# LECTURE 8:
# DATA STRUCTURES

CS 2110
Fall 2021

# Agenda

Previously in 2110:

- Encapsulation
- Inheritance
- Polymorphism

Today:

- Data structures
- Abstract data types (ADTs) and interfaces
- ADT List: array lists, Linked lists (intro to A3)
- Abstract classes

# EQUALITY CONTINUED…

CS 2110
Fall 2021

# What Is Equality?

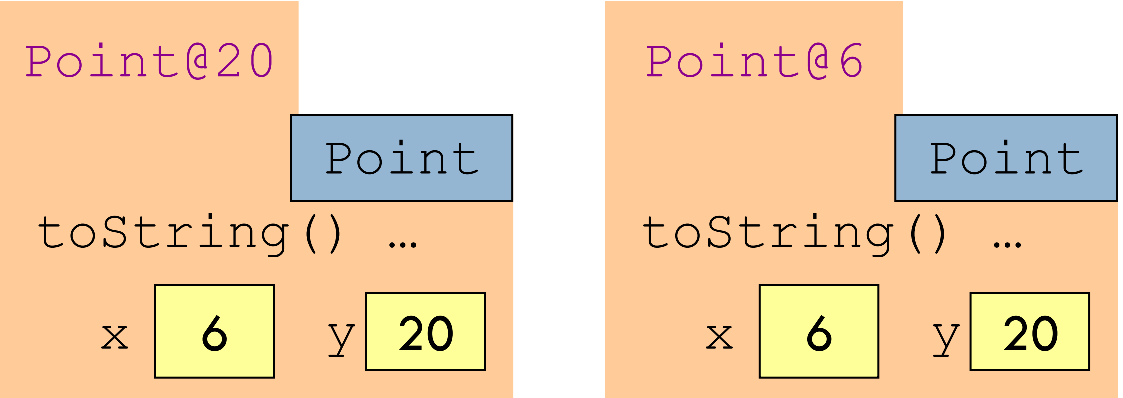
- Referential equality: two references point to the same object in memory
  - Java: `o1 == o2`

... and not this (except with primitives)

- Equivalence: two different objects are deemed “equivalent” according to the programmer
  - Java: `o1.equals(o2)`

**Almost always want this!**

These objects could be considered equivalent, or equal



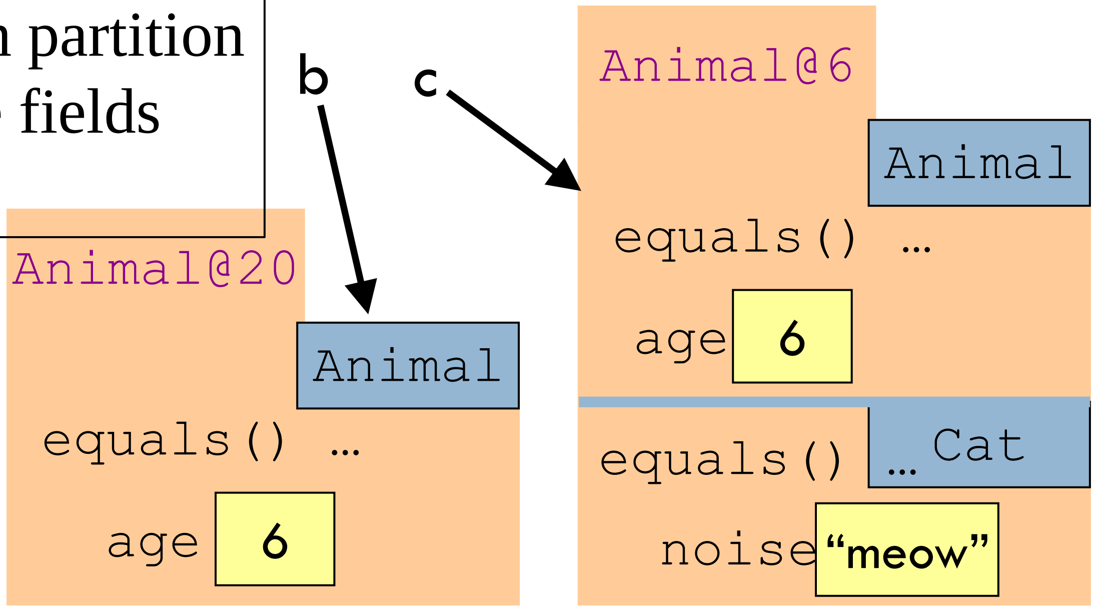
# Overriding function equals

We will override function equals in classes Animal and Cat.

(Note: it wouldn’t make sense to think of objects b and c as being equal).

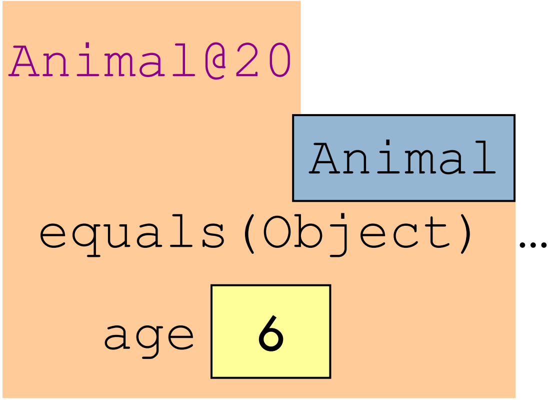
c.equals(b)
Calls method equals in partition Cat; try to check noise fields and return false

b.equals(c)
Probably check age fields and return true



# Overriding equals in class Animal

```
/** = “this object and ob are of the same class”  and
          their ages are the same  */
@Override
public boolean equals(Object ob) {
     if (ob == null || getClass() != ob.getClass()) return false;
     Animal oba= (Animal)ob;
     return age == oba.age;     // return age == ((Animal)ob).age;
}
```



