DOWNLOAD PDF JAVASCRIPT OBJECT ORIENTED PROGRAMMING TUTORIAL

Chapter 1 : Servoy: Object-Oriented Programming with Prototypal Inheritance

Objective: To understand the basic theory behind object-oriented programming, how this relates to JavaScript ("everything is an object"), and how to create constructors and object instances. To start with, let's give you a simplistic, high-level view of what Object-oriented programming (OOP) is. We.

This file is part of the first edition of Eloquent JavaScript. Consider reading the third edition instead. Most of the ideas behind it were not really new at the time, but they had finally gained enough momentum to start rolling, to become fashionable. Books were being written, courses given, programming languages developed. All of a sudden, everybody was extolling the virtues of object-orientation, enthusiastically applying it to every problem, convincing themselves they had finally found the right way to write programs. When a process is hard and confusing, people are always on the lookout for a magic solution. When something looking like such a solution presents itself, they are prepared to become devoted followers. For many programmers, even today, object-orientation or their view of it is the gospel. The above paragraphs are by no means meant to discredit these ideas. What I want to do is warn the reader against developing an unhealthy attachment to them. So far, we have used objects as loose aggregations of values, adding and altering their properties whenever we saw fit. In an object-oriented approach, objects are viewed as little worlds of their own, and the outside world may touch them only through a limited and well-defined interface, a number of specific methods and properties. We used only three functions, makeReachedList, storeReached, and findReached to interact with it. These three functions form an interface for such objects. Instead of providing regular functions for working with the objects, they provide a way to create such objects, using the new keyword, and a number of methods and properties that provide the rest of the interface. For example, if there are different rabbits, the speak method must indicate which rabbit is speaking. For this purpose, there is a special variable called this, which is always present when a function is called, and which points at the relevant object when the function is called as a method. A function is called as a method when it is looked up as a property, and immediately called, as in object. This argument can be used to specify the object that the function must be applied to. For non-method functions, this is irrelevant, hence the null. When a function is called with the word new in front of it, its this variable will point at a new object, which it will automatically return unless it explicitly returns something else. Functions used to create new objects like this are called constructors. Here is a constructor for rabbits: This makes it easy to distinguish them from other functions. After all, we could have simply written this: For one thing, our killerRabbit has a property called constructor, which points at the Rabbit function that created it. It is part of the prototype of a rabbit. Prototypes are a powerful, if somewhat confusing, part of the way JavaScript objects work. Every object is based on a prototype, which gives it a set of inherent properties. The simple objects we have used so far are based on the most basic prototype, which is associated with the Object constructor. This means that all simple objects have a toString method, which converts them to a string. Our rabbit objects are based on the prototype associated with the Rabbit constructor. Because the rabbit prototype is itself an object, it is based on the Object prototype, and shares its toString method. The properties of the prototype influence the object based on it, but the properties of this object never change the prototype. When looking up the value of a property, JavaScript first looks at the properties that the object itself has. If there is a property that has the name we are looking for, that is the value we get. If there is no such property, it continues searching the prototype of the object, and then the prototype of the prototype, and so on. If no property is found, the value undefined is given. On the other hand, when setting the value of a property, JavaScript never goes to the prototype, but always sets the property in the object itself. For example, it might become necessary for our rabbits to dance. Here is a new approach to the Rabbit constructor: It means that using an object to store a set of things, such as the cats from chapter 4 , can go wrong. If, for example, we wondered whether there is a cat called "constructor", we would have checked it like this: A related problem is that it can often be practical to extend the prototypes of standard constructors such as Object and Array with new useful functions.

