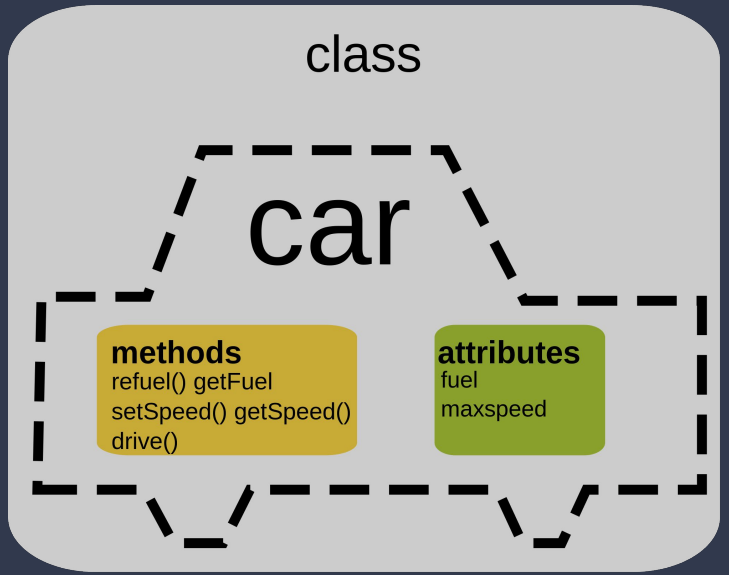
Lecture 8.2

Objects

Intro to OOP

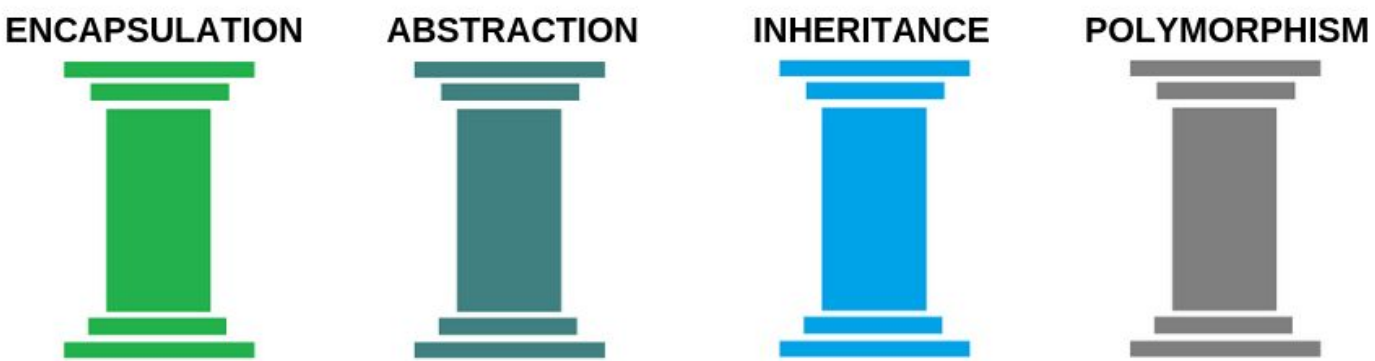
Object Oriented Programming



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.