Animal@20

Animal

equals(Object) …

age 6

By compile-time reference rule,
`ob.age` is illegal

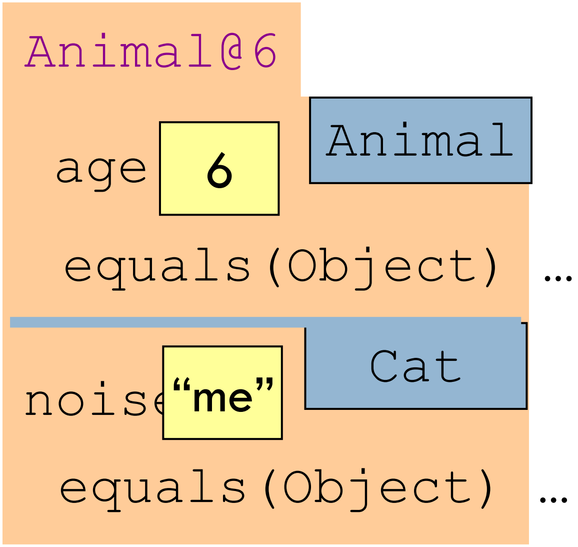
# Overriding equals in class Cat

**In class Cat**

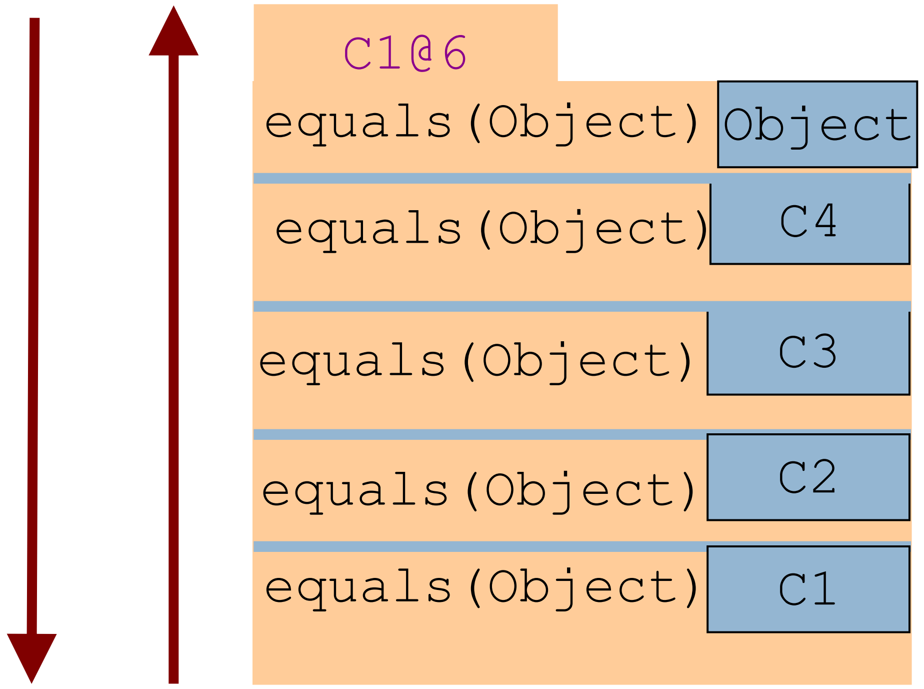
```
/** = "this object and ob are of the same class" and
            their ages and noises are the same */
public boolean equals(Object ob) {
   if (!super.equals(ob))  return false;
   Cat oba=  (Cat) ob;
   return noise.equals(oba.noise);
}
```

**In class Animal**

```
/** = "this object and ob are of the same class"
            and their ages are the same */
public boolean equals(Object ob)
```



