EXPLAINER · PROGRAMMING CONCEPTS

Everything Is an Object.

Object-oriented programming is how most of the software you use every day was built. Here is the idea behind it, and why it took over the industry.

By Avalynn Circe

Before you write a single line of code, you have to make a decision that will shape everything that follows: how are you going to organize this thing?

That question sounds abstract, but it is intensely practical. A program of any real size, such as a banking app, a hospital records system, or a social network, is made up of thousands of moving parts. Data flows in, gets transformed, and flows out. Rules get applied. Decisions get made. States change. Without an organizing principle, all of that becomes an unmanageable tangle of logic that only its original author can navigate, and sometimes not even them.

Object-oriented programming, often shortened to OOP, is one answer to that question. It has been the dominant answer in software development since the 1990s. Java, Python, Ruby, C++, C#, Swift, and Kotlin are all built around it. If you have ever used a smartphone app, filed taxes online, or checked a bank balance, you have interacted with systems built on OOP principles, whether you knew it or not.

The idea is elegant enough to explain at a dinner table. The details are where it gets interesting.

The Blueprint and the Building

The central concept in object-oriented programming is the object. An object is a bundle of two things: data that describes something, and behavior, meaning the actions that thing can perform or have performed on it.

Consider a bank account. A bank account has data associated with it: an account number, an owner's name, a current balance. It also has behaviors: deposit money, withdraw money, check the balance, or close the account. In an object-oriented program, a bank account would be modeled as an object that contains both the balance and the rules for how the balance changes.

A bank has millions of accounts, not one. You do not want to describe the structure of a bank account a million times. Instead, you write it once, as a template. In OOP, that

template is called a class. The individual accounts created from that template are called instances, or more commonly, objects.

The class is the blueprint. The object is the building.

> *The class is the blueprint. The object is the building, and you can build as many as you need from the same plan.*

In code, a simple bank account class might look like this:

```
class BankAccount {
  #balance;

  constructor(owner, initialBalance) {
    this.owner = owner;
    this.#balance = initialBalance;
  }

  deposit(amount) {
    this.#balance += amount;
  }

  withdraw(amount) {
    if (amount > this.#balance) {
      throw new Error('Insufficient funds');
    }
    this.#balance -= amount;
  }

  getBalance() {
    return this.#balance;
  }
}

// Create two accounts from the same blueprint
const aliceAccount = new BankAccount('Alice', 1000);
const bobAccount = new BankAccount('Bob', 250);
```

Alice and Bob now each have their own bank account object. They were created from the same class, but they are independent. A deposit into Alice's account has no effect on Bob's.

Encapsulation: What You Do Not Need to Know

One of the most important ideas in object-oriented programming goes by the somewhat intimidating name encapsulation. The concept is simple: an object should manage its own data. The outside world interacts with it through a defined set of actions, and does not reach in to manipulate the numbers directly.

In the bank account example above, the balance is declared as a private field using the # prefix. That means code outside the class cannot read or assign to it. The only way to change the balance is through deposit() or withdraw(). Those methods contain the rules, which is the business logic. The withdrawal method checks whether the account has enough money before subtracting anything. A direct assignment would skip that check, and the language prevents it from compiling.

Encapsulation is why, when you use an ATM, the machine does not let you type any number you want into the balance field. The balance is protected. You interact with it through defined operations, each of which enforces the appropriate rules.

This is more than good security practice. It is good software design. When the rules for how a balance changes live in one place, inside the class, you only have to update them in one place if the rules ever change. The rest of the program does not care. It calls deposit() and expects it to work correctly.

Inheritance: Building on What Already Exists

The second major pillar of OOP is inheritance. The idea is that classes can be built on top of other classes, inheriting their data and behavior, then adding or changing what is needed.

Suppose your bank also offers savings accounts and checking accounts. Both share the core properties of a bank account: owner, balance, deposit, withdraw. A savings account might also earn interest, and a checking account might have an overdraft protection feature. Rather than writing two completely separate classes from scratch, you can write a base BankAccount class and then extend it:

```
class SavingsAccount extends BankAccount {
  constructor(owner, initialBalance, interestRate) {
    super(owner, initialBalance); // Inherit from BankAccount
    this.interestRate = interestRate;
  }

  applyInterest() {
    const interest = this.getBalance() * this.interestRate;
    this.deposit(interest);
  }
}

const aliceSavings = new SavingsAccount('Alice', 5000, 0.04);
aliceSavings.deposit(500);     // Inherited from BankAccount
aliceSavings.applyInterest();  // New behavior unique to savings
```

SavingsAccount gets everything BankAccount already knows how to do, for free. It only has to define what is different about itself. This is the software equivalent of evolution: new species inherit traits from their ancestors and add adaptations of their own.

