**OBJECT-ORIENTED PROGRAMMING**

**Object-oriented programming** (**OOP**) is a [programming paradigm](http://en.wikipedia.org/wiki/Programming_paradigm) using "[objects](http://en.wikipedia.org/wiki/Object_(computer_science))" – [data structures](http://en.wikipedia.org/wiki/Data_structure) consisting of [data fields](http://en.wikipedia.org/wiki/Field_(computer_science)) and [methods](http://en.wikipedia.org/wiki/Method_(computer_science)" \o "Method (computer science))together with their interactions – to design applications and computer programs. Programming techniques may include features such as [data abstraction](http://en.wikipedia.org/wiki/Data_abstraction), [encapsulation](http://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming)), [messaging](http://en.wikipedia.org/wiki/Message_passing), [modularity](http://en.wikipedia.org/wiki/Module_(programming)), [polymorphism](http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming), and [inheritance](http://en.wikipedia.org/wiki/Inheritance_(computer_science)). Many modern [programming languages](http://en.wikipedia.org/wiki/Programming_language) now support OOP.

**Overview**

Many people first learn to program using a language that is not object-oriented. Simple, non-OOP programs may be one long list of commands. More complex programs will group lists of commands into [functions](http://en.wikipedia.org/wiki/Function_(programming)) or [subroutines](http://en.wikipedia.org/wiki/Subroutine) each of which might perform a particular task. With designs of this sort, it is common for the program's data to be accessible from any part of the program. As programs grow in size, allowing any function to modify any piece of data means that bugs can have wide-reaching effects.

By contrast, the object-oriented approach encourages the programmer to place data where it is not directly accessible by the rest of the program. Instead the data is accessed by calling specially written 'functions', commonly called [methods](http://en.wikipedia.org/wiki/Method_(computer_science)), which are either bundled in with the data or inherited from "class objects" and act as the intermediaries for retrieving or modifying those data. The programming construct that combines data with a set of methods for accessing and managing those data is called an object.

An object-oriented program will usually contain different types of objects, each type corresponding to a particular kind of complex data to be managed or perhaps to a real-world object or concept such as a bank account, a hockey player, or a bulldozer. A program might well contain multiple copies of each type of object, one for each of the real-world objects the program is dealing with. For instance, there could be one bank account object for each real-world account at a particular bank. Each copy of the bank account object would be alike in the methods it offers for manipulating or reading its data, but the data inside each object would differ reflecting the different history of each account.

Objects can be thought of as wrapping their data within a set of functions designed to ensure that the data are used appropriately, and to assist in that use. The object's methods will typically include checks and safeguards that are specific to the types of data the object contains. An object can also offer simple-to-use, standardized methods for performing particular operations on its data, while concealing the specifics of how those tasks are accomplished. In this way alterations can be made to the internal structure or methods of an object without requiring that the rest of the program be modified. This approach can also be used to offer standardized methods across different types of objects. As an example, several different types of objects might offer print methods. Each type of object might implement that print method in a different way, reflecting the different kinds of data each contains, but all the different print methods might be called in the same standardized manner from elsewhere in the program. These features become especially useful when more than one programmer is contributing code to a project or when the goal is to reuse code between projects.

Object-oriented programming has roots that can be traced to the 1960s. As hardware and software became increasingly complex, manageability often became a concern. Researchers studied ways to maintain software quality and developed object-oriented programming in part to address common problems by strongly emphasizing discrete, reusable units of programming logic[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]. The technology focuses on data rather than processes, with programs composed of self-sufficient modules ("classes"), each instance of which ("objects") contains all the information needed to manipulate its own data structure ("members"). This is in contrast to the existing [modular programming](http://en.wikipedia.org/wiki/Modular_programming) that had been dominant for many years that focused on the *function* of a module, rather than specifically the data, but equally provided for [code reuse](http://en.wikipedia.org/wiki/Code_reuse), and self-sufficient reusable units of programming logic, enabling [collaboration](http://en.wikipedia.org/wiki/Collaboration) through the use of linked modules ([subroutines](http://en.wikipedia.org/wiki/Subroutine)). This more conventional approach, which still persists, tends to consider data and behavior separately.

