Javascript prototypes as abstract objects

by Martin Vézina | 2012-12-10

Abstract classes

Simply put, an abstract class in class-based OOP languages like Java is a class definition that has an incomplete implementation (or no implementation at all) so that it cannot be instantiated. It is useful to define methods that are common to many types of objects, while leaving the differences declared in the interface of the class but not defined until a concrete class extends the abstract.

For example, a Shape abstract class could have a calculateArea abstract method declared and leave its implementation to concrete classes such as Rectangle and Ellipse. The actual calculation would thus be defined only in Rectangle and Ellipse.

Abstract classes don't exist in Javascript, and it would'nt make much sense to try to port the concept directly from classical OOP. Still, we can get some inspiration from the idea of having objects that are not complete and need to be extended by a concrete definition in order to be useful.

Inheritance in Javascript

Objects in Javascript inherit their properties from what is called a prototype istead of a class definition. Prototypes are standard objects themselves, so can inherit from other prototypes, who themselves are objects. Who can inherit from prototypes. Who are objects... This forms a chain of objects, and it is exactly what it is called : the prototype chain.

When a property of an object is requested, Javascript looks at the object instance for the property and gets it if it is found. If not, it attemps to get it from its prototype. If it is not there, it continues up the chain until the property is found in one of the prototypes.

Note that when a property is found up the prototype chain, the sequence ends there. A property, anywhere in the object or its prototype chain, obscures all other properties higher up the chain that are named the same.

The concrete object that inherits from a prototype chain is not of a different type than any object in the prototype chain. In fact, it could at any time become the prototype of another object. It also means that an object in the prototype chain could be used as if it were an instance like any other. There is no distinction in Javascript between the definition of an object and an instance of an object. An object is its own definition.

"Same prototype" means "same object instance"

If two objects share some functionnality but differ on other things, normally their prototype should be the same object and contain the shared functionnality. Its important to note that **"same" in that context does not merely mean an identical object, but really the same instance of an object to be the prototype.** It is an important precision.

In classical OOP, the distinction between class and instance is clear, and we tend to view Javascript like that as well. Class function definitions are not copied with each new instance because they don't change from instance to instance. In javascript, definitions like that don't exist. If you want to inherit from a common "definition", you use the same object instance as a prototype.

Consider this:

```
//this will be our prototype
var Base = function(){
  this.echo = function(){
    console.log('I am Mr. ' + this.name);
  }
}

//here is a "subclass"
var SubA = function(name) {
  this.name = name + ' A';
}
SubA.prototype = new Base();


//here is another "subclass"
var SubB = function(name) {
  this.name = name + ' B';
}
SubB.prototype = new Base();

//now instances
var a = new SubA('John');
a.echo(); // I am Mr. John A


var b = new SubB('John');
b.echo(); // I am Mr. John B
```

In that example, we created two objects with the Base constructor to be used as prototypes. Is there a need for two instances? The answer is no. The two instances of Base are identical, but are not the same object as they should be. Here's the proof that they are not the same object:

```
console.log(Object.getPrototypeOf(a).isPrototypeOf(b));//false

//let's try changing a property of one of the prototypes for testing purposes
SubB.prototype.echo = function(){
  console.log('I am Mr. ' + this.name + ' and I hate you');
}

a.echo(); // I am Mr. John A
b.echo(); // I am Mr. John B and I hate you
```

What it means is that SubA and SubB DO NOT share the same prototype. They share a similar prototype, but this is just an illusion. You should really see it as a different prototype, because the moment you create a new instance, you create an object that lives on its own and has absolutely no connection to its "siblings" or even to the way it was when it was created.

If we wanted SubA and SubB to share the same prototype, we could do:

```
//this will be our prototype : an object litteral
var Base = {
  echo : function(){
    console.log('I am Mr. ' + this.name);
  }
}

//here is a "subclass"
var SubA = function(name) {
  this.name = name + ' A';
}
SubA.prototype = Base;


//here is another "subclass"
var SubB = function(name) {
  this.name = name + ' B';
}
SubB.prototype = Base;


//now instances
var a = new SubA('John');
a.echo(); // I am Mr. John A

var b = new SubB('John');
b.echo(); // I am Mr. John B

//let's try changing a property of one of the prototypes for testing purposes
SubB.prototype.echo = function(){
  console.log('I am Mr. ' + this.name + ' and I hate you');
}

//since both instances have the same prototype, echo was changed for both
a.echo(); // I am Mr. John A and I hate you
b.echo(); // I am Mr. John B and I hate you
```

Now Base is a single object, a unique instance, from which SubA and SubB inherit. Both behave the same and ever will. But what if SubA and SubB really should have a different behavior when we call echo? Well, then this function is not shared behavior, so should not be in their common prototype! Its not because two functions have the same name that they are the same function. Shared prototypes must contain everything that is exactly the same across all instances.

