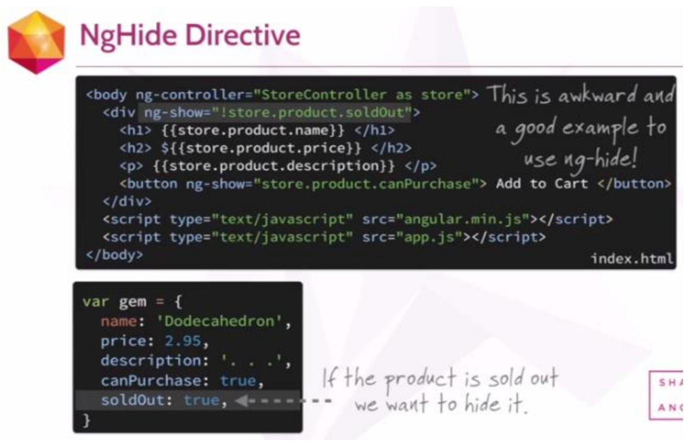


NgShow Directive

```
<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.



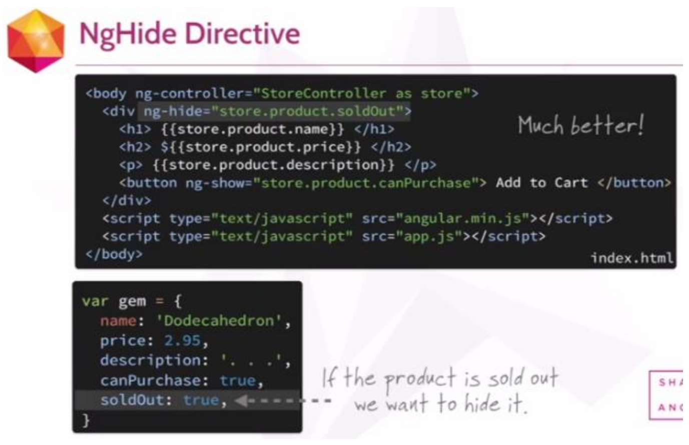
NgHide Directive

```
<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>
```
index.html

This is awkward and a good example to use ng-hide!

```
var gem = {
  name: 'Dodecahedron',
  price: 2.95,
  description: '. . .',
  canPurchase: true,
  soldOut: true,
}
```

If the product is sold out we want to hide it.



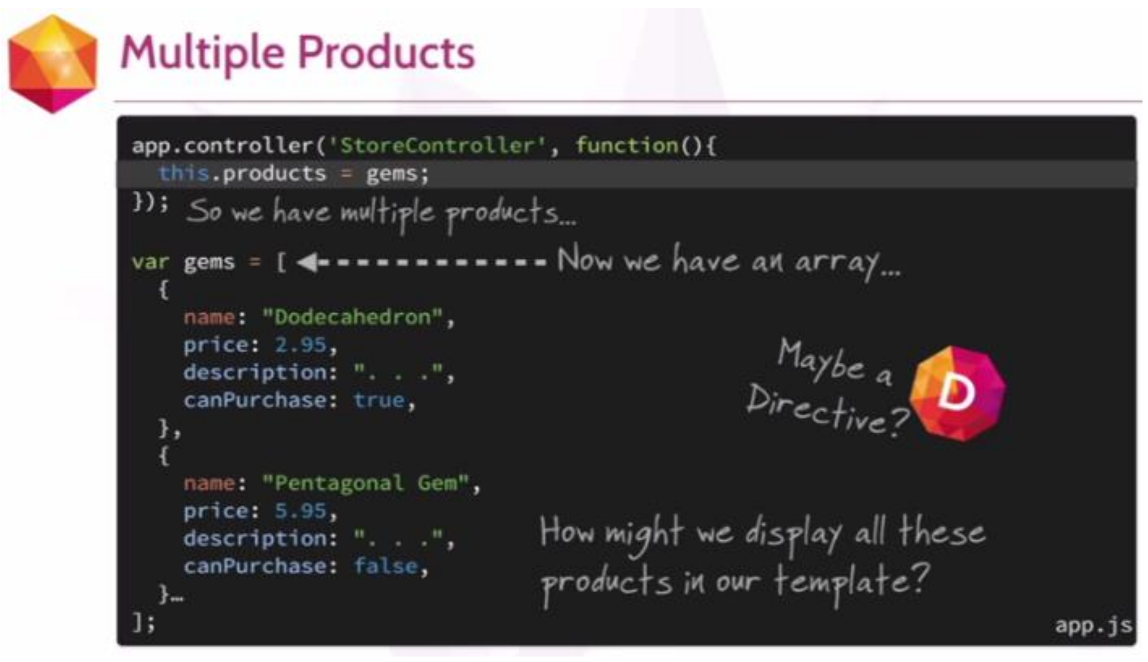
NgHide Directive

```
<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>
```
index.html

Much better!

```
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.

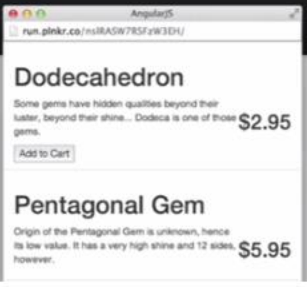
index.html

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



Dodecahedron

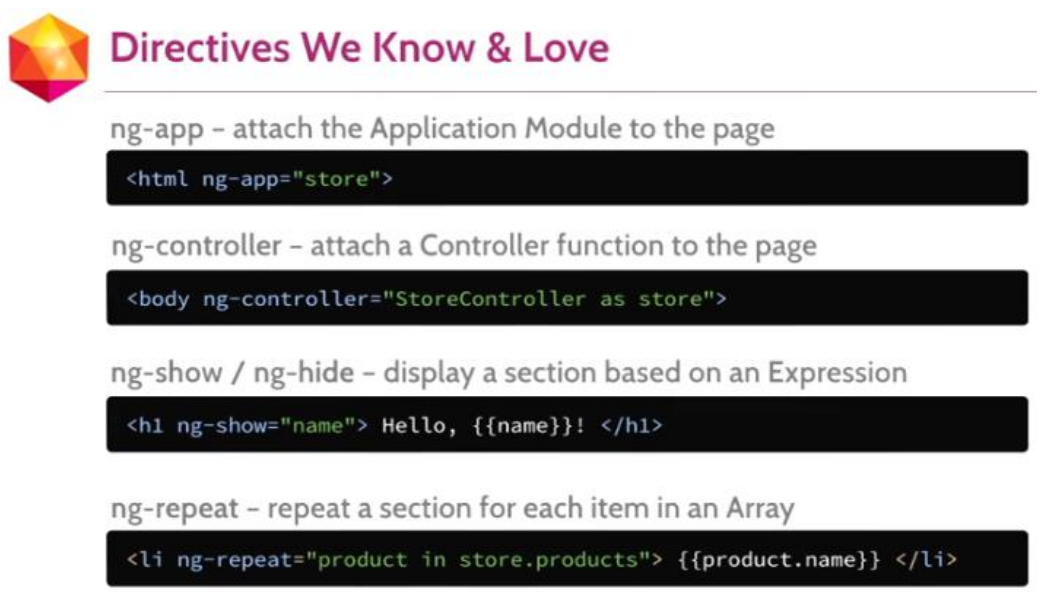
Some gems have hidden qualities beyond their luster, beyond their shine... Dodeca is one of those gems. $2.95

Pentagonal Gem

Origin of the Pentagonal Gem is unknown, hence its low value. It has a very high shine and 12 sides, however. $5.95

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

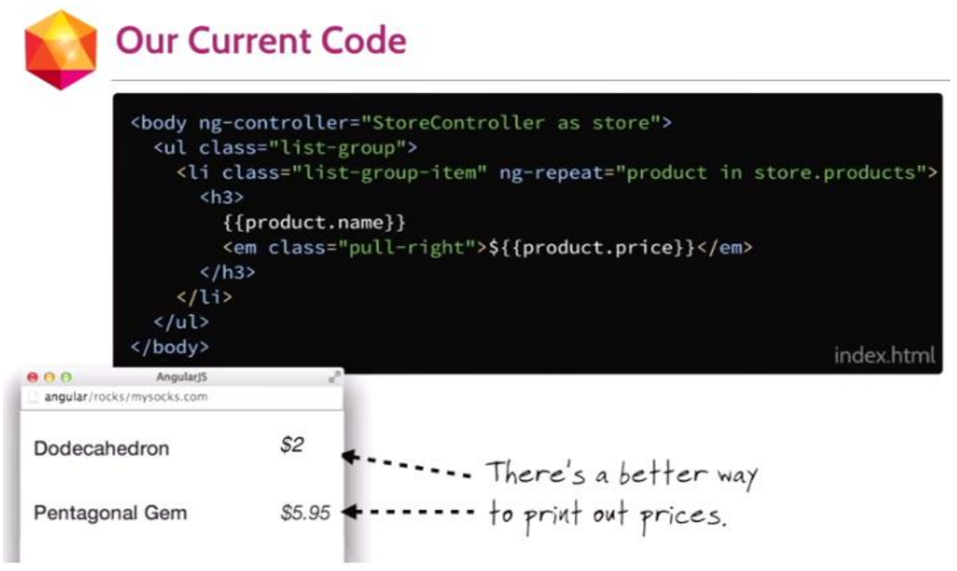
```
<body ng-controller="StoreController as store">
```

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

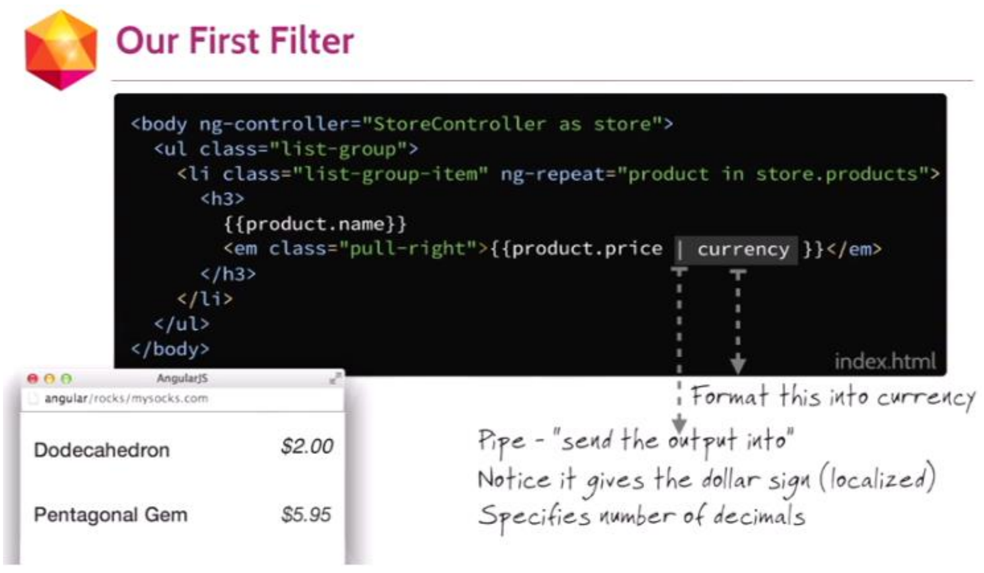
```
<body ng-controller="StoreController as store">
  <ul class="list-group">
    <li class="list-group-item" ng-repeat="product in store.products">
      <h3>
        {{product.name}}
        <em class="pull-right">${{product.price}}</em>
      </h3>
    </li>
  </ul>
</body>
```

index.html

Dodecahedron	$2
Pentagonal Gem	$5.95

There's a better way to print out prices.

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

Dodecahedron	$2.00
Pentagonal Gem	$5.95

Format this into currency

Pipe - "send the output into"
Notice it gives the dollar sign (localized)
Specifies number of decimals

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">
```

orderBy

Will list products by descending price.

```
<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: [           Our New Array
      {                                  Image Object
        full: 'dodecahedron-01-full.jpg',
        thumb: 'dodecahedron-01-thumb.jpg'
      },
      {
        full: "dodecahedron-02-full.jpg",
        . . .
```

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.

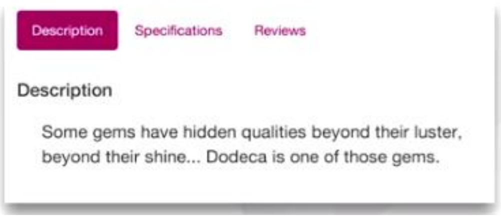
```
<body ng-controller="StoreController as store">
  <ul class="list-group">
    <li class="list-group-item" ng-repeat="product in store.products">
      <h3>
        {{product.name}}
        <em class="pull-right">{{product.price | currency}}</em>
        <img ng-src="{{product.images[0].full}}"/>
      </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



Description

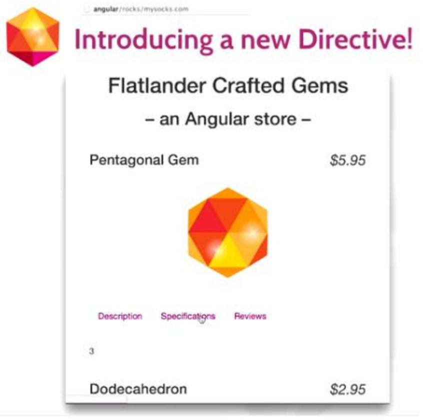
Some gems have hidden qualities beyond their luster, beyond their shine... Dodeca is one of those gems.

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.



Flatlander Crafted Gems

– an Angular store –

Pentagonal Gem $5.95

3

Dodecahedron $2.95

Whoa, it's dynamic and stuff...

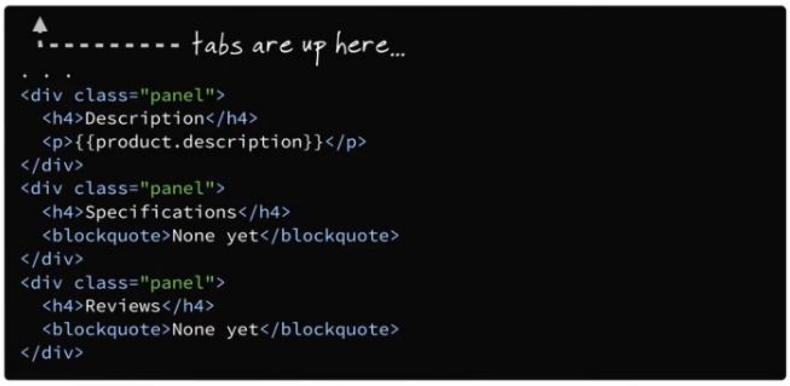
When `ng-click` changes the value of `tab` ...

... the `{{tab}}` expression automatically gets updated!

Expressions define a 2-way Data Binding ...

this means Expressions are re-evaluated when a property changes.