An object-oriented program may thus be viewed as a collection of interacting *objects*, as opposed to the conventional model, in which a program is seen as a list of tasks ([subroutines](http://en.wikipedia.org/wiki/Subroutine)) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent "machine" with a distinct role or responsibility. The actions (or "[methods](http://en.wikipedia.org/wiki/Method_(computer_science))") on these objects are closely associated with the object. For example, OOP [data structures](http://en.wikipedia.org/wiki/Data_structures) tend to 'carry their own operators around with them' (or at least "[inherit](http://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming))" them from a similar object or class).

**History**

The terms "objects" and "oriented" in something like the modern sense of object-oriented programming seem to make their first appearance at [MIT](http://en.wikipedia.org/wiki/MIT) in the late 1950s and early 1960s. In the environment of the [artificial intelligence](http://en.wikipedia.org/wiki/Artificial_intelligence) group, as early as 1960, "object" could refer to identified items ([LISP](http://en.wikipedia.org/wiki/LISP) atoms) with properties (attributes);[[1]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-0)[[2]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-1) Alan Kay was later to cite a detailed understanding of LISP internals as a strong influence on his thinking in 1966.[[3]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-alanKayOnOO-2) Another early MIT example was [Sketchpad](http://en.wikipedia.org/wiki/Sketchpad) created by [Ivan Sutherland](http://en.wikipedia.org/wiki/Ivan_Sutherland) in 1960-61; in the glossary of the 1963 technical report based on his dissertation about Sketchpad, Sutherland defined notions of "object" and "instance" (with the class concept covered by "master" or "definition"), albeit specialized to graphical interaction.[[4]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-3) Also, an MIT [ALGOL](http://en.wikipedia.org/wiki/ALGOL) version, AED-0, linked data structures ("plexes", in that dialect) directly with procedures, prefiguring what were later termed "messages", "methods" and "member functions".[[5]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-simuladev-4)[[6]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-5)

Objects as a formal concept in programming were introduced in the 1960s in [Simula](http://en.wikipedia.org/wiki/Simula) 67, a major revision of Simula I, a programming language designed for [discrete event simulation](http://en.wikipedia.org/wiki/Discrete_event_simulation), created by [Ole-Johan Dahl](http://en.wikipedia.org/wiki/Ole-Johan_Dahl) and [Kristen Nygaard](http://en.wikipedia.org/wiki/Kristen_Nygaard) of the [Norwegian Computing Center](http://en.wikipedia.org/wiki/Norwegian_Computing_Center) in [Oslo](http://en.wikipedia.org/wiki/Oslo).[[7]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-6) Simula 67 was influenced by [SIMSCRIPT](http://en.wikipedia.org/wiki/SIMSCRIPT) and [Hoare's](http://en.wikipedia.org/wiki/Tony_Hoare) proposed "record classes".[[5]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-simuladev-4)[[8]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-7)Simula introduced the notion of classes and instances or objects (as well as subclasses, virtual methods, coroutines, and discrete event simulation) as part of an explicit programming paradigm. The language also used automatic [garbage collection](http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) that had been invented earlier for the [functional programming](http://en.wikipedia.org/wiki/Functional_programming) language [Lisp](http://en.wikipedia.org/wiki/Lisp_(programming_language)). Simula was used for physical modeling, such as models to study and improve the movement of ships and their content through cargo ports. The ideas of Simula 67 influenced many later languages, including Smalltalk, derivatives of LISP ([CLOS](http://en.wikipedia.org/wiki/CLOS)), [Object Pascal](http://en.wikipedia.org/wiki/Object_Pascal), and [C++](http://en.wikipedia.org/wiki/C%2B%2B).

