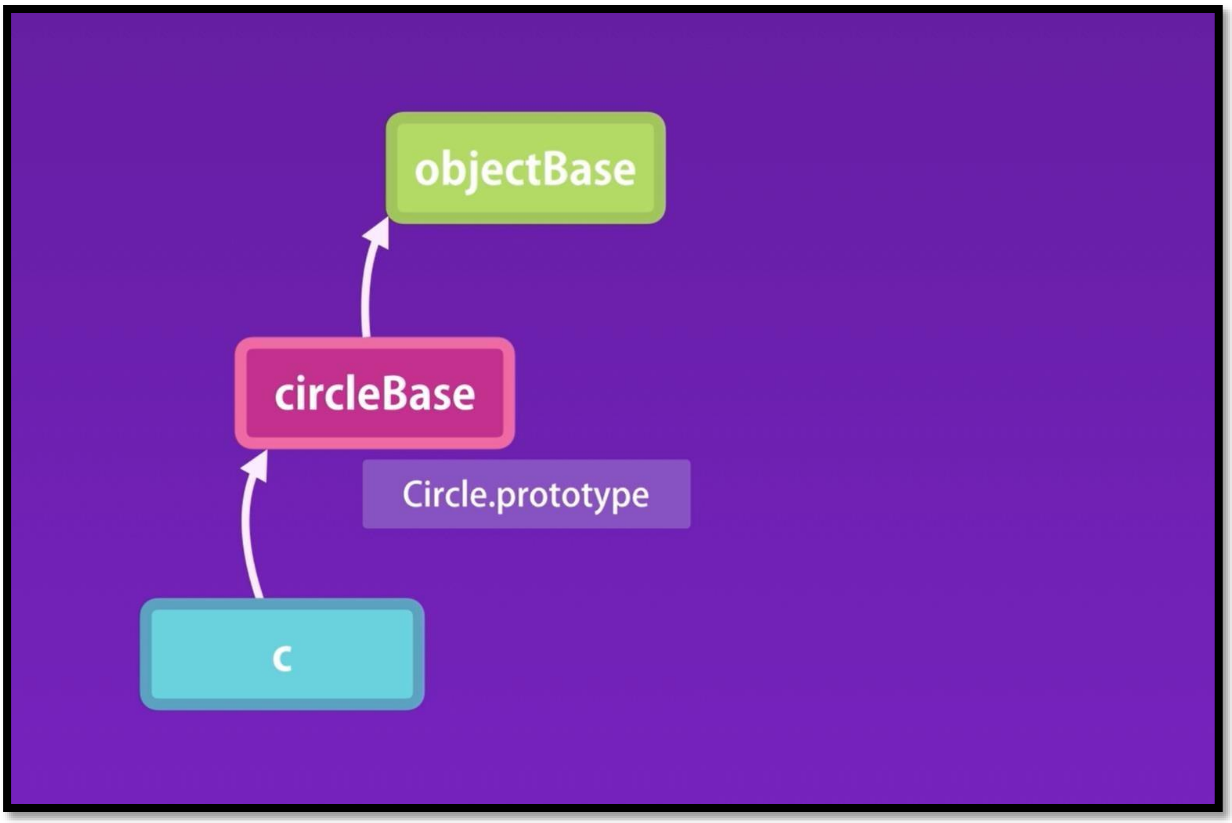
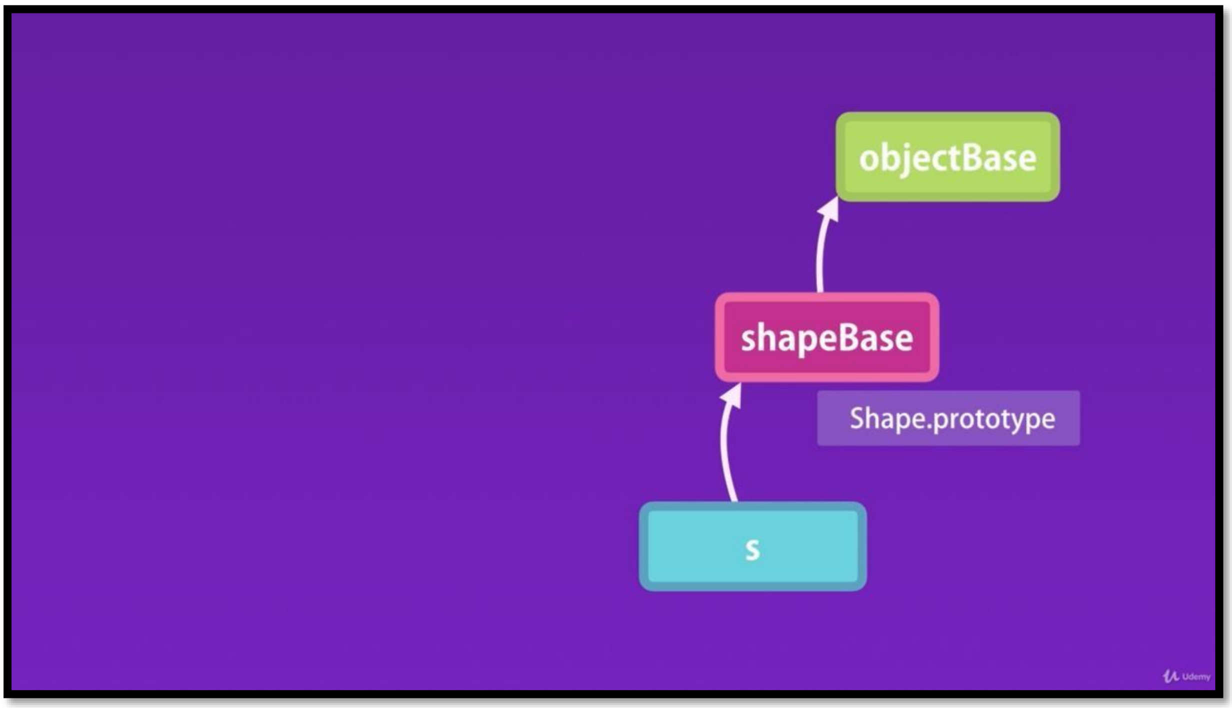
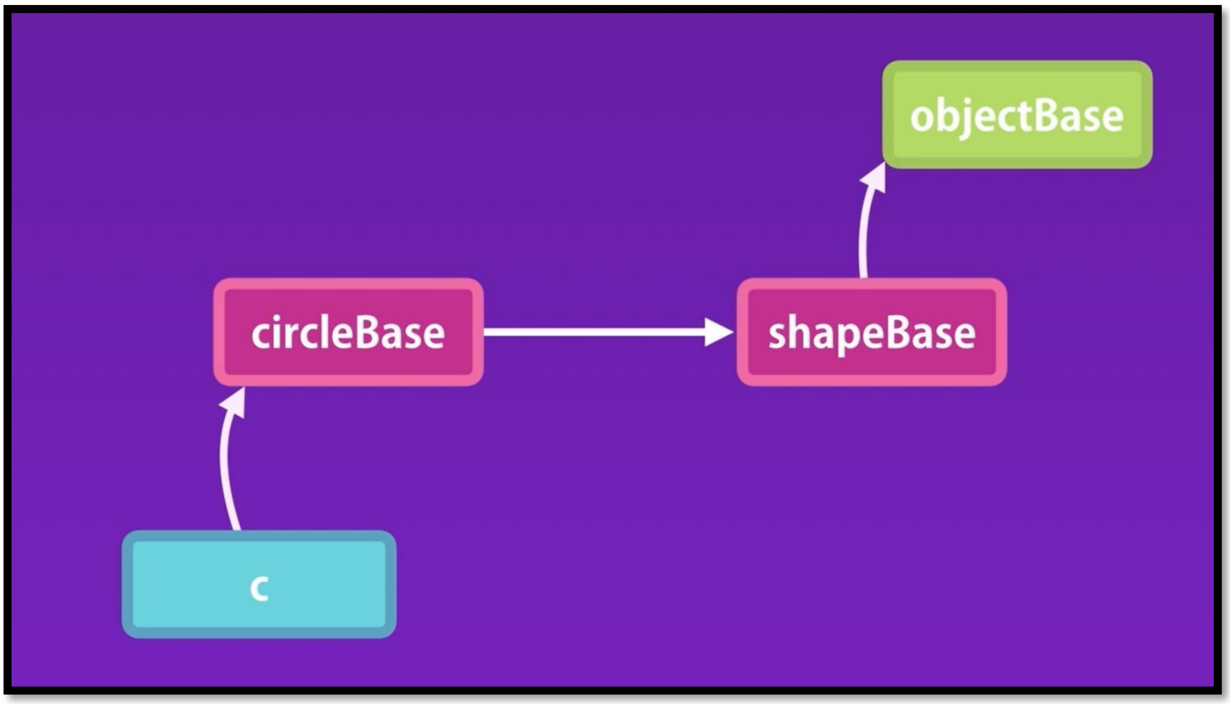
Prototypical Inheritance







```
function Shape()
{

}

Shape.prototype.duplicate = function()
{
    console.log('duplicate');
}


function Circle(radius)
{
    this.radius = radius;
}

Circle.prototype.draw = function()
{
    console.log('draw');
}

//Circle.prototype = Object.create(Object.prototype);
Circle.prototype = Object.create(Shape.prototype);

const c1 = new Circle(1);
c1.duplicate();
```