# OOP: process superclass partitions first

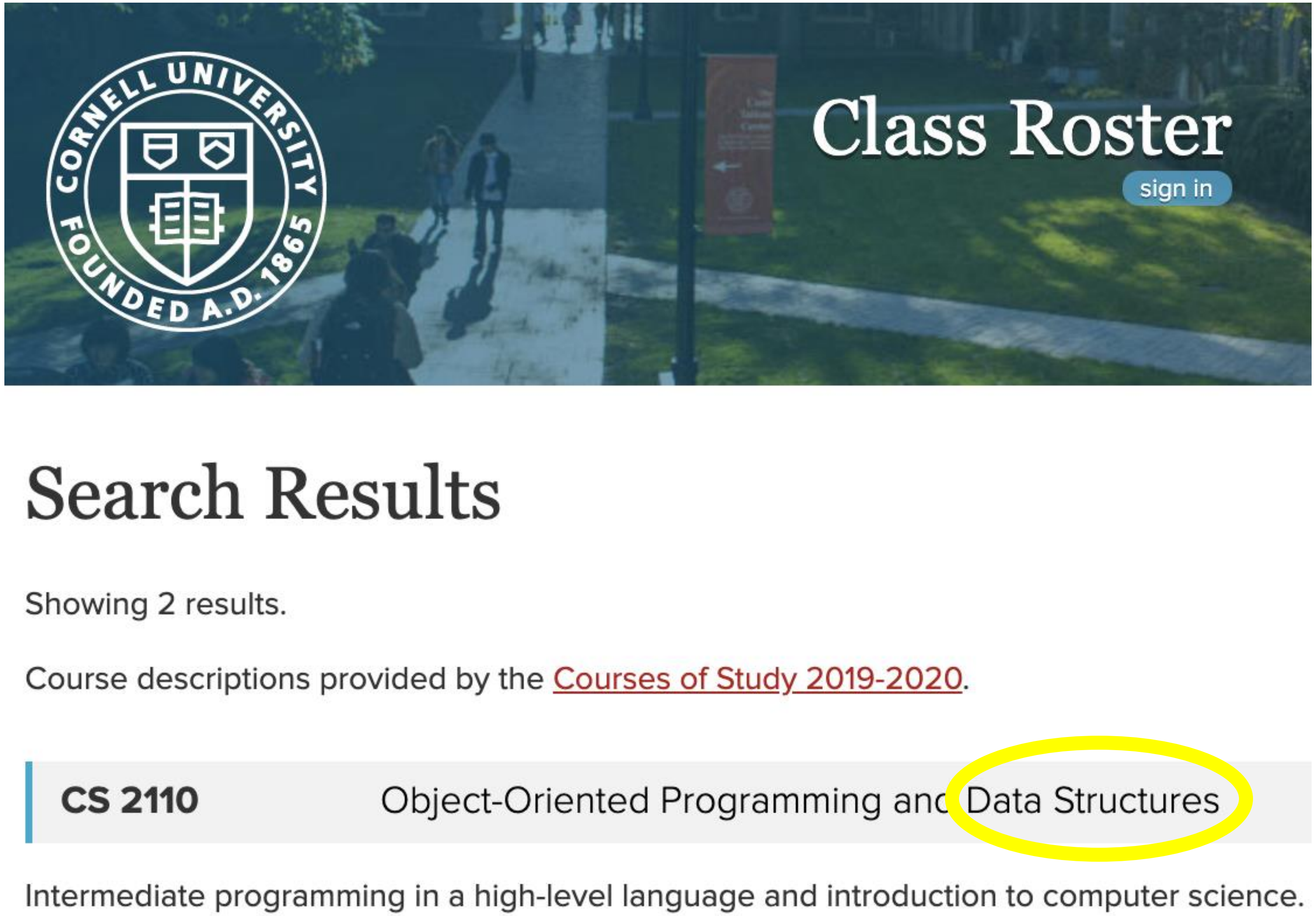


Where have we seen this before?

# LECTURE 8:
# DATA STRUCTURES

## PART 1: WHY DATA STRUCTURES

CS 2110
Fall 2021



Class Roster

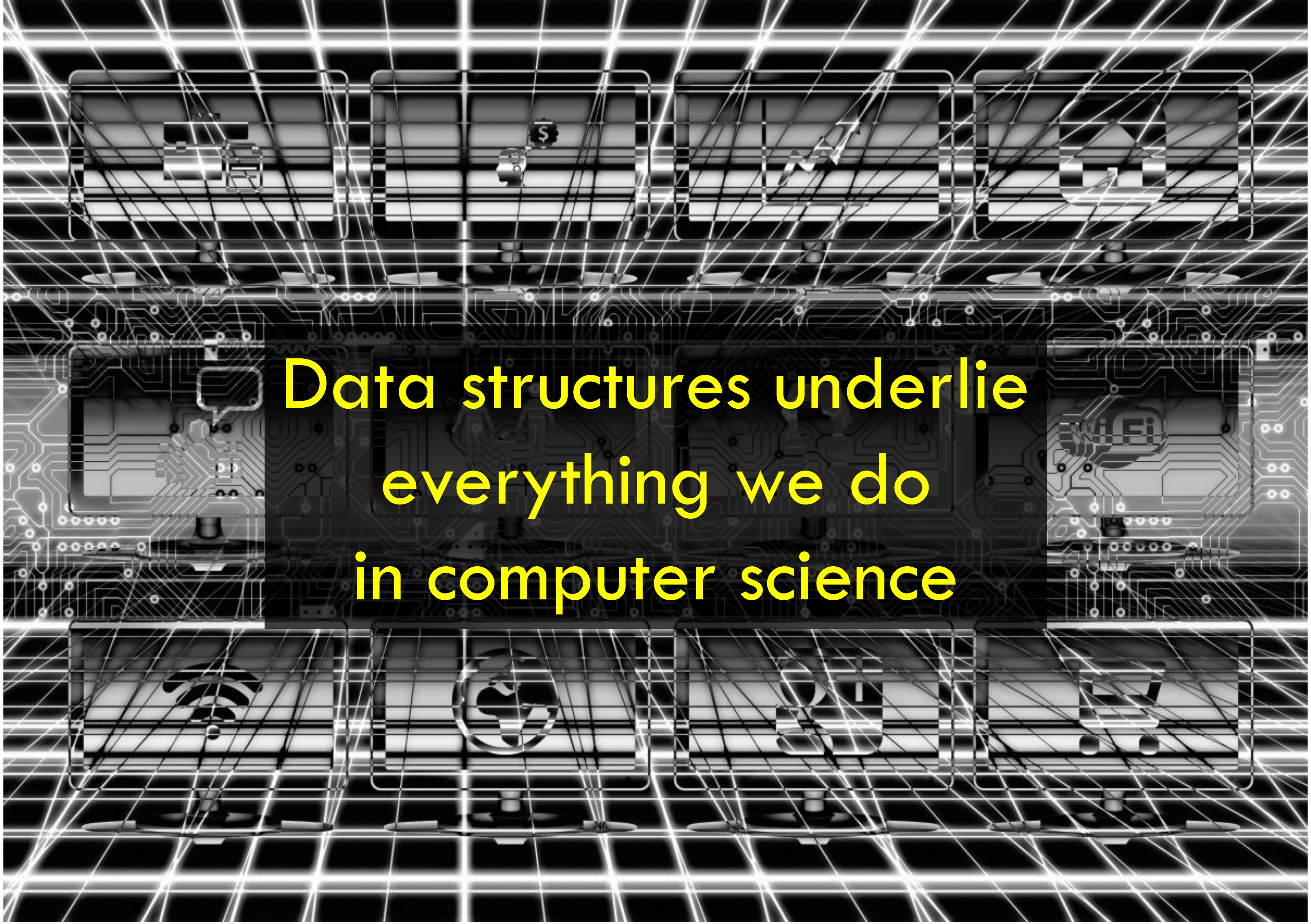
# Search Results

Showing 2 results.

