

## **Brief Summation of Design Trade-Offs Between Dynamic & Static** **Scoping, Compile-Time Binding & Run-Time Binding in** **Programming Languages**

A static-scoped programming language defines its scope by the locations of declarations, meaning that to discover which definition of a variable or function is utilized when the program runs, one must only examine the surroundings of the code's moment of utilization; the nesting, whether any variable has been shadowed, things of this sort that are directly visible through the structure of the code. A dynamic-scoped programming language, however, defines its scope based on an ever-expanding call stack which stores the most recently-called function or variable, meaning there is much more flexibility to how a function or variable could be evaluated based on the order of execution than in static-scoping languages. While a heaping majority of programming languages today are statically-scoped (C++, Java, JavaScript, Python, etc...), there exist a handful of readily-available programming languages (Emacs Lisp, LaTeX/Tex, Bash) which deploy dynamic scoping for its versatility and freer flow of information.

Static scoping languages are designed primarily for efficiency and predictability in both how they run and how they are read by programmers, since defining scope by where things are recorded in code itself is much more understandable to human eyes and can be easily inferred and thus analyzed. Most static-scoped languages utilize some form of compile-time binding, whether pure (C++) or hybridized (Java, JavaScript), in which the compiler is allowed to determine scope far before runtime (this is referred to as "compile time binding" for variables, though some languages like Java or Python may also say they use "run time binding" due to the load being handled by both a simplified compile process and run-time interpreting); this makes static-scoped languages run much faster than the constant call-stack abuse performed during runtime (this is

referred to as “run time binding” for variables) of the dynamic-scoped languages. Static-scoped languages are less likely to encounter errors due to the limitations of scope access that are not present in dynamic-scoped languages, though dynamic-scoped languages lacking this security enables them to reuse their code far more between segments and to debug recursive functions far easier. Dynamic-scoped languages adapt more easily than static-scoped ones to erratic changes in their environment and the prerequisites for their code to work, since constant re-accessing of variables and functions does not limit programmers to a single-time commitment. Dynamic-scoped languages are essentially slower, riskier and more unwieldy to comprehend than static-scoped languages, but once mastered, can become far more flexible and less redundant. Compile-time binding is similar to its frequent user in static-scoped languages insofar as it is safe, understandable, fast to do, but limited by its restrictions on typing; once a variable is defined and cast as a certain type, it is far harder to recast its type than it is with run-time binding. Run-time binding, like its frequent user, dynamic-scoped languages, is slow and costly on a surface level, though has a myriad use case for situations where extra room for redeclaration and change is necessary– which in turn means you may have to use far less variables and functions than compile-time binding does for the same effect.