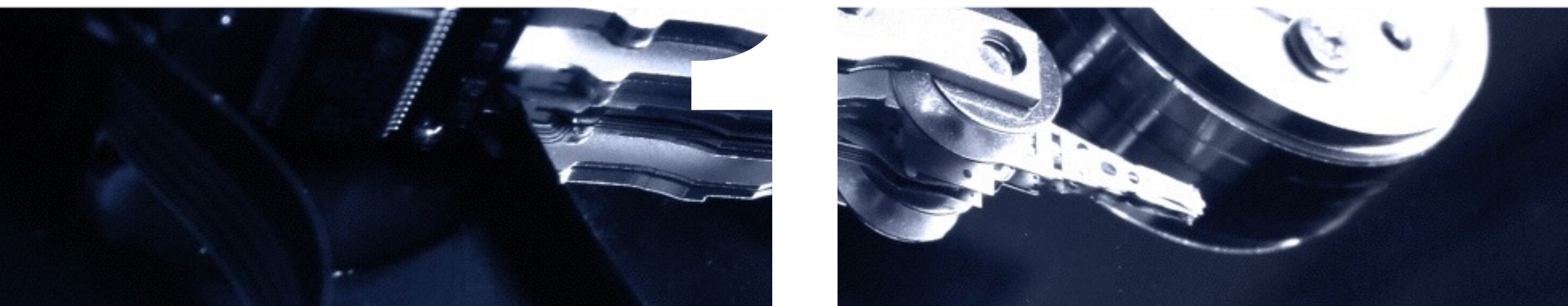




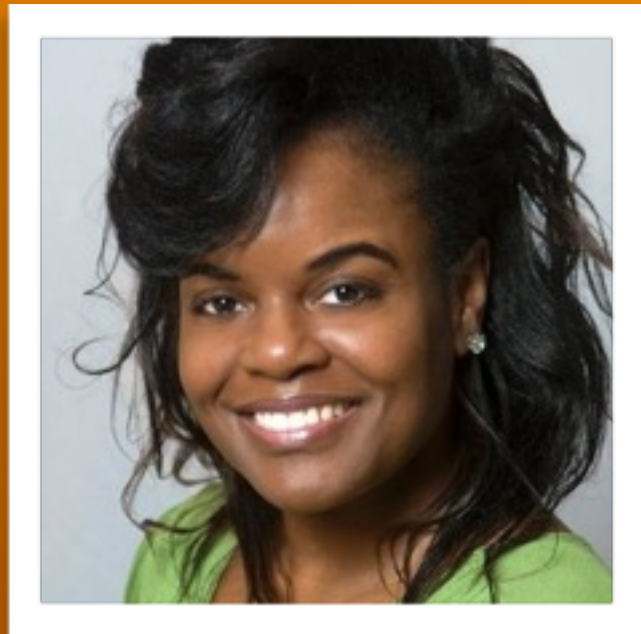
FULL SAIL
UNIVERSITY

web design and development



programming for web applications 1

courseMaterial.7



courseDirector

Fialishia O'Loughlin

foloughlin@fullsail.com

labSpecialist

Eric Silvay

esilvay@fullsail.com

courseMaterial.7
goal6.**Recap**

goal6.Recap

- ▶ review last assignment - mid term

courseMaterial.Objective

- ▶ **courseMaterial**

- ▶ *more.Objects*
 - ▶ *constructors*
 - ▶ *constructor context (“this”)*
 - ▶ *methods and functions in constructors*
 - ▶ *prototype*
- ▶ *practice all the new materials*

- ▶ **assignment**

- ▶ *fine tune the concepts from the course materials*

courseMaterial.7
more.**Objects**

more.Objects

▶ review of objects

- ▶ object's link variables and functions together inside of a storage container
- ▶ the variables and functions are called object members
- ▶ objects are just variables and functions combined into a single data structure
 - ▶ variables = properties (this is owned by the object)
 - ▶ function = method (this is owned by the object)
- ▶ functions placed within an object can access variables in the object w/o having to pass an argument

more.Objects

- ▶ custom objects = constructors
 - ▶ custom objects extend JavaScript to include features to suit your needs
 - ▶ classical inheritance (languages like C++) use classes to create objects - Javascript doesn't have classes, but we do have constructors
 - ▶ since object properties and methods have to be initialized when a object is created, a special method called a **constructor** is needed to get an object up and running
 - ▶ every custom object requires its own **constructor** which is named the same as the object name
 - ▶ the **constructor** is responsible for creating object instances