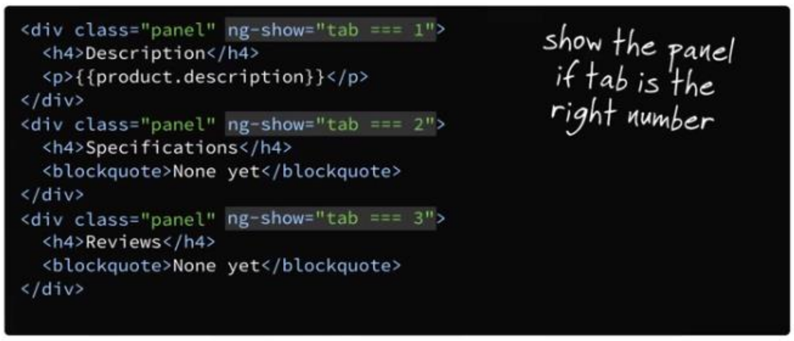
Let's add the tab content panels



```
. . .
<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

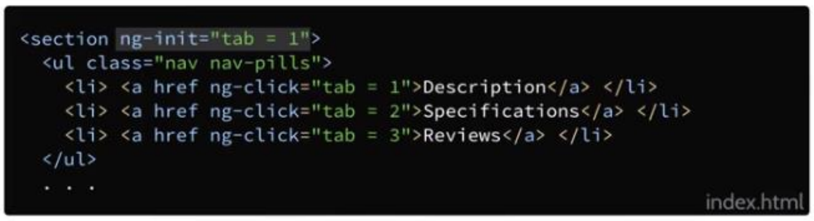


```
<div class="panel" ng-show="tab === 1">
  <h4>Description</h4>
  <p>{{product.description}}</p>
</div>
<div class="panel" ng-show="tab === 2">
  <h4>Specifications</h4>
  <blockquote>None yet</blockquote>
</div>
<div class="panel" ng-show="tab === 3">
  <h4>Reviews</h4>
  <blockquote>None yet</blockquote>
</div>
```

Now when a tab is selected it will show the appropriate panel!

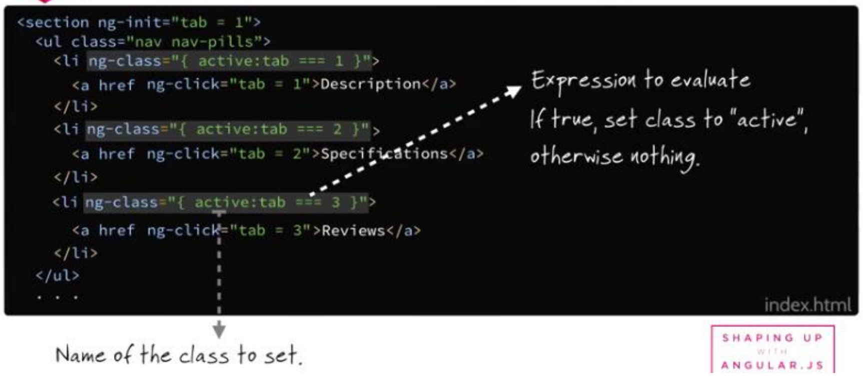
Setting the Initial Value

`ng-init` allows us to evaluate an expression in the current scope.



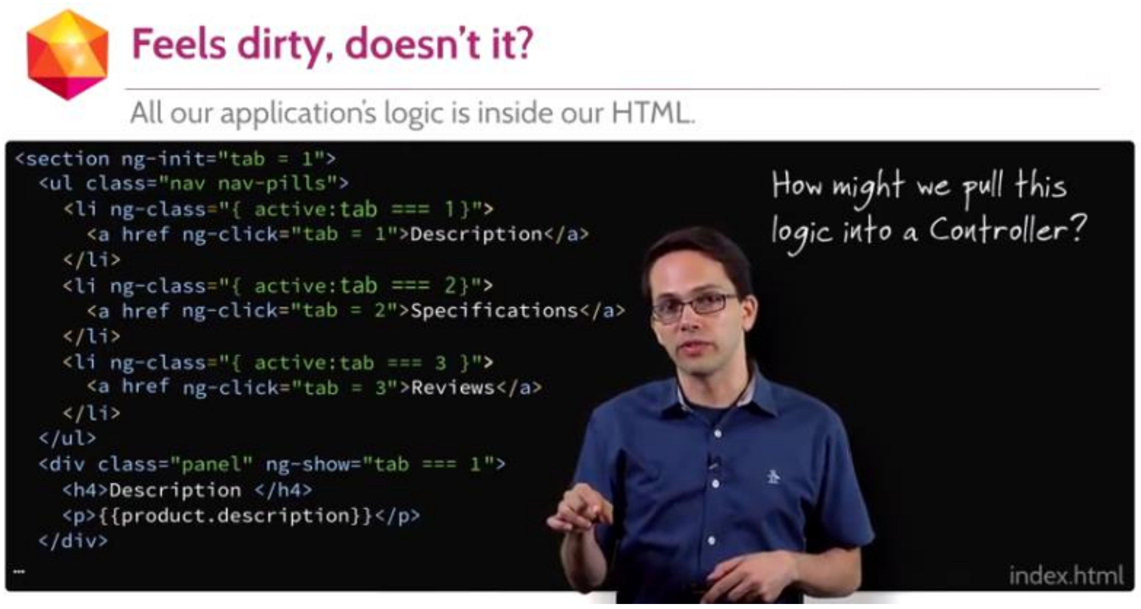
```
<section ng-init="tab = 1">
  <ul class="nav nav-pills">
    <li> <a href ng-click="tab = 1">Description</a> </li>
    <li> <a href ng-click="tab = 2">Specifications</a> </li>
    <li> <a href ng-click="tab = 3">Reviews</a> </li>
  </ul>
  . . .
```

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>
  . . .
```
Expression to evaluate
If true, set class to "active", otherwise nothing.
index.html
Name of the class to set.
SHAPING UP WITH ANGULAR.JS

Decoupling



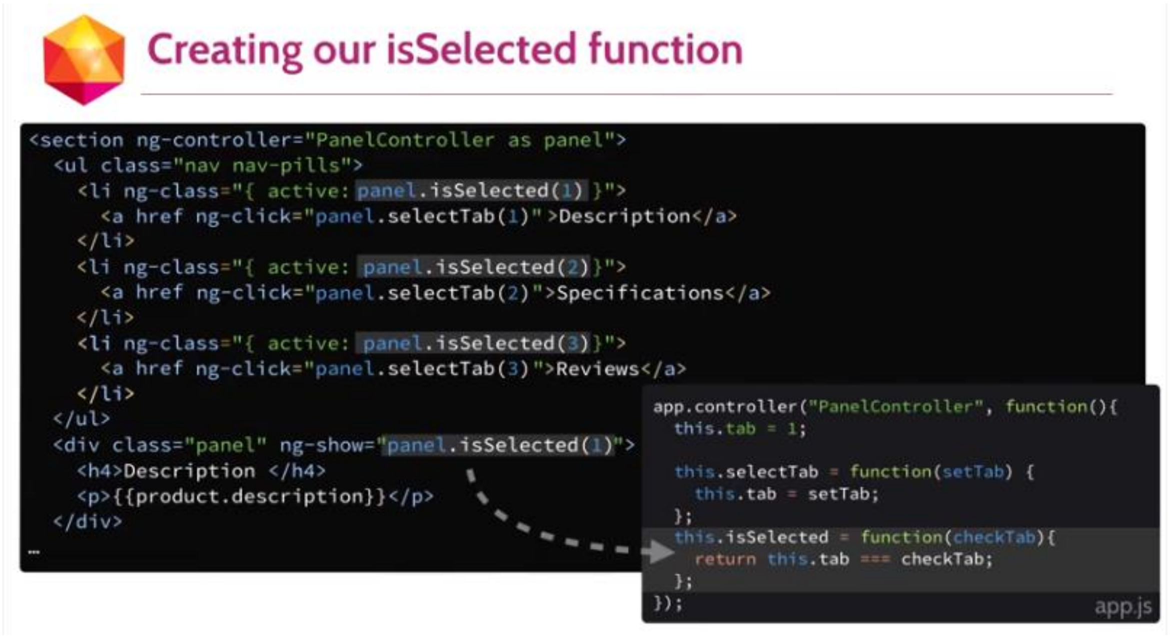
Feels dirty, doesn't it?

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</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>
  <div class="panel" ng-show="tab === 1">
    <h4>Description </h4>
    <p>{{product.description}}</p>
  </div>
...
```

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</a>
    </li>
    <li ng-class="{ active: panel.isSelected(2)}">
      <a href ng-click="panel.selectTab(2)">Specifications</a>
    </li>
    <li ng-class="{ active: panel.isSelected(3)}">
      <a href ng-click="panel.selectTab(3)">Reviews</a>
    </li>
  </ul>
  <div class="panel" ng-show="panel.isSelected(1)">
    <h4>Description </h4>
    <p>{{product.description}}</p>
  </div>
...
```

```
app.controller("PanelController", function(){
  this.tab = 1;

  this.selectTab = function(setTab) {
    this.tab = setTab;
  };
  this.isSelected = function(checkTab){
    return this.tab === checkTab;
  };
});
```

app.js

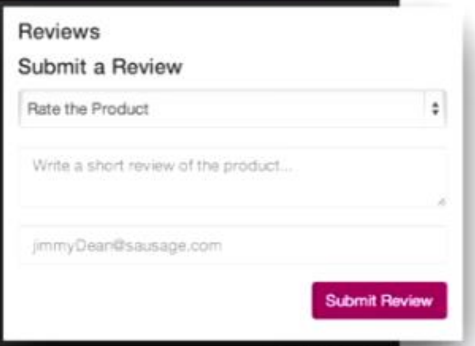
Looping Over Reviews in our Tab

```
<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>
```



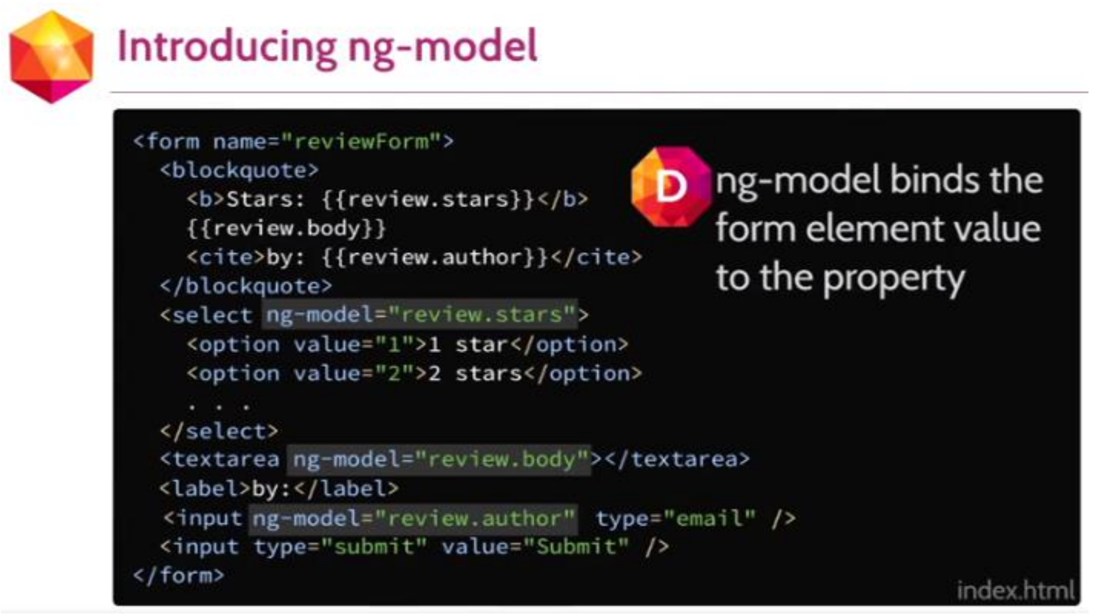
index.html

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

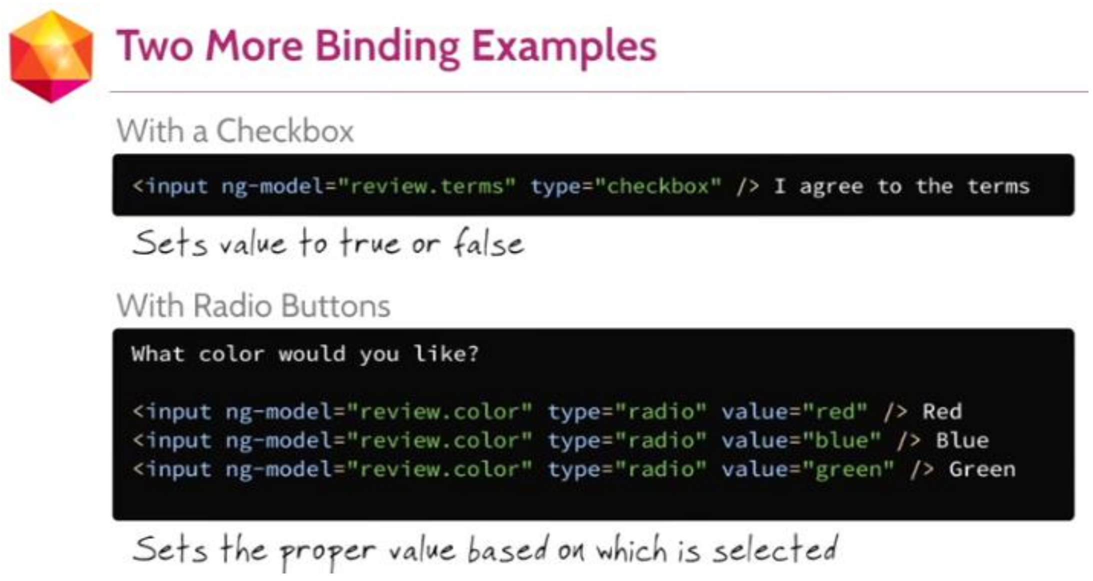


Introducing ng-model

```
<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

D ng-model binds the form element value to the property



Two More Binding Examples

With a Checkbox

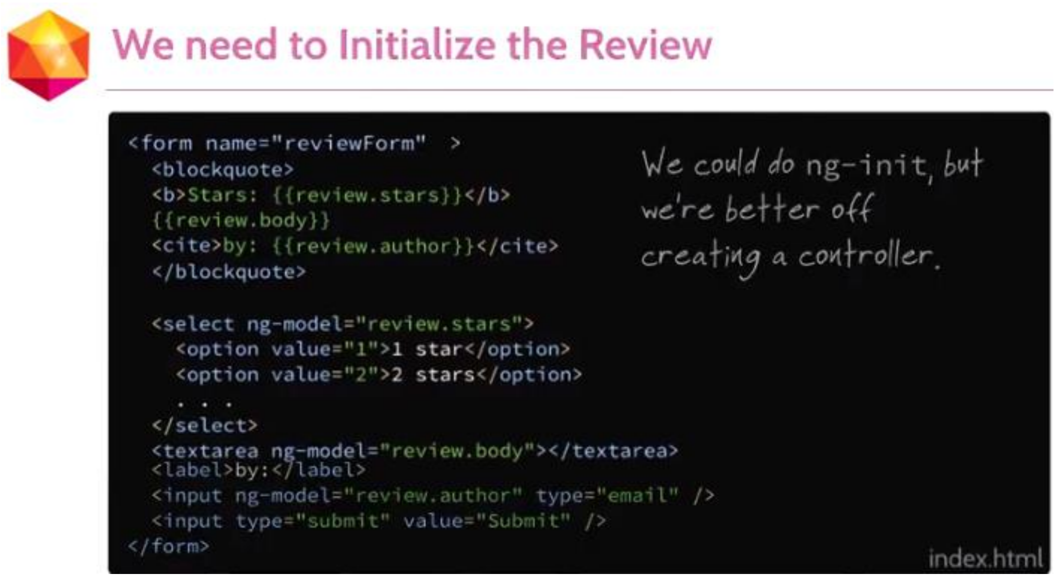
```
<input ng-model="review.terms" type="checkbox" /> I agree to the terms
```

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

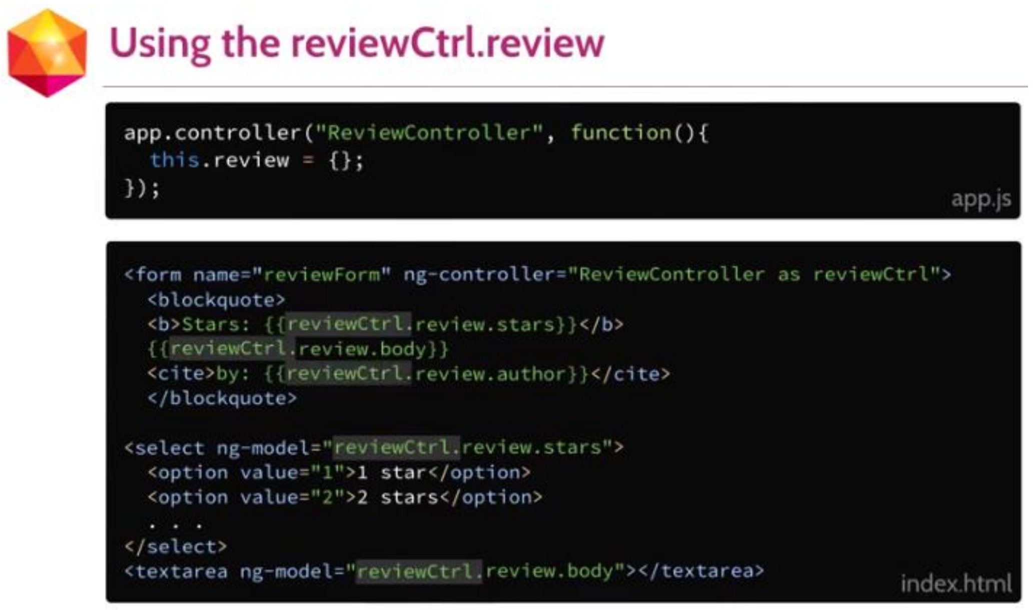


We need to Initialize the Review

```
<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.



