We will be on the VM today. Go ahead and pull.

$ sudo apt update
$ sudo apt install openjdk-11-jdk-headless

Then open a 2nd terminal while that's installing.

After install – open today's lecture and read the files in the c directory and explain what you expect the output to be with a neighbor.

This installs an open source implementation of Java.

Compile C to Object Code File

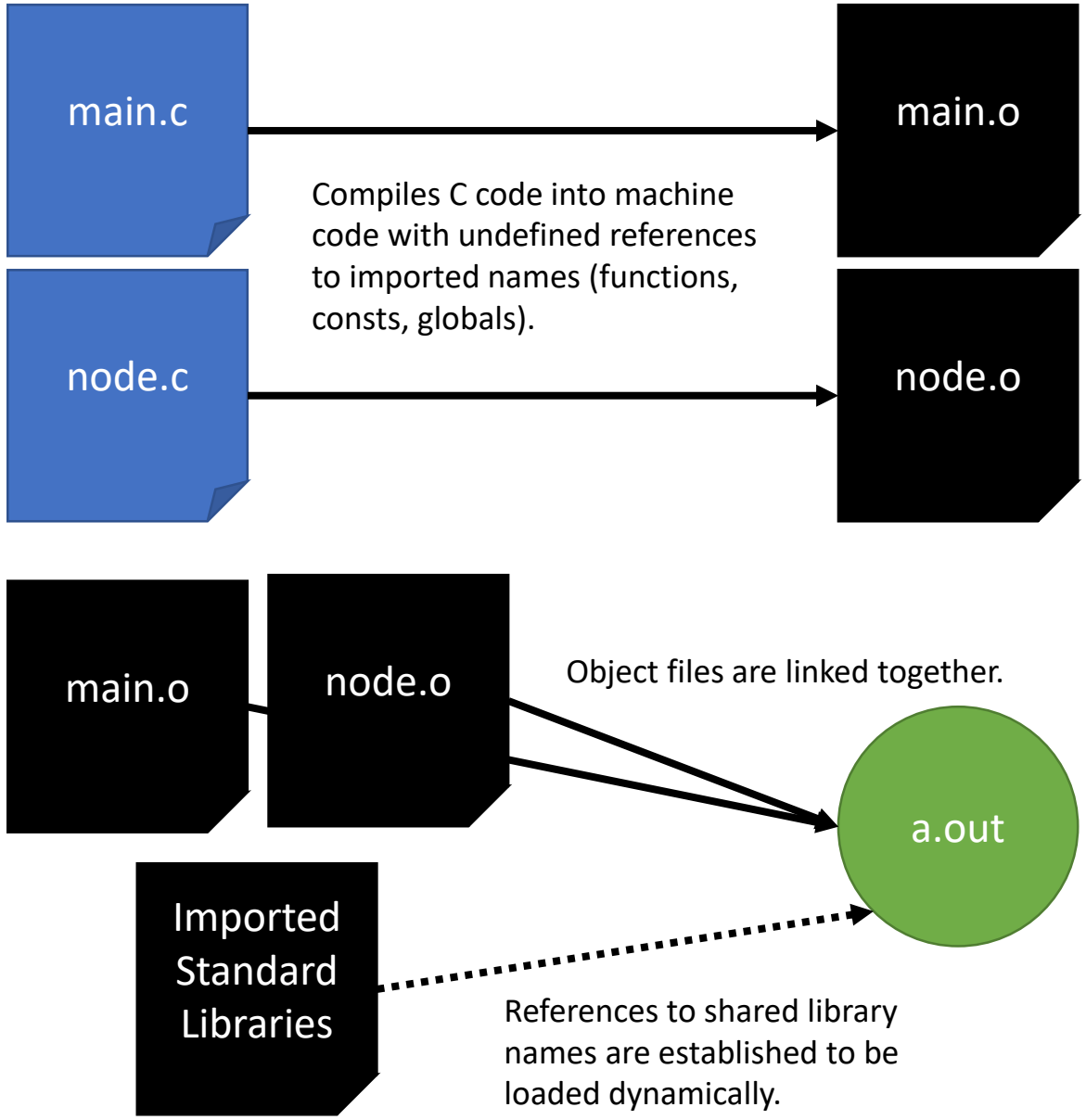
- Generate the "object" code – binary machine code – of a single C file
  ```
  gcc -c <filename>.c
  ```
- Print the "names" in the object code with **nm**
  ```
  nm –n <filename>.o
  ```
- You can disassemble the machine code into assembly code with objdump
  ```
  objdump -D <filename>.o
  ```
- To compile multiple object files into an executable program:
  ```
  gcc <file1.o> <file2.o>
  ```
- Run the executable
  ```
  ./a.out
  ```

