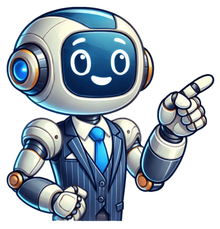
Click Here



This article possibly contains original research. Please improve it by verifying the claims made and adding inline citations. Statements consisting only of original research should be removed. (August 2009) (Learn how and when to remove this message) Programming languages can be grouped by the number and types of paradigms supported. A concise reference for the programming paradigms listed in this article. Concurrent programming – have language constructs for concurrency, these may involve multi-threading, support for distributed computing, message passing, shared resources (including shared memory), or futures Actor programming – concurrent computation with actors that make local decisions in response to the environment (capable of selfish or competitive behaviour) Constraint programming – relations between variables are expressed as constraints (or constraint networks), directing allowable solutions (uses constraint satisfaction or simplex algorithm) Dataflow programming – forced recalculation of formulas when data values change (e.g. spreadsheets) Declarative programming – describes what computation should perform, without specifying detailed state changes cf. imperative programming (functional and logic programming are major subgroups of declarative programming) Distributed programming – have support for multiple autonomous computers that communicate via computer networks Functional programming – uses evaluation of mathematical functions and avoids state and mutable data Generic programming – uses algorithms written in terms of to-be-specified-later types that are then instantiated as needed for specific types provided as parameters Imperative programming – explicit statements that change a program state Logic programming – uses explicit mathematical logic for programming Metaprogramming – writing programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime Template metaprogramming – metaprogramming methods in which a compiler uses templates to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled Reflective programming – metaprogramming methods in which a program modifies or extends itself Object-oriented programming – uses data structures consisting of data fields and methods together with their interactions (objects) to design programs Class-based – object-oriented programming in which inheritance is achieved by defining classes of objects, versus the objects themselves Prototype-based – object-oriented programming that avoids classes and implements inheritance via cloning of instances Pipeline programming – a simple syntax change to add syntax to nest function calls to language originally designed with none Rule-based programming – a network of rules of thumb that comprise a knowledge base and can be used for expert systems and problem deduction & resolution Visual programming – manipulating program elements graphically rather than by specifying them textually (e.g. Simulink); also termed diagrammatic programming[1] List of multi-paradigm programming languages Language Paradigm count Concurrent Constraints Dataflow Declarative Distributed Functional Metaprogramming Generic Imperative Logic Reflection Object-oriented Pipelines Visual Rule-based Other Ada[2][3][4][5][6] 5 Yes[a 1] — — — Yes — — Yes Yes — — Yes[a 2] — — — — ALF 2 — — — — — Yes — — — Yes — — — — — — AmigaE[citation needed] 2 — — — — — — — — Yes — — Yes[a 2] — — — — APL 3 — — — — — Yes — — Yes — — — — — — Array (multi-dimensional) BETA[citation needed] 3 — — — — — Yes — — Yes — — Yes[a 2] — — — — C++ 7 (15) Yes[7][8][9] Library[10] Library[11][12] Library[13][14] Library[15][16] Yes Yes[17] Yes[a 3] Yes Library[18][19] Library[20] Yes[a 2] Library[21] — Library[22] Array (multi-dimensional; using STL) C# 6 (7) Yes — Library[a 4] — — Yes[a 5] — Yes Yes — Yes Yes[a 2] — — — Reactive[a 6] ChucK[citation needed] 3 Yes — — — — — — — Yes — — Yes[a 2] — — — — Claire 2 — — — — — Yes — — — — — Yes[a 2] — — — — Clojure 5 Yes[23][24] — — Yes — Yes[25] Yes[26] — — Library[27] — — Yes[28] — — Multiple dispatch,[29] Agents[30] Common Lisp 7 (14) Library[31] Library[32] Library[33] Yes[34] Library[35] Yes Yes Yes[36] Yes Library[37] Yes Yes[a 7][a 