CSWD

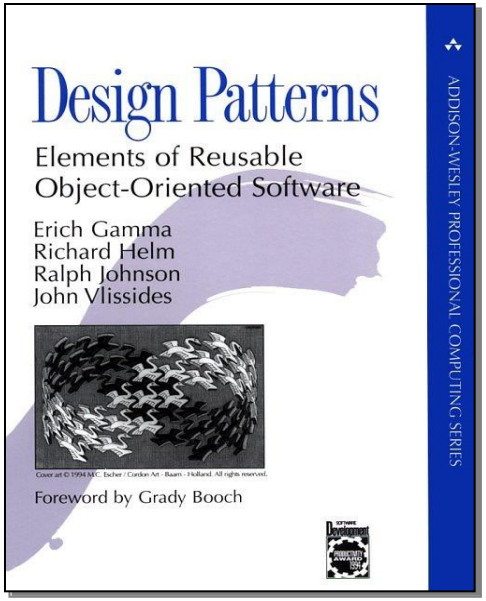
Client Side Web Development

Week 7

Design Patterns …

A pattern in the broader sense of the word is a "theme of recurring events or objects … it can be a template or model which can be used to generate things"

(http://en.wikipedia.org/wiki/Pattern)



CSWD

Overview

Part I: Motivation & Concept

- the issue
- what design patterns are
- what they're good for
- how we develop & categorize them

Part II: Application

- use patterns
- demonstrate usage & benefits

Part III: Wrap-Up

- observations, caveats, & conclusion

Part I: Motivation & Concept

- **Good OO designers rely on lots of experience**
  - **At least as important as syntax, sometime more so …**
- **Most powerful reuse is *design* reuse**
  - **Where match problems based on design experience**
- **OO systems exhibit recurring structures that promote**
  - **Abstraction - flexibility**
  - **Modularity - elegance**

Problem:
How to capture, communicate, & apply this knowledge?

Therein lies valuable design knowledge

Part I: Motivation & Concept (cont'd)

A *Design Pattern*…

- **abstracts a recurring design structure**
- **comprises class and/or object**
  - **dependencies**
  - **structures**
  - **interactions**
  - **Conventions**
- **names & specifies the design structure explicitly**
- **they are a guideline, implementation must be grounded to the specific problem**

Part I: Motivation & Concept (cont'd)

Goals

Codify good design

- distill & generalize experience
- aid to novices & experts alike

Give design structures explicit names

- common vocabulary
- reduced complexity
- greater expressiveness

Facilitate restructuring/refactoring

- patterns are interrelated
- additional flexibility

CSWD

Patterns & Frameworks

- **Frameworks: another option to reuse existing architecture**
  - **something that provides you a frame to be filled!**
- **Reuse of existing software objects that just need to be properly configured**
- **Bound to a specific technology (rigid)**
  - **Require training**
  - **High cost of switch**
  - **Level of customization not always acceptable/available**
  - **Eg Angular, Ember, Express, Meteor, BackBone …**

CSWD

Patterns, Architectures & Frameworks

- There can be confusion between patterns, architectures and frameworks.
- Let's try to distinguish them:
  - Architectures model software structure at the highest possible level, and give the overall system view.
    - » An architecture can use many different patterns in different components.
  - Patterns are more like small-scale or local architectures for architectural components or sub-components
  - Frameworks are partially completed software systems that may be targeted at a particular type of application.
    - » These are tailored by completing the unfinished components.

Summary of Differences

- **Patterns are more general and abstract than frameworks.**
  - **A pattern is a description of a solution, not a solution itself.**
- **A pattern cannot be directly implemented.**
  - **An implementation is an example of a pattern.**
- **Patterns are more primitive than frameworks.**
  - **A framework can employ several patterns.**

Algorithms are not design patterns

- **Algorithms are not design patterns because they have different objectives.**
- **The purpose of an algorithm is to solve a specific problem (sorting, searching, etc.) in a computationally efficient way as measured in terms of *time* and *space* complexity.**
- **The purpose of a design pattern is to organize code in a developmentally efficient way as measured in terms of *flexibility*, *maintainability*, *reusability*, etc.**

One-off designs are not patterns

- **Not every software design rises to the level of a pattern.**
- **Patterns are often described as *reusable* and *well-proven* solutions.**
  - **You can't be sure a one-off design is reusable or well-proven.**
- **The general rule-of-thumb is a software design can't be considered a pattern until it has been applied in a real-world solution at least three times (the so called "Rule of Three").**

1977
Christopher Alexander coauthors the first book on patterns, *A Pattern Language: Towns, Buildings, Construction*. It documents 253 patterns in urban planning and building architecture.