For example, we could give all objects a method called properties, which returns an array with the names of the non-hidden properties that the object has: Now that the Object prototype has a property called properties, looping over the properties of any object, using for and in, will also give us that shared property, which is generally not what we want. We are interested only in the properties that the object itself has. Unfortunately, it does make looping over the properties of an object a bit clumsier. Every object has a method called hasOwnProperty, which tells us whether the object has a property with a given name. Using this, we could rewrite our properties method like this: Note that the action function is called with both the name of the property and the value it has in the object. It will be stored in the object, and the next time we want to go over the collection of cats, calling object. This can be solved by doing something even uglier: This example does not currently work correctly in Internet Explorer 8, which apparently has some problems with overriding built-in prototype properties. Unless someone actually messes with the method in Object. There is one more catch, however. Having access to the prototype of an object can be very convenient, but making it a property like that was not a very good idea. Still, Firefox is a widely used browser, so when you write a program for the web you have to be careful with this. An expression such as this one can be used to reliably work around this: This is one of the not-so-well-designed aspects of JavaScript. We could put it into a function, but an even better approach is to write a constructor and a prototype specifically for situations like this, where we want to approach an object as just a set of properties. Because you can use it to look things up by name, we will call it a Dictionary. Note that the values property of a Dictionary object is not part of this interface, it is an internal detail, and when you are using Dictionary objects you do not need to directly use it. This way, when someone, possibly yourself three months after you wrote it, wants to work with the interface, they can quickly see how to use it, and do not have to study the whole program. To prevent wasting your time, it is advisable to document your interfaces only after they have been used in a few real situations and proven themselves to be practical. When it feels ready, it is time to write something about it, and to see if it sounds as good in English or whatever language as it does in JavaScript or whatever programming language. Firstly, having a small, clearly described interface makes an object easier to use. You only have to keep the interface in mind, and do not have to worry about the rest unless you are changing the object itself. When outside code is accessing every single property and detail in the object, you can not change any of them without also updating a lot of other code. If outside code only uses a small interface, you can do what you want, as long as you do not change the interface. This way, if they ever want to change their object in such a way that it no longer has a length property, for example because it now has some internal array whose length it must return, they can update the function without changing the interface. Adding a getLength method which only contains return this. Especially the Array and String prototypes in JavaScript could use a few more basic methods. We could, for example, replace forEach and map with methods on arrays, and make the startsWith function we wrote in chapter 4 a method on strings. For this reason, some people prefer not to touch these prototypes at all. Of course, if you are careful, and you do not expect your code to have to coexist with badly-written code, adding methods to standard prototypes is a perfectly good technique. There will be some objects involved this is, after all, the chapter on object-oriented programming. We will take a rather simple approach, and make the terrarium a two-dimensional grid, like the second map in chapter 7. On this grid there are a number of bugs. When the terrarium is active, all the bugs get a chance to take an action, such as moving, every half second. This usually makes things easier to model in a program, but of course has the drawback of being wildly inaccurate. Fortunately, this terrarium-simulator is not required to be accurate in any way, so we can get away with it. We could have used a single string, but because JavaScript strings must stay on a single line it would have been a lot harder to type. This object keeps track of the shape and content of the terrarium, and lets the bugs inside move. It has four methods: Firstly toString, which converts the terrarium back to a string similar to the plan it was based on, so that you can see what is going on inside it. Then there is step, which allows all the bugs in the terrarium to move one step, if they so desire. When it is running, step is automatically called every half second, so the bugs keep moving. In chapter 7 we used three functions, point, addPoints, and samePoint

to work with points. This time, we will use a constructor and two methods. Write the constructor Point, which takes two arguments, the x and y coordinates of the point, and produces an object with x and y properties. Give the prototype of this constructor a method add, which takes another point as argument and returns a new point whose x and y are the sum of the x and y of the two given points. Also add a method isEqualTo, which takes a point and returns a boolean indicating whether the this point refers to the same coordinates as the given point. Code which uses point objects may freely retrieve and modify x and y. Some things are best written as methods of your objects, other things are better expressed as separate functions, and some things are best implemented by adding a new type of object. To keep things clear and organised, it is important to keep the amount of methods and responsibilities that an object type has as small as possible. When an object does too much, it becomes a big mess of functionality, and a formidable source of confusion. The bugs themselves will also be objects, and these objects are responsible for deciding what they want to do. The terrarium merely provides the infrastructure that asks them what to do every half second, and if they decide to move, it makes sure this happens.

Chapter 2 : JavaScript and Object Oriented Programming (OOP)

JavaScript is an excellent language to write object oriented web applications. It can support OOP because it supports inheritance through prototyping as well as properties and methods. Many developers cast off JS as a suitable OOP language because they are so used to the class style of C# and Java.