2][38] Library[39] Library[40] Library[41] Multiple dispatch, meta-OOP system,[42] Language is extensible via metaprogramming. Curl 5 — — — — — Yes — Yes[a 3] Yes — Yes Yes[a 2] — — — — Curry 4 Yes Yes — — — Yes — — — Yes — — — — — — D (version 2.0)[43][44] 7 Yes[a 8] — — — — Yes Yes[45][a 3] Yes[a 3] Yes — Yes Yes[a 2] — — — — Delphi 3 — — — — — — — Yes[a 3] Yes — — Yes[a 2] — — — — Dylan[citation needed] 3 — — — — — Yes — — — — Yes Yes[a 2] — — — — E 3 Yes — — — Yes — — — — — — Yes[a 2] — — — — ECMAScript[46][47] (ActionScript, E4X, JavaScript, JScript) 4 (5) Partial[a 9][a 10] — — Library[48][49] — Yes — — Yes — Yes Yes[a 11] Library[50][51] — — Reactive,[a 12][52] event driven[a 13][a 14] Erlang 3 Yes — — Yes Yes Yes — — — — — — Yes — — — Elixir 4 Yes — — — Yes Yes Yes — — — — — Yes — — — Elm 6 Yes — Yes Yes — Yes (pure)[a 15] — Yes — — — — Yes — — Reactive F# 7 (8) Yes[a 8] — Library[a 4] Yes — Yes — Yes Yes — Yes Yes[a 2] — — — Reactive[a 6] Fortran 4 (5) Yes — — — — Yes[a 15] — Yes[a 16] — — — Yes[a 2] — — — Array (multi-dimensional) Go 4 Yes — — — — — — — Yes — Yes — Yes — — — Haskell 8 (15) Yes Library[53] Library[54] Yes Library[55] Yes (lazy) (pure)[a 15] Yes[56] Yes Yes Library[57] — Partial[a 17] Yes Yes Library[58] Literate, reactive, dependent types (partial) Io 4 Yes[a 8] — — — — Yes — — Yes — — Yes[a 11] — — — — J[citation needed] 3 — — — — — Yes — — Yes — — Yes[a 2] — — — — Java 6 Yes Library[59] Library[60] — — Yes — Yes Yes — Yes Yes[a 2] — — — — Julia 9 (17) Yes Library[61] Library[62][63] Library[64] Yes Yes (eager) Yes Yes Yes Library[65] Yes Partial[a 18] Yes — Library[66][67] Multiple dispatch,Array (multi-dimensional); optionally lazy[68] and reactive (with libraries) Kotlin 8 Yes — — — — Yes Yes Yes Yes — Yes Yes Yes — — — LabVIEW 4 Yes — Yes — — — — — — — — — Yes — Yes — — Lava 2 — — — — — — — — — — — Yes[a 2] — Yes — — LispWorks (version 6.0 with support for symmetric multi-processing, rules, logic (Prolog), CORBA) 9 Yes — — — Yes Yes Yes — Yes Yes Yes Yes[a 2] — — Yes — Lua[citation needed] 3 — — — — — Yes — — Yes — — Yes[a 11] — — — — MATLAB 6 (10) Toolbox[69] Toolbox[70] Yes[71] — Toolbox[72] — Yes[73] Yes[74] — — Yes[75] Yes[76] — Yes[77] — Array (multi-dimensional) Nemerle 7 Yes — — — — Yes Yes Yes Yes — Yes Yes[a 2] — — — — Object Pascal 4 Yes — — — — Yes — — Yes — — Yes[a 2] — — — — OCaml 4 — — — — — Yes — Yes Yes — — Yes[a 2] — — — — Oz 11 Yes Yes Yes Yes Yes Yes — — Yes Yes — Yes[a 2] Yes — Yes — Perl[citation needed] 8 (9) Yes[78] — Yes[79] — — Yes Yes — Yes — Yes[a 2] Yes[a 2] Yes — — — PHP[80][81][82] 4 — — — — — Yes — — Yes — Yes Yes[a 2] — — — — Poplog 3 — — — — — Yes — — Yes Yes — — — — — — Prograph 3 — — Yes — — — — — — — — — Yes[a 2] — Yes — — Python 5 (10) Library[83][84] Library[85] — — Library[86] Yes Yes[87][88] Yes[89][90] Yes Library[91] Yes Yes[a 2] — — — Structured R 4 (6) Library[92] — — — Library[93] Yes — — Yes — Yes Yes Yes[94] — — Array (multi-dimensional) Racket 10 Yes[95] Yes[96] Yes[97] — Yes[98] Yes Yes — Yes Yes Yes Yes — — — Lazy[99] Raku 10 Yes[100] Library[101] Yes[102] — Library[103] Yes Yes[104] Yes[105] Yes — Yes[106] Yes[107] Yes — — Multiple dispatch, lazy lists, reactive. ROOP 3 — — — — — — — — Yes Yes — — — — Yes — Ruby 5 — — — — — Yes Yes — Yes — Yes Yes[a 2] — — — — Rust (version 1.0.0-alpha) 6 Yes[a 8] — — — — Yes Yes[108][109] Yes[110] Yes — — Yes — — — Linear, affline, and ownership types Sather[citation needed] 2 — — — — — Yes — — — — — Yes[a 2] — — — — Scala[111][112] 9 Yes[a 8] — Yes[a 19] Yes — Yes Yes Yes Yes — Yes Yes[a 2] — — — — Simula[citation needed] 2 — — — — — — — — Yes — — Yes[a 2] — — — — SISAL 3 Yes — Yes — — Yes — — — — — — — — — — — Spreadsheets 2 — — — — — Yes — — — — — — — — Yes — — Swift 7 Yes — — — — Yes Yes Yes Yes — Yes Yes[a 2] — — — Block-structured Tcl with Snit extension[citation needed] 3 — — — — — Yes[113] — — Yes — — Yes[a 11][114] — — — — Visual Basic .NET 6 (7) Yes — Library[a 4] — — Yes — Yes Yes — Yes Yes[a 2] — — — Reactive[a 6] Windows PowerShell 6 — — — — — Yes — Yes Yes — Yes Yes[a 2] Yes — — — Wolfram Language & Mathematica 13[115] (14) Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes[116] — Yes Knowledge Based Programming paradigm List of programming languages by type