Object Oriented JavaScript

https://www.thruskills.com

+91 831 737 5392

What you will learn

- Object basics
- Object-oriented JavaScript for beginners
- Object prototypes
- Inheritance in JavaScript
- Working with JSON data
- Object building practice

Object Basics

An object is a collection of related data and/or functionality (which usually consists of several variables and functions — which are called properties and methods when they are inside objects.)

```
var person = {
  name: ['Bob', 'Smith'],
  age: 32,
  gender: 'male',
  interests: ['music', 'skiing'],
  bio: function() {
    alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age +
    ' years old. He likes ' + this.interests[0] + ' and ' + this.interests[1] + '.');
  },
  greeting: function() {
    alert('Hi! I\'m ' + this.name[0] + '.');
  }
};
```

```
person.name[0]
person.age
person.interests[1]
person.bio()
person.greeting()
```

Dot notation

We've accessed the object's properties and methods using **dot notation**. The object name (**person**) acts as the namespace — it must be entered first to access anything encapsulated inside the object. Next you write a **dot**, then the item you want to access — this can be the name of a simple property, an item of an array property, or a call to one of the object's methods, for example:

```
person.name[0]
person.age
person.interests[1]
person.bio()
person.greeting()
```

Sub-namespaces

```
name: ['Bob', 'Smith']

name : {
  first: 'Bob',
  last: 'Smith'
}

person.name.first
person.name.last
```

Bracket notation

There is another way to access object properties — using bracket notation. Instead of using these:

```
1 person.age
2 person.name.first
3
4 person['age']
5 person['name']['first']
```

This looks very similar to how you access the items in an array, and it is basically the same thing — instead of using an index number to select an item, you are using the name associated with each member's value. It is no wonder that objects are sometimes called **associative** arrays — they map strings to values in the same way that arrays map numbers to values.

One useful aspect of bracket notation is that it can be used to set not only member values dynamically, but member names too.

```
1 var myDataName = 'height';
2 var myDataValue = '1.75m';
3 person[myDataName] = myDataValue;
4
5 person.height
```

Setting object members

So far we've only looked at retrieving (or **getting**) object members — you can also set (**update**) the value of object members by simply declaring the member you want to set (using dot or bracket notation), like this:

```
person.age = 45;
person['name']['last'] = 'Cratchit';

person.age
person['name']['last']

person['eyes'] = 'hazel';
person.farewell = function() { alert("Bye everybody!"); }
```

What is "this"?

```
1 greeting: function() {
2   alert('Hi! I\'m ' + this.name.first + '.');
3 }
```

The **this** keyword refers to the current object the code is being written inside — so in this case **this** is equivalent to **person**. So why not just write person instead? **`this`** is very useful — it will always ensure that the correct values are used when a member's context changes (e.g. two different `person` object instances may have different names, but will want to use their own name when saying their greeting).

```
 1 var person1 = {
 2   name: 'Chris',
 3   greeting: function() {
 4     alert('Hi! I\'m ' + this.name + '.');
 5   }
 6 }
 7
 8 var person2 = {
 9   name: 'Brian',
10   greeting: function() {
11     alert('Hi! I\'m ' + this.name + '.');
12   }
13 }
```

Object-oriented JavaScript for beginners

- [x] Object-oriented programming — the basics
- [x] Constructors and object instances
- [x] Other ways to create object instances

Object-oriented programming — the basics

The basic idea of OOP is that we use objects to model real world things that we want to represent inside our programs, and/or provide a simple way to access functionality that would otherwise be hard or impossible to make use of.

