CSE 143
Lecture 5

References and Linked Nodes

slides created by Marty Stepp and Ethan Apter
http://www.cs.washington.edu/143/

Values vs. References

- Does the following `swap` method work? Why or why not?

```
public static void main(String[] args) {
    int a = 7;
    int b = 35;

    // swap a with b
    swap(a, b);

    System.out.println(a + " " + b);
}

public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Value semantics

- **value semantics**: Behavior where values are copied when assigned to each other or passed as parameters.
  - When one primitive is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

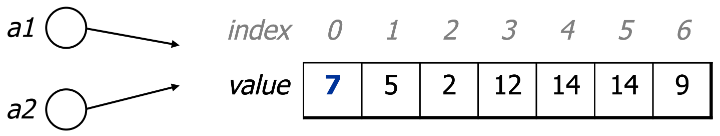
```
int x = 5;
int y = x;      // x = 5, y = 5
y = 17;         // x = 5, y = 17
x = 8;          // x = 8, y = 17
```

3

Reference semantics

- **reference semantics**: Behavior where variables actually store the address of an object in memory.
  - When one reference variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.

```
int[] a1 = {4, 5, 2, 12, 14, 14, 9};
int[] a2 = a1;      // refers to same array as a1
a2[0] = 7;
System.out.println(a1[0]);    // 7
```

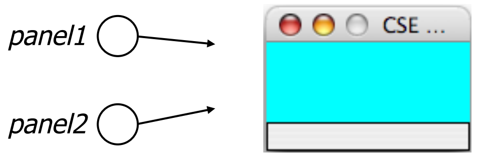


4

References and objects

- In Java, objects and arrays use reference semantics. Why?
  - *efficiency.* Copying large objects slows down a program.
  - *sharing.* It's useful to share an object's data among methods.

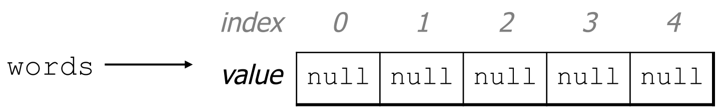
```
DrawingPanel panel1 = new DrawingPanel(80, 50);
DrawingPanel panel2 = panel1;    // same window
panel2.setBackground(Color.CYAN);
```



Null references

- **`null`** **:** A value that does not refer to any object.
  - The elements of an array of objects are initialized to `null`.

```
String[] words = new String[5];
```



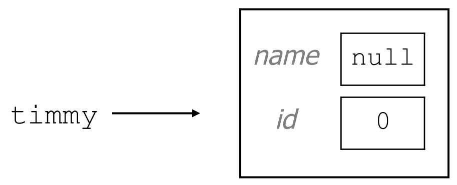
  - not the same as the empty string `""` or the string `"null"`

Null references

– Uninitialized reference fields of an object are initialized to `null`.

```
public class Student {
    String name;
    int id;
}

Student timmy = new Student();
```



7

Things you can do w/ `null`

- store `null` in a variable or an array element

```
String s = null;
words[2] = null;
```

- print a `null` reference

```
System.out.println(timmy.name);        // null
```

- ask whether a variable or array element is `null`

```
if (timmy.name == null) { ...          // true
```

- pass `null` as a parameter to a method
  – some methods don't like `null` parameters and throw exceptions

- return `null` from a method (often to indicate failure)

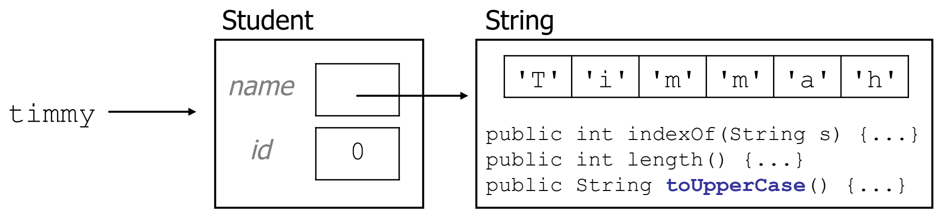
```
return null;
```

8

Dereferencing

- **dereference**: To access data or methods of an object.
  - Done with the dot notation, such as `s.length()`
  - When you use a `.` after an object variable, Java goes to the memory for that object and looks up the field/method requested.