Course descriptions provided by the Courses of Study 2019-2020.

**CS 2110** Object-Oriented Programming and Data Structures

Intermediate programming in a high-level language and introduction to computer science.



Data structures underlie everything we do in computer science

Image: public domain

# What is a Data Structure?

- □ A format for storing data, and
- □ Algorithms for operations that manage the data

Format: what data,
and where to put it

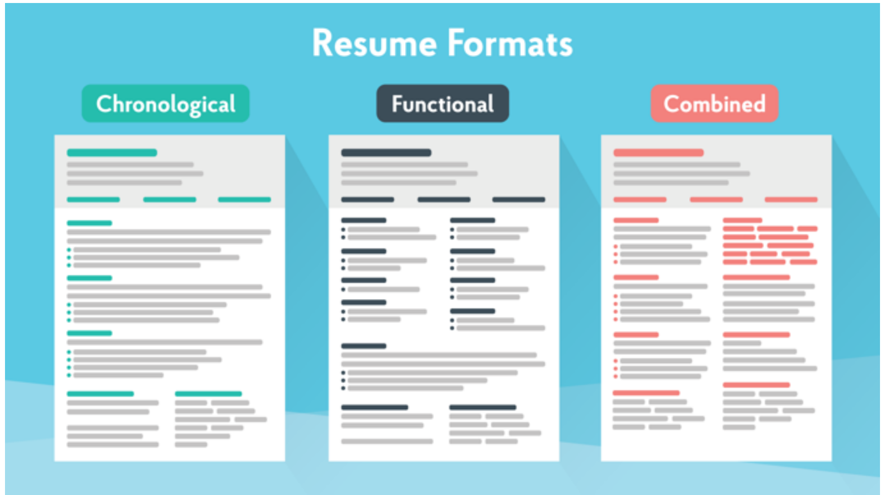


Image: https://novoresume.com/career-blog/resume-formats

Operations: how to
transform data

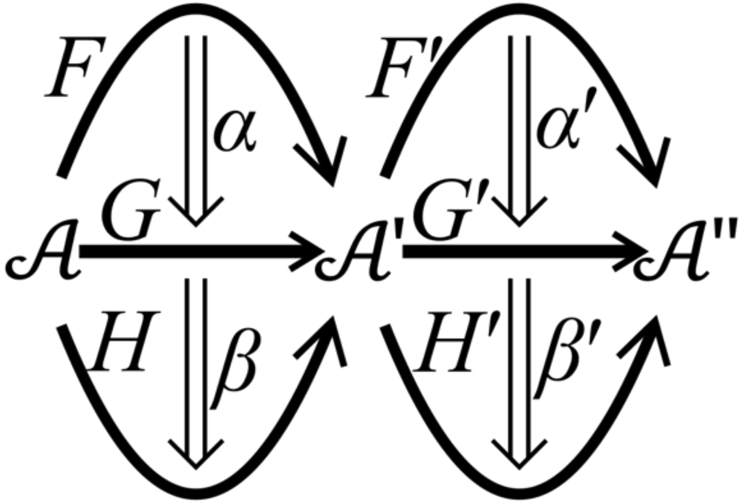


Image: public domain

# Why Study Data Structures?

- Data management is a primary use of computers
- Libraries provide them for our use
- Choices affect performance, usability of software
- Implementing them ourselves gives us:
  - excellent coding practice
  - deep understanding of the data structures

# LECTURE 8: DATA STRUCTURES

## PART 2: ADTS AND THE JAVA INTERFACE

CS 2110
Fall 2021

# ADT

Type: a set of values together with operations on them.

Java: type **int**, **double**, …

class types, like String, Animal, Account

Abstract Data Type, or ADT

A type:

(1) defines a data structure

(2) It's abstract —no implementation is given

INDEPENDENT SINCE 1880

The Cornell Daily Sun

# List:

# 161 Things Every Cornellian Should Do

0. [Redacted]
1. Finally meet the dazzling Denice Cassaro
2. Go to the Cornell-Harvard men’s hockey game and throw fish on the ice
3. Take off to NYC for Fall Break, being sure to post on Instagram about it at least twice
4. Sled down Libe Slope during a snowstorm
5. Take Hotel Administration 4300: Introduction to Wines

…

158. Tell a professor what you really think of their class
159. Attend a Sun meeting
160. Climb all 161 steps to the top of McGraw Tower

https://cornellsun.com/161-things-every-cornellian-should-do/

# An ADT: List

A list, or sequence, is an ordered collection of items

s1: (5, 8, -2, 8, 7)

s2: (“who”, “what”, “when”, “where”, “why”)

s3: ('h', 'o', 'n', 'e', 's', 't', 'y')

s4: (“word”, 4, “the”, 3, “and”, 3)