The [Smalltalk](http://en.wikipedia.org/wiki/Smalltalk) language, which was developed at [Xerox PARC](http://en.wikipedia.org/wiki/Xerox_PARC) (by [Alan Kay](http://en.wikipedia.org/wiki/Alan_Kay) and others) in the 1970s, introduced the term *object-oriented programming* to represent the pervasive use of objects and messages as the basis for computation. Smalltalk creators were influenced by the ideas introduced in Simula 67, but Smalltalk was designed to be a fully dynamic system in which classes could be created and modified dynamically rather than statically as in Simula 67.[[9]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-st-8) Smalltalk and with it OOP were introduced to a wider audience by the August 1981 issue of [*Byte Magazine*](http://en.wikipedia.org/wiki/Byte_(magazine)).

In the 1970s, Kay's Smalltalk work had influenced the [Lisp community](http://en.wikipedia.org/wiki/Lisp_(programming_language)#Language_innovations) to incorporate [object-based techniques](http://en.wikipedia.org/wiki/Lisp_(programming_language)#Object_systems) that were introduced to developers via the [Lisp machine](http://en.wikipedia.org/wiki/Lisp_machine). Experimentation with various extensions to Lisp (like [LOOPS](http://en.wikipedia.org/w/index.php?title=LOOPS_(programming_language)&action=edit&redlink=1) and [Flavors](http://en.wikipedia.org/wiki/Flavors_(programming_language)) introducing [multiple inheritance](http://en.wikipedia.org/wiki/Multiple_inheritance) and [mixins](http://en.wikipedia.org/wiki/Mixins" \o "Mixins)), eventually led to the [Common Lisp Object System](http://en.wikipedia.org/wiki/Common_Lisp_Object_System) (CLOS, a part of the first standardized object-oriented programming language, [ANSI](http://en.wikipedia.org/wiki/ANSI) [Common Lisp](http://en.wikipedia.org/wiki/Common_Lisp)), which integrates functional programming and object-oriented programming and allows extension via a [Meta-object protocol](http://en.wikipedia.org/wiki/Meta-object_protocol). 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](http://en.wikipedia.org/wiki/Intel_iAPX_432) and the [Linn Smart](http://en.wikipedia.org/wiki/Linn_Products) [Rekursiv](http://en.wikipedia.org/wiki/Rekursiv).

Object-oriented programming developed as the dominant programming methodology in the early and mid 1990s when programming languages supporting the techniques became widely available. These included Visual FoxPro 3.0,[[10]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-9)[[11]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-10)[[12]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-11) [C++](http://en.wikipedia.org/wiki/C%2B%2B)[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)], and Delphi[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]. Its dominance was further enhanced by the rising popularity of [graphical user interfaces](http://en.wikipedia.org/wiki/Graphical_user_interface), which rely heavily upon object-oriented programming techniques. An example of a closely related dynamic GUI library and OOP language can be found in the [Cocoa](http://en.wikipedia.org/wiki/Cocoa_(software)" \o "Cocoa (software))frameworks on [Mac OS X](http://en.wikipedia.org/wiki/Mac_OS_X), written in [Objective-C](http://en.wikipedia.org/wiki/Objective-C), an object-oriented, dynamic messaging extension to C based on Smalltalk. OOP toolkits also enhanced the popularity of [event-driven programming](http://en.wikipedia.org/wiki/Event-driven_programming) (although this concept is not limited to OOP). Some[[*who?*](http://en.wikipedia.org/wiki/Wikipedia:Avoid_weasel_words)] feel that association with GUIs (real or perceived) was what propelled OOP into the programming mainstream.

At [ETH Zürich](http://en.wikipedia.org/wiki/ETH_Z%C3%BCrich), [Niklaus Wirth](http://en.wikipedia.org/wiki/Niklaus_Wirth) and his colleagues had also been investigating such topics as [data abstraction](http://en.wikipedia.org/wiki/Data_abstraction) and [modular programming](http://en.wikipedia.org/wiki/Modularity_(programming)) (although this had been in common use in the 1960s or earlier). [Modula-2](http://en.wikipedia.org/wiki/Modula-2) (1978) included both, and their succeeding design, [Oberon](http://en.wikipedia.org/wiki/Oberon_(programming_language)), included a distinctive approach to object orientation, classes, and such. The approach is unlike Smalltalk, and very unlike C++.