Domain-specific language Domain-specific multimodeling ^ rendezvous and monitor-like based ^ a b c d e f g h i j k l m n o p q r s t u v w x y z aa ab ac ad ae af ag ah ai class-based ^ a b c d e template metaprogramming ^ a b c using TPL Dataflow ^ only lambda support (lazy functional programming) ^ a b c using Reactive Extensions (Rx) ^ multiple dispatch, method combinations ^ a b c d e actor programming ^ promises, native extensions ^ using Node.js' cluster module or child_process.fork method, web workers in the browser, etc. ^ a b c d Prototype-based ^ using Reactive Extensions (RxJS) ^ in Node.js via their events module ^ in browsers via their native EventTarget API ^ a b c purely functional ^ parameterized classes ^ immutable ^ Uses structs with function polymorphism and multiple dispatch ^ Akka Archived 2013-01-19 at the Wayback Machine ^ Bragg, S.D.; Driskill, C.G. (20–22 September 1994). "Diagrammatic-graphical programming languages and DoD-STD-2167A". Proceedings of AUTOTESTCON '94 (IEEEXplore). Institute of Electrical and Electronics Engineers (IEEE). pp. 211–220. doi:10.1109/AUTEST.1994.381508. ISBN 978-0-7803-1910-3. S2CID 62509261. ^ Ada Reference Manual, ISO/IEC 8652:2005(E) Ed. 3, Section 9: Tasks and Synchronization ^ Ada Reference Manual, ISO/IEC 8652:2005(E) Ed. 3 Annex E: Distributed Systems ^ Ada Reference Manual, ISO/IEC 8652:2005(E) Ed. 3, Section 12: Generic Units ^ Ada Reference Manual, ISO/IEC 8652:2005(E) Ed. 3, Section 6: Subprograms ^ Ada Reference Manual, ISO/IEC 8652:2005(E) Ed. 3, 3.9 Tagged Types and Type Extensions ^ Thread support ^ Atomics support ^ Memory model ^ Gecode ^ SystemC ^ Boost.Iostreams ^ Booling ^ "AraRat" (PDF). Archived from the original (PDF) on 2019-08-19. Retrieved 2019-09-15. ^ OpenMPI ^ Boost.MPI ^ Boost.MPL ^ LC++ ^ Castor Archived 2013-01-25 at the Wayback Machine ^ Reflect Library ^ N3534 ^ Boost.Spirit ^ Clojure - Concurrent Programming ^ Clojure - core.async ^ Clojure - Functional Programming ^ Clojure - Macros ^ Clojure - core.logic ^ Clojure - Threading Macros Guide ^ Multimethods and Hierarchies ^ Agents and Asynchronous Actions ^ "concurrency". CLiki. ^ [1] constraint programming inside CL through extensions ^ [2] dataflow extension ^ [3] by creating DSLs using the built-in metaprogramming; also see note on functional, constraint and logic paradigms, which are part of declarative ^ [4] MPI, etc via language extensions ^ template metaprogramming using macros (see C++) ^ [5] [6] [7] Prolog implemented as a language extension ^ Common Lisp Object System see Wikipedia article on CLOS, the Common Lisp Object System. ^ implemented by the user via a short macro, example of implementation ^ - Visual programming tool based on Common Lisp ^ [8] rule-based programming extension ^ [9] Archived 2018-04-26 at the Wayback Machine through the Meta Object Protocol ^ D Language Feature Table ^ Phobos std.algorithm ^ D language String Mixins ^ The Little JavaScripter demonstrates fundamental commonality with Scheme, a functional language. ^ Object-Oriented Programming in JavaScript Archived 2019-02-10 at the Wayback Machine gives an overview of object-oriented programming techniques in JavaScript. ^ "React – A JavaScript library for building user interfaces". 2019-04-08. ^ "TNG-Hooks". GitHub. 2019-04-08. ^ "Lodash documentation". 2019-04-08. ^ "mori". 2019-04-08. ^ "TNG-Hooks". GitHub. 2019-04-08. ^ "Prolog embedding". Haskell.org. ^ "Functional Reactive Programming". HaskellWiki. ^ Cloud Haskell ^ "Template Haskell". HaskellWiki. ^ "Logict: A backtracking logic-programming monad". Haskell.org. ^ Kollmansberger, Steve; Erwig, Martin (30 May 2006). "Haskell Rules: Embedding Rule Systems in Haskell" (PDF). Oregon State University. ^ JSR 331: Constraint Programming API ^ Google Cloud Platform Dataflow SDK ^ "JuliaOpt/JuMP.jl". GitHub. JuliaOpt. 11 February 2020. Retrieved 12 February 2020. ^ "GitHub - MikeInnes/DataFlow.jl". GitHub. 2019-01-15. ^ "GitHub - JuliaGizmos/Reactive.jl: Reactive programming primitives for Julia". GitHub. 2018-12-28. ^ Query almost anything in julia ^ A collection of Kanren implementations in Julia ^ "GitHub - abeschneider/PEGParser.jl: PEG Parser for Julia". GitHub. 