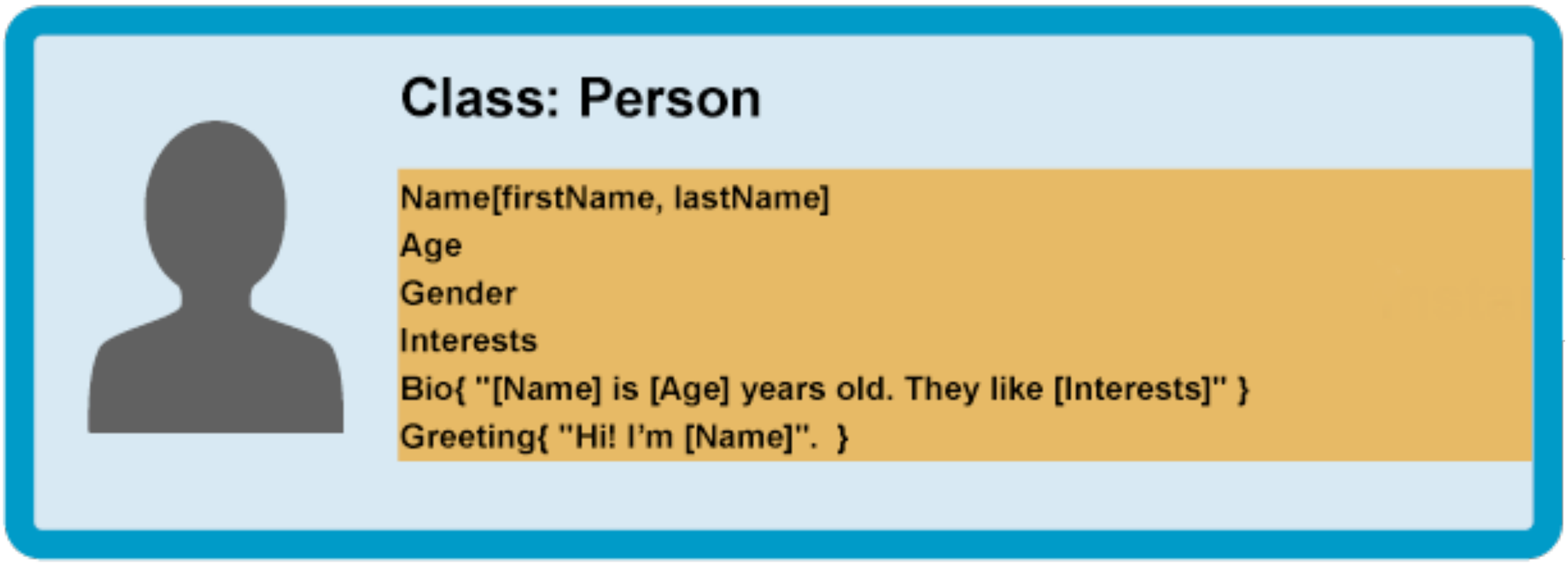
Objects can contain related data and code, which represent information about the thing you are trying to model, and functionality or behaviour that you want it to have.

Object data (and often, functions too) can be stored neatly (the official word is encapsulated) inside an object package (which can be given a specific name to refer to, which is sometimes called a namespace), making it easy to structure and access; objects are also commonly used as data stores that can be easily sent across the network.

Defining an object template

Let's consider a simple program that displays information about the students and teachers at a school. Here we'll look at OOP theory in general, not in the context of any specific programming language.