1994
The "Gang of Four" launch the software patterns movement with the publication of the first book of software patterns: *Design Patterns: Elements of Reusable Object-Oriented Software*. It documents 23 mid-level software design patterns.

1987
Kent Beck and Ward Cunningham suggest creating a pattern language for software design in the spirit of Christopher Alexander's pattern language for town and building design.

Important milestones in the history of design patterns

CSWD

The Beginning of Patterns

- **Christopher Alexander, architect**
  - **A Pattern Language--Towns, Buildings, Construction**
  - **Timeless Way of Building (1979)**
  - **"Each pattern describes a *problem* which occurs over and over again in our environment, and then describes the core of the *solution* to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."**
- **Other familiar patterns:**
  - **novels (tragic, romantic, crime), movies genres (drama, comedy, documentary)**

Alexander's Patterns

Five parts:

Name: short familiar, descriptive name or phrase

usually indicative of the solution

Example: illustrate prototypical application

pictures, diagrams, and/or descriptions

Context: situations in which the pattern applies

Problem: relevant forces, constraints, interactions

Solution: relationships and rules to construct artifacts

often listing several variants

What do you need to change for software?

History of software design patterns

- **In 1987 *Kent Beck* and *Ward Cunningham* proposed creating a pattern language for software design.**
- **Their original vision was to empower users "to write their own programs"**

  ***"Our initial success using a pattern language for user interface design has left us quite enthusiastic about the possibilities for computer users designing and programming their own applications."* [Sowizral]**

"Gang of Four" (GoF) Book

- **Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1994**
- **Written by the "gang of four"**
  - **Dr. Erich Gamma, then Software Engineer, Taligent, Inc.**
  - **Dr. Richard Helm, then Senior Technology Consultant, DMR Group**
  - **Dr. Ralph Johnson, then and now at University of Illinois, Computer Science Department**
  - **Dr. John Vlissides, then a researcher at IBM**

Object-Oriented Design Patterns

- **Their book defined 23 patterns in *three categories***
  - ***Creational patterns* deal with the process of creating an object in a flexible way**
    - » **Separates creation from use**
  - ***Structural patterns* deal primarily with the static composition and structure of classes and objects to deal with constructs like inheritance, polymorphism**
  - ***Behavioural patterns* deal primarily with dynamic interaction among classes and objects**
    - » **Suggests relationships and patterns of communication**

GoF Patterns

- *Creational Patterns*
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- *Structural Patterns*
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy
- *Behavioural Patterns*
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

CSWD

Patterns Exist at All Levels

- Machine code
- Assemblers
- High Level Languages
- Abstract Data Types (queues, stacks)
- Objects
- Software Architectures

- Even Plenty of Web Usability patterns

Why Study Patterns?