Object-oriented features have been added to many existing languages during that time, including [Ada](http://en.wikipedia.org/wiki/Ada_(programming_language)" \o "Ada (programming language)), [BASIC](http://en.wikipedia.org/wiki/BASIC), [Fortran](http://en.wikipedia.org/wiki/Fortran), [Pascal](http://en.wikipedia.org/wiki/Pascal_(programming_language)), and others. Adding these features to languages that were not initially designed for them often led to problems with compatibility and maintainability of code.

More recently, a number of languages have emerged that are primarily object-oriented yet compatible with procedural methodology, such as [Python](http://en.wikipedia.org/wiki/Python_(programming_language)) and [Ruby](http://en.wikipedia.org/wiki/Ruby_programming_language). Probably the most commercially important recent object-oriented languages are [Visual Basic.NET](http://en.wikipedia.org/wiki/Visual_Basic.NET) (VB.NET) and [C#](http://en.wikipedia.org/wiki/C_Sharp_(programming_language)), both designed for Microsoft's [.NET](http://en.wikipedia.org/wiki/.NET_Framework) platform, and [Java](http://en.wikipedia.org/wiki/Java_(programming_language)), developed by [Sun Microsystems](http://en.wikipedia.org/wiki/Sun_Microsystems). Both frameworks show the benefit of using OOP by creating an abstraction from implementation in their own way. VB.NET and C# support cross-language inheritance, allowing classes defined in one language to subclass classes defined in the other language. Java runs in a [virtual machine](http://en.wikipedia.org/wiki/Virtual_machine), making it possible to run on all different operating systems. VB.NET and C# make use of the Strategy pattern to accomplish cross-language inheritance, whereas Java makes use of the Adapter pattern

Just as [procedural programming](http://en.wikipedia.org/wiki/Procedural_programming) led to refinements of techniques such as [structured programming](http://en.wikipedia.org/wiki/Structured_programming), modern object-oriented software design methods include refinementssuch as the use of [design patterns](http://en.wikipedia.org/wiki/Design_pattern_(computer_science)), [design by contract](http://en.wikipedia.org/wiki/Design_by_contract), and [modeling languages](http://en.wikipedia.org/wiki/Modeling_language) (such as [UML](http://en.wikipedia.org/wiki/Unified_Modeling_Language)).

**Fundamental concepts and features**

A survey by Deborah J. Armstrong of nearly 40 years of computing literature identified a number of "quarks", or fundamental concepts, found in the strong majority of definitions of OOP.[[13]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-ArmstrongQuarks-12)

Not all of these concepts are to be found in all object-oriented programming languages. For example, object-oriented programming that uses classes is sometimes called [class-based programming](http://en.wikipedia.org/wiki/Class-based_programming), while [prototype-based programming](http://en.wikipedia.org/wiki/Prototype-based_programming) does not typically use classes. As a result, a significantly different yet analogous terminology is used to define the concepts of *object*and *instance*.

Benjamin C. Pierce and some other researchers view as futile any attempt to distill OOP to a minimal set of features. He nonetheless identifies fundamental features that support the OOP programming style in most object-oriented languages:[[14]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-13)

* [Dynamic dispatch](http://en.wikipedia.org/wiki/Dynamic_dispatch) – when a method is invoked on an object, the object itself determines what code gets executed by looking up the method at run time in a table associated with the object. This feature distinguishes an object from an [abstract data type](http://en.wikipedia.org/wiki/Abstract_data_type) (or module), which has a fixed (static) implementation of the operations for all instances. It is a programming methodology that gives modular component development while at the same time being very efficient.
* [Encapsulation](http://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming)) (or [multi-methods](http://en.wikipedia.org/wiki/Multi-methods), in which case the state is kept separate)
* [Subtype polymorphism](http://en.wikipedia.org/wiki/Subtype_polymorphism)
* Object [inheritance](http://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)" \o "Inheritance (object-oriented programming)) (or [delegation](http://en.wikipedia.org/wiki/Delegation_(programming)" \l "As_a_language_feature" \o "Delegation (programming)))
* [Open recursion](http://en.wikipedia.org/wiki/Open_recursion) – a special variable (syntactically it may be a keyword), usually called this or self, that allows a method body to invoke another method body of the same object. This variable is *late-bound*; it allows a method defined in one class to invoke another method that is defined later, in some subclass thereof.

