How to Develop a Memory Diagram

A *memory diagram* shows the contents of two areas of memory that a programming environment maintains as part of running code.

- the *heap* stores objects (or other structured data) in memory locations/addresses
- the *environment* associates each variable name with a heap location or basic datum (e.g., int, bool)

Working with memory diagrams can help us understand how a program is—or should be—running. They should also help you develop good conceptual models about how language constructs work.

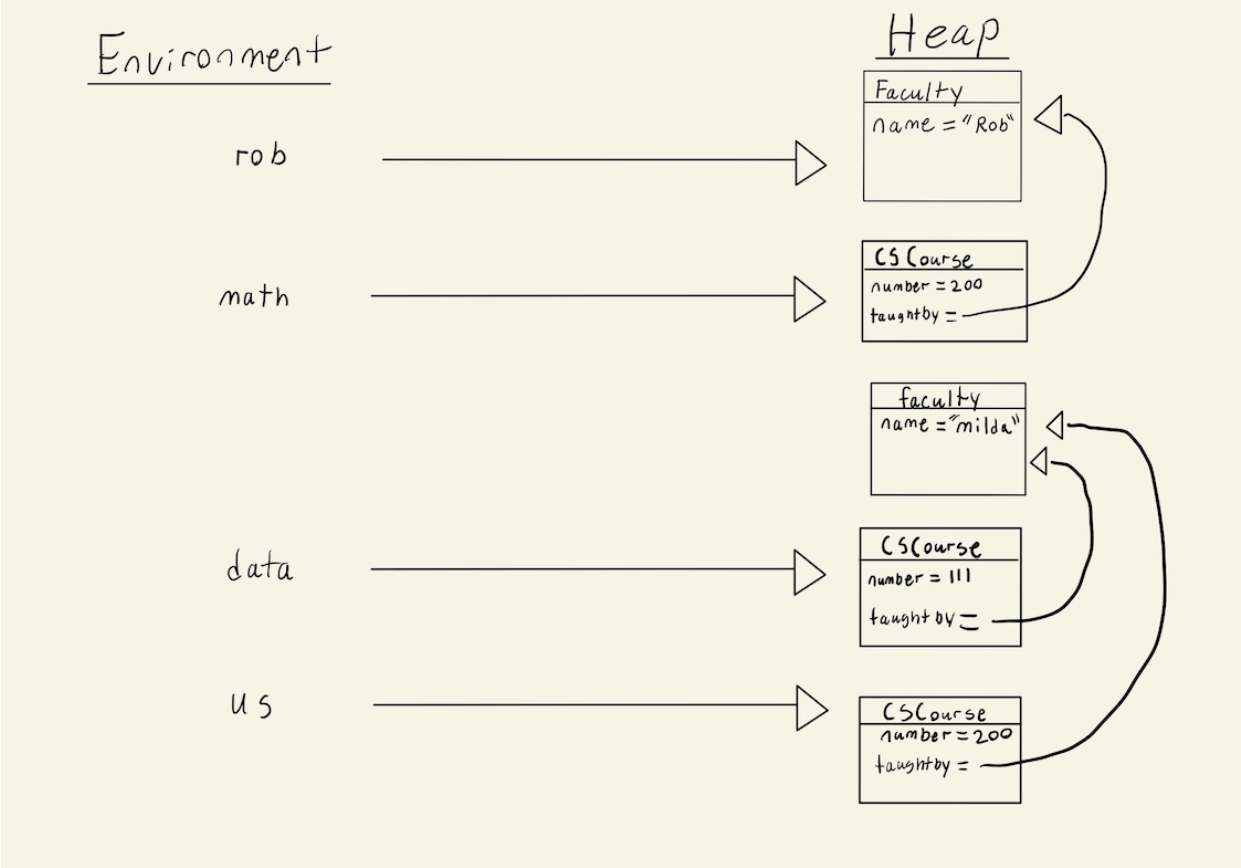
We create memory diagrams at two different levels of detail, depending on context. The first level uses hand-drawn arrows to show references between objects. The second level uses addresses to represent which object a field or name refers to.