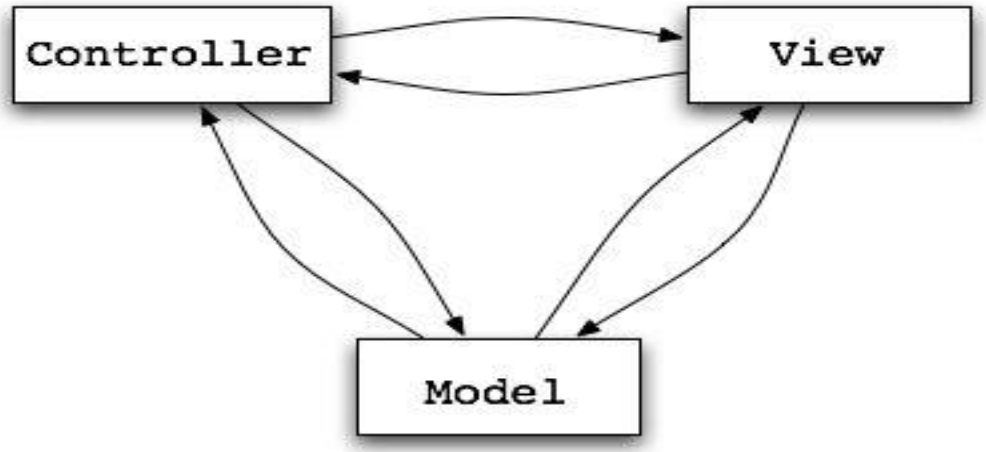
- **Reuse tried, proven solutions**
  - **Provides a head start**
  - **Avoids gotchas later (unanticipated things)**
  - **No need to reinvent the wheel**
- **Establish common terminology**
  - **Design patterns provide a common point of reference**
  - **Easier to say, "We could use Strategy here."**
- **Provide a higher level prospective**
  - **Frees us from dealing with the details too early**

Other advantages

- **Most design patterns make software more modifiable, less brittle**
  - **we are using time tested solutions**
- **Using design patterns makes software systems easier to change—more maintainable**
- **Helps increase the understanding of basic object-oriented design principles**
  - **encapsulation, inheritance, interfaces, polymorphism**

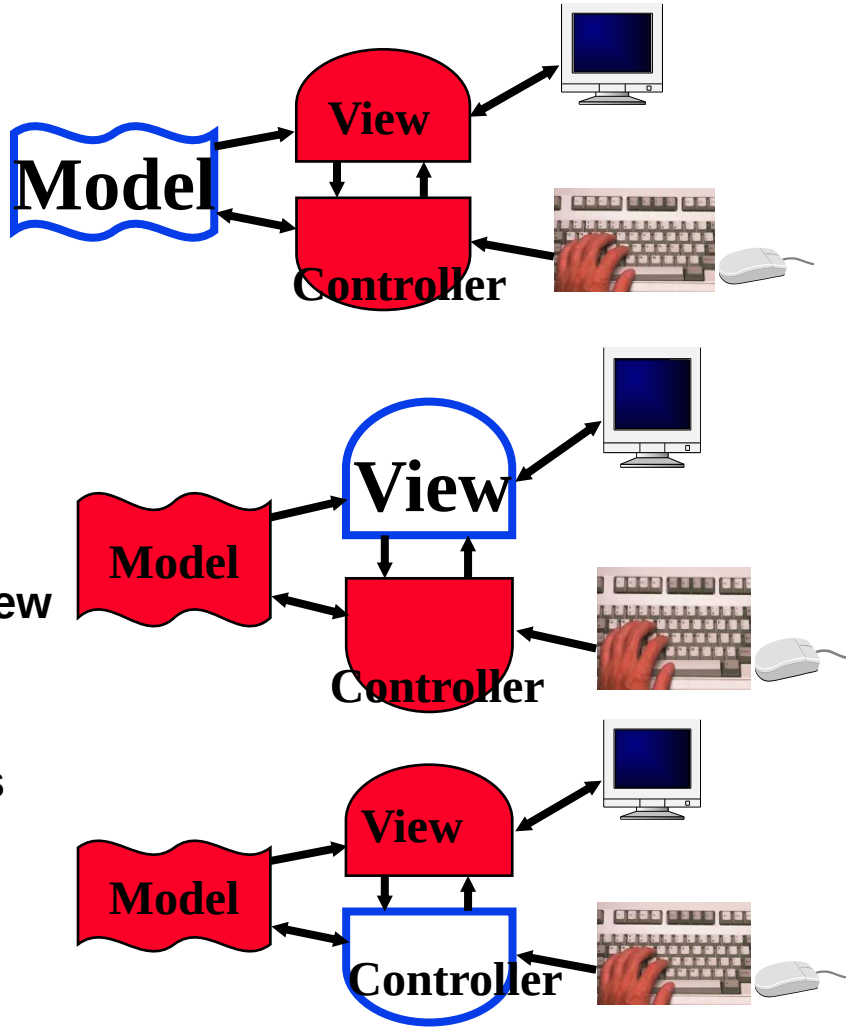
Example: Model-View-Controller Architecture

