Objects All the Way Down

The History and Philosophy of Object-Oriented Programming

Brendan Shea, PhD

Programming and Problem Solving • Rochester Community and Technical College

Introduction

When you write `String greeting = "Hello"` in Java, you're participating in a philosophy of programming that has shaped software development for over fifty years. That `String` isn't just data—it's an **object**: a bundle of data and behavior, a little machine that knows how to do things like report its length, convert to uppercase, or find substrings within itself.

Object-oriented programming (OOP) is so dominant today that it can seem like the natural, obvious way to write software. But OOP is a human invention with a specific history, created by specific people trying to solve specific problems. Understanding that history illuminates not just how to use objects, but *why* they exist, what problems they solve, and what tradeoffs they embody.

This case study traces OOP from its origins in 1960s Norway through its explosive growth in the 1990s to ongoing debates about its future. Along the way, you'll meet the visionaries who shaped how we think about software, explore the philosophical ideas underlying OOP, and examine how modern languages like Java continue to evolve.

The Central Metaphor

Object-oriented programming asks us to think about software as a collection of **objects**—independent entities that have their own data (state) and their own behaviors (methods). Objects interact by sending messages to each other, much like people or organizations in the real world.

This metaphor is powerful but not inevitable. Other paradigms organize code around functions, logic rules, or data transformations. OOP's dominance reflects both its genuine strengths and historical contingency.

1 The Problem of Complexity

To understand why OOP emerged, we must first understand what problems it was trying to solve.

By the 1960s, software was growing increasingly complex. Programs that had once fit in a few hundred lines now sprawled across thousands. Teams of programmers needed to collaborate on single systems. And the **software crisis** loomed: projects ran over budget, delivered late, and failed to meet requirements. The tools and techniques that worked for small programs didn't scale.

The fundamental challenge was **managing complexity**. How do you organize a program with 100,000 lines of code so that humans can understand it? How do you divide work among programmers so they don't constantly interfere with each other? How do you change one part of a system without breaking everything else?

The Software Crisis

At the 1968 NATO Software Engineering Conference, participants coined the term "software crisis" to describe the state of the industry. Software projects routinely failed. The IBM OS/360 operating system, one of the largest software projects of its era, ran years late and millions over budget. Project manager Fred Brooks later wrote *The Mythical Man-Month*, observing that adding programmers to a late project makes it later—because communication overhead grows faster than productivity.

The conference concluded that software development needed to become more like engineering: disciplined, methodical, and based on sound principles. The search was on for better ways to structure programs.

2 Simula: The Birth of Objects

The first object-oriented language emerged not from the software crisis directly, but from an unexpected domain: simulation.

Kristen Nygaard and Ole-Johan Dahl (1960s)

At the Norwegian Computing Center in Oslo, Kristen Nygaard and Ole-Johan Dahl were working on simulation problems: modeling ships in harbors, customers in banks, traffic in cities. They realized that simulation naturally involves *entities* with their own properties and behaviors—a ship has a position and speed; it can dock, load cargo, and depart.

In 1962, they began developing **Simula**, a language designed around this insight. By 1967, Simula 67 introduced the key concepts we now associate with OOP: classes, objects, inheritance, and polymorphism. Nygaard and Dahl had invented object-oriented programming—though the term wouldn't be coined until later.

Their work earned them the 2001 Turing Award, computing's highest honor. Sadly, both died in 2002, just months after receiving it.

Simula introduced the **class**: a template or blueprint defining what data an object contains and what operations it can perform. You could define a `Ship` class once, then create many individual ship objects from it.

Simula 67: An Early Class

```
CLASS Ship;
BEGIN
    INTEGER length, cargo;
    REAL speed;

    PROCEDURE dock;
    BEGIN
        speed := 0;
        ! Ship stops when docked
```

```
    END;

    PROCEDURE loadCargo(amount);
    INTEGER amount;
    BEGIN
        cargo := cargo + amount;
    END;
END;
```

Simula also introduced **inheritance**: a new class could be defined as an extension of an existing one, inheriting its data and behavior while adding or modifying features. A `CargoShip` could inherit from `Ship`, adding cargo-specific capabilities.

The key insight was that simulation's need to model real-world entities was actually a general programming need. The same techniques that helped simulate ships could help build any complex system.

3 Smalltalk: Objects as Philosophy

If Simula invented the mechanisms of OOP, **Smalltalk** articulated its philosophy. Developed at Xerox PARC in the 1970s under the leadership of Alan Kay, Smalltalk pursued the object metaphor to its logical extreme.

