Click Here

Computer science concept This article is about type systems in computer programming. For the formal study of type systems, see Type theory. This article includes a list of general references, but it lacks sufficient corresponding inline citations. Please help to improve this article by introducing more precise citations. (October 2010) (Learn how and when to remove this message) This article is written like a personal reflection, personal essay, or argumentative essay that states a Wikipedia editor's personal feelings or presents an original argument about a topic. Please help improve it by rewriting it in an encyclopedic style. (July 2016) (Learn how and when to remove this message) Type systems General concepts Type safety Strong vs. weak typing Major categories Static vs. dynamic Manifest vs. inferred Nominal vs. structural Duck typing Minor categories Abstract Dependent Flow-sensitive Gradual Intersection Latent Refinement Substructural Unique Session vte In computer programming, a type system is a logical system comprising a set of rules that assigns a property called a type (for example, integer, floating point, string) to every term (a word, phrase, or other set of symbols). Usually the terms are various language constructs of a computer program, such as variables, expressions, functions, or modules.[1] A type system dictates the operations that can be performed on a term. For variables, the type system determines the allowed values of that term. Type systems formalize and enforce the otherwise implicit categories the programmer uses for algebraic data types, data structures, or other data types, such as "string", "array of float", "function returning boolean". Type systems are often specified as part of programming languages and built into interpreters and compilers, although the type system of a language can be extended by optional tools that perform added checks using the language's original type syntax and grammar. The main purpose of a type system in a programming language is to reduce possibilities for bugs in computer programs due to type errors.[2] The given type system in question determines what constitutes a type error, but in general, the aim is to prevent operations expecting a certain kind of value from being used with values of which that operation does not make sense (validity errors). Type systems allow defining interfaces between different parts of a computer program, and then checking that the parts have been connected in a consistent way. This checking can happen statically (at compile time), dynamically (at run time), or as a combination of both. Type systems have other purposes as well, such as expressing business rules, enabling certain compiler optimizations, allowing for multiple dispatch, and providing a form of documentation. An example of a simple type system is that of the C language. The portions of a C program are the function definitions. One function is invoked by another function. The interface of a function states the name of the function and a list of parameters that are passed to the function's code. The code of an invoking function states the name of the invoked, along with the names of variables that hold values to pass to it. During a computer program's execution, the values are placed into temporary storage, then execution jumps to the code of the invoked function. The invoked function's code accesses the values and makes use of them. If the instructions inside the function are written with the assumption of receiving an integer value, but the calling code passed a floating-point value, then the wrong result will be computed by the invoked function. The C compiler checks the types of the arguments passed to a function when it is called against the types of the parameters declared in the function's definition. If the types do not match, the compiler throws a compile-time error or warning. A compiler may also use the static type of a value to optimize the storage it needs and the choice of algorithms for operations on the value. In many C compilers the float data type, for example, is represented in 32 bits, in accord with the IEEE specification for single-precision floating point numbers. They will thus use floating-point-specific microprocessor operations on those values (floating-point addition, multiplication, etc.). The depth of type constraints and the manner of their evaluation affect the typing of the language. A programming language may further associate an operation with various resolutions for each type, in the case of type polymorphism. Type theory is the study of type systems. The concrete types of some programming languages, such as integers and strings, depend on practical issues of computer architecture, compiler implementation, and language design. Formally, type theory studies type systems. A programming language must have the opportunity to type check using the type system whether at compile time or runtime, manually annotated or automatically inferred. As Mark Manasse concisely put it:[3] The fundamental problem addressed by a type theory is to ensure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension. Assigning a data type, termed typing, gives meaning to a sequence of bits such as a value in memory or some object such as a variable. The hardware of a general purpose computer is unable to discriminate between for example a memory address and an instruction code, or between a character, an integer, or a floating-point number, because it makes no intrinsic distinction between any of the possible values that a sequence of bits might mean.