Identical property does not mean shared property

When you create an object with a constructor, that is a function that you call using `new` just as above, Javascript creates an empty object whose prototype is the constructor function's prototype, and calls the constructor function in the context of this empty object, then returns the object. Any property set in the constructor function thus gets created for the new object.

In the following example (the world famous Animal example), only `eat` is the same function shared by both instances of Animal (brian and dino). `speak` is NOT shared by both instances. It just happens to be an identical function for both objects.

```
var Animal = function(name) {
  this.name = name;
  this.speak = function() {
    console.log(this.name + ' is barking');
  }

};

Animal.prototype = {
  eat : function() {
    console.log(this.name + ' is eating');
  }
}

var brian = new Animal('brian');
var dino = new Animal('dino');

console.dir(brian);
console.dir(dino);
```

The output for the `console.dir` would look like that:

```
Animal
  speak: function () {
  name: "brian"
  __proto__: Object
    eat: function () {
    __proto__: Object

Animal
  speak: function () {
  name: "dino"
  __proto__: Object
    eat: function () {
    __proto__: Object
```

There's no need for two identical objects

As you can see, there are two instances of a similar function `speak`, one for each `Animal` instance. As for `eat`, it is in the prototype that's shared by both, i.e. the prototype of both Animal instances is a reference to a third object that exists only once.

Even if two objects are created with the same constructor, they do not share much, other than their prototype chain. Any property that is created in the constructor is completely distinct in both objects. Don't be fooled by the fact that they look the same. If some of these properties were methods (functions), they would get created distinctly for each instance as well. They are pretty damn good candidates for becoming properties of the prototype : normally, methods are shared across different objects of the same type.

In the previous example, the only reason for `speak` functions to be different is if they were to do different things, which is not the case here. So `speak` should be brought back in the prototype of `Animal`.

Let's reprogram the example above with `speak` in the prototype:

```
var Animal = function(name) {
  this.name = name;
};

Animal.prototype = {
  eat : function() {
    console.log(this.name + ' is eating');
  },
  speak : function() {
    console.log(this.name + ' is barking');
  }
}

var brian = new Animal('brian');
var dino = new Animal('dino');
```

Now we get only one instance of the function definitions, no matter how many instances of `Animal` we create.

If you're following, you'll remember that we stated earlier that there is no difference between an object that serves as a prototype and any other object. Yet in the example above, instances of `Animal` are created by calling a constructor, but we chose to create the prototype as a litteral. Why?

The whole point of prototyping, I hope you get it by now, is to minimize the number of objects by sharing what is identical. Objects that are used as prototypes are therefore always entirely unique. **Every object instance should get its behavior from functions in its prototype, not from functions in itself, unless the behavior is truly unique to that instance.** In other words, you should never duplicate the same function definition by creating a new instance of it.

If a type of object needs to exist only as one instance, why would we need a constructor for it, as we would call it only once? After all, an object created from a constructor is not very different from one defined as a litteral. You can pass arguments to a constructor to get a different object with each instance creation, but if you have only one creation, there is no point in having arguments.

Chaining prototypes

Going back to our example, let's say that we now need a third Animal instance, but this time it is scratchy that we need to create. But wait, aint scratchy a cat? From what I learned in school, a cat doesn't bark, so we'll need a different implementation of `speak` for scratchy.

