

# neo

**neo is written lowercase. Always.**

## Abstract

A computer should be a living medium, not a toolbox. The program, operating system, live world environment, and development environment should be one thing, not four things communicating through APIs.

neo is that one thing: a live, homoiconic, classless, prototype based, pure object system that is simultaneously a programming language, an operating system, and a game engine (live world environment). Not three subsystems, but three ways of looking at the same object graph.

| *neo currently specifies an object-graph substrate, not yet a complete programming language.*

## Everything is an object

Every value in neo is an object. Numbers, booleans, running processes, methods, errors, and the editor itself are all objects responding to the same mechanism.

## Objects communicate through objects

Objects communicate by sending messages. Messages and running calls are themselves objects, participating in the same world as their senders and receivers. Just as a chemical signal between cells belongs to the same physical world as the cells, communication in neo belongs to the same object graph as the objects communicating.

This is an ontological commitment. How messages are dispatched and calls are executed is an implementation question; their place in the object world is part of the theory.

## Code is data

A neo program is not text that compiles into something else. It is a graph of objects in the same image as everything else. A method is an object. A running call is an object. Program structure, communication, and execution state all inhabit the same object ontology.

Homoiconicity follows from representing programs as objects in that world. Treating messages and running calls as objects extends this uniformity to communication and execution. It is a broader

commitment than homoiconicity alone requires.

This is homoiconicity in the strongest sense. Code isn't merely inspectable as data, it is data, completely, without exception. You can reach into a running program, find a method object, change one of its slots, and the change is live in the next call. There is no edit-compile-run cycle, only the image.

## **The object graph is n-dimensional**

Every object inhabits a space. Dimensionality belongs to the object itself, not to an external coordinate system imposed from outside. A 2D panel, a 3D simulation body, a 10-dimensional embedding, and a scalar with no spatial meaning coexist in the same image under a uniform spatial protocol. Objects that need no spatial properties pay nothing for the ones that do.

Spaces are first-class objects. Rendering is a morphism from an object's native space onto the viewer's space. The scene graph is not a separate data structure; it is the natural relation between objects that share a geometric space. Spatial indexing, physics, and constraints follow the same pattern, one mechanism instead of four.

## **Prototype-based, no classes**

There are no class definitions. Objects inherit from objects directly. To make a new kind of thing, clone an existing thing and change what differs. The clone is immediately live.

This is how the physical world works. You start from an example and vary it. The blueprint is a fiction. In neo, the prototype is the first example, real, inspectable, and modifiable, and every other instance is a variation of it.

## **Security from the reference graph**

To act on an object you must hold a reference to it. References are unforgeable. There is no global namespace, no ambient authority, and no root access that bypasses the object graph. A process that starts with no references can do nothing, not because anything checks, but because there is nothing it can reach.

Reflection is a capability: an object you must hold, not a power every developer has by default. This resolves the long-standing tension in live systems where openness and security are treated as opposites. In neo they are the same property; the reference graph enforces both.

## **Self-reproduction**

A system is self-reproducing when it can describe itself completely and reconstruct itself from that description. neo achieves this at three levels.

The image serialises to text. That text, when evaluated, reconstructs the exact image that produced it, including the serialiser. The image is its own source code.

The prototype chain bottoms out at a fixed point: ``Object``, whose prototype is itself. From this fixed point the entire object hierarchy is generated by cloning. The system contains the description of the thing that generates it.

The running image holds a representation of the kernel itself as objects. The image can modify those objects, emit a new kernel, and restart into it while preserving the image. The substrate is not fixed; it can be modified from within the system it hosts.

This is what von Neumann described in 1948 as a universal constructor: a machine containing a complete description of itself, capable of reproducing that description, and capable of modifying the description before reproducing. neo is a software realisation of that structure.

## The prior tradition

Self contributed prototype delegation, uniform slots, and the inline cache. Smalltalk contributed the live image and message passing as the only operation. Lisp contributed homoiconicity. None of them completed the synthesis. Self had no homoiconicity. Smalltalk had classes, compiler primitives, and a UI model that was not genuinely object-oriented. Lisp had no objects at the ontological level; functions are the primitive, not objects.

neo takes what each got right and refuses what each got wrong. The per-object dimensional model is neo's original contribution; no prior system gave dimensionality to the object itself.

The prior tradition is the floor, not the ceiling.

## o-expression: the reference syntax

neo has no canonical syntax. Surface representations are external readers, objects that consume text and produce graph fragments. Any reader that produces the correct object graph is a valid neo reader.

The system ships with *o-expression* as its reference syntax. o-expression is named by contrast with s-expression. Where s-expressions are lists, sequences of symbols and nested lists, o-expressions are objects containing objects. Every ``(...)`` is one object. The ``o`` names that.

Every ``(...)`` is exactly one object. The first token is the object's name. The remaining contents are its child objects. Nesting is containment. Three characters, nothing else:

| Character | Meaning         |
|-----------|-----------------|
| (         | open an object  |
| )         | close an object |
| >         | line comment    |

---

## Example syntax

---

> an object containing two children

```
(+ (21) (21))
```

> a nested object

```
(counter  
  (count (0))  
  (increment  
    (body (+ (count (self)) (1)))))
```

> control flow: if is an object with three children

```
(if  
  (> (x) (0))  
  (print (x))  
  (print (0)))
```

> a simulation body in 3D space

```
(body  
  (space (Euclidean3D))  
  (position (vec3 (x (0)) (y (0)) (z (0))))  
  (mass (1.0)))
```

> duplication

```
(counter (signal (copy)))
```

---