**Object-oriented programming (OOP)** is a fundamental programming paradigm used by nearly every developer at some point in their career. OOP is the **most popular programming paradigm** used for software development and is taught as the standard way to code for most of a programmer’s educational career. Another popular programming paradigm is functional programming, but we won’t get into that right now.

Today we will break down the basics of what makes a program object-oriented so that you can start to utilize this paradigm in your algorithms, projects, and interviews.

**What is Object-Oriented Programming?**

Object-Oriented Programming (OOP) is a programming paradigm in computer science that relies on the concept of **classes** and **objects**. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects. There are many object-oriented programming languages, including JavaScript, [C++](https://www.educative.io/blog/how-to-learn-cpp-the-guide-for-beginners), [Java](https://www.educative.io/blog/object-oriented-programming-concepts-java), and [Python](https://www.educative.io/blog/how-to-use-oop-in-python).

OOP languages are not necessarily restricted to the object-oriented programming paradigm. Some languages, such as JavaScript, Python, and PHP, all allow for both procedural and object-oriented programming styles.

A **class** is an abstract blueprint that creates more specific, concrete objects. Classes often represent broad categories, like Car or Dog that share **attributes**. These classes define what attributes an instance of this type will have, like color, but not the value of those attributes for a specific object.

Classes can also contain functions called **methods** that are available only to objects of that type. These functions are defined within the class and perform some action helpful to that specific object type.

For example, our Car class may have a repaint method that changes the color attribute of our car. This function is only helpful to objects of type Car, so we declare it within the Car class, thus making it a method.

Class templates are used as a blueprint to create individual **objects**. These represent specific examples of the abstract class, like myCar or goldenRetriever. Each object can have unique values to the properties defined in the class.

For example, say we created a class, Car, to contain all the properties a car must have, color, brand, and model. We then create an instance of a Car type object, myCar to represent my specific car.

We could then set the value of the properties defined in the class to describe my car without affecting other objects or the class template.

We can then reuse this class to represent any number of cars.

Class blueprint being used to create two Car type objects, myCar and helensCar

**Benefits of OOP for software engineering**

* OOP models complex things as reproducible, simple structures
* Reusable, OOP objects can be used across programs
* Polymorphism allows for class-specific behavior
* Easier to debug, classes often contain all applicable information to them
* Securely protects sensitive information through encapsulation

**How to structure OOP programs**

Let’s take a real-world problem and conceptually design an OOP software program.

Imagine running a dog-sitting camp with hundreds of pets where you keep track of the names, ages, and days attended for each pet.

How would you design simple, reusable software to model the dogs?

With hundreds of dogs, it would be inefficient to write unique entries for each dog because you would be writing a lot of redundant code. Below we see what that might look like with objects rufus and fluffy.

1

//Object of one individual dog

var rufus = {

    name: "Rufus",

    birthday: "2/1/2017",

    age: function() {

        return Date.now() - this.birthday;

    },

    attendance: 0

}

//Object of second individual dog

var fluffy = {

    name: "Fluffy",

    birthday: "1/12/2019",

    age: function() {

        return Date.now() - this.birthday;

    },

    attendance: 0

}



ou can see above, there is a lot of duplicated code between both objects. The age() function appears in each object. Since we want the same information for each dog, we can use objects and classes instead.

**Grouping related information together** to form a class structure makes the code shorter and easier to maintain.

In the dogsitting example, here’s how a programmer could think about organizing an OOP:

1. **Create a class for all dogs** as a blueprint of information and behaviors (methods) that all dogs will have, regardless of type. This is also known as the **parent class**.
2. **Create subclasses** to represent different subcategories of dogs under the main blueprint. These are also referred to as *child classes*.
3. **Add unique attributes and behaviors** to the child classes to represent differences
4. **Create objects from the child class** that represent dogs within that subgroup

The diagram below represents how to design an OOP program by grouping the related data and behaviors together to form a simple template and then creating subgroups for specialized data and behavior.

