Object-oriented programming — the basicsSection

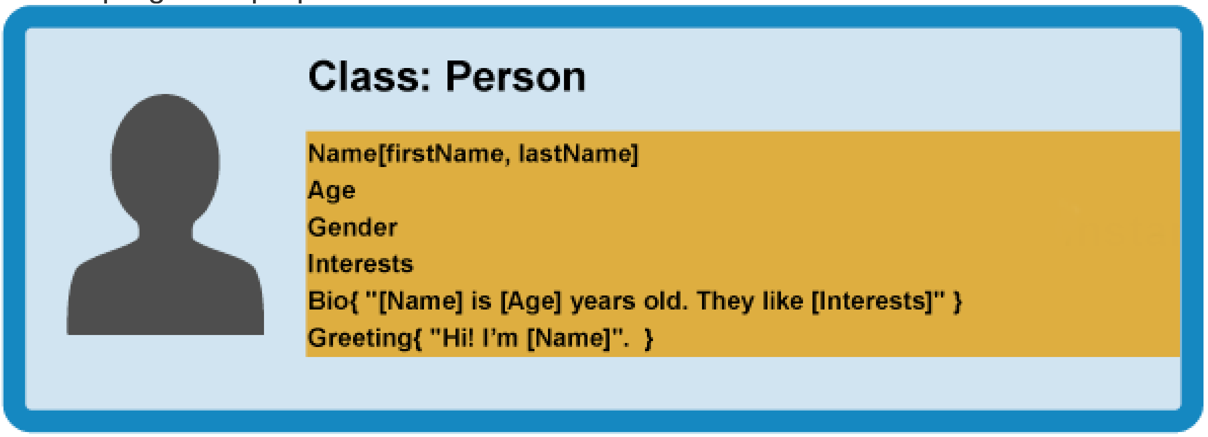
To start with, let's give you a simplistic, high-level view of what Object-oriented programming (OOP) is. We say simplistic, because OOP can quickly get very complicated, and giving it a full treatment now would probably confuse more than help. The basic idea of OOP is that we use objects to model real 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 behavior 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 templateSection

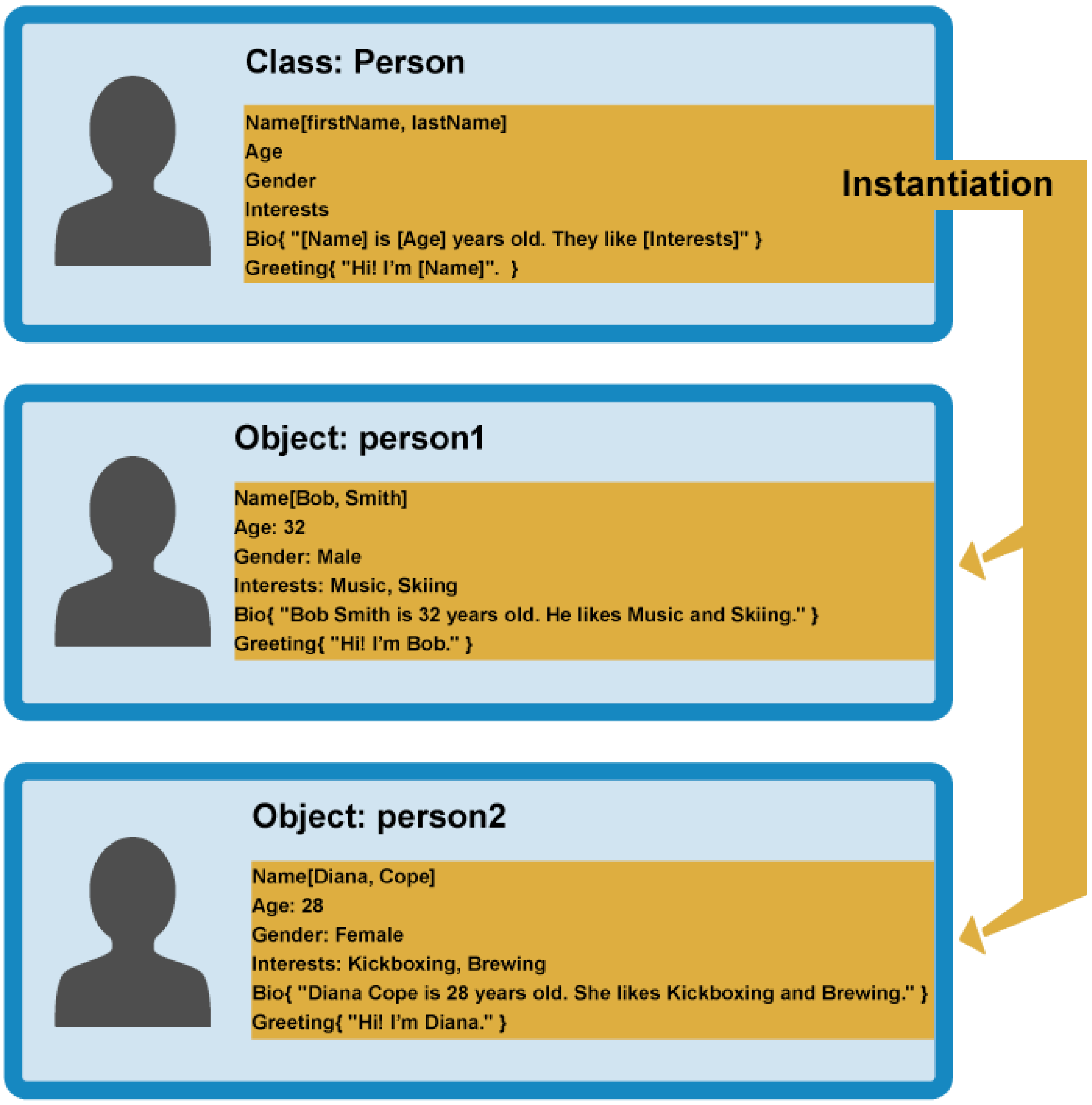
Let's consider a simple program that displays information about the students and teachers at a school. Here we'll look at OOP theory in general, not in the context of any specific programming language.

