To represent objects in your intermediate language (IL) effectively while enabling optimizations, consider the following structured approach:

1. Object Representation

- Type Tags: Each object carries a type identifier (e.g., File, String).
- Property Storage: Objects store properties as key-value pairs (or indexed slots for static languages).
- Method Table: Reference to functions/methods associated with the type.

2. IL Instructions for Objects

- Create Object: NEW <Type> [args...]
  Instantiates an object of type <Type> with optional arguments.
- Get/Set Property: GET_PROPERTY <obj_ref> <prop>, SET_PROPERTY <obj_ref> <prop> <value>
  Accesses or modifies an object's property.
- Method Call: CALL_METHOD <obj_ref> <method> [args...]
  Invokes a method on the object.

3. Example IL for Your Code

rust

Copy

```
; Load the 'io' module
LOAD_MODULE "io" -> module_io

; Create File object: File("file.txt")
NEW module_io.File, "file.txt" -> file
STORE_GLOBAL "file", file

; Access file.text
LOAD_GLOBAL "file" -> file_ref
GET_PROPERTY file_ref, "text" -> content
STORE_GLOBAL "content", content

; Print content
LOAD_GLOBAL "content" -> content_ref
CALL_PRINT content_ref
```

4. Optimization-Friendly Design

- Static Single Assignment (SSA): Use SSA form for variables to simplify data flow analysis.
- Explicit Type Annotations: Allow optional type hints in the IL (e.g., NEW File:io.File).
- Aggregate Initialization: Combine object creation and property setup in one instruction where possible.

5. Key Decisions

- Dynamic vs. Static: For dynamic scripting, use string-based property names (like GET_PROPERTY file_ref, "text"). For static typing, use indexed slots (e.g., GET_PROPERTY_IDX file_ref, 0).
- Memory Model: Decide if objects are heap-allocated (with garbage collection) or stack-managed.

6. Why This Works

- Optimization Opportunities: Explicit object operations let you eliminate redundant creations, inline method calls, or hoist property accesses.
- Portability: The IL abstracts implementation details, allowing future compilation to bytecode or machine code.

Final Pseudo-Code for Your IL

rust

Copy

```rust
; Module system
get_module "io" -> io

; Object creation
new io.File, args=["file.txt"] -> tmp1
set_global "file", tmp1

; Property access
get_global "file" -> tmp2
get_property tmp2, "text" -> tmp3
set_global "content", tmp3

; Function call
get_global "print" -> tmp4
get_global "content" -> tmp5
call tmp4, args=[tmp5]
```

This design reduces "trips" by making object operations explicit and concise, while leaving room for optimizations like constant propagation or dead store elimination.