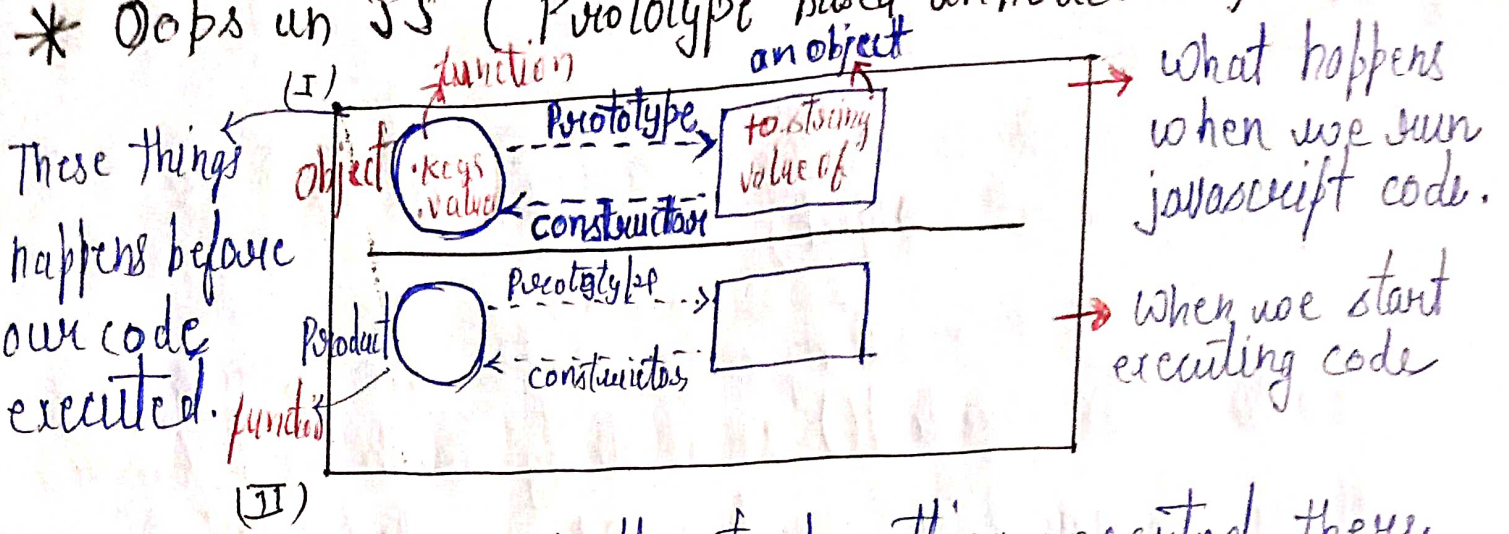
* Oops in JS (Prototype based inheritance)



- Before our code actually start getting executed, there is a function named as capital 'O' object (Object()). and this function has some values like properties like .keys, .values etc. and it has one more property called .prototype.

- This .prototype property actually points to another js javascript object and this object is very important because this object has all the functions like toString(), valueOf(), all those functions that are available on an object but we never wrote ourself.

- And on this object, there is a property called constructor that actually points back to this original object.