History Introduction Many JavaScript programmers overlook or do not know of the ability to write object-oriented JavaScript. Whilst not conventionally an object-oriented language, JavaScript is a prototype-based language, which means that inherited classes are not derived directly from a base class, but rather are created by cloning the base class which serves as a prototype. Object-oriented JavaScript also has several advantages. As JavaScript supports variable data types, class properties do not have to take a fixed data type such as boolean or string but rather can be changed at any time. Furthermore, object-oriented JavaScript is more flexible and efficient than procedural JavaScript, as objects fully support encapsulation and inheritance and polymorphism can be implemented using the prototype property. Prerequisites Although this is an introductory article to object-oriented JavaScript, it would be beneficial to have an understanding of object-oriented programming, as this article does not go into this aspect in too much detail. However, a list of key object-oriented programming terms are listed and defined below for some guidance in this respect. Key Terms Several key terms will be used in this article which are summarized below: This is where the data passed around inside an instance of an object is kept contained within the instance of that object. When a new instance of the object is created, a new set of data for that instance is created. Where a subclass of a class can call the same generic inherited function in its own context. A variable associated with a class. A function associated with a class. All that must be done in order to define a class is for a function to be declared: Creating Class Properties This code so far is just a simple class with only its constructor declared. To add properties to the class, we use the this operator, followed by the name of the property. As previously stated, methods and properties can be created anywhere in JavaScript, and not just in the class constructor. Here is an example of adding properties to MyClass. These properties can be accessed by: This can be accessed anywhere within the class constructor and the class methods using the this operator, so MyData can be accessed by using this. Also note that unlike some object-oriented languages, class properties are accessed with a. This is because JavaScript does not differentiate between pointers and variables. If the class reference is stored in a variable when created, then the class properties can be accessed by the variable name followed by a. Creating Class Methods As said earlier in the article, class methods are created using the prototype property. When a method is created in a class in JavaScript, the method is added to the class object using the prototype property, as shown in the following piece of code: Equally, the method can be created by declaring function MyClass. What this code does is make MyFunction a method of MyClass using the prototype property. This then gives MyFunction access to any other methods or properties created in MyClass using the this operator. This piece of code creates the MyClass class, then creates a property called MyData in the class constructor. A method, called MyFunction is then added to the MyClass object, using the prototype operator, so that it can access the MyClass properties and methods. In this method, MyData is changed to newtext, which is the only argument of the method. This new value is then displayed using an alert box. Encapsulation Encapsulation is a useful part of object-oriented programming that "encapsulates" or contains data in an instance of a class from the data in another instance of the same class. This is why the this operator is used within a class, so that it retrieves the data for that variable within that instance of the class. Public, Protected and Private Members Encapsulation is implemented in JavaScript by separating instance data within a class. However, there is no varying scale of encapsulation through the public, protected and private operators. This means that access to data cannot be restricted, as seen in other object-oriented programming languages. The reason for this is that in JavaScript it is simply not necessary to do so, even for fairly large projects. As of this, class properties and methods can be accessed from anywhere, either inside the class or outside of it.

Encapsulation in Practice An example of encapsulation can be shown below: MyData would return "Some More Text" and c2. MyData would return "Some Different Text", showing that the data is encapsulated within the class. Conclusion to Encapsulation Encapsulation is an important part of object-oriented programming, so that data in different instances of a class are separate from one another; this is implemented in JavaScript by using the this operator. However, unlike other object-oriented programming languages, JavaScript does not restrict access to data within an instance of a class. Inheritance Inheriting Properties As said earlier in the article, there is no direct inheritance in JavaScript, as it is a prototype language. Therefore, for a class to inherit from another class, the prototype operator is used, to clone the parent class constructor, and in doing so, inheriting its methods and properties. In the code example above, two classes are created â€" a base class called Animal and a subclass called Dog which inherits from Animal. In the base class constructor, a property called name is created, and set a value passed to it. When an inherited class is constructed, two lines of code are needed to inherit from the base class, as demonstrated with Dog: Arguments needed by the called function are passed also, starting from the second argument of call , as seen with name. What this means is that the base class constructor is called from within the subclass constructor, therefore applying the methods and properties created in Animal to the subclass. The second line of code needed to inherit from a base class is: Notice also that once a subclass has inherited from a parent class, any data that needs to be accessed from the parent class can be accessed using the this operator, from within the subclass, as the methods and properties are now part of the subclass object. Inheriting Methods Like properties, methods can also be inherited from a parent class in JavaScript, similar to the inheritance of properties, as shown below: An inherited method can be called as so: Creating an Instance of an Inherited Class Classes that inherit from another class can be called in JavaScript as a base class would be called, and methods and properties can be called similarly. It is implemented in JavaScript using the prototype and call functions. Polymorphism Polymorphism is an extension on the principle of inheritance in object-oriented programming and can also be implemented in JavaScript using the prototype operator. Polymorphism is where a subclass of a class can call the same generic inherited function in its own context. Then after that we do something else with it, which for Dog is alert "woof" ; and for Cat is alert "miaow" ; If called, this would look like: The second two lines would alert "Lucy says: Conclusion to Polymorphism Polymorphism is a very useful part of object-oriented programming, and whilst in this article it is not pursued too deeply, the principles remain constant and can be applied like this in most aspects of polymorphism in JavaScript. Conclusion After reading this article you should be able to: Create classes with methods and properties Create inherited classes Create polymorphic functions This is just an introductory article, but I hope you can use these learned skills for more complicated object-oriented structures. History 20th July, Unmodified first copy 4th August,