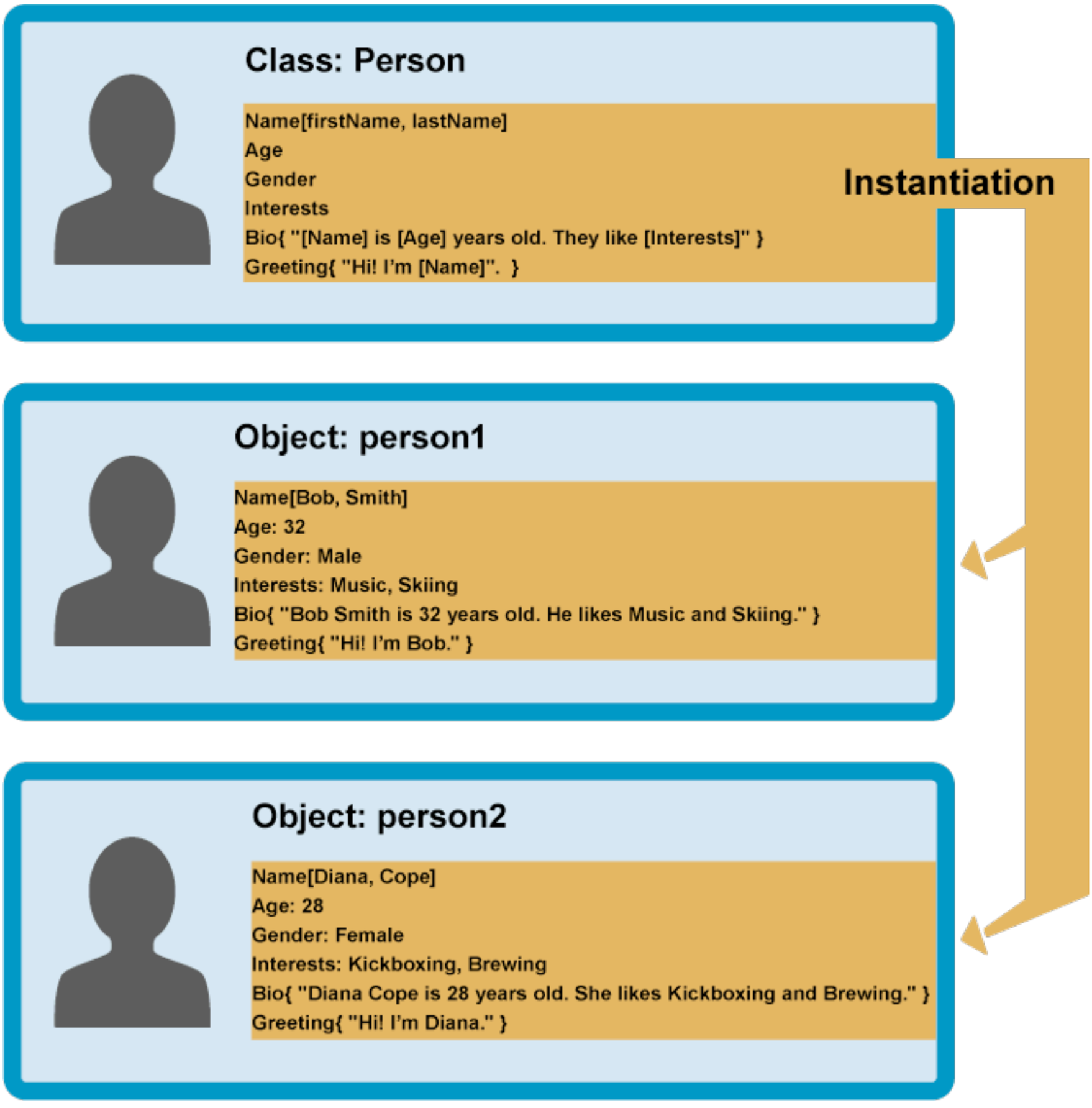
This is known as **abstraction** — creating a simple model of a more complex thing, which represents its most important aspects in a way that is easy to work with for our program's purposes.



Creating actual objects

From our class, we can create **object instances** — objects that contain the data and functionality defined in the class. From our Person class, we can now create some actual people.

When an object instance is created from a class, the class's **constructor function** is run to create it. This process of creating an object instance from a class is called **instantiation** — the object instance is **instantiated** from the class.