Resetting the Constructor

Problem with this Implementation

```
const c2 = new Circle.prototype.constructor(2);
console.log(c2); //Shape {}
```

Solution

```
function Shape()
{

}

Shape.prototype.duplicate = function()
{
    console.log('duplicate');
}

function Circle(radius)
{
    this.radius = radius;
}

Circle.prototype.draw = function()
{
    console.log('draw');
}

//Circle.prototype = Object.create(Object.prototype);
//Circle.prototype.constructor = Circle

Circle.prototype = Object.create(Shape.prototype);
Circle.prototype.constructor = Circle

const c1 = new Circle(1);
c1.duplicate();

const c2 = new Circle.prototype.constructor(2);
console.log(c2); //Circle { radius: 2 }
```

Another Solution

```
function Animals()
{

}

Animals.prototype.habitat = function()
{
    console.log('Animals live in group.');
}


function Cats(name)
{
    this.name = name;
}

Cats.prototype.like = function()
{
    console.log('Cats like milk.');
}

Object.setPrototypeOf(Cats.prototype, Animals.prototype);

const cat = new Cats('Kitty');
console.log(cat._proto_=== Cats.prototype); // true
console.log(cat.__proto__);
//catBase: Animals { like: [Function (anonymous)] }

console.log(cat._proto_._proto_== Animals.prototype); // true
console.log(cat.__proto__.__proto__);
//animalBase: { habitat: [Function (anonymous)] }

console.log(cat._proto_._proto_._proto_== Object.prototype);//true
console.log(cat.__proto__.__proto__.__proto__);
//objBase: [Object: null prototype] {}

console.log(cat.__proto__.__proto__.__proto__.__proto__); //null
```

Super Constructor

```
function Animals(color)
{
    this.color = color;
}

Animals.prototype.habitat = function()
{
    console.log('Animals live in group.');
}

function Cats(name, color)
{
    Animals.call(this, color); // Calling the super
    this.name = name;
}

Cats.prototype.like = function()
{
    console.log('Cats like milk.');
}

Object.setPrototypeOf(Cats.prototype, Animals.prototype);

const cat = new Cats('Kitty', 'white');
console.log(cat); //Cats { color: 'white', name: 'Kitty' }
```

Method Overriding

Sometimes, it may happen that parent’s method implementation may not be ideal for the child. In such situations we use method overriding.

```
function Animals(color)
{
    this.color = color;
}

Animals.prototype.habitat = function()
{
    console.log('Animals live in group.');
}

function Cats(name, color)
{
    Animals.call(this, color); // Calling the super
    this.name = name;
}

Cats.prototype.like = function()
{
    console.log('Cats like milk.');
}

Object.setPrototypeOf(Cats.prototype, Animals.prototype);

Cats.prototype.habitat = function()
{
    Animals.prototype.habitat.call(this);
    console.log('Cats live in group.');
}

const cat = new Cats('Kitty', 'white');
cat.habitat();
//Animals live in group.
//Cats live in group.
```

Polymorphism

Poly means many and morph means form. So, polymorphism means many forms.