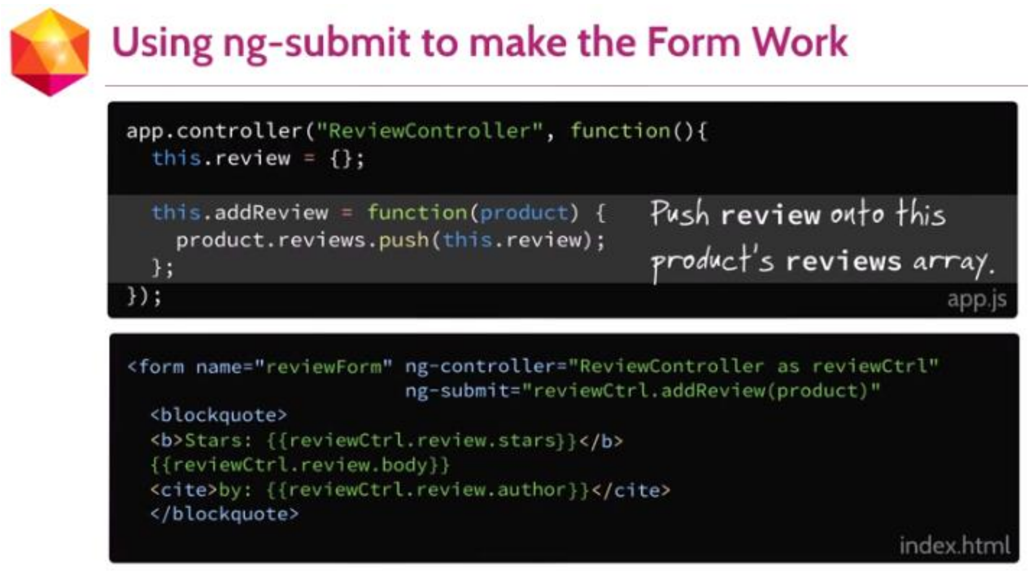
Using the reviewCtrl.review

```
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);
  };
});
```

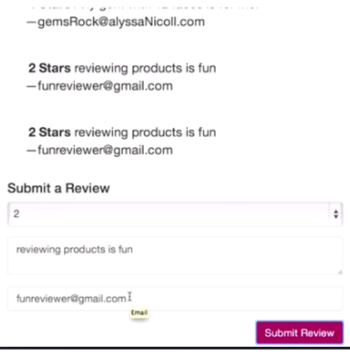
Push review onto this product's reviews array.

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!



—gemsRock@alyssaNicoll.com

2 Stars reviewing products is fun
—funreviewer@gmail.com

2 Stars reviewing products is fun
—funreviewer@gmail.com

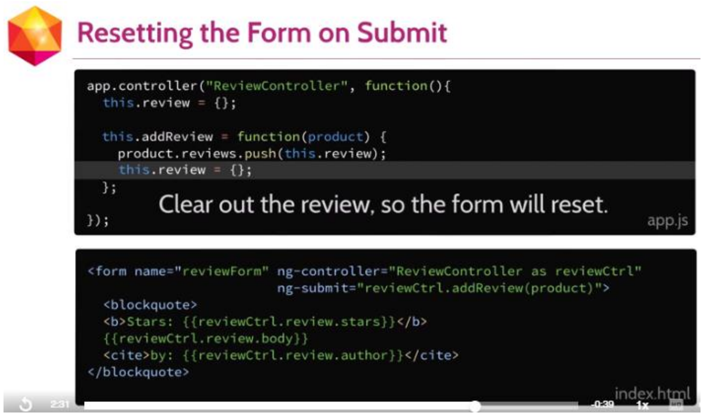
Submit a Review

2

reviewing products is fun

funreviewer@gmail.com

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 {          Red border for invalid
  border-color: #FA787E;
}

.ng-valid.ng-dirty {            Green border for valid
  border-color: #78FA89;
}
```
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">
```

Can also define min & max with numbers

```
<input type="number" name="quantity">
```

```
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?

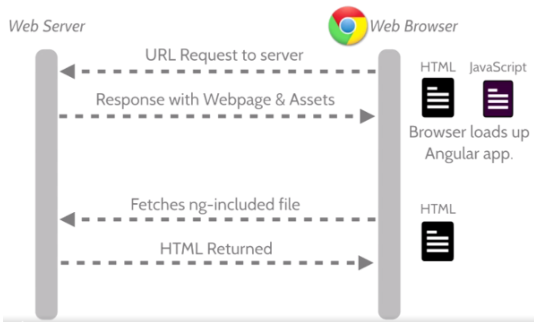
Using ng-include for Templates

```
<ul class="list-group">
  <li class="list-group-item" ng-repeat="product in store.products">
    <h3 ng-include="'product-title.html'"> ⊢ - → name of file to include
    </h3>
    <section ng-controller="PanelController as panel">
```
index.html

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>
```

Can you tell what this does?

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:

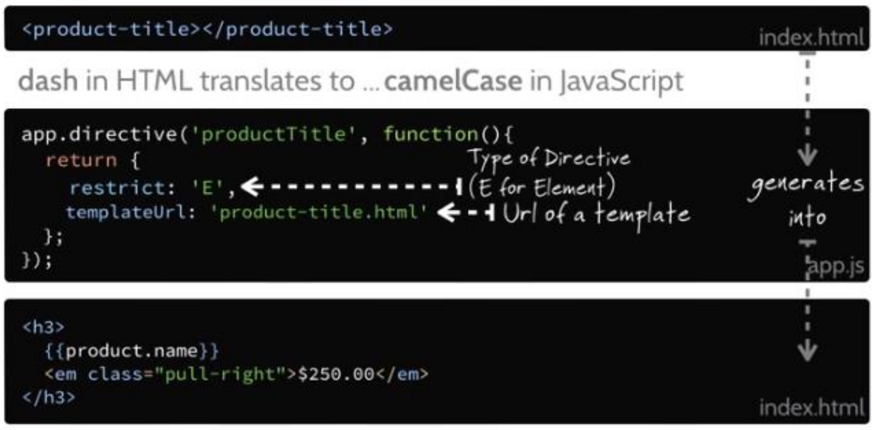
- Expressing complex UI
- Calling events and registering event handlers
- Reusing common components

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>
```

dash in HTML translates to ... camelCase in JavaScript

```
app.directive('productTitle', function(){
  return {
    restrict: 'E',
    templateUrl: 'product-title.html'
  };
});
```

```
<h3>
  {{product.name}}
  <em class="pull-right">$250.00</em>
</h3>
```

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.

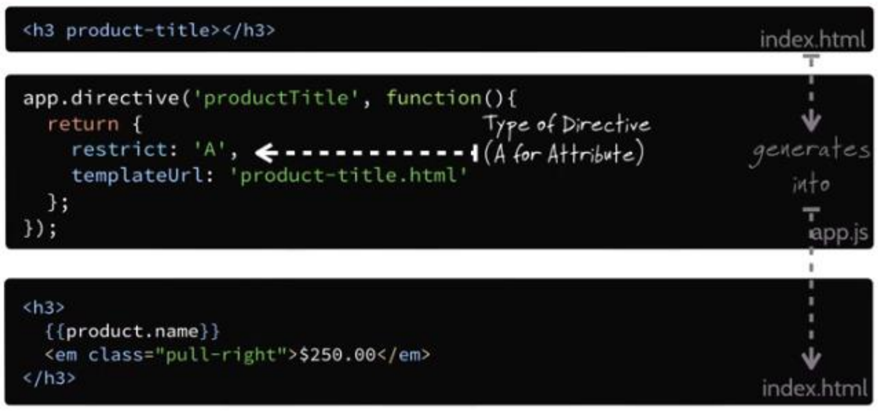
Attribute Directive

```
<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>
```

```
app.directive('productTitle', function(){
  return {
    restrict: 'A',
    templateUrl: 'product-title.html'
  };
});
```

```
<h3>
  {{product.name}}
  <em class="pull-right">$250.00</em>
</h3>
```

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.

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>
```

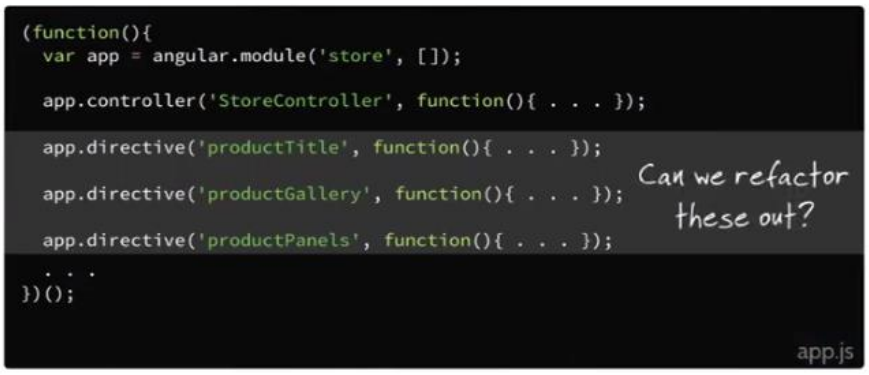
index.html

```
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?

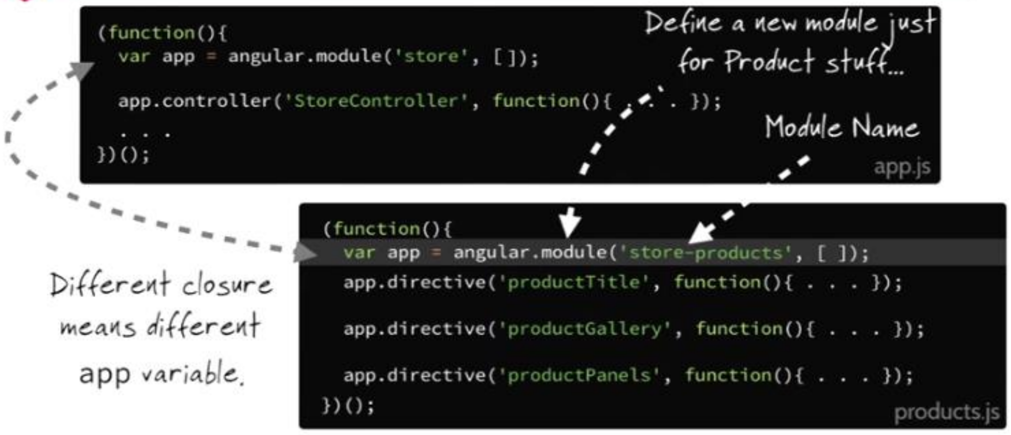


```
(function(){
  var app = angular.module('store', []);

  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...

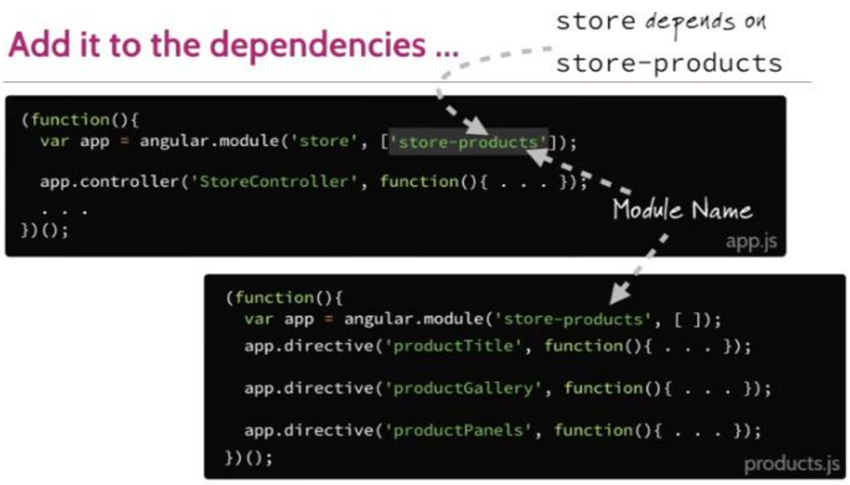
Module Name

```
(function(){
  var app = angular.module('store-products', [ ]);
  app.directive('productTitle', function(){ . . . });

  app.directive('productGallery', function(){ . . . });

  app.directive('productPanels', function(){ . . . });
})();
```

products.js

Different closure means different app variable.

Add it to the dependencies ...



store depends on store-products

```
(function(){
  var app = angular.module('store', ['store-products']);

  app.controller('StoreController', function(){ . . . });
  . . .
})();
```

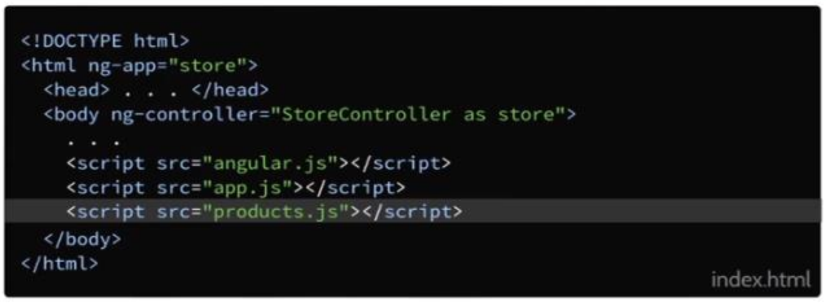
app.js

Module Name

```
(function(){
  var app = angular.module('store-products', [ ]);
  app.directive('productTitle', function(){ . . . });

  app.directive('productGallery', function(){ . . . });

  app.directive('productPanels', function(){ . . . });
})();
```

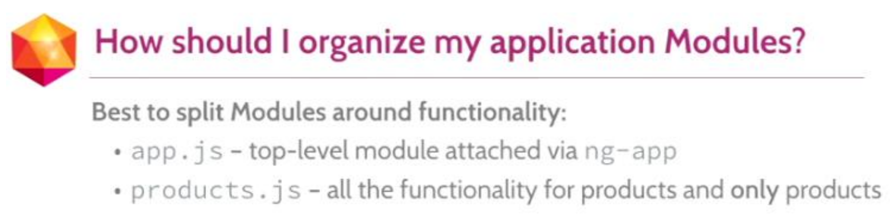
products.js

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>
```

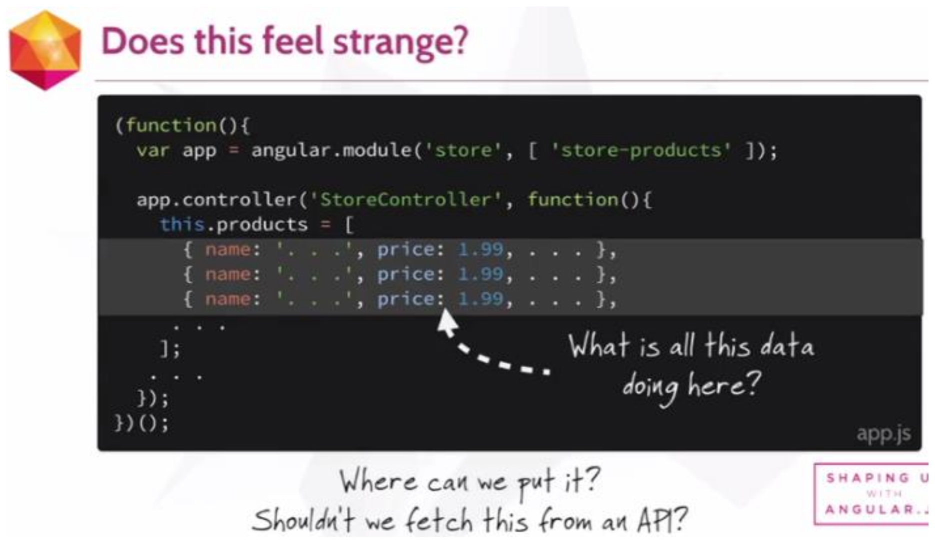
index.html



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



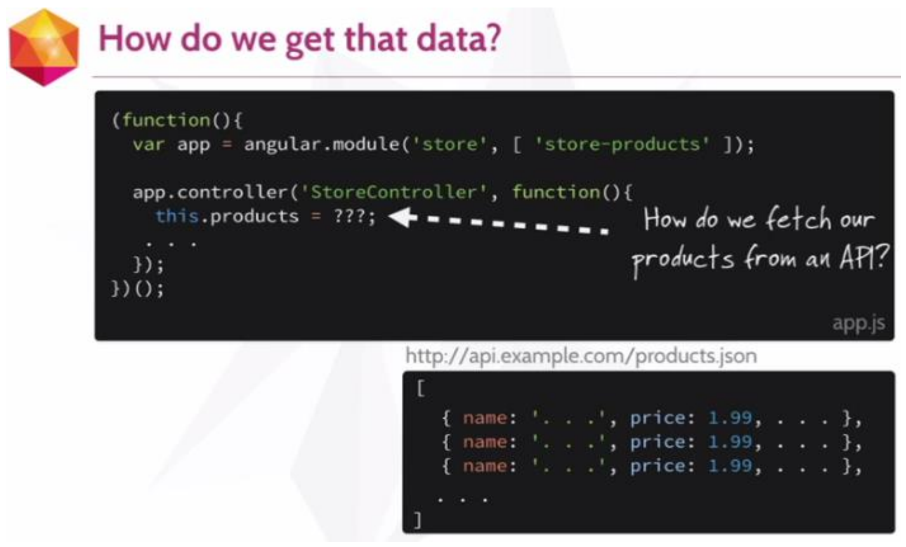
Does this feel strange?