DOWNLOAD PDF JAVASCRIPT OBJECT ORIENTED PROGRAMMING TUTORIAL

Chapter 3 : oop - Is JavaScript object-oriented? - Stack Overflow

Object Oriented Programming (OOP) refers to using self-contained pieces of code to develop applications. We call these self-contained pieces of code objects, better known as Classes in most OOP programming languages and Functions in JavaScript.

This lesson will introduce you to objects, classes, inheritance, interfaces, and packages. Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Java programming language. What Is an Object? An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner. What Is a Class? A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior. Inheritance provides a powerful and natural mechanism for organizing and structuring your software. This section explains how classes inherit state and behavior from their superclasses, and explains how to derive one class from another using the simple syntax provided by the Java programming language. What Is an Interface? An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. This section defines a simple interface and explains the necessary changes for any class that implements it. What Is a Package? A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface API provided by the Java platform.

DOWNLOAD PDF JAVASCRIPT OBJECT ORIENTED PROGRAMMING TUTORIAL

Chapter 4 : Object-oriented Programming -- Eloquent JavaScript

Object-Oriented Programming is a popular style of programming that has taken root in JavaScript since the beginning. It's so deeply rooted in JavaScript that many of JavaScript's native functions and methods are written in the Object Oriented style; you'll also find many popular libraries.

Features[edit] Object-oriented programming uses objects, but not all of the associated techniques and structures are supported directly in languages that claim to support OOP. The features listed below are, however, common among languages considered strongly class- and object-oriented or multi-paradigm with OOP support , with notable exceptions mentioned. Comparison of programming languages object-oriented programming and List of object-oriented programming terms Shared with non-OOP predecessor languages[edit] Variables that can store information formatted in a small number of built-in data types like integers and alphanumeric characters. This may include data structures like strings , lists , and hash tables that are either built-in or result from combining variables using memory pointers Procedures â€” also known as functions, methods, routines, or subroutines â€” that take input, generate output, and manipulate data. Modern languages include structured programming constructs like loops and conditionals. Modular programming support provides the ability to group procedures into files and modules for organizational purposes. Modules are namespaced so identifiers in one module will not be accidentally confused with a procedure or variable sharing the same name in another file or module. Objects and classes[edit] Languages that support object-oriented programming typically use inheritance for code reuse and extensibility in the form of either classes or prototypes. Those that use classes support two main concepts: Classes â€” the definitions for the data format and available procedures for a given type or class of object; may also contain data and procedures known as class methods themselves, i. For example, a graphics program may have objects such as "circle", "square", "menu". An online shopping system might have objects such as "shopping cart", "customer", and "product". It is essential to understand this; using classes to organize a bunch of unrelated methods together is not object orientation. Junade Ali, Mastering PHP Design Patterns [8] Each object is said to be an instance of a particular class for example, an object with its name field set to "Mary" might be an instance of class Employee. Procedures in object-oriented programming are known as methods ; variables are also known as fields , members, attributes, or properties. This leads to the following terms: Class variables â€” belong to the class as a whole; there is only one copy of each one Instance variables or attributes â€” data that belongs to individual objects; every object has its own copy of each one Member variables â€” refers to both the class and instance variables that are defined by a particular class Class methods â€” belong to the class as a whole and have access only to class variables and inputs from the procedure call Instance methods â€” belong to individual objects, and have access to instance variables for the specific object they are called on, inputs, and class variables Objects are accessed somewhat like variables with complex internal structure, and in many languages are effectively pointers , serving as actual references to a single instance of said object in memory within a heap or stack. They provide a layer of abstraction which can be used to separate internal from external code. External code can use an object by calling a specific instance method with a certain set of input parameters, read an instance variable, or write to an instance variable. Objects are created by calling a special type of method in the class known as a constructor. A program may create many instances of the same class as it runs, which operate independently. This is an easy way for the same procedures to be used on different sets of data. Object-oriented programming that uses classes is sometimes called class-based programming , while prototype-based programming does not typically use classes. As a result, a significantly different yet analogous terminology is used to define the concepts of object and instance. In some languages classes and objects can be composed using other concepts like traits and mixins. Class-based vs prototype-based[edit] In class-based languages the classes are defined beforehand and the objects are instantiated based on the classes. If two objects apple and orange are instantiated from the class Fruit, they are inherently fruits and it is

