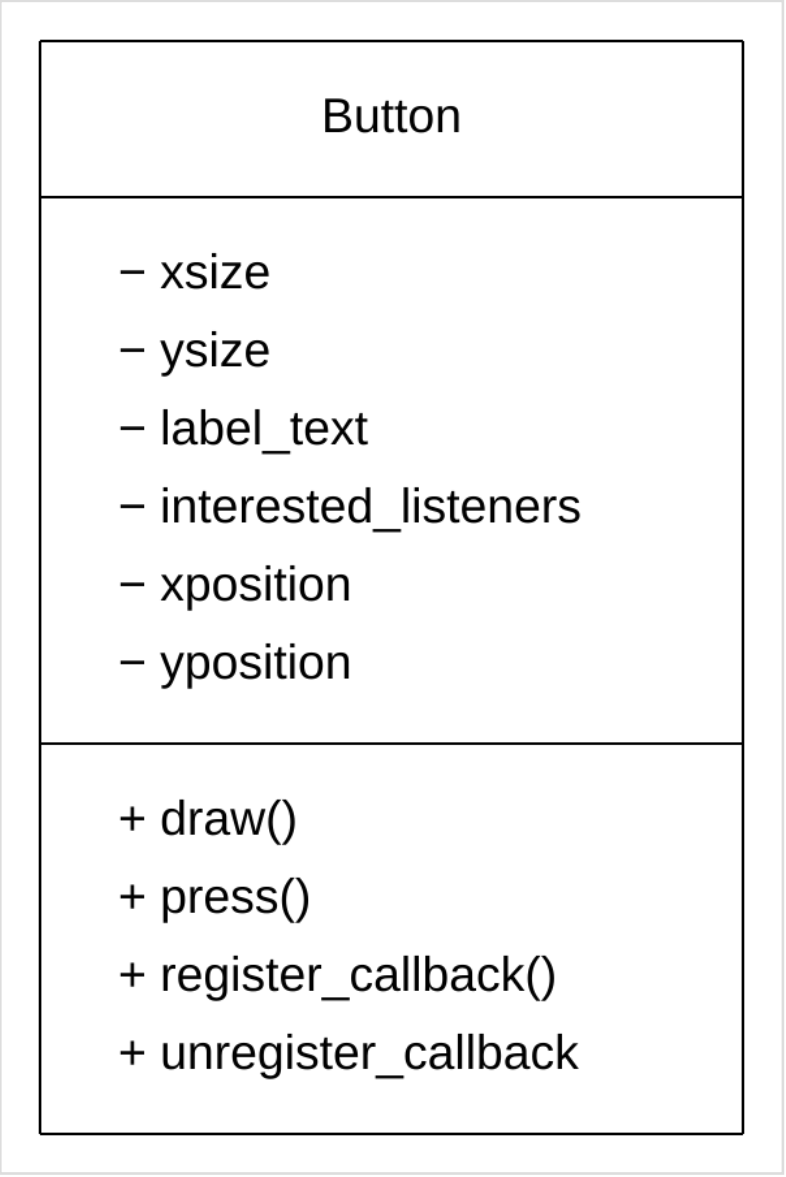
WIKIPEDIA
The Free Encyclopedia

Object-oriented programming

Object-oriented programming (**OOP**) is a programming paradigm based on the concept of *objects*.[1] Objects can contain data (called fields, attributes or properties) and have actions they can perform (called procedures or methods and implemented in code). In OOP, computer programs are designed by making them out of objects that interact with one another.[2][3]



UML notation for a class. This Button class has variables for data, and functions. Through inheritance, a subclass can be created as a subset of the Button class. Objects are instances of a class.

Many of the most widely used programming languages (such as C++, Java,[4] and Python) support object-oriented programming to a greater or lesser degree, typically as part of multiple paradigms in combination with others such as imperative programming and declarative programming.

Significant object-oriented languages include Ada, ActionScript, C++, Common Lisp, C#, Dart, Eiffel, Fortran 2003, Haxe,

Java,[4] JavaScript, Kotlin, Logo, MATLAB, Objective-C, Object Pascal, Perl, PHP, Python, R, Raku, Ruby, Scala, SIMSCRIPT, Simula, Smalltalk, Swift, Vala and Visual Basic.NET.

History

The idea of "objects" in programming started with the artificial intelligence group at MIT in the late 1950s and early 1960s. Here, "object" referred to LISP atoms with identified properties (attributes).[5][6] Another early example was Sketchpad created by Ivan Sutherland at MIT in 1960–1961. In the glossary of his technical report, Sutherland defined terms like "object" and "instance" (with the class concept covered by "master" or "definition"), albeit specialized to graphical interaction.[7] Later, in 1968, AED-0, MIT's version of the ALGOL programming language, connected data structures ("plexes") and procedures, prefiguring what were later termed "messages", "methods", and "member functions".[8][9] Topics such as data abstraction and modular programming were common points of discussion at this time.

Meanwhile, in Norway, Simula was developed during the years 1961–1967.[8] Simula introduced essential object-oriented ideas, such as classes, inheritance, and dynamic binding.[10] Simula was used mainly by researchers involved with physical modelling, like the movement of ships and their content through cargo ports.[10] Simula is generally accepted as being the first language with the primary features and framework of an object-oriented language.[11]

Influenced by both MIT and Simula, Alan Kay began developing his own ideas in November 1966. He would go on to create Smalltalk, an influential object-oriented programming language. By 1967, Kay was already using the term "object-oriented programming" in

> I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning – it took a while to see

conversation.[1] Although sometimes called the "father" of object-oriented programming,[12] Kay has said his ideas differ from how object-oriented programming is commonly understood, and has implied that the computer science establishment did not adopt his notion.[1] A 1976 MIT memo co-authored by Barbara Liskov lists Simula 67, CLU, and Alphard as object-oriented languages, but does not mention Smalltalk.[13]

> how to do messaging in a programming language efficiently enough to be useful).
>
> Alan Kay, [1]

In the 1970s, the first version of the Smalltalk programming language was developed at Xerox PARC by Alan Kay, Dan Ingalls and Adele Goldberg. Smalltalk-72 was notable for use of objects at the language level and its graphical development environment.[14] Smalltalk was a fully dynamic system, allowing users to create and modify classes as they worked.[15] Much of the theory of OOP was developed in the context of Smalltalk, for example multiple inheritance.[16]

In the late 1970s and 1980s, object-oriented programming rose to prominence. The Flavors object-oriented Lisp was developed starting 1979, introducing multiple inheritance and mixins.[17] In August 1981, Byte Magazine highlighted Smalltalk and OOP, introducing these ideas to a wide audience.[18] LOOPS, the object system for Interlisp-D, was influenced by Smalltalk and Flavors, and a paper about it was published in 1982.[19] In 1986, the first *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA) was attended by 1,000 people. This conference marked the beginning of efforts to consolidate Lisp object systems, eventually resulting in the Common Lisp Object System. In the 1980s, there were a few attempts to design processor architectures that included hardware support for objects in memory, but these were not successful. Examples include the Intel iAPX 432 and the Linn Smart Rekursiv.