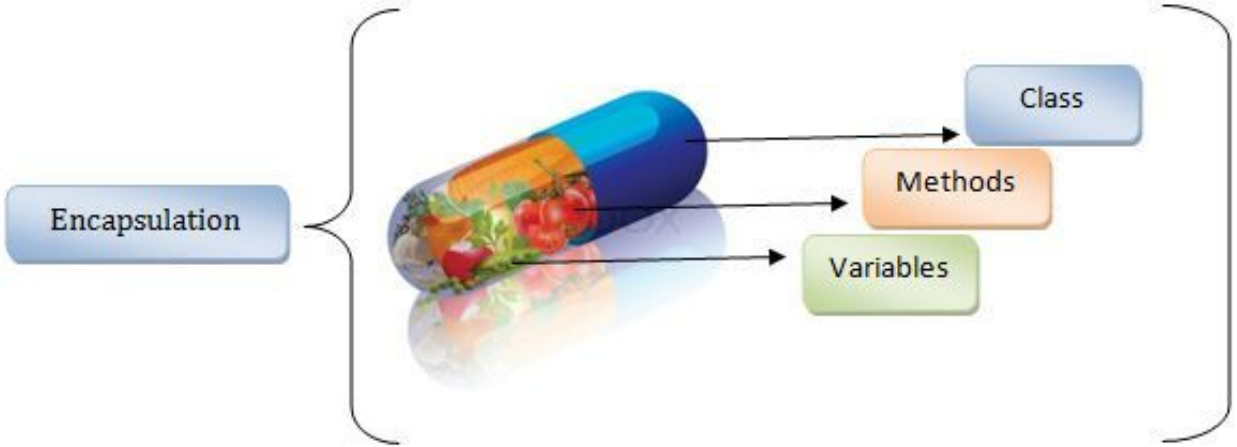
Pillars of Object Oriented Programming



Encapsulation

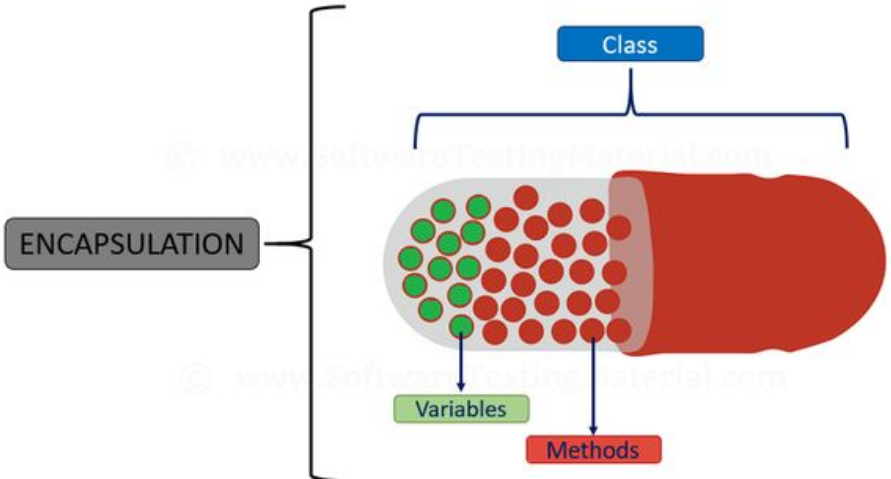
Encapsulation

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



Encapsulation

This concept is also often used to hide the internal representation, or state, of an object from the outside. This is called information hiding. The general idea of this mechanism is simple. If you have an attribute that is not visible from the outside of an object, and bundle it with methods that provide read or write access to it, then you can hide specific information and control access to the internal state of the object.



Abstraction

Defining an Object Template

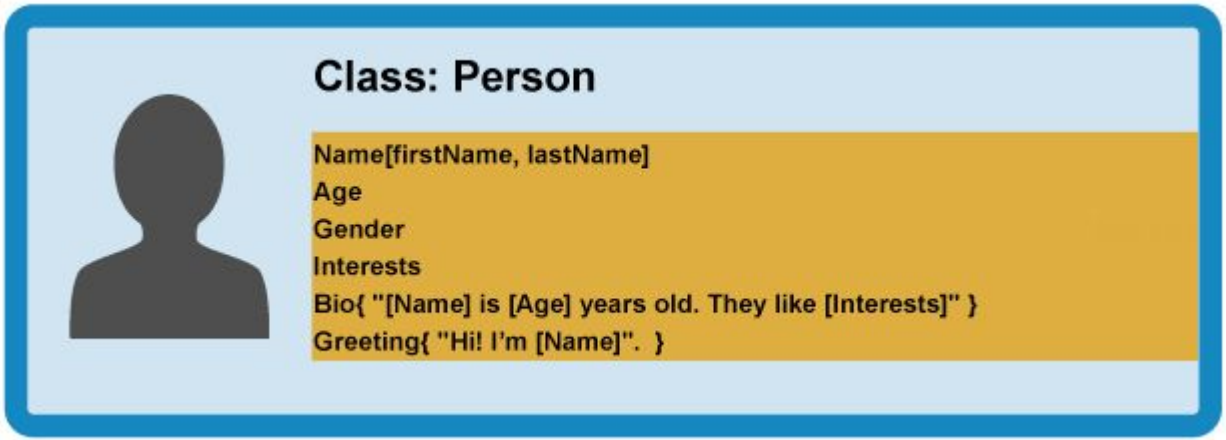
To start this off, we could return to our Person object type, which defines the generic data and functionality of a person.

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.