Ok, should we bring back speak in the constructor so that each animal has its own speak function? Let's check the code:

```
var Animal = function(name) {
  this.name = name;
  this.speak = function() {
    console.log(this.name + ' is barking');
  }
};
```

Hum, no. True every animal would get its own distinct speak function, but these would be identical. Should we pass the speak function as an argument, so we can set a different one for each animal? Let's try:

```
var Animal = function(name, speakFcn) {
  this.name = name;
  this.speak = speakFcn;
};


var brian = new Animal('brian',
  function() {
    console.log(this.name + ' is barking');
  }
);
var dino = new Animal('dino',
  function() {
    console.log(this.name + ' is barking');
  }
);
var scratchy = new Animal('scratchy',
  function() {
    console.log(this.name + ' is meowing');
  }
);
```

Well, each animal gets its own speak function allright, but that's a pain to create an instance as this is no different to hardcoding each object. Plus, there is a lot of duplication, as each animal of the same type gets a different instance of a similar speak function. Let's try something else. Could'nt we create a prototype for Cat and one for Dog?

```
var Dog = function(name) {
  this.name = name;
};

Dog.prototype = {
  speak : function() {
    console.log(this.name + ' is barking');
  }
}
var Cat = function(name) {
  this.name = name;
};

Cat.prototype = {
  speak : function() {
    console.log(this.name + ' is meowing');
  }
}

var brian = new Dog('brian');
var dino = new Dog('dino');
var scratchy = new Cat('scratchy');
```

Now that's better, but what about eating? the `eat` function is the same for both cats and dogs (assuming they eat the same thing... that is, garbage). Shouldn't it be present in the prototype as well? It should, but an object can have only one prototype, and now it is different for both types of animals. We could write the function twice, once in each prototype, but that would be changing 4 quarters for a dollar.

Since prototypes are objects just like any other, why not add a prototype to them? Cat's prototype and Dog's prototype could share a common Animal prototype. We're beginning to see the chain!

prototype and [[prototype]]... now it gets messed up

The question is now, how can we attribute a prototype to an object litteral. While you could think that `Dog.prototype.prototype = Animal` should do the trick, things are never that easy. `Dog` is a Function, whereas `Dog.prototype` is an Object. Only Functions have a `prototype` property, while objects have the `__proto__` property, refered to in common language as [[prototype]]

The prototype chain truly refers to the chain of [[prototype]]s of objects. The Function's `prototype` property is just an indication of which object to use as [[prototype]] for the empty object that is created and returned when the function is invoked as a constructor.

Why this confusion? Because in Javascript, everything is an object, and every object has a [[prototype]]. Even a function is an object, with its own [[prototype]], which is not the same thing as the `prototype` to use when it is invoked as a constructor. A function's [[prototype]] is its constructor's prototype... and since every function is an instance of Javascript's Function type, every function instance has `Function.prototype` as its [[prototype]]. Phew. Reread that, it will make more sense.

So what we need to set is not `Dog.prototype.prototype` but `Dog.prototype.__proto__`. Problem is, `__proto__` is how it is named internally for Javascript, but it is not intended to be accessed directly. In fact, the `__proto__` property is deprecated, so we can't use it as is.

Fortunately, there is a way to create an object without a constructor function and to set its [[prototype]] in the process. Introduced with ECMAScript 5 is a function called

`Object.create` that creates a new, empty object with a specified object as [[prototype]].

You can think of `Object.create` as a way to unshift a prototype chain. It creates a new object with an existing one as [[prototype]], the latter retaining its own [[prototype]]. It's like adding an empty link to the beginning of the prototype chain.

Let's try with our example:

```
var Animal = {
  eat : function() {
    console.log(this.name + ' is eating');
  }
}

var Dog = function(name) {
  this.name = name;
};

Dog.prototype = Object.create(Animal);
//Dog.prototype is now an empty object with Animal as [[prototype]]

Dog.prototype.speak = function() {
  console.log(this.name + ' is barking');
};

var Cat = function(name) {
  this.name = name;
};

Cat.prototype = Object.create(Animal);
Cat.prototype.speak = function() {
  console.log(this.name + ' is meowing');
};

var brian = new Dog('brian');
var dino = new Dog('dino');
var scratchy = new Cat('scratchy');
console.dir(brian);
```

Output of `console.dir`:

```
Dog
  name: "brian"
  __proto__: Object
    speak: function () {
    __proto__: Object
      eat: function () {
      __proto__: Object
```

That is exactly what we were after. Nothing is duplicated. The eat function, of which only one instance exists, is shared among the 3 final instances that we created. As for speak, there are two instances, one for each type of animal. If we were to create thousands of dogs and cats instances, the one and only thing that would be completely unique to all would be the name property. Everything else would be shared.