2018-12-03. ^ "GitHub - gitfoxi/Parsimonious.jl: A PEG parser generator for Julia". GitHub. 2017-08-03. ^ Lazy ^ "Execute loop iterations in parallel". mathworks.com. Retrieved 21 October 2016. ^ "Write Constraints". mathworks.com. Retrieved 21 October 2016. ^ "Getting Started with SimEvents". mathworks.com. Retrieved 21 October 2016. ^ "Execute loop iterations in parallel". mathworks.com. Retrieved 21 October 2016. ^ "Execute MATLAB expression in text - MATLAB eval". mathworks.com. Retrieved 21 October 2016. ^ "Determine class of object". mathworks.com. Retrieved 21 October 2016. ^ "Class Metadata". mathworks.com. Retrieved 21 October 2016. ^ "Object-Oriented Programming". mathworks.com. Retrieved 21 October 2016. ^ "Simulink". mathworks.com. Retrieved 21 October 2016. ^ interpreter based threads ^ Higher Order Perl ^ PHP Manual, Chapter 17. Functions ^ PHP Manual, Chapter 19. Classes and Objects (PHP 5) ^ PHP Manual, Anonymous functions ^ "Parallel Processing and Multiprocessing in Python". Python Wiki. Retrieved 21 October 2016. ^ "threading — Higher-level threading interface". docs.python.org. Retrieved 21 October 2016. ^ "python-constraint". pypi.python.org. Retrieved 21 October 2016. ^ "DistributedProgramming". Python Wiki. Retrieved 21 October 2016. ^ "Chapter 9. Metaprogramming". chimera.labs.oreilly.com. Archived from the original on 23 October 2016. Retrieved 22 October 2016. ^ "Metaprogramming". readthedocs.io. Retrieved 22 October 2016. ^ "PEP 443 – Single-dispatch generic functions". python.org. Retrieved 22 October 2016. ^ "PEP 484 – Type Hints". python.org. Retrieved 22 October 2016. ^ "PyDatalog". Retrieved 22 October 2016. ^ "Futureverse". ^ "future batchtools". ^ "Magrittr: A Forward Pipe Operator for R". cran.r-project.org\access-date=13 July 2017. 17 November 2020. ^ Racket Guide: Concurrency and Synchronization ^ The Rosette Guide ^ FrTime: A Language for Reactive Programs ^ Racket Guide: Distributed Places ^ Lazy Racket ^ Channels and other mechanisms ^ "Problem Solver module". ^ Feed operator ^ Cro module ^ "Meta-programming: What, why and how". 2011-12-14. ^ Parametrized Roles ^ "Meta-object protocol (MOP)". ^ Classes and Roles ^ "The Rust macros guide". Rust. Retrieved 19 January 2015. ^ "The Rust compiler plugins guide". Rust. Retrieved 19 January 2015. ^ The Rust Reference §6.1.3.1 ^ An Overview of the Scala Programming Language ^ Scala Language Specification ^ "Tcl Programming/Introduction". en.wikibooks.org. Retrieved 22 October 2016. ^ "TCLLIB - Tcl Standard Library: snitfaq". sourceforge.net. Retrieved 22 October 2016. ^ Notes for Programming Language Experts, Wolfram Language Documentation. ^ External Programs, Wolfram Language Documentation. Jim Coplien, Multiparadigm Design for C++, Addison-Wesley Professional, 1998. Retrieved from "

Unlock the secrets of JavaScript date handling! Tackle timezone confusion and master date manipulation with expert tips and powerful libraries. Handling dates and times in JavaScript can be a complex and occasionally frustrating task, especially when it comes to time zones. JavaScript's date handling capabilities can be perplexing due to inconsistencies and the fact that the language operates on UTC (Coordinated Universal Time) behind the scenes. In this article, we will explore best practices for managing dates and time zones effectively in JavaScript and its related frameworks. Understanding JavaScript's Date Object The foundation of date handling in JavaScript lies with the Date object. This built-in object allows you to create, manipulate, and format date and time. However, it is important to grasp some of its quirks. Creating Dates You can create a date using the following syntax: // Create a new Date object with the current date and time const currentDate = new Date(); // Create a Date object for a specific date and time const specificDate = new Date('2023-10-01T10:20:30Z'); In the example above, the first line creates a date object representing the current date and time. The second line creates a date for October 1, 2023, at 10:20:30 UTC. Remember that dates will be represented in UTC if a "Z" is included in the string. Using the built-in Date object also raises some considerations about time zones. By default, Date objects are instantiated