Alan Kay and the Dynabook Dream

Alan Kay joined Xerox PARC with a radical vision: the **Dynabook**, a portable computer for children that would revolutionize education and creativity. Though the hardware wasn't yet feasible, Kay's team built the software environment it would need: Smalltalk.

Kay was influenced by biology (cells as independent units), mathematics (Lisp's elegant uniformity), and educational psychology (children's learning styles). He wanted a system where everything—numbers, windows, even the programming environment itself—was an object that could be inspected, modified, and combined.

Kay is often credited with coining "object-oriented," though he later said the term led people astray. He emphasized *messaging*—objects communicating—over classes and inheritance. "I made up the term object-oriented," he wrote, "and I can tell you I did not have C++ in mind."

Smalltalk embodied a purer vision of OOP than most languages that followed:

Alan Kay on Objects

"I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages... OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things."

In Smalltalk, *everything* is an object—including numbers, booleans, and classes themselves. Even control flow works through message-passing: the boolean `true` responds to an `ifTrue:` message by executing the associated block.

Smalltalk: Everything is an Object

```
"Even integers are objects that receive messages"
3 + 4           "sends + message to 3 with argument 4"
'hello' size    "sends size message to string"

"Conditionals are messages to booleans"
(x > 0) ifTrue: [Transcript show: 'positive']
        ifFalse: [Transcript show: 'non-positive']

"Creating objects"
ship := Ship new.
ship loadCargo: 50.
```

Smalltalk also pioneered the graphical programming environment: windows, menus, and a mouse-driven interface—technologies that Xerox PARC developed and Apple later commercialized. The Smalltalk IDE allowed programmers to inspect and modify running objects, blurring the line between programming and using software.

4 The OOP Explosion: C++ and Java

Simula and Smalltalk were influential but not widely adopted in mainstream industry. That changed in the 1980s and 1990s as OOP went from academic curiosity to industry standard.

Bjarne Stroustrup and C++ (1979–)

Bjarne Stroustrup, a Danish computer scientist at Bell Labs, wanted to combine the efficiency of C with the organizational power of Simula. Starting in 1979, he developed "C with Classes," which evolved into **C++** by 1983.

C++ made a pragmatic choice: rather than pursuing object purity like Smalltalk, it added OOP features to a language programmers already knew. You could write C-style code where needed for performance, and use objects where they helped organization. This "multi-paradigm" approach—OOP without dogma—proved enormously appealing.

By the 1990s, C++ dominated systems programming, games, and applications requiring high performance. Its influence shaped a generation of languages.

James Gosling and Java (1991–1995)

At Sun Microsystems, James Gosling led a team developing software for interactive television. The project failed commercially, but produced a new language: **Java**.

Java took lessons from C++ (and its pain points): automatic memory management eliminated whole categories of bugs; a simpler object model avoided C++'s complexity; and compilation to bytecode running on a virtual machine enabled "write once, run anywhere" portability.

Java launched publicly in 1995, riding the wave of the early web (Java applets could run in browsers). By the 2000s, it dominated enterprise software development. Its influence extended to C#, Kotlin, and countless other languages.

Java struck a balance between Smalltalk's object purity and C++'s pragmatism. Almost everything is an object (except primitives like `int` and `boolean`), but the language provides familiar

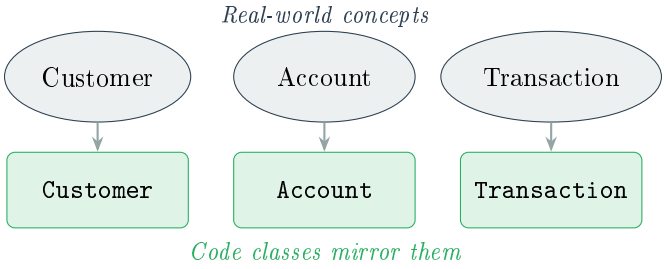
control structures rather than message-passing for conditionals and loops.

5 The Philosophy: Modeling the World

What makes OOP philosophically distinctive? Several interconnected ideas:

5.1 Objects as Models

OOP encourages thinking of programs as **models** of some domain. A banking system has `Account`, `Customer`, and `Transaction` objects because banks have accounts, customers, and transactions. The code mirrors the conceptual structure of what it represents.



This modeling perspective makes OOP intuitive for many domains. But it also raises questions: How do we model concepts that don't have real-world counterparts? Is the "real world" really made of objects, or is that just one way of seeing it?

5.2 Encapsulation: Hiding Complexity

Encapsulation means bundling data with the operations that manipulate it, and hiding internal details behind a public interface. A `BankAccount` object manages its own balance; other code interacts only through methods like `deposit()` and `withdraw()`, never touching the balance directly.