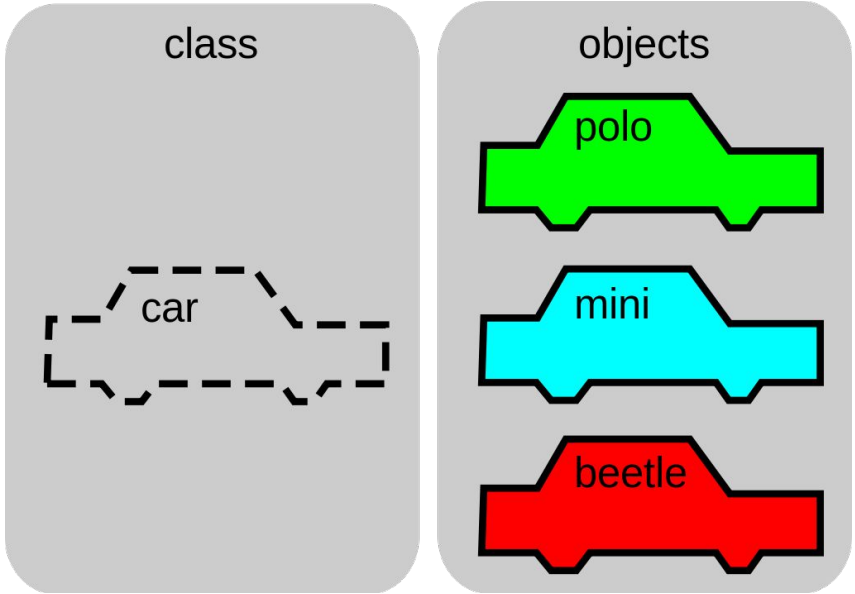
Abstraction

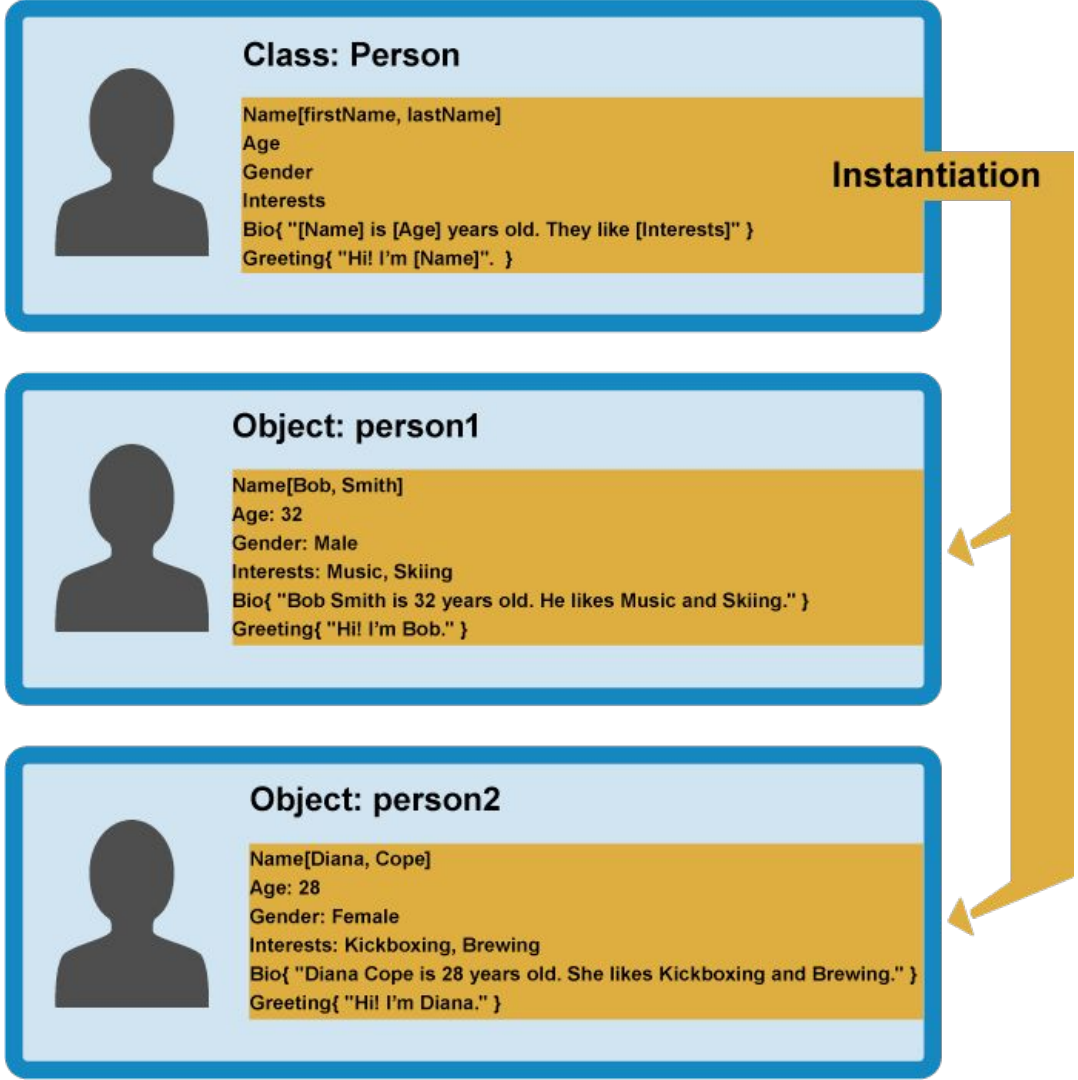
This is known as **abstraction** — creating a simple model of a more complex thing, which represents its most important aspects in a way that is easy to work with for our program's purposes.



