Lecture 7: Linked Lists

Sam McCauley
Data Structures, Spring 2026

How Variables and Objects are Stored in Java

Before we talk about more advanced data structures, we need to address some specifics of how Java stores data. We'll talk about how Java stores primitive types, and how it stores objects.

Memory in your computer is made up of 1s and 0s, called "bits"; these bits are grouped into sets of 8 which are called "bytes." All of the data you store in your program, be it a variable in a loop, a large array, an object of a user-defined class, or even input obtained from the user, is stored using these 1s and 0s. The goal for today is to discuss more about what is stored, and how to use that to help us write good programs.

Storing Primitive Types. Remember that a "primitive type" in Java is something like `int`, `double`, `char`, or `boolean`. This is as opposed to objects, like the `Student` objects we declared in previous lectures. Note that `String` is *not* a primitive type in Java.

When you declare a variable, Java allocates space for that variable. For example, an `int` is always 32 bits, and stores the normal binary representation of the number. For example, the number 2 is 10 in binary, so when we write

```
1  int x = 2;
```

Java creates a section of memory that looks like this:

$$\underbrace{0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0}_{x}$$

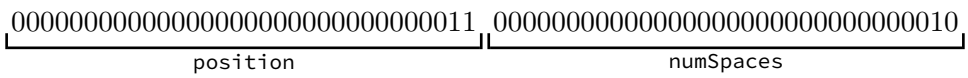
Simple enough: each time we read x, Java reads these 32 bits to get its current value. Each time we write to x, Java changes the 32 bits to match the new value.

This may give some intuition about why we need to declare and initialize variables in Java. When we declare the variable, Java finds the bits in memory—it creates the label "x". When we initialize it, we set those bits to the value we want. We can't use the variable until both of these steps are complete.

Storing Instance Variables. When instance variables are primitive types, Java works largely the same way. Consider our `Move` class from Lab 2. The class declaration had the following instance variables:

```
1 public class Move {
2     private int position;
3     private int numSpaces;
4 }
```

For each `Move` object, Java just stores each of these `int`s next to each other[1] in memory. Assume that `position` is 3 and `numSpaces` is 2; their data will be stored something like the following.



Instance Variables: Default Values. Interestingly, instance variables in Java do not need to be initialized: Java automatically sets them to a default value. The default value of an `int` is 0, and the default value of a `boolean` is `false`.

You *should not* use default values intentionally: you should always set the value of all instance variables in the constructor. However, it is important that you know they exist, as this knowledge is useful for debugging. Specifically: if you forget to initialize an instance variable, you will not get an error; Java will simply use the default value. You should keep an eye out for this as a possible way to explain unexpected behavior in your programs.

Storing Objects. Objects are *not* stored in the same way.

When we create an object variable, what it actually stores is the address of the relevant information—by "address," I mean it stores where to find it in memory. Let's draw this as an arrow when we draw memory. You can think of this address as storing *where* to find the actual data.

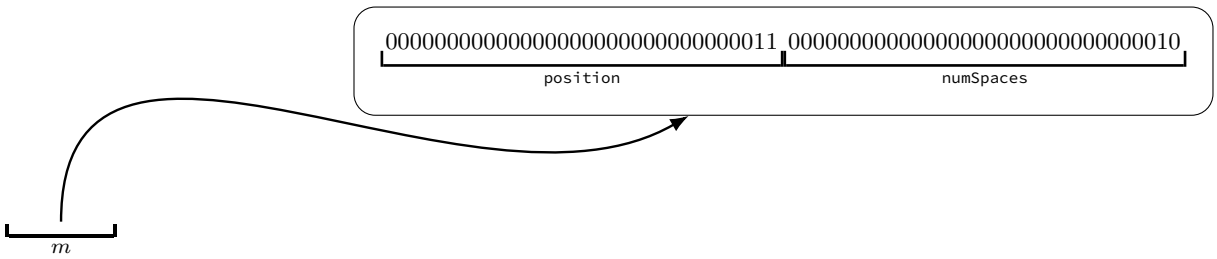
This address is called a *reference*. (These are sometimes called "pointers" in other languages.)

Let's draw a picture of what an object looks like. Again, let's say that we have a `Move` with `position` 3 and `numSpaces` 2. When we write the following code

```
1 Move m = new Move(3, 2);
```

what we actually store in memory is the following.

[1] We're oversimplifying here: the actual memory layout may not exactly match this description. However, this is sufficient for our purposes today.



Bear in mind that the arrow is purely a visual aid: what m actually stores is a reference: a binary number storing the address the data.