more.Objects

► constructors

- **constructors** are a special category of *functions*
- in syntax, they are no different than any other function
- what changes is the invocation (how the function is called and run)

```
//the normal non constructor way  
var Blog = function(){};  
var result = Blog();           //result becomes the return  
value
```

more.Objects

► constructors

- if a function is invoked using the keyword **new**, it is executed as a **constructor** - the constructor is called to initialize an object upon creation
- this causes the function to return an object, with properties inherited from the function itself

```
var Blog = function(){};  
var result = new Blog();           //result is an object
```

more.Objects

► constructors

- as a best practice, functions that are acting as **constructors** should be given a name starting with a capital letter, purely for making it easier to find in the code
- just like any other function, we can pass arguments into it

```
var Blog = function(str, date) {                                     //  
  constructor  
  //code goes here  
};  
  
var blog = new Blog("this is a string", 04/13/13); //instantiation
```

more.Objects

▶ the constructor

- ▶ the constructor's job is establishing the properties of an object, along with their initial values
- ▶ to create a property within a constructor, you set the property using the JavaScript keyword, **this**
- ▶ **this** assigns ownership of the property to the object, and sets it's initial value
- ▶ we are assigning a property that belongs to **this** object, as oppose to just a local variable within the constructor

more.Objects

▶ keyword: **this**

- ▶ the keyword **this** is essentially a reference to the object that owns the function it is being used in
- ▶ because it is a reference, any changes or updates made to **this** are reflected in the owner object
- ▶ using the **this** keyword, any properties or methods on the owner object are accessible as if **this** was the object itself

more.Objects

► constructor context

- **this** becomes a reference to the object that the constructor creates
- object properties are created and initialized in a constructor by using dot notation and the **this** keyword
- note that if we pass arguments into the constructor, they become part of the constructor's *scope*, as opposed to properties...

```
var Blog = function(str, date){  
    this.body = str;           //object property  
    this.date = date;         //object property  
};  
  
var blog = new Blog("this is a string", "04/13/13"); //  
instantiation
```


more.Objects

► constructor context

- note the difference between the argument and the property

```
var Blog = function(str, date){  
  this.body = str + '!!!';           //object property  
  this.date = date;                 //object property  
  
  alert(str);  
  
  alert(this.body);  
};  
  
var blog = new Blog("this is a string", "04/13/13"); //  
instantiation
```

this is a string

this is a string!!!

more.Objects

► constructor context

- similarly, any variables declared inside the constructor function's scope are considered local to the function (*cannot be accessed outside of it*)

```
var Blog = function(str, date){  
    var text = 'LOL';           //local variable ONLY  
    this.body = str;           //object property  
    this.date = date;          //object property  
};  
  
var blog = new Blog("this is a string", 04/13/13); //  
instantiation
```

more.Objects

► methods in constructors

- since **this** references the new object, we can create **methods** in that object as well
- **methods** will have access to both local variables and the object's **this**

```
var Blog = function(str, date){  
  var text = 'LOL';           //local variable ONLY  
  this.body = str;            //object property  
  this.date = date;           //object property  
  this.toHTML = function(){  
    alert( this.body );        //alert would be "this is a  
    string" alert( text );     //alert would be "LOL"  
  };  
};  
  
var blog = new Blog("this is a string", 04/13/13); //  
instantiation
```

more.Objects

► functions in constructors

- **functions** in a constructor are declared as normal, with the **var** keyword
- like local variables, they are accessible only inside the constructor...
- constructors are used to bring instances to life - therefore are NOT capable of creating a class properties - class properties must be created outside of the constructor

more.Objects

- ▶ removing object properties

- ▶ the only way to remove an object property is to use the *delete* operator

```
delete obj.property;  
  
//example  
delete chapter.title;
```

more.Objects

▶ prototype-inheritance

- ▶ we addressed earlier that objects inherit from other objects
- ▶ at the core of javascript, *all objects* inherit from a foundation, the **prototype object**
- ▶ the prototype object is an *automatic* part of every constructor, and is what all objects inherit from
- ▶ think of prototype as the cookie cutter that creates all objects in javascript