Encapsulation in Java

```
public class BankAccount {
    private double balance;  // Hidden from outside

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public boolean withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            return true;
        }
        return false;  // Withdrawal failed
    }

    public double getBalance() {
        return balance;  // Controlled access
```

```
    }
}
```

Encapsulation enables **information hiding**: you can change how a class works internally without affecting code that uses it. As long as the public interface stays stable, the implementation can evolve freely.

5.3 Inheritance: Sharing and Specializing

Inheritance allows new classes to be defined as extensions of existing ones. A `SavingsAccount` can inherit from `BankAccount`, gaining all its capabilities while adding interest calculation.

Inheritance: Extending a Class

```
public class SavingsAccount extends BankAccount {
    private double interestRate;

    public SavingsAccount(double rate) {
        this.interestRate = rate;
    }

    public void applyInterest() {
        double interest = getBalance() * interestRate;
        deposit(interest);
    }
}
```

Inheritance creates an "is-a" relationship: a `SavingsAccount` *is a* `BankAccount`. Anywhere a `BankAccount` is expected, a `SavingsAccount` can be used.

5.4 Polymorphism: Many Forms

Polymorphism (from Greek: "many forms") means that code can work with objects of different types through a common interface. A method that accepts a `BankAccount` can work with any subclass—`SavingsAccount`, `CheckingAccount`, or types not yet invented.

Polymorphism in Action

```
public void printStatement(BankAccount account) {
    // Works with ANY kind of BankAccount
    System.out.println("Balance: $" + account.getBalance());
}

// Can be called with different types
printStatement(new SavingsAccount(0.02));
printStatement(new CheckingAccount());
```

Polymorphism enables extensibility: new types can be added without modifying existing code that works with the parent type.

6 Critiques and Ongoing Debates

OOP's dominance hasn't gone unchallenged. Several critiques have emerged over decades of experience.

Inheritance: Blessing or Curse?

Inheritance seemed like a great idea: define common behavior once, specialize where needed. In practice, deep inheritance hierarchies often became fragile and confusing. Changing a base class could break subclasses in unexpected ways (the "fragile base class problem").

Many experienced developers now advise: **"Favor composition over inheritance."** Instead of making `SavingsAccount` inherit from `BankAccount`, have it *contain* a `BankAccount` and delegate to it. This is more flexible but more verbose.

Java's designers have moved in this direction: default methods on interfaces, records, and sealed classes provide alternatives to classical inheritance hierarchies.

Objects vs. Functions

Functional programming—organizing code around pure functions and immutable data—offers a different paradigm. Functional advocates argue that OOP's mutable objects make programs harder to reason about, especially in concurrent systems. When objects can change state, tracking what happened when becomes difficult.

The debate isn't binary. Modern languages increasingly blend paradigms: Java added lambdas and streams; functional languages like Scala include objects. Perhaps the question isn't "which is better?" but "which is better for what?"

Does OOP Model Reality?

OOP claims to model the real world, but critics argue this is misleading. The world doesn't obviously consist of discrete objects with clear boundaries. A "customer" in code might correspond to a database record, a legal entity, a biological person, or a social role—all different things with different behaviors.

Moreover, some problems don't fit the object model well. Mathematical computations, data transformations, and event-driven systems may be more naturally expressed in other paradigms.

7 Modern Developments: Java Evolves

Java hasn't stood still. Recent versions have introduced features that respond to decades of OOP experience—and borrow ideas from other paradigms.

7.1 Records: Data Without Boilerplate

Sometimes you just want a class that holds data. Traditional Java required writing constructors, getters, `equals()`, `hashCode()`, and `toString()`—tedious boilerplate. **Records** (Java 14+) generate all of this automatically:

Records: Concise Data Classes (Java 14+)

```
// Old way: ~30 lines of boilerplate
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public int getX() { return x; }
    public int getY() { return y; }
    // Plus equals(), hashCode(), toString()...
}

// New way: one line
public record Point(int x, int y) { }

// Usage is identical
Point p = new Point(3, 4);
System.out.println(p.x());  // 3
```

Records are immutable by default—a nod toward functional programming's emphasis on unchanging data.

7.2 Sealed Classes: Controlled Inheritance

Traditional inheritance is open: anyone can extend your class. **Sealed classes** (Java 17) let you specify exactly which classes can extend yours:

Sealed Classes: Controlled Hierarchies (Java 17)

```
public sealed class Shape
    permits Circle, Rectangle, Triangle {
    // Only these three can extend Shape
}

public final class Circle extends Shape {
    private double radius;
    // ...
}

public final class Rectangle extends Shape {
    private double width, height;
    // ...
}
```

This enables exhaustive pattern matching: the compiler can verify you've handled all cases.