Example

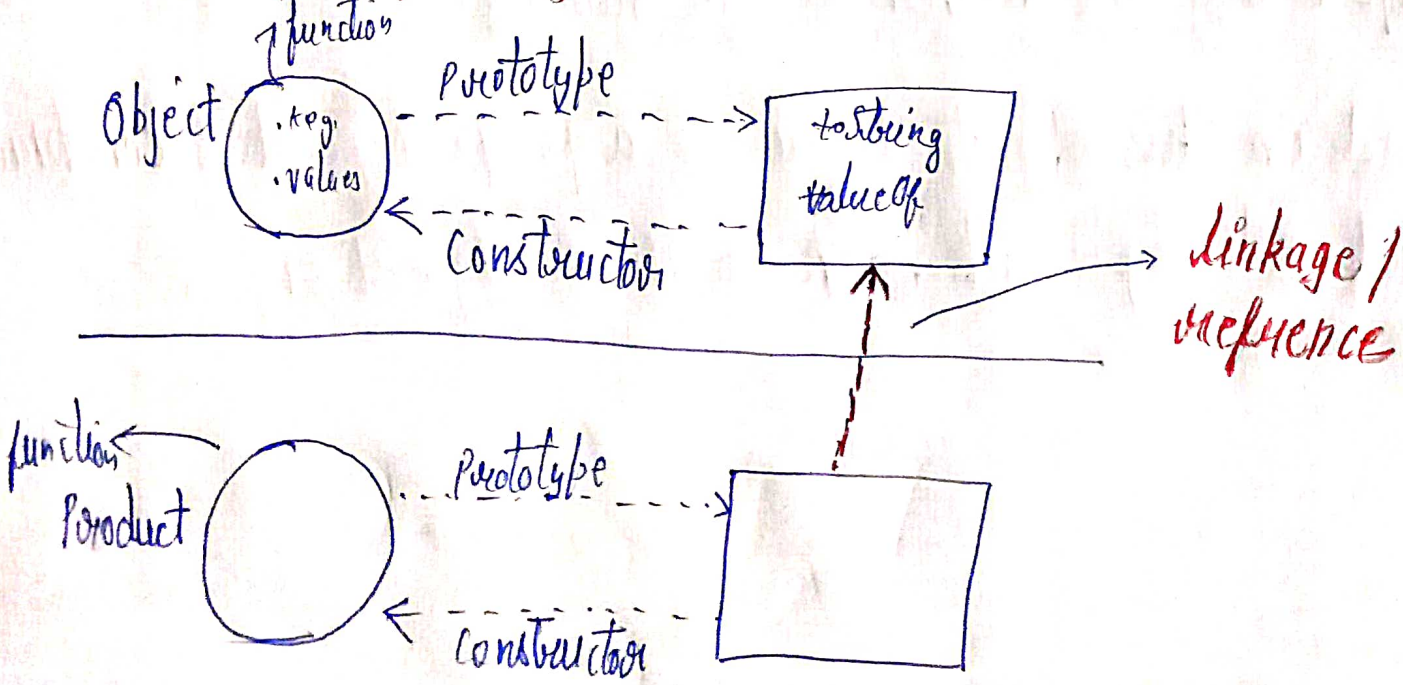
→ Before we start actually executing any code, we have Capital O object (Object).

→ Once we start executing code, let's we wrote code function like product() and along with that function there is another object that is setup. We can refer to this object by product.prototype and from this object we can refer back to the product by doing .constructor to this object.

Example:—

product.prototype.constructor

→ Product.prototype. is actually linked with internally with Object.prototype.



→ The moment we hit new Product() it creates a brand new plain javascript object and now before anything else happened, this object is actually linked with Product.prototype.