Creating actual objects

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




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." }

Instantiation

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.

Video - OOP

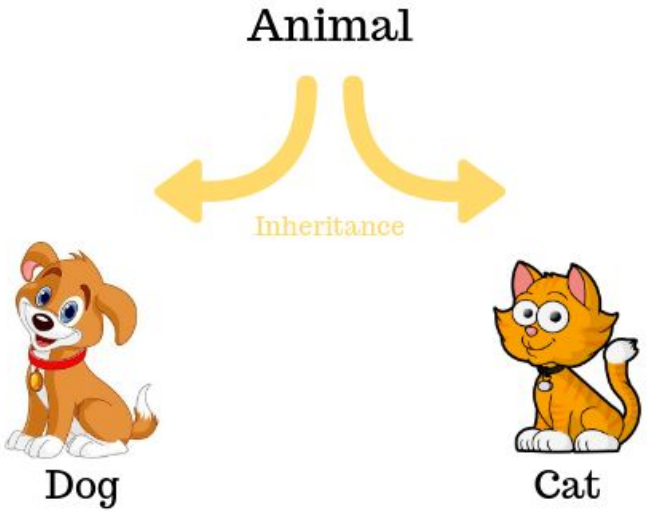
Inheritance

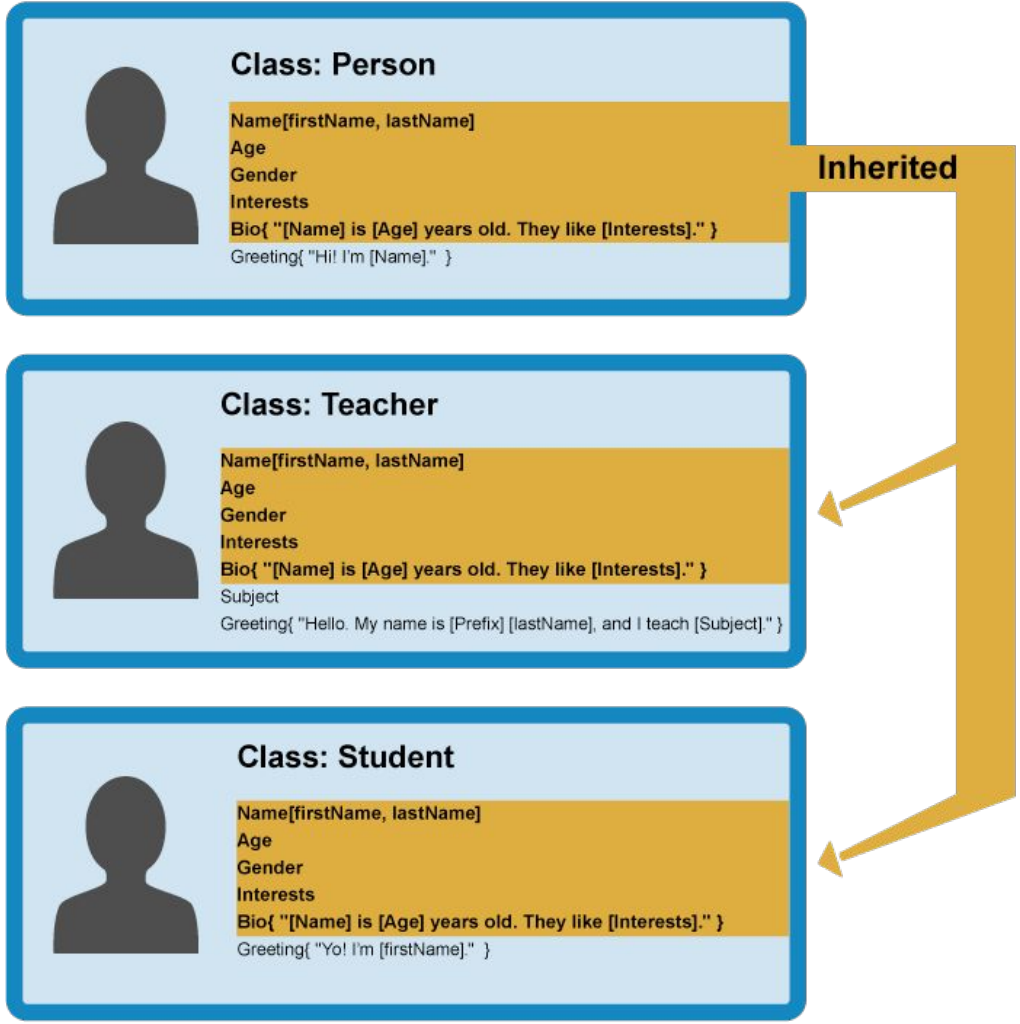
Inheritance - Specialist Classes

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.