Notation for lists (*not* Java)
s[0], s[1], s[2], …, s[s.length-1]

s[i..j]: stands for sublist consisting of s[i], s[i+1], …, s[j]

s[i..]: stands for sublist consisting of s[i], s[i+1], …, s[s.length-1]

s1 + s2: list catenation

# An ADT: List —its operations

In introducing ADTs and a Java feature that can be used to define an ADT, we limit our operations on a List to 3:

prepend(e): insert item e at the beginning

get(i): return item s[i]

size(): number of elements

Examples:

Prepend 5 to (4, 5, 7, 2): Result is (5, 4, 5, 7, 2)

get(3) from the list (5, 4, 5, 7, 2) is 7

size() of (5, 4, 5, 7, 2) is 5

# Interfaces

- An interface describes the operations of an ADT
- The notion of “interface” can be found in many languages
- Java actually provides a syntax for it …

# Java Interface

```
public interface List<T> {
    void prepend(T val);
    T get(int i);
    int size();
}
```

No method bodies. Declares what operations must be provided, not how they are to be implemented.

No constructors.  Cannot instantiate —not a class.

Methods are automatically public. No need to put in access modifier

Description of values and method specifications given in Javadoc comments

Demo

# Java Interface

```
public interface List<T> {
    void prepend(T val);
    T get(int i);
    int size();
}
```

How do use this interface? What good is it?

In part 3 of this lecture, we will see that we can write several different implementations of the ADT defined by interface List. Then, we can choose one of them to use in any program, and switch to a different one if it performs better in our program.

# LECTURE 8:
# DATA STRUCTURES

## PART 3: TWO IMPLEMENTATIONS OF ADT LIST

CS 2110
Fall 2021

# An implementation of List: ArrayList

```
public class ArrayList<T>                    implements List{<T>
    /** The items of the list are in values[0..size-1] */
    T[] values;
    /** number of items in the list */
    int size;

    /** Constructor: An empty list with capacity c. */
    public ArrayList(int c) {
        values= (T[]) new Object[c];
    }
```

… more to come …

Demo in Eclipse

Implements clause: Requires that the class implements all methods declared in interface List

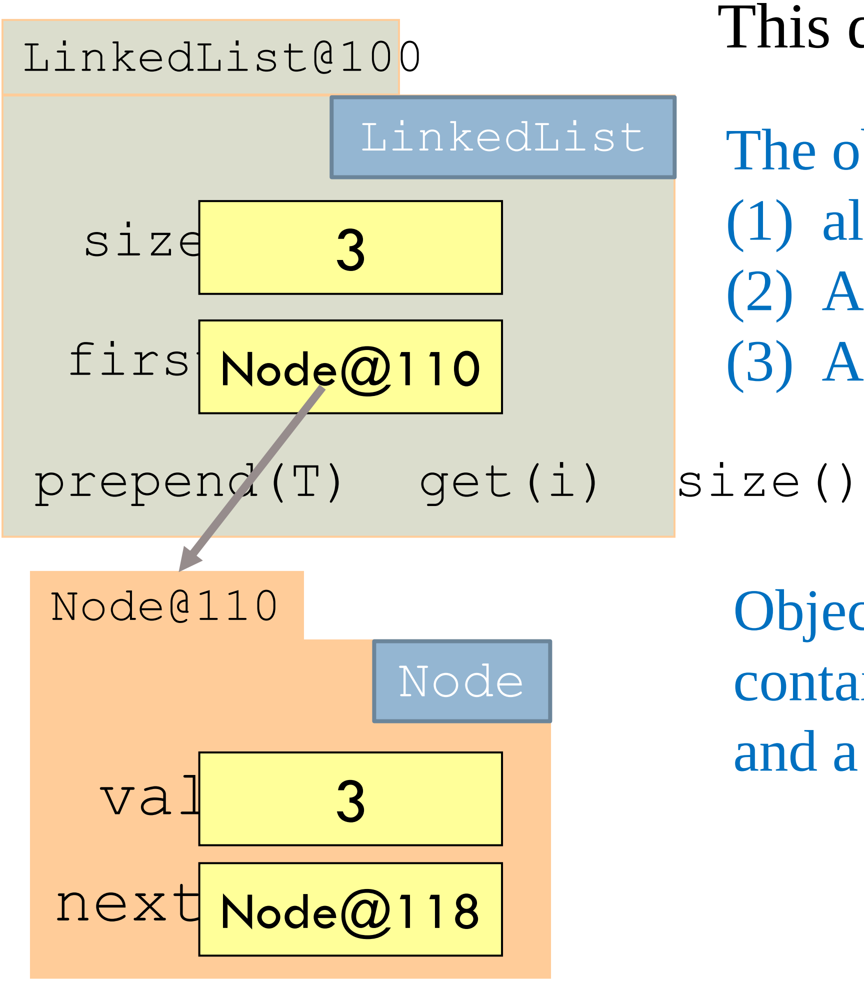
# An implementation of List: LinkedList

We introduce the notion of a linked list, as an implementation of abstract data type List.

Assignment A4 is all about linked lists. For A4, you are expected to read about linked lists in JavaHyperText, entry linked list
The emphasis is on learning about a data structure by reading about it. This short intro will help.

You will learn about a new Java feature: the inner class.

# Singly Linked List: How To Store



This data structure contains the list (3, 5, 2)

The object of class LinkedList contains

(1) all three methods of interface/ADT List

(2) A field that contains the size of the list

(3) A pointer to an object of class Node:

Object Node@110, of class Node, contains the first value of the list, 3, and a pointer to another Node object.