Inheritance reduces repetition. When a bug is fixed in the base class, every class that inherits from it benefits automatically. It creates a natural hierarchy that mirrors how we tend to think about categories of things in the real world.

> *Inheritance is the software equivalent of evolution: new types inherit traits from ancestors and add what makes them distinct.*

Polymorphism: One Interface, Many Behaviors

The third pillar has the most formidable name: polymorphism. From Greek, meaning many forms. The idea is that different types of objects can respond to the same message in different ways.

Imagine your banking software needs to print a summary for any account. You do not want to write separate summary-printing code for every account type. Instead, you give every account type a printSummary() method, and each one handles it appropriately:

```
class CheckingAccount extends BankAccount {
  printSummary() {
    return `Checking: ${this.owner} - Balance: $${this.getBalance()}`;
  }
}

class SavingsAccount extends BankAccount {
  printSummary() {
    return `Savings: ${this.owner} - Balance: $${this.getBalance()} ` +
           `(Rate: ${this.interestRate * 100}%)`;
  }
}

// The calling code does not need to know which type each account is
const accounts = [
  new CheckingAccount('Bob', 500),
  new SavingsAccount('Alice', 5000, 0.04),
];

accounts.forEach(account => console.log(account.printSummary()));
```

The loop does not care what kind of account it is dealing with. It calls printSummary() and trusts that each object knows what to do. Each type responds in its own way to the same request. That is polymorphism, and it is why large systems can add new account types years after the original code was written without changing any of the code that uses them.

Why OOP Took Over

Object-oriented programming became dominant in the 1980s and 1990s for practical reasons. As software systems grew larger and more complex, the old approach of long lists of instructions executing from top to bottom became impossible to manage. Finding a bug meant reading thousands of lines of procedural code. Making a change meant worrying about what else might break.

OOP gave programmers a way to divide large systems into manageable pieces. Each object is a self-contained unit of concern. A team building a banking system could divide the work: one group builds the account classes, another group builds the transaction processing, another group builds the reporting. As long as each piece has a clearly defined interface, the teams do not have to understand each other's code in detail to work together.

This modularity, the ability to build complex systems out of well-defined, interchangeable parts, is what made OOP indispensable at industrial scale.

The Critiques

OOP is not without its critics, and the criticisms are worth understanding.

The inheritance hierarchies that OOP encourages can become deeply tangled over time. A class that inherits from a class that inherits from a class creates chains of dependencies that are hard to follow, harder to debug, and difficult to change without breaking something upstream. There is a well-known piece of software developer advice: favor composition over inheritance. Rather than building deep family trees of classes, build objects that contain other objects as components.

There is also the argument that OOP encourages programmers to model state, meaning the current values of all those object properties, when state is often the source of the hardest bugs to find. A balance that changes because five different methods modified it in a complex sequence is harder to reason about than a balance that was calculated fresh from a transaction log every time it was needed. This critique is at the heart of functional programming, which approaches the problem from the opposite direction.

These are arguments about philosophy and tradeoffs, not about OOP being wrong. For the right problems, such as modeling real-world entities with complex rules, or building large systems with many developers, OOP remains one of the most powerful organizational tools in software development.

A Note on What Object-Oriented Actually Means in Practice

Most working programmers do not sit down with a pure OOP mindset. Modern languages like Python and JavaScript are multi-paradigm. They support OOP but do not require it. A real codebase uses classes where classes make sense and simpler functions where they do not. The goal is always the same: code that is readable, maintainable, and correct.

OOP is a lens, not a law. It is a way of seeing a problem, as a collection of entities with data and behavior, that has proven extraordinarily useful. The fact that you have probably never thought about it while using the software it produced is, in a way, the highest compliment you can pay it.

Avalynn Circe is a technical writer and software developer based in St. Paul, MN. She is the author of JavaScript: The Parts, available at leanpub.com/jsparts.