[note 1] Associating a sequence of bits with a type conveys that meaning to the programmable hardware to form a symbolic system composed of that hardware and some program. A program associates each value with at least one specific type, but it also can occur that one value is associated with many subtypes. Other entities, such as objects, modules, communication channels, and dependencies can become associated with a type. Even a type can become associated with a type. An implementation of a type system could in theory associate identifications called data type (a type of a value), class (a type of an object), and kind (a type of a type, or metatype). These are the abstractions that typing can go through, on a hierarchy of levels contained in a system. When a programming language evolves a more elaborate type system, it gains a more finely grained rule set than basic type checking, but this comes at a price when the type inferences (and other properties) become undecidable, and when more attention must be paid by the programmer to annotate code or to consider computer-related operations and functioning. It is challenging to find a sufficiently expressive type system that satisfies all programming practices in a type safe manner. A programming language compiler can also implement a dependent type or an effect system, which enables even more program specifications to be verified by a type checker. Beyond simple value-type pairs, a virtual "region" of code is associated with an "effect" component describing what is being done with what, and enabling for example to "throw" an error report. Thus the symbolic system may be a type and effect system, which endows it with more safety checking than type checking alone. Whether automated by the compiler or specified by a programmer, a type system renders program behavior illegal if it falls outside the type-system rules. Advantages provided by programmer-specified type systems include: Abstraction (or modularity) – Types enable programmers to think at a higher level than the bit or byte, not bothering with low-level implementation. For example, programmers can begin to think of a string as a set of character values instead of as an array of bytes. Higher still, types enable programmers to think about and express interfaces between two of any-sized subsystems. This enables more levels of localization so that the definitions required for interoperability of the subsystems remain consistent when those two subsystems communicate. Documentation – In more expressive type systems, types can serve as a form of documentation clarifying the intent of the programmer. For example, if a programmer declares a function as returning a timestamp type, this documents the function when the timestamp type can be explicitly declared deeper in the code to be an integer type. Advantages provided by compiler-specified type systems include: Optimization – Static type-checking may provide useful compile-time information. For example, if a type requires that a value must align in memory at a multiple of four bytes, the compiler may be able to use more efficient machine instructions. Safety – A type system enables the compiler to detect meaningless or invalid code. For example, we can identify an expression 3 / "Hello, World" as invalid, when the rules do not specify how to divide an integer by a string. Strong typing offers more safety, but cannot guarantee complete type safety. A type error occurs when an operation receives a different type of data than it expected.[4] For example, a type error would happen if a line of code divides two integers, and is passed a string of letters instead of an integer.[4] It is an unintended condition[note 2] which might manifest in multiple stages of a program's development. Thus a facility for detection of the error is needed in the type system. In some languages, such as Haskell, for which type inference is automated, lint might be available to its compiler to aid in the detection of error. Type safety contributes to program correctness, but might only guarantee correctness at the cost of making the type checking itself an undecidable problem (as in the halting problem). In a type system with automated type checking, a program may prove to run incorrectly yet produce no compiler errors. Division by zero is an unsafe and incorrect operation, but a type checker which only runs at compile time does not scan for division by zero in most languages; that division would surface as a runtime error. To prove the absence of these defects, other kinds of formal methods, collectively known as program analyses, are in common use. Alternatively, a sufficiently expressive type system, such as in dependently typed languages, can prevent these kinds of errors (for example, expressing the type of non-zero numbers). In addition, software testing is an empirical method for finding errors that such a type checker would not detect. The process of verifying and enforcing the constraints of types—type checking—may occur at compile time (a static check) or at run-time (a dynamic check). If a language specification requires its typing rules strongly, more or less allowing only those automatic type conversions that do not lose information, one can refer to the process as strongly typed; if not, as weakly typed. The terms are not usually used in a strict sense. See also: Category:Statically typed programming languages Static type checking is the process of verifying the type safety of a program based on analysis of a program's text (source code). If a program passes a static type checker, then the program is guaranteed to satisfy some set of type safety properties for all possible inputs. Static type checking can be considered a limited form of program verification (see type safety), and in a type-safe language, can also be considered an optimization. If a compiler can prove that a program is well-typed, then it does not need to emit dynamic safety checks, allowing the resulting compiled binary to run faster and to be smaller. Static type checking for Turing-complete languages is inherently conservative. That is, if a type system is both sound (meaning that it rejects all incorrect programs) and decidable (meaning that it is possible to write an algorithm that determines whether a program is well-typed), then it must be incomplete (meaning there are correct programs, which are also rejected, even though they do not encounter runtime errors).[7] For example, consider a program containing the code: if then else Even if the expression always evaluates to true at run-time, most type checkers will reject the program as ill-typed, because it is difficult (if not impossible) for a static analyzer to determine that the else branch will not be taken.