```
(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?

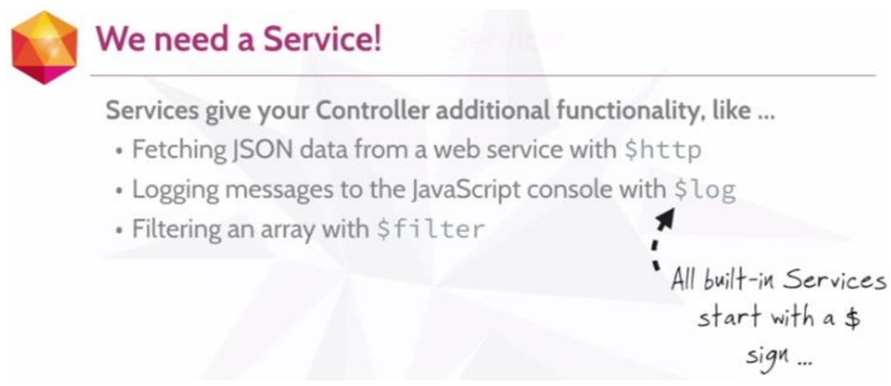
```
(function(){
  var app = angular.module('store', [ 'store-products' ]);

  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, . . . },
  . . .
]
```

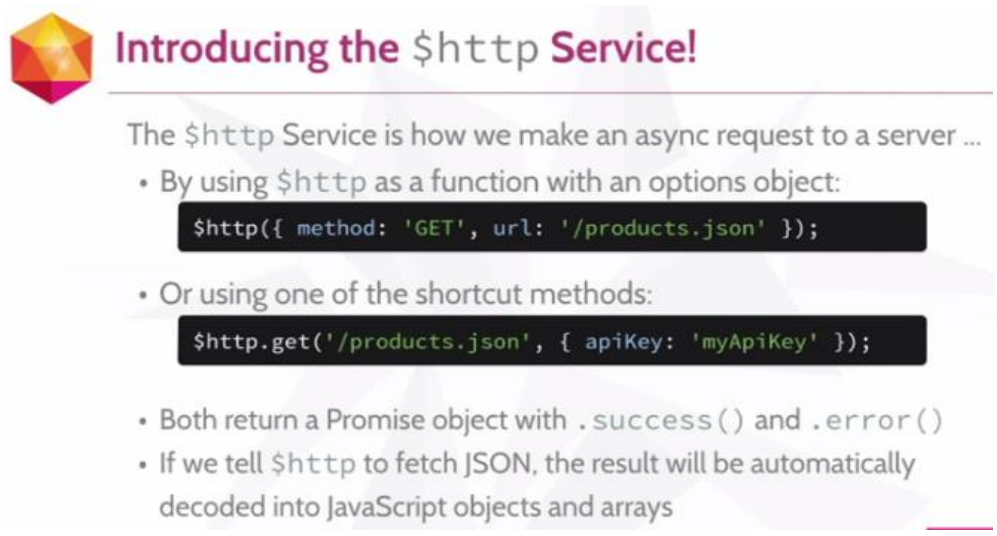


We need a Service!

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 ...

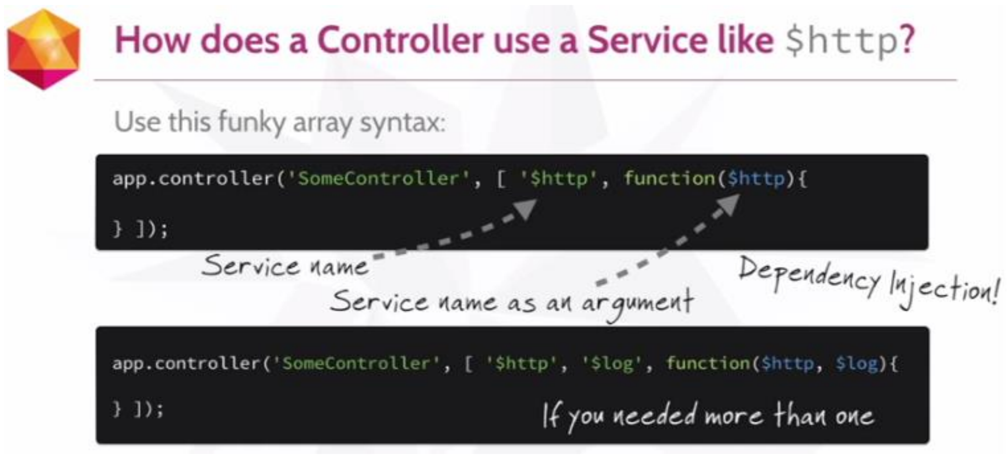
- By using `$http` as a function with an options object:

```
$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

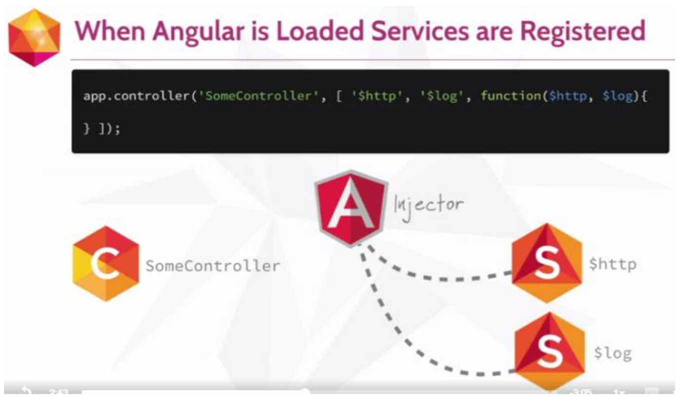


How does a Controller use a Service like `$http`?

Use this funky array syntax:

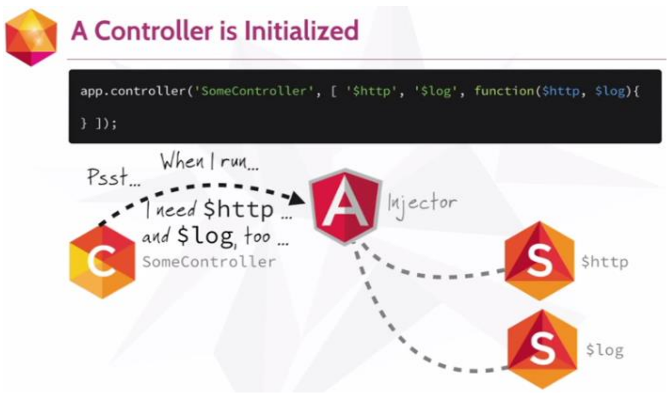
```
app.controller('SomeController', [ '$http', function($http){
} ]);
```

```
app.controller('SomeController', [ '$http', '$log', function($http, $log){
} ]);
```



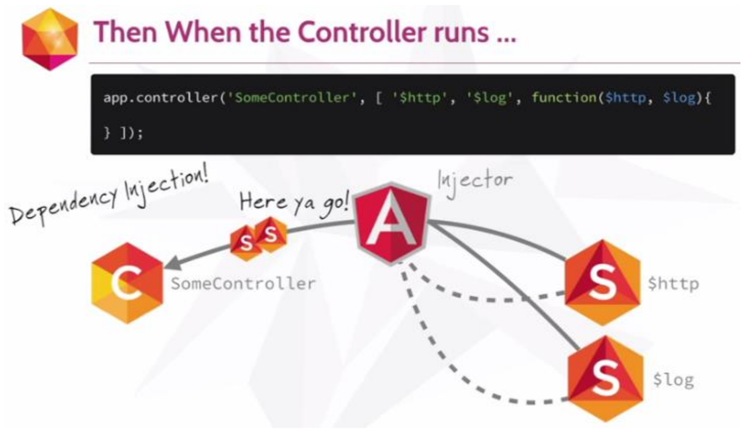
When Angular is Loaded Services are Registered

```
app.controller('SomeController', [ '$http', '$log', function($http, $log){
} ]);
```



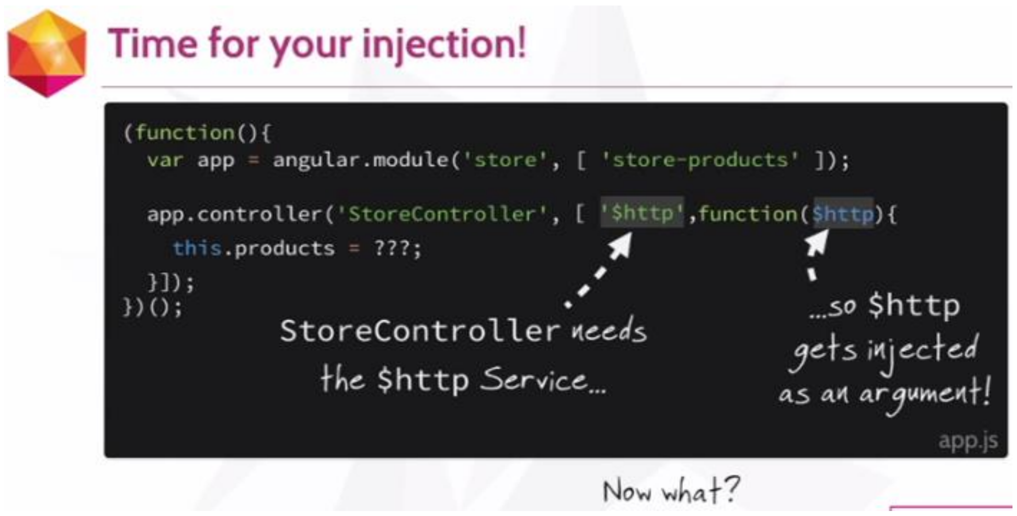
A Controller is Initialized

```
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!

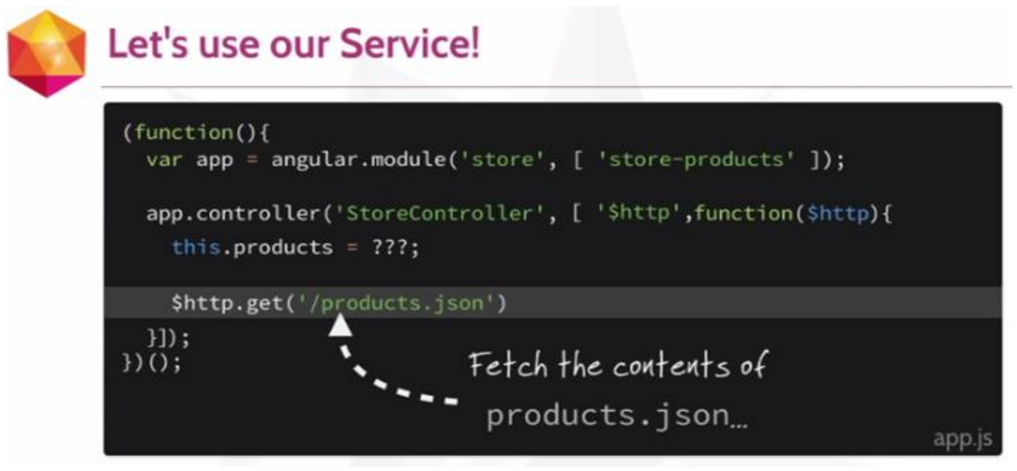
```
(function(){
  var app = angular.module('store', [ 'store-products' ]);

  app.controller('StoreController', [ '$http',function($http){
    this.products = ???;
  }]);
})();
```

StoreController needs the $http Service...

...so $http gets injected as an argument!

app.js

Now what?

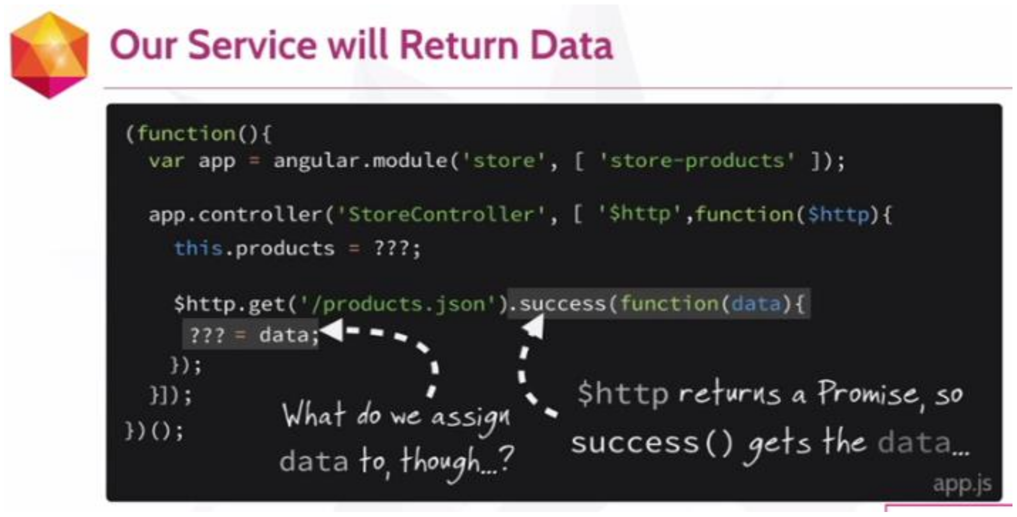


Let's use our Service!

```
(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



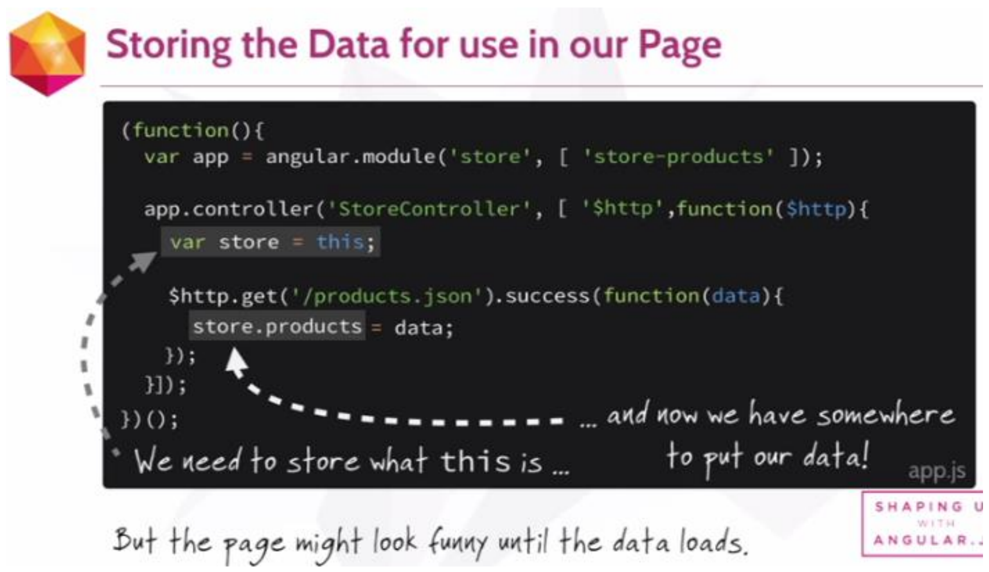
Our Service will Return Data