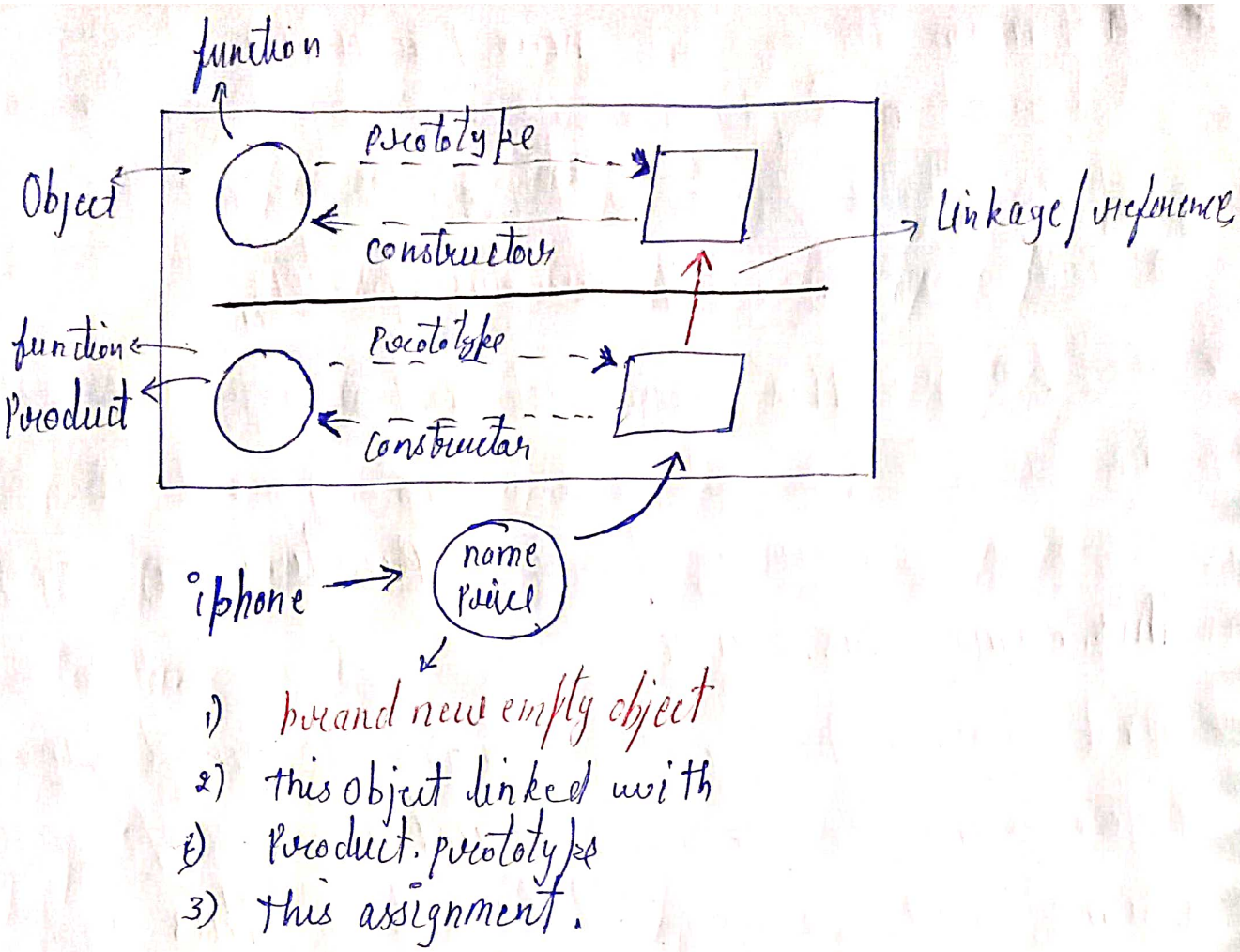
→ We assign a "this" keyword that actually refers to the call site (brand new js object) and then we start executing function Product which says this.name, this.price.

→ And if we have not manually returned another object from the function Product(), javascript will assume that you wanted to return this and this object actually return and stored inside the variable "Iphone". So "Iphone" variable actually refering to this particular object

Example

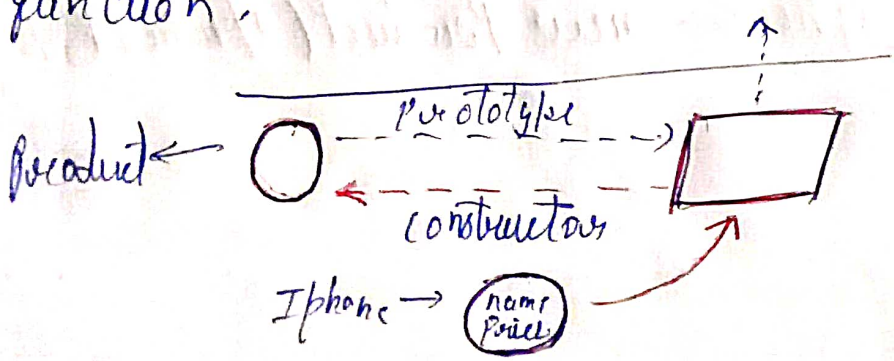
```
function Product(n, p){
    this.name = n;
    this.price = p;
}

const Iphone = new Product("Iphone 14", 10000);
```



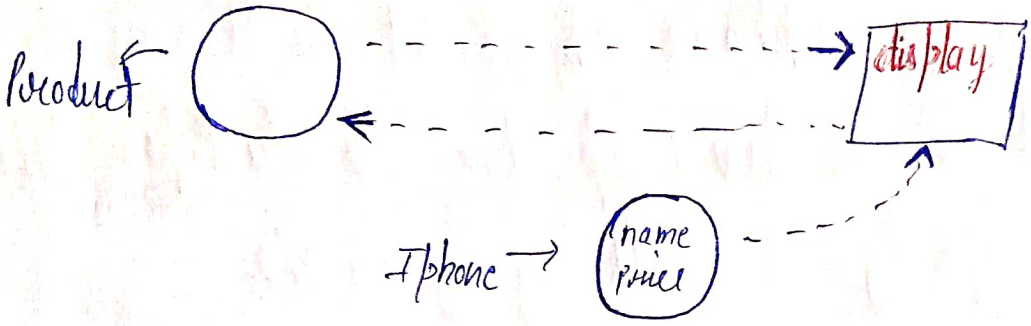
1) brand new empty object
2) this object linked with
   Product.prototype
3) this assignment.