```
Student timmy = new Student();
timmy.name = "Timmah";
String s = timmy.name.toUpperCase();
```



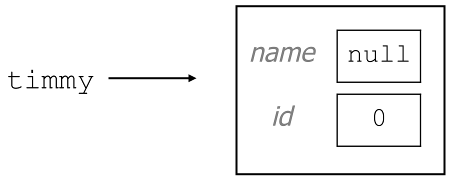
Student
String
timmy
name
id
0
'T' 'i' 'm' 'm' 'a' 'h'
public int indexOf(String s) {...}
public int length() {...}
public String toUpperCase() {...}

9

Null pointer exception

- It is illegal to dereference `null` (it causes an exception).
  - `null` does not refer to any object, so it has no methods or data.

```
Student timmy = new Student();
String s = timmy.name.toUpperCase();    // ERROR
```



Output:

```
Exception in thread "main"
java.lang.NullPointerException
        at Example.main(Example.java:8)
```

10

References to same type

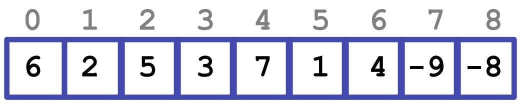
- What would happen if we had a class that declared one of its own type as a field?

```
public class Strange {
    private String name;
    private Strange other;
}
```

  - Will this compile?
    - If so, what is the behavior of the `other` field? What can it do?
    - If not, why not? What is the error and the reasoning behind it?

Array-Based List Review

- Array-based lists are what we've studied so far
  - **`ArrayIntList`**, **`ArrayList`**, **`SortedIntList`** all use arrays
- Arrays use a contiguous block of memory
- This means all elements are adjacent to each other



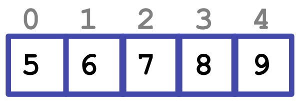
Advantages and Disadvantages

- Advantages of array-based lists
  - random access: can get *any* element in the entire array quickly
    - kind of like jumping to any scene on a DVD (no fast-forwarding required)
- Disadvantages of array-based lists
  - can't insert/remove elements at the front/middle easily
    - have to shift the other elements
  - can't resize array easily
    - have to create a new, bigger array

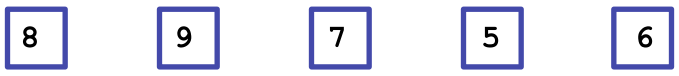
13

Linked Lists

- A linked list is a type of list
- But instead of a contiguous block of memory, like this:



- Linked list elements are scattered throughout memory:

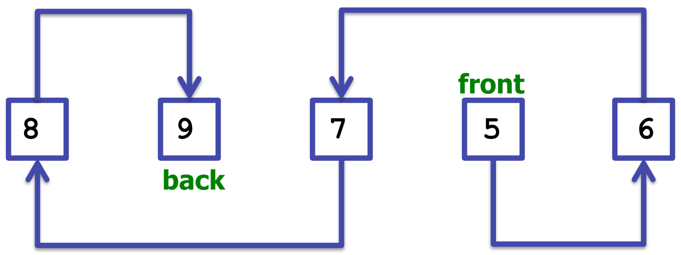


- But now the elements are unordered. How do linked lists keep track of everything?

14

Linked Lists

- Each element must have a reference to the next element:



- Now, so long as we keep track of the first element (the front), we can keep track of all the elements

15

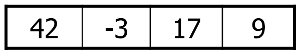
Linked Lists

- These references to the next element mean that linked lists have sequential access
- This means that to get to elements in the middle of the list, we must first start at the front and follow all the links until we get to the middle:
  - kind of like fast-forwarding on a VHS tape
  - so getting elements from the middle/back is slow
- Linked lists also do some things well:
  - linked lists can insert elements quickly (no "shifting" needed)
  - linked lists can always add more elements (no set capacity)
- So there are tradeoffs between array lists and linked lists

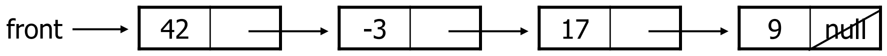
16

Linked data structures

- All of the collections we will use and implement in this course use one of the following two underlying data structures:
  - an **array** of all elements
    - `ArrayList`, `Stack`, `HashSet`, `HashMap`



  - a set of **linked objects**, each storing one element, and one or more reference(s) to other element(s)
    - `LinkedList`, `TreeSet`, `TreeMap`



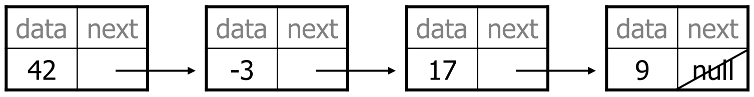
17

A list node class