- Architectural Pattern from Smalltalk (1979)
- Decouples data and presentation
- Eases the development by partitioning application
  - scalable
  - maintainable



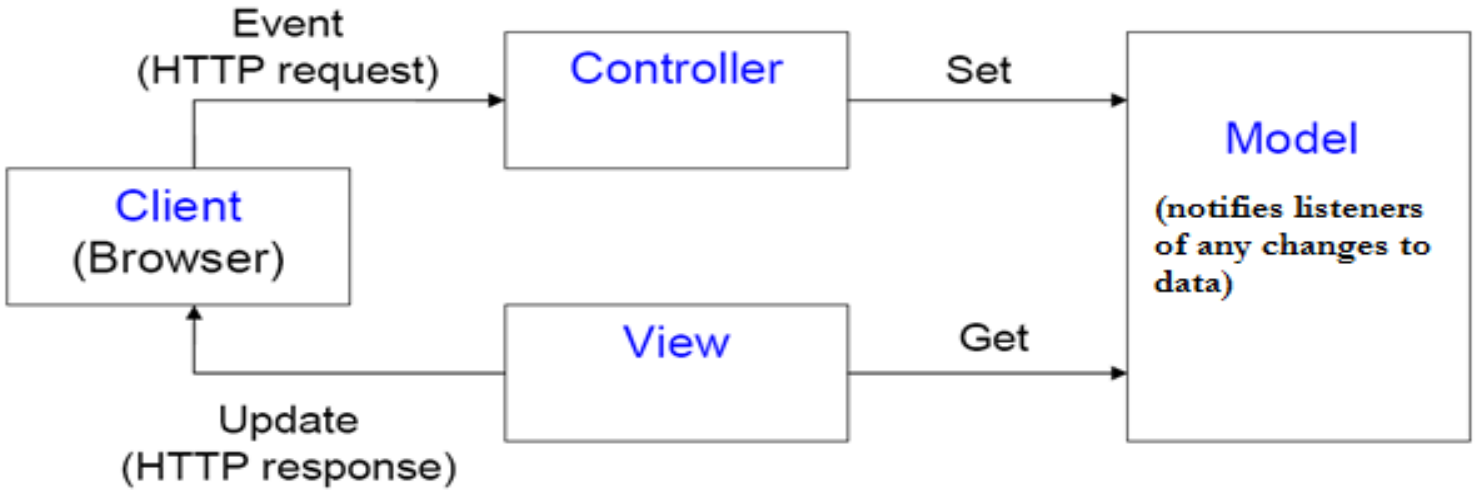
Model View Controller

- **Model**
  - Representation of real-world data
  - encapsulates application state
  - responds to state queries
  - exposes application functionality
  - notifies views of changes
- **View**
  - renders the models
  - requests updates from models
  - sends user interaction to controller
  - allows controller to select correct view
- **Controller**
  - defines application behavior
  - maps user actions to model updates
  - selects view for response
  - one for each functionality



Model-View-Controller 2 (MVC 2)

- **Adaptation of MVC for the Web**
  - **stateless connection between the client and the server**
  - **notification of view changes**
  - **re-querying the server to discover modification of application's state**