In the mid-1980s, new object-oriented languages like Objective-C, C++, and Eiffel emerged. Objective-C was developed by Brad Cox, who had used Smalltalk at ITT Inc.. Bjarne Stroustrup created C++ based on his experience using Simula for his PhD thesis.[14] Bertrand Meyer produced the first design of the Eiffel language in 1985, which focused on software quality using a design by contract approach.[20]

In the 1990s, object-oriented programming became the main way of programming, especially as more languages supported it. These included Visual FoxPro 3.0,[21][22] C++,[23] and Delphi. OOP became even more popular with the rise of graphical user interfaces, which used objects for buttons, menus and other elements. One well-known example is Apple's Cocoa framework, used on Mac OS X and written in Objective-C. OOP toolkits also enhanced the popularity of event-driven programming.

At ETH Zürich, Niklaus Wirth and his colleagues created new approaches to OOP. Modula-2 (1978) and Oberon (1987), included a distinctive approach to object orientation, classes, and type checking across module boundaries. Inheritance is not obvious in Wirth's design since his nomenclature looks in the opposite direction: It is called type extension and the viewpoint is from the parent down to the inheritor.

Many programming languages that existed before OOP have added object-oriented features, including Ada, BASIC, Fortran, Pascal, and COBOL. This sometimes caused compatibility and maintainability issues, as these languages were not originally designed with OOP in mind.

In the new millenium, new languages like Python and Ruby have emerged that combine object-oriented and procedural styles. The most commercially important "pure" object-oriented languages continue to be Java, developed by Sun Microsystems, as well as C# and Visual Basic.NET (VB.NET), both designed for Microsoft's .NET platform. These languages show the benefits of OOP by creating abstractions from implementation. The .NET platform supports cross-language inheritance, allowing programs to use objects from multiple languages together.

Features

Object-oriented programming focuses on working with objects, but not all OOP languages have every feature linked to OOP. Below are some common features of languages that are considered strong in OOP or support it along with other programming styles. Important exceptions are also noted.[24][25][26][27] Christopher J. Date pointed out that comparing OOP with other styles, like relational programming, is difficult because there isn't a clear, agreed-upon definition of OOP.[28]

Imperative programming

Features from imperative and structured programming are present in OOP languages and are also found in non-OOP languages.

- Variables hold different data types like integers, strings, lists, and hash tables. Some data types are built-in while others result from combining variables using memory pointers.
- Procedures – also known as functions, methods, routines, or subroutines – take input, generate output, and work with data. Modern languages include structured programming constructs like loops and conditionals.

Support for modular programming lets programmers organize related procedures into files and modules. This makes programs easier to manage. Each modules has its own namespace, so items in one module will not conflict with items in another.

Object-oriented programming (OOP) was created to make code easier to reuse and maintain.[29] However, it was not designed to clearly show the flow of a program's instructions—that was left to the compiler. As computers began using more parallel processing and multiple threads, it became more important to understand and control how instructions flow. This is difficult to do with OOP.[30][31][32][33]

Objects

An object is a type of data structure that has two main parts: fields and methods. Fields may also be known as members, attributes, or properties, and hold information in the form of state variables. Methods are actions, subroutines, or procedures, defining the object's behavior in code. Objects are usually stored in memory, and in many programming languages, they work like pointers that link directly to a contiguous block containing the object instances's data.

Objects can contain other objects. This is called object composition. For example, an Employee object might have an Address object inside it, along with other information like "first_name" and "position". This type of structures shows "has-a" relationships, like "an employee has an address".

Some believe that OOP places too much focus on using objects rather than on algorithms and data structures.[34][35] For example, programmer Rob Pike pointed out that OOP can make programmers think more about type hierarchy than composition.[36] He has called object-oriented programming "the Roman numerals of computing".[37] Rich Hickey, creator of Clojure, described OOP as overly simplistic, especially when it comes to representing real-world things that change over time.[35] Alexander Stepanov said that OOP tries to fit everything into a single type, which can be limiting. He argued that sometimes we need multisorted algebras—families of interfaces that span multiple types, such as in generic programming. Stepanov also said that calling everything an "object" doesn't add much understanding.[34]

Real-world modeling and relationships

Sometimes, objects represent real-world things and processes in digital form.[38] For example, a graphics program may have objects such as "circle", "square", and "menu". An online shopping system might have objects such as "shopping cart", "customer", and "product". Niklaus Wirth said, "This paradigm [OOP] closely reflects the structure of systems in the real world and is therefore well suited to model complex systems with complex behavior".[39]

However, more often, objects represent abstract entities, like an open file or a unit converter. Not everyone agrees that OOP makes it easy to copy the real world exactly or that doing so is even necessary. Bob Martin suggests that because classes are software, their relationships don't match the real-world relationships they represent.[40] Bertrand Meyer argues in *Object-Oriented Software Construction*, that a program is not a model of the world but a model of some part of the world; "Reality is a cousin twice removed".[41] Steve Yegge noted that natural languages lack the OOP approach of strictly prioritizing *things* (objects/nouns) before *actions* (methods/verbs), as opposed to functional programming which does the reverse.[42] This can sometimes make OOP solutions more complicated than those written in procedural programming.[43]

Inheritance

Most OOP languages allow reusing and extending code through "inheritance". This inheritance can use either "classes" or "prototypes", which have some differences but use similar terms for ideas like "object" and "instance".

Class-based

In class-based programming, the most common type of OOP, every object is an instance of a specific *class*. The class defines the data format, like variables (e.g., name, age) and methods (actions the object can take). Every instance of the class has the same set of variables and methods. Objects are created using a special method in the class known as a constructor.

Here are a few key terms in class-based OOP:

- Class variables – belong to the *class itself*, so all objects in the class share one copy.
- Instance variables – belong to individual *objects*; every object has its own version of these variables.
- Member variables – refers to both the class and instance variables that are defined by a particular class.

- Class methods – linked to the *class itself* and can only use class variables.
- Instance methods – belong to *individual objects*, and can use both instance and class variables

Classes may inherit from other classes, creating a hierarchy of "subclasses". For example, an "Employee" class might inherit from a "Person" class. This means the Employee object will have all the variables from Person (like name variables) plus any new variables (like job position and salary). Similarly, the subclass may expand the interface with new methods. Most languages also allow the subclass to *override* the methods defined by superclasses. Some languages support multiple inheritance, where a class can inherit from more than one class, and other languages similarly support mixins or traits. For example, a mixin called UnicodeConversionMixin might add a method unicode_to_ascii() to both a FileReader and a WebPageScraper class.