Declaring and Instantiating an Object in More Detail. We can now discuss how declaring and instantiating an object works in more detail. Let's go step by step through each piece of the following line of code.

```
1  Move m = new Move(3, 2);
```

First, `Move m` creates a space to hold the *reference* stored by `m`: Java allocates enough bits to store this address in memory.

The keyword `new` is responsible for allocating memory. It creates the rectangle on the right in the diagram, with enough space to hold all of the instance variables for the object: in this case, `position` and `numSpaces`. It sets these variables to their default values (see the discussion above).

The constructor of `Move` is called. In this case, the constructor sets `position` and `numSpaces` equal to -1 (see `Move.java`).

Finally, the value of `m` is set equal to the address of the newly-created `Move`, and we've arrived at the diagram above.

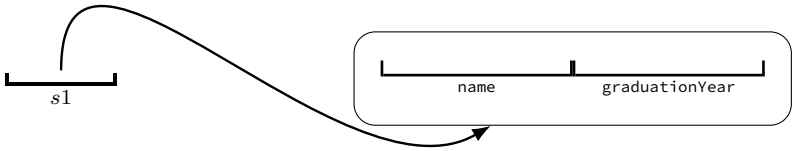
Accessing Data Stored Using References Consider for a moment a class with a `public` instance variable. In a recent lecture, we had a `Student` class that looked like the following.

```
1  public String name;
2  public int graduationYear;
```

Let's go through each line of the following code.

```
1  Student s1 = new Student();
2  s1.graduationYear = 2026;
3  System.out.println(s1.graduationYear);
```

We already went through line 1: `s1` contains a reference that gives the address of the data in a new `Student` object. The state looks something like this (I won't draw the 1s and 0s in the instance variables from now on—they're still there; we should just ignore them to avoid the diagrams getting too busy).



In Line 2, Java looks at the reference stored in `s1` and follows it. It arrives at the collection of instance variables; then, it finds the bits assigned to the instance variable `graduationYear`. It sets those bits to store 2026 in binary.

Line 3 is largely like Line 2: Java follows the reference from `s1` to the instance variables, finds the bits for `graduationYear`, and proceeds with the `println()` method from there, ultimately printing the value `2026` to the screen.

The `null` Keyword. In Java, there is a special reference value, called `null`, that represents that no memory address is stored. This is the default value for any object.

You will sometimes see Java give a "null pointer exception." This means that it attempted to get a value from an address—but the address was null.

Consider the following code.

```
1 Student s1 = null;  //OK
2 s1.graduationYear = 2029; //Null pointer exception
```

As seen immediately above, it is legal to set an object to `null` explicitly. We can also test if the address is `null`.

```
1 if(s1 != null) {  //do not access s1 if null is stored
2     s1.graduationYear = 2029;
3 }
```

What it Looks Like When an Object is an Instance Variable. Most instance variables we've seen so far have been primitive types, but it is also possible for instance variables to be objects. References are crucial to this behavior.

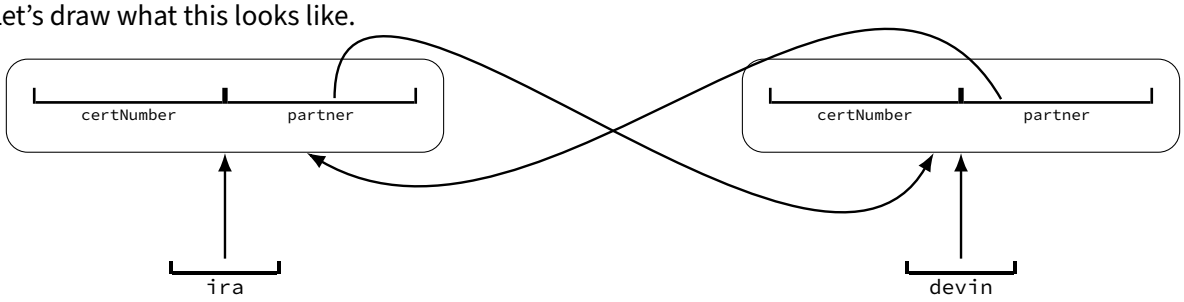
Let's say we want a class to help keep track of EMTs. Each EMT has a certification number. Also, each EMT has a partner who they work with. We'll keep these public for a moment so we can work with them without getters/setters.

```java
public class EMT {
    public int certNumber;
    public EMT partner;
}
```

Each EMT keeps track of the partner they work with. Let's say that Devin and Ira are partners: Ira is Devin's partner, and Devin is Ira's partner.