Similarly, in his 2003 book, *Concepts in programming languages*, John C. Mitchell identifies four main features: dynamic dispatch, [abstraction](http://en.wikipedia.org/wiki/Abstraction_(computer_science)), subtype polymorphism, and inheritance.[[15]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-14) Michael Lee Scott in *Programming Language Pragmatics* considers only encapsulation, inheritance and dynamic dispatch.[[16]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-15)

Additional concepts used in object-oriented programming include:

* [Classes](http://en.wikipedia.org/wiki/Class_(computer_science)) of objects
* [Instances](http://en.wikipedia.org/wiki/Instance_(computer_science)) of classes
* [Methods](http://en.wikipedia.org/wiki/Method_(computer_science)) which act on the attached objects.
* [Message passing](http://en.wikipedia.org/wiki/Message_passing)
* [Abstraction](http://en.wikipedia.org/wiki/Abstraction_(computer_science))

**Decoupling**

Decoupling allows for the separation of object interactions from classes and inheritance into distinct layers of abstraction. A common use of decoupling is to polymorphically decouple the encapsulation, which is the practice of using reusable code to prevent discrete code modules from interacting with each other. However, in practice decoupling often involves trade-offs with regard to which patterns of change to favor. The science of measuring these trade-offs in respect to actual change in an objective way is still in its infancy.

**Formal semantics**

There have been several attempts at formalizing the concepts used in object-oriented programming. The following concepts and constructs have been used as interpretations of OOP concepts:

* [coalgebraic data types](http://en.wikipedia.org/wiki/F-Coalgebra)
* [abstract data types](http://en.wikipedia.org/wiki/Abstract_data_type) (which have [existential types](http://en.wikipedia.org/wiki/Existential_types)) allow the definition of [modules](http://en.wikipedia.org/wiki/Module_(programming)) but these do not support [dynamic dispatch](http://en.wikipedia.org/wiki/Dynamic_dispatch)
* [recursive types](http://en.wikipedia.org/wiki/Recursive_type)
* encapsulated state
* [Inheritance (object-oriented programming)](http://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming))
* [records](http://en.wikipedia.org/wiki/Record_(computer_science)) are basis for understanding objects if [function literals](http://en.wikipedia.org/wiki/Function_literal) can be stored in fields (like in functional programming languages), but the actual calculi need be considerably more complex to incorporate essential features of OOP. Several extensions of [System F<:](http://en.wikipedia.org/wiki/System_F-sub) that deal with mutable objects have been studied;[[17]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-AbadiCardelli-16) these allow both [subtype polymorphism](http://en.wikipedia.org/wiki/Subtype_polymorphism) and[parametric polymorphism](http://en.wikipedia.org/wiki/Parametric_polymorphism) (generics)

Attempts to find a consensus definition or theory behind objects have not proven very successful (however, see Abadi & Cardelli, [*A Theory of Objects*](http://portal.acm.org/citation.cfm?id=547964&dl=ACM&coll=portal)[[17]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-AbadiCardelli-16) for formal definitions of many OOP concepts and constructs), and often diverge widely. For example, some definitions focus on mental activities, and some on mere program structuring. One of the simpler definitions is that OOP is the act of using "map" data structures or arrays that can contain functions and pointers to other maps, all with some syntactic and scoping sugar on top. Inheritance can be performed by cloning the maps (sometimes called "prototyping"). OBJECT:=>> Objects are the run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