[8] Consequently, a static type checker will quickly detect type errors in rarely used code paths. Without static type checking, even code coverage tests with 100% coverage may be unable to find such type errors. The tests may fail to detect such type errors, because the combination of all places where values are created and all places where a certain value is used must be taken into account. A number of useful and common programming language features cannot be checked statically, such as downcasting. Thus, many languages will have both static and dynamic type checking; the static type checker verifies what it can, and dynamic checks verify the rest. Many languages with static type checking provide a way to bypass the type checker. Some languages allow programmers to choose between static and dynamic type safety. For example, historically C# declares variables statically,[9]:77,Section 3.2 but C# 4.0 introduces the dynamic keyword, which is used to declare variables to be checked dynamically at runtime.[9]:117,Section 4.1 Other languages allow writing code that is not type-safe; for example, in C, programmers can freely cast a value between any two types that have the same size, effectively subverting the type concept. See also: Dynamic programming language, Interpreted language, and Category:Dynamically typed programming languages Dynamic type checking is the process of verifying the type safety of a program at runtime. Implementations of dynamically type-checked languages generally associate each runtime object with a type tag (i.e., a reference to a type) containing its type information. This runtime type information (RTTI) can also be used to implement dynamic dispatch, late binding, downcasting, reflective programming (reflection), and similar features. Most type-safe languages include some form of dynamic type checking, even if they also have a static type checker.[10] The reason for this is that many useful features or properties are difficult or impossible to verify statically. For example, suppose that a program defines two types, A and B, where B is a subtype of A. If the program tries to convert a value of type A to type B, which is known as downcasting, then the operation is legal only if the value being converted is actually a value of type B. Thus, a dynamic check is needed to verify that the operation is safe. This requirement is one of the criticisms of downcasting. By definition, dynamic type checking may cause a program to fail at runtime. In some programming languages, it is possible to anticipate and recover from these failures. In others, type-checking errors are considered fatal. Programming languages that include dynamic type checking but not static type checking are often called "dynamically typed programming languages". "Type hinting" redirects here. For hinting of typefaces, see font hinting. Certain languages allow both static and dynamic typing. For example, Java and some other ostensibly statically typed languages support downcasting types to their subtypes, querying an object to discover its dynamic type and other type operations that depend on runtime type information. Another example is C++ RTTI. More generally, most programming languages include mechanisms for dispatching over different 'kinds' of data, such as disjoint unions, runtime polymorphism, and variant types. Even when not interacting with type annotations or type checking, such mechanisms are materially similar to dynamic typing implementations. See programming language for more discussion of the interactions between static and dynamic typing. Objects in object-oriented languages are usually accessed by a reference whose static target type (or manifest type) is equal to either the object's run-time type (its latent type) or a supertype thereof. This is conformant with the Liskov substitution principle, which states that all operations performed on an instance of a given type can also be performed on an instance of a subtype. This concept is also known as subsumption or subtype polymorphism. In some languages subtypes may also possess covariant or contravariant return types and argument types respectively. Certain languages, for example Clojure, Common Lisp, or Cython are dynamically type checked by default, but allow programs to opt into static type checking by providing optional annotations. One reason to use such hints would be to optimize the performance of critical sections of a program. This is formalized by gradual typing. The programming environment DrRacket, a pedagogic environment based on Lisp, and a precursor of the language Racket is also soft-typed.[11] Conversely, as of version 4.0, the C# language provides a way to indicate that a variable should not be statically type checked. A variable whose type is dynamic will not be subject to static type checking. Instead, the program relies on runtime type information to determine how the variable may be used.[12][9]:113–119 In Rust, the dyn std::any::Any type provides dynamic typing of 'static types.[13] The choice between static and dynamic typing requires certain trade-offs. Static typing can find type errors reliably at compile time, which increases the reliability of the delivered program. However, programmers disagree over how commonly type errors occur, resulting in further disagreements over the proportion of those bugs that are coded that would be caught by appropriately representing the designed types in code.[14][15] Static typing advocates[who?] believe programs are more reliable when they have been well type-checked, whereas dynamic-typing advocates[who?] point to distributed code that has proven reliable and to small bug databases.[citation needed] The value of static typing increases as the strength of the type system is increased. Advocates of dependent typing,[who?] implemented in languages such as Dependent ML and Epigram, have suggested that almost all bugs can be considered type errors, if the types used in a program are properly declared by the programmer or correctly inferred by the compiler.[16] Static typing usually results in compiled code that executes faster. When the compiler knows the exact data types that are in use (which is necessary for static verification, either through declaration or inference) it can produce optimized machine code. Some dynamically typed languages such as Common Lisp allow optional type declarations for optimization for this reason. By contrast, dynamic typing may allow compilers to run faster and interpreters to dynamically load new code, because changes to source code in dynamically typed languages may result in less checking to perform and less code to revisit.