**Beginner's Guide to Object-Oriented Programming (OOP)**

Unlocking OOP: Essential Concepts With Real-World Examples.

**Object-Oriented Programming (OOP)** is a game-changing and **popular programming paradigm** that streamlines software design and development by centering on objects, their unique qualities and actions. Almost every developer deal with OOP at some point in their career so it’s quite essential to master OOP concepts.

Whether you are an experienced programmer or a young developer learning programming, this beginner-friendly tutorial will take you through the basics of OOP with easy-to-understand examples so that you can utilize this modern programming paradigm in your projects and jobs interviews.

Let’s get started and discover the world of object-oriented programming together!

Topics covered in this this tutorial:

* What is Object-Oriented Programming?
* Building Blocks of OOP
* Four Pillars of OOP
* Principles of OOP
* How to advance in OOP

**What is Object-Oriented Programming?**

Object-Oriented Programming (OOP) is a widely-used programming approach in computer science, which is centered around the concepts of classes and objects. It enables software developers to create well-organized, reusable code blueprints (commonly referred to as classes) that can be used to build individual instances of objects. There are numerous object-oriented programming languages, including Java, C++, Python, and JavaScript.

A **class** serves as a high-level blueprint for generating more specific, concrete objects. Classes typically represent general categories, like Bicycle or Book, which have common **attributes**. These classes define the attributes that an object of this type will possess, such as color, brand, but not the specific values for those attributes in a given object.

Classes can also include functions known as methods, which are exclusive to objects of that particular type. These functions are defined inside the class and execute actions that are useful for the specific object type.

For example, our Bicycle class might have a method called changeGear() that adjusts the gear setting on a bicycle. This function is only useful for objects of the Bicycle type, so it is defined within the Bicycle class, making it a method.

Class blueprints are used to generate individual objects, which represents specific instances of the abstract class, like bianchiCycle or orbeaCycle. Each object can hold unique values for the properties outlined in the class.

Suppose we create a class called Bicycle that contains all the essential properties a bicycle must have, such as color, brand, and gearCount. We then build an instance of a Bicycle object, bianchi, to symbolize my specific bicycle.

We can then assign values to the properties defined in the class to describe my bicycle without impacting other objects or the class blueprint itself.

This class can be reused to represent any number of bicycles.

Two Bicycle objects bianchiCycle, orbeaCycle being created from Bicycle class blueprint

**History and Evolution of OOP**

The fascinating journey of Object-Oriented Programming (OOP) began in the 1960s with the groundbreaking development of the Simula programming language. Simula was a pioneer, being the first language to introduce the world to the revolutionary concept of classes and objects. This laid the crucial foundation for OOP that we know and appreciate today.

However, the true potential of OOP wasn't fully realized until the 1980s, when the Smalltalk language emerged. With its innovative design, Smalltalk made OOP more accessible and versatile, leading to its widespread adoption. Since that pivotal moment, numerous programming languages have embraced OOP principles, including Java, C++, Python, and C#.

**Procedural vs. Object-Oriented Programming**

Procedural programming, an older approach that revolves around a series of actions or procedures to manipulate data. In procedural languages, we organize code into functions or procedures, which are then called to carry out specific tasks. While this method works well for smaller projects, I found that managing complex, large-scale projects could become quite challenging with this approach.

On the other side, OOP, which offers a different perspective on organizing code. In OOP, we create classes that serve as blueprints for constructing objects. Objects are instances of these classes, and they interact with one another using methods that define their behavior. This structure emphasizes modularity, reusability, and maintainability, which I found to be incredibly helpful when working on larger, more intricate projects.

**Benefits of OOP**

Object-oriented programming offers several advantages over procedural programming, including:

**Modularity:** OOP encourages the separation of concerns, enabling developers to break down a software system into smaller, more manageable modules. This approach makes it easier for everyone involved to understand and contribute to a project.