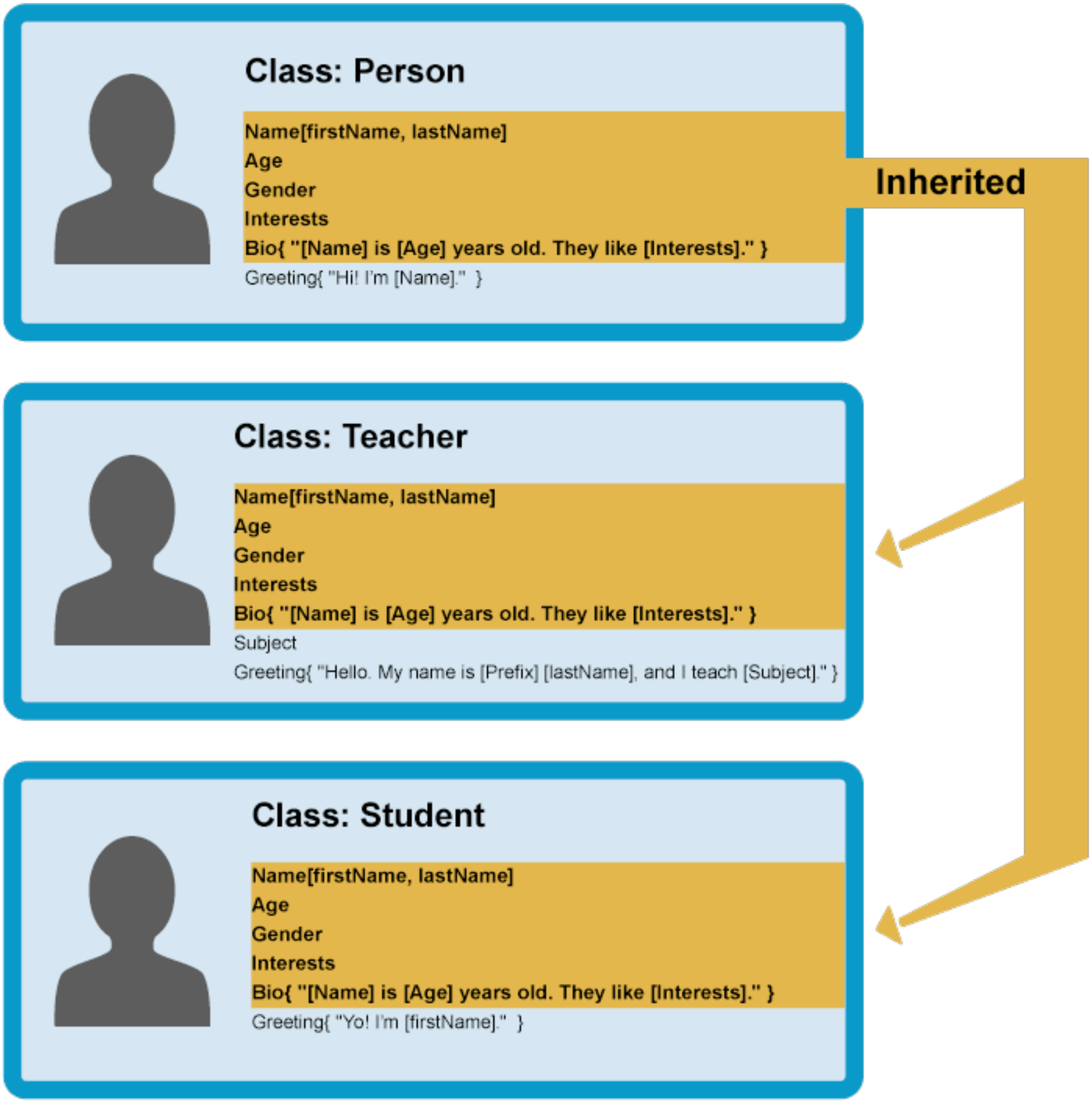


Specialist classes

In this case we don't want generic people — we want teachers and students, which are both more specific types of people.

In OOP, we can create new classes based on other classes — these new child classes can be made to inherit the data and code features of their parent class, so you can reuse functionality common to all the object types rather than having to duplicate it.

Where functionality differs between classes, you can define **specialized** features directly on them as needed.



Constructors and object instances

- Some people argue that JavaScript is not a true object-oriented language — for example, its class statement is just syntactical sugar over existing prototypical inheritance and is not a class in a traditional sense.
- JavaScript, uses special functions called **constructor functions** to define objects and their features.
- They are useful because you'll often come across situations in which you don't know how many objects you will be creating; constructors provide the means to create as many objects as you need in an effective way, attaching data and functions to them as required.
- When a new object instance is created from a constructor function, its core functionality (as defined by its prototype) is not all copied over to the new object like "classic" OO languages — instead the functionality is linked to via a reference chain called a prototype chain.
- So this is not true instantiation, strictly speaking — JavaScript uses a different mechanism to share functionality between objects.

A simple example

```
function createNewPerson(name) {
  var obj = {};
  obj.name = name;
  obj.greeting = function() {
    alert('Hi! I\'m ' + this.name + '.');
  };
  return obj;
}
```

```
var salva = createNewPerson('Salva');
salva.name;
salva.greeting();
```

This works well enough, but it is a bit long-winded; if we know we want to create an object, why do we need to explicitly create a new empty object and return it? Fortunately JavaScript provides us with a handy shortcut, in the form of constructor functions — let's make one now!

The constructor function

The constructor function is JavaScript's version of a class. You'll notice that it has all the features you'd expect in a function, although it doesn't return anything or explicitly create an object — it basically just defines properties and methods.

You'll see the this keyword being used here as well — it is basically saying that whenever one of these object instances is created, the object's name property will be equal to the name value passed to the constructor call, and the greeting() method will use the name value passed to the constructor call too.