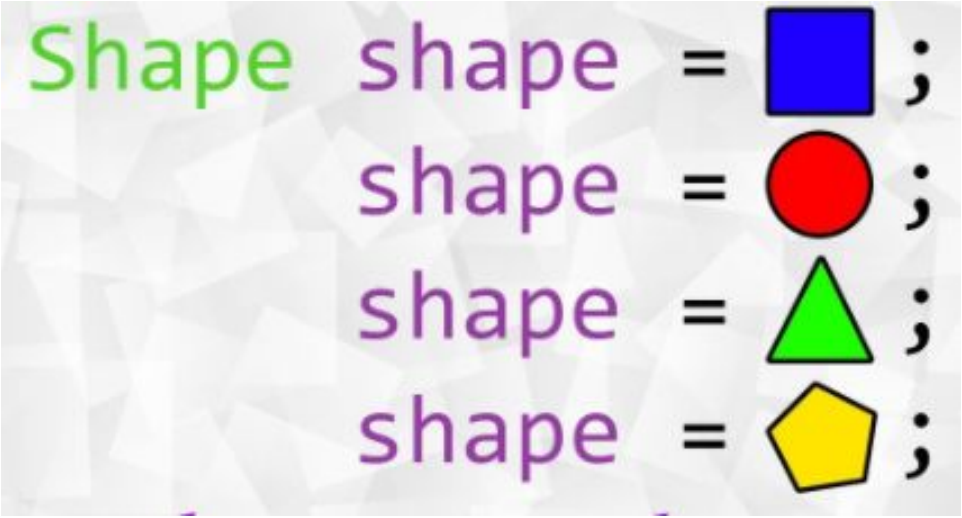


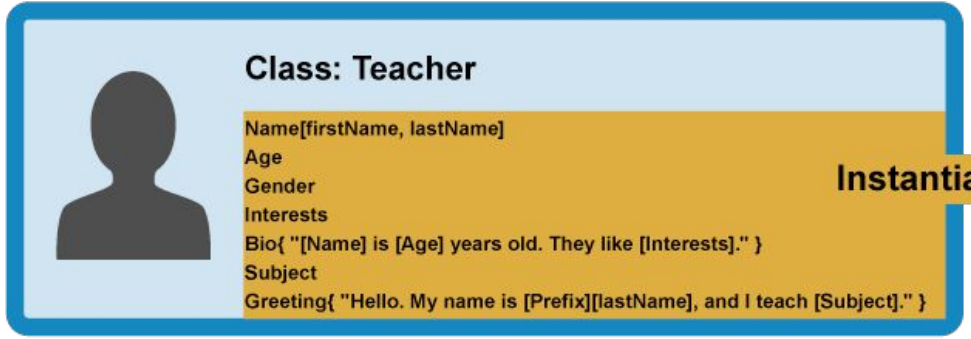

Class: Person
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]." }
Greeting{ "Hi! I'm [Name]." }
Inherited
Class: Teacher
Name[firstName, lastName]
Age
Gender
Interests
Bio{ "[Name] is [Age] years old. They like [Interests]." }
Subject
Greeting{ "Hello.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