C's Object Files and Executable Machine Code

```
$ cc -c main.c
```

```
$ cc -c node.c
```

```
$ cc main.o node.o
```



Inspecting the locations of *names* in object files using **nm**

```
$ nm -n node.o | grep ' T '
0000000000000000 T cons
0000000000000038 T first
000000000000007f T rest
00000000000000c8 T destroy
```

The address of the definition in the object file.

The "T" means the name is found in the text (think: code) segment of the binary file.

The symbol being defined.

```
$ nm -n main.o | grep ' T '
0000000000000000 T main
0000000000000080 T print_list
00000000000000e2 T sum
```

Notice the addresses of the functions in these files overlap with one another.

If we want to join these two files a single program, at least one set of functions will need to be *relocated.*

```
$ nm -n main.o | grep ' U '
                 U _GLOBAL_OFFSET_TABLE_
                 U puts
                 U cons
                 U printf
                 U first
                 U rest
                 U putchar
```

If instead we filter main for its *undefined* names, notice there are references to *cons, first,* and *rest.*

There are also references to library functions (some we didn't even call directly!)

The Results of Linking Object Files into an Executable Binary

```
$ nm -n main.o | grep ' T '
0000000000000000 T main
0000000000000080 T print_list
00000000000000e2 T sum
```

```
$ nm -n node.o | grep ' T '
0000000000000000 T cons
0000000000000038 T first
000000000000007f T rest
00000000000000c8 T destroy
```

```
$ nm -n a.out | grep ' T '
0000000000000680 T _init
0000000000000730 T _start
000000000000083a T main
00000000000008ba T print_list
000000000000091c T sum
000000000000095c T cons
0000000000000994 T first
00000000000009db T rest
0000000000000a24 T destroy
0000000000000a70 T __libc_csu_init
0000000000000ae0 T __libc_csu_fini
0000000000000ae4 T _fini
```

Notice in the executable binary, the code from both object files is added in but has been relocated.

If the location of a function definition changes, what also must change? All calls to it are updated by linker, too!

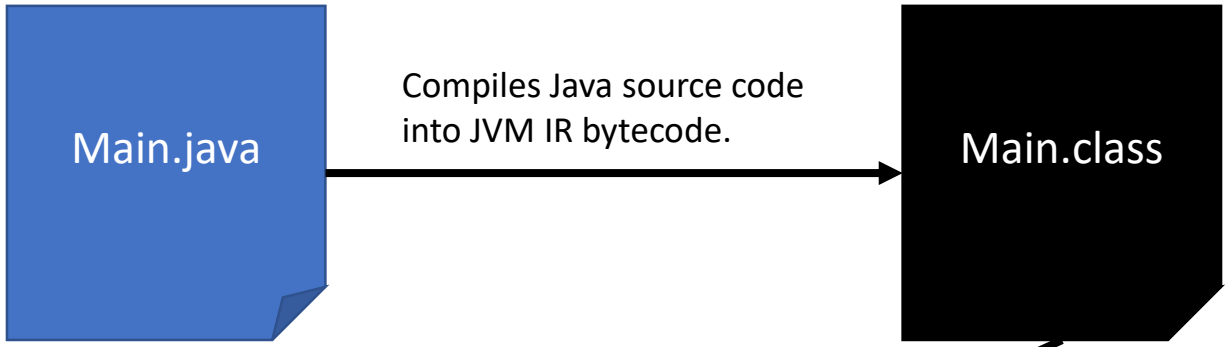
Additional symbols are the front matter and back matter to conventionally start up and wind down the program and any additional library dependencies (i.e. libc here)