Some classes are abstract, meaning they cannot be directly instantiated into objects; they're only meant to be inherited into other classes. Other classes are *utility* classes which contain only class variables and methods and are not meant to be instantiated or subclassed.[44]

Prototype-based

In prototype-based programming, there aren't any classes. Instead, each object is linked to another object, called its *prototype* or *parent*. In Self, an object may have multiple or no parents,[45] but in the most popular prototype-based language, Javascript, every object has exactly one *prototype* link, up to the base Object type whose prototype is null.

The prototype acts as a model for new objects. For example, if you have an object *fruit*, you can make two objects *apple* and *orange*, based on it. There is no *fruit* class, but they share traits from the *fruit* prototype. Prototype-based languages also allow objects to have their own unique properties, so the *apple* object might have an attribute *sugar_content*, while the *orange* or *fruit* objects do not.

No inheritance

Some languages, like Go, don't use inheritance at all.[46] Instead, they encourage "composition over inheritance", where objects are built using smaller parts instead of parent-child relationships. For example, instead of inheriting from class Person, the Employee class could simply contain a Person object. This lets the Employee class control how much of Person it exposes to other parts of the program. Delegation is another language feature that can be used as an alternative to inheritance.

Programmers have different opinions on inheritance. Bjarne Stroustrup, author of C++, has stated that it is possible to do OOP without inheritance.[47] Rob Pike has criticized inheritance for creating complicated hierarchies instead of simpler solutions.[48]

Inheritance and behavioral subtyping

People often think that if one class inherits from another, it means the subclass "is a" more specific version of the original class. This presumes the program semantics are that objects from the subclass can always replace objects from the original class without problems. This concept is known as behavioral subtyping, more specifically the Liskov substitution principle.

However, this is often not true, especially in programming languages that allow mutable objects, objects that change after they are created. In fact, subtype polymorphism as enforced by the type checker in OOP languages cannot guarantee behavioral subtyping in most if not all contexts. For example, the circle-ellipse problem is notoriously difficult to handle using OOP's concept of inheritance. Behavioral subtyping is undecidable in general, so it cannot be easily implemented by a compiler. Because of this, programmers must carefully design class hierarchies to avoid mistakes that the programming language itself cannot catch.

Dynamic dispatch

When a method is called on an object, the object itself—not outside code—decides which specific code to run. This process, called dynamic dispatch, usually happens at run time by checking a table linked to the object to find the correct method. In this context, a method call is also known as message passing, meaning the method name and its inputs are like a message sent to the object for it to act on. If the method choice depends on more than one type of object (such as other objects passed as parameters), it's called multiple dispatch.

Dynamic dispatch works together with inheritance: if an object doesn't have the requested method, it looks up to its parent class (delegation), and continues up the chain until it finds the method or reaches the top.

Data abstraction and encapsulation

Data abstraction is a way of organizing code so that only certain parts of the data are visible to related functions (data hiding). This helps prevent mistakes and makes the program easier to manage. Because data abstraction works well, many programming styles, like object-oriented programming and functional programming, use it as a key principle. Encapsulation is another important idea in programming. It means keeping the internal details of an object hidden from the outside code. This makes it easier to change how an object works on the inside without affecting other parts of the program, such as in code refactoring. Encapsulation also helps keep related code together (decoupling), making it easier for programmers to understand.

In object-oriented programming, objects act as a barrier between their internal workings and external code. Outside code can only interact with an object by calling specific *public* methods or variables. If a class only allows access to its data through methods and not directly, this is called information hiding. When designing a program, it's often recommended to keep data as hidden as possible. This means using local variables inside functions when possible, then private variables (which only the object can use), and finally public variables (which can be accessed by any part of the program) if

necessary. Keeping data hidden helps prevent problems when changing the code later.[49] Some programming languages, like Java, control information hiding by marking variables as `private` (hidden) or `public` (accessible).[50] Other languages, like Python, rely on naming conventions, such as starting a private method's name with an underscore. Intermediate levels of access also exist, such as Java's `protected` keyword, (which allows access from the same class and its subclasses, but not objects of a different class), and the `internal` keyword in C#, Swift, and Kotlin, which restricts access to files within the same module.[51]

Abstraction and information hiding are important concepts in programming, especially in object-oriented languages.[52] Programs often create many copies of objects, and each one works independently. Supporters of this approach say it makes code easier to reuse and intuitively represents real-world situations.[53] However, others argue that object-oriented programming does not enhance readability or modularity.[54][55] Eric S. Raymond has written that object-oriented programming languages tend to encourage thickly layered programs that destroy transparency.[56] Raymond compares this unfavourably to the approach taken with Unix and the C programming language.[56]

One programming principle, called the "open/closed principle", says that classes and functions should be "open for extension, but closed for modification". Luca Cardelli has stated that OOP languages have "extremely poor modularity properties with respect to class extension and modification", and tend to be extremely complex.[54] The latter point is reiterated by Joe Armstrong, the principal inventor of Erlang, who is quoted as saying:[55]

> The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

Leo Brodie says that information hiding can lead to copying the same code in multiple places (duplicating code),[57] which goes against the don't repeat yourself rule of software development.[58]

Polymorphism

Polymorphism is the use of one symbol to represent multiple different types.[59] In object-oriented programming, polymorphism more specifically refers to subtyping or subtype polymorphism, where a function can work with a specific interface and thus manipulate entities of different classes in a uniform manner.[60]

For example, imagine a program has two shapes: a circle and a square. Both come from a common class called "Shape." Each shape has its own way of drawing itself. With subtype polymorphism, the program doesn't need to know the type of each shape, and can simply call the "Draw" method for each shape. The programming language runtime will ensure the correct version of the "Draw" method runs for each shape. Because the details of each shape are handled inside their own classes, this makes the code simpler and more organized, enabling strong separation of concerns.

Open recursion

In object-oriented programming, objects have methods that can change or use the object's data. Many programming languages use a special word, like `this` or `self`, to refer to the current object. In languages that support open recursion, a method in an object can call other methods in the same object, including itself, using this special word. This allows a method in one class to call another method defined later in a subclass, a feature known as late binding.