```
(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

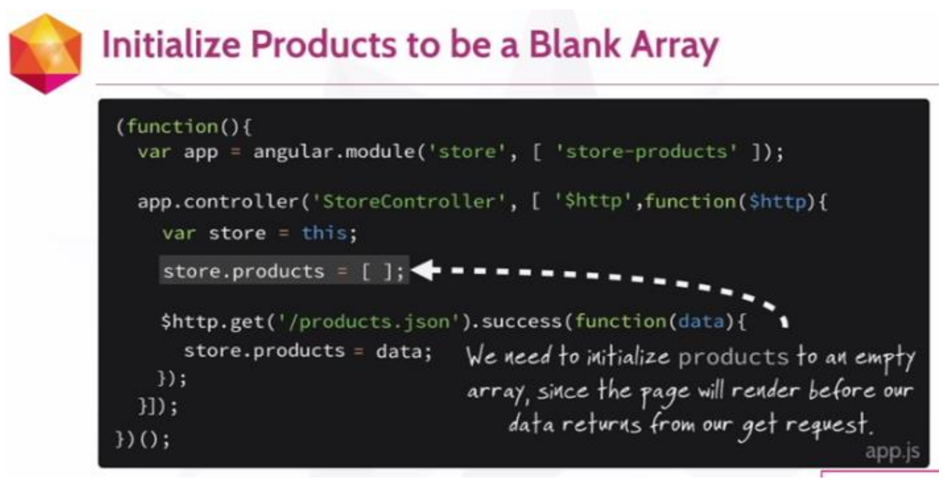
```
(function(){
  var app = angular.module('store', [ 'store-products' ]);

  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.

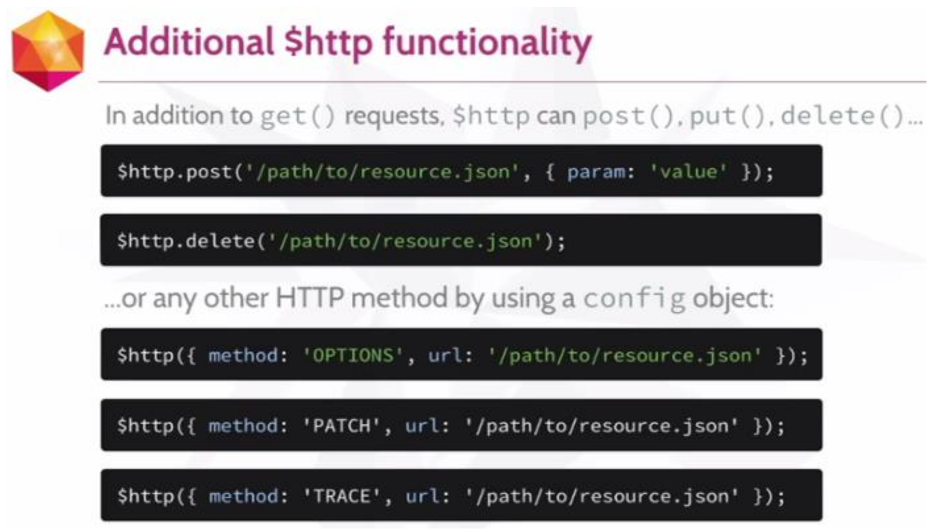


Initialize Products to be a Blank Array

```
(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');
```

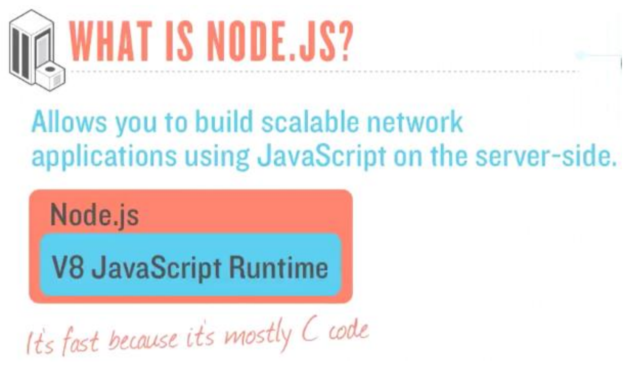
...or any other HTTP method by using a `config` object:

```
$http({ method: 'OPTIONS', url: '/path/to/resource.json' });
```

```
$http({ method: 'PATCH', url: '/path/to/resource.json' });
```

```
$http({ method: 'TRACE', url: '/path/to/resource.json' });
```

Node.js



WHAT IS NODE.JS?

Allows you to build scalable network applications using JavaScript on the server-side.

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.

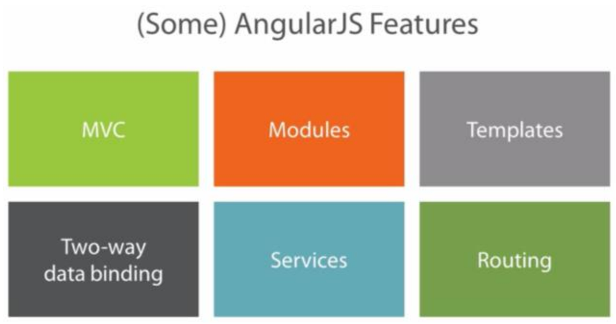
This pattern is often used when trying to avoid polluting the global namespace, because all the variables used inside the IIFE (like in any other normal function) are not visible outside its scope.
This is why, maybe, you confused this construction with an event-handler for window.onload, because it’s 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

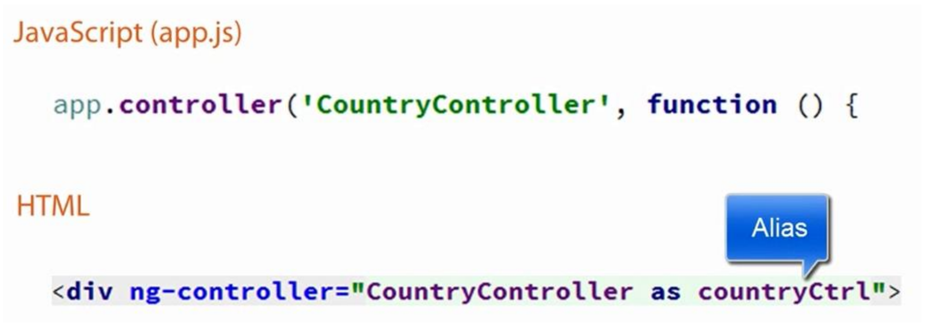


IIFE

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)

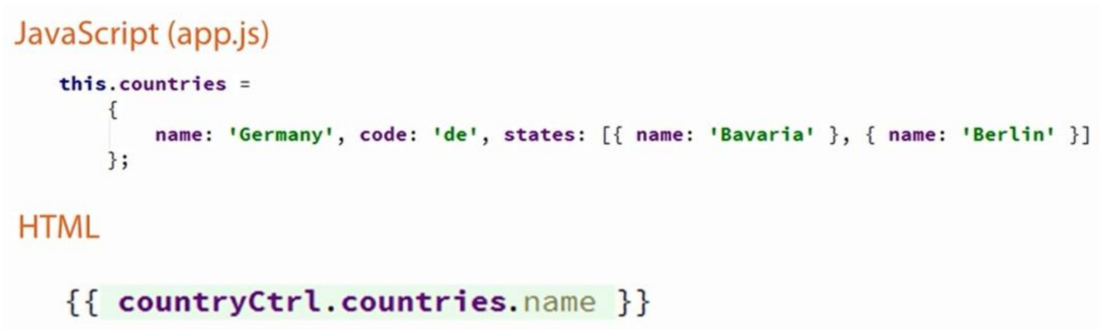
```
app.controller('CountryController', function () {
```

HTML

```
<div ng-controller="CountryController as countryCtrl">
```

You should look at **countryCtrl** as an instance of the **CountryController** class.

Databinding/Expressions



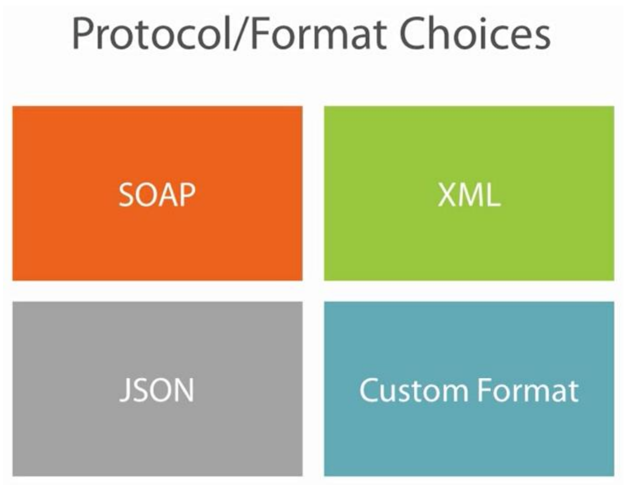
JavaScript (app.js)

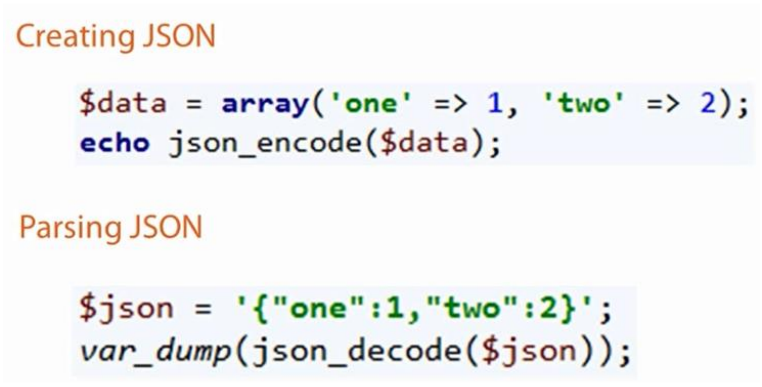
```
this.countries =
    {
        name: 'Germany', code: 'de', states: [{ name: 'Bavaria' }, { name: 'Berlin' }]
    };
```

HTML

```
{{ countryCtrl.countries.name }}
```

We bind the **countryCtrl** **instance**, then to the **countries** **property**, and then to the **name** **sub-property**.


Protocol/Format Choices

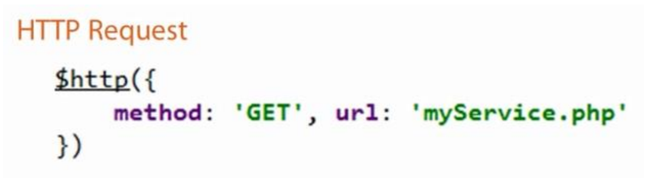

Creating JSON

```
$data = array('one' => 1, 'two' => 2);
echo json_encode($data);
```

Parsing JSON

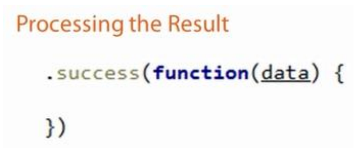
```
$json = '{"one":1,"two":2}';
var_dump(json_decode($json));
```

Calling the service from Angular


HTTP Request

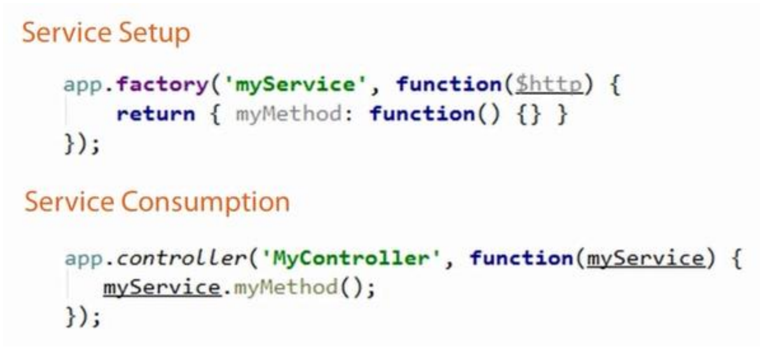
```
$http({
    method: 'GET', url: 'myService.php'
})
```

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

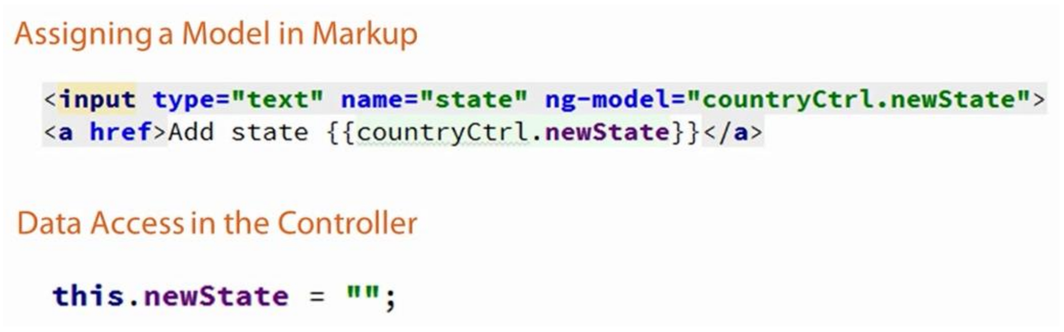

Service Setup

```
app.factory('myService', function($http) {
    return { myMethod: function() {} }
});
```

Service Consumption

```
app.controller('MyController', function(myService) {
    myService.myMethod();
});
```

Two way data binding

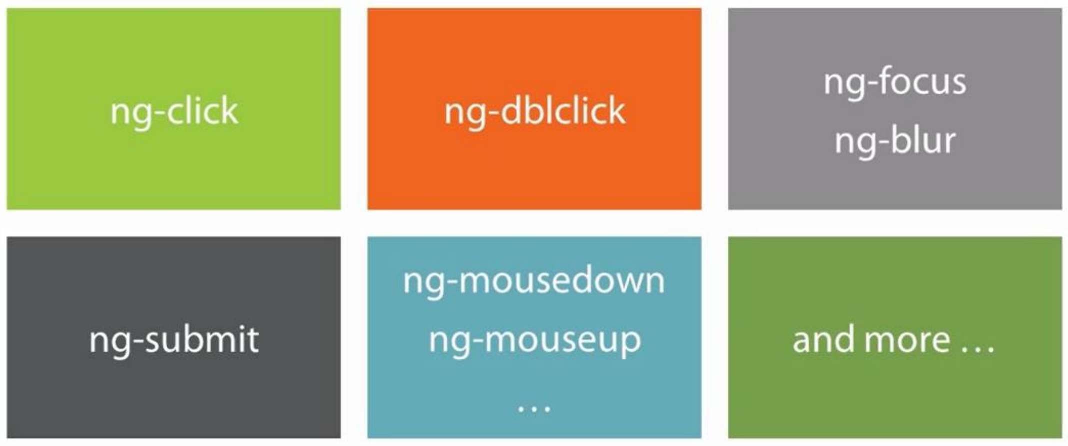

Assigning a Model in Markup

```
<input type="text" name="state" ng-model="countryCtrl.newState">
<a href>Add state {{countryCtrl.newState}}</a>
```

Data Access in the Controller

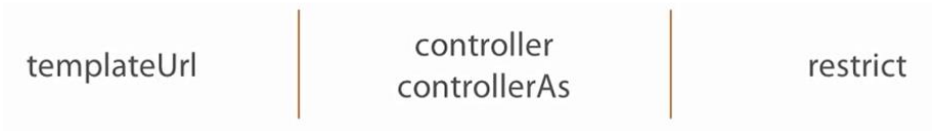
```
this.newState = "";
```

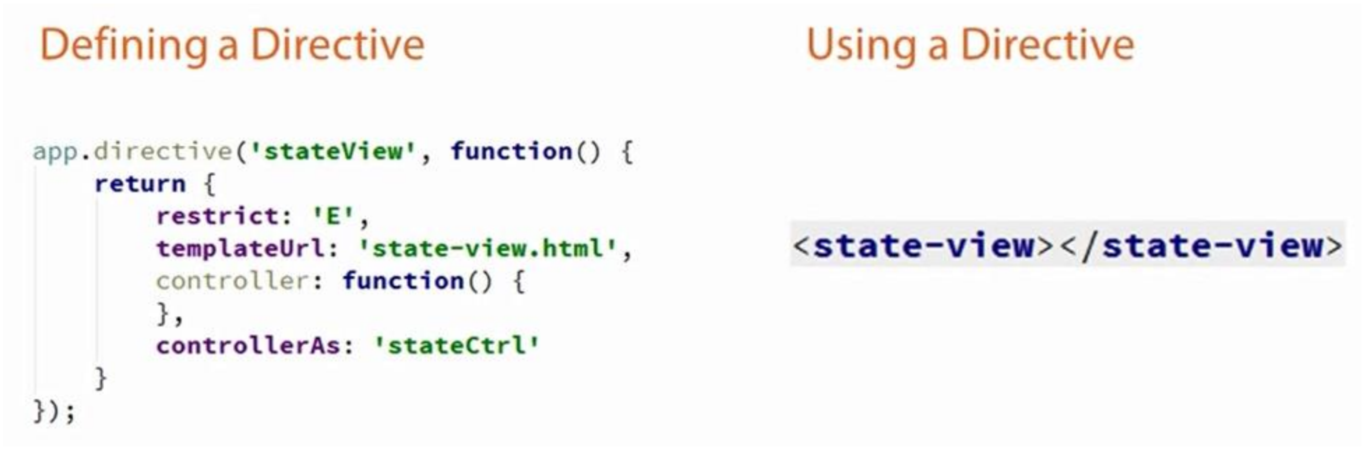
Event Handling Directives



Custom Directives

Reusable HTML tags that reference some code as well as templates.



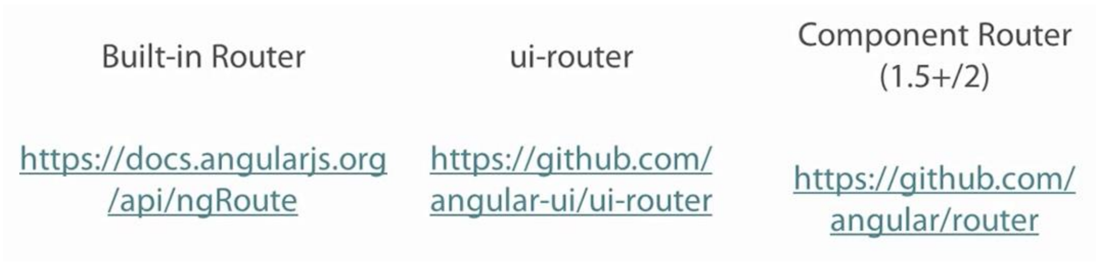

Defining a Directive