→ On this brand new ~~empty~~ empty object, we linked with Product.prototype and then we try to call constructor product with this object (empty object). Now this object doesn't have a constructor but this object is linked to another object which has a property constructor that is referring to your Product() function.



→ Let's assign a function on the Product.prototype i.e display() function. like this.

```
Product.prototolype.display = function () {
    console.log("Details of the product are", this);
}
```



→ Now if I do Iphone.display();, it will print

" details of the product are { name: 'Iphone 14', price: 10000 }

So, what is going on?

→ During runtime, we come to our object (empty object) Iphone.display() (some property), javascript will check does your object have this property (display), No. Iphone object does not have display property, then it will go one step above in the prototype chain and does Product.prototype. have display property Javascript says, yes it has display property which is technically a function. and now it will call it.

→ If we do something like this

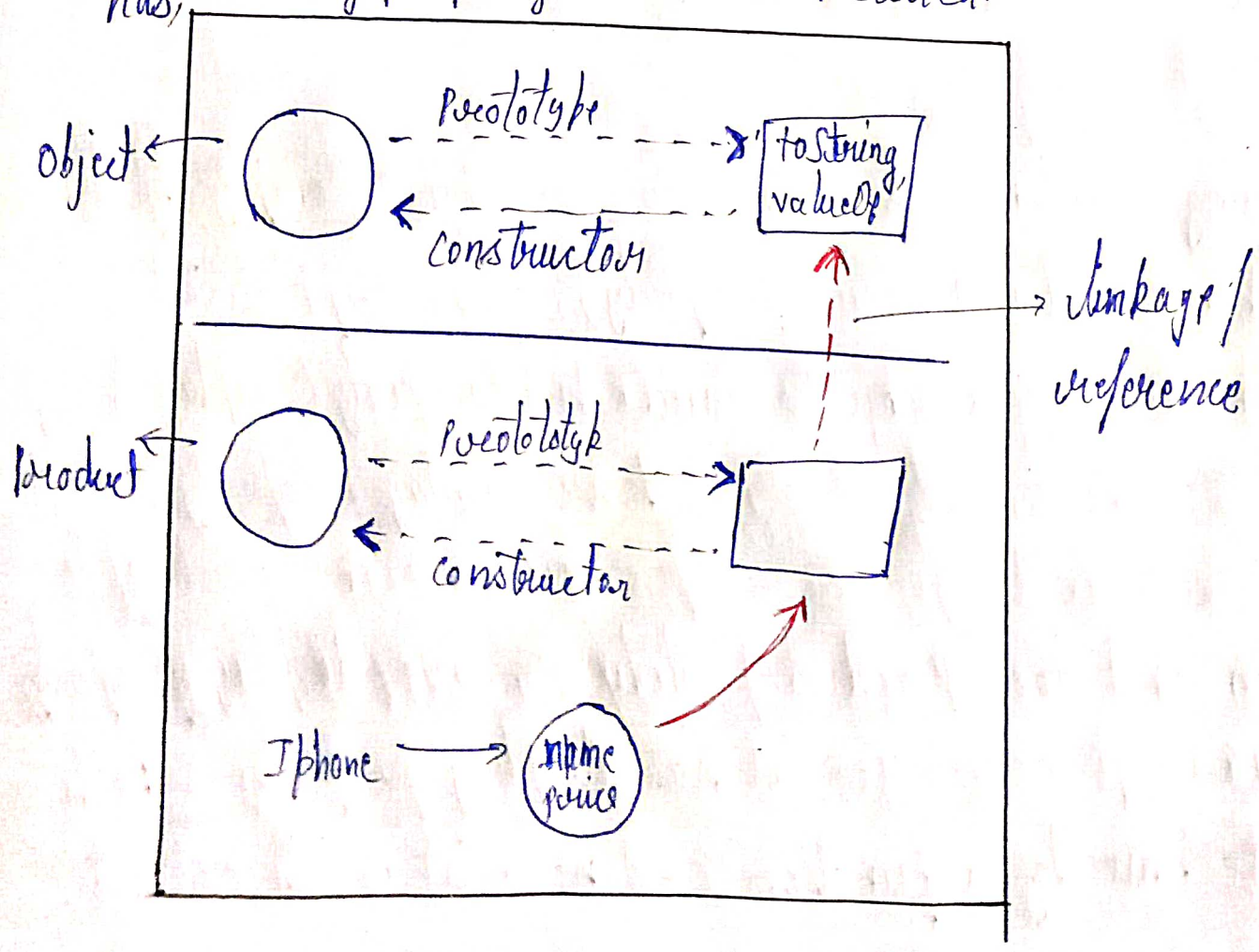
Iphone.toString()