OOP languages

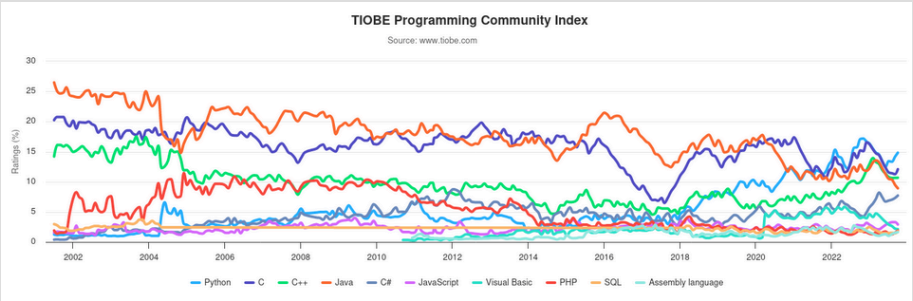
OOP languages can be grouped into different types based on how they support and use objects:

- Pure OOP languages: In these languages, everything is treated as an object, even basic things like numbers and characters. They are designed to fully support and enforce OOP. Examples: Ruby, Scala, Smalltalk, Eiffel, Emerald,[61] JADE, Self, Raku.
- Mostly OOP languages: These languages focus on OOP but also include some procedural programming features. Examples: Java, Python, C++, C#, Delphi/Object Pascal, VB.NET.
- Retrofitted OOP languages: These were originally designed for other types of programming but later added some OOP features. Examples: PHP, JavaScript, Perl, Visual Basic (derived from BASIC), MATLAB, COBOL 2002, Fortran 2003, ABAP, Ada 95, Pascal.
- Unique OOP languages: These languages have OOP features like classes and inheritance but use them in their own way. Examples: Oberon, BETA.
- Object-based languages: These support some OOP ideas but avoid traditional class-based inheritance in favor of direct manipulation of objects. Examples: JavaScript, Lua, Modula-2, CLU, Go.
- Multi-paradigm languages: These support both OOP and other programming styles, but OOP is not the predominant style in the language. Examples include Tcl, where TclOO allows both prototype-based and class-based OOP, and Common Lisp, with its Common Lisp Object System.

Popularity and reception

Many popular programming languages, like C++, Java, and Python, use object-oriented programming. In the past, OOP was widely accepted,[62] but recently, some programmers have criticized it and prefer functional programming instead.[63] A study by Potok et al. found no major difference in productivity between OOP and other methods.[64]

Paul Graham, a well-known computer scientist, believes big companies like OOP because it helps manage large teams of average programmers. He argues that OOP adds structure, making it harder for one person to make serious mistakes, but at the same time restrains smart programmers.[65] Eric S. Raymond, a Unix programmer and open-source software advocate, argues that OOP is not the best way to write programs.[56]



The TIOBE programming language popularity index graph from 2002 to 2023. In the 2000s the object-oriented Java (orange) and the procedural C (dark blue) competed for the top position.

Richard Feldman says that, while OOP features helped some languages stay organized, their popularity comes from other reasons.[66] Lawrence Krubner argues that OOP doesn't offer special advantages compared to other styles, like functional programming, and can make coding more complicated.[67] Luca Cardelli says that OOP is slower and takes longer to compile than procedural programming.[54]

OOP in dynamic languages

In recent years, object-oriented programming (OOP) has become very popular in dynamic programming languages. Some languages, like Python, PowerShell, Ruby and Groovy, were designed with OOP in mind. Others, like Perl, PHP, and ColdFusion, started as non-OOP languages but added OOP features later (starting with Perl 5, PHP 4, and ColdFusion version 6).

On the web, HTML, XHTML, and XML documents use the Document Object Model (DOM), which works with the JavaScript language. JavaScript is a well-known example of a prototype-based language. Instead of using classes like other OOP languages, JavaScript creates new objects by copying (or "cloning") existing ones. Another language that uses this method is Lua.

OOP in a network protocol

When computers communicate in a client-server system, they send messages to request services. For example, a simple message might include a length field (showing how big the message is), a code that identifies the type of message, and a data value. These messages can be designed as structured objects that both the client and server understand, so that each type of message corresponds to a class of objects in the client and server code. More complex messages might include structured objects as additional details. The client and server need to know how to serialize and deserialize these messages so they can be transmitted over the network, and map them to the appropriate object types. Both clients and servers can be thought of as complex object-oriented systems.

The Distributed Data Management Architecture (DDM) uses this idea by organizing objects into four levels:

1. Basic message details - Information like message length, type, and data.
2. Objects and collections - Similar to how objects work in Smalltalk, storing messages and their details.
3. Managers - Like file directories, these organize and store data, as well as provide memory and processing power. They are similar to IBM i Objects.
4. Clients and servers - These are full systems that include managers and handle security, directory services, and multitasking.

The first version of DDM defined distributed file services. Later, it was expanded to support databases through the Distributed Relational Database Architecture (DRDA).

Design patterns

Design patterns are common solutions to problems in software design. Some design patterns are especially useful for object-oriented programming, and design patterns are typically introduced in an OOP context.

Object patterns

The following are notable software design patterns for OOP objects.[68]

- Function object: Class with one main method that acts like an anonymous function (in C++, the function operator, `operator()`)
- Immutable object: does not change state after creation
- First-class object: can be used without restriction
- Container object: contains other objects
- Factory object: creates other objects
- Metaobject: Used to create other objects (similar to a class, but an object)
- Prototype object: a specialized metaobject that creates new objects by copying itself
- Singleton object: only instance of its class for the lifetime of the program
- Filter object: receives a stream of data as its input and transforms it into the object's output

A common anti-pattern is the God object, an object that knows or does too much.

Gang of Four design patterns

Design Patterns: Elements of Reusable Object-Oriented Software is a famous book published in 1994 by four authors: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. People often call them the "Gang of Four". The book talks about the strengths and weaknesses of object-oriented programming and explains 23 common ways to solve programming problems.

These solutions, called "design patterns," are grouped into three types:

- *Creational patterns* (5): Factory method pattern, Abstract factory pattern, Singleton pattern, Builder pattern, Prototype pattern

- *Structural patterns* (7): Adapter pattern, Bridge pattern, Composite pattern, Decorator pattern, Facade pattern, Flyweight pattern, Proxy pattern
- *Behavioral patterns* (11): Chain-of-responsibility pattern, Command pattern, Interpreter pattern, Iterator pattern, Mediator pattern, Memento pattern, Observer pattern, State pattern, Strategy pattern, Template method pattern, Visitor pattern

Object-orientation and databases

Both object-oriented programming and relational database management systems (RDBMSs) are widely used in software today. However, relational databases don't store objects directly, which creates a challenge when using them together. This issue is called object-relational impedance mismatch.

To solve this problem, developers use different methods, but none of them are perfect.[69] One of the most common solutions is object-relational mapping (ORM), which helps connect object-oriented programs to relational databases. Examples of ORM tools include Visual FoxPro, Java Data Objects, and Ruby on Rails ActiveRecord.

Some databases, called object databases, are designed to work with object-oriented programming. However, they have not been as popular or successful as relational databases.

Date and Darwen have proposed a theoretical foundation that uses OOP as a kind of customizable type system to support RDBMSs, but it forbids objects containing pointers to other objects.[70]

Responsibility- vs. data-driven design

In responsibility-driven design, classes are built around what they need to do and the information they share, in the form of a contract. This is different from data-driven design, where classes are built based on the data they need to store. According to Wirfs-Brock and Wilkerson, the originators of responsibility-driven design, responsibility-driven design is the better approach.[71]

SOLID and GRASP guidelines

SOLID is a set of five rules for designing good software, created by Michael Feathers:

- Single responsibility principle: A class should have only one reason to change.
- Open/closed principle: Software entities should be open for extension, but closed for modification.
- Liskov substitution principle: Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
- Interface segregation principle: Clients should not be forced to depend upon interfaces that they do not use.
- Dependency inversion principle: Depend upon abstractions, not concretes.