more.Objects

▶ prototype-inheritance

- ▶ every *constructor* in javascript is automatically given an internal *prototype object*
- ▶ whenever an object is created, it is inheriting from the constructor's prototype - this is referred to as *prototype chain*, and is what defines **prototypal inheritance**
- ▶ if our constructor was *Person* ...
 - ▶ **var** myObj = **new** *Person*();
- ▶ *myObj* is inheriting *Person.prototype*

more.Objects

▶ prototype-inheritance

- ▶ so what does this mean?
- ▶ any constructor's *prototype* property can be accessed and even modified
- ▶ we can access the **.prototype** property of our constructors to add new properties and methods on the fly, or change existing ones

```
var Person = function(name) {  
    this.name = name;  
};  
Person.prototype.newMethod = function( ) { } ;  
Person.prototype.newProp = "Im new!";  
Person.prototype.name = "James Bond";
```

more.Objects

▶ prototype-inheritance

- ▶ it is extremely important to note that any *methods* created using *.prototype* will NOT have access to local variables inside the constructor
- ▶ we can still utilize the **this** variable to reference *public* members:

```
var Person = function(name) {  
    this.name = name;  
    var species = "Human";  
};  
  
Person.prototype.sayHi = function(){  
    alert(this.name);           //works  
    alert(species);             //causes error  
};
```

more.Objects

► prototype-inheritance

- the next important concept is that any object created from a constructor will automatically inherit any **prototype** changes, even after being created

```
var Person = function(name) {  
    this.name = name;  
}  
  
var me = new Person("JamesBond");  
  
Person.prototype.sayHi = function( ) {  
    alert (this.name);  
};  
  
me.sayHi( ); //alerts "JamesBond"
```

more.Objects

▶ prototype-inheritance

- ▶ one efficient use of *prototype* we will see is to create a method based on what browser the user has
- ▶ by using the **prototype** object, we can do the browser check during the declaration
- ▶ we can also use prototype to extend javascript's existing constructors...

more.Objects

▶ prototype (using the Blog example)

- ▶ the methods in the Blog object were created inside of the constructor using the "this" keyword
- ▶ this approach works but it ends up creating a new copy of the methods for EVERY Blog object that is created - so if there are 100 entries, there are 100 copies of all the methods of the object - since it is just duplicating code and NOT values it is inefficient
- ▶ versus a property which normally has a unique value for each different object
- ▶ so, a better design would be for methods to share a single copy of each method and each instance of the object has access to the method
- ▶ we do this by understanding the following....

more.Objects

- ▶ **prototype (object class vs object instance)**
 - ▶ **Object class:**
 - ▶ is an object description - a template that outlines what an object is made of
 - ▶ describes the properties and methods of an object
 - ▶ like the blue print of a house
 - ▶ **Object instance:**
 - ▶ an actual object that has been created from a object class - like a specific house that was built from the house blue print (above)

more.Objects

► prototype

- **classes** in JavaScript are made possible by a hidden object called a "prototype" - it exists in every object as a property
- the **prototype object** allows us to **set properties and methods** that are owned at the class level as oppose to within an instance - this provides the mechanism for creating a class
- a "prototype" object is how we establish that a class owns a method OR a property

```
Blog.prototype.toHTML = function(){};  
  // Blog = class name  
  // prototype = prototype object - accessed as a property of the  
    class  
  //toHTML = name of the function
```

more.Objects

▶ prototype property

- ▶ **properties in a class** are similar to a class owned instance method - owned by the class with a **single copy** available for all instances
- ▶ the property in a class means the property only has one value that is shared by all instances - if we change the value, it changes the property for all instances
- ▶ an example might be a bloggers signature that is readily accessible to any instance that wants to access the blog authors signature

```
Blog.prototype.signature = 'James Bond';
```

```
// Blog = class name
```

```
// prototype = prototype object - accessed as a property of the  
class
```

```
//signature = property name
```

Assignment / Goal 7

- **Goal7: Assignment:**

- Log into FSO. This is where all your assignment files will be located as well as Rubrics and assignment instructions

- **Commit your completed work into GitHub**

- As part of your grade you will need at least 6 reasonable GIT commits for each assignment.

- **In FSO there is an announcement with “Course Schedule & Details” in the title, in that announcement you will see a “Schedule” link which has the due dates for assignments.**