Sample Memory Diagram

Consider this sample of code:

```
01|  class CSCourse {
02|    private int number;
03|    private Faculty taughtBy;
04|  }
05|
06|  class Faculty {
07|    private String name;
08|  }
09|
10|  Faculty rob = new Faculty("Rob");
11|  CSCourse math = new CSCourse(220, rob);
12|  CSCourse data = new CSCourse(111, new Faculty("Milda"));
13|  CSCourse us = new CSCourse(200, data.getTaughtBy());
```

Here is the diagram with arrows (not addresses):



Here is the diagram with addresses (not arrows):

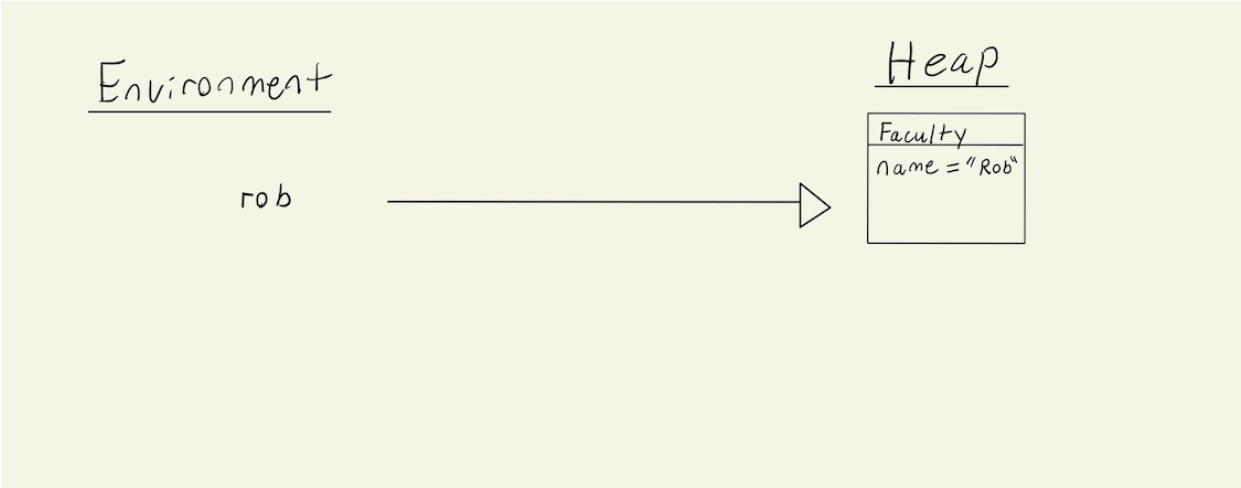
Environment	**Heap**
rob -> loc1010	loc1010 -> Faculty("rob")
math -> loc1011	loc1011 -> CSCourse(220, loc1010)
data -> loc1013	loc1012 -> Faculty("Milda")
us -> loc1014	loc1013 -> CSCourse(111, loc1012)
	loc1014 -> CSCourse(200, loc1012)

Breaking Down The Memory Diagram

Now, let's break down the code line by line and see what the memory looks like at each stage.

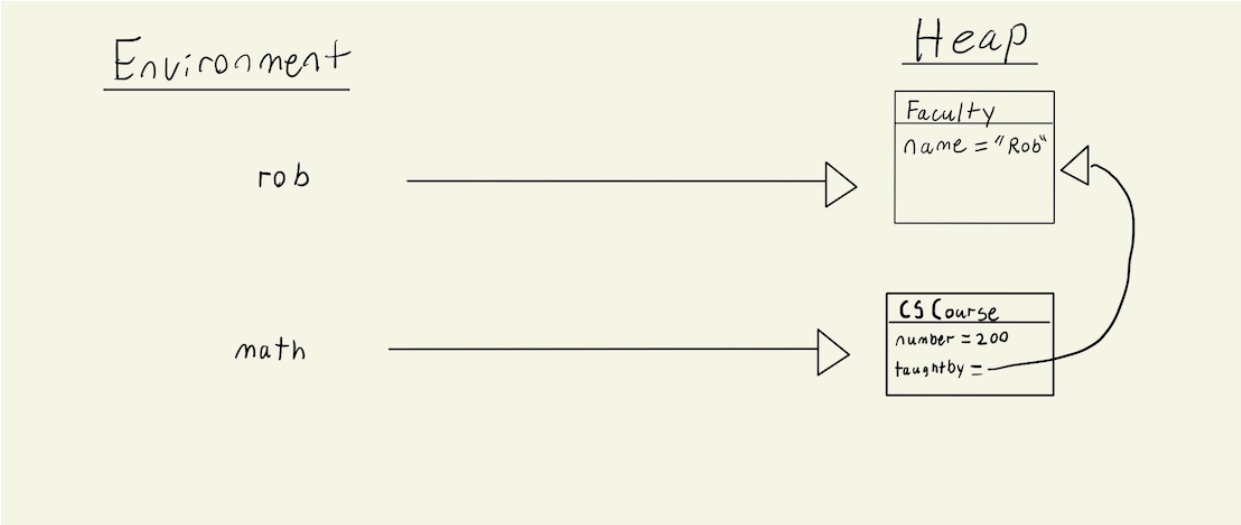
1. `Faculty rob = new Faculty("Rob");`

This code creates a variable within the environment called 'rob' which points to a new faculty object stored in the heap:



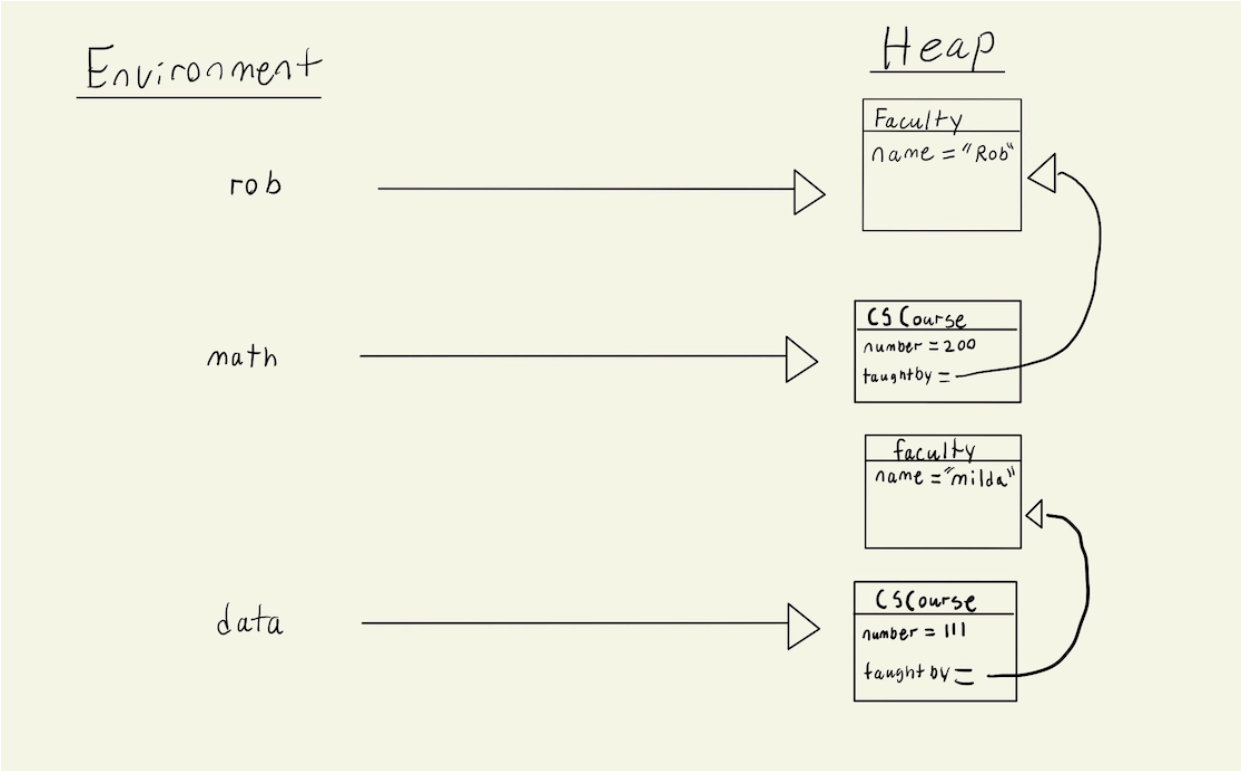
2. `CSCourse math = new CSCourse(220, rob);`

This line creates a new variable in the environment called ‘math’ which points at a new CSCourse object located within the heap. This new CSCourse object points to the faculty object for its ‘taughtby’ attribute:



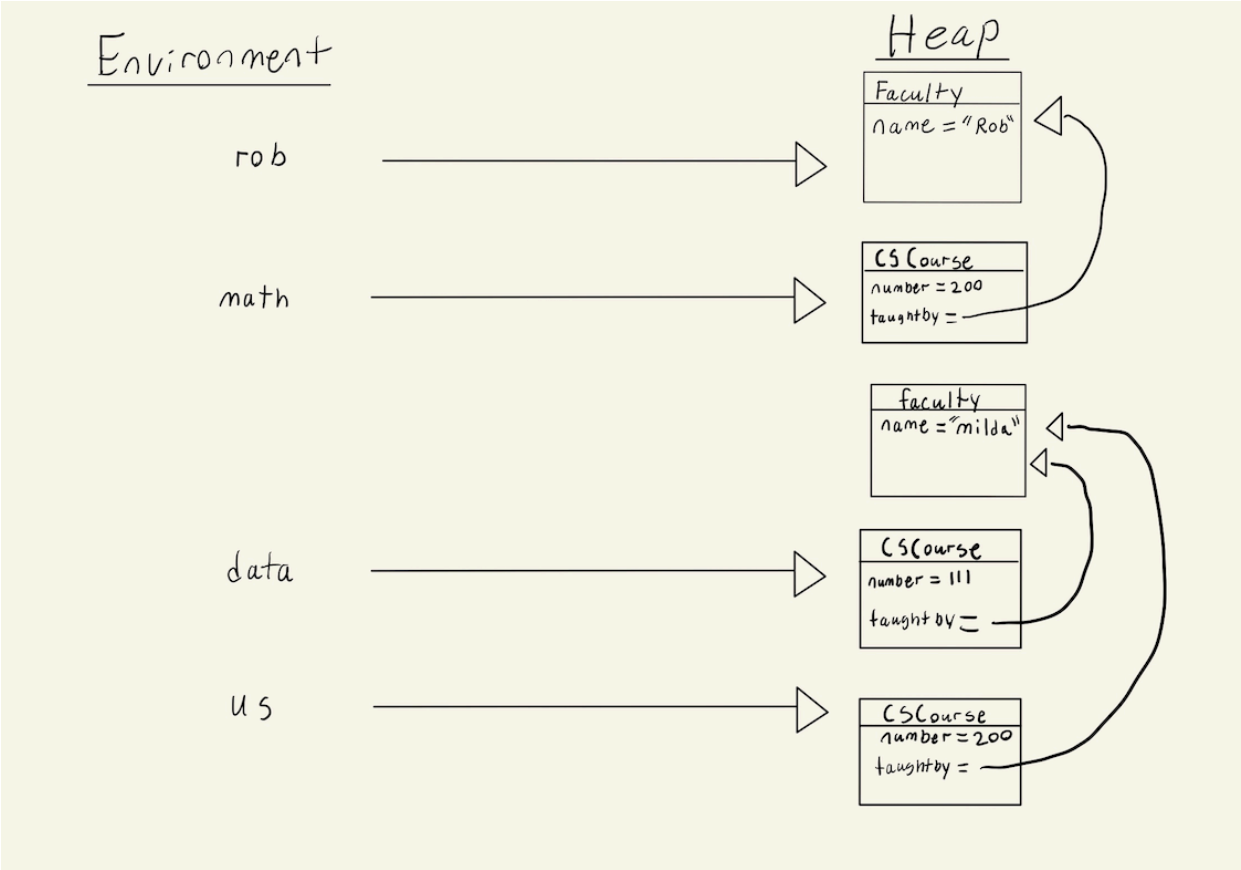
3. `CSCourse data = new CSCourse(111, new Faculty("Milda"));`

This line starts by creating a new `Faculty` object that isn’t pointed to by any variable. This object is stored within the new CSCourse’s ‘taughtby’ attribute. This CSCourse can be accessed through the ‘data’ variable.



4. `CSCourse us = new CSCourse(200, data.getTaughtBy());`

Now another new CSCourse object is created (accessible through the ‘us’ variable). This new course has 200 as its ‘number’ attribute and points to the same heap object as data’s ‘taughtby’ attribute.



Creating a Memory Diagram from Code

We create memory diagrams by manually tracing the steps of the code, in order, and taking the following steps:

- Whenever your code **creates an object** (with new), put the object in the heap (at the next unused heap location)
    - If the object is an ArrayList, set aside 8 locations for its elements (even if some are unused)
- Whenever your code **introduces a new name**, such as a local variable, constant, or method parameter, put the name in the environment. Associate the name with the heap location of the object (or the basic value) that the name refers to.
- Whenever your code **calls a method or function**, box off a local segment of the environment. Parameters and local variables go into this local segment. When the method/function call finishes, delete the local segment from the environment (the heap is not affected)
- Whenever your code **assigns a new value to a field**, change the field contents in the *heap*.

- Whenever your code **assigns a new value to a name**, change the *environment* to associate the name with the heap location or basic data of its new value
- Whenever your code **uses a variable name or field name** start from the environment and follow the references until you get to a basic value or location address. That value/address is the result of that expression.

FAQ

- **Do objects ever leave the heap?** Language implementations use a process called *garbage collection* to remove objects from the heap when they are no longer being used. When you draw memory diagrams, ignore garbage collection and assume objects live in the heap forever. We will do a lecture on garbage collection towards the end of the course.
- **Do names ever leave the environment?** Names that are introduced by methods (parameters, local variables) leave the environment when the method call has finished.
- **Do the location numbers mean anything?** Yes. The order of the location numbers indicates the order in which objects were created while running your program (this can change with garbage collection, but we'll ignore this for now). The gaps between location numbers also correspond to the size of the object, in terms of the number of fields (or reserved slots in the case of ArrayList).
- **Can we ever run out of location numbers?** Technically yes, because a computer has only a limited amount of memory (and only some of that memory can be used for the heap for any one program). We're not going to ask you to trace anything large enough for this to be an issue.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS200 document by filling out the anonymous feedback form! (you do have to sign in but we don't see it)