**Reusability:** OOP allows us to create reusable components, which simplifies the process of building and maintaining software systems. This means that we can efficiently leverage existing work and focus on enhancing our projects.

**Scalability:** The modular nature of OOP makes it easier to scale projects as they grow and evolve. This adaptability is invaluable, whether you're working on a personal project or collaborating with a team on a large-scale endeavor.

**Maintainability:** OOP's encapsulation of code and data within objects simplifies understanding, modifying, and [maintaining code](https://www.designgurus.io/answers/detail/how-to-maintain-code-quality-in-coding-interviews). This fosters a cleaner codebase, saving time and effort for everyone involved.

**Extensibility:** OOP promotes easy extension of existing code, facilitating the addition of new features and functionality. This flexibility is crucial for accommodating the ever-changing requirements and goals that we all face in our projects.

**Building Blocks of Object-Oriented Programming (OOP)**

Object-oriented programming (OOP) revolves around a set of fundamental concepts that simplify software development and promote better design practices. In this tutorial, we will dive into the building blocks of OOP, including:

* Classes
* Objects
* Attributes
* Methods
* Constructors and Destructors
* Interfaces

**Classes**

In simple terms, classes are custom data types created by users. They provide a blueprint for organizing attributes and methods. Objects are created using this blueprint as a base.

Classes consist of fields (attributes) and methods (behaviors). For example, in a superhero class like Iron Man, attributes could be name and suit color, while methods might be fly() and shootLaser().

Here's a basic example of how to create an Iron Man class using JavaScript:

This code demonstrates the creation of an Iron Man class with name and suit color attributes, as well as fly() and shootLaser() methods.

**Objects**

Objects are instances of classes, representing real-world entities or abstract concepts. They are the primary components of an OOP-based software system. Objects store data in the form of attributes and have associated methods that define their behavior. For example, in the code snippet below, ironman is an instance IronMan class.

When the class IronMan’s constructor is called with name and suit color, two things happen:

1. A new object is created named ironman
2. The constructor for class IronMan runs and assigns name & suitColor arguments to their respective attributes.

In JavaScript, objects are a special kind of variable. This can be confusing because, unlike some other languages, objects can be created directly without needing a class template.

Objects have states and behaviors. States are data, such as name or suitColor, that store information about the object. Behaviors are functions, or methods, that the object can perform.

| **Aspect** | **Class** | **Object** |
| --- | --- | --- |
| Definition | A blueprint that defines attributes and methods | An instance created using a class |
| Example | IronMan class with name, suitColor, fly(), and shootLaser() methods | ironMan: a specific instance of the IronMan class with the name 'Tony Stark' and suit color 'Red and Gold' |
| Attributes | Defined as fields within the class, such as name and suitColor | Have specific values for each object, such as 'Tony Stark' for name and 'Red and Gold' for suitColor |
| Methods | Defined within the class, such as fly() and shootLaser() | Can call the methods defined in the class, such as ironMan.fly() and ironMan.shootLaser() |
| Purpose | To create a reusable structure for organizing similar data and behaviours | To represent a specific instance with its own data and behaviours based on the class blueprint |

The table shows how classes define the structure for attributes and methods, while objects are individual instances with specific values and behaviours based on the class.

**Attributes**

Attributes, also known as properties or fields, represent the data or state of an object. They store information about an object's characteristics, such as its color, size, or name. Attributes are typically defined within a class and are assigned values when an object is instantiated.

The state of an object is determined by the data in its attribute fields. For instance, an Iron Man suit with different features might be treated differently in various situations. The suit color could define the state of the object and allow the software to handle Iron Man suits with different appearances accordingly.

**Methods**

Methods are functions that belong to a class and define the behavior of its objects. They perform actions, manipulate data, and interact with other objects. Methods allow objects to communicate with one another, enabling developers to create complex systems with a high degree of organization and modularity.

When creating specific instances of objects, these objects can use the methods defined in the class. In the example below, the fly() method is defined in the IronMan class, and the fly() method is called on the ironMan object.