**OOP languages**

[Simula](http://en.wikipedia.org/wiki/Simula) (1967) is generally accepted as the first language to have the primary features of an object-oriented language. It was created for making [simulation programs](http://en.wikipedia.org/wiki/Computer_simulation), in which what came to be called objects were the most important information representation. [Smalltalk](http://en.wikipedia.org/wiki/Smalltalk) (1972 to 1980) is arguably the canonical example, and the one with which much of the theory of object-oriented programming was developed. Concerning the degree of object orientation, following distinction can be made:

* Languages called "pure" OO languages, because everything in them is treated consistently as an object, from primitives such as characters and punctuation, all the way up to whole classes, prototypes, blocks, modules, etc. They were designed specifically to facilitate, even enforce, OO methods. Examples: [Scala](http://en.wikipedia.org/wiki/Scala_(programming_language)" \o "Scala (programming language)), [Smalltalk](http://en.wikipedia.org/wiki/Smalltalk), [Eiffel](http://en.wikipedia.org/wiki/Eiffel_(programming_language)" \o "Eiffel (programming language)), [Ruby](http://en.wikipedia.org/wiki/Ruby_(programming_language)" \o "Ruby (programming language)), [JADE](http://en.wikipedia.org/wiki/JADE_(programming_language)), Emerald.[[18]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-17)
* Languages designed mainly for OO programming, but with some procedural elements. Examples: [C++](http://en.wikipedia.org/wiki/C%2B%2B), [C#](http://en.wikipedia.org/wiki/C_Sharp_(programming_language)), [VB.NET](http://en.wikipedia.org/wiki/VB.NET), [Java](http://en.wikipedia.org/wiki/Java_(programming_language)), [Python](http://en.wikipedia.org/wiki/Python_(programming_language)). (Note: [C# and VB.NET](http://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Visual_Basic_.NET) are both exclusively part of Microsoft's .NET Framework development platform and compile to the same intermediate language (IL). Although there are some construct differences, they are minimal and in the context of this grouping, some might consider them part of one language with simply two syntax translation engines).
* Languages that are historically [procedural languages](http://en.wikipedia.org/wiki/Procedural_programming), but have been extended with some OO features. Examples: [Visual Basic](http://en.wikipedia.org/wiki/Visual_Basic) (derived from BASIC), [Fortran 2003](http://en.wikipedia.org/wiki/Fortran_2003), [Perl](http://en.wikipedia.org/wiki/Perl), [COBOL](http://en.wikipedia.org/wiki/COBOL)2002, [PHP](http://en.wikipedia.org/wiki/PHP), [ABAP](http://en.wikipedia.org/wiki/ABAP).
* Languages with most of the features of objects (classes, methods, inheritance, reusability), but in a distinctly original form. Examples: [Oberon](http://en.wikipedia.org/wiki/Oberon_(programming_language)" \o "Oberon (programming language)) (Oberon-1 or Oberon-2).
* Languages with [abstract data type](http://en.wikipedia.org/wiki/Abstract_data_type) support, but not all features of object-orientation, sometimes called object-*based* languages. Examples: [Modula-2](http://en.wikipedia.org/wiki/Modula-2) (with excellent encapsulation and information hiding), [Pliant](http://en.wikipedia.org/w/index.php?title=Pliant&action=edit&redlink=1), [CLU](http://en.wikipedia.org/wiki/CLU_(programming_language)).

**Design patterns**

Challenges of object-oriented design are addressed by several methodologies. Most common is known as the [design patterns codified by Gamma *et al.*](http://en.wikipedia.org/wiki/Design_Patterns_(book)). More broadly, the term "[design patterns](http://en.wikipedia.org/wiki/Design_pattern_(computer_science))" can be used to refer to any general, repeatable solution to a commonly occurring problem in software design. Some of these commonly occurring problems have implications and solutions particular to object-oriented development.