guaranteed that you may handle them in the same way; e. In prototype-based languages the objects are the primary entities. No classes even exist. The prototype of an object is just another object to which the object is linked. Every object has one prototype link and only one. New objects can be created based on already existing objects chosen as their prototype. You may call two different objects apple and orange a fruit, if the object fruit exists, and both apple and orange have fruit as their prototype. The attributes and methods of the prototype are delegated to all the objects of the equivalence class defined by this prototype. The attributes and methods owned individually by the object may not be shared by other objects of the same equivalence class; e. Only single inheritance can be implemented through the prototype. This feature is known as dynamic dispatch , and distinguishes an object from an abstract data type or module , which has a fixed static implementation of the operations for all instances. If the call variability relies on more than the single type of the object on which it is called i. A method call is also known as message passing. It is conceptualized as a message the name of the method and its input parameters being passed to the object for dispatch. Encapsulation[edit] Encapsulation is an object-oriented programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding. If a class does not allow calling code to access internal object data and permits access through methods only, this is a strong form of abstraction or information hiding known as encapsulation. Some languages Java, for example let classes enforce access restrictions explicitly, for example denoting internal data with the private keyword and designating methods intended for use by code outside the class with the public keyword. Methods may also be designed public, private, or intermediate levels such as protected which allows access from the same class and its subclasses, but not objects of a different class. In other languages like Python this is enforced only by convention for example, private methods may have names that start with an underscore. Encapsulation prevents external code from being concerned with the internal workings of an object. This facilitates code refactoring , for example allowing the author of the class to change how objects of that class represent their data internally without changing any external code as long as "public" method calls work the same way. It also encourages programmers to put all the code that is concerned with a certain set of data in the same class, which organizes it for easy comprehension by other programmers. Encapsulation is a technique that encourages decoupling. Composition, inheritance, and delegation[edit] Objects can contain other objects in their instance variables; this is known as object composition. Object composition is used to represent "has-a" relationships: Languages that support classes almost always support inheritance. This allows classes to be arranged in a hierarchy that represents "is-a-type-of" relationships. For example, class Employee might inherit from class Person. All the data and methods available to the parent class also appear in the child class with the same names. These will also be available in class Employee, which might add the variables "position" and "salary". This technique allows easy re-use of the same procedures and data definitions, in addition to potentially mirroring real-world relationships in an intuitive way. Rather than utilizing database tables and programming subroutines, the developer utilizes objects the user may be more familiar with: Multiple inheritance is allowed in some languages, though this can make resolving overrides complicated. Some languages have special support for mixins , though in any language with multiple inheritance, a mixin is simply a class that does not represent an is-a-type-of relationship. Mixins are typically used to add the same methods to multiple classes. Abstract classes cannot be instantiated into objects; they exist only for the purpose of inheritance into other "concrete" classes which can be instantiated. In Java, the final keyword can be used to prevent a class from being subclassed. The doctrine of composition over inheritance advocates implementing has-a relationships using composition instead of inheritance. For example, instead of inheriting from class Person, class Employee could give each Employee object an internal Person object, which it then has the opportunity to hide from external code even if class Person has many public attributes or methods. Some languages, like Go do not support inheritance at all. Delegation is another language feature that can be used as an alternative to inheritance. Polymorphism[edit] Subtyping - a form of polymorphism - is when calling code can be agnostic as to whether an object belongs to a parent class or one