[clarification needed] This too may reduce the edit-compile-test-debug cycle. Statically typed languages that lack type inference (such as C and Java prior to version 10) require that programmers declare the types that a method or function must use. This can serve as added program documentation, that is active and dynamic, instead of static. This allows a compiler to prevent it from drifting out of synchrony, and from being ignored by programmers. However, a language can be statically typed without requiring type declarations (examples include Haskell, Scala, OCaml, F#, Swift, and to a lesser extent C# and C++), so explicit type declaration is not a necessary requirement for static typing in all languages. Dynamic typing allows constructs that some (simple) static type checking would reject as illegal. For example, eval functions, which execute arbitrary data as code, become possible. An eval function is possible with static typing, but requires advanced uses of algebraic data types. Further, dynamic typing better accommodates transitional code and prototyping, such as allowing a placeholder data structure (mock object) to be transparently used in place of a full data structure (usually for the purposes of experimentation and testing). Dynamic typing typically allows duck typing (which enables easier code reuse). Many[specify] languages with static typing also feature duck typing or other mechanisms like generic programming that also enable easier code reuse. Dynamic typing typically makes metaprogramming easier to use. For example, C++ templates are typically more cumbersome to write than the equivalent Ruby or Python code since C++ has stronger rules regarding type definitions (for both functions and variables). This forces a developer to write more boilerplate code for a template than a Python developer would need to. More advanced run-time constructs such as metaclasses and introspection are often harder to use in statically typed languages. In some languages, such features may also be used e.g. to generate new types and behaviors on the fly, based on run-time data. Such advanced constructs are often provided by dynamic programming languages; many of these are dynamically typed, although dynamic typing need not be related to dynamic programming languages. Main article: Strong and weak typing Languages are often colloquially referred to as strongly typed or weakly typed. In fact, there is no universally accepted definition of what these terms mean. In general, there are more precise terms to represent the differences between type systems that lead people to call them "strong" or "weak". Main articles: Type safety and Memory safety A third way of categorizing the type system of a programming language is by the safety of typed operations and conversions. Computer scientists use the term type-safe language to describe languages that do not allow operations or conversions that violate the rules of the type system. Computer scientists use the term memory-safe language (or just safe language) to describe languages that do not allow programs to access memory that has not been assigned for their use. For example, a memory-safe language will check array bounds, or else statically guarantee (i.e., at compile time before execution) that array accesses out of the array boundaries will cause compile-time and perhaps runtime errors. Consider the following program of a language that is both type-safe and memory-safe:[17] var x := 5; var y := "37"; var z := x + y; In this example, the variable z will have the value 42. Although this may not be what the programmer anticipated, it is a well-defined result. If y were a different string, one that could not be converted to a number (e.g. "Hello World"), the result would be well-defined as well. Note that a program can be type-safe or memory-safe and still crash on an invalid operation. This is for languages where the type system is not sufficiently advanced to precisely specify the validity of operations on all possible operands. But if a program encounters an operation that is not type-safe, terminating the program is often the only option. Now consider a similar example in C: int x = 5; char y[] = "37"; char* z = x + y; printf("%c", *z); In this example z will point to a memory address five characters beyond y, equivalent to three characters after the terminating zero character of the string pointed to by y. This is memory that the program is not expected to access. In C terms this is simply undefined behaviour and the program may do anything; with a simple compiler it might actually print whatever byte is stored after the string "37". As this example shows, C is not memory-safe. As arbitrary data was assumed to be a character, it is also not a type-safe language. In general, type-safety and memory-safety go hand in hand. For example, a language that supports pointer arithmetic and number-to-pointer conversions (like C) is neither memory-safe nor type-safe, because it allows arbitrary memory to be accessed as if it were valid memory of any type. Some languages allow different levels of checking to apply to different regions of code. Examples include: The use strict directive in JavaScript[18][19][20] and Perl applies stronger checking. The declare(strict_types=1) in PHP[21] on a per-file basis allows only a variable of exact type of the type declaration will be accepted, or a TypeError will be thrown. The Option Strict On in VB.NET allows the compiler to require a conversion between objects. Additional tools such as lint and IBM Rational Purify can also be used to achieve a higher level of strictness. It has been proposed, chiefly by Gilad Bracha, that the choice of type system be made independent of choice of language; that a type system should be a module that can be plugged into a language as needed. He believes this is advantageous, because what he calls mandatory type systems make languages less expressive and code more fragile.[22] The requirement that the type system does not affect the semantics of the language is difficult to fulfill. Optional typing is related to, but distinct from, gradual typing. While both typing disciplines can be used to perform static analysis of code (static typing), optional type systems do not enforce type safety at runtime (dynamic typing).