# Singly Linked List: How To Store

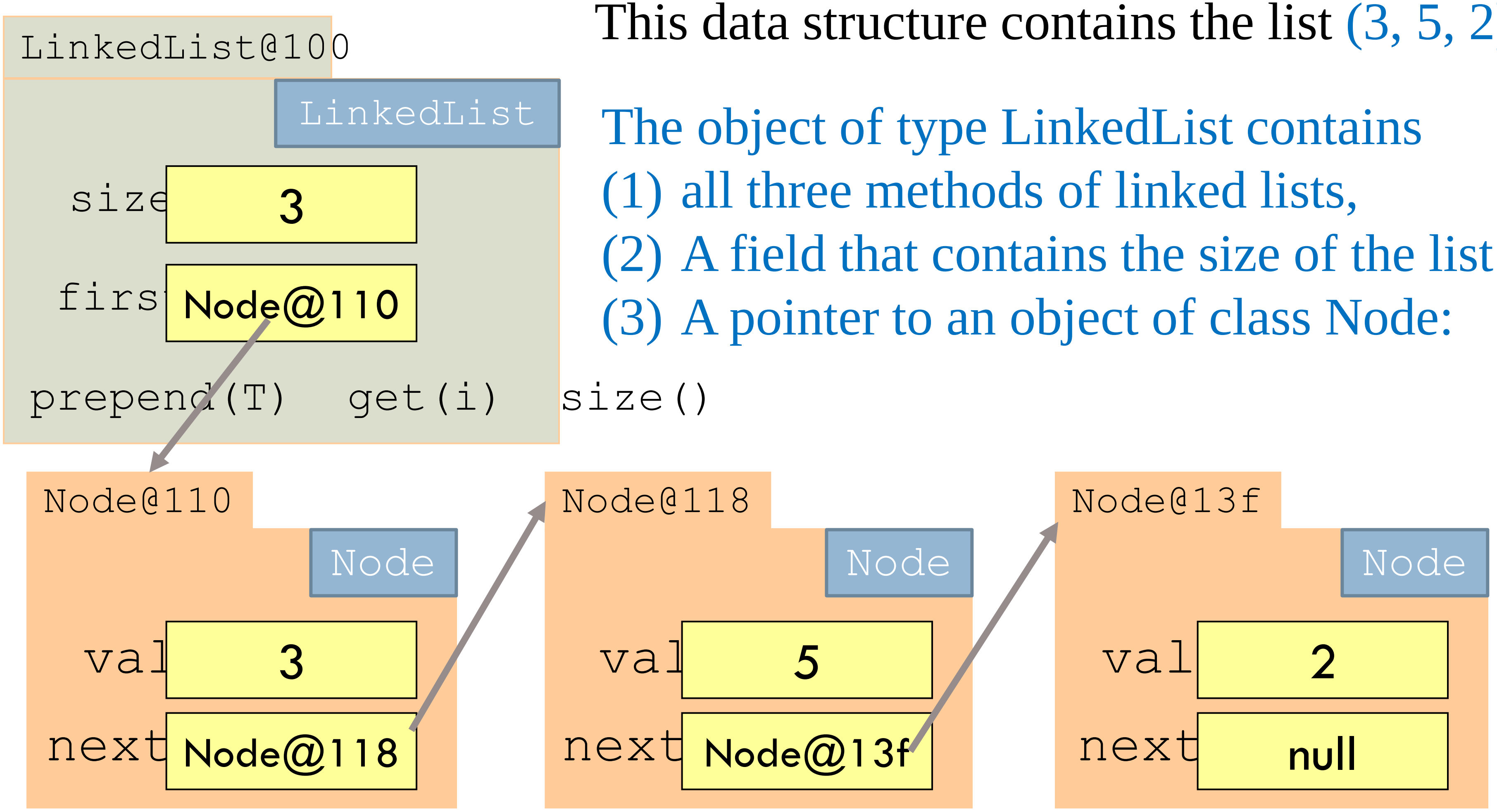
This data structure contains the list (3, 5, 2)

The object of type LinkedList contains

(1) all three methods of linked lists,

(2) A field that contains the size of the list

(3) A pointer to an object of class Node:



# LinkedList

Inner class: Node nested inside LinkedList. No reason to expose to outside world.

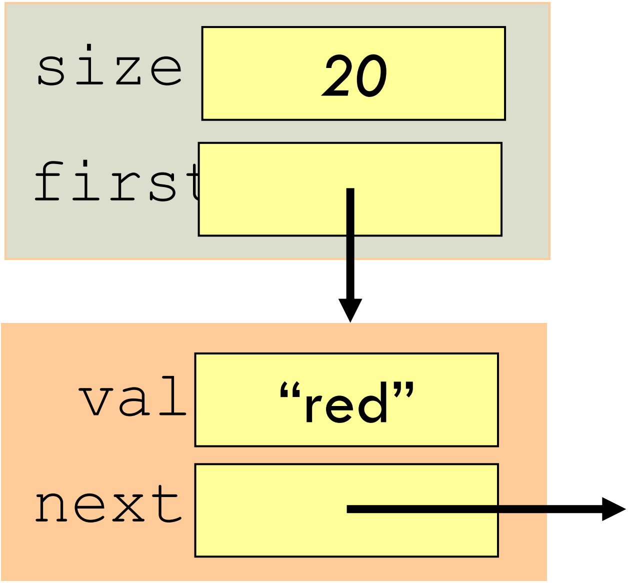
```
public class LinkedList<T> {
    private class Node {
        T val;
        Node next;
        Node(T val, Node next) {
            this.val= val;  this.next= next;
        }
    }

    private Node first;
    private int size;
    …
```