we will get in output '[object, object]'

→ Because js will check, do you have toString property in your Iphone object, it will say No.

→ Then it will go one level up, does Product.prototype has toString property, it will say again No.

→ Then again it will go one level up, does Object.prototype have toString property, yes it has, 'toString' property and it will called.



this concept is called prototype chaining

our object is actually chained to a prototype that is chained to another prototype and so on.

→ When we actually sees that [[Prototype]] that is actually referring to the prototype object that we are kind of linked to. If we want programmatical access to that prototype object, we can do something like this __proto__ this value is actually called as dunder proto.

→ How it determines, what is the property of iphone.__proto__ should point to?

This __proto__ actually invokes a function internally and it will check which object is the calling context (iphone→here). So it dynamically resolves who is the prototype of iphone (i.e Product function) and it actually gives the Product() function.

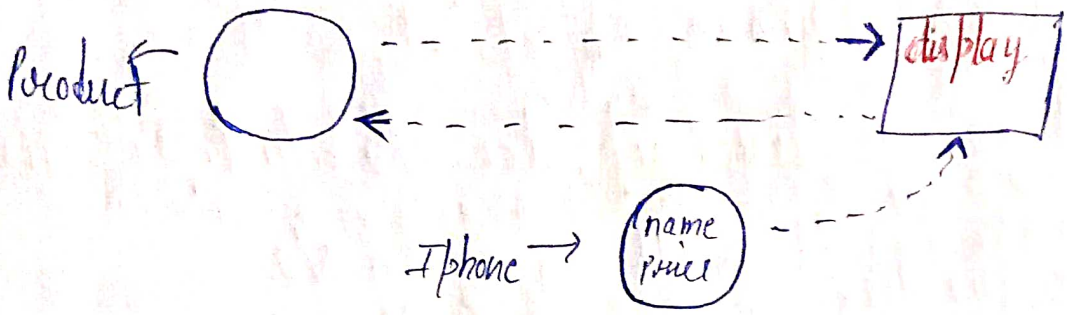
* Super()

→ Super refers to the same class that is mention by our prototype.

→ It is commonly used in class inheritance to invoke the parent class's constructor or methods.

→ Let's assign a function on the Product.prototype i.e display() function. like this.

```
Product.prototolype.display = function () {
  console.log("Details of the product are", this);
}
```



→ Now if I do Iphone.display();, it will print

" details of the product are { name: 'Iphone 14', price: 10000 }

So, what is going on?

→ During runtime, we come to our object (empty object) Iphone.display() (some property), javascript will check does your object have this property (display), No. Iphone object does not have display property, then it will go one step above in the prototypple chain and does Product.prototolype. have display property Javascript says, yes it has display property which is technically a function. and now it will call it.