To start this off, we could return to our Person object type from our first objects article, which defines the generic data and functionality of a person. There are lots of things you *could* know about a person (their address, height, shoe size, DNA profile, passport number, significant personality traits ...) , but in this case we are only interested in showing their name, age, gender, and interests, and we also want to be able to write a short introduction about them based on this data, and get them to say hello. 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.


Class: Person
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]" }
Greeting{ "Hi! I'm [Name]". }

Creating actual objectsSection

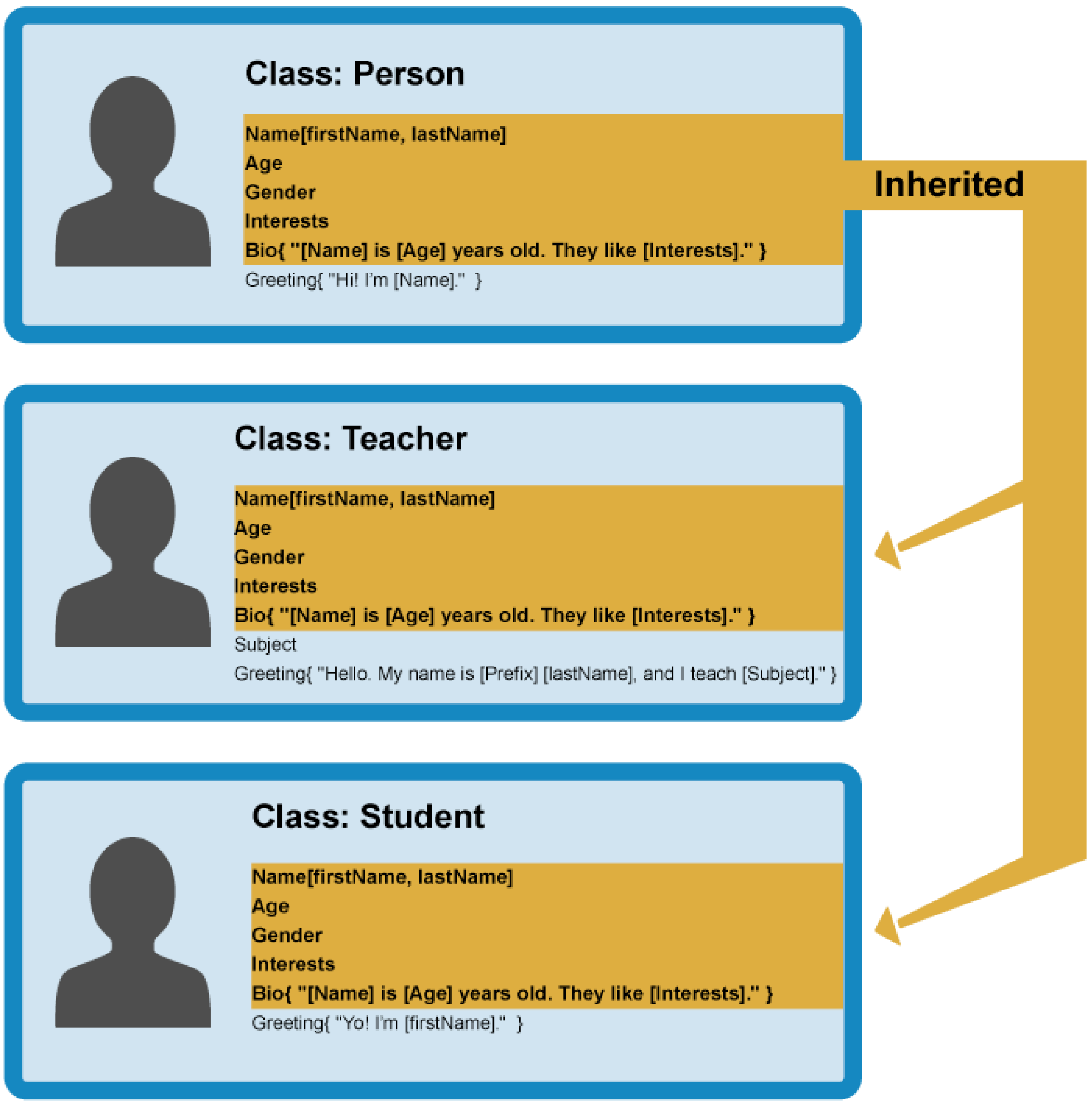
From our class, we can create **object instances** — objects that contain the data and functionality defined in the class. From our Person class, we can now create some actual people:


Class: Person
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]" }
Greeting{ "Hi! I'm [Name]". }
Instantiation
Object: person1
Name[Bob, Smith]
Age: 32
Gender: Male
Interests: Music, Skiing
Bio{ "Bob Smith is 32 years old. He likes Music and Skiing." }
Greeting{ "Hi! I'm Bob." }
Object: person2
Name[Diana, Cope]
Age: 28
Gender: Female
Interests: Kickboxing, Brewing
Bio{ "Diana Cope is 28 years old. She likes Kickboxing and Brewing." }
Greeting{ "Hi! I'm Diana." }

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 classesSection

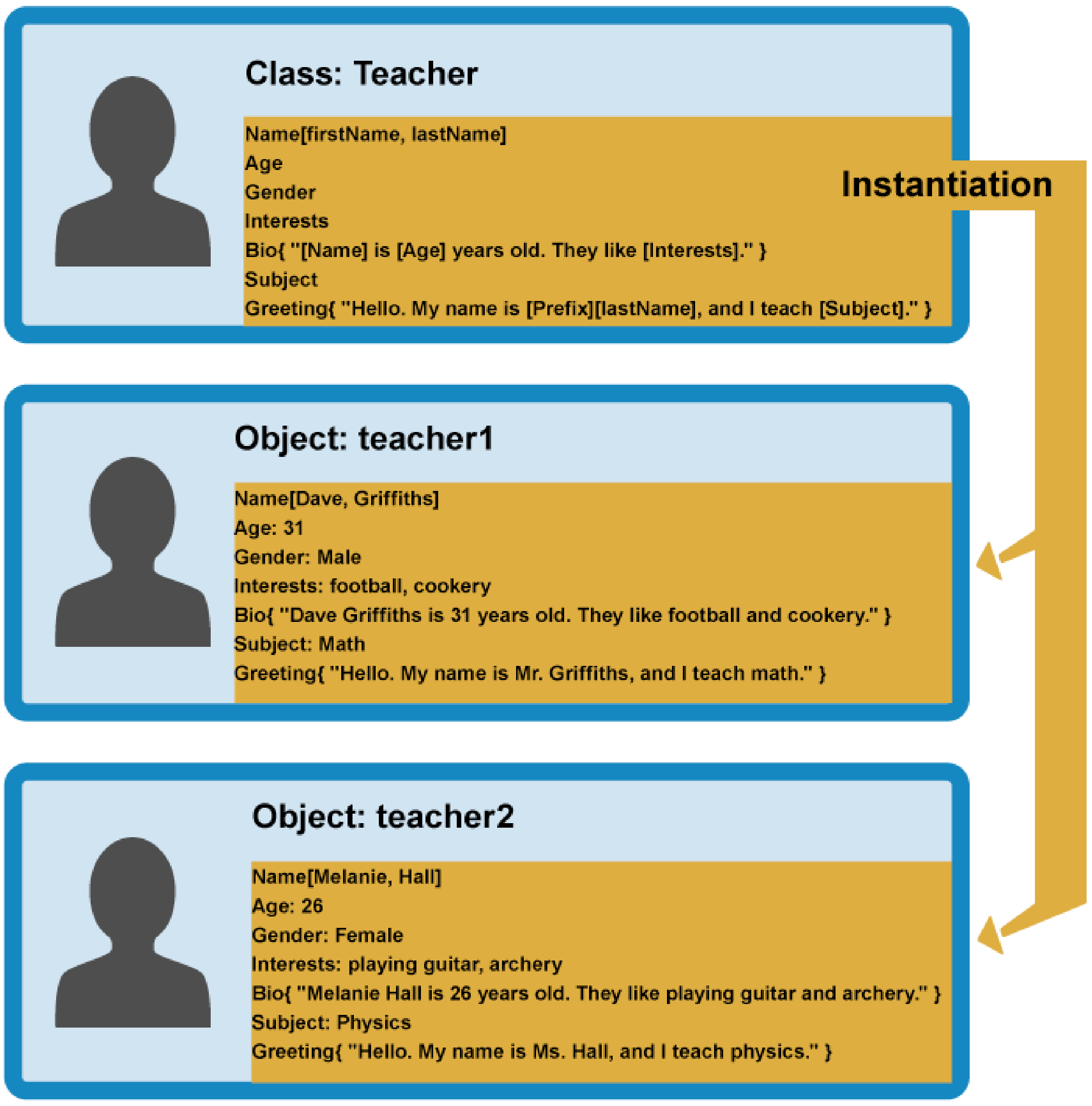
In this case we don't want generic people — we want teachers and students, which are both more specific types of people. In OOP, we can create new classes based on other classes — these new **child classes** can be made to **inherit** the data and code features of their **parent class**, so you can reuse 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.


Class: Person
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]." }
Greeting{ "Hi! I'm [Name]." }
Inherited
Class: Teacher
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]." }
Subject
Greeting{ "Hello. My name is [Prefix] [lastName], and I teach [Subject]." }
Class: Student
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]." }
Greeting{ "Yo! I'm [firstName]." }

This is really useful — teachers and students share many common features such as name, gender, and age, so it is convenient to only have to define those features once. You can also define the same feature separately in different classes, as each definition of that feature will be in a different namespace. For example, a student's greeting might be of the form "Yo, I'm [firstName]" (e.g *Yo, I'm Sam*), whereas a teacher might use something more formal, such as "Hello, my name is [Prefix] [lastName], and I teach [Subject]." (e.g *Hello, My name is Mr Griffiths, and I teach Chemistry*).