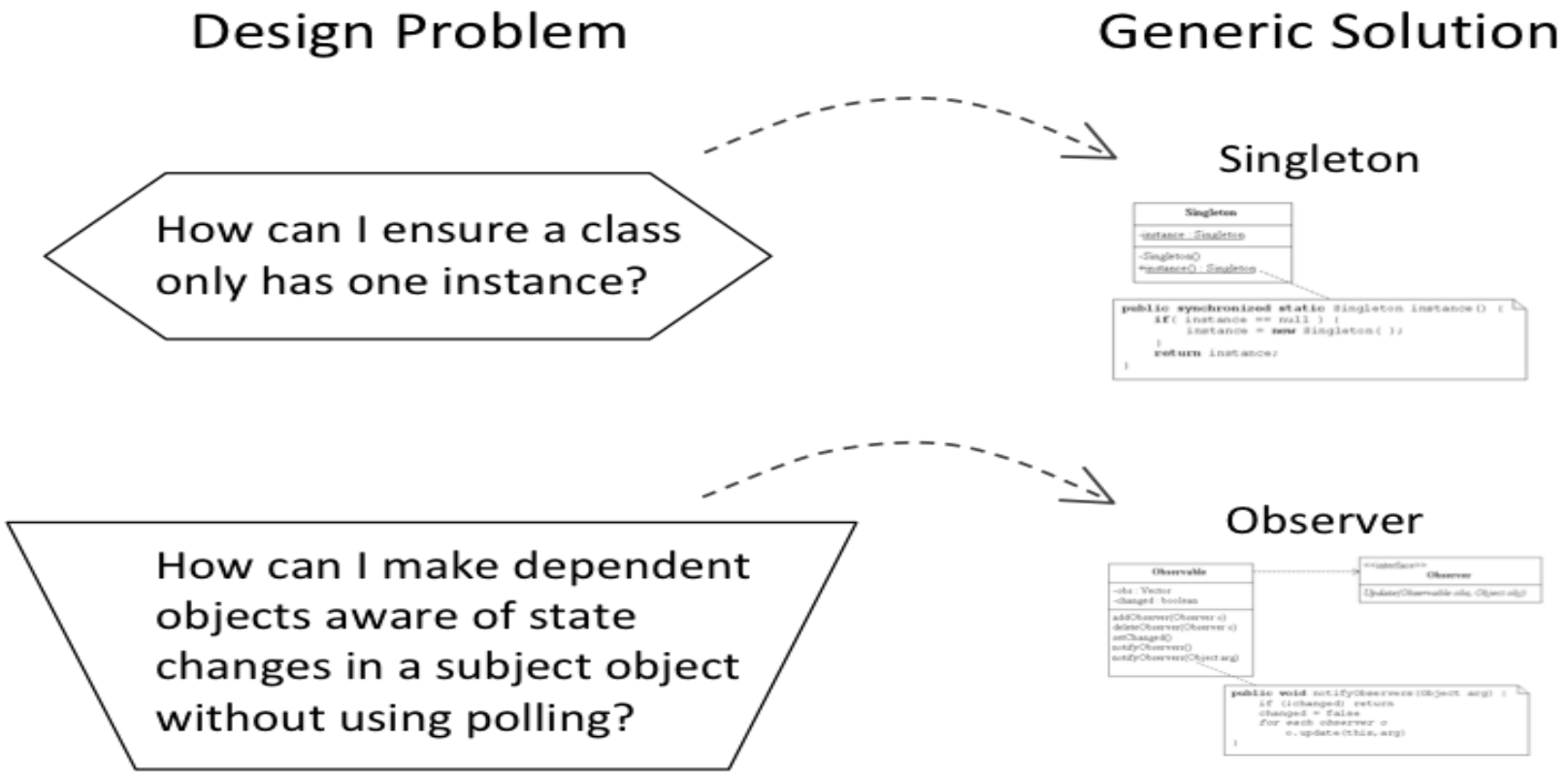


Why MVC?

- Combining MVC into one class or using global variables will not scale. Why?
  - model may have more than one view
    - » each different & needing update on model changes
- Separation eases maintenance. Why?
  - easy to add a new view later
    - » may need new model info, but old views still work
  - can change a view later
    - » e.g., draw shapes in 3-d
      - recall that the view handles selection