Compiling Java to IR Bytecode

- javac <ClassName>.java
- Produces <ClassName>.class
- This is an Intermediate Representation (IR) called bytecode the JVM runs.
- You can view "disassembled" bytecode with **javap -c <ClassName>.class**
- To run the Java program, you indicate the class containing main: **java <ClassName>**
  - Java Virtual Machine reads this code and interprets it or compiles to machine code Just-in-Time
  - Class loader responsible for loading referenced outside classes

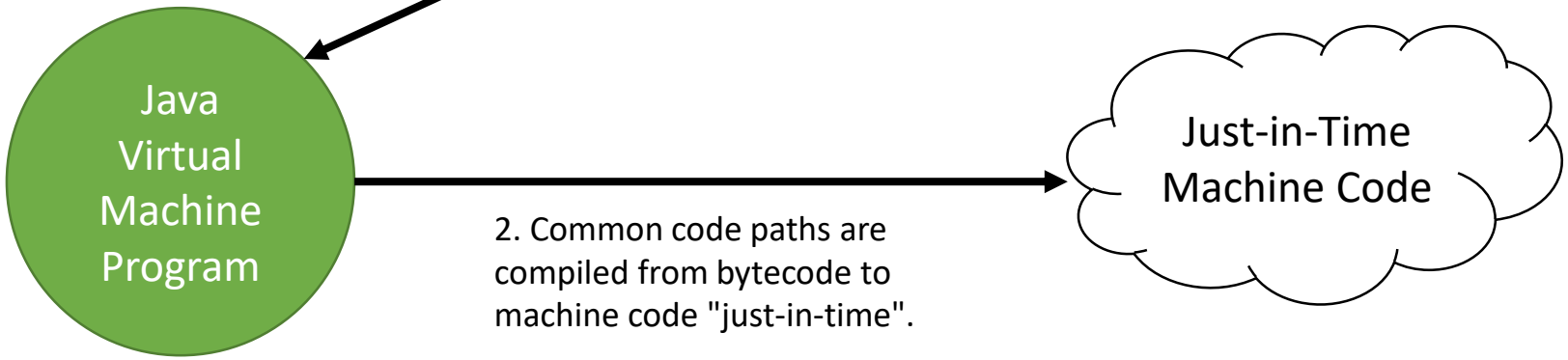
Java's Intermediate Representation Model

```
$ javac Main.java
```



1. Reads the IR bytecode and interprets its instructions.

```
$ java Main
```



2. Common code paths are compiled from bytecode to machine code "just-in-time".

Trade-offs of IR Bytecode vs. Machine Code

	Pros	Cons
Machine Code	Loads directly into memory and CPU can execute at raw speed. Raw access to low-level machinery. Simpler, smaller program to "deploy" because less dependent on giant libraries and runtimes.	Not portable across operating systems nor CPU architectures. Dynamic library loading is fragile based on system settings and very operating system dependent.
IR Bytecode	Portable to any architecture with a Virtual Machine to execute the bytecode. Long-running programs can be compiled to machine code JIT and not lose *as much* performance. Virtual machine offers convenient features (i.e. garbage collection).	Slower to start a program (by many orders of magnitude). Slower to execute a program (by how much depends on whether interpreted or JITed). Overhead and fundamental limitations of virtual machine runtime features.

Compiling Rust to a Shared Object File

- Start Rust project with `--lib` option in Cargo

- Add the following to `Cargo.toml`

```
[lib]
name = "hello"
crate-type = ["staticlib", "cdylib"]
```

- Now, when we build a "libhello.so" file is produced in the target dir

Compiling C that relies on a Rust library...

- cd into the Rust library

- gcc main.c -L. -l:./libhello.so
  - The -L flag says adds the current directory to the library file search path
  - The -l flag specifically links the library file ./libhello.so

- Now try running ./a.out

Writing Python that relies on a Rust library…

- cd into the python library
- Copy libhello.so into this directory
- Checkout ffi.py to see how the linkage is made in Python
- Try running python ffi.py