Note: The fancy word for the ability of multiple object types to implement the same functionality is **polymorphism**. Just in case you were wondering.

You can now create object instances from your child classes. For example:


Class: Teacher
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]." }
Subject
Greeting{ "Hello. My name is [Prefix][lastName], and I teach [Subject]." }
Instantiation
Object: teacher1
Name[Dave, Griffiths]
Age: 31
Gender: Male
Interests: football, cookery
Bio{ "Dave Griffiths is 31 years old. They like football and cookery." }
Subject: Math
Greeting{ "Hello. My name is Mr. Griffiths, and I teach math." }
Object: teacher2
Name[Melanie, Hall]
Age: 26
Gender: Female
Interests: playing guitar, archery
Bio{ "Melanie Hall is 26 years old. They like playing guitar and archery." }
Subject: Physics
Greeting{ "Hello. My name is Ms. Hall, and I teach physics." }

In the rest of the article, we'll start to look at how OOP theory can be put into practice in JavaScript.

Constructors and object instancesSection

JavaScript uses special functions called **constructor functions** to define and initialize 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.

Let's explore creating classes via constructors and creating object instances from them in JavaScript. First of all, we'd like you to make a new local copy of the oojs.html file we saw in our first Objects article.

A simple exampleSection

1. Let's start by looking at how you could define a person with a normal function. Add this function within the `script` element:

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

```
}
```

9. You can now create a new person by calling this function — try the following lines in your browser's JavaScript console:

```
const 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!

12. Replace your previous function with the following:

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

```
}
```

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.

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

So how do we call a constructor to create some objects?

1. Add the following lines below your previous code addition:

```
2. const person1 = new Person('Bob');
```

```
const person2 = new Person('Sarah');
```

3. Save your code and reload it in the browser, and try entering the following lines into your JS console:

```
4. person1.name
5. person1.greeting()
6. person2.name
```

```
person2.greeting()
```

Cool! You'll now see that we have two new objects on the page, each of which is stored under a different namespace — when you access their properties and methods, you have to start calls with `person1` or `person2`; the functionality contained within is neatly packaged away so it won't clash with other functionality. They do, however, have the same `name` property and `greeting()` method available. Note that they are using their own `name` value that was assigned to them when they were created; this is one reason why it is very important to use `this`, so they will use their own values, and not some other value.
Let's look at the constructor calls again:

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

In each case, the `new` keyword is used to tell the browser we want to create a new object instance, followed by the function name with its required parameters contained in parentheses, and the result is stored in a variable — very similar to how a standard function is called. Each instance is created according to this definition:

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

After the new objects have been created, the `person1` and `person2` variables contain the following objects:

```
{
  name: 'Bob',
  greeting: function() {
    alert('Hi! I\'m ' + this.name + '.');
  }
}
```

```
{
  name: 'Sarah',
  greeting: function() {
    alert('Hi! I\'m ' + this.name + '.');
  }
}
```

Note that when we are calling our constructor function, we are defining `greeting()` every time, which isn't ideal. To avoid this, we can define functions on the prototype instead, which we will look at later.

Creating our finished constructorSection

The example we looked at above was only a simple example to get us started. Let's now get on and create our final `Person()` constructor function.

1. Remove the code you inserted so far, and add in this replacement constructor — this is exactly the same as the simple example in principle, with just a bit more complexity:

```
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 + '.');
  };
}
```

16. Now add in the following line below it, to create an object instance from it:

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

You'll now see that you can access the properties and methods just like we did previously — try these in your JS console:

```
person1['age']
person1.interests[1]
person1.bio()
// etc.
```

Note: If you are having trouble getting this to work, try comparing your code against our version — see oojs-class-finished.html (also see it running live).

Further exercisesSection

To start with, try adding a couple more object creation lines of your own, and try getting and setting the members of the resulting object instances.

In addition, there are a couple of problems with our `bio()` method — the output always includes the pronoun "He", even if your person is female, or some other preferred gender classification. And the bio will only include two interests, even if more are listed in the `interests` array. Can you work out how to fix this in the class definition (constructor)? You can put any code you like inside a constructor (you'll probably need a few conditionals and a loop). Think about how the sentences should be structured differently depending on gender, and depending on whether the number of listed interests is 1, 2, or more than 2.

Note: If you get stuck, we have provided an answer inside our GitHub repo (see it live) — try writing it yourself first though!

Other ways to create object instancesSection

So far we've seen two different ways to create an object instance — declaring an object literal, and using a constructor function (see above).
These make sense, but there are other ways — we want to make you familiar with these in case you come across them in your travels around the Web.

The Object() constructorSection

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.

1. Try entering this into your browser's JavaScript console:

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

2. This stores an empty object in the `person1` variable. You can then add properties and methods to this object using dot or bracket notation as desired; try these examples in your console:

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

```
};
```

7. You can also pass an object literal to the `Object()` constructor as a parameter, to prefill it with properties/methods. Try this in your JS console:

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

Using the create() methodSection

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. With your finished exercise from the previous sections loaded in the browser, try this in your JavaScript console:

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

2. Now try these:

```
3. person2.name
```

```
person2.greeting()
```

You'll see that `person2` has been created based on `person1`—it has the same properties and method available to it.
One limitation of `create()` is that IE8 does not support it. So constructors may be more effective if you want to support older browsers.
We'll explore the effects of `create()` in more detail later on.

Object-oriented programming — the basicsSection

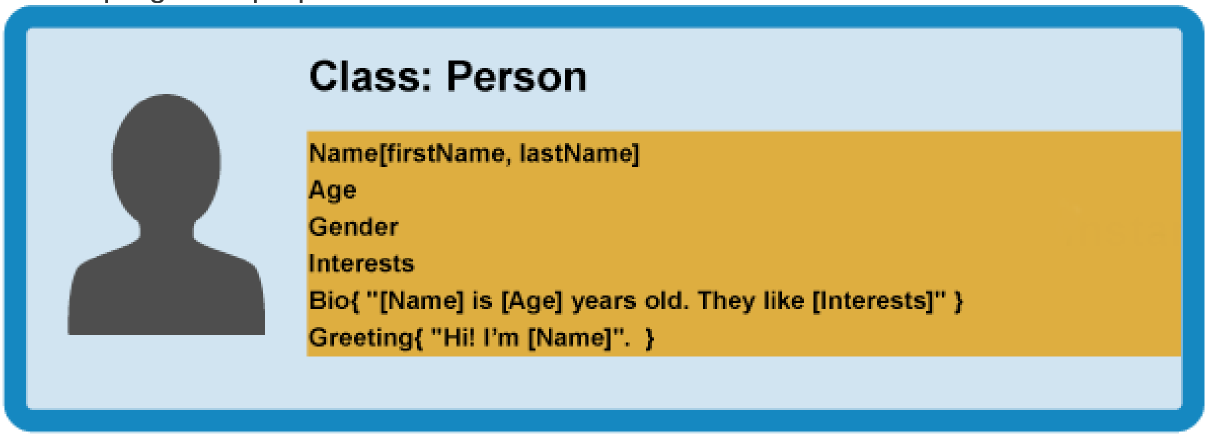
To start with, let's give you a simplistic, high-level view of what Object-oriented programming (OOP) is. We say simplistic, because OOP can quickly get very complicated, and giving it a full treatment now would probably confuse more than help. The basic idea of OOP is that we use objects to model real 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 behavior 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 templateSection

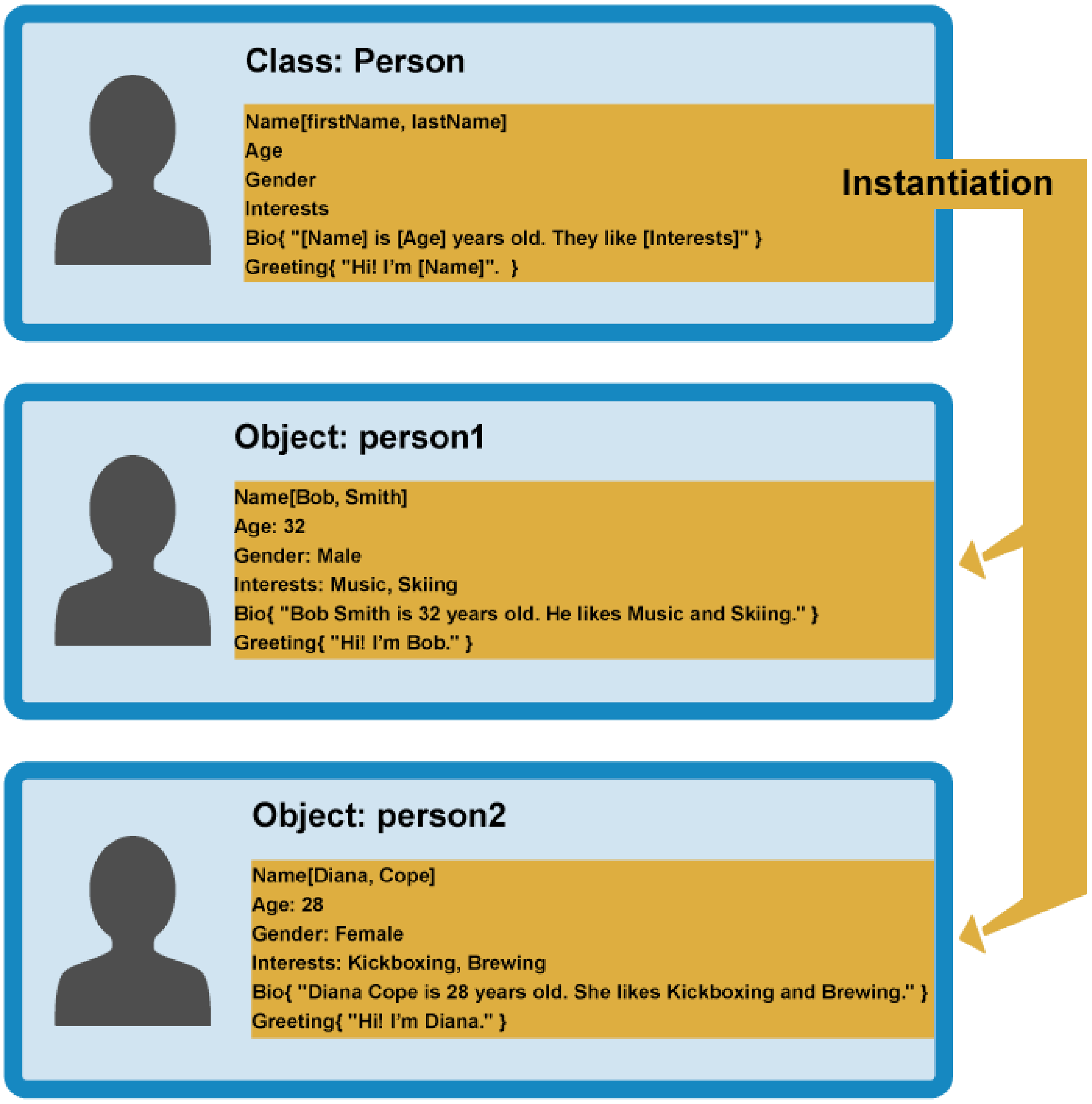
Let's consider a simple program that displays information about the students and teachers at a school. Here we'll look at OOP theory in general, not in the context of any specific programming language.

To start this off, we could return to our Person object type from our first objects article, which defines the generic data and functionality of a person. There are lots of things you *could* know about a person (their address, height, shoe size, DNA profile, passport number, significant personality traits ...) , but in this case we are only interested in showing their name, age, gender, and interests, and we also want to be able to write a short introduction about them based on this data, and get them to say hello. 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.


Class: Person
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]" }
Greeting{ "Hi! I'm [Name]". }

Creating actual objectsSection