Methods often change, update, or remove data. However, they don't always need to modify data. For example, the fly() method doesn't update any data because flying doesn't alter any of the IronMan class's attributes: name or suitColor.

A method, like updateArmorStrength(), could be used to increase the strength of Iron Man's armor. This attribute is essential to keep track of during battles.

Methods help programmers encourage reusability and keep features contained within an object. This makes debugging easier, as errors can be found and fixed in a single location.

In the modified IronMan class above, the underscore in \_armorStrength signifies that the variable is protected and should not be changed directly. Instead, the updateArmorStrength() method is used to modify \_armorStrength.

**Constructors and Destructors**

Constructors are special methods that are called when an object is created. They are typically used to initialize an object's attributes or set up its initial state. Destructors, on the other hand, are called when an object is destroyed or goes out of scope. They are used to release resources or perform cleanup tasks.

In our IronMan’s class example, the constructor sets up the initial state of the Iron Man object with name, suit color, and armor strength. Note that JavaScript does not have destructors like some other languages. Instead, memory management and resource cleanup are handled by the garbage collector.

**Interfaces**

An interface is a contract that defines a set of methods that a class must implement. Interfaces establish a common structure and enable polymorphism, allowing objects of different classes to be treated as objects of a common type. This promotes flexibility, extensibility, and maintainability in software systems.

Although JavaScript doesn't have built-in support for interfaces, we can mimic the functionality using classes and inheritance. Inheritance is one of the pillars of OOP which extends code reusability to next level. We will discuss more about Inheritance later in this tutorial.

Now it’s time to understand the concept of Interface using Superhero class.

In this example, we create a "Superhero" class that acts as an interface. It has a fight() method that derived classes must implement. We then create IronMan and CaptainAmerica classes that extend the Superhero class and implement the fight() method. Finally, we create a startBattle() function that accepts any object that implements the Superhero "interface" and calls the fight() method.

**Four Pillars of OOP**

Object-oriented programming (OOP) has four main pillars: encapsulation, inheritance, polymorphism, and abstraction.

* **Inheritance:** allows new classes to inherit properties and methods from existing ones, promoting code reusability.
* **Encapsulation:** involves bundling data and methods within a class, keeping them organized and secure.
* **Polymorphism:** lets objects of different classes be treated as objects of a common class, enhancing flexibility in code.
* **Abstraction:** simplifies complex systems by focusing on essential features and hiding unnecessary details, making it easier to understand and maintain code.

**Inheritance**

Inheritance is a mechanism that allows a class to inherit the attributes and methods of another class. This promotes code reuse and modularity by enabling the creation of specialized subclasses that share common functionality with their parent class, while also having their unique behavior. Inheritance lets classes gain features from other classes. In simple terms, child classes get attributes and behaviors from parent classes. This makes code reusable.

By defining basic attributes and behaviors in a parent class, you can create child classes that extend the parent class's functionality and add new features.

For example, imagine Iron Man has a special suit with extra abilities. All special suits are Iron Man suits, but not all Iron Man suits have extra abilities. To represent this, create a child class called SpecialIronMan that inherits from the parent class IronMan, and add the unique extra ability.

The advantage of inheritance is that you can create a general parent class and more specific child classes as needed. This makes coding easier because child classes automatically get access to the parent class's features without duplicating code.

In the example below, the child class SpecialIronMan inherits the fly() and shootLaser() methods from the parent class IronMan and adds an additional method, useExtraAbility().

Observe that the SpecialIronMan class does not have a copy of the fly() and shootLaser() methods. It inherits these methods from the parent IronMan class.

When the code calls specialIronMan.fly() or specialIronMan.shootLaser() methods, they look up the chain of child to parent classes to find where the fly() and shootLaser() methods are defined. Note: Keep in mind that parent classes can also be called superclasses or base classes. Similarly, child classes may be referred to as subclasses, derived classes, or extended classes.