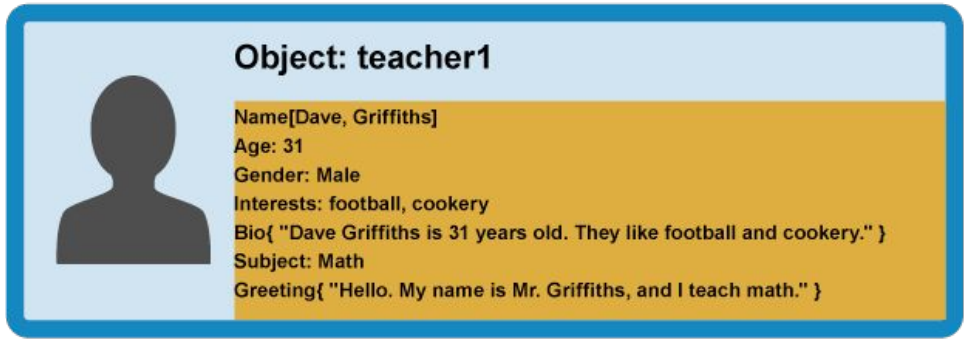
Polymorphism

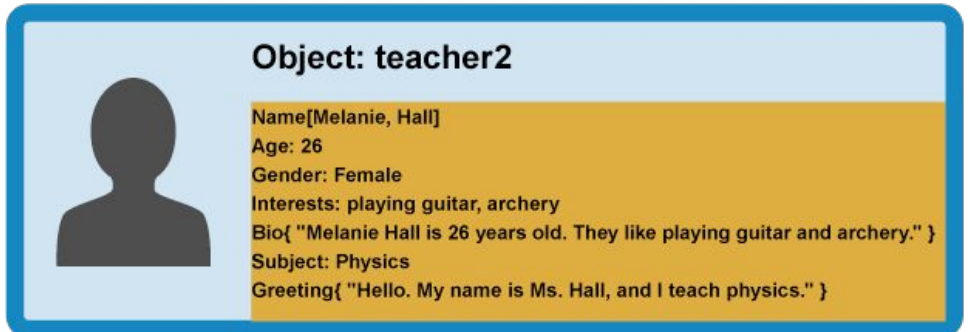
Polymorphism

The word polymorphism is used in various contexts and describes situations in which something occurs in several different forms. In computer science, it describes the concept that objects of different types can be accessed through the same interface.




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]." }
Instantiat


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." }

Constructor Functions

Constructors and object instances

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.

Factory Pattern

The factory pattern uses a function to abstract away the process of creating specific objects.

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

const mike = createNewPerson('Mike');
mike.greeting();
```

Constructor Pattern

JavaScript allows you to create a custom constructor function that defines the properties and methods of user-defined objects.

By convention, the name of a constructor function in JavaScript starts with an uppercase letter.

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

Constructor Pattern cont..

The constructor function is JavaScript's version of a class. 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.

Notice also 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.

```
var mike = new Person('Mike');
mike.greeting();
```

Additional ways to Create Objects

Constructor - Video

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.

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:

```
let person1 = new Object();

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

The Object.create() method

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, using an existing object as the prototype of the newly created object.

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

person2.name;
person2.greeting();
```

Object Built-In Methods

The Object.assign() method

The **Object.assign()** method copies all enumerable own properties from one or more *source objects* to a *target object*. It returns the target object.

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);
// expected output: Object { a: 1, b: 4, c: 5 }

console.log(returnedTarget);
// expected output: Object { a: 1, b: 4, c: 5 }
```

The Object.keys() method

The **Object.keys()** method returns an array of a given object's own enumerable property **names**, iterated in the same order that a normal loop would.

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false
};

console.log(Object.keys(object1));
// expected output: Array ["a", "b", "c"]
```

The Object.values() method

The **Object.values()** method returns an array of a given object's own enumerable property values, in the same order as that provided by a for...in loop.

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false
};

console.log(Object.values(object1));
// expected output: Array ["somestring", 42, false]
```

The Object.entries() method

The **Object.entries()** method returns an array of a given object's own enumerable string-keyed property [key, value] pairs, in the same order as that provided by a for...in loop.

```
const object1 = {
  a: 'somestring',
  b: 42
};

for (const [key, value] of Object.entries(object1)) {
  console.log(`${key}: ${value}`);
}

// expected output:
// "a: somestring"
// "b: 42"
// order is not guaranteed
```