in the user's local time zone. If you need to work with non-local time, you must convert or format dates accordingly. Getting Date Components To extract specific components like the year, month, or day from a date, you can use methods like getFullYear(), getMonth(), and getDate(): const date = new Date('2023-10-01T10:20:30Z'); const year = date.getUTCFullYear(); // Use getUTC* methods to avoid local time zone offsets const month = date.getUTCMonth() + 1; // Months are zero-indexed (0-11) const day = date.getUTCDate(); console.log(`Year: ${year}, Month: ${month}, Day: ${day}`); Time Zones: The Challenges One major complication arises from the different time zones around the world. JavaScript's native Date object is not timezone-aware. This means that if you're handling time-sensitive data across different geographical locations, you'll run into potentially confusing scenarios. To mitigate issues related to time zones, consider using libraries like Luxon, Date-fns, or Moment.js (although Moment.js is being deprecated). These libraries can help manage, convert, and format dates more easily than the native Date methods. Best Practices for Time Zone Management Below are best practices to effectively manage dates and times in your JavaScript applications: 1. Use UTC for Storage When storing datetime values in databases or APIs, it is best to use the UTC format. This avoids discrepancies when users in different time zones access the data. // Create a UTC date string for API storage const utcDateString = new Date().toISOString(); console.log(`UTC Date String: ${utcDateString}`); 2. Display Dates in Localized Formats When presenting dates to users, convert them into the local time zone. You can achieve this using the toLocaleString() method: const localDateTime = currentDate.toLocaleString('en-US', { timeZone: 'America/New_York', // Specify the required time zone }); console.log(`Local Date/Time: ${localDateTime}`); 3. Leverage Date Libraries Using libraries like Luxon can significantly simplify your time zone management process. Here is how to use Luxon for handling a datetime object: // Import the Luxon library (assuming it is installed) const { DateTime } = require('luxon'); // Create a DateTime object in a specific time zone const dt = DateTime.fromISO('2023-10-01T10:20:30', { zone: 'America/New_York' }); console.log(`Local Time: ${dt.toString()}`); console.log(`UTC Equivalent: ${dt.toUTC().toString()}`); 4. Handle Input Date Formats Always validate and parse incoming date strings accurately. Use libraries that can parse various date formats smoothly. const { DateTime } = require('luxon'); const input = '2023-10-01T10:20:30'; const parsedDate = DateTime.fromISO(input); if (!parsedDate.isValid) { console.error('Invalid date format!'); } else { console.log(`Parsed Date: ${parsedDate.toString()}`); } 5. Testing Your Dates Ensure your application behaves as expected in different time zones by conducting thorough testing. // Example test case for displaying a date in a different time zone const testDateTime = DateTime.fromISO('2023-10-01T10:20:30', { zone: 'UTC' }); console.log(`UTC Time: ${testDateTime.toString()}`); console.log(`New York Time: ${testDateTime.setZone('America/New_York').toString()}`); The Last Word Navigating date and time handling in JavaScript can be a challenging task due to time zone discrepancies and client-side variability. By adhering to the best practices outlined in this article, you can effectively manage dates within your applications. Remember to always store datetime values in UTC, display them in the user's local timezone, and leverage reliable libraries for parsing and formatting. For more in-depth understanding, refer to the excellent MDN documentation on Date and consider exploring the functionalities offered by libraries like Luxon or Date-fns. Adopt these practices, and eliminate the confusion surrounding dates and timezones in your JavaScript projects! Happy coding!