```
app.directive('stateView', function() {
    return {
        restrict: 'E',
        templateUrl: 'state-view.html',
        controller: function() {
        },
        controllerAs: 'stateCtrl'
    }
});
```

Using a Directive

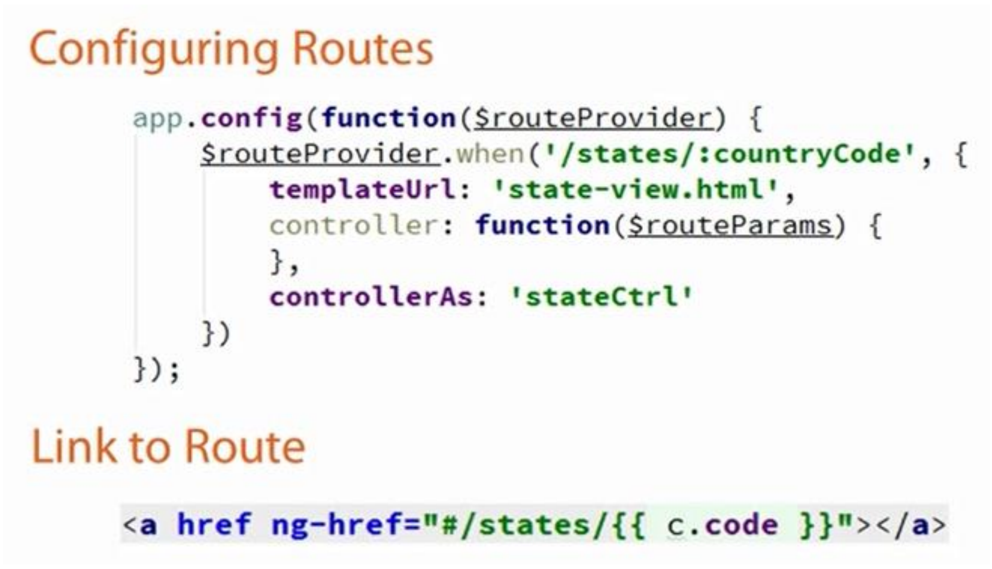
```
<state-view></state-view>
```

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

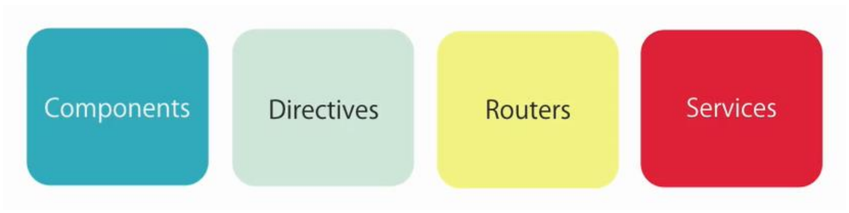
```
app.config(function($routeProvider) {
    $routeProvider.when('/states/:countryCode', {
        templateUrl: 'state-view.html',
        controller: function($routeParams) {
        },
        controllerAs: 'stateCtrl'
    })
});
```

Link to Route

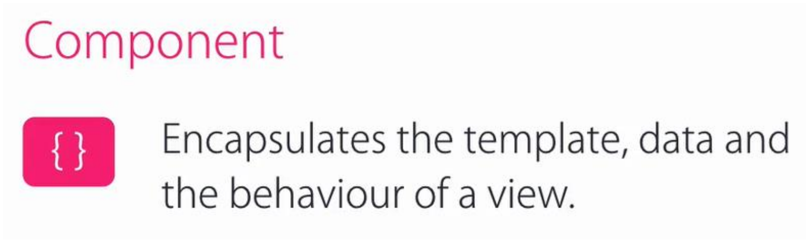
```
<a href ng-href="#/states/{{ c.code }}"></a>
```

Angular 2 – Mosh Hammedani

Architecture of Angular 2 Apps

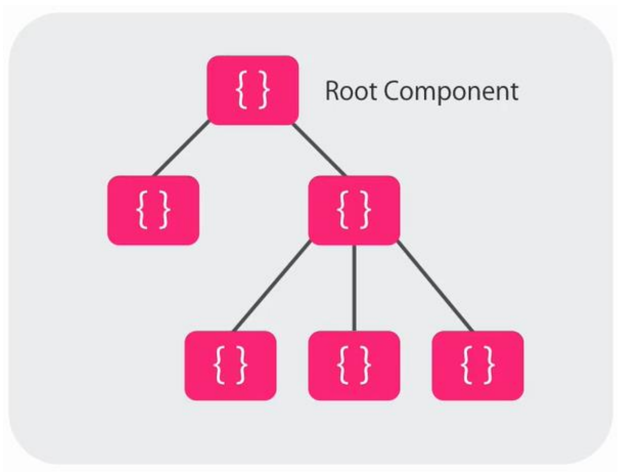


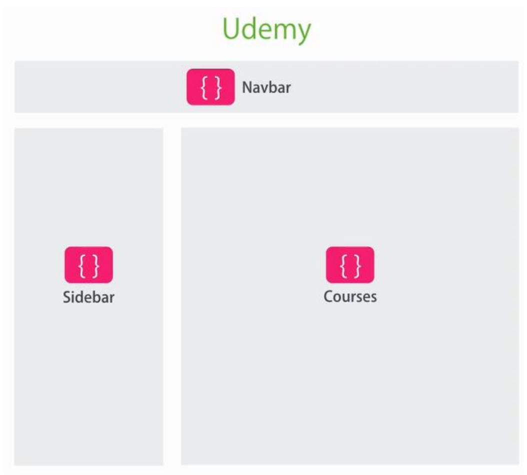
Components



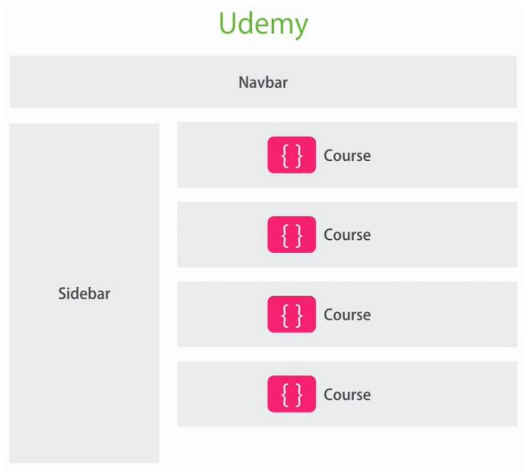
Component

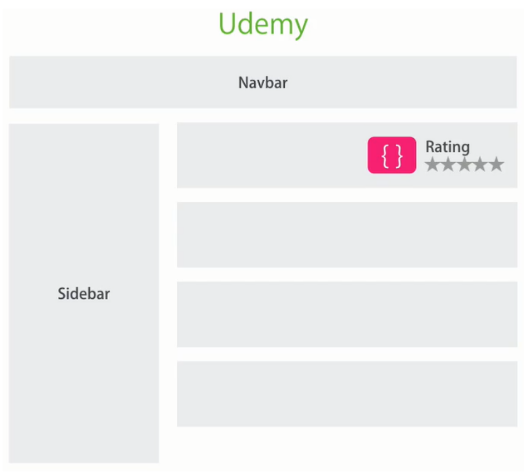
Encapsulates the template, data and the behaviour of a view.

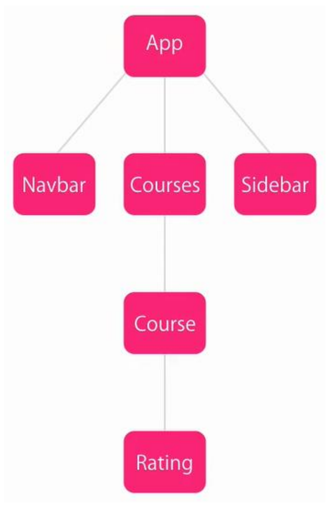




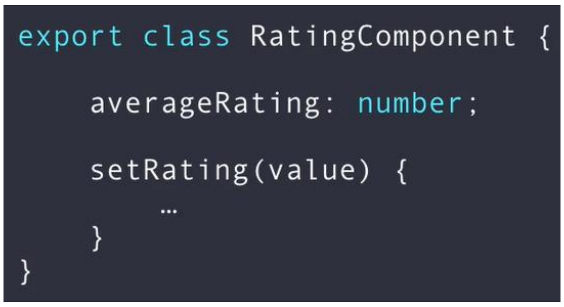
Each component can be comprised of multiple components and so on.







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.

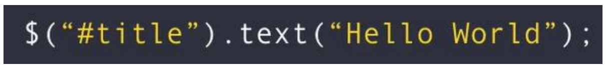


```
export class RatingComponent {

    averageRating: number;

    setRating(value) {
        …
    }
}
```

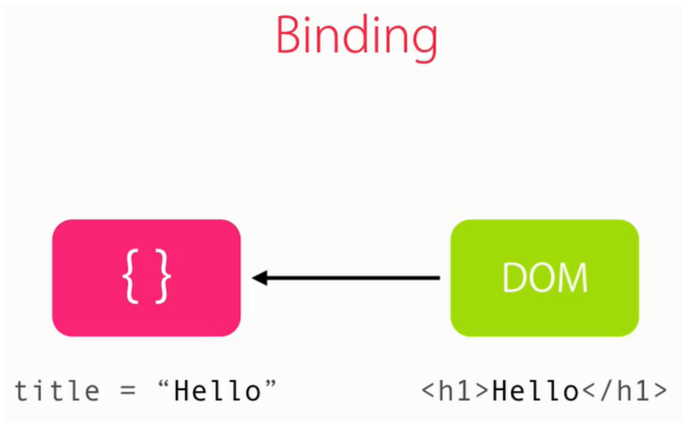
A component is nothing but a plain TypeScript class. The properties hold the data for the view whie the methods implement the behavior of the view; like what happens when a button is clicked.

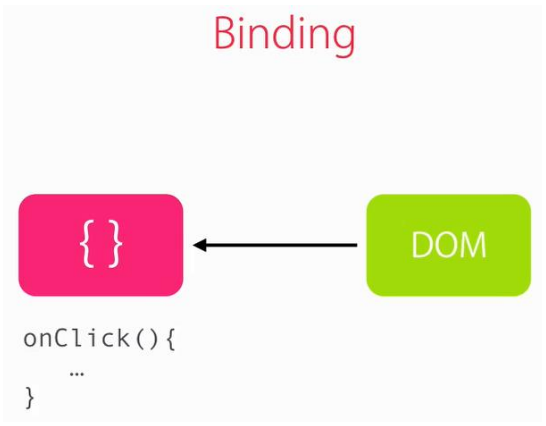
The components are COMPLETELY decoupled from the DOM. Instead of modifying DOM elements with each change, binding is used. In the view, we bind to the methods and properties of our component. To handle an event raised from a DOM element by a click, we bind that element to a method in our component.

The reason for the decoupling is for making the components unit testable.



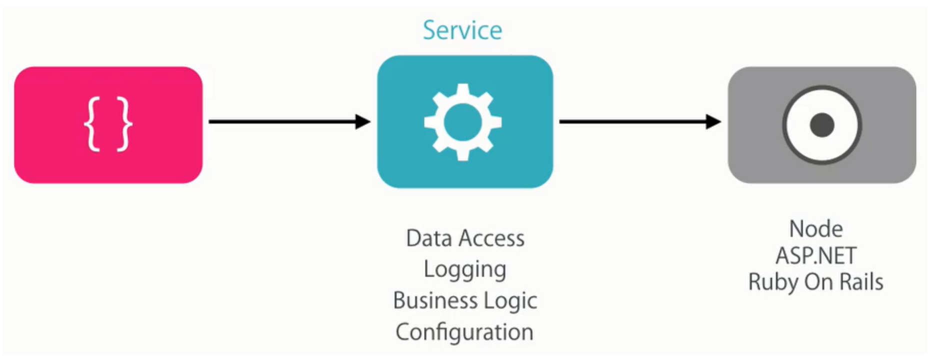
```
$("#title").text("Hello World");
```





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

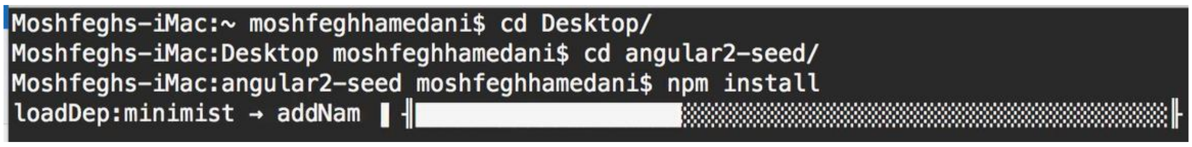
Similar to components, we use directives to work with the DOM. The directive, unlike a component, doesn't have the template or HTML markup for a view. They are often used to add behavior to existing DOM elements.

Directive
D
To modify DOM elements and/or extend their behaviour.
Angular

```
<input type="text" autoGrow />
```

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

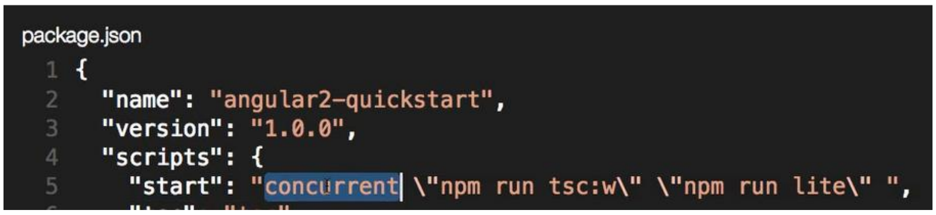


```
Moshfeghs-iMac:~ moshfeghhamedani$ cd Desktop/
Moshfeghs-iMac:Desktop moshfeghhamedani$ cd angular2-seed/
Moshfeghs-iMac:angular2-seed moshfeghhamedani$ npm install
loadDep:minimist → addNam ▌╢████████████████░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░╟
```

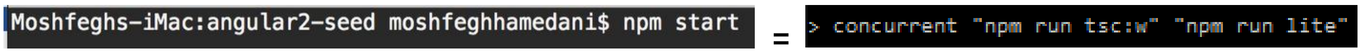
Navigate to the application directory and install node (dependencies) by using **npm install**. In the project in package.json under scripts there are some custom node commands. The **start** command is a shortcut for **concurrently** running two commands.

- **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.

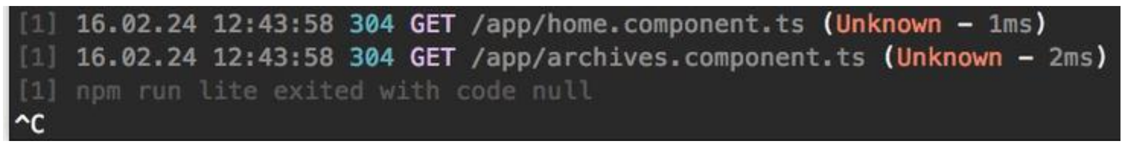


```
package.json
{
  "name": "angular2-quickstart",
  "version": "1.0.0",
  "scripts": {
    "start": "concurrent \"npm run tsc:w\" \"npm run lite\" ",
```



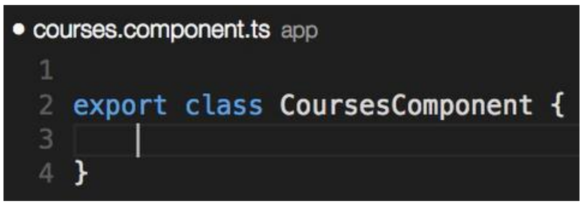
```
Moshfeghs-iMac:angular2-seed moshfeghhamedani$ npm start
```
=
```
> concurrent "npm run tsc:w" "npm run lite"
```

If the server terminates on its own, run this command **^C** and start it up again with **npm start**.