Compatibility

Like all good things, there's a catch to `Object.create` : it is not available in all browsers. But, in its simplest form, it is quite easy to emulate. If you want to use it and want your code to be compatible with older browsers (IE8 and IE7 being the major ones you'd want to support), just add the following code to your script:

```
if (typeof Object.create !== 'function') {
  Object.create = function (o) {
    function F() {}
    F.prototype = o;
    return new F();
  };
}
```

This is not the exact functionnality of `Object.create`, but should do for most cases. For those who are wondering, `Object.create` can take a second argument, the list of properties for the created object. These properties have to be described in a specific format, which is a pain to use. Look up for the resources at the end of this post if you want to learn more about this.

Defining the new object

To populate an object created with `Object.create` with properties, you can use many syntaxes. You can set them directly, like I did above, or if you use jQuery, you could do the following to create an object mostly as a litteral.

```
var Dog = function(name) {
  this.name = name;
};

Dog.prototype = $.extend(Object.create(Animal),
  {
    speak : function() {
      console.log(this.name + ' is barking');
    }
  }
);
```

How to instantiate

Remember that we do not have constructors for objects to be used as prototypes, because we don't need the flexibility of being able to create many instances. The reason we don't need flexibility is that we seldom need similar prototype objects. As for real, usable instances, we might need to create a lot of those. So we need a way to instantiate distinct objects.

In the above example, we have set a constructor function for both Cats and Dogs, which looks like this for both:

```
var Dog = function(name) {
  this.name = name;
};
```

That's not a problem with simple objects such as our example, but sometimes instances are more complicated to construct than that. Sure, you should keep your process simple, but still you might want to have a common way to initialize different types of end objects. There might be many ways, but one that is widespread is with the use of an initializing function in a prototype instead of a constructor function.

The initialize function looks like the constructor function, but it is brought back in a prototype along the chain. When we call it on the instance, `this` refers to the instance anyway, not to the prototype, so it has exactly the same effect as a constructor.

Since we drop the constructor functions, we can rewrite our Dog and Cat objects as litterals. They will be used as the prototype for every instance we create. We will get instances of them with `Object.create`, just as we did when we needed prototype objects, but will differentiate these objects with the init method.

A nice trick that can be optionnaly used for the init method is to return the object instance, so you can chain your initialization process in a single line. The last function in the line, init, returns what will be assigned to the instance variable.

Rewriting our example with this technique would give:

```
var Animal = {
  init : function(name) {
    this.name = name;
    return this;
  },
  eat : function() {
    console.log(this.name + ' is eating');
  }
}

//Dog and Cat are no longer creator functions, they are objects litterals
//to be used as prototypes. Their own prototype is Animal.
var Dog = Object.create(Animal);
Dog.speak = function() {
  console.log(this.name + ' is barking');
};

var Cat = Object.create(Animal);
Cat.speak = function() {
  console.log(this.name + ' is meowing');
};

var brian = Object.create(Dog).init('brian');
var dino = Object.create(Dog).init('dino');
var scratchy = Object.create(Cat).init('scratchy');
```

Prototypes are abstract objects

In a way, most of the objects that are used as prototypes therefore become some sort of abstract definitions, as they are just a group of general functionnalities shared by other objects. Much like abstract classes, some details are left intentionally undefined in prototypes, details that will be defined futher down the chain. That would make these objects unusable by themselves. But in Javascript, it's only a decision that we make, it's not that Javascript would prevent us from using these objects as if they were a normal instance.

But it's not because all objects can be used by themselves that they should. We can design some objects to be used only in certain contexts. We can decide that some objects are to be used only as prototypes.

Prototype objects can and, arguably, should be seen as a different kind of objects. Understand me : from the language point of view, all objects are the same. But from the programmer's point of view, objects to be used as prototypes should be designed and optimized for their task. And their task is strictly to act as a prototype. I don't think that it's helpful to design prototypes to be used directly.

Concretely, **there are objects in which `this` refers to themselves, and objects in which `this` refers to something else.** The first are what I call "instance" objects, the second "abstract" objects. Don't make hybrid objects.

Prototypes are not interfaces

As you might have noticed, there is a major difference between "abstract" prototypes and Abstract classes in classical OOP.