Programming language type systems 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, one of the many ways that programming languages are colloquially classified is whether the language's type system makes it strongly typed or weakly typed (loosely typed). However, there is no precise technical definition of what the terms mean and different authors disagree about the implied meaning of the terms and the relative rankings of the "strength" of the type systems of mainstream programming languages.[1] For this reason, writers who wish to write unambiguously about type systems often eschew the terms "strong typing" and "weak typing" in favor of specific expressions such as "type safety". Generally, a strongly typed language has stricter typing rules at compile time, which implies that errors are more likely to happen during compilation. Most of these rules affect variable assignment, function return values, procedure arguments and function calling. Dynamically typed languages (where type checking happens at run time) can also be strongly typed. In dynamically typed languages, values, rather than variables, have types. A weakly typed language has looser typing rules and may produce unpredictable or even erroneous results or may perform implicit type conversion at runtime.[2] A different but related concept is latent typing. In 1974, Barbara Liskov and Stephen Zilles defined a strongly typed language as one in which "whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function."[3] In 1977, K. Jackson wrote, "In a strongly typed language each data area will have a distinct type and each process will state its communication requirements in terms of these types."[4] A number of different language design decisions have been referred to as evidence of "strong" or "weak" typing. Many of these are more accurately understood as the presence or absence of type safety, memory safety, static type-checking, or dynamic type-checking. "Strong typing" generally refers to use of programming language types in order to both capture invariants of the code, and ensure its correctness, and definitely exclude certain classes of programming errors. Thus there are many "strong typing" disciplines used to achieve these goals. Some programming languages make it easy to use a value of one type as if it were a value of another type. This is sometimes described as "weak typing". For example, Aahz Maruch observes that "Coercion occurs when you have a statically typed language and you use the syntactic features of the language to force the usage of one type as if it were a different type (consider the common use of void* in C). Coercion is usually a symptom of weak typing. Conversion, on the other hand, creates a brand-new object of the appropriate type."[5] As another example, GCC describes this as type-punning and warns that it will break strict aliasing. Thiago Macieira discusses several problems that can arise when type-punning causes the compiler to make inappropriate optimizations.[6] There are many examples of languages that allow implicit type conversions, but in a type-safe manner. For example, both C++ and C# allow programs to define operators to convert a value from one type to another with well-defined semantics. When a C++ compiler encounters such a conversion, it treats the operation just like a function call. In contrast, converting a value to the C type void* is an unsafe operation that is invisible to the compiler. Some programming languages expose pointers as if they were numeric values, and allow users to perform arithmetic on them. These languages are sometimes referred to as "weakly typed", since pointer arithmetic can be used to bypass the language's type system. Some programming languages support untagged unions, which allow a value of one type to be viewed as if it were a value of another type. In Luca Cardelli's article Typeful Programming,[7] a "strong type system" is described as one in which there is no possibility of an unchecked runtime type error. In other writing, the absence of unchecked run-time errors is referred to as safety or type safety; Tony Hoare's early papers call this property security.