Polymorphism is a key concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It enables you to write more flexible and reusable code by allowing different types of objects to be used interchangeably. Polymorphism is often associated with inheritance and method overriding.

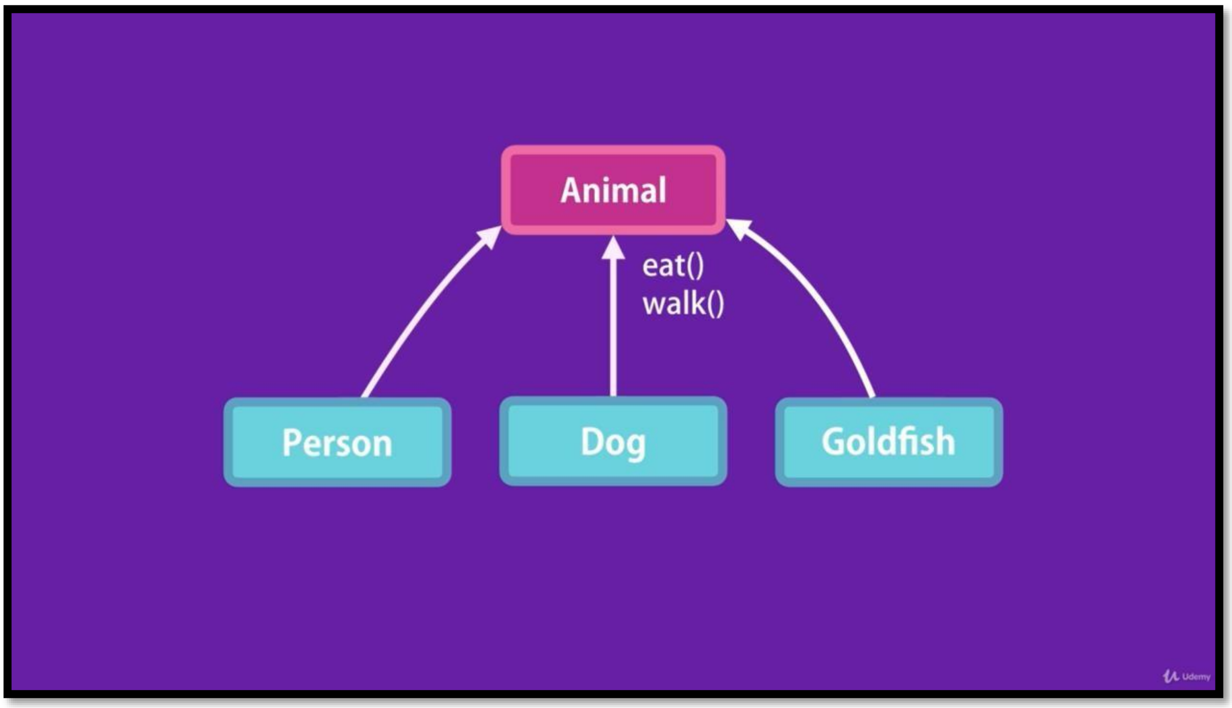
In JavaScript, polymorphism can be achieved through the concept of dynamic typing, where the type of an object is determined at runtime.

```
function Animals(color)
{
    this.color = color;
}

Animals.prototype.habitat = function()
{
    console.log('Animals live in group.');
}

function Cats(name, color)
{
    Animals.call(this, color); // Calling the super
    this.name = name;
}

Object.setPrototypeOf(Cats.prototype, Animals.prototype);

Cats.prototype.habitat = function()
{
    console.log('Cats live in group.');
}


function Dogs(name, color)
{
    Animals.call(this, color); // Calling the super
    this.name = name;
}

Object.setPrototypeOf(Dogs.prototype, Animals.prototype);

Dogs.prototype.habitat = function()
{
    console.log('Dogs live in group.');
}

const animals = [ new Cats('Kitty', 'white'), new Dogs('Dabbu', 'Black')];
```