The Dog class is a generic template containing only the structure of data and behaviors common to all dogs as attributes.

We then create two child classes of Dog, HerdingDog and TrackingDog. These have the inherited behaviors of Dog (bark()) but also behavior unique to dogs of that subtype.

Finally, we create objects of the HerdingDog type to represent the individual dogs Fluffy and Maisel.

We can also create objects like Rufus that fit under the broad class of Dog but do not fit under either HerdingDog or TrackingDog.

**Building blocks of OOP**

Next, we’ll take a deeper look at each of the fundamental building blocks of an OOP program used above:

* Classes
* Objects
* Methods
* Attributes

**Classes**

In a nutshell, classes are essentially **user-defined data types**. Classes are where we create a blueprint for the structure of methods and attributes. Individual objects are instantiated from this blueprint.

Classes contain fields for attributes and methods for behaviors. In our Dog class example, attributes include name & birthday, while methods include bark() and updateAttendance().

Here’s a code snippet demonstrating how to program a Dog class using the [JavaScript](https://www.educative.io/blog/javascript-oop-primer) language.

    constructor(name, birthday) {

        this.name = name;

        this.birthday = birthday;

    }

    //Declare private variables

    \_attendance = 0;

    getAge() {

        //Getter

        return this.calcAge();

    }

    calcAge() {

        //calculate age using today's date and birthday

        return Date.now() - this.birthday;

    }

    bark() {

        return console.log("Woof!");

    }

    updateAttendance() {

        //add a day to the dog's attendance days at the petsitters

        this.\_attendance++;

    }

}





Remember, the class is a template for modeling a dog, and an object is instantiated from the class representing an individual real-world item.

***Enjoying the article?****Scroll down to*[*sign up*](https://www.educative.io/blog/blog-newsletter-annoucement)*for our free, bi-monthly newsletter.*

**Objects**

Objects are, unsurprisingly, a huge part of OOP! Objects are **instances of a class** created with specific data. For example, in the code snippet below, Rufus is an instance of the Dog class.

1

class Dog {

    constructor(name, birthday) {

        this.name = name;

        this.birthday = birthday;

    }

    //Declare private variables

    \_attendance = 0;

    getAge() {

        //Getter

        return this.calcAge();

    }

    calcAge() {

        //calculate age using today's date and birthday

        return Date.now() - this.birthday;

    }

    bark() {

        return console.log("Woof!");

    }

    updateAttendance() {

        //add a day to the dog's attendance days at the petsitters

        this.\_attendance++;

    }

}

//instantiate a new object of the Dog class, and individual dog named Rufus

const rufus = new Dog("Rufus", "2/1/2017");





When the new class Dog is called:

* A new object is created named rufus
* The constructor runs name & birthday arguments, and assigns values

**Programming vocabulary:**

In JavaScript, objects are a type of variable. This may cause confusion because objects can be declared without a class template in JavaScript, as shown at the beginning.

Objects have states and behaviors. The state of an object is defined by data: things like names, birthdates, and other information you’d want to store about a dog. Behaviors are methods the object can undertake.

| N/A | What is it? | Information Contained | Actions | Example |
| --- | --- | --- | --- | --- |
| Classes | Blueprint | Attributes | Behaviors defined through methods | Dog Template |
| Objects | Instance | State, Data | Methods | Rufus, Fluffy |

**Attributes**

Attributes are the information that is stored. Attributes are defined in the Class template. When objects are instantiated, individual objects contain data stored in the Attributes field.

The state of an object is defined by the data in the object’s attributes fields. For example, a puppy and a dog might be treated differently at a pet camp. The birthday could define the state of an object and allow the software to handle dogs of different ages differently.

**Methods**

Methods represent behaviors. Methods perform actions; methods might return information about an object or update an object’s data. The method’s code is defined in the class definition.

When individual objects are instantiated, these objects can call the methods defined in the class. In the code snippet below, the bark method is defined in the Dog class, and the bark() method is called on the Rufus object.