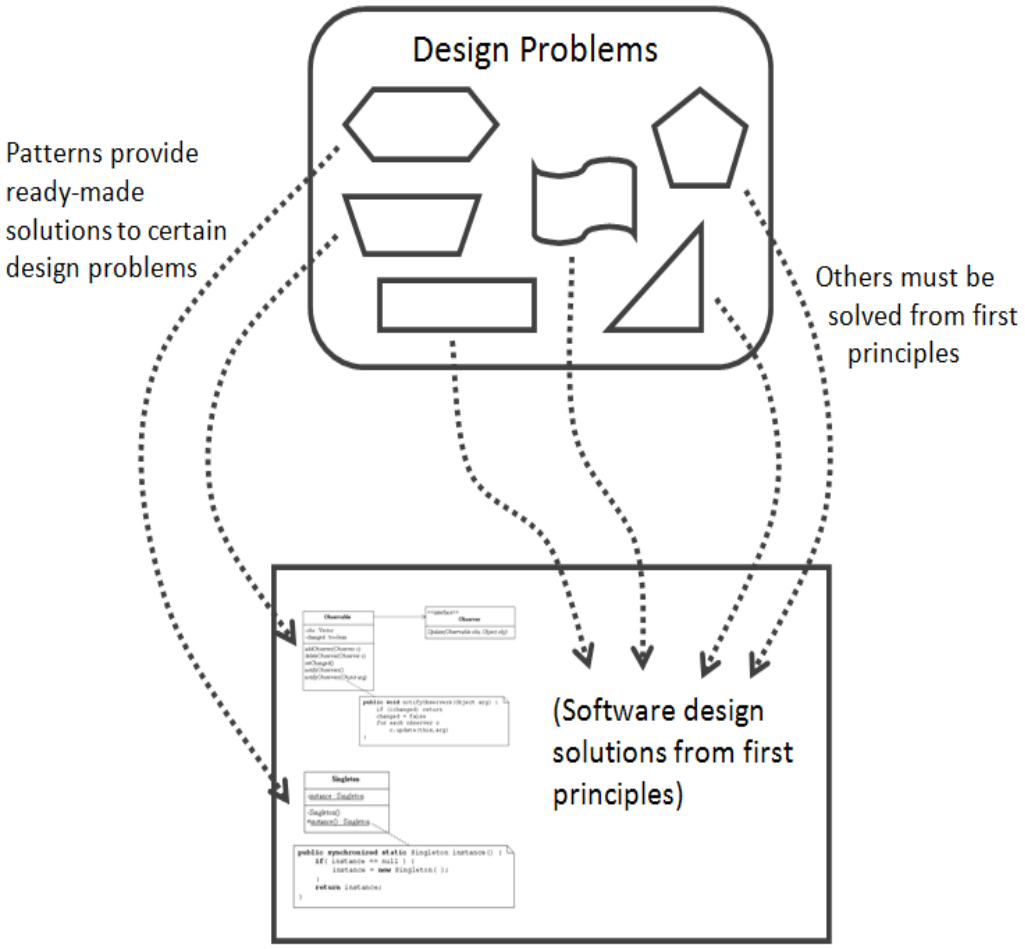
Design Patterns as Problem → Solution Pairs

- **Conceptually, a design pattern provides a mapping from a specific design problem to a generic solution**



Design Patterns Defined [Cont.]

- **Knowledge of design patterns simplifies software design by reducing the number of design problems that have to be solved from first principles.**
- **Design problems that match documented design patterns, have ready-made solutions.**
- **The remaining problems that don't match documented design patterns must be solved from first principles.**



CSWD

Style for Describing Patterns

- **Often see this structure:**
  - ***Pattern name***
  - ***Recurring problem:*** **what problem the pattern addresses**
  - ***Solution:*** **the general approach of the pattern**
  - ***UML for the pattern***
    - » ***Participants:*** **a description as a class diagram**
  - ***Use Example(s):*** **examples of this pattern, in JavaScript**

CSWD

A few OO Design Patterns

- **Coming up:**
  - **Constructor & Factory**
    - » **allows objects to be created in a robust manner**
  - **Module & Revealing Module**
    - » **provides the tools to create self-contained decoupled pieces of code, which can be treated as black boxes of functionality and added, replaced, or removed according to the (ever-changing) requirements of the software you're writing.**
  - **Iterator**
    - » **access the elements of an aggregate object sequentially without exposing its underlying representation**
  - **Singleton**
    - » **ensures that not more than one instance of a class/object is created and provides a global point of access to this instance**
  - **Observer**
    - » **One object stores a list of observers that are updated when the state of the object is changed**