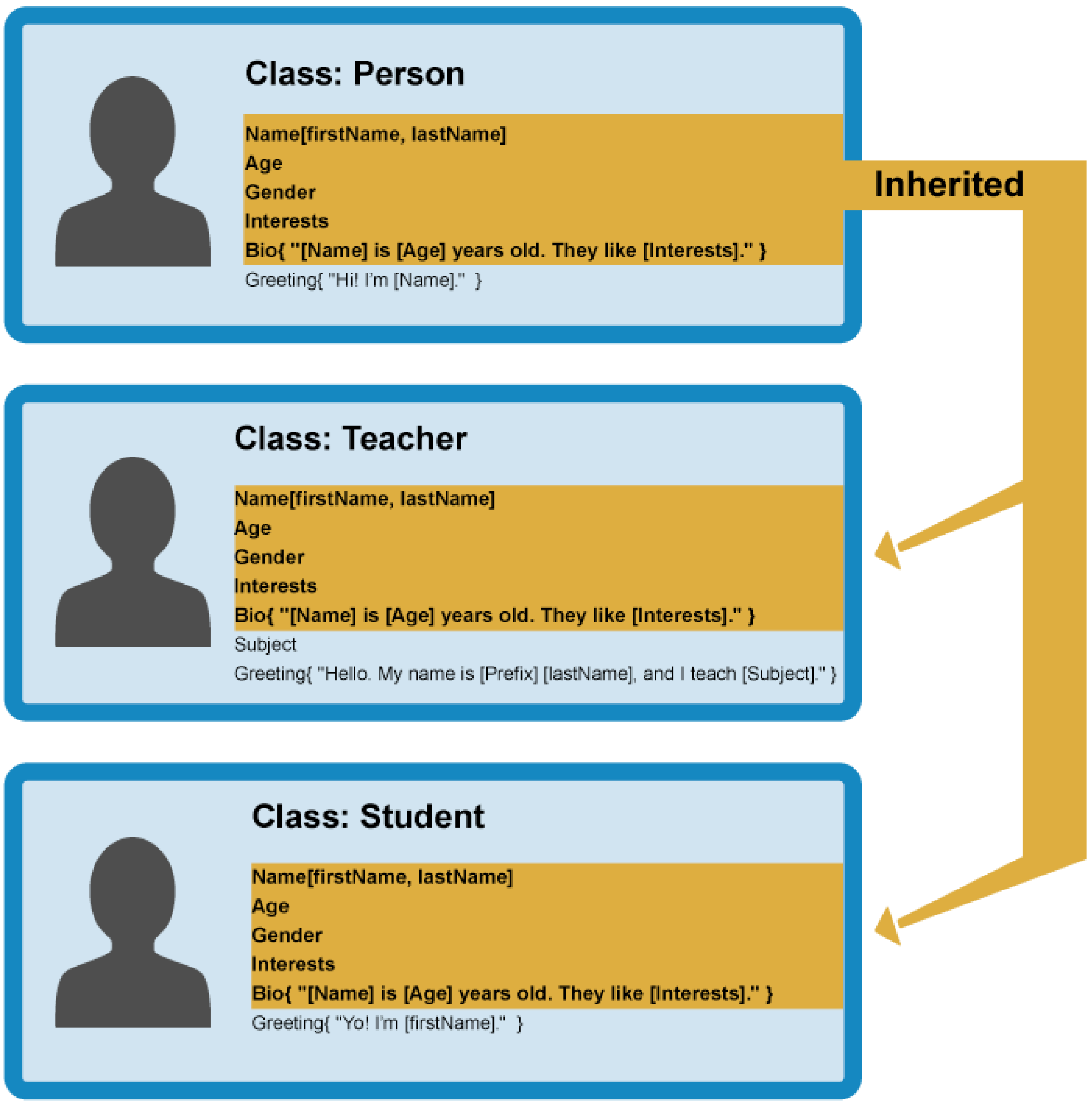
From our class, we can create **object instances** — objects that contain the data and functionality defined in the class. From our Person class, we can now create some actual people:


Class: Person
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]" }
Greeting{ "Hi! I'm [Name]". }
Instantiation
Object: person1
Name[Bob, Smith]
Age: 32
Gender: Male
Interests: Music, Skiing
Bio{ "Bob Smith is 32 years old. He likes Music and Skiing." }
Greeting{ "Hi! I'm Bob." }
Object: person2
Name[Diana, Cope]
Age: 28
Gender: Female
Interests: Kickboxing, Brewing
Bio{ "Diana Cope is 28 years old. She likes Kickboxing and Brewing." }
Greeting{ "Hi! I'm Diana." }

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 classesSection

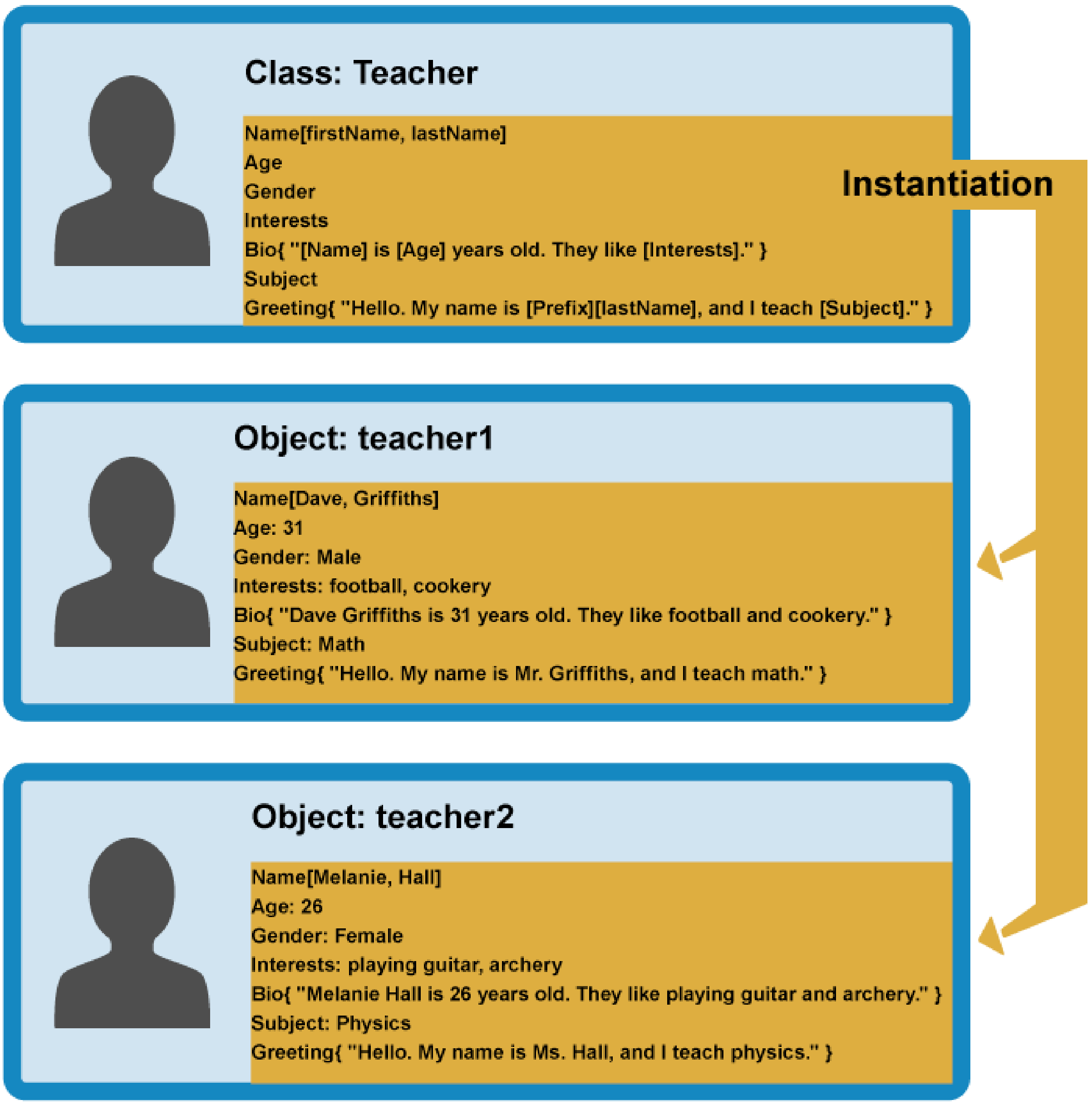
In this case we don't want generic people — we want teachers and students, which are both more specific types of people. In OOP, we can create new classes based on other classes — these new **child classes** can be made to **inherit** the data and code features of their **parent class**, so you can reuse 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.


Class: Person
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]." }
Greeting{ "Hi! I'm [Name]." }
Inherited
Class: Teacher
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]." }
Subject
Greeting{ "Hello. My name is [Prefix] [lastName], and I teach [Subject]." }
Class: Student
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]." }
Greeting{ "Yo! I'm [firstName]." }

This is really useful — teachers and students share many common features such as name, gender, and age, so it is convenient to only have to define those features once. You can also define the same feature separately in different classes, as each definition of that feature will be in a different namespace. For example, a student's greeting might be of the form "Yo, I'm [firstName]" (e.g *Yo, I'm Sam*), whereas a teacher might use something more formal, such as "Hello, my name is [Prefix] [lastName], and I teach [Subject]." (e.g *Hello, My name is Mr Griffiths, and I teach Chemistry*).