of its descendants. Meanwhile, the same operation name among objects in an inheritance hierarchy may behave differently. For example, objects of type Circle and Square are derived from a common class called Shape. The Draw function for each type of Shape implements what is necessary to draw itself while calling code can remain indifferent to the particular type of Shape is being drawn. This is another type of abstraction which simplifies code external to the class hierarchy and enables strong separation of concerns. Open recursion[edit] In languages that support open recursion , object methods can call other methods on the same object including themselves , typically using a special variable or keyword called this or self. This variable is late-bound ; it allows a method defined in one class to invoke another method that is defined later, in some subclass thereof. History[edit] UML notation for a class. This Button class has variables for data, and functions. Through inheritance a subclass can be created as subset of the Button class. Objects are instances of a class. Terminology invoking "objects" and "oriented" in the modern sense of object-oriented programming made its first appearance at MIT in the late s and early s. In the environment of the artificial intelligence group, as early as , "object" could refer to identified items LISP atoms with properties attributes ; [10] [11] Alan Kay was later to cite a detailed understanding of LISP internals as a strong influence on his thinking in Alan Kay, [12] Another early MIT example was Sketchpad created by Ivan Sutherland in â€”61; in the glossary of the technical report based on his dissertation about Sketchpad, Sutherland defined notions of "object" and "instance" with the class concept covered by "master" or "definition" , albeit specialized to graphical interaction. For programming security purposes a detection process was implemented so that through reference counts a last resort garbage collector deleted unused objects in the random-access memory RAM. Simula launched in , and was promoted by Dahl and Nygaard throughout and , leading to increasing use of the programming language in Sweden, Germany and the Soviet Union. In , the language became widely available through the Burroughs B computers , and was later also implemented on the URAL computer. In , Dahl and Nygaard wrote a Simula compiler. They settled for a generalised process concept with record class properties, and a second layer of prefixes. Through prefixing a process could reference its predecessor and have additional properties. Simula thus introduced the class and subclass hierarchy, and the possibility of generating objects from these classes. The Simula 1 compiler and a new version of the programming language, Simula 67, was introduced to the wider world through the research paper "Class and Subclass Declarations" at a conference. By , the Association of Simula Users had members in 23 different countries. Early a Simula 67 compiler was released free of charge for the DecSystem mainframe family. The object-orientated Simula programming language was used mainly by researchers involved with physical modelling , such as models to study and improve the movement of ships and their content through cargo ports. Smaltalk included a programming environment and was dynamically typed , and at first was interpreted , not compiled. Smalltalk got noted for its application of object orientation at the language level and its graphical development environment. Smalltalk went through various versions and interest in the language grew. Experimentation with various extensions to Lisp such as LOOPS and Flavors introducing multiple inheritance and mixins eventually led to the Common Lisp Object System , which integrates functional programming and object-oriented programming and allows extension via a Meta-object protocol. In the s, there were a few attempts to design processor architectures that included hardware support for objects in memory but these were not successful. In , Goldberg edited the August issue of Byte Magazine , introducing Smalltalk and object-orientated programming to a wider audience.

Chapter 5 : JavaScript Object-Oriented Programming Part 1 â€” SitePoint

Object-oriented programming (OOP) is a popular programming paradigm or style of programming. It's been around since '70s, but unlike tools and frameworks that come and go, OOP is still very relevant today.

Chapter 6 : Lesson: Object-Oriented Programming Concepts (The Javaâ„¢ Tutorials > Learning the Java L

DOWNLOAD PDF JAVASCRIPT OBJECT ORIENTED PROGRAMMING TUTORIAL

Learn React - React Crash Course - React Tutorial with Examples | Mosh - Duration: Programming with Mosh , views.

Chapter 7 : Introduction to Object-Oriented JavaScript - CodeProject

Â¶ In most programming languages with explicit support for object-oriented programming, inheritance is a very straightforward thing. In JavaScript, the language doesn't really specify a simple way to do it.

Chapter 8 : Object Oriented Programming Using C# .NET

Object Oriented Programming with TypeScript Tutorial (JavaScript OOP) Posted by robert | Filed under TypeScript. Update: New and better Boilerplate that uses ES6 Modules in TypeScript check it out.

Chapter 9 : Object-oriented programming - Wikipedia

Yes, there is a better way -- object-oriented programming will allow you to build websites using reusable blocks of code known as libraries, similar to using bricks to build a house. This course is designed to teach web developers how to utilize the various object-oriented programming features within JavaScript, and more importantly, how to.