[8] This section possibly contains original research. Please improve it by verifying the claims made and adding inline citations. Statements consisting only of original research should be removed. (May 2018) (Learn how and when to remove this message) This section needs additional citations for verification. Please help improve this article by adding citations to reliable sources in this section. Unsourced material may be challenged and removed. (May 2020) (Learn how and when to remove this message) Some of these definitions are contradictory, others are merely conceptually independent, and still others are special cases (with additional constraints) of other, more "liberal" (less strong) definitions. Because of the wide divergence among these definitions, it is possible to defend claims about most programming languages that they are either strongly or weakly typed. For instance: Java, Pascal, Ada, and C require variables to have a declared type, and support the use of explicit casts of arithmetic values to other arithmetic types. Java, C#, Ada, and Pascal are sometimes said to be more strongly typed than C, because C supports more kinds of implicit conversions, and allows pointer values to be explicitly cast while Java and Pascal do not. Java may be considered more strongly typed than Pascal as methods of evading the static type system in Java are controlled by the Java virtual machine's type system. C# and VB.NET are similar to Java in that respect, though they allow disabling of dynamic type checking by explicitly putting code segments in an "unsafe context". Pascal's type system has been described as "too strong", because the size of an array or string is part of its type, making some programming tasks very difficult. However, Delphi fixes this issue.[9][10] Smalltalk, Ruby, Python, and Self are all "strongly typed" in the sense that typing errors are prevented at runtime and they do little implicit type conversion, but these languages make no use of static type checking: the compiler does not check or enforce type constraint rules. The term duck typing is now used to describe the dynamic typing paradigm used by the languages in this group. The Lisp family of languages are all "strongly typed" in the sense that typing errors are prevented at runtime. Some Lisp dialects like Common Lisp or Clojure do support various forms of type declarations[11] and some compilers (CMU Common Lisp (CMUCL)[12] and related) use these declarations together with type inference to enable various optimizations and limited forms of compile time type checks. Standard ML, F#, OCaml, Haskell, Go and Rust are statically type-checked, but the compiler automatically infers a precise type for most values. Assembly language and Forth can be characterized as untyped. There is no type checking; it is up to the programmer to ensure that data given to functions is of the appropriate type. Comparison of programming languages Data type includes a more thorough discussion of typing issues Design by contract (strong typing as implicit contract form) Latent typing Memory safety Type safety Type system Strongly typed identifier

^ "What to know before debating type systems | Ovid [blogs.perl.org]". blogs.perl.org. Retrieved 2023-06-27. ^ "CS1130. Transition to OO programming. – Spring 2012 --self-paced version". Cornell University, Department of Computer Science. 2005. Archived from the original on 2015-11-23. Retrieved 2015-11-23.{{cite web}}: CS1 maint: bot: original URL status unknown (link) ^ Liskov, B; Zilles, S (1974). "Programming with abstract data types". ACM SIGPLAN Notices. 9 (4): 50–59. CiteSeerX 10.1.1.136.3043. doi:10.1145/942572.807045. ^ Jackson, K. (1977). "Parallel processing and modular software construction". Design and Implementation of Programming Languages. Lecture Notes in Computer Science. Vol. 54. pp. 436–443. doi:10.1007/BFb0021435. ISBN 3-540-08360-X. ^ Aahz. "Typing: Strong vs. Weak, Static vs. Dynamic". Retrieved 16 August 2015. ^ "Type-punning and strict-aliasing - Qt Blog". Qt Blog. Retrieved 18 February 2020. ^ Luca Cardelli, "Typeful programming" ^ Hoare, C. A. R. 1974. Hints on Programming Language Design. In Computer Systems Reliability, ed. C. Bunyan. Vol. 20 pp. 505–534. ^ InfoWorld. 1983-04-25. Retrieved 16 August 2015. ^ Kernighan, Brian (1981). "Why Pascal is not my favorite programming language". Archived from the original on 2012-04-06. Retrieved 2011-10-22. ^ "CLHS: Chapter 4". Retrieved 16 August 2015. ^ "CMUCL User's Manual: The Compiler". Archived from the original on 8 March 2016. Retrieved 16 August 2015. Retrieved from "