Constructor

- **Already seen can create objects in JavaScript; refer to previous lecture for examples**

```
var myObject= new Object();
var myObject= { };
```

- **When you just want to create an object there's no benefit of creating "constructor-based" objects using "new" operator.**
  - **It's same as creating an object using "object literal" syntax.**
- **However, "constructor-based" objects created with the "new" operator come to the fore when you are thinking about "*prototypal inheritance*".**
- **You cannot maintain inheritance chain(s) with objects created with literal syntax.**
  - **But you can create a constructor function, attach properties and methods to its prototype.**

CSWD

Constructor – object initializer

- **Using object initializers is sometimes referred to as creating objects with literal notation.**
  - **"Object initializer" is consistent with the terminology used by C++.**
- **The syntax for an object using an object initializer is:**

```
var myObject = { property1: value1, … propertyN: valueN };
```

  - **where `myObject` is the name of the new object, each property_i is an identifier (either a name, a number, or a string literal), and each value_i is an expression whose value is assigned to the property_i.**
- **Object initializers are *expressions*, and each object initializer results in a new object being created whenever the statement in which it appears is executed.**
- **Identical object initializers create distinct objects that will not compare to each other as equal.**
- **Objects are created as if a call to new Object() were made**
  - **eg**

```
var as = {firstname:"Adam", surname:"Smith"};
```

CSWD

Constructor functions

- **ES5 JavaScript doesn't have classes but has constructor functions invoked with new :**

```
Eg var as = new Person("Adam", "Smith");
```

- **Because constructors are still just functions, it helps if you can tell, just by looking at a function name, whether it was supposed to behave as a *constructor* or as a *normal* function.**
- **Naming constructors with a capital first letter provides that hint.**
  - **Using lowercase for functions and methods indicates that they are not supposed to be called with new :**

```
function MyConstructor() {...}

function myFunction() {...}
```

  - **there are some patterns that enable you to programmatically force your constructors to behave like constructors, but simply following the naming convention is helpful in itself, at least for programmers reading the source code.**

CSWD

Constructor functions cont.

- Eg

```
function Person(firstname, surname) {
    this.firstname = firstname;
    this.surname = surname;
}
Person.prototype.fullname = function() {
    console.log( this.firstname + ' ' +
                 this.surname);
}
var adamSmith = new Person('Adam', 'Smith');
var janeDoe = new Person('Jane', 'Doe');
adamSmith.fullname();
janeDoe.fullname();
```

Prototypical inheritance

you can create as many objects as you want by instantiating Person constructor function and all of them will inherit fullname()

Q2. Define class for Car with make, model and year properties and a display method.

CSWD

Constructor functions cont.

- **NB if you wanted to obtain the same functionality with "object literal" syntax, you would have to create fullname() on all of the objects like below:**

```
var janeDoe = {
    firstname: 'Jane',
    lastname: 'Doe',
    fullname: function() {
        console.log(this.firstname + ' ' + this.lastname);
    }
};
var johnDoe = {
    firstname: 'John',
    lastname: 'Doe',
    fullname: function() {
        console.log(this.firstname + ' ' + this.lastname);
    }
};
janeDoe.fullname();
johnDoe.fullname();
```

Factory

- The Factory pattern is another creational pattern concerned with the notion of creating objects.
-
- Where it differs from the other patterns in its category is that it doesn't explicitly require us to use a constructor.

- Definition: a Factory can provide a generic interface for creating objects, where we can specify the type of factory object we wish to be created.

CSWD

Factory cont.

```
// Define a skeleton vehicle factory
function VehicleFactory() {}

// Define the prototypes and utilities for this factory, assuming
// default vehicleClass is Car
VehicleFactory.prototype.vehicleClass = Car; // see earlier defn

// Our Factory method for creating new Vehicle instances
VehicleFactory.prototype.createVehicle = function ( options ) {

  switch(options.vehicleType){
    case "car":
      this.vehicleClass = Car;
      break;
    case "van":
      this.vehicleClass = Van;
      break;
    //defaults to VehicleFactory.prototype.vehicleClass (Car)
  }

  return new this.vehicleClass( options );
};
```

CSWD

Factory cont.

```
// Create an instance of our factory that makes cars
var carFactory = new VehicleFactory();
var car = carFactory.createVehicle( {
            vehicleType: "car",
            make: "Ford",
            model: "Capri",
            year: "1992" } );

// Test to confirm our car was created using the vehicleClass/
prototype Car

// Outputs: true
console.log( car instanceof Car );

// Outputs: Car object of male "Ford", model: "Capri" year "1992"
console.log( car );
```

Q2. You can define a class for Van with colour, state and insured properties. Remember to test!