GRASP (General Responsibility Assignment Software Patterns) is another set of software design rules, created by Craig Larman, that helps developers assign responsibilities to different parts of a program:[72]

- Creator Principle: allows classes create objects they closely use.
- Information Expert Principle: assigns tasks to classes with the needed information.
- Low Coupling Principle: reduces class dependencies to improve flexibility and maintainability.
- High Cohesion Principle: designing classes with a single, focused responsibility.
- Controller Principle: assigns system operations to separate classes that manage flow and interactions.
- Polymorphism: allows different classes to be used through a common interface, promoting flexibility and reuse.
- Pure Fabrication Principle: create helper classes to improve design, boost cohesion, and reduce coupling.

Formal semantics

In object-oriented programming, objects are things that exist while a program is running. An object can represent anything, like a person, a place, a bank account, or a table of data. Many researchers have tried to formally define how OOP works. Records are the basis for understanding objects. They can represent fields, and also methods, if function literals can be stored. However, inheritance presents difficulties, particularly with the interactions between open recursion and encapsulated state. Researchers have used recursive types and co-algebraic data types to incorporate essential features of OOP.[73] Abadi and Cardelli defined several extensions of System $F_{<:}$ that deal with mutable objects, allowing both subtype polymorphism and parametric polymorphism (generics), and were able to formally model many OOP concepts and constructs.[74] Although far from trivial, static analysis of object-oriented programming languages such as Java is a mature field,[75] with several commercial tools.[76]

See also

Computer programming portal

- Comparison of programming languages (object-oriented programming)
- Component-based software engineering
- Object association
- Object modeling language
- Object-oriented analysis and design
- Object-oriented ontology

Systems

- CADES
- Common Object Request Broker Architecture (CORBA)
- Distributed Component Object Model

- Jeroo

Modeling languages

- IDEF4
- Interface description language
- UML

References

1. "Dr. Alan Kay on the Meaning of "Object-Oriented Programming" " (http://www.purl.org/stefan_ram/pub/doc_kay_oop_en). 2003. Retrieved 11 February 2010.
2. Kindler, E.; Krivy, I. (2011). "Object-Oriented Simulation of systems with sophisticated control". *International Journal of General Systems*. **40** (3): 313–343. doi:10.1080/03081079.2010.539975 (https://doi.org/10.1080%2F03081079.2010.539975).
3. Lewis, John; Loftus, William (2008). *Java Software Solutions Foundations of Programming Design 6th ed*. Pearson Education Inc. ISBN 978-0-321-53205-3., section 1.6 "Object-Oriented Programming"
4. Bloch 2018, pp. xi–xii, Foreword.
5. McCarthy, J.; Brayton, R.; Edwards, D.; Fox, P.; Hodes, L.; Luckham, D.; Maling, K.; Park, D.; Russell, S. (March 1969). "LISP I Programmers Manual" (https://web.archive.org/web/20100717111134/http://history.siam.org/sup/Fox_1960_LISP.pdf) (PDF). *Computation Center and Research Laboratory of Electronics*. Boston, Massachusetts: Artificial Intelligence Group, M.I.T. Computation Center and Research Laboratory: 88f. Archived from the original (http://history.siam.org/sup/Fox_1960_LISP.pdf) (PDF) on 17 July 2010. "In the local M.I.T. patois, association lists [of atomic symbols] are also referred to as "property lists", and atomic symbols are sometimes called "objects"."
6. McCarthy, John; Abrahams, Paul W.; Edwards, Daniel J.; Hart, swapnil d.; Levin, Michael I. (1962). *LISP 1.5 Programmer's Manual* (https://archive.org/details/lisp15programmer00john/page/105). MIT Press. p. 105 (https://archive.org/details/lisp15programmer00john/page/105). ISBN 978-0-262-13011-0. "Object — a synonym for atomic symbol"

7. Ivan E. Sutherland (May 1963). *Sketchpad: a man-machine graphical communication system*. AFIPS '63 (Spring): Proceedings of the May 21–23, 1963 Spring Joint Computer Conference. AFIPS Press. pp. 329–346. doi:10.1145/1461551.1461591 (https://doi.org/10.1145%2F1461551.1461591).
8. Kristen Nygaard; Ole-Johan Dahl (1 August 1978). "The development of the SIMULA languages" (https://doi.org/10.1145%2F960118.808391). *ACM SIGPLAN Notices*. **13** (8): 245–272. doi:10.1145/960118.808391 (https://doi.org/10.1145%2F960118.808391).
9. Ross, Doug. "The first software engineering language" (http://www.csail.mit.edu/timeline/timeline.php?query=event&id=19). *LCS/AI Lab Timeline*. MIT Computer Science and Artificial Intelligence Laboratory. Retrieved 13 May 2010.
10. Holmevik, Jan Rune (Winter 1994). "Compiling Simula: A historical study of technological genesis" (https://web.archive.org/web/20170830065454/http://www.idi.ntnu.no/grupper/su/publ/simula/holmevik-simula-ieeeannals94.pdf) (PDF). *IEEE Annals of the History of Computing*. **16** (4): 25–37. doi:10.1109/85.329756 (https://doi.org/10.1109%2F85.329756). S2CID 18148999 (https://api.semanticscholar.org/CorpusID:18148999). Archived from the original (http://www.idi.ntnu.no/grupper/su/publ/simula/holmevik-simula-ieeeannals94.pdf) (PDF) on 30 August 2017. Retrieved 3 March 2018.
11. Madsen, Ole Lehrman. "Kristen Nygaard" (https://amturing.acm.org/award_winners/nygaard_5916220.cfm). *A.M. Turing Award Laureates*. Retrieved 4 February 2025.
12. Butcher, Paul (30 June 2014). *Seven Concurrency Models in Seven Weeks: When Threads Unravel* (https://books.google.com/books?id=Xg9QDwAAQBAJ&pg=PT204). Pragmatic Bookshelf. p. 204. ISBN 978-1-68050-466-8.
13. Jones, Anita K.; Liskov, Barbara H. (April 1976). *An Access Control Facility for Programming Languages* (http://csg.csail.mit.edu/CSGArchives/memos/Memo-137.pdf) (PDF) (Technical report). MIT. CSG Memo 137.
14. Bertrand Meyer (2009). *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer Science & Business Media. p. 329. Bibcode:2009tclp.book.....M (https://ui.adsabs.harvard.edu/abs/2009tclp.book.....M). ISBN 978-3-540-92144-8.
15. Alan C. Kay (March 1993). "The early history of Smalltalk" (https://doi.org/10.1145%2F155360.155364). *ACM SIGPLAN Notices*. **28** (3): 69–95. doi:10.1145/155360.155364 (https://doi.org/10.1145%2F155360.155364).