    //Declare protected (private) fields

    \_attendance = 0;

    constructor(name, birthday) {

        this.namee = name;

        this.birthday = birthday;

    }

    getAge() {

        //Getter

        return this.calcAge();

    }

    calcAge() {

        //calculate age using today's date and birthday

        return this.calcAge();

    }

    bark() {

        return console.log("Woof!");

    }

    updateAttendance() {

        //add a day to the dog's attendance days at the petsitters

        this.\_attendance++;

    }

}





Methods often modify, update or delete data. Methods don’t have to update data though. For example, the bark() method doesn’t update any data because barking doesn’t modify any of the attributes of the Dog class: name or birthday.

The updateAttendance() method adds a day the Dog attended the pet-sitting camp. The attendance attribute is important to keep track of for billing Owners at the end of the month.

Methods are how programmers promote reusability and keep functionality encapsulated inside an object. This reusability is a great benefit when debugging. If there’s an error, there’s only one place to find it and fix it instead of many.

The underscore in \_attendance denotes that the variable is protected and shouldn’t be modified directly. The updateAttendance() method changes \_attendance.

**Four Principles of OOP**

The four pillars of object-oriented programming are:

* **Inheritance:** child classes inherit data and behaviors from the parent class
* **Encapsulation:** containing information in an object, exposing only selected information
* **Abstraction:** only exposing high-level public methods for accessing an object
* **Polymorphism:** many methods can do the same task

**Inheritance**

Inheritance allows classes to inherit features of other classes. Put another way, parent classes extend attributes and behaviors to child classes. **Inheritance supports reusability**.

If basic attributes and behaviors are defined in a parent class, child classes can be created, extending the functionality of the parent class and adding additional attributes and behaviors.

For example, herding dogs have the unique ability to herd animals. In other words, all herding dogs are dogs, but not all dogs are herding dogs. We represent this difference by creating a child class HerdingDog from the parent class Dog, and then adding the unique herd() behavior.

The benefits of inheritance are programs can create a generic parent class and then create more specific child classes as needed. This simplifies programming because instead of recreating the structure of the Dog class multiple times, **child classes automatically gain access to functionalities within their parent class.**

In the following code snippet, child class HerdingDog inherits the method bark from the parent class Dog, and the child class adds an additional method, herd().

Parent class Dog

class Dog{

    //Declare protected (private) fields

    \_attendance = 0;

    constructor(namee, birthday) {

        this.name = name;

        this.birthday = birthday;

    }

    getAge() {

        //Getter

        return this.calcAge();

    }

    calcAge() {

        //calculate age using today's date and birthday

        return this.calcAge();

    }

    bark() {

        return console.log("Woof!");

    }

    updateAttendance() {

        //add a day to the dog's attendance days at the petsitters

        this.\_attendance++;

    }

}

//Child class HerdingDog, inherits from parent Dog





Notice that the HerdingDog class does not have a copy of the bark() method. It inherits the bark() method defined in the parent Dog class.

When the code calls fluffy.bark() method, the bark() method walks up the chain of child to parent classes to find where the bark method is defined.

class Dog{

    //Declare protected (private) fields

    \_attendance = 0;

    constructor(namee, birthday) {

        this.name = name;

        this.birthday = birthday;

    }

    getAge() {

        //Getter

        return this.calcAge();

    }

    calcAge() {

        //calculate age using today's date and birthday

        return this.calcAge();

    }

    bark() {

        return console.log("Woof!");

    }

    updateAttendance() {

        //add a day to the dog's attendance days at the petsitters

        this.\_attendance++;

    }

}

//Child class HerdingDog, inherits from parent Dog





**Note:** Parent classes are also known as superclasses or base classes. The child class can also be called a subclass, derived class, or extended class.

In JavaScript, inheritance is also known as [**prototyping**](https://www.educative.io/blog/understanding-and-using-prototypal-inheritance-in-javascript). A prototype object is a template for another object to inherit properties and behaviors. There can be multiple prototype object templates, creating a prototype chain.

