Inheritance and Instance Variables

```ruby
class Animal
  attr_accessor :name

  def initialize(name)
    @name = name
  end

  def name
    @name
  end
end

class Dog < Animal
  def initialize(name, color)
    super(name)
    @color = color
  end

  def color
    @color
  end

  def name
    "Yes..... " + @name
  end
end



dog = Dog.new("labdradog","brown")
```

```
 => #<Dog:0x0000000090d260 @name="labdradog", @color="brown">
```

```ruby
dog.name
```

```
 => "Yes..... labdradog"
```

When you invoke super with arguments, Ruby sends a message to the parent of the current object, asking it to invoke a method of the same name as the method invoking super. super sends exactly those arguments.

```
dog.name
 => "Yes..... labdradog"
```

Because this code behaves as expected, you may be tempted to say that these variables are inherited. *That is not how Ruby works*.

All Ruby objects have a set of instance variables. These are not defined by the object's class - they are simply created when a value is assigned to them. Because instance variables are not defined by a class, they are unrelated to subclassing and the inheritance mechanism.

In the above code, Dog defines an initialize method that chains to the initialize method of its superclass. The chained method assigns values to the variable @name, which makes those variables come into existence for a particular instance of Dog.

The reason that they sometimes appear to be inherited is that instance variables are created by the methods that first assign values to them, and those methods are often inherited or chained.

Since instance variables have nothing to do with inheritance, it follows that an instance variable used by a subclass cannot "shadow" an instance variable in the superclass. If a subclass uses an instance variable with the same name as a variable used by one of its ancestors, it will overwrite the value of its ancestor's variable.