ObjectIR Language Specification

A portable, stack-based intermediate representation

Version 1.0
January 18, 2026

Contents

1 Introduction

ObjectIR is a portable, stack-based intermediate representation designed for execution, analysis, and compilation across multiple runtimes. It emphasizes explicit control flow, verifiable semantics, and strong tooling support.

2 Design Goals

- Platform-agnostic execution
- Explicit and deterministic semantics
- Verifiable stack discipline
- Friendly to JIT, AOT, and interpreter backends

3 Execution Model

3.1 Evaluation Stack

ObjectIR uses an implicit evaluation stack. Most instructions consume zero or more values from the stack and may push results back onto it.

3.2 Arguments and Locals

Arguments and local variables are addressed by identifier. Instance methods implicitly receive a `this` argument.

4 Stack Discipline and Verification

This section defines mandatory rules that all valid ObjectIR programs must satisfy. These rules enable static verification and predictable execution.

4.1 Stack Balance Rules

- The evaluation stack is empty at method entry.
- At each `ret` instruction:
  - `void` methods must leave the stack empty.
  - Non-void methods must leave exactly one value matching the declared return type.
- No instruction may consume more stack values than are present.

4.2 Control Flow Merge Rules

At any control-flow merge point (e.g. end of `if`, loop headers):

- All incoming paths must have identical stack height.
- Stack value types must match exactly or be verifier-compatible.

4.3 Type Safety

- Stack operations are fully type-checked.
- Arithmetic instructions require compatible numeric types.
- Method calls must match the exact target signature.
- Field, local, and argument accesses must reference declared symbols.

4.4 Initialization Rules

- Local variables must be assigned before use.
- Fields must be initialized before being read.
- Constructors are responsible for establishing object invariants.

4.5 Undefined Behavior

Programs that violate verifier rules have undefined behavior. Backends may reject invalid IR or apply stricter validation.

5 Instruction Set

5.1 Textual IR Format

The C++ runtime includes a textual IR parser used by the tooling provided by the default ObjectIR stack, The format below reflects the concrete syntax consumed by `IRTextParser` and the JSON operands expected by the runtime. this is the required first option besides Fob format for saving ObjectIR to a file.

5.1.1 Lexical Rules

- Whitespace separates tokens; newlines terminate instruction lines.
- Line comments start with `//` and continue to the end of the line.
- Identifiers may include letters, digits, underscores, dots, and backticks (e.g. `Listı̀`).
- String literals use double quotes and support escapes: ¨,
  , `\n`, `\r`, `\t`.
- Numeric literals support integer and decimal forms; negative literals are written with a leading `-`.

5.1.2 Keywords and Declarations

- Keywords: `module`, `class`, `interface`, `struct`, `enum`, `method`, `constructor`, `field`, `property`, `static`, `virtual`, `abstract`, `private`, `public`, `protected`, `local`, `if`, `else`, `while`, `for`, `switch`, `case`, `return`, `implements`, `version`.
- Local declarations inside method bodies use `local name: type`.
- Labels are declared as `labelName:` and resolve branch targets.

5.1.3 Method Bodies

- A method body is a sequence of local declarations and instruction lines inside `{}`.
- Each instruction consumes the remainder of its line as operands; the parser stops at newline or a brace.
- Branch operands may be numeric instruction indices or label names; labels are mapped to instruction indices during parsing.

5.1.4 Primitive Types

- `void`, `bool`
- `int8`, `int16`, `int32`, `int64`
- `uint8`, `uint16`, `uint32`, `uint64`
- `float32`, `float64`
- `char`, `string`

5.1.5 Method References

Method references used by `call`/`callvirt` follow the textual shape:

```
        TypeName.MethodName ( paramType1, paramType2 ) -> returnType
```

Constructors use `TypeName..ctor` as the method name in the text form.

5.2 Conventions

- Mnemonics are case-insensitive.
- Identifiers are case-sensitive.
- Stack effects are written as `inputs` `outputs`.

5.3 Opcode Summary

The following mnemonics are recognized by the Reference C++ runtime. Variants noted in the *Aliases* column map to the same opcode.

Mnemonic	Operands	Stack Effect	Notes / Aliases
Stack			
`nop`	–	$\varnothing \to \varnothing$	No operation.
`dup`	–	$a \to a, a$	Duplicates the top stack value.
`pop`	–	$a \to \varnothing$	Discards the top stack value.
Loads / Constants			
`ldarg`	`name`	$\varnothing \to v$	Loads argument by name.
`ldloc`	`name`	$\varnothing \to v$	Loads local by name.
`ldfld`	`Type.Field`	$obj \to v$	Uses instance on stack; falls back to `this` if available.
`ldstr`	`"text"`	$\varnothing \to string$	String literal.
`ldc`	`literal`	$\varnothing \to number$	Integer constant (text form).
`ldi4`	`literal`	$\varnothing \to int32$	Alias of integer constant load.