This is the same concept as the parent/child inheritance. Inheritance is from parent to child. In our example, all three dogs can bark, but only Maisel and Fluffy can herd.

The herd() method is defined in the child HerdingDog class, so the two objects, Maisel and Fluffy, instantiated from the HerdingDog class have access to the herd() method.

Rufus is an object instantiated from the parent class Dog, so Rufus only has access to the bark() method.

| Object | Instantiated from Class | Parent Class | Methods |
| --- | --- | --- | --- |
| Rufus | Dog | N/A | bark() |
| Maisel | Herding Dog | Dog | bark(), herd() |
| Fluffy | Herding Dog | Dog | bark(), herd() |

**Encapsulation**

Encapsulation means containing all important information **inside an object**, and only exposing selected information to the outside world. Attributes and behaviors are defined by code inside the class template.

Then, when an object is instantiated from the class, the data and methods are encapsulated in that object. Encapsulation hides the internal software code implementation inside a class and hides the internal data of inside objects.

Encapsulation requires defining some fields as private and some as public.

* **Private/ Internal interface:** methods and properties accessible from other methods of the same class.
* **Public / External Interface:** methods and properties accessible from outside the class.

Let’s use a car as a metaphor for encapsulation. The information the car shares with the outside world, using blinkers to indicate turns, are public interfaces. In contrast, the engine is hidden under the hood.

It’s a private, internal interface. When you’re driving a car down the road, other drivers require information to make decisions, like whether you’re turning left or right. However, exposing internal, private data like the engine temperature would confuse other drivers.

Encapsulation adds **security**. Attributes and methods can be set to private, so they can’t be accessed outside the class. To get information about data in an object, public methods & properties are used to access or update data.

This adds a layer of security where the developer chooses what data can be seen on an object by exposing that data through public methods in the class definition.

Within classes, most programming languages have public, protected, and private sections. The public section is the limited selection of methods accessible from the outside world or other classes within the program. Protected is only accessible to child classes.

Private code can only be accessed from within that class. To go back to our dog/owner example, encapsulation is ideal so owners can’t access private information about other people’s dogs.

**Note:** JavaScript has private and protected properties and methods. Protected Fields are prefixed with a \_; private fields are prefixed with a #. Protected fields are inherited. Private ones aren’t.

1

2

3

class Dog{

    //Declare protected (private) fields

    \_attendance = 0;

    constructor(namee, birthday) {

        this.name = name;

        this.birthday = birthday;

    }

    getAge() {

        //Getter

        return this.calcAge();

    }

    calcAge() {

        //calculate age using today's date and birthday

        return this.calcAge();

    }

    bark() {

        return console.log("Woof!");

    }

    updateAttendance() {

        //add a day to the dog's attendance days at the petsitters

        this.\_attendance++;

    }

}

//instantiate a new instance of Dog class, an individual dog named Rufus





Consider the getAge() method in our example code, the calculation details are hidden inside the Dog class. The rufus object uses the getAge() method to calculate Rufus’s age.

**Encapsulating & updating data:** Since methods can also update an object’s data, the developer controls what values can be changed through public methods.

This allows us to **hide important information** that should not be changed from phishing and the more likely scenario of other developers mistakenly changing important data.

Encapsulation adds security to code and makes it easier to collaborate with external developers. When you’re programming to share information with an external company, you wouldn’t want to expose the classes’ templates or private data because your company owns that intellectual property.

Instead, developers create public methods that allow other developers to call methods on an object. Ideally, these public methods come with documentation for external developers.

The benefits of encapsulation are summarized here:

* **Adds security:** Only public methods and attributes are accessible from the outside
* **Protects against common mistakes:** Only public fields & methods are accessible, so developers don’t accidentally change something dangerous
* **Protects IP:** Code is hidden in a class; only public methods are accessible by the outside developers
* **Supportable:** Most code undergoes updates and improvements
* **Hides complexity:** No one can see what’s behind the object’s curtain!