The key insight here is that these two objects *point to* each other. It's similar to Ira having a piece of paper that has Devin's name on it (and vice-versa)—but instead of a name, what's actually stored is a memory address. It's not that one partner is "stored inside" the other: they're just keeping a note of who the other is.

Let's draw what this looks like.



The `ira` object stores a reference to its data. Within the data, there is a reference to the data for Ira's partner. The same idea holds for the `devin` object.

We can do some interesting things with this. To get the `certNumber` of Ira's partner, we can write:

```java
int irasPartnerCert = ira.partner.certNumber;
```

Java evaluates this expression left to right. It goes to `ira`, and follows the reference to Ira's data; then, it finds the reference stored in `partner`, and follows that reference to Ira's partner's data (also known as Devin's data); then, it finds the `certNumber`, and stores that value in `irasPartnerCert`.

If we were to use getters and setters and private variables, the same idea would still hold. Let's say the getters were called `getCertNumber()` and `getPartner()`. Then the above line of code would become:

```java
int irasPartnerCert = ira.getPartner().getCertNumber();
```

A New Strategy For Storing a List of Items

Let's begin by explaining the strategy using a word problem that does not involve computers. Then, we'll talk about how the data structure works, and how to implement it in Java.

Let's say that in our CS 136 class we want to remember a sequence of 15 numbers. This is hard for an individual, but very doable as a group. First, let's start with a strategy similar to an array. We'll have the student in the first seat memorize the first number; the second seat memorize the second number, and so on. If we want to recover the sequence, we just go seat by seat and have the student recite their number.

Now, let's add a wrinkle to the problem. The students may not sit in the same seats each time they come into class. How can we make our method resilient to students swapping seats?

The answer is that each student can remember not just their number, but also which student has the next number in the sequence. Then all we have to do to recover the sequence is to remember the first student to ask. We ask them both for their number, and for the next student. We go to that student and repeat, and keep going until a student says that there is no such number.

Linked List. The strategy above is the basic idea behind the new data structure we will see today, the "Linked List." (Specifically, a Singly Linked List—we'll go over what we mean by that next class.)

Our linked list will store a sequence of integers. In fact, it will be able to perform exactly the same methods as our ArrayList implementation. The difference will be in performance: the Linked List will be faster for some methods, and slower for others. As with the ArrayList, we'll generalize our implementation to handle sequences of types other than integers later this week.

Linked List Nodes. Our linked list consists of "Nodes." Each node has two jobs: first, to remember an integer; second, to remember the next node in the list.

This class looks like this. In short, it has two instance variables, each with a getter and setter. We'll also include a short constructor.

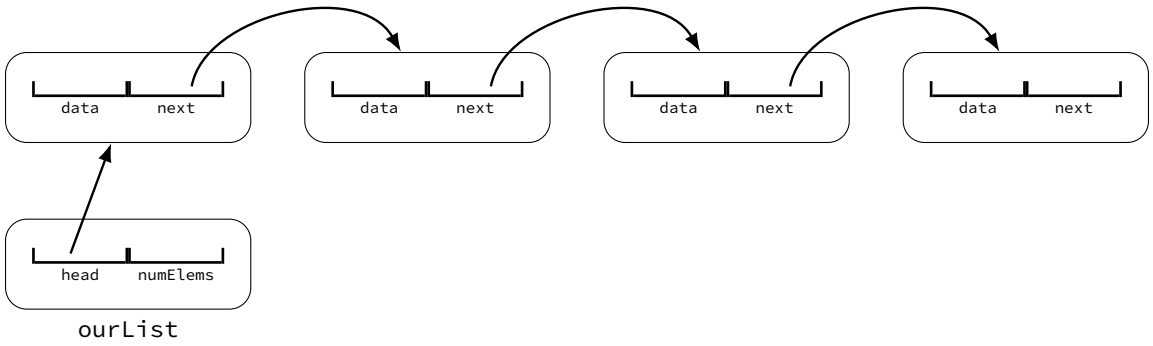
```
1  public class IntNode {
2      private int data;
3      private IntNode next;
4
5      public int getData() {
6          return data;
7      }
8      public IntNode getNext() {
9          return next;
10     }
11     public void setData(int newValue) {
12         data = newValue;
13     }
14     public void setNext(IntNode newNext) {
15         next = newNext;
16     }
```

```
17      public IntNode() {
18          data = 0;
19          next = null;
20      }
21  }
```