# LinkedList

```
public class LinkedList<T> {
     …
     public void prepend(T val) {
          Node n= new Node(val, first);
          first= n;
          size++ ;
     }
```

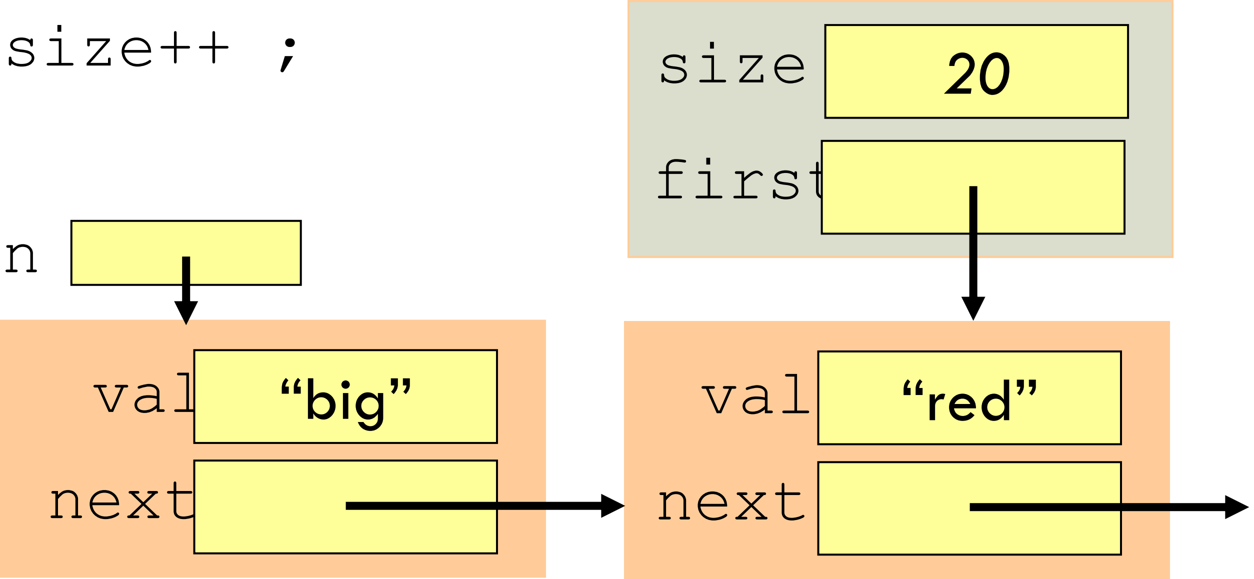
prepend("big")



# LinkedList

```
public class LinkedList<T> {
    …
    public void prepend(T val) {
        Node n= new Node(val, first);
        first= n;
        size++ ;
    }
```

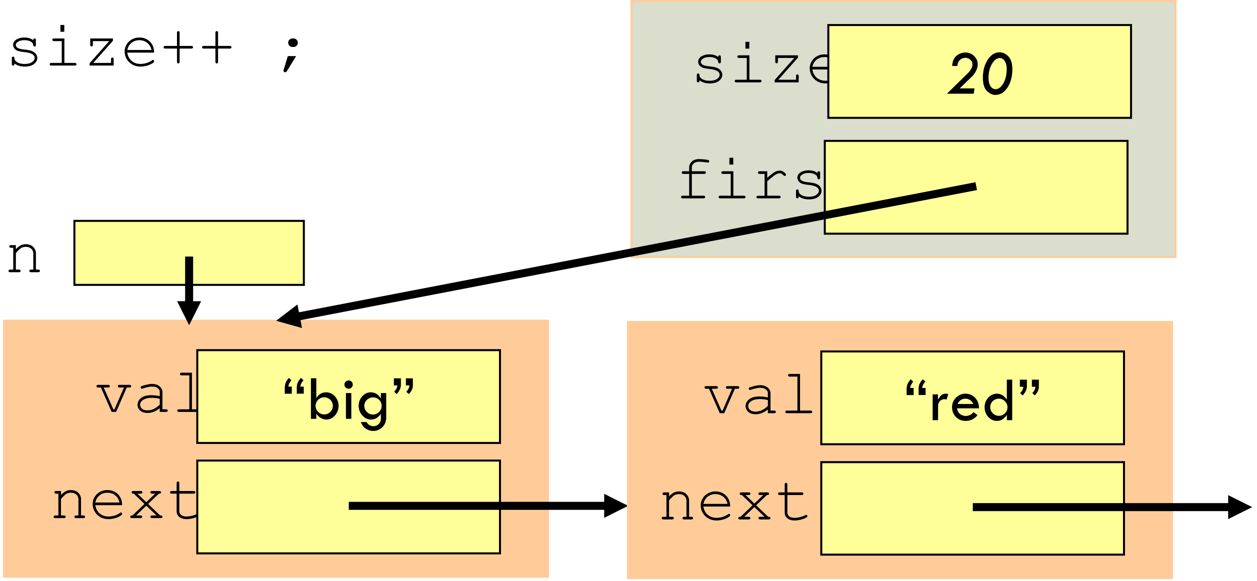
prepend(“big”)



# LinkedList

```
public class LinkedList<T> {
    …
    public void prepend(T val) {
        Node n= new Node(val, first);
        first= n;
        size++ ;
    }
```