16. Borning, Alan Hamilton (1979). Thinglab--a constraint-oriented simulation laboratory (https://constraints.cs.washington.edu/ui/thinglab-tr.pdf) (PDF) (Report). Stanford University.
17. Moon, David A. (June 1986). "Object-Oriented Programming with Flavors" (https://www.cs.tufts.edu/comp/150FP/archive/david-moon/flavors.pdf) (PDF). *Conference proceedings on Object-oriented Programming Systems Languages and Applications*. OOPSLA '86. pp. 1–8. doi:10.1145/28697.28698 (https://doi.org/10.1145%2F28697.28698). ISBN 978-0-89791-204-4. S2CID 17150741 (https://api.semanticscholar.org/CorpusID:17150741). Retrieved 17 March 2022.
18. "Introducing the Smalltalk Zoo" (https://computerhistory.org/blog/introducing-the-smalltalk-zoo-48-years-of-smalltalk-history-at-chm/). *CHM*. 17 December 2020.
19. Bobrow, D. G.; Stefik, M. J (1982). *LOOPS: data and object oriented Programming for Interlisp* (https://www.markstefik.com/wp-content/uploads/2011/04/1982-Bobrow-Stefik-Data-Object-Pgming.pdf) (PDF). European AI Conference.
20. Meyer 1997.
21. 1995 (June) Visual FoxPro 3.0, FoxPro evolves from a procedural language to an object-oriented language. Visual FoxPro 3.0 introduces a database container, seamless client/server capabilities, support for ActiveX technologies, and OLE Automation and null support. Summary of Fox releases (http://www.foxprohistory.org/foxprotimeline.htm#summary_of_fox_releases)
22. 1995 Reviewers Guide to Visual FoxPro 3.0: DFpug.de (http://www.dfpug.de/loseblattsammlung/migration/whitepapers/vfp_rg.htm)
23. Khurana, Rohit (1 November 2009). *Object Oriented Programming with C++, 1E* (https://books.google.com/books?id=MHmqfSBTXsAC&pg=PA16). Vikas Publishing House Pvt Limited. ISBN 978-81-259-2532-3.
24. Deborah J. Armstrong. *The Quarks of Object-Oriented Development*. A survey of nearly 40 years of computing literature identified several fundamental concepts found in the large majority of definitions of OOP, in descending order of popularity: Inheritance, Object, Class, Encapsulation, Method, Message Passing, Polymorphism, and Abstraction.
25. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, ISBN 0-521-78098-5, p.278. Lists: Dynamic dispatch, abstraction, subtype polymorphism, and inheritance.

26. Michael Lee Scott, *Programming language pragmatics*, Edition 2, Morgan Kaufmann, 2006, ISBN 0-12-633951-1, p. 470. Lists encapsulation, inheritance, and dynamic dispatch.
27. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press. ISBN 978-0-262-16209-8., section 18.1 "What is Object-Oriented Programming?" Lists: Dynamic dispatch, encapsulation or multi-methods (multiple dispatch), subtype polymorphism, inheritance or delegation, open recursion ("this"/"self")
28. C. J. Date, Introduction to Database Systems, 6th-ed., Page 650
29. Ambler, Scott (1 January 1998). "A Realistic Look at Object-Oriented Reuse" (http://www.drdobbs.com/184415594). drdobbs.com. Retrieved 4 July 2010.
30. Shelly, Asaf (22 August 2008). "Flaws of Object Oriented Modeling" (http://software.intel.com/en-us/blogs/2008/08/22/flaws-of-object-oriented-modeling/). Intel Software Network. Retrieved 4 July 2010.
31. James, Justin (1 October 2007). "Multithreading is a verb not a noun" (https://web.archive.org/web/20071010105117/http://blogs.techrepublic.com.com/programming-and-development/?p=518). techrepublic.com. Archived from the original (http://blogs.techrepublic.com.com/programming-and-development/?p=518) on 10 October 2007. Retrieved 4 July 2010.
32. Shelly, Asaf (22 August 2008). "HOW TO: Multicore Programming (Multiprocessing) Visual C++ Class Design Guidelines, Member Functions" (http://support.microsoft.com/?scid=kb%3Ben-us%3B558117). support.microsoft.com. Retrieved 4 July 2010.
33. Robert Harper (17 April 2011). "Some thoughts on teaching FP" (http://existentialtype.wordpress.com/2011/04/17/some-advice-on-teaching-fp/). Existential Type Blog. Retrieved 5 December 2011.
34. Stepanov, Alexander. "STLport: An Interview with A. Stepanov" (http://www.stlport.org/resources/StepanovUSA.html). Retrieved 21 April 2010.
35. Rich Hickey, JVM Languages Summit 2009 keynote, Are We There Yet? (http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey) November 2009.
36. Pike, Rob (25 June 2012). "Less is exponentially more" (https://commandcenter.blogspot.com/2012/06/less-is-exponentially-more.html). Retrieved 1 October 2016.
37. Pike, Rob (2 March 2004). "[9fans] Re: Threads: Sewing badges of honor onto a Kernel" (https://groups.google.com/group/comp.os.plan9/msg/006fec195aeeff15). *comp.os.plan9* (Mailing list). Retrieved 17 November 2016.

38. Booch, Grady (1986). *Software Engineering with Ada* (https://en.wikiquote.org/wiki/Grady_Booch). Addison Wesley. p. 220. ISBN 978-0-8053-0608-8. "Perhaps the greatest strength of an object-oriented approach to development is that it offers a mechanism that captures a model of the real world."
39. Niklaus Wirth (23 January 2006). "Good ideas, through the looking glass" (https://web.archive.org/web/20161012215755/https://pdfs.semanticscholar.org/10bd/dc49b85196aaa6715dd46843d9dcffa38358.pdf) (PDF). *IEEE Computer*. Cover Feature. **39** (1): 28–39. doi:10.1109/MC.2006.20 (https://doi.org/10.1109%2FMC.2006.20). S2CID 6582369 (https://api.semanticscholar.org/CorpusID:6582369). Archived from the original (https://pdfs.semanticscholar.org/10bd/dc49b85196aaa6715dd46843d9dcffa38358.pdf) (PDF) on 12 October 2016.
40. "Uncle Bob SOLID principles" (https://www.youtube.com/watch?v=zHiWqnTWsn4). *YouTube*. 2 August 2018.
41. Meyer 1997, p. 230.
42. Yegge, Steve (30 March 2006). "Execution in the Kingdom of Nouns" (http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html). steve-yegge.blogspot.com. Retrieved 3 July 2010.
43. Boronczyk, Timothy (11 June 2009). "What's Wrong with OOP" (http://zaemis.blogspot.com/2009/06/whats-wrong-with-oop.html). zaemis.blogspot.com. Retrieved 3 July 2010.
44. Bloch 2018, p. 19, Chapter §2 Item 4 Enforce noninstantiability with a private constructor.
45. Dony, C; Malenfant, J; Bardon, D (1999). "Classifying prototype-based programming languages" (https://www.lirmm.fr/~dony/postscript/proto-book.pdf) (PDF). *Prototype-based programming: concepts, languages and applications*. Singapore Berlin Heidelberg: Springer. ISBN 9789814021258.
46. "Is Go an object-oriented language?" (https://golang.org/doc/faq#Is_Go_an_object-oriented_language). Retrieved 13 April 2019. "Although Go has types and methods and allows an object-oriented style of programming, there is no type hierarchy."
47. Stroustrup, Bjarne (2015). *Object-Oriented Programming without Inheritance (Invited Talk)* (https://www.youtube.com/watch?v=xcpSLRpOMJM). 29th European Conference on Object-Oriented Programming (ECOOP 2015). 1:34. doi:10.4230/LIPIcs.ECOOP.2015.1 (https://doi.org/10.4230%2FLIPIcs.ECOOP.2015.1).