Mnemonic	Operands	Stack Effect	Notes / Aliases
`ldi8`	`literal`	$\varnothing \to int64$	Alias of 64-bit integer constant load.
`ldr4`	`literal`	$\varnothing \to float$	Alias of 32-bit float constant load.
`ldr8`	`literal`	$\varnothing \to double$	Alias of 64-bit float constant load.
`ldc.r4`	`literal`	$\varnothing \to float$	Float constant.
`ldc.r8`	`literal`	$\varnothing \to double$	Double constant.
`ldtrue`	–	$\varnothing \to true$	Push boolean true.
`ldfalse`	–	$\varnothing \to false$	Push boolean false.
`ldnull`	–	$\varnothing \to null$	Push null.
Stores			
`starg`	`name`	$v \to \varnothing$	Stores to argument by name.
`stloc`	`name`	$v \to \varnothing$	Stores to local by name.
`stfld`	`Type.Field`	$obj, v \to \varnothing$	Pops value then instance; falls back to `this` when needed.
Arithmetic / Unary			
`add`	–	$a, b \to (a + b)$	String concatenation when either operand is string.
`sub`	–	$a, b \to (a - b)$	
`mul`	–	$a, b \to (a \times b)$	
`div`	–	$a, b \to (a/b)$	Division by zero throws.
`rem`	–	$a, b \to (a \bmod b)$	Integer-only.
`neg`	–	$a \to (-a)$	
Comparisons			
`ceq`	–	$a, b \to (a = b)$	
`cne`	–	$a, b \to (a \neq b)$	
`clt`	–	$a, b \to (a < b)$	
`cle`	–	$a, b \to (a \leq b)$	
`cgt`	–	$a, b \to (a > b)$	
`cge`	–	$a, b \to (a \geq b)$	
Control Flow			
`ret`	–	$[v] \to \varnothing$	Returns top of stack (or void if empty).
`br`	`label|index`	$\varnothing \to \varnothing$	Unconditional branch.
`brtrue`	`label|index`	$c \to \varnothing$	Branch if condition is truthy.
`brfalse`	`label|index`	$c \to \varnothing$	Branch if condition is falsey.
`beq`	`label|index`	$a, b \to \varnothing$	Branch if $a = b$.
`bne`	`label|index`	$a, b \to \varnothing$	Branch if $a \neq b$.
`bgt`	`label|index`	$a, b \to \varnothing$	Branch if $a > b$.
`blt`	`label|index`	$a, b \to \varnothing$	Branch if $a < b$.
`bge`	`label|index`	$a, b \to \varnothing$	Branch if $a \geq b$.
`ble`	`label|index`	$a, b \to \varnothing$	Branch if $a \leq b$.
Object / Call			
`newobj`	`Type`	$\varnothing \to obj$	Allocates object instance.
`call`	method ref	$[args] \to [ret]$	Static call.
`callvirt`	method ref	$obj, [args] \to [ret]$	Virtual call; consumes instance.
`castclass`	`Type`	$obj \to obj$	Throws on invalid cast.
`isinst`	`Type`	$obj \to obj \mid null$	Returns null if not instance.
Arrays			
`newarr`	`Type`	$len \to arr$	Allocates array with length from stack.
`ldlen`	–	$arr \to len$	Reads array length.
`ldelem`	–	$arr, idx \to elem$	Loads array element.
`stelem`	–	$arr, idx, val \to \varnothing$	Stores array element.

Mnemonic	Operands	Stack Effect	Notes / Aliases
Structured Control (JSON + IR Text)			
`if`	blocks	*cond* → ∅	Operand provides then/else blocks.
`while`	condition+body	∅ → ∅	Operand provides condition and body.
`break`	–	∅ → ∅	Breaks out of structured loop.
`continue`	–	∅ → ∅	Continues structured loop.
`throw`	–	∅ → ∅	Reserved; currently not implemented.

5.4 Opcode Aliases

The runtime accepts the following alias mnemonics when decoding JSON or text IR:

- `ldcon` and `ldc` are equivalent.
- `ldi4`/`ldi32`/`ldc.i4` map to integer constant load.
- `ldi8`/`ldi64`/`ldc.i8` map to 64-bit integer constant load.
- `ldr4`/`ldc.r4` map to 32-bit float constant load.
- `ldr8`/`ldc.r8` map to 64-bit float constant load.
- `beq.s`, `bne.s`, `bne.un`, `bgt.s`, `bgt.un`, `blt.s`, `blt.un`, `bge.s`, `bge.un`, `ble.s`, `ble.un` are treated as their base comparison branches.

5.5 Operands and JSON Shapes

For JSON IR, instruction operands are encoded as objects. The following shapes are required by the C++ runtime:

- `ldarg`/`starg`: `{ argumentName: "name" }`
- `ldloc`/`stloc`: `{ localName: "name" }`
- `ldstr`/`ldc`: `{ value: <literal>, type: "string|int32|float64|..." }`
- `ldfld`/`stfld`: `{ field: "Type.Field" }` or `{ field: { declaringType, name, type } }`
- `call`/`callvirt`: `{ method: { declaringType, name, parameterTypes: [...], returnType } }`
- `newobj`/`newarr`/`castclass`/`isinst`: `{ type: "Type" }`
- Branches: `{ target: "label" }` or `{ target: 12 }` or `{ offset: 12 }`
- `if`: `{ thenBlock: [instructions], elseBlock: [instructions] }`
- `while`: `{ condition: <Condition>, body: [instructions] }`

5.6 Structured Conditions (JSON)

Conditions are encoded as objects with a `kind`:

- `{ kind: "stack" }`: consumes a boolean-like value from the stack.
- `{ kind: "binary", operation: "ceq|cne|clt|cle|cgt|cge" }`: compares two stack values.
- `{ kind: "expression", expression: <instruction> }`: evaluates a single instruction to a boolean-like value.

6 Notes and Edge Cases

This section records undefined behavior, verifier caveats, and backend-specific constraints.

Json Support for ObjectIR Runtimes are to be removed in a later date as Json is much too inefficient for storing programs, the switch to Fob is recommended if you want small file size or Text IR for everything else.

7 Future Extensions

8 Revision History

Version	Date	Notes
1.0	January 18, 2026	Initial ObjectIR language specification