```
[1] 16.02.24 12:43:58 304 GET /app/home.component.ts (Unknown - 1ms)
[1] 16.02.24 12:43:58 304 GET /app/archives.component.ts (Unknown - 2ms)
[1] npm run lite exited with code null
^C
```

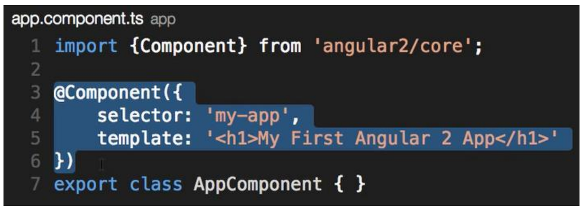
TypeScript



```
● courses.component.ts app

export class CoursesComponent {

}
```

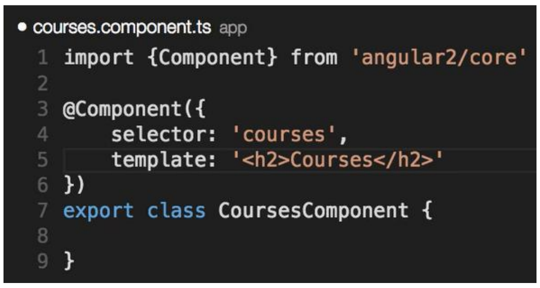
In TypeScript, each file is considered a module. In each module, we export one or more things, like a class, a function or a variable. The **export** keyword makes them available to other modules in the application. Later we can **import** said classes as needed. To make a class a component, we need to add a decorator.

Decorators / Annotations



```
app.component.ts app
import {Component} from 'angular2/core';

@Component({
    selector: 'my-app',
    template: '<h1>My First Angular 2 App</h1>'
})
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.



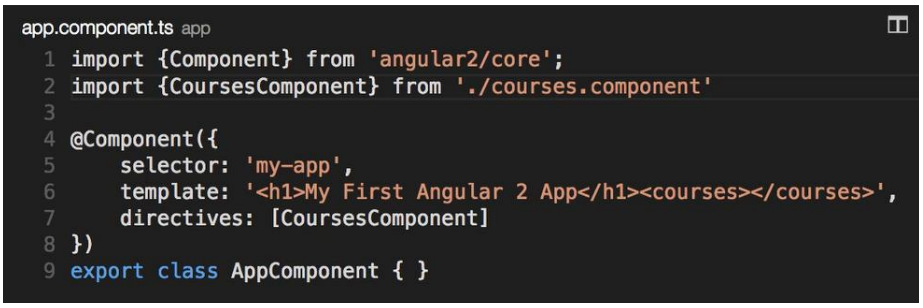
```
● courses.component.ts app
import {Component} from 'angular2/core'

@Component({
    selector: 'courses',
    template: '<h2>Courses</h2>'
})
export class CoursesComponent {

}
```

The decorator “component” is a function that needs to be imported from the angular2

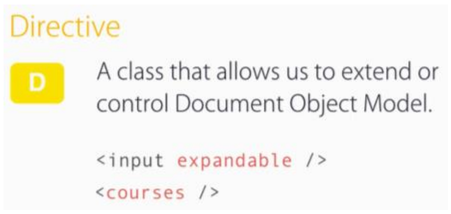
module. All decorators must be prefixed with an @ sign and then called. The function takes an object for which attributes are selected.

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
import {Component} from 'angular2/core';
import {CoursesComponent} from './courses.component'

@Component({
    selector: 'my-app',
    template: '<h1>My First Angular 2 App</h1><courses></courses>',
    directives: [CoursesComponent]
})
export class AppComponent { }
```

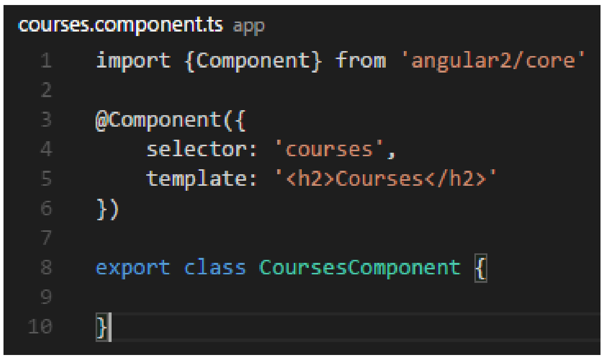


Directive

D A class that allows us to extend or control Document Object Model.

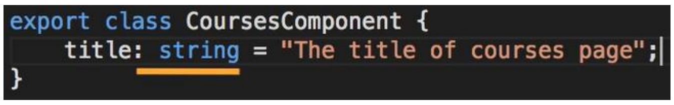
```
<input expandable />
<courses />
```

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
import {Component} from 'angular2/core'

@Component({
    selector: 'courses',
    template: '<h2>Courses</h2>'
})

export class CoursesComponent {

}
```

A component encapsulates the data and logic behind a view. We can define properties in our component and display them in the template.

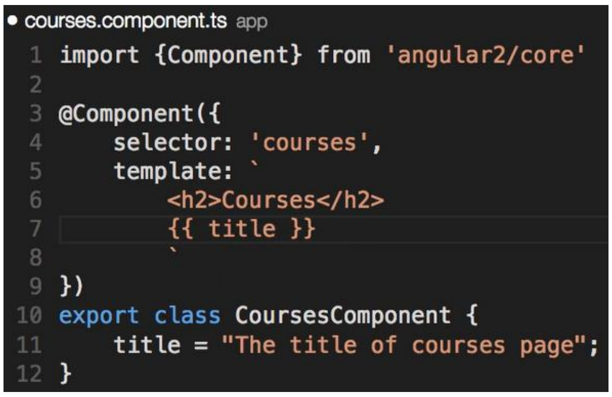


```
export class CoursesComponent {
    title: string = "The title of courses page";
}
```

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.

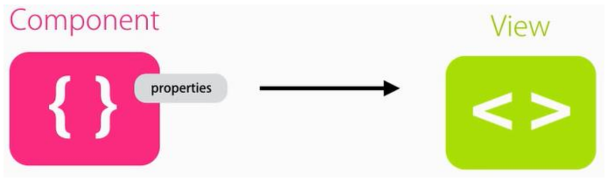
Interpolation

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.

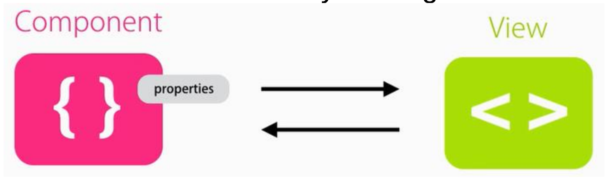


courses.component.ts app

```
import {Component} from 'angular2/core'

@Component({
    selector: 'courses',
    template: `
        <h2>Courses</h2>
        {{ title }}
        `
})
export class CoursesComponent {
    title = "The title of courses page";
}
```

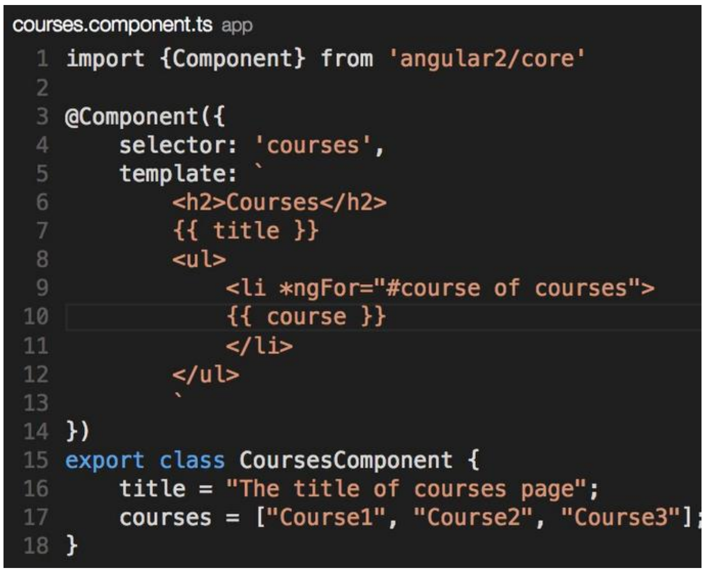


One way binding



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.

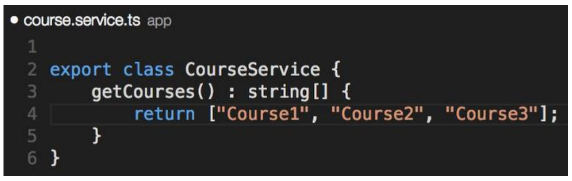
Every iteration will hold one course at a time.

That course can be displayed by {{ course }}

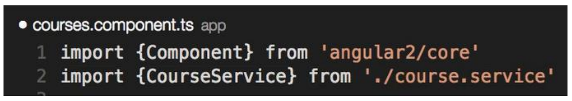
Services

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 shouuld be named like **serviceName.service.ts** and it is a simple class.

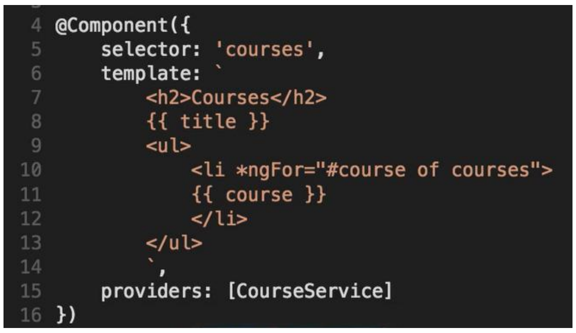
This service returns a list and it simulates a response from a RESTful API.


course.service.ts app

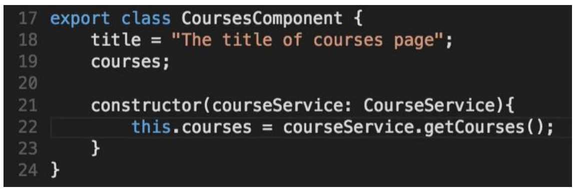
```
export class CourseService {
    getCourses() : string[] {
        return ["Course1", "Course2", "Course3"];
    }
}
```


courses.component.ts app

```
import {Component} from 'angular2/core'
import {CourseService} from './course.service'
```



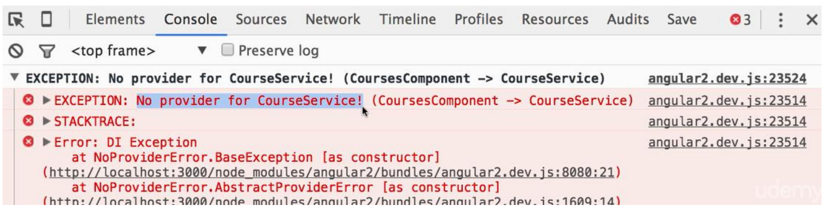
```
@Component({
    selector: 'courses',
    template: `
        <h2>Courses</h2>
        {{ title }}
        <ul>
            <li *ngFor="#course of courses">
            {{ course }}
            </li>
        </ul>
        `,
    providers: [CourseService]
})
```



```
export class CoursesComponent {
    title = "The title of courses page";
    courses;

    constructor(courseService: CourseService){
        this.courses = courseService.getCourses();
    }
}
```

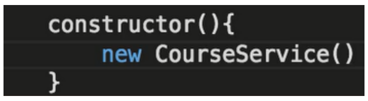
The service needs to be imported first. **./** means start from the current directory.

Providers: [CourseService] solves a common "**No provider for**" problem in Angular 2 apps.



```
▼ EXCEPTION: No provider for CourseService! (CoursesComponent -> CourseService)   angular2.dev.js:23524
  ▶ EXCEPTION: No provider for CourseService! (CoursesComponent -> CourseService)  angular2.dev.js:23514
  ▶ STACKTRACE:                                                                     angular2.dev.js:23514
  ▶ Error: DI Exception                                                             angular2.dev.js:23514
        at NoProviderError.BaseException [as constructor]
    (http://localhost:3000/node_modules/angular2/bundles/angular2.dev.js:8080:21)
        at NoProviderError.AbstractProviderError [as constructor]
    (http://localhost:3000/node_modules/angular2/bundles/angular2.dev.js:1609:14)
```

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

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

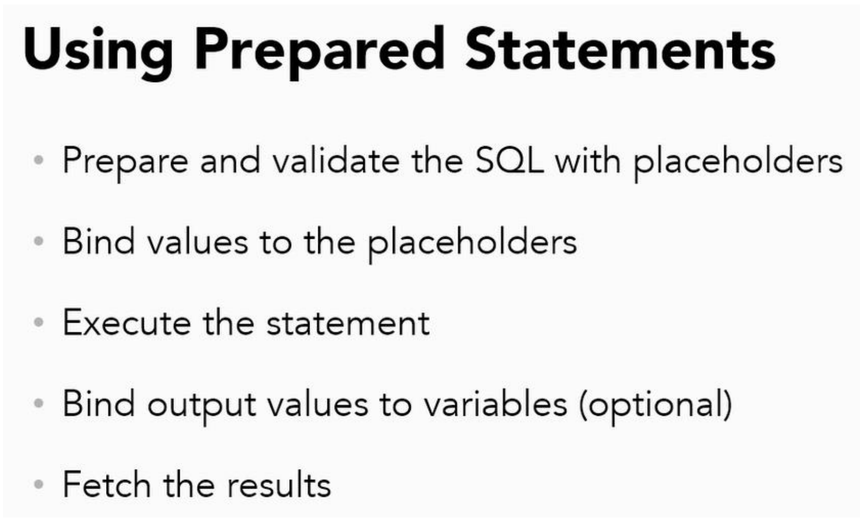
- Template for SQL query that uses values from user input
- Placeholders for values stored in variables
- Prevents SQL injection
- More efficient when same query is reused
- Can bind results to named variables

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 = ?';
```

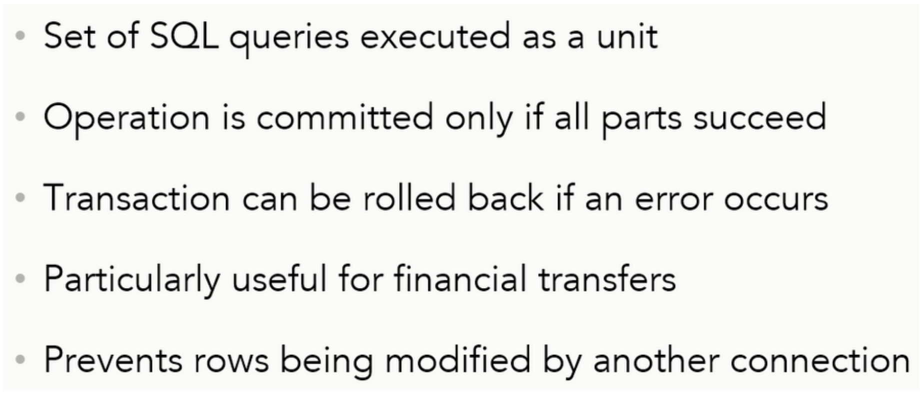
```
$sql = 'SELECT user_id, first_name, last_name
        FROM users
        WHERE username = :username AND password = :pwd';
```



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

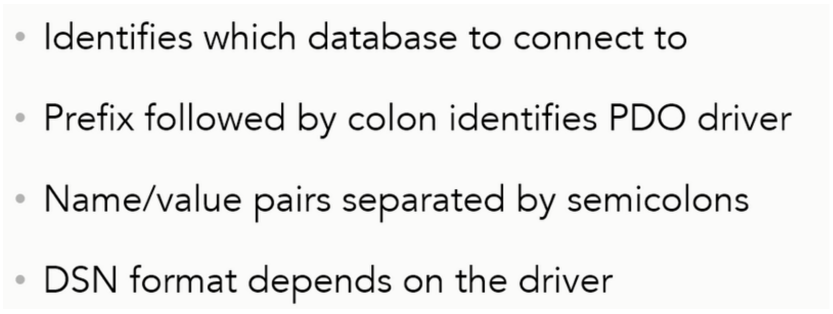
Transactions



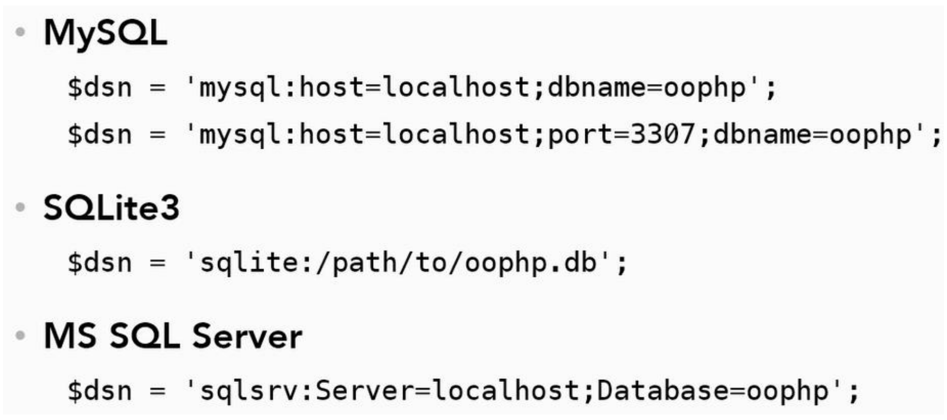
- Set of SQL queries executed as a unit
- Operation is committed only if all parts succeed
- Transaction can be rolled back if an error occurs
- Particularly useful for financial transfers
- Prevents rows being modified by another connection

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**

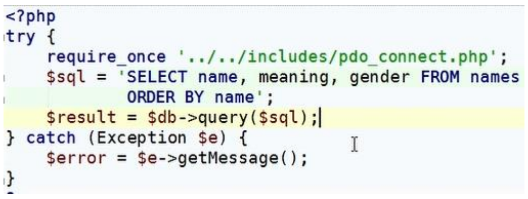
```
$dsn = 'sqlite:/path/to/oophp.db';
```

- **MS SQL Server**

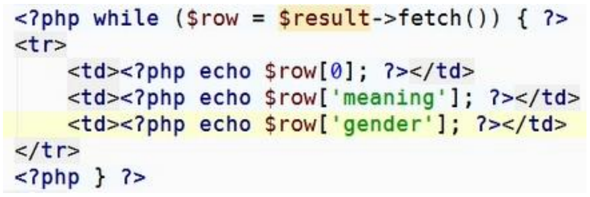
```
$dsn = 'sqlsrv:Server=localhost;Database=oophp';
```

Fetching Results (4 ways)

- `fetch()` gets the next row from a result set

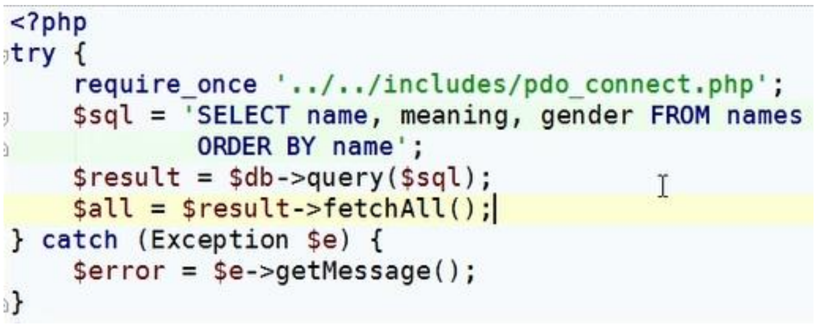