48. Pike, Rob (14 November 2012). "A few years ago I saw this page" (https://web.archive.org/web/20180814173134/http://plus.google.com/+RobPikeTheHuman/posts/hoJdanihKwb). Archived from the original (https://plus.google.com/+RobPikeTheHuman/posts/hoJdanihKwb) on 14 August 2018. Retrieved 1 October 2016.
49. McDonough, James E. (2017). "Encapsulation". *Object-Oriented Design with ABAP: A Practical Approach*. Apress. doi:10.1007/978-1-4842-2838-8 (https://doi.org/10.1007%2F978-1-4842-2838-8). ISBN 978-1-4842-2837-1 – via O'Reilly.
50. Bloch 2018, pp. 73–77, Chapter §4 Item15 Minimize the accessibility of classes and members.
51. "What is Object Oriented Programming (OOP) In Simple Words? – Software Geek Bytes" (https://softwaregeekbytes.com/object-oriented-programming-simple-words/). 5 January 2023. Retrieved 17 January 2023.
52. Cardelli, Luca; Wegner, Peter (10 December 1985). "On understanding types, data abstraction, and polymorphism" (https://doi.org/10.1145%2F6041.6042). *ACM Computing Surveys*. **17** (4): 471–523. doi:10.1145/6041.6042 (https://doi.org/10.1145%2F6041.6042). ISSN 0360-0300 (https://search.worldcat.org/issn/0360-0300).
53. Jacobsen, Ivar; Magnus Christerson; Patrik Jonsson; Gunnar Overgaard (1992). *Object Oriented Software Engineering* (https://archive.org/details/objectorientedso00jaco/page/43). Addison-Wesley ACM Press. pp. 43–69 (https://archive.org/details/objectorientedso00jaco/page/43). ISBN 978-0-201-54435-0.
54. Cardelli, Luca (1996). "Bad Engineering Properties of Object-Oriented Languages" (http://lucacardelli.name/Papers/BadPropertiesOfOO.html). *ACM Comput. Surv*. **28** (4es): 150–es. doi:10.1145/242224.242415 (https://doi.org/10.1145%2F242224.242415). ISSN 0360-0300 (https://search.worldcat.org/issn/0360-0300). S2CID 12105785 (https://api.semanticscholar.org/CorpusID:12105785). Retrieved 21 April 2010.
55. Armstrong, Joe. In *Coders at Work: Reflections on the Craft of Programming.* Peter Seibel, ed. Codersatwork.com (http://www.codersatwork.com/) Archived (https://web.archive.org/web/20100305165150/http://www.codersatwork.com/) 5 March 2010 at the Wayback Machine, Accessed 13 November 2009.
56. Eric S. Raymond (2003). "The Art of Unix Programming: Unix and Object-Oriented Languages" (http://www.catb.org/esr/writings/taoup/html/unix_and_oo.html). Retrieved 6 August 2014.

57. Brodie, Leo (1984). *Thinking Forth* (http://thinking-forth.sourceforge.net/thinking-forth-ans.pdf) (PDF). pp. 92–93. Retrieved 4 May 2018.
58. Hunt, Andrew. "Don't Repeat Yourself" (http://wiki.c2.com/?DontRepeatYourself). *Category Extreme Programming*. Retrieved 4 May 2018.
59. Cardelli, Luca; Wegner, Peter (December 1985). "On understanding types, data abstraction, and polymorphism" (http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf) (PDF). *ACM Computing Surveys*. **17** (4): 471–523. CiteSeerX 10.1.1.117.695 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.695). doi:10.1145/6041.6042 (https://doi.org/10.1145%2F6041.6042). S2CID 2921816 (https://api.semanticscholar.org/CorpusID:2921816).: "Polymorphic types are types whose operations are applicable to values of more than one type."
60. Stroustrup, Bjarne (19 February 2007). "Bjarne Stroustrup's C++ Glossary" (http://www.stroustrup.com/glossary.html#Gpolymorphism). "polymorphism – providing a single interface to entities of different types."
61. "The Emerald Programming Language" (http://www.emeraldprogramminglanguage.org/). 26 February 2011.
62. Brucker, Achim D.; Wolff, Burkhart (2008). "Extensible Universes for Object-Oriented Data Models". *ECOOP 2008 – Object-Oriented Programming*. Lecture Notes in Computer Science. Vol. 5142. pp. 438–462. doi:10.1007/978-3-540-70592-5_19 (https://doi.org/10.1007%2F978-3-540-70592-5_19). ISBN 978-3-540-70591-8. "object-oriented programming is a widely accepted programming paradigm"
63. Cassel, David (21 August 2019). "Why Are So Many Developers Hating on Object-Oriented Programming?" (https://thenewstack.io/why-are-so-many-developers-hating-on-object-oriented-programming/). *The New Stack*.
64. Potok, Thomas; Mladen Vouk; Andy Rindos (1999). "Productivity Analysis of Object-Oriented Software Developed in a Commercial Environment" (http://www.csm.ornl.gov/~v8q/Homepage/Papers%20Old/spetep-%20printable.pdf) (PDF). *Software: Practice and Experience*. **29** (10): 833–847. doi:10.1002/(SICI)1097-024X(199908)29:10<833::AID-SPE258>3.0.CO;2-P (https://doi.org/10.1002%2F%28SICI%291097-024X%28199908%2929%3A10%3C833%3A%3AAID-SPE258%3E3.0.CO%3B2-P). S2CID 57865731 (https://api.semanticscholar.org/CorpusID:57865731). Retrieved 21 April 2010.
65. Graham, Paul. "Why ARC isn't especially Object-Oriented" (http://www.paulgraham.com/noop.html). PaulGraham.com. Retrieved 13 November 2009.