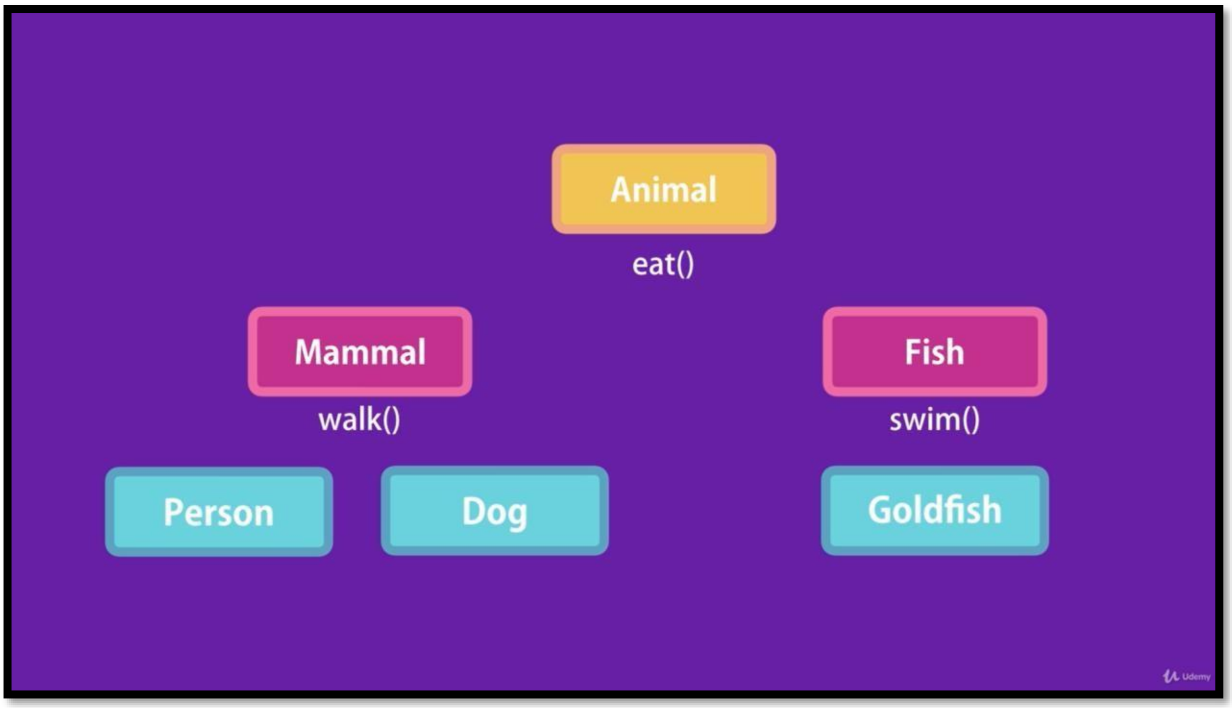
```
for(let animal of animals)
{
    animal.habitat();
}

//Cats live in group.
//Dogs live in group.
```

```
for(let animal in animals)
{
    if(animal.type === 'Dogs')
    {
        habitatDogs();
    }
    else if(animal.type === 'Cats')
    {
        habitatCats();
    }
    else
    {
        habitatAnimals();
    }
}
```