[22][23] Main article: Polymorphism (computer science) The term polymorphism refers to the ability of code (especially, functions or classes) to act on values of multiple types, or to the ability of different instances of the same data structure to contain elements of different types. Type systems that allow polymorphism generally do so in order to improve the potential for code re-use: in a language with polymorphism, programmers need only implement a data structure such as a list or an associative array once, rather than once for each type of element with which they plan to use it. For this reason computer scientists sometimes call the use of certain forms of polymorphism generic programming. The type-theoretic foundations of polymorphism are closely related to those of abstraction, modularity and (in some cases) subtyping. Many type systems have been created that are specialized for use in certain environments with certain types of data, or for out-of-band static program analysis. Frequently, these are based on ideas from formal type theory and are only available as part of prototype research systems. The following table gives an overview over type theoretic concepts that are used in specialized type systems. The names M, N, O range over terms and the names σ , τ {\displaystyle \sigma ,\tau } range over types. The following notation will be used: M : σ {\displaystyle M:\sigma } means that M {\displaystyle M} has type σ {\displaystyle \sigma } ; M (N) {\displaystyle M(N)} is that application of M {\displaystyle M} on N {\displaystyle N} ; τ [α := σ] {\displaystyle \tau [\alpha :=\sigma]} (resp. τ [x := N] {\displaystyle \tau [x:=N]}) describes the type which results from replacing all occurrences of the type variable α (resp. term variable x) in τ {\displaystyle \tau } by the type σ (resp. term N). Type notion Notation Meaning Function σ → τ {\displaystyle \sigma \to \tau } If M : σ → τ {\displaystyle M:\sigma \to \tau } and N : σ {\displaystyle N:\sigma } , then M (N) : τ {\displaystyle M(N):\tau } . Product σ × τ {\displaystyle \sigma \times \tau } If M : σ × τ {\displaystyle M:\sigma \times \tau } , then M = (N , O) {\displaystyle M=(N,O)} is a pair s.t. N : σ {\displaystyle N:\sigma } and O : τ {\displaystyle O:\tau } . Sum σ + τ {\displaystyle \sigma +\tau } If M : σ + τ {\displaystyle M:\sigma +\tau } , then M = ι 1 (N) {\displaystyle M=\iota _{1}(N)} is the first injection s.t. N : σ {\displaystyle N:\sigma } , or M = ι 2 (N) {\displaystyle M=\iota _{2}(N)} is the second injection s.t. N : τ {\displaystyle N:\tau } . Intersection σ ∩ τ {\displaystyle \sigma \cap \tau } If M : σ ∩ τ {\displaystyle M:\sigma \cap \tau } , then M : σ {\displaystyle M:\sigma } and M : τ {\displaystyle M:\tau } . Union σ ∪ τ {\displaystyle \sigma \cup \tau } If M : σ ∪ τ {\displaystyle M:\sigma \cup \tau } , then M : σ {\displaystyle M:\sigma } or M : τ {\displaystyle M:\tau } . Record ⟨ x : τ ⟩ {\displaystyle \langle x:\tau \rangle } If M : ⟨ x : τ ⟩ {\displaystyle M:\langle x:\tau \rangle } , then M has a member x : τ {\displaystyle x:\tau } . Polymorphic ∀ α . τ {\displaystyle \forall {}\alpha .\tau } If M : ∀ α . τ {\displaystyle M:\forall {}\alpha .\tau } , then M : τ [α := σ] {\displaystyle M:\tau [\alpha :=\sigma]} for any type σ. Existential ∃ α . τ {\displaystyle \exists {}\alpha .\tau } If M : ∃ α . τ {\displaystyle M:\exists {}\alpha .\tau } , then M : τ [α := σ] {\displaystyle M:\tau [\alpha :=\sigma]} for some type σ. Recursive μ α . τ {\displaystyle \mu \alpha .\tau } If M : μ α . τ {\displaystyle M:\mu \alpha .\tau } , then M : τ [α := μ α . τ] {\displaystyle M:\tau [\alpha :=\mu \alpha .\tau]} . Dependent function[a] (x : σ) → τ {\displaystyle (x:\sigma)\to \tau } If M : (x : σ) → τ {\displaystyle M:(x:\sigma)\to \tau } and N : σ {\displaystyle N:\sigma } , then M (N) : τ [x := N] {\displaystyle M(N):\tau [x:=N]} . Dependent pair[b] (x : σ) × τ {\displaystyle (x:\sigma)\times \tau } If M : (x : σ) × τ {\displaystyle M:(x:\sigma)\times \tau } , then M = (N , O) {\displaystyle M=(N,O)} is a pair s.t. N : σ {\displaystyle N:\sigma } and O : τ [x := N] {\displaystyle O:\tau [x:=N]} . Dependent intersection[24] (x : σ) ∩ τ {\displaystyle (x:\sigma)\cap \tau } If M : (x : σ) ∩ τ {\displaystyle M:(x:\sigma)\cap \tau } , then M : σ {\displaystyle M:\sigma } and M : τ [x := M] {\displaystyle M:\tau [x:=M]} . Familial intersection[24] ⋂ x : σ τ {\displaystyle \bigcap _{x:\sigma }\tau } If M : ⋂ x : σ τ {\textstyle M:\bigcap _{x:\sigma }\tau } , then M : τ [x := N] {\displaystyle M:\tau [x:=N]} for any term N : σ {\displaystyle N:\sigma } . Familial union[24] ⋃ x : σ τ {\displaystyle \bigcup _{x:\sigma }\tau } If M : ⋃ x : σ τ {\textstyle M:\bigcup _{x:\sigma }\tau } , then M : τ [x := N] {\displaystyle M:\tau [x:=N]} for some term N : σ {\displaystyle N:\sigma } . ^ Also referred to as dependent product type, since (x : σ) → τ = ∏ x : σ τ {\textstyle (x:\sigma)\to \tau =\prod _{x:\sigma }\tau } . ^ Also referred to as dependent sum type, since (x : σ) × τ = ∑ x : σ τ {\textstyle (x:\sigma)\times \tau =\sum _{x:\sigma }\tau } . Main article: Dependent type Dependent types are based on the idea of using scalars or values to more precisely describe the type of some other value. For example, m a t r i x (3 , 3) {\displaystyle \mathrm {matrix} (3,3)} might be the type of a 3 × 3 {\displaystyle 3\times 3} matrix. We can then define typing rules such as the following rule for matrix multiplication: m a t r i x m u l t i p l y : m a t r i x (k , m) × m a t r i x (m , n) → m a t r i x (k , n) {\displaystyle \mathrm {matrix} _{\mathrm {multiply} }:\mathrm {matrix} (k,m)\times \mathrm {matrix} (m,n)\to \mathrm {matrix} (k,n)} where k, m, n are arbitrary positive integer values. A variant of ML called Dependent ML has been created based on this type system, but because type checking for conventional dependent types is undecidable, not all programs using them can be type-checked without some kind of limits. Dependent ML limits the sort of equality it can decide to Presburger arithmetic. Other languages such as Epigram make the value of all expressions in the language decidable so that type checking can be decidable. However, in general proof of decidability is undecidable, so many programs require hand-written annotations that may be very non-trivial. As this impedes the development process, many language implementations provide an easy way out in the form of an option to disable this condition. This, however, comes at the cost of making the type-checker run in an infinite loop when fed programs that do not type-check, causing the compilation to fail. Main article: Linear type Linear types, based on the theory of linear logic, and closely related to uniqueness types, are types assigned to values having the property that they have one and