Note: The fancy word for the ability of multiple object types to implement the same functionality is **polymorphism**. Just in case you were wondering.

You can now create object instances from your child classes. For example:


Class: Teacher
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]." }
Subject
Greeting{ "Hello. My name is [Prefix][lastName], and I teach [Subject]." }
Instantiation
Object: teacher1
Name[Dave, Griffiths]
Age: 31
Gender: Male
Interests: football, cookery
Bio{ "Dave Griffiths is 31 years old. They like football and cookery." }
Subject: Math
Greeting{ "Hello. My name is Mr. Griffiths, and I teach math." }
Object: teacher2
Name[Melanie, Hall]
Age: 26
Gender: Female
Interests: playing guitar, archery
Bio{ "Melanie Hall is 26 years old. They like playing guitar and archery." }
Subject: Physics
Greeting{ "Hello. My name is Ms. Hall, and I teach physics." }

In the rest of the article, we'll start to look at how OOP theory can be put into practice in JavaScript.

Constructors and object instancesSection

JavaScript uses special functions called **constructor functions** to define and initialize 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.

Let's explore creating classes via constructors and creating object instances from them in JavaScript. First of all, we'd like you to make a new local copy of the oojs.html file we saw in our first Objects article.

A simple exampleSection

1. Let's start by looking at how you could define a person with a normal function. Add this function within the `script` element:

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