Module Pattern

- Modules are an integral piece of any robust application's architecture and typically help in keeping the units of code for a project both cleanly separated and organized.
- The Module pattern was originally defined as a way to provide both private and public encapsulation for classes in conventional software engineering.
- In JavaScript, the Module pattern is used to further emulate the concept of classes in such a way that we're able to include both public/private methods and variables inside a single object, thus shielding particular parts from the global scope.
  - What this results in is a reduction in the likelihood of our function names conflicting with other functions defined in additional scripts on the page.
  - Sounds Good!

CSWD

Module Pattern cont.

- The Module pattern encapsulates "privacy", state and organization using closures. .
- It provides a way of wrapping a mix of public and private methods and variables, protecting pieces from leaking into the global scope and accidentally colliding with another developer's interface.
- With this pattern, only a public API is returned, keeping everything else within the closure private.
- Modules should be Immediately-Invoked-Function-Expressions (IIFE) to allow for private scopes - that is, a closure that protects variables and methods.

- NB it will return an *object* instead of a *function*

CSWD

Module Pattern cont.

```
(function() {

    // declare private variables and / or
    // functions

    return {
        // declare public variables and / or
        // functions
    }

})();
```

IIFE template

CSWD

Module Pattern - example

```
var htmlChanger = (function() {
    var contents = "Hello World!";
    var changeHTML = function () {
        var el = document.getElementsByTagName
                    ('body')[0];
        el.innerHTML = contents;
    }

    return {
       callChangeHTML: function() {
                          changeHTML();
                          console.log(contents);
                       }
     }
})();
htmlChanger.callChangeHTML();// "Hello World!"
console.log(htmlChanger.contents);  // undefined
```

Revealing Module Pattern

- **A variation of the module pattern is called the Revealing Module Pattern – *Christian Heilman***
- **The purpose is to maintain encapsulation and reveal certain variables and methods returned in an object literal.**

CSWD

Revealing Module Pattern - example

```
var htmlChanger = (function() {
    var h = "Hello ", w = "World!", contents = h;

    var changeHTML = function () {
        contents = h;
        var el = document.getElementsByTagName('body')[0];
        el.innerHTML = contents;
    }

    var sayHello = function () {
                    contents = h + w;
                    changeHTML();
                    console.log(contents);
                }
    return {
       first: changeHTML,
       second: sayHello
    }
})();
htmlChanger.first(); // "Hello"
htmlChanger.second(); // "Hello World!"
console.log(htmlChanger.changeHTML());  // undefined
```

Revealing Module Pattern cont

- **Advantages**
  - **This pattern allows the syntax of our scripts to be more consistent.**
  - **It also makes it more clear at the end of the module which of our functions and variables may be accessed publicly which eases readability.**
- **Disadvantages**
  - **A disadvantage of this pattern is that if a private function refers to a public function, that public function can't be overridden if a patch/update is necessary.**
    - » **This is because the private function will continue to refer to the private implementation and the pattern doesn't apply to public members, only to functions.**
  - **Public object members which refer to private variables are also subject to the no-patch rule notes above.**
  - **As a result of this, modules created with the Revealing Module pattern may be more fragile than those created with the original Module pattern, so care should be taken during usage.**

Summary

- Brief intro to theory behind patterns
- Practical examples of:
  - Constructor
  - Factory
  - Prototype
  - Module
  - Revealing Module

Check out:
Learning JavaScript Design Patterns
Book by Addy Osmani

Practice, Practice, Practice …

  - The Module and Revealing Module patterns are expected as part of the Coursework but the others will also be used.
- Next: Part 2, more patterns …