A common mistake when creating objects for the prototype chain, from programmers that come from classical OOP anyway, is to define common properties high up in the chain because they exist for all instances. We feel the need to define the property as if the abstract object described an interface. Yet **there is no point in defining in a prototype a property that will be present in objects that descend from it**. Javascript is not like Java : you don't declare in the base objects variables that will be different to all instances of the descendants. You declare a variable only on the level where it will be defined.

Take the `name` property of our animals. Since every animal has a name, it's natural to consider this property as common to all, and define it in the common denominator which is the Animal prototype. The thing is, `Animal` has no name. A `Dog` instance has a name.

In Javascript, you cannot say ***an** Animal has a name*. `Animal` is an object, not a definition, even if we use it like so. And that object has no `name` property. Why then is `name` referred to in Animal's methods if Animal has no name? Because Animal is abstract : it is not intended to be used by itself. `this`, in Animal, will *never* refer to Animal. It will refer to whatever object descends from Animal, dino for example. And dino has a name.

Now, you'll say, the old reflex does not hurt either. After all, if you put the `name` property in `Animal`, it will always be overridden so would not influence anything. You might argue that defining it in Animal gives you information, as a programmer, about the way you have to extend this object lower down the prototype chain. Yet I find that it gives a false sense of security. I'd say that this kind of insight should be commented instead of "declared". The only properties that should be defined are those that are used.

Functions vs any other type of property

From the above naturally follows that Function properties should be defined higher up the prototype chain, whereas any other property should be nearer the instance object, the one that is not extended. The reason is that the behavior is probably what's common to many instances, and that comes from the methods. Other properties are prone to be different from instance to instance, and since properties should be defined at the level where they are different, that should be near or at the instance level.

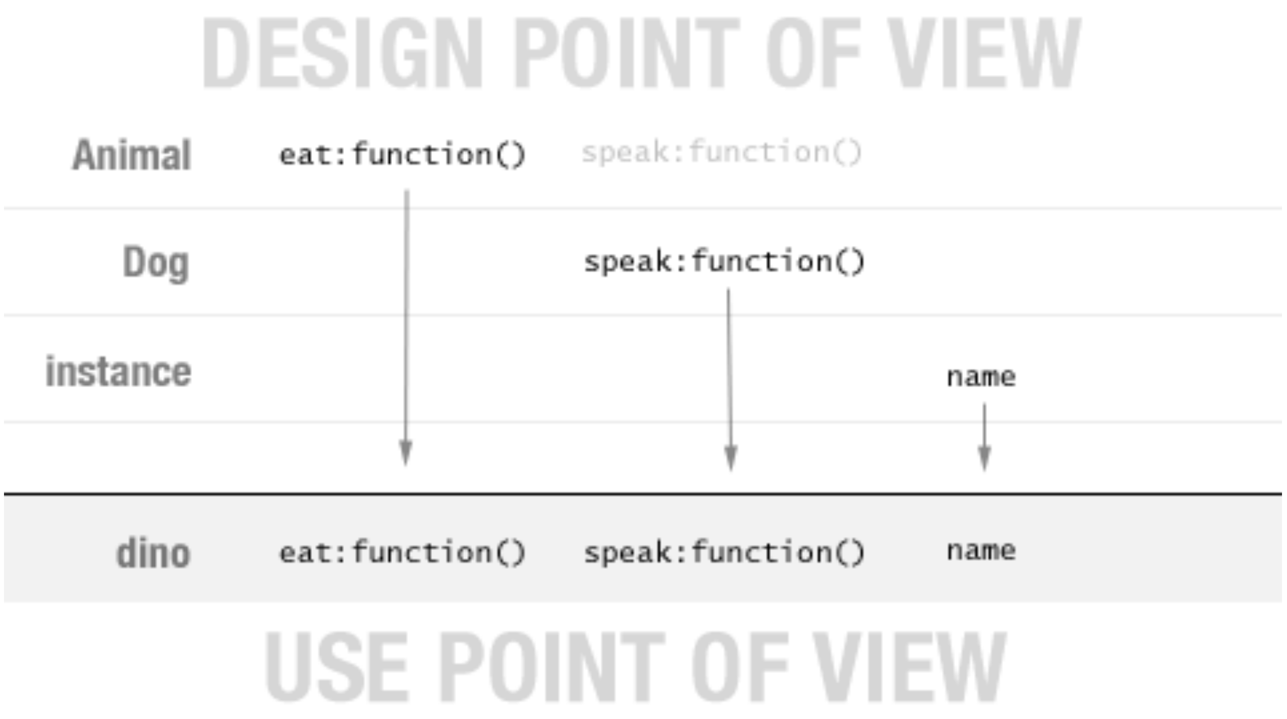
As a nice side effect of using an initializing function instead of a constructor, as described above, is to bring back instance variable declaration in what seems like the prototype where we feel it belongs. The key is using `this`. In the prototype's init function, `this` refer to the instance, not the prototype so the properties are really defined for the instance. Yet, the defining process is coded in the prototype so variable declarations is closer to the functions that use them. It is different from defining the properties in the prototype litteral, which should be avoided as already stated.