7.3 Pattern Matching: Smarter Conditionals

Pattern matching (Java 16+) lets you test an object's type and extract data in one step:

Pattern Matching (Java 16+)

```java
// Old way: instanceof then cast
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println(s.length());
}

// New way: pattern matching
if (obj instanceof String s) {
    System.out.println(s.length());  // s already available
}

// Switch expressions with patterns (Java 21)
String description = switch (shape) {
    case Circle c    -> "Circle with radius " + c.radius();
    case Rectangle r -> "Rectangle " + r.width() + "x" + r.height();
    case Triangle t  -> "Triangle";
};
```

7.4 Functional Features

Java 8 introduced **lambdas** and the **Stream API**, bringing functional programming into the object-oriented world:

Functional Programming in Java

```java
List<String> names = List.of("Alice", "Bob", "Charlie");

// Lambda expression
names.forEach(name -> System.out.println(name));

// Stream pipeline: filter, transform, collect
List<String> longNames = names.stream()
    .filter(name -> name.length() > 4)
    .map(String::toUpperCase)
    .toList();
// Result: ["ALICE", "CHARLIE"]
```

These features don't replace OOP but complement it. Modern Java is increasingly multi-paradigm.

8 Beyond Java: The Landscape Today

Java's influence extends to newer languages that refine its ideas:

Kotlin (JetBrains, 2011) runs on the JVM but with more concise syntax, null safety built into the type system, and first-class support for functional programming. It's now Google's preferred language for Android development.

Scala (Martin Odersky, 2004) blends OOP and functional programming more deeply, with powerful type inference and pattern matching. It's popular for big data processing (Apache Spark).

Swift (Apple, 2014) and **Rust** (Mozilla, 2010) take different approaches to OOP: Swift uses protocols (like interfaces) rather than inheritance; Rust doesn't have inheritance at all, relying on traits and composition.

The trend is toward **multi-paradigm** languages that support objects when useful but don't force everything into the object mold.

9 The Ongoing Conversation

Object-oriented programming emerged from specific historical circumstances: the software crisis, simulation research, visions of personal computing. Its success reshaped the industry, but its core ideas remain contested and evolving.

Some questions for reflection: Is OOP's dominance deserved, or partly historical accident? Would a world that had bet on functional programming look very different? As AI increasingly writes code, does the paradigm that helps *humans* organize programs even matter?

These aren't questions with settled answers. The history of programming is still being written—and you're now part of it.

Discussion Questions

Discussion Questions

1. **Modeling Reality:** OOP encourages modeling programs after real-world entities. But the "real world" can be carved up in many ways—there's no single right set of objects. How do programmers decide what objects to create? Is this a technical decision, a design decision, or something else?

2. **Inheritance Reconsidered:** Early OOP emphasized inheritance as a key benefit. Modern practice often prefers composition. What went wrong with inheritance? Are there cases where it's still the right choice? How do newer features like records and sealed classes change the calculus?

3. **Paradigm Wars:** Object-oriented and functional programming represent different philosophies. Is one "better," or are they suited to different problems? What might a program look like that uses both approaches effectively?

4. **Historical Contingency:** OOP's dominance owes much to historical factors: C++'s compatibility with C, Java's timing with the web, corporate backing, university curricula. If history had gone differently—if Smalltalk or Lisp had become mainstream—how might software development be different today?

5. **The Future of OOP:** Languages keep evolving, blending paradigms, and adding features. Will "object-oriented programming" still be a meaningful category in 20 years? What might replace or transform it?

Key Terms

Glossary of Key Terms

Class	A blueprint or template defining the data and behavior that objects of that type will have.
Composition	Building complex objects by combining simpler ones, often preferred over inheritance for flexibility.
Encapsulation	Bundling data with methods that operate on it, hiding internal details behind a public interface.
Functional Programming	A paradigm organizing code around pure functions and immutable data, contrasting with OOP's mutable objects.
Inheritance	A mechanism where new classes are defined as extensions of existing ones, inheriting their data and behavior.
Lambda	An anonymous function that can be passed as a value, enabling functional programming patterns.
Object	An instance of a class, combining state (data) and behavior (methods) into a single entity.
Pattern Matching	A feature allowing code to test an object's type and extract its data simultaneously.
Polymorphism	The ability to treat objects of different types uniformly through a common interface.
Record	A concise way to declare classes that are primarily data carriers, with automatically generated boilerplate.
Sealed Class	A class that specifies exactly which other classes may extend it, enabling exhaustive handling.
Stream API	Java's library for functional-style operations on collections, using pipelines of filter, map, and reduce.