```
public class ListNode {
    int data;
    ListNode next;
}
```

- Each list node object stores:
  - one piece of integer data
  - a *reference* to another list node (it does NOT contain another **`ListNode`** object)

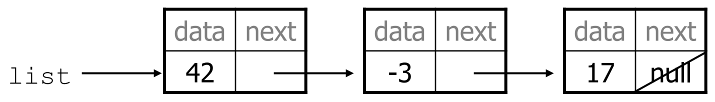
- `ListNode`s can be "linked" into chains to store a list of values:



18

List node client example

```
public class ConstructList1 {
    public static void main(String[] args) {
        ListNode list = new ListNode();
        list.data = 42;
        list.next = new ListNode();
        list.next.data = -3;
        list.next.next = new ListNode();
        list.next.next.data = 17;
        list.next.next.next = null;
        System.out.println(list.data + " " + list.next.data
                           + " " + list.next.next.data);
        // 42 -3 17
    }
}
```



List node w/ constructor

```
public class ListNode {
    int data;
    ListNode next;

    public ListNode(int data) {
        this.data = data;
        this.next = null;
    }

    public ListNode(int data, ListNode next) {
        this.data = data;
        this.next = next;
    }
}
```

– Exercise: Modify the previous client to use these constructors.

List node client example

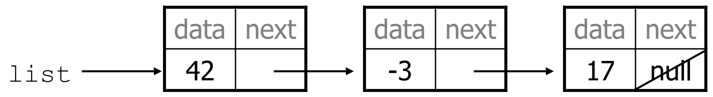
- Two possible solutions:

```
ListNode list = new ListNode(42, new ListNode(-3, new ListNode
  (17)));

ListNode list = new ListNode(17);
list = new ListNode(-3, list);
list = new ListNode(42, list);
```

  - NOTE: In the second solution, the nodes are added in reverse order!

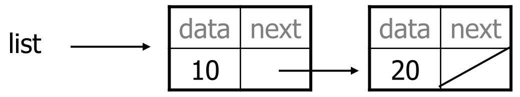
- List creation is still somewhat tedious. More on that next time.



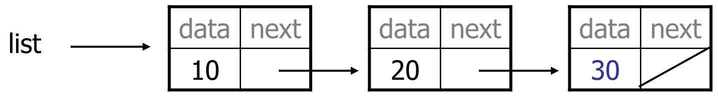
21

Linked node problem 1

- What set of statements turns this picture:



- Into this?

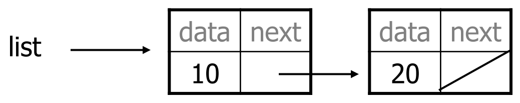


```
list.next.next = new ListNode(30);
```

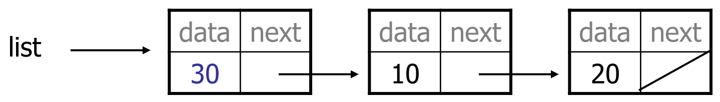
22

Linked node problem 2

- What set of statements turns this picture:



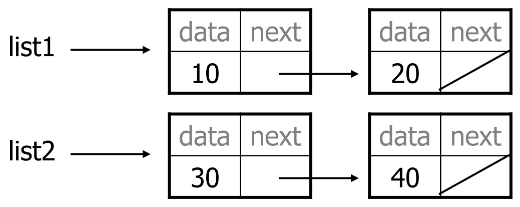
- Into this?



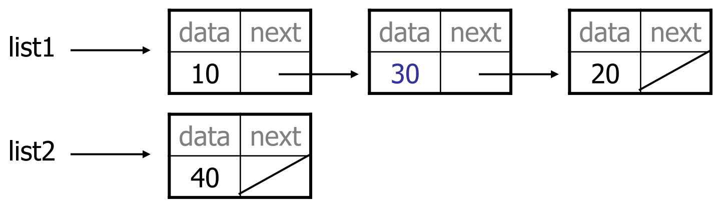
```
list = new ListNode(30, list);
```

Linked node problem 3

- What set of statements turns this picture:



- Into this?



Linked node problem 3

- Two possible solutions:

```
ListNode temp = list1.next;
list1.next = list2;
list2 = list2.next;
list1.next.next = temp;
```

```
ListNode temp = list2.next;
list2.next = list1.next;
list1.next = list2;
list2 = temp;
```

Final Thoughts

- Working with linked lists can be hard
- **Draw lots of pictures!**
- jGRASP's debugger can also be helpful
  - but remember: you won't have jGRASP on the exams
  - and linked lists are *definitely* on the exams
- Sometimes, solving one of these problems requires a temporary variable:

```
ListNode temp = p;
```

This creates a `ListNode` variable. It does *not* create a new `ListNode` object (no call on `new`).