**Abstraction**

Abstraction is an extension of encapsulation that uses classes and objects, which contain data and code, to hide the internal details of a program from its users. This is done by creating a **layer of abstraction** between the user and the more complex source code, which helps protect sensitive information stored within the source code.

Abstraction

* Reduces complexity and improves code readability
* Facilitates code reuse and organization
* Data hiding improves data security by hiding sensitive details from users
* Enhances productivity by abstracting away low-level details

Abstraction can also be explained using cars. Think of how a driver operates a vehicle using only the car’s dashboard.

A driver uses the car’s steering wheel, accelerator, and brake pedals to control the vehicle. The driver does not have to worry about how the engine works or what parts are used for each movement. This is an abstraction – only the important aspects necessary for a driver to use the car are visible.

Similarly, data abstraction allows developers to work with complex information without worrying about its inner workings. In this way, it helps to improve code quality and readability.

Abstraction also serves an important security role. By only displaying selected pieces of data and only allowing data to be **accessed through** classes and **modified through methods**, we protect the data from exposure. To continue with the car example, you wouldn’t want an open gas tank while driving a car.

The benefits of abstraction are summarized below:

* Simple, high-level user interfaces
* Complex code is hidden
* Security
* Easier software maintenance
* Code updates rarely change the abstraction

**Polymorphism**

Polymorphism means designing objects to **share behaviors**. Using inheritance, objects can override shared parent behaviors with specific child behaviors. Polymorphism allows the same method to execute different behaviors in two ways: method overriding and method overloading.

**Method Overriding**

Runtime polymorphism uses method overriding. In method overriding, a child class can implement differently than its parent class. In our dog example, we may want to give TrackingDog a specific type of bark different than the generic dog class.

Method overriding could create a bark() method in the child class that overrides the bark() method in the parent Dog class.

class Dog{

    //Declare protected (private) fields

    \_attendance = 0;

    constructor(namee, birthday) {

        this.name = name;

        this.birthday = birthday;

    }

    getAge() {

        //Getter

        return this.calcAge();

    }

    calcAge() {

        //calculate age using today's date and birthday

        return this.calcAge();

    }

    bark() {

        return console.log("Woof!");

    }

    updateAttendance() {

        //add a day to the dog's attendance days at the petsitters

        this.\_attendance++;

    }

}

//Child class TrackingDog, inherits from parent



TrackingDog's overriding the bark() method

**Method Overloading**

**Compile Time polymorphism** uses method overloading. Methods or functions may have the same name but a different number of parameters passed into the method call. Different results may occur depending on the number of parameters passed in.

//Parent class Dog

class Dog{

    //Declare protected (private) fields

    \_attendance = 0;

    constructor(namee, birthday) {

        this.name = name;

        this.birthday = birthday;

    }

    getAge() {

        //Getter

        return this.calcAge();

    }

    calcAge() {

        //calculate age using today's date and birthday

        return this.calcAge();

    }

    bark() {

        return console.log("Woof!");

    }

    updateAttendance() {

        //add a day to the dog's attendance days at the petsitters

        this.\_attendance++;

    }

    updateAttendance(x) {

        //adds multiple to the dog's attendance days at the petsitters





In this code example, if no parameters are passed into the updateAttendance() method. One day is added to the count. If a parameter is passed in updateAttendance(4), then 4 is passed into the x parameter in updateAttendance(x), and 4 days are added to the count.

The benefits of Polymorphism are:

* Objects of different types can be passed through the same interface
* Method overriding
* Method overloading

**Conclusion**

Object-oriented programming requires thinking about the structure of the program and planning out an object-oriented design before you start coding. OOP in computer programming focuses on how to break up the requirements into simple, reusable classes that can be used to blueprint instances of objects. Overall, implementing OOP allows for better data structures and reusability, saving time in the long run.