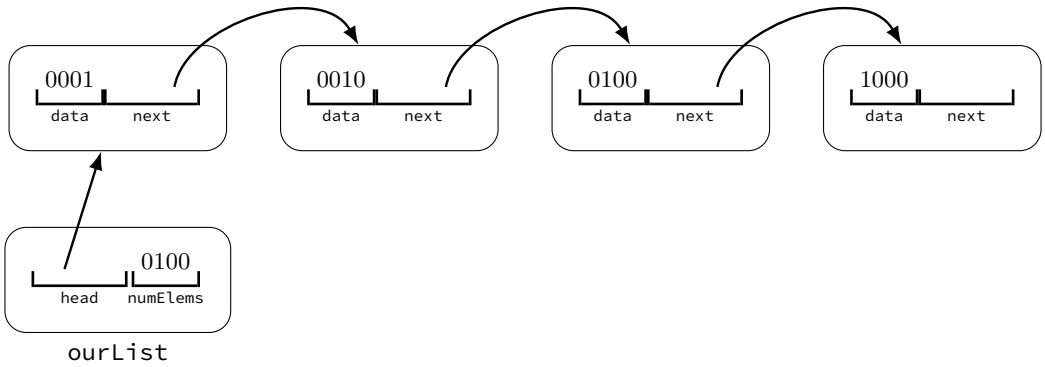
We are ready to make our Linked List. The only thing it needs to store is the first node in the list. This node is called the "head" of the list. For convenience, we'll also store the number of elements in the list in a separate variable. With these instance variables in mind, we can write the first few lines of our class.

```
1  public class IntLinkedList {
2      private IntNode head;
3      private int numElems;
```

IntLinkedList Overview Let's review our plan. Our `IntLinkedList` class will keep a reference to an `IntNode` containing the first element of the list. Each `IntNode` in the list will store a reference to the next element of the list. Overall, our strategy looks something like the following. Assume that we have an `IntLinkedList` named `ourList`.



The `data` fields of the four consecutive nodes store the values of the list. So if our list stores the values 1, 2, 4, 8, the linked list will look something like the following. (The `int`s are truncated to 4 bits for space.)



IntArrayList Methods. Now, let's fill in all of the methods we did for `IntArrayList`.

`int size()` returns the number of elements stored in our list.

```
public int size() {
    return numElems;
}
```

As in the `ArrayList`, many methods require accessing a specific element of the list. Let's write a simple method to handle error checking, to prevent duplicate code.

```
private void checkInBounds(int index) {
    if(index < 0 || index >= numElems) {
        System.out.println("Error: List access out of bounds at index "
            + index);
        System.exit(1);  //exit the program
    }
}
```

`int get(int index)` returns element `index` stored in the linked list. We'll start by accessing the Node stored in `head`; its value is correct for index 0. Then, we'll go to the `next` element of that Node, then its next, etc., `index` times.

```
public int get(int index) {
    checkInBounds(index);
    IntNode current = head;
    for(int x = 0; x < index; x++) {
        //current is now the xth node in the list
        current = current.getNext();
    }
    return current.getData();
}
```

`int set(int index, int newElement)` sets the element stored at `index` to be `newElement`, and returns the element *previously* stored at index `i`. We'll accomplish this similarly: scan through the list until the ith node, but now we set its value rather than getting it.

```
public int set(int index, int newElement) {
    checkInBounds(index);
    IntNode current = head;
    for(int x = 0; x < index; x++) {
        //current is now the xth node in the list
        current = current.getNext();
    }
    int oldValue = current.getData();
    current.setData(newElement);
    return oldValue;
}
```

Alarm bells are going off—we have duplicate code in these two methods! The `for` loop to find the `index`-th Node in the list is common to both methods. Let's factor it out into a new method that does exactly this operation. It seems likely that this will be useful for us in the future as well. This is a helper method, which we don't want users to access (we don't want them accessing or modifying `intNode` objects) so we'll set it to `private`.

```
1  private IntNode getNode(int index) {
2      IntNode current = head;
3      for(int x = 0; x < index; x++) {
4          //current is now the xth node in the list
5          current = current.getNext();
6      }
7      return current;
8  }
```

Now, let's refactor `set` and `get`.

```
1  public int get(int index) {
2      checkInBounds(index);
3      return getNode(index).getData();
4  }
5
6  public int set(int index, int newElement) {
7      checkInBounds(index);
8      IntNode indexNode = getNode(index);
9      int oldValue = indexNode.getData();
10     indexNode.setData(newElement);
11     return oldValue;
12 }
```

`void add(int newElement)` adds `newElement` to the end of the list. To make a new element, we'll need a new place to store it: a new `IntNode`. Then, we need to add it to the end of the list. To do that, we first need to find the end of the current list. Finally, we need to "join" our new node to the end of the list: the `next` value of the node at the end of the list should be the new node we made.