In JavaScript, inheritance is often called prototyping. A prototype object serves as a template for other objects to inherit properties and behaviors from. Multiple prototype objects can form a prototype chain.

This idea is similar to parent/child inheritance. Inheritance passes down from parent to child. In the Iron Man example, all Iron Man suits can fly and shoot lasers, but only the special suits have an extra ability.

The useExtraAbility() method is defined in the child SpecialIronMan class, so objects created from the SpecialIronMan class, like a suit with invisibility, have access to the useExtraAbility() method.

However, a suit created from the parent class IronMan only has access to the fly() and shootLaser() methods.

**Encapsulation**

Encapsulation is the process of hiding an object's internal state and exposing a well-defined interface for interacting with it. This principle promotes information hiding, reducing the complexity of software systems and protecting the integrity of an object's data. Encapsulation means keeping important information within an object and only revealing certain information to the outside world. Attributes and behaviors are defined by code inside the class template.

When an object is created from the class, the data and methods are contained within that object. Encapsulation hides the internal code implementation and data within a class and its objects.

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

* Private/Internal Visibility: methods and properties accessible only from other methods within the same class.
* Public/External Visibility: methods and properties accessible from outside the class. Imagine encapsulation using a house metaphor. Consider a house with different areas that have different levels of privacy. The front door, exterior design, entrance and living room are public areas where guests are welcome, while the bedrooms and store rooms are private areas where only family members should go.

Just like in a house, it's important to have public interfaces to share information that others need, like the address or the house's exterior appearance. But it's also important to keep private information hidden, like the family's daily routines or the location of valuable belongings. If we shared that private information, it might make guests uncomfortable or put the family's safety at risk.

Encapsulation increases security. Attributes and methods can be set to private, preventing access from outside the class. Public methods and properties are used to access or update data, allowing the developer to control what information is visible.

Most programming languages have public, protected, and private sections within classes. The public section contains methods accessible from outside the class or other classes in the program. Protected methods are only accessible to child classes, while private methods can only be accessed within that class.

Note: Keep in mind that JavaScript has private and protected properties and methods. Protected fields have a prefix of an underscore (\_), while private fields are prefixed with a hash symbol (#). Protected fields can be inherited, but private ones cannot.

Using the Iron Man example, encapsulation is useful so that only certain information about the suit is available to others, keeping private data secure.

Consider the getArmorStrength() method in our example code, the details of armor strength are hidden inside the IronMan class. The ironMan object uses the getArmorStrength() method to get the armor strength of Iron Man's suit. Encapsulation helps manage and protect an object's data by controlling what values can be changed through public methods. This prevents important information from being accidentally modified or exposed to security risks.

Encapsulation enhances code security and makes collaboration with external developers easier. When sharing information with another company, it's important not to expose class templates or private data since they are part of your company's intellectual property.

Developers create public methods that allow others to call methods on an object, ideally with accompanying documentation.

Here are the key benefits of encapsulation:

* **Simplifies code:** Encapsulation hides complexity, keeping the inner workings of an object concealed.
* **Increases security:** Only public methods and attributes are accessible from outside.
* **Safeguards intellectual property:** Code within a class is hidden, and only public methods are accessible to external developers.
* **Prevents mistakes:** Public fields and methods are accessible, avoiding accidental changes to crucial data.
* **Supports updates:** Encapsulation makes it easier to maintain and improve code.

**Polymorphism**

Polymorphism is the ability of objects belonging to different classes to be treated as objects of a common superclass or interface. It enables developers to write more flexible and extensible code, as a single function or method can work with multiple object types, reducing code duplication. With inheritance, objects can replace shared parent behaviors with unique child behaviors. Polymorphism lets a single method perform various actions in two ways:

* Method Overriding
* Method Overloading.

Method Overriding is also referred as Runtime Polymorphism. This allows a child class to have a different behavior than its parent class. For example, suppose we want a specialized IronMan suit to have a unique flying ability compared to the generic IronMan suit.