On the use of *super*

It can sometimes be difficult to intellectually manage the fact that for the same function, `this` is not always the same thing depending on what context the function is executed, but you really have to get used to it. You have to consider that any function defined in a prototype will not be executed in the context of the prototype, but rather in the context of the final instance.

Prototypes are object instances, but the key point is that they are NOT used as such. They are like ghosts who lend their properties, if need be, to whatever object descends from them. And all objects along the chain act like that: **prototypes do not lend their properties to the *next* object in the chain, they rather lend it to the *first* object in the chain**, the instance that you manipulate. As soon as the instance has a property that is named the same as one of its prototype's properties, it does not need it anymore from its prototype chain. It does not even have access to it.

As an analogy, when I design the *structure* of my objects with their prototype chain, I look at them from a bird's eye view. I see the hierarchy, and the depth of each property along the chain. I work hard to put each property in its right place along the chain. But when I *use* an object, I look at its face from the ground. I see all properties next to one another, and don't know how deep they are or if any property hides another behind.



The inner code of each function is designed with the *use* paradigm in mind. A function is a property of an object and might or might not override another. But when it overrides another, it fully overrides. The overridden value does not exist anymore for the object.

Prototyping is a natural way, in Javascript, for objects to lend their properties to other ones, but it is not the only way to do so. Any object can lend its properties to any other. Look at that:

```
var Foo = {
  doFoo : function() {
    console.log(this.name + ' is doing foo');
  }
}

var Bar = {
  name : 'baz',
  doBar : function() {
    console.log(this.name + ' is doing bar');
  },

  doFooEvenIfYoureBar : Foo.doFoo
}
```

This is completely legal Javascript. Sure the same behavior could be accomplished with Foo as the [[prototype]] to Bar, but the end result is the same. In either case, Foo.doFoo would not be executed in the context of Foo, but in the context of Bar.

What I'm trying to say is that `super` is not natural in Javascript. If an object overrides a method of one of its prototypes, then this method is not part of itself anymore, just as Foo.doFoo is not part of Bar. I argue that if an object should retain an overridden function of one of its prototypes, it should borrow it just as it would if it came from any other object not in its prototype chain. Hardcoded. (I never thought I'd say that. It makes me shiver.)

```
var Animal = {
  init : function(name) {
    this.name = name;
    return this;
  },
  eat : function() {
    console.log(this.name + ' is eating');
  },
  speak : function() {
    console.log(this.name + ' is growling');
  }
}

var Dog = Object.create(Animal);
```

```
//Animal.speak is overridden for Dog
Dog.speak = function() {
  this.basicSpeak();
  console.log(this.name + ' is barking');
};

//Dog borrows the speak function from an object, which happens to be its
//prototype, with a different name because speak is already taken.
Dog.basicSpeak = Animal.speak;
```

You will find many resources and tutorials, some at the end of the post you're reading, whose authors are not adherent to that school of thought and implement a `super()` method to call the overriden one. I would not say it is bad practice, as they are better programmers than I'll probably ever be, I'm just saying that you can have good reasons not to implement the super method as if you were in classical OOP. And you have other options.

Conclusion

Javascript is a very flexible language, but flexibility means that there is never only one way of doing something. There is hardly ever an accepted way of doing something either. There are many articles out there that show you how to deal with inheritance the prototypal way.

For my part, I wanted to deal with it without relying on a library or extra helper code. I wanted to implement inheritance only with core Javascript, so that my code could be part of any system and did not depend on any specific code. I feel that a helper object to implement inheritance can be replaced by just a little more structure when coding.

I hope my post will help you deal with inheritance the vanilla way...