```
<?php
try {
    require_once '../../includes/pdo_connect.php';
    $sql = 'SELECT name, meaning, gender FROM names
            ORDER BY name';
    $result = $db->query($sql);
} catch (Exception $e) {
    $error = $e->getMessage();
}
```

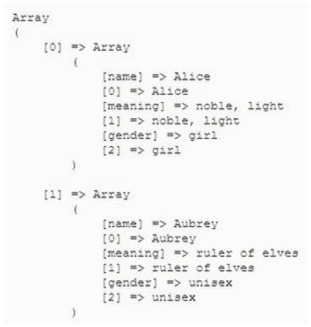


```
<?php while ($row = $result->fetch()) { ?>
<tr>
    <td><?php echo $row[0]; ?></td>
    <td><?php echo $row['meaning']; ?></td>
    <td><?php echo $row['gender']; ?></td>
</tr>
<?php } ?>
```

- `fetchAll()` creates an array containing all rows



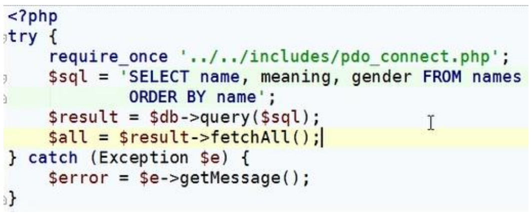
```
<?php
try {
    require_once '../../includes/pdo_connect.php';
    $sql = 'SELECT name, meaning, gender FROM names
            ORDER BY name';
    $result = $db->query($sql);
    $all = $result->fetchAll();
} catch (Exception $e) {
    $error = $e->getMessage();
}
```



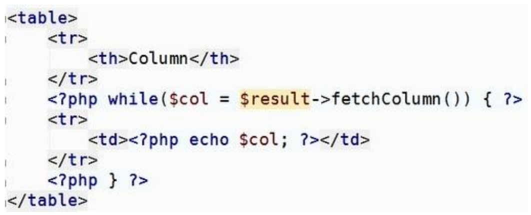
```
Array
(
    [0] => Array
        (
            [name] => Alice
            [0] => Alice
            [meaning] => noble, light
            [1] => noble, light
            [gender] => girl
            [2] => girl
        )

    [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



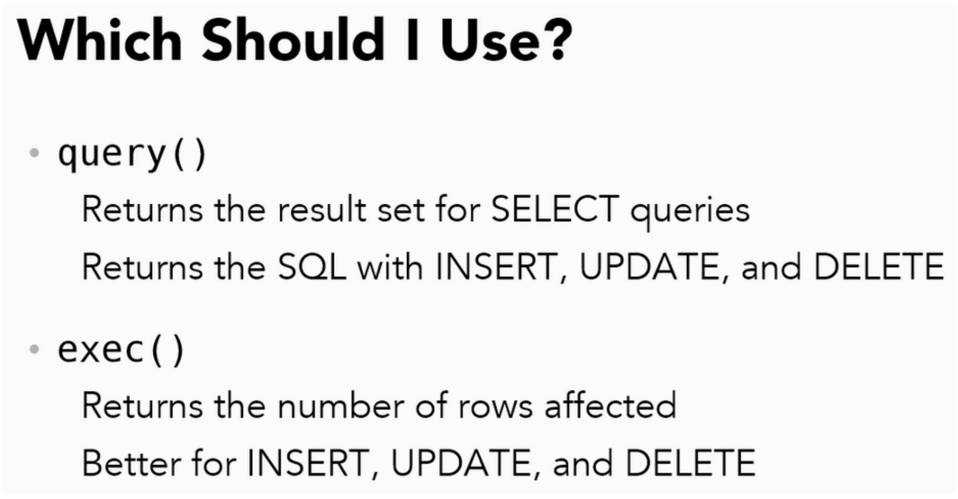
```
<?php
try {
    require_once '../../includes/pdo_connect.php';
    $sql = 'SELECT name, meaning, gender FROM names
            ORDER BY name';
    $result = $db->query($sql);
    $all = $result->fetchAll();
} catch (Exception $e) {
    $error = $e->getMessage();
}
```



```
<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

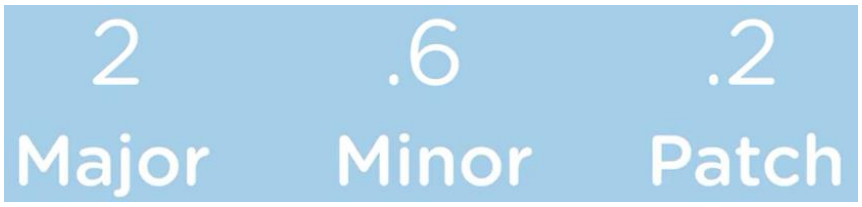
Query vs Exec

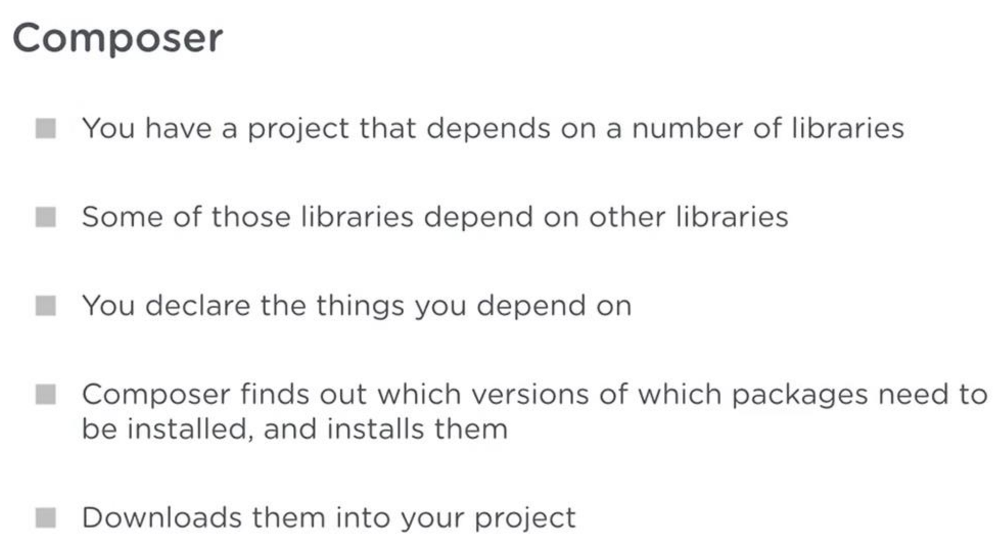


Which Should I Use?

- `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

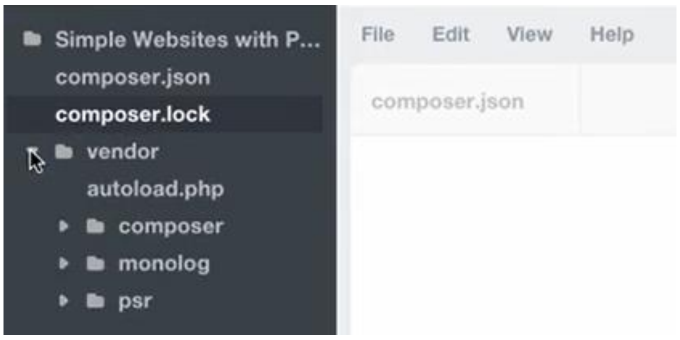
Slim Framework





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

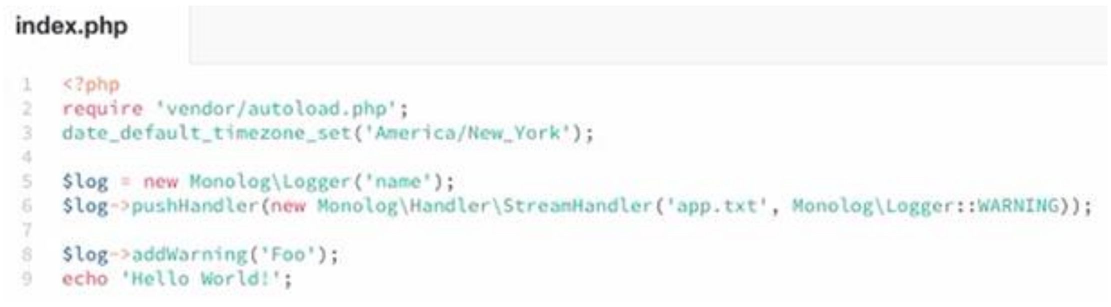


Everything in the vendor folder is managed by composer.json

Autoload.php fixes the messy usage of include at the top of every page when doing OOP.

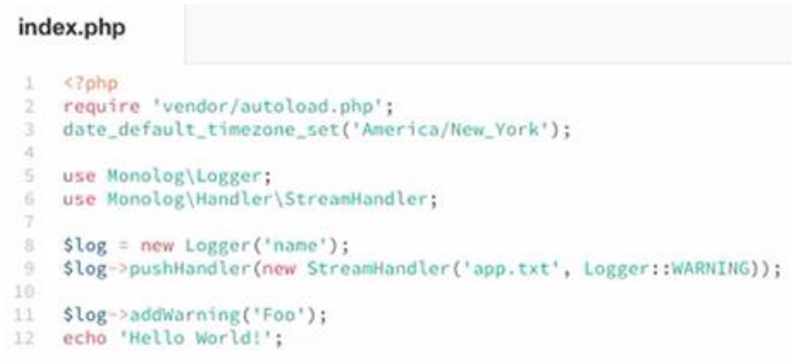
Namespacing

Without Namespacing


index.php

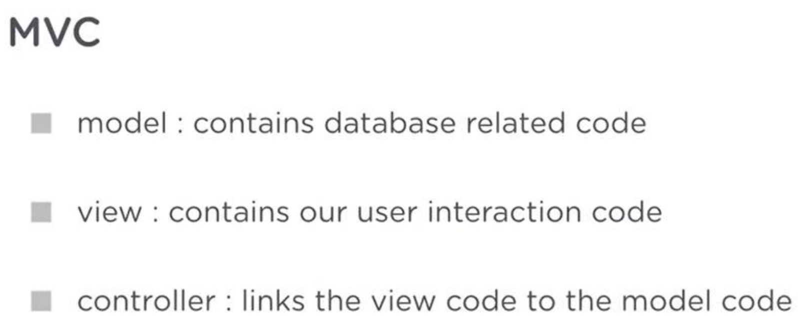
```php
<?php
require 'vendor/autoload.php';
date_default_timezone_set('America/New_York');

$log = new Monolog\Logger('name');
$log->pushHandler(new Monolog\Handler\StreamHandler('app.txt', Monolog\Logger::WARNING));

$log->addWarning('Foo');
echo 'Hello World!';
```

With Namespacing

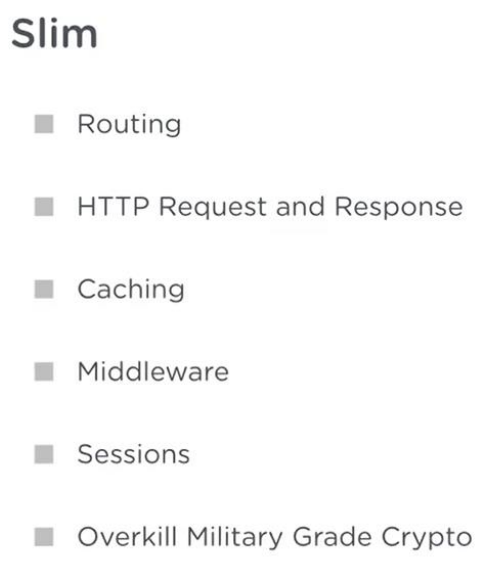

index.php

```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

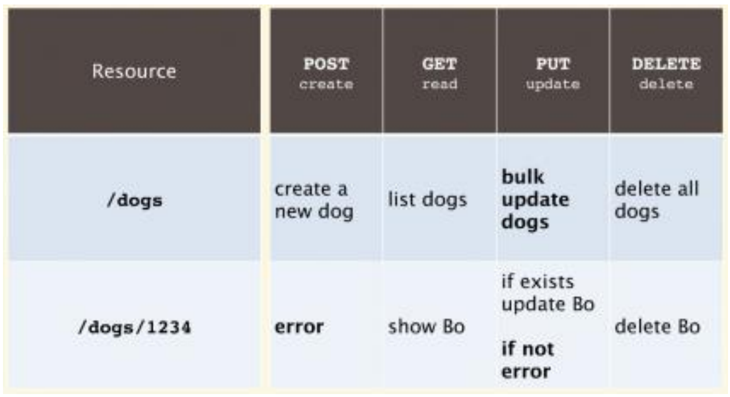
Restart all services

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.

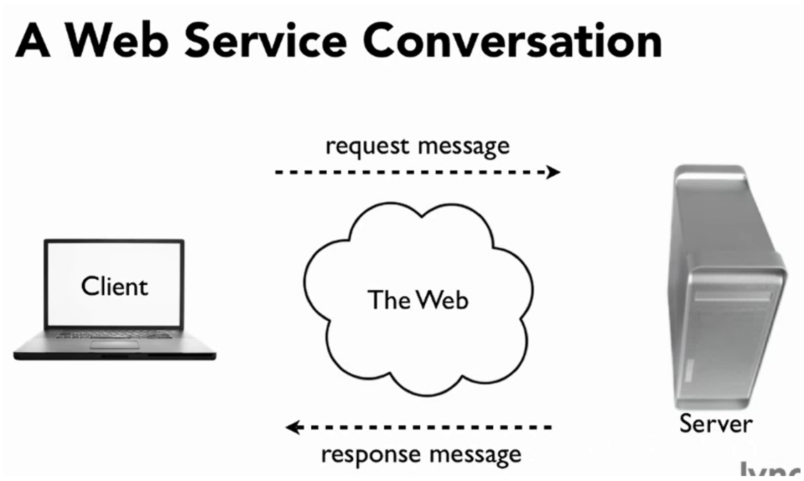
HTTP status codes

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.


A Web Service Conversation
request message
Client
The Web
Server
response message