To do this, we can create a fly() method in the child class that overrides the fly() method in the parent IronMan class.

Here's an example using IronMan:

In this code, we define a parent class IronMan with a fly() method. We then create a child class SpecialSuit that extends IronMan and overrides the fly() method with a unique behavior.

Method Overloading is also referred as Compile-time Polymorphism. Methods can have the same name but differ in the number of parameters passed to them. The outcome can vary based on the number of parameters provided during the method call.

In this example, we have an IronMan class with an attack() method that can accept different types of arguments. If the method is called without any argument, it will log a message asking the user to specify a target. If the method is called with a string argument, it will log a message indicating that Iron Man is attacking the target with his current weapon. If the method is called with an object argument that has a name property, it will log a message indicating that Iron Man is attacking the object with his current weapon.

We also have an equipWeapon() method that allows us to change the current weapon of Iron Man.

Finally, we create an instance of the IronMan class and call the attack() method with different types of arguments to demonstrate method overloading.

Here are some benefits of Polymorphism:

* **Flexibility:** Polymorphism allows objects to take on different forms and behaviors, making them more flexible and adaptable to different situations. **Code Reusability:** Polymorphism helps reduce redundancy in code by allowing developers to reuse existing code and functionality.
* **Extensibility:** Polymorphism allows for easy extension of existing code, enabling developers to add new functionality to their programs without having to start from scratch.
* **Readability:** By promoting code reuse, polymorphism can make code more readable and easier to understand.
* **Maintainability:** Polymorphism can make code easier to maintain over time, as changes or updates to one object's behavior can be made without affecting the behavior of other objects in the program.

**Abstraction**

Abstraction is the process of simplifying complex systems by breaking them down into smaller, more manageable components. In OOP, abstraction is achieved by creating classes and objects that represent high-level concepts, hiding implementation details, and focusing on essential characteristics. Abstraction builds on encapsulation by using classes and objects to hide a program's internal details from its users. It creates a layer of simplicity between the user and the complex source code, helping protect sensitive information. Abstraction is all about

* Simplifying code and improving readability
* Enhancing data security by hiding sensitive information from users
* Boosting productivity by focusing on high-level details
* Encouraging code reuse and organization

Abstraction is like a laptop's user interface. When you use a laptop, you interact with the interface (e.g., the keyboard, trackpad, and screen) without needing to know about the underlying hardware (e.g., the Processor, RAM, and Hard Disk).

For example, when you click on an icon to open an application, you don't need to know how the laptop's processor retrieves and executes the application's code. You simply interact with the interface at a high level of abstraction, without needing to worry about the low-level details.

Abstraction also plays a crucial role in security. By showing only certain data and allowing access and modification through classes and methods, data is protected from exposure. Continuing with the house metaphor, you wouldn't want to expose electrical wires or plumbing to the residents.

Here are the key benefits of abstraction:

* **Easy maintenance:** Simplified code allows for more straightforward updates
* **User-friendly interfaces:** Simplified, high-level interactions
* **Consistent abstraction:** Code updates rarely change the high-level interactions
* **Enhanced security:** Sensitive data is protected
* **Hidden complexity:** Complex code is concealed

**Conclusion**

Mastering the concepts of object-oriented programming is essential for any software developer looking to create efficient, maintainable, and scalable software systems. By understanding these core concepts, you will be well on your way to leveraging the full potential of OOP in your projects.

**How to Advance in OOP?**

Mastering OOP isn't always an easy task. It takes time, dedication, and practice to truly understand these concepts and apply them effectively. That's why resources like DesignGurus.io are so valuable. With their easy-to-follow tutorials, real-world examples, and in-browser coding exercises, you can continue to build your skills and take your coding to the next level.

So, keep up the good work and never stop learning. With the right tools and resources at your disposal, you can become a master of object-oriented programming and build amazing software that solves real-world problems.