only one reference to them at all times. These are valuable for describing large immutable values such as files, strings, and so on, because any operation that simultaneously destroys a linear object and creates a similar object (such as str = str + "a") can be optimized "under the hood" into an in-place mutation. Normally this is not possible, as such mutations could cause side effects on parts of the program holding other references to the object, violating referential transparency. They are also used in the prototype operating system Singularity for interprocess communication, statically ensuring that processes cannot share objects in shared memory in order to prevent race conditions. The Clean language (a Haskell-like language) uses this type system in order to gain a lot of speed (compared to performing a deep copy) while remaining safe. Main article: Intersection type Intersection types are types describing values that belong to both of two other given types with overlapping value sets. For example, in most implementations of C the signed char has range -128 to 127 and the unsigned char has range 0 to 255, so the intersection type of these two types would have range 0 to 127. Such an intersection type could be safely passed into functions expecting either signed or unsigned chars, because it is compatible with both types. Intersection types are useful for describing overloaded function types: for example, if "int → int" is the type of functions taking an integer argument and returning an integer, and "float → float" is the type of functions taking a float argument and returning a float, then the intersection of these two types can be used to describe functions that do one or the other, based on what type of input they are given. Such a function could be passed into another function expecting an "int → int" function safely; it simply would not use the "float → float" functionality. In a subclassing hierarchy, the intersection of a type and an ancestor type (such as its parent) is the most derived type. The intersection of sibling types is empty. The Forsythe language includes a general implementation of intersection types. A restricted form is refinement types. Main article: Union type Union types are types describing values that belong to either of two types. For example, in C, the signed char has a -128 to 127 range, and the unsigned char has a 0 to 255 range, so the union of these two types would have an overall "virtual" range of -128 to 255 that may be used partially depending on which union member is accessed. Any function handling this union type would have to deal with integers in this complete range. More generally, the only valid operations on a union type are operations that are valid on both types being unioned. C's "union" concept is similar to union types, but is not typesafe, as it permits operations that are valid on either type, rather than both. Union types are important in program analysis, where they are used to represent symbolic values whose exact nature (e.g., value or type) is not known. In a subclassing hierarchy, the union of a type and an ancestor type (such as its parent) is the ancestor type. The union of sibling types is a subtype of their common ancestor (that is, all operations permitted on their common ancestor are permitted on the union type, but they may also have other valid operations in common). Main article: Existential quantifier Existential types are frequently used in connection with record types to represent modules and abstract data types, due to their ability to separate implementation from interface. For example, the type "T = ∃X { a: X; f: (X → int); }" describes a module interface that has a data member named a of type X and a function named f that takes a parameter of the same type X and returns an integer. This could be implemented in different ways; for example: intT = { a: int; f: (int → int); } floatT = { a: float; f: (float → int); } These types are both subtypes of the more general existential type T and correspond to concrete implementation types, so any value of one of these types is a value of type T. Given a value "t" of type "T", we know that "t.f(t.a)" is well-typed, regardless of what the abstract type X is. This gives flexibility for choosing types suited to a particular implementation, while clients that use only values of the interface type—the existential type—are isolated from these choices. In general it's impossible for the typechecker to infer which existential type a given module belongs to. In the above example intT { a: int; f: (int → int); } could also have the type ∃X { a: X; f: (int → int); }. The simplest solution is to annotate every module with its intended type, e.g.: intT = { a: int; f: (int → int); } as ∃X { a: X; f: (X → int); } Although abstract data types and modules had been implemented in programming languages for quite some time, it wasn't until 1988 that John C. Mitchell and Gordon Plotkin established the formal theory under the slogan: "Abstract [data] types have existential type".[25] The theory is a second-order typed lambda calculus similar to System F, but with existential instead of universal quantification. Main article: Gradual typing In a type system with Gradual typing, variables may be assigned a type either at compile-time (which is static typing), or at run-time (which is dynamic typing).[26] This allows software developers to choose either type paradigm as appropriate, from within a single language.[26] Gradual typing uses a special type named dynamic to represent statically unknown types; gradual typing replaces the notion of type equality with a new relation called consistency that relates the dynamic type to every other type. The consistency relation is symmetric but not transitive.