Lists1 Study Guide | CS 61B Spring 2018

sp18.datastructur.es/materials/lectures/lec3/lec3.html

Lecture Code

Code from this lecture available at https://github.com/Berkeley-CS61B/lectureCode-sp18/tree/master/lists1.

Overview

Bits The computer stores information as memory, and represents this information using sequences of bits, which are either 0 or 1.

Primitives Primitives are representations of information. There are 8 primitive types in Java: byte, short, int, long, float, double, boolean, and char. Each primitive is represented by a certain number of bits. For example, ints are 32 bit primitives, while bytes are 8 bit primitives.

Declaring Primitives When we declare a variable to be a primitive (i.e. `int x;`), we set aside enough memory space to hold the bits (in this case, 32). We can think of this as a box holding the bits. Java then maps the variable name to this box. Say we have a line of code `int y = x;` where `x` was defined before. Java will copy the bits inside the `x` box into the bits in the `y` box.

Creating Objects When we create an instance of a class using the `new` keyword, Java creates boxes of bits for each field, where the size of each box is defined by the type of each field. For example, if a Walrus object has an `int` variable and a `double` variable, then Java will allocate two boxes totaling 96 bits (32 + 64) to hold both variables. These will be set to a default value like 0. The constructor then comes in and fills in these bits to their appropriate values. The return value of the constructor will return the location in memory where the boxes live, usually an address of 64 bits. This address can then be stored in a variable with a "reference type."

Reference Types If a variable is not a primitive type, then it is a reference type. When we declare object variables, we use reference type variables to store the location in memory of where an object is located. Remember this is what the constructor returns. A reference type is always a box of size 64 bits. Note that the variable does not store the entire object itself!

Golden Rule of Equals For primitives, the line `int y = x` copies the bits inside the `x` box into the `y` box. For reference types, we do the exact same thing. In the line `Walrus newWalrus = oldWalrus;`, we copy the 64 bit address in the `oldWalrus` box into the `newWalrus` box. So we can think of this golden rule of equals (GroE) as: when we assign a value with equals, we are just copying the bits from one memory box to another!

Parameter Passing Say we have a method `average(double a, double b)`. This method takes two doubles as parameters. Parameter passing also follows the GRoE, i.e. when we call this method and pass in two doubles, we copy the bits from those variables into the parameter variables.

Array Instantiation. Arrays are also Objects, and are also instantiated using the `new` keyword. This means declaring an array variable (i.e. `int[] x;`) will create a 64-bit reference type variable that will hold the location of this array. Of course, right now, this box contains the value null, as we have not created the array yet. The `new` keyword for arrays will create the array and return the location of this array in memory. So by saying `int[] x = new int[]{0, 1, 2, 3, 4};`, we set the location of this newly created array to the variable x. Note that the size of the array was specified when the array was created, and cannot be changed!

IntLists. Using references, we recursively defined the `IntList` class. `IntLists` are lists of integers that can change size (unlike arrays), and store an arbitrarily large number of integers. Writing a `size` helper method can be done with either recursion or iteration.

Exercises

C Level

1. Complete the exercises from the online textbook.
2. If doubles are more versatile than ints, why don't we always use them? Are there any disadvantages to doing this?
3. How much does the memory cost differ between the storing of an address of a 32 entry int array and a 300 entry int array?

B Level

1. Rewrite the `size`, `iterativeSize`, and `get` methods from lecture by using this the starter code for the IntList class.
2. Write methods `incrList` and `dincrList` as described in Lists1Exercises. If your solution uses `size`, `iterativeSize`, or `get`, you'll need to complete the previous exercise first.