→ If we do something like this

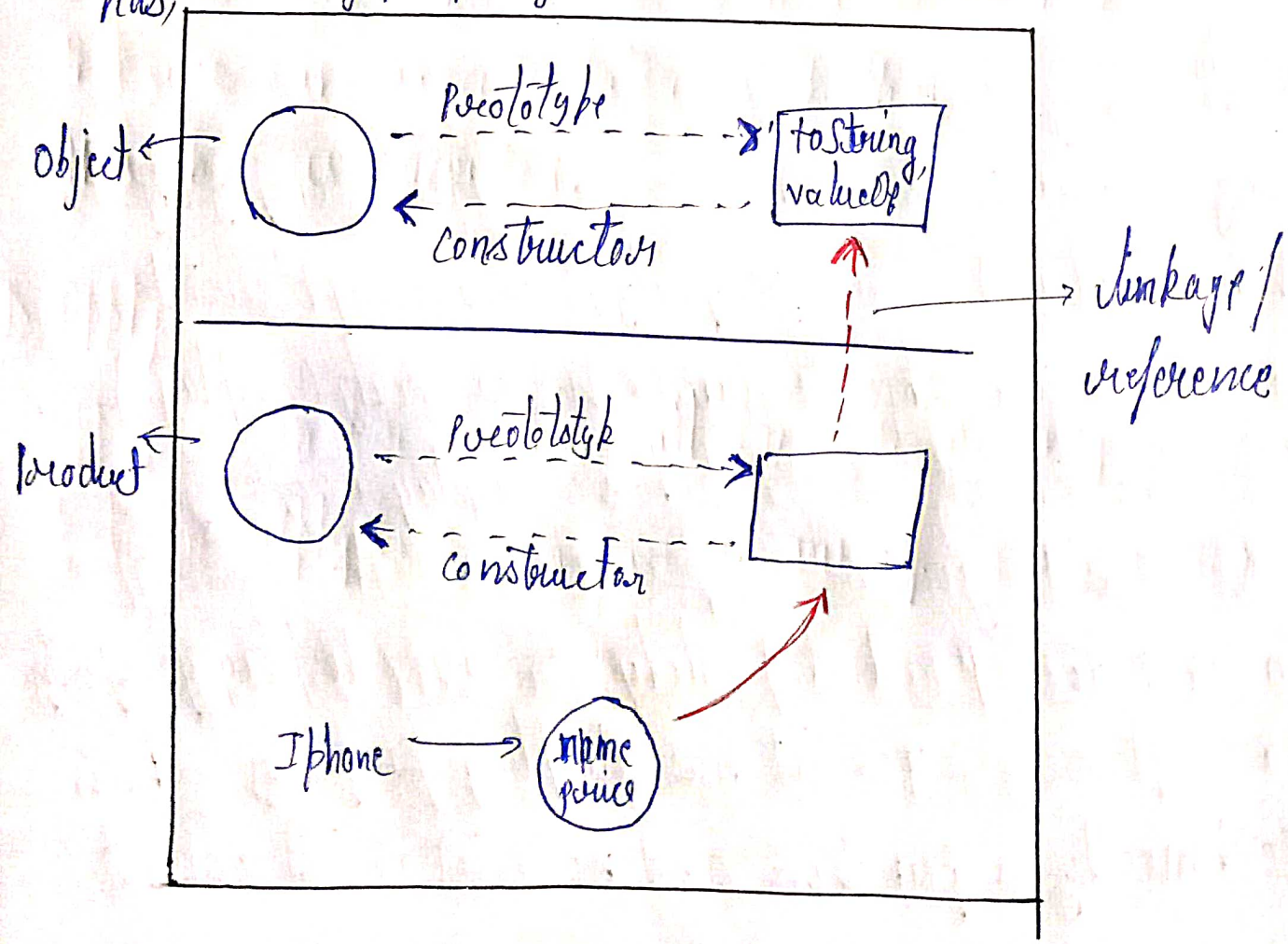
Iphone.toString()

we will get in output '[object, object]'

→ Because js will check, do you have toString property in your Iphone object, it will say No.

→ Then it will go one level up, does Product.prototype has toString property, it will say again No.

→ Then again it will go one level up, does Object.prototype have toString property, Yes it has, 'toString' property and it will called.



This concept is called Prototype chaining. [illegible] our object is actually chained to a prototype that is chained to another prototype and so on.

→ When we actually sees that [[Prototype]] that is actually referring to the prototype object that we are kind of linked to. If we want programmatical access to that prototype object, we can do something like this __proto__ this value is actually called as dunder proto.

→ How it determines, what is the property of iphone.__proto__ should point to?

This __proto__ actually invokes a function internally and it will check which object is the calling context (iphone here). So it dynamically resolves who is the prototype of iphone (i.e Product function) and it actually gives the Product() function.

* Super()

→ Super refers to the same class that is mention by our prototype.

→ It is commonly used in class inheritance to invoke the parent class's constructor or methods.