A constructor function name usually starts with a capital letter — this convention is used to make constructor functions easier to recognize in code.

```
function Person(name) {
  this.name = name;
  this.greeting = function() {
    alert('Hi! I\'m ' + this.name + '.');
  };
}
```

```
var person1 = new Person('Bob');
var person2 = new Person('Sarah');
```

Creating our finished constructor

```
function Person(first, last, age, gender, interests) {
  this.name = {
    'first': first,
    'last' : last
  };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
  this.bio = function() {
    alert(this.name.first + ' ' + this.name.last + ' is ' + this.age + ' years old.
    He likes ' + this.interests[0] + ' and ' + this.interests[1] + '.');
  };
  this.greeting = function() {
    alert('Hi! I\'m ' + this.name.first + '.');
  };
}
```

```
var person1 = new Person('Bob', 'Smith', 32, 'male', ['music', 'skiing']);
```

Other ways to create object instances

So far we've seen two different ways to create an object instance:

- declaring an object literal - { }
- using a constructor function

The Object() constructor

First of all, you can use the Object() constructor to create a new object. Yes, even generic objects have a constructor, which generates an empty object.

```
var person1 = new Object();
```

```
person1.name = 'Chris';
person1['age'] = 38;
person1.greeting = function() {
  alert('Hi! I\'m ' + this.name + '.');
};
```

```
var person1 = new Object({
  name: 'Chris',
  age: 38,
  greeting: function() {
    alert('Hi! I\'m ' + this.name + '.');
  }
});
```

Using the create() method

Constructors can help you give your code order—you can create constructors in one place, then create instances as needed, and it is clear where they came from.

However, some people prefer to create object instances without first creating constructors, especially if they are creating only a few instances of an object. JavaScript has a built-in method called create() that allows you to do that. With it, you can create a new object based on any existing object.

```
1 | var person2 = Object.create(person1);
```

```
1 | person2.name
2 | person2.greeting()
```

Object prototypes

Prototypes are the mechanism by which JavaScript objects inherit features from one another, and they work differently than **inheritance** mechanisms in classical object-oriented programming languages.

A prototype-based language?

JavaScript is often described as a prototype-based language — each object has a prototype object, which acts as a template object that it inherits methods and properties from.

An object's prototype object may also have a prototype object, which it inherits methods and properties from, and so on.

This is often referred to as a prototype chain, and explains why different objects have properties and methods defined on other objects available to them.

The properties and methods are defined on the prototype property on the Objects' constructor functions, not the object instances themselves.

__proto__

In classic OOP, classes are defined, then when object instances are created all the properties and methods defined on the class are copied over to the instance.

In JavaScript, they are not copied over — instead, a link is made between the object instance and its prototype (its __proto__ property, which is derived from the prototype property on the constructor), and the properties and methods are found by walking up the chain of prototypes.

Note: It's important to understand that there is a distinction between an object's prototype (which is available via **Object.getPrototypeOf(obj)**, or via the deprecated **__proto__** property) and the prototype property on constructor functions.

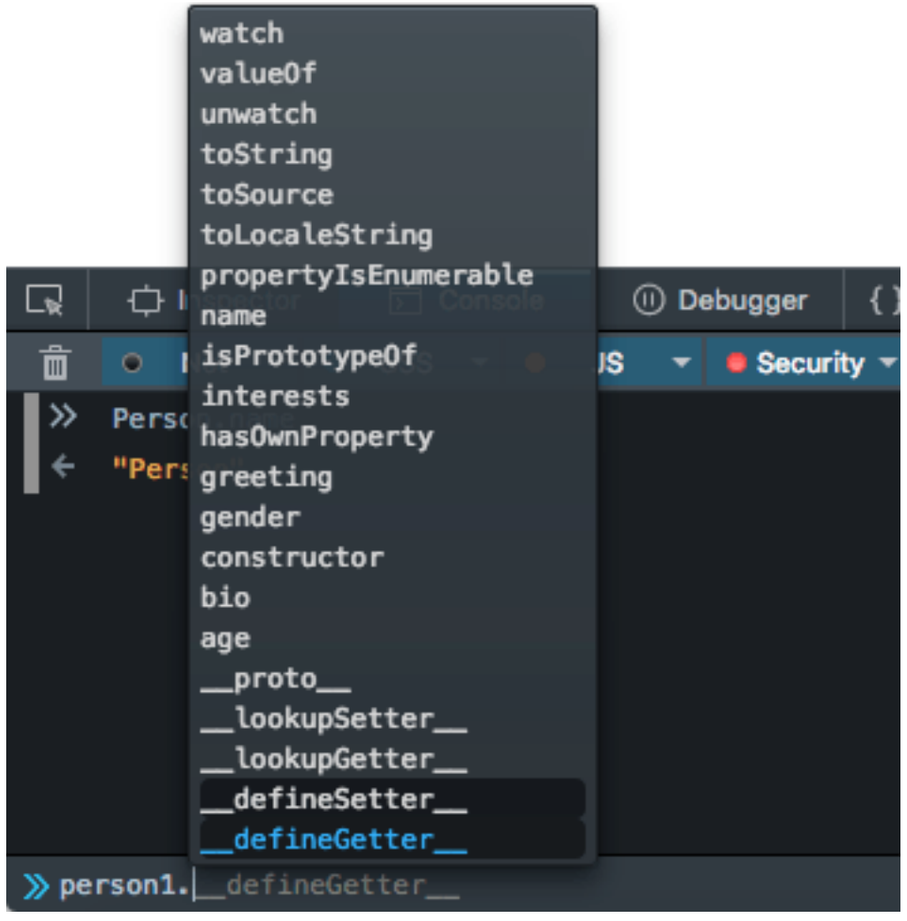
The former is the property on each instance, and the latter is the property on the constructor. That is, **Object.getPrototypeOf**(new Foobar()) refers to the same object as Foobar.prototype.