**Inheritance and behavioral subtyping**

It is intuitive to assume that inheritance creates a [semantic](http://en.wikipedia.org/wiki/Program_semantics) "[is a](http://en.wikipedia.org/wiki/Is_a)" relationship, and thus to infer that objects instantiated from subclasses can always be *safely* used instead of those instantiated from the superclass. This intuition is unfortunately false in most OOP languages, in particular in all those that allow [mutable](http://en.wikipedia.org/wiki/Mutable) objects. [Subtype polymorphism](http://en.wikipedia.org/wiki/Subtype_polymorphism) as enforced by the [type checker](http://en.wikipedia.org/wiki/Type_checker) in OOP languages (with mutable objects) cannot guarantee [behavioral subtyping](http://en.wikipedia.org/wiki/Behavioral_subtyping) in any context. Behavioral subtyping is undecidable in general, so it cannot be implemented by a program (compiler). Class or object hierarchies need to be carefully designed considering possible incorrect uses that cannot be detected syntactically. This issue is known as the [Liskov substitution principle](http://en.wikipedia.org/wiki/Liskov_substitution_principle).

**Gang of Four design patterns**

[*Design Patterns: Elements of Reusable Object-Oriented Software*](http://en.wikipedia.org/wiki/Design_Patterns_(book)) is an influential book published in 1995 by [Erich Gamma](http://en.wikipedia.org/wiki/Erich_Gamma), [Richard Helm](http://en.wikipedia.org/w/index.php?title=Richard_Helm&action=edit&redlink=1), [Ralph Johnson](http://en.wikipedia.org/wiki/Ralph_Johnson_(computer_scientist)), and [John Vlissides](http://en.wikipedia.org/wiki/John_Vlissides), sometimes casually called the "Gang of Four". Along with exploring the capabilities and pitfalls of object-oriented programming, it describes 23 common programming problems and patterns for solving them. As of April 2007, the book was in its 36th printing.

The book describes the following patterns:

* [*Creational patterns*](http://en.wikipedia.org/wiki/Creational_pattern) (5): [Factory Method Pattern](http://en.wikipedia.org/wiki/Factory_method_pattern), [Abstract Factory Pattern](http://en.wikipedia.org/wiki/Abstract_Factory_Pattern), [Singleton Pattern](http://en.wikipedia.org/wiki/Singleton_pattern), [Builder Pattern](http://en.wikipedia.org/wiki/Builder_pattern), [Prototype Pattern](http://en.wikipedia.org/wiki/Prototype_pattern)
* [*Structural patterns*](http://en.wikipedia.org/wiki/Structural_pattern) (7): [Adapter Pattern](http://en.wikipedia.org/wiki/Adapter_Pattern), [Bridge Pattern](http://en.wikipedia.org/wiki/Bridge_Pattern), [Composite Pattern](http://en.wikipedia.org/wiki/Composite_Pattern), [Decorator Pattern](http://en.wikipedia.org/wiki/Decorator_Pattern), [Facade Pattern](http://en.wikipedia.org/wiki/Facade_Pattern), [Flyweight Pattern](http://en.wikipedia.org/wiki/Flyweight_Pattern), [Proxy Pattern](http://en.wikipedia.org/wiki/Proxy_pattern)
* [*Behavioral patterns*](http://en.wikipedia.org/wiki/Behavioral_pattern) (11): [Chain of Responsibility Pattern](http://en.wikipedia.org/wiki/Chain-of-responsibility_pattern), [Command Pattern](http://en.wikipedia.org/wiki/Command_Pattern), [Interpreter Pattern](http://en.wikipedia.org/wiki/Interpreter_Pattern), [Iterator Pattern](http://en.wikipedia.org/wiki/Iterator_Pattern" \o "Iterator Pattern), [Mediator Pattern](http://en.wikipedia.org/wiki/Mediator_Pattern), [Memento Pattern](http://en.wikipedia.org/wiki/Memento_Pattern), [Observer Pattern](http://en.wikipedia.org/wiki/Observer_Pattern), [State Pattern](http://en.wikipedia.org/wiki/State_pattern),[Strategy Pattern](http://en.wikipedia.org/wiki/Strategy_Pattern), [Template Method Pattern](http://en.wikipedia.org/wiki/Template_Method_Pattern), [Visitor Pattern](http://en.wikipedia.org/wiki/Visitor_Pattern)