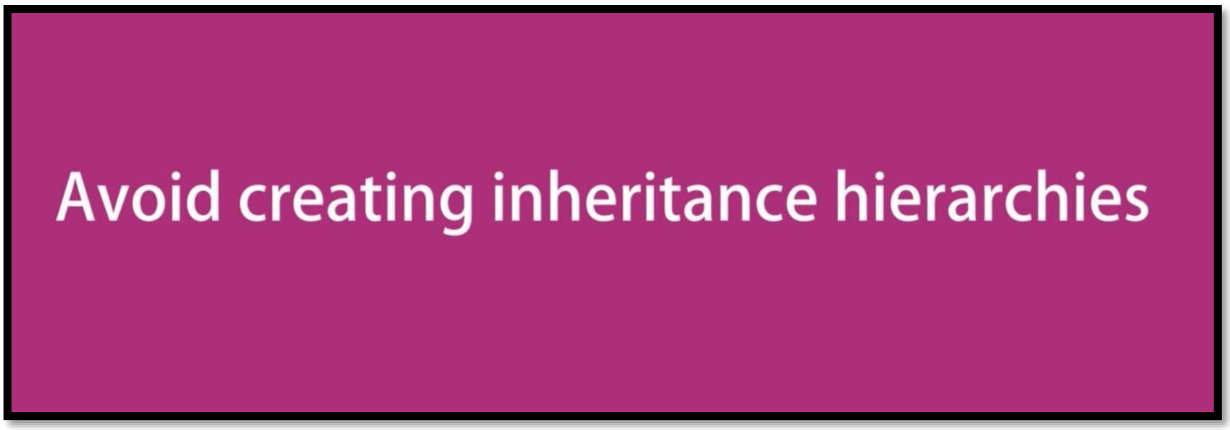
When to use Inheritance



Hierarchy gets broken because goldfish can not walk.



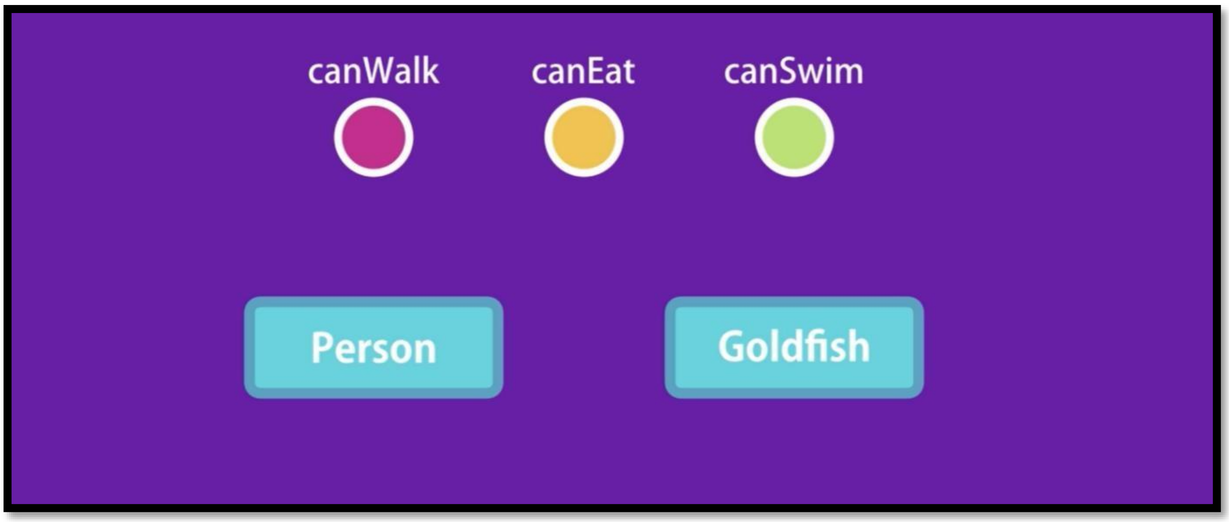
This time we even have more complex hierarchy. If we have 10 more other animals, this hierarchy will become more complex. We have to constantly go back and forth to constantly update our hierarchy.



Avoid creating inheritance hierarchies

If we want to use inheritance, keep it at one level. Don’t use inheritance for multi-levels.

Favor Composition over Inheritance



In Javascript we can use mixins to achieve composition.

Mixins

```
const canEat =
{
    eat: function()
    {
        console.log('eating');
    }
}

const canWalk =
{
    walk: function()
    {
        console.log('walking');
    }
}

const person = Object.assign({}, canEat, canWalk);
console.log(person); //{ eat: [Function: eat], walk: [Function: walk] }
```

Constructor Functions using Mixins

```
function Person()
{

}

Object.assign(Person.prototype, canEat, canWalk);

const person = new Person();
person.canEat(); //eating
person.canWalk();//walking
```

Now, suppose we have a new category of goldfish. Gold fish can eat and swim. Now, without updating the inheritance hierarchy we can add a new feature.

```
function GoldFish() {}

Object.assign(GoldFish.prototype, canEat, canSwim);

const goldFish = new GoldFish();
goldFish.canEat(); //eating
goldFish.canSwim();//swimming
```