[27] Further information: Type inference Many static type systems, such as those of C and Java, require type declarations: the programmer must explicitly associate each variable with a specific type. Others, such as Haskell's, perform type inference: the compiler draws conclusions about the types of variables based on how programmers use those variables. For example, given a function f(x, y) that adds x and y together, the compiler can infer that x and y must be numbers—since addition is only defined for numbers. Thus, any call to f elsewhere in the program that specifies a non-numeric type (such as a string or list) as an argument would signal an error. Numerical and string constants and expressions in code can and often do imply type in a particular context. For example, an expression 3.14 might imply a type of floating-point, while [1, 2, 3] might imply a list of integers—typically an array. Type inference is in general possible, if it is computable in the type system in question. Moreover, even if inference is not computable in general for a given type system, inference is often possible for a large subset of real-world programs. Haskell's type system, a version of Hindley–Milner, is a restriction of System Fω to so-called rank-1 polymorphic types, in which type inference is computable. Most Haskell compilers allow arbitrary-rank polymorphism as an extension, but this makes type inference not computable. (Type checking is decidable, however, and rank-1 programs still have type inference; higher rank polymorphic programs are rejected unless given explicit type annotations.) Main article: Type theory § Decision problems A type system that assigns types to terms in type environments using typing rules is naturally associated with the decision problems of type checking, typability, and type inhabitation.[28] Given a type environment Γ {\displaystyle \Gamma } , a term e {\displaystyle e} , and a type τ {\displaystyle \tau } , decide whether the term e {\displaystyle e} can be assigned the type τ {\displaystyle \tau } in the type environment. Given a term e {\displaystyle e} , decide whether there exists a type environment Γ {\displaystyle \Gamma } and a type τ {\displaystyle \tau } such that the term e {\displaystyle e} can be assigned the type τ {\displaystyle \tau } in the type environment Γ {\displaystyle \Gamma } . Given a type environment Γ {\displaystyle \Gamma } and a type τ {\displaystyle \tau } , decide whether there exists a term e {\displaystyle e} that can be assigned the type τ {\displaystyle \tau } in the type environment. Some languages like C# or Scala have a unified type system.[29] This means that all C# types including primitive types inherit from a single root object. Every type in C# inherits from the Object class. Some languages, like Java and Raku, have a root type but also have primitive types that are not objects.[30] Java provides wrapper object types that exist together with the primitive types so developers can use either the wrapper object types or the simpler non-object primitive types. Raku automatically converts primitive types to objects when their methods are accessed.[31] A type checker for a statically typed language must verify that the type of any expression is consistent with the type expected by the context in which that expression appears. For example, in an assignment statement of the form x := e, the inferred type of the expression e must be consistent with the declared or inferred type of the variable x. This notion of consistency, called compatibility, is specific to each programming language. If the type of e and the type of x are the same, and assignment is allowed for that type, then this is a valid expression. Thus, in the simplest type systems, the question of whether two types are compatible reduces to that of whether they are equal (or equivalent). Different languages, however, have different criteria for when two type expressions are understood to denote the same type. These different equational theories of types vary widely, two extreme cases being structural type systems, in which any two types that describe values with the same structure are equivalent, and nominative type systems, in which no two syntactically distinct type expressions denote the same type (i.e., types must have the same "name" in order to be equal). In languages with subtyping, the compatibility relation is more complex: If B is a subtype of A, then a value of type B can be used in a context where one of type A is expected (covariant), even if the reverse is not true. Like equivalence, the subtype relation is defined differently for each programming language, with many variations possible. The presence of parametric or ad hoc polymorphism in a language may also have implications for type compatibility. Computer programming portal Comparison of type systems Covariance and contravariance (computer science) Polymorphism in object-oriented programming Type signature Type theory ^ The Burroughs ALGOL computer line determined a memory location's contents by its flag bits. Flag bits specify the contents of a memory location. Instruction, data type, and functions are specified by a 3 bit code in addition to its 48 bit contents. Only the MCP (Master Control Program) could write to the flag code bits. ^ For example, a leaky abstraction might surface during development, which may show that more type development is needed. —"The evaluation of a well-typed program always terminates".—B. Nordström, K. Petersson, and J. M. Smith[5] A systematic change in variables to avoid capture of a free variable can introduce error, in a functional programming language where functions are first class citizens.[6] —From the lambda calculus article. ^ Pierce 2002, p. 1: "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." ^ Cardelli 2004, p. 1: "The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program." ^ Pierce 2002, p. 208. ^ a b Sethi, R. (1996). Programming languages: Concepts and constructs (2nd ed.). Addison-Wesley. p. 142. ISBN 978-0-201-59065-4. OCLC 604732680. ^ Nordström, B.; Petersson, K.; Smith, J.M. (2001). "Martin-Löf's Type Theory". Algebraic and Logical Structures. Handbook of Logic in Computer Science. Vol. 5. Oxford University Press. p. 2. ISBN 978-0-19-154627-3. ^ Turner, D.A. (12 June 2012). "Some History of Functional Programming Languages" (PDF). invited lecture at TFP12, at St Andrews University. See the section on Algol 60. ^ "... any sound, decidable type system must be incomplete" —D. Remy (2017). p. 29, Remy, Didier. "Type systems for programming languages" (PDF). Archived from the original (PDF) on 14 November 2017. Retrieved 26 May 2013. ^ Pierce 2002. ^ a b c Skeet, Jon (2019). C# in Depth (4 ed.). Manning. ISBN 978-1617294532. ^ Miglani, Gaurav (2018). "Dynamic Method Dispatch or Runtime Polymorphism in Java". Archived from the original on 2020-12-07. Retrieved 2021-03-28. ^ Wright, Andrew K. (1995). Practical Soft Typing (PhD). Rice University. hdl:1911/16900. ^ "dynamic (C# Reference)". MSDN Library. Microsoft. Retrieved 14 January 2014. ^ "std::any — Rust". doc.rust-lang.org. Retrieved 2021-07-07. ^ Meijer, Erik; Drayton, Peter. "Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages" (PDF). Microsoft Corporation. ^ Laucher, Amanda; Snively, Paul (2012). "Types vs Tests". InfoQ. ^ Xi, Hongwei (1998). Dependent Types in Practical Programming (PhD). Department of Mathematical Sciences, Carnegie Mellon University. CiteSeerX 10.1.1.41.548.Xi, Hongwei; Pfenning, Frank (1999). "Dependent Types in Practical Programming". Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM. pp. 214–227. CiteSeerX 10.1.1.69.2042. doi:10.1145/292540.292560. ISBN 1581130953. S2CID 245490. ^ Visual Basic is an example of a language that is both type-safe and memory-safe. ^ "4.2.2 The Strict Variant of ECMAScript". ECMAScript® 2020 Language Specification (11th ed.). ECMA. June 2020. ECMA-262. ^ "Strict mode – JavaScript". MDN. Developer.mozilla.org. 2013-07-03. Retrieved 2013-07-17. ^ "Strict Mode (JavaScript)". MSDN. Microsoft. Retrieved 2013-07-17. ^ "Strict typing". PHP Manual: Language Reference: Functions. ^ a b Bracha, G. "Pluggable Types" (PDF). ^ "Sure. It's called "gradual typing", and I would qualify it as trendy. ..." Is there a language that allows both static and dynamic typing?. stackoverflow. 2012. ^ a b c Kopylov, Alexei (2003). "Dependent intersection: A new way of defining records in type theory". 18th IEEE Symposium on Logic in Computer Science. LICS 2003. IEEE Computer Society. pp. 86–95. CiteSeerX 10.1.1.89.4223. doi:10.1109/LICS.2003.1210048. ^ Mitchell, John C.; Plotkin, Gordon D. (July 1988). "Abstract Types Have Existential Type" (PDF). ACM Trans. Program. Lang. Syst. 10 (3): 470–502. doi:10.1145/44501.45065. S2CID 1222153. ^ a b Siek, Jeremy (24 March 2014). "What is gradual typing?". ^ Siek, Jeremy; Taha, Walid (September 2006). Gradual Typing for Functional Languages (PDF). Scheme and Functional Programming 2006. University of Chicago. pp. 81–92. ^ Barendregt, Henk; Dekkers, Wil; Statman, Richard (20 June 2013). Lambda Calculus with Types. Cambridge University Press. p. 66. ISBN 978-0-521-76614-2. ^ "8.2.4 Type system unification". C# Language Specification (5th ed.). ECMA. December 2017. ECMA-334. ^ "Native Types". Perl 6 Documentation. ^ "Numerics, § Auto-boxing". Perl 6 Documentation. Cardelli, Luca; Wegner, Peter (December 1985). "On Understanding Types, Data Abstraction, and Polymorphism" (PDF). ACM Computing Surveys. 17 (4): 471–523. CiteSeerX 10.1.1.117.695. doi:10.1145/6041.6042. S2CID 2921816. Pierce, Benjamin C. (2002). Types and Programming Languages. MIT Press. ISBN 978-0-262-16209-8. Cardelli, Luca (2004). "Type systems" (PDF). In Allen B. Tucker (ed.). CRC Handbook of Computer Science and Engineering (2nd ed.). CRC Press. ISBN 978-1584883609. Tratt, Laurence (July 2009). "5. Dynamically Typed Languages". Advances in Computers. Vol. 77. Elsevier. pp. 149–184. doi:10.1016/S0065-2458(09)01205-4. ISBN 978-0-12-374812-6. The Wikibook Ada Programming has a page on the topic of: Types The Wikibook Haskell has a page on the topic of: Class declarations Media related to Type systems at Wikimedia Commons Smith, Chris (2011). "What to Know Before Debating Type Systems". Retrieved from " Programming paradigm based on the concept of objects "Object-oriented" redirects here. For other meanings of object-oriented, see Object-orientation. 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. 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] 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. 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] 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 how to do messaging in a programming language efficiently enough to be useful). Alan Kay, [1] 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 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] 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[citation needed]. 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.[citation needed] 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. See also: Comparison of programming languages (object-oriented programming) and List of object-oriented programming terms 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] Further information: Imperative programming and Structured 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] Main article: Object (computer science) 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] 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] 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". 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] 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. 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] See also: Object-oriented design 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. 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 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 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. 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. This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (August 2009) (Learn how and when to remove this message) See also: List of object-oriented programming 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. 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. 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] 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] 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. 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: Basic message details - Information like message length, type, and data. Objects and collections - Similar to how objects work in Smalltalk, storing messages and their details. Managers - Like file directories, these organize and store data, as well as provide memory and processing power. They are similar to IBM i Objects. 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 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. 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. Main article: Design pattern (computer science) 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 Main articles: Object-relational impedance mismatch, Object-relational mapping, and Object database 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] 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 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. See also: Formal semantics of programming languages 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