66. Feldman, Richard (30 September 2019). "Why Isn't Functional Programming the Norm?" (https://www.youtube.com/watch?v=QyJZzq0v7Z4&t=2069s). *YouTube*.
67. Krubner, Lawrence. "Object Oriented Programming is an expensive disaster which must end" (https://web.archive.org/web/20141014233854/http://www.smashcompany.com/technology/object-oriented-programming-is-an-expensive-disaster-which-must-end). smashcompany.com. Archived from the original (http://www.smashcompany.com/technology/object-oriented-programming-is-an-expensive-disaster-which-must-end) on 14 October 2014. Retrieved 14 October 2014.
68. Martin, Robert C. "Design Principles and Design Patterns" (https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf) (PDF). Archived from the original (http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf) (PDF) on 6 September 2015. Retrieved 28 April 2017.
69. Neward, Ted (26 June 2006). "The Vietnam of Computer Science" (https://web.archive.org/web/20060704030226/http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx). Interoperability Happens. Archived from the original (http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx) on 4 July 2006. Retrieved 2 June 2010.
70. C. J. Date, Hugh Darwen. *Foundation for Future Database Systems: The Third Manifesto* (2nd Edition)
71. Wirfs-Brock, Rebecca; Wilkerson, Brian (1989). "Object-Oriented Design: A Responsibility-Driven Approach" (https://doi.org/10.1145%2F74878.74885). *ACM SIGPLAN Notices*. **24** (10): 74. doi:10.1145/74878.74885 (https://doi.org/10.1145%2F74878.74885).
72. Karsh, Patrick (19 July 2023). "GRASP Principles: Object-Oriented Design Patterns" (https://patrickkarsh.medium.com/object-oriented-design-with-grasp-principles-8049fa63e52). *Medium*. Retrieved 30 March 2025.
73. Poll, Erik. "Subtyping and Inheritance for Categorical Datatypes" (https://www.cs.ru.nl/E.Poll/papers/kyoto97.pdf) (PDF). Retrieved 5 June 2011.
74. Abadi, Martin; Cardelli, Luca (1996). *A Theory of Objects* (http://portal.acm.org/citation.cfm?id=547964&dl=ACM&coll=portal). Springer-Verlag New York, Inc. ISBN 978-0-387-94775-4. Retrieved 21 April 2010.
75. Tan, Tian; Li, Yue (12 July 2023). *Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics*. ISSTA 2023. pp. 1093–1105. doi:10.1145/3597926.3598120 (https://doi.org/10.1145%2F3597926.3598120).

76. Bhutani, Vikram; Toosi, Farshad Ghassemi; Buckley, Jim (1 June 2024). "Analysing the Analysers: An Investigation of Source Code Analysis Tools". *Applied Computer Systems*. **29** (1): 98–111. doi:10.2478/acss-2024-0013 (https://doi.org/10.2478%2Facss-2024-0013).

Further reading

- Abadi, Martin; Luca Cardelli (1998). *A Theory of Objects*. Springer Verlag. ISBN 978-0-387-94775-4.
- Abelson, Harold; Gerald Jay Sussman (1997). *Structure and Interpretation of Computer Programs* (https://web.archive.org/web/20171226134539/http://mitpress.mit.edu/sicp/). MIT Press. ISBN 978-0-262-01153-2. Archived from the original (http://mitpress.mit.edu/sicp/) on 26 December 2017. Retrieved 22 January 2006.
- Armstrong, Deborah J. (February 2006). "The Quarks of Object-Oriented Development". *Communications of the ACM*. **49** (2): 123–128. doi:10.1145/1113034.1113040 (https://doi.org/10.1145%2F1113034.1113040). ISSN 0001-0782 (https://search.worldcat.org/issn/0001-0782). S2CID 11485502 (https://api.semanticscholar.org/CorpusID:11485502).
- Bloch, Joshua (2018). *"Effective Java: Programming Language Guide"* (third ed.). Addison-Wesley. ISBN 978-0134685991.
- Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications* (https://archive.org/details/objectorientedan00booc). Addison-Wesley. ISBN 978-0-8053-5340-2.
- Eeles, Peter; Oliver Sims (1998). *Building Business Objects* (https://archive.org/details/buildingbusiness0000eele). John Wiley & Sons. ISBN 978-0-471-19176-6.
- Gamma, Erich; Richard Helm; Ralph Johnson; John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software* (https://archive.org/details/designpatternsel00gamm). Addison-Wesley. Bibcode:1995dper.book.....G (https://ui.adsabs.harvard.edu/abs/1995dper.book.....G). ISBN 978-0-201-63361-0.
- Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook – Lessons from Award-Winning Business Applications* (https://archive.org/details/objecttechnology00harm). John Wiley & Sons. ISBN 978-0-471-14717-6.
- Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley. Bibcode:1992oose.book.....J (htt

ps://ui.adsabs.harvard.edu/abs/1992oose.book.....J). ISBN 978-0-201-54435-0.
- Kay, Alan. *The Early History of Smalltalk* (https://web.archive.org/web/20050404075821/http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html). Archived from the original (http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html) on 4 April 2005. Retrieved 18 April 2005.
- Meyer, Bertrand (1997). *Object-Oriented Software Construction* (https://bertrandmeyer.com/OOSC2/). Prentice Hall. ISBN 978-0-13-629155-8.
- Pecinovsky, Rudolf (2013). *OOP – Learn Object Oriented Thinking & Programming* (http://pub.bruckner.cz/titles/oop). Bruckner Publishing. ISBN 978-80-904661-8-0.
- Rumbaugh, James; Michael Blaha; William Premerlani; Frederick Eddy; William Lorensen (1991). *Object-Oriented Modeling and Design* (https://archive.org/details/objectorientedmo00rumb). Prentice Hall. ISBN 978-0-13-629841-0.
- Schach, Stephen (2006). *Object-Oriented and Classical Software Engineering, Seventh Edition*. McGraw-Hill. ISBN 978-0-07-319126-3.
- Schreiner, Axel-Tobias (1993). *Object oriented programming with ANSI-C*. Hanser. hdl:1850/8544 (https://hdl.handle.net/1850%2F8544). ISBN 978-3-446-17426-9.
- Taylor, David A. (1992). *Object-Oriented Information Systems – Planning and Implementation* (https://archive.org/details/objectorientedin00tayl). John Wiley & Sons. ISBN 978-0-471-54364-0.
- Weisfeld, Matt (2009). *The Object-Oriented Thought Process, Third Edition*. Addison-Wesley. ISBN 978-0-672-33016-2.
- West, David (2004). *Object Thinking (Developer Reference)*. Microsoft Press. ISBN 978-0-7356-1965-4.

External links

- Introduction to Object Oriented Programming Concepts (OOP) and More (http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep) by L.W.C. Nirosh
- Discussion on Cons of OOP (https://thenewstack.io/why-are-so-many-developers-hating-on-object-oriented-programming/)
- OOP Concepts (Java Tutorials) (http://java.sun.com/docs/books/tutorial/java/concepts/index.html)