**Matching real world**

OOP can be used to translate from real-world phenomena to program elements (and vice versa). OOP was even invented for the purpose of physical modeling in the [Simula](http://en.wikipedia.org/wiki/Simula) 67 language. However, not everyone agrees that direct real-world mapping is facilitated by OOP (see [Criticisms](http://en.wikipedia.org/wiki/Object-oriented_programming#Criticism) section), or is even a worthy goal; [Bertrand Meyer](http://en.wikipedia.org/wiki/Bertrand_Meyer) argues in [*Object-Oriented Software Construction*](http://en.wikipedia.org/wiki/Object-Oriented_Software_Construction)[[20]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-Meyer230-19) 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". At the same time, some principal limitations of OOP had been noted.[[21]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-20) An example for a real world problem that cannot be modeled elegantly with OOP techniques is the [Circle-ellipse problem](http://en.wikipedia.org/wiki/Circle-ellipse_problem).

However, [Niklaus Wirth](http://en.wikipedia.org/wiki/Niklaus_Wirth) (who popularized the adage now known as [Wirth's law](http://en.wikipedia.org/wiki/Wirth%27s_law): "Software is getting slower more rapidly than hardware becomes faster") said of OOP in his paper, "Good Ideas through the Looking Glass", "This paradigm closely reflects the structure of systems 'in the real world', and it is therefore well suited to model complex systems with complex behaviours" (contrast [KISS principle](http://en.wikipedia.org/wiki/KISS_principle)).

However, it was also noted (e.g. in [Steve Yegge](http://en.wikipedia.org/wiki/Steve_Yegge)'s essay *Execution in the Kingdom of Nouns*[[1]](http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html)) that the OOP approach of strictly prioritizing *things* (objects/[nouns](http://en.wikipedia.org/wiki/Noun)) before *actions*(methods/[verbs](http://en.wikipedia.org/wiki/Verb)) is a paradigm not found in natural languages.[[22]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-executioniKoN-21) This limitation may lead for some real world modelling to overcomplicated results compared e.g. to procedural approaches.[[23]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-executioniKoN2-22)

**OOP and control flow**

OOP was developed to increase the [reusability](http://en.wikipedia.org/wiki/Code_reuse) and [maintainability](http://en.wikipedia.org/wiki/Software_maintenance) of source code.[[24]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-realisticcodereuse-23) Transparent representation of the [control flow](http://en.wikipedia.org/wiki/Control_flow) had no priority and was meant to be handled by a compiler. With the increasing relevance of parallel hardware and [multithreaded coding](http://en.wikipedia.org/wiki/Thread_(computer_science)), developer transparent control flow becomes more important, something hard to achieve with OOP.[[25]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-flaws-24)[[26]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-multithreadingisaverb-25)[[27]](http://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-multicore-26)

**Responsibility vs. data-driven design**

[Responsibility-driven design](http://en.wikipedia.org/wiki/Responsibility-driven_design) defines classes in terms of a contract, that is, a class should be defined around a responsibility and the information that it shares. This is contrasted by Wirfs-Brock and Wilkerson with [data-driven design](http://en.wikipedia.org/wiki/Data-driven_design), where classes are defined around the data-structures that must be held. The authors hold that responsibility-driven design is preferable.

**The value of object orientation**

A large number of software engineers agree that gathering commands and data into discrete objects in this way makes their software easier to develop, document and maintain. However, a significant number of engineers feel the reverse may be true: that software becomes more complex to maintain and document, or even to engineer from the start. The conditions under which OOP prevails over alternative techniques (and vice-versa) often remain unstated by either party, however, making rational discussion of the topic difficult, and often leading to heated debates[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] over the matter.

A number of well-known researchers and programmers have analysed the utility of OOP.