Understanding prototype objects

```
function Person(first, last, age, gender, interests) {

  // property and method definitions
  this.first = first;
  this.last = last;
//...
}
```

```
var person1 = new Person('Bob', 'Smith', 32, 'male', ['music', 'skiing']);
```



In this list, you will see the members defined on person1's prototype object, which is the Person() (Person() is the constructor) — name, age, gender, interests, bio, and greeting. You will however also see some other members — watch, valueOf, etc — these are defined on the Person() 's prototype object, which is Object. This demonstrates the prototype chain working.

Understanding prototype objects...

So what happens if you call a method on person1, which is actually defined on Object? For example:

```
1 | person1.valueOf()
```

- The browser initially checks to see if the person1 object has a valueOf() method available on it.
- It doesn't, so the browser then checks to see if the person1 object's prototype object (Person() constructor's prototype) has a valueOf() method available on it.
- It doesn't either, so the browser then checks to see if the Person() constructor's prototype object's prototype object (Object() constructor's prototype) has a valueOf() method available on it. It does, so it is called, and all is good!

The prototype property:

Where inherited members are defined

- So, where are the inherited properties and methods defined? If you look at the Object reference page, you'll see listed in the left hand side a large number of properties and methods — many more than the number of inherited members we saw available on the person1 object in the above screenshot. Some are inherited, and some aren't — why is this?
- The answer is that the inherited ones are the ones defined on the prototype property (you could call it a sub-namespace) — that is, the ones that begin with Object.prototype., and not the ones that begin with just Object.
- The prototype property's value is an object, which is basically a bucket for storing properties and methods that we want to be inherited by objects further down the prototype chain.
- So Object.prototype.watch(), Object.prototype.valueOf(), etc., are available to any object types that inherit from Object.prototype, including new object instances created from the constructor.

The prototype property...

Object.is(), Object.keys(), and other members not defined inside the prototype bucket are not inherited by object instances or object types that inherit from Object.prototype. They are methods/properties available just on the Object() constructor itself.

This seems strange — how can you have a method defined on a constructor, which is itself a function? Well, a function is also a type of object — see the Function() constructor reference if you don't believe us.

```
Person.prototype
```

```
Object.prototype
```

Important: The `prototype` property is one of the most confusingly-named parts of JavaScript — you might think that `this` points to the prototype object of the current object, but it doesn't (that's an internal object that can be accessed by `__proto__`, remember?). `prototype` instead is a property containing an object on which you define members that you want to be inherited.

Revisiting create()

Earlier on we showed how the Object.create() method can be used to create a new object instance.

```
var person2 = Object.create(person1);
```

What **create**() actually does is to create a new object from a specified prototype object.

Here, person2 is being created using person1 's prototype as a prototype object. You can check this by entering the following in the console:

```
person2.__proto__
```

The constructor property

Every constructor function has a prototype property whose value is an object containing a constructor property.

This constructor property points to the original constructor function. As you will see in the next section that properties defined on the Person.prototype property (or in general on a constructor function's prototype property, which is an object, as mentioned in the above section) become available to all the instance objects created using the Person() constructor.