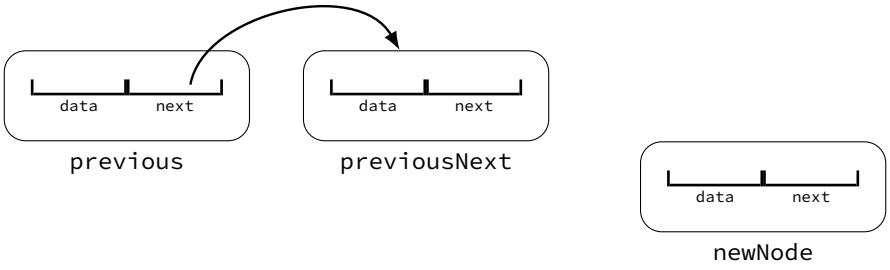
```
1  public void add(int newElement) {
2      IntNode newNode = new IntNode();
3      newNode.setData(newElement);
4      if(numElems == 0) {
5          head = newNode;
6      } else {
7          IntNode lastNode = getNode(numElems - 1); //find the last node
8          lastNode.setNext(newNode);  //set it to point to our new node
9      }
10     numElems++;
11 }
```

`void add(int index, int newElement)`: adds `newElement` to index `index` in the list,

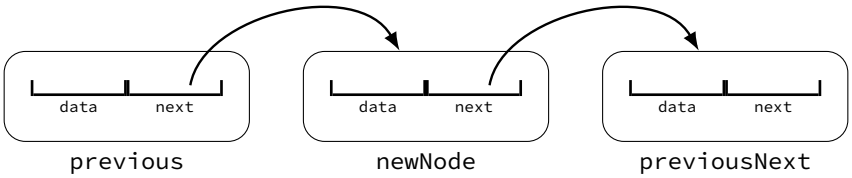
pushing all later elements in the list down by one. In a linked list, we just place the new node in the correct location. We do not need to manually "push down" elements: all later elements will automatically be one further down in the list, since there is now a new node in front of them.

This method requires some careful "surgery" on the linked list: we need to disconnect the reference to insert the new node. Let's draw a picture of what that looks like. Let's say we have found the $(\texttt{index}-1)$-st node in our list; its next node is the current `index`-th node. We want our new node to go in between them.

Before any operations, the relevant nodes look like this (the rest of the list is ommitted). We will use `previous` to refer to the $(\texttt{index}-1)$-st node in the list before the operation, and `previousNext` to refer to the next node: the `index`-th in the list.



We will set the references one at a time to insert newNode in between them.



Alternatively, if `index` is 0 (we are adding to the beginning of the list), we need to set our new node to be the head of the list, and set its `next` to point to the previous head of the list.

```
1  public void add(int index, int newElement) {
2      checkInBounds(index);
3      IntNode newNode = new IntNode();
4      newNode.setData(newElement);
5      if(index == 0) {
6          newNode.setNext(head);
7          head = newNode;
8      } else {
9          IntNode previous = getNode(index - 1); //find the node
               immediately before the index-th node
10         IntNode previousNext = previous.getNext(); //find the node
               immediately after previous
11         previous.setNext(newNode);  //set it to point to our new node
12         newNode.setNext(previousNext); //continue the list
```

```
    }
    numElems++;
}
```

Remove works similarly: once we find the correct node, we take it out of the list. The node that was previously before it immediately points to the node after it.

```
public int remove(int index){
    checkInBounds(index);
    int toReturn;
    if(index == 0) {
        toReturn = head.getData();
        head = head.getNext();
    } else {
        IntNode previous = getNode(index - 1);
        IntNode toRemove = previous.getNext();
        previous.setNext(toRemove.getNext());
        toReturn = toRemove.getData();
    }
    numElems--;
    return toReturn;
}
```

`int indexOf(int element)` returns the first index in the list that contains `element`, or -1 if there is no such index. As in `ArrayList`, we'll just loop through the list until we find an element.

```
public int indexOf(int element) {
    IntNode current = head;
    for(int index = 0; index < numElems; index++) {
        if(current.getData() == element) {
            return index;
        }
        current = current.getNext();
    }
    return -1;
}
```

`boolean contains(int element)`: returns `true` if `element` is in the list, and `false` otherwise.

```
public boolean contains(int element) {
    return indexOf(element) != -1;
}
```