```
}
```

9. You can now create a new person by calling this function — try the following lines in your browser's JavaScript console:

```
const 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!

12. Replace your previous function with the following:

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

```
}
```

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.

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

So how do we call a constructor to create some objects?

1. Add the following lines below your previous code addition:

```
2. const person1 = new Person('Bob');
```

```
const person2 = new Person('Sarah');
```

3. Save your code and reload it in the browser, and try entering the following lines into your JS console:

```
4. person1.name
5. person1.greeting()
6. person2.name
```

```
person2.greeting()
```

Cool! You'll now see that we have two new objects on the page, each of which is stored under a different namespace — when you access their properties and methods, you have to start calls with `person1` or `person2`; the functionality contained within is neatly packaged away so it won't clash with other functionality. They do, however, have the same `name` property and `greeting()` method available. Note that they are using their own `name` value that was assigned to them when they were created; this is one reason why it is very important to use `this`, so they will use their own values, and not some other value.
Let's look at the constructor calls again:

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

In each case, the `new` keyword is used to tell the browser we want to create a new object instance, followed by the function name with its required parameters contained in parentheses, and the result is stored in a variable — very similar to how a standard function is called. Each instance is created according to this definition:

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

After the new objects have been created, the `person1` and `person2` variables contain the following objects:

```
{
  name: 'Bob',
  greeting: function() {
    alert('Hi! I\'m ' + this.name + '.');
  }
}
```

```
{
  name: 'Sarah',
  greeting: function() {
    alert('Hi! I\'m ' + this.name + '.');
  }
}
```

Note that when we are calling our constructor function, we are defining `greeting()` every time, which isn't ideal. To avoid this, we can define functions on the prototype instead, which we will look at later.

Creating our finished constructorSection

The example we looked at above was only a simple example to get us started. Let's now get on and create our final `Person()` constructor function.

1. Remove the code you inserted so far, and add in this replacement constructor — this is exactly the same as the simple example in principle, with just a bit more complexity:

```
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 + '.');
  };
}
```

16. Now add in the following line below it, to create an object instance from it:

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

You'll now see that you can access the properties and methods just like we did previously — try these in your JS console:

```
person1['age']
person1.interests[1]
person1.bio()
// etc.
```

Note: If you are having trouble getting this to work, try comparing your code against our version — see oojs-class-finished.html (also see it running live).

Further exercisesSection

To start with, try adding a couple more object creation lines of your own, and try getting and setting the members of the resulting object instances.

In addition, there are a couple of problems with our `bio()` method — the output always includes the pronoun "He", even if your person is female, or some other preferred gender classification. And the bio will only include two interests, even if more are listed in the `interests` array. Can you work out how to fix this in the class definition (constructor)? You can put any code you like inside a constructor (you'll probably need a few conditionals and a loop). Think about how the sentences should be structured differently depending on gender, and depending on whether the number of listed interests is 1, 2, or more than 2.

Note: If you get stuck, we have provided an answer inside our GitHub repo (see it live) — try writing it yourself first though!

Other ways to create object instancesSection

So far we've seen two different ways to create an object instance — declaring an object literal, and using a constructor function (see above).
These make sense, but there are other ways — we want to make you familiar with these in case you come across them in your travels around the Web.

The Object() constructorSection

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.

1. Try entering this into your browser's JavaScript console:

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

2. This stores an empty object in the `person1` variable. You can then add properties and methods to this object using dot or bracket notation as desired; try these examples in your console:

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

```
};
```

7. You can also pass an object literal to the `Object()` constructor as a parameter, to prefill it with properties/methods. Try this in your JS console:

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

Using the create() methodSection

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. With your finished exercise from the previous sections loaded in the browser, try this in your JavaScript console:

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

2. Now try these:

```
3. person2.name
```

```
person2.greeting()
```

You'll see that `person2` has been created based on `person1`—it has the same properties and method available to it.
One limitation of `create()` is that IE8 does not support it. So constructors may be more effective if you want to support older browsers.
We'll explore the effects of `create()` in more detail later on.