Hence, the constructor property is also available to both person1 and person2 objects.

```
1 var person3 = new person1.constructor(
2 'Karen', 'Stephenson', 26, 'female', ['playing drums', 'mountain climbing']);
```

```
1 person3.name.first
2 person3.age
3 person3.bio()
```

Modifying prototypes

Let's have a look at an example of modifying the prototype property of a constructor function (methods added to the prototype are then available on all object instances created from the constructor).

```
Person.prototype.farewell = function() {
  alert(this.name.first + ' has left the building. Bye for now!');
};
```

```
person1.farewell();
```

```
function Person(first, last, age, gender, interests) {

  // property and method definitions

}

var person1 = new Person('Tammi', 'Smith', 32, 'neutral', ['music', 'skiing', 'kickboxing']);

Person.prototype.farewell = function() {
  alert(this.name.first + ' has left the building. Bye for now!');
};
```

Modifying prototypes…

```
1  // Constructor with property definitions
2
3  function Test(a, b, c, d) {
4    // property definitions
5  }
6
7  // First method definition
8
9  Test.prototype.x = function() { ... };
10
11 // Second method definition
12
13 Test.prototype.y = function() { ... };
14
15 // etc.
```

Inheritance in JavaScript

Prototypal inheritance

So far we have seen some inheritance in action — we have seen how prototype chains work, and how members are inherited going up a chain. But mostly this has involved built-in browser functions. How do we create an object in JavaScript that inherits from another object?

As mentioned earlier in the course, some people think JavaScript is not a true object-oriented language. In "classic OO" languages, you tend to define class objects of some kind, and you can then simply define which classes inherit from which other classes (see C++ inheritance for some simple examples). JavaScript uses a different system — "inheriting" objects do not have functionality copied over to them, instead the functionality they inherit is linked to via the prototype chain (often referred to as prototypal inheritance).

Inheritance in JavaScript

```
function Person(first, last, age, gender, interests) {
  this.name = {
    first,
    last
  };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
};
```

```
Person.prototype.greeting = function() {
  alert('Hi! I\'m ' + this.name.first + '.');
};
```

Inheritance in JavaScript

Say we wanted to create a Teacher class, like the one we described in our initial object-oriented definition, which inherits all the members from Person, but also includes:

- A new property, subject — this will contain the subject the teacher teaches.
- An updated greeting() method, which sounds a bit more formal than the standard greeting() method — more suitable for a teacher addressing some students at school.

```
function Teacher(first, last, age, gender, interests, subject) {
  Person.call(this, first, last, age, gender, interests);

  this.subject = subject;
}
```

Inheriting from a constructor with no parameters

Note that if the constructor you are inheriting from doesn't take its property values from parameters, you don't need to specify them as additional arguments in call(). So, for example, if you had something really simple like this:

```
function Brick() {
  this.width = 10;
  this.height = 20;
}
```

```
function BlueGlassBrick() {
  Brick.call(this);

  this.opacity = 0.5;
  this.color = 'blue';
}
```

Setting Teacher()'s prototype

All is good so far, but we have a problem. We have defined a new constructor, and it has a prototype property, which by default just contains a reference to the constructor function itself. It does not contain the methods of the Person constructor's prototype property.

To see this, enter Object.getOwnPropertyNames(Teacher.prototype) into either the text input field or your JavaScript console. Then enter it again, replacing Teacher with Person. Nor does the new constructor inherit those methods. To see this, compare the outputs of Person.prototype.greeting and Teacher.prototype.greeting. We need to get Teacher() to inherit the methods defined on Person()'s prototype. So how do we do that?

```
1 | Teacher.prototype = Object.create(Person.prototype);
```

```
1 | Teacher.prototype.constructor = Teacher;
```

Giving Teacher() a new greeting() function

To finish off our code, we need to define a new greeting() function on the Teacher() constructor. The easiest way to do this is to define it on Teacher()'s prototype — add the following at the bottom of your code:

```
Teacher.prototype.greeting = function() {
  var prefix;

  if (this.gender === 'male' || this.gender === 'Male' || this.gender === 'm' || this.gender
=== 'M') {
    prefix = 'Mr.';
  } else if (this.gender === 'female' || this.gender === 'Female' || this.gender === 'f' ||
this.gender === 'F') {
    prefix = 'Mrs.';
  } else {
    prefix = 'Mx.';
  }

  alert('Hello. My name is ' + prefix + ' ' + this.name.last + ', and I teach ' +
this.subject + '.');
};
```

Any Questions

Thank You

https://www.thruskills.com

+91 831 737 5392

thruskills