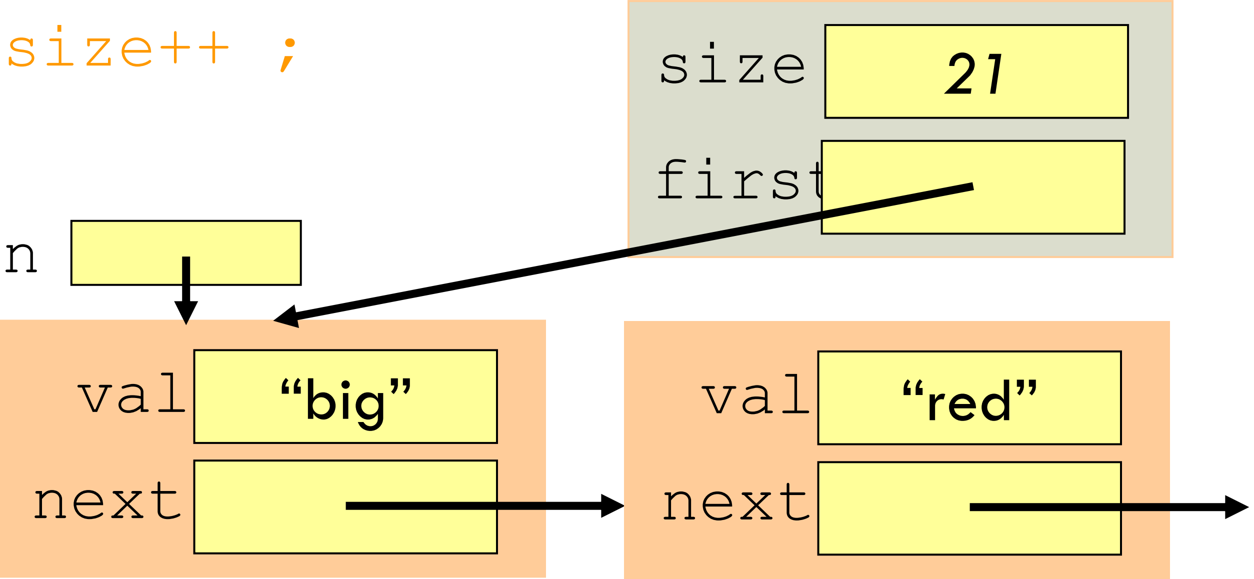
prepend(“big”)



# LinkedList

```
public class LinkedList<T> {
    …
    public void prepend(T val) {
        Node n= new Node(val, first);
        first= n;
        size++ ;
    }
```

prepend(“big”)



# LinkedList

```
public class LinkedList<T> {
    …
    public void prepend(T val) {
        Node n= new Node(val, first);
        first= n;
        size++ ;
    }
```

Cost: inexpensive! Create one `Node object`, do three assignments.

Doesn't matter how many items are already in list.

We call it a "constant time" operation.

# LinkedList: Method get(i)

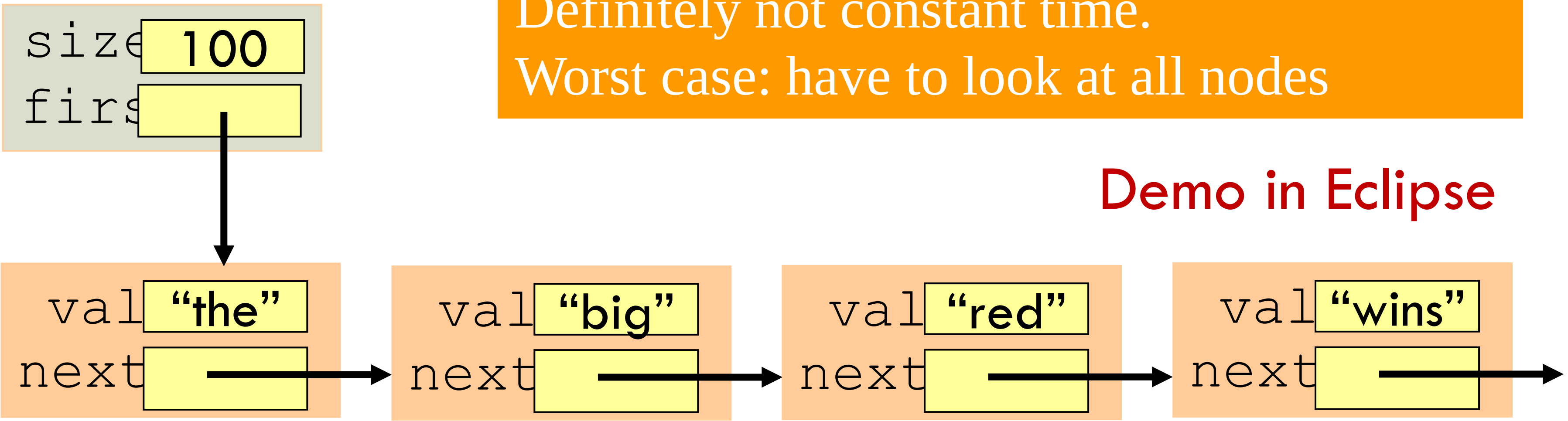
The call get(3) has to search the list from the beginning to find item number 3 and return the value "wins".

get(3)

The call get(99) has to look at all nodes to get to the last one!

The call get(i) takes time proportional to i.
It's a linear operation in i.
Definitely not constant time.
Worst case: have to look at all nodes

Demo in Eclipse



# LinkedList vs. ArrayList

Efficiency of operations

| | LinkedList | ArrayList |
|---|---|---|
| prepend | constant time | linear time |
| get | linear time | constant time |

constant time: inexpensive; independent of number of list elements

linear time: relatively expensive; proportional to number of list elements

Next part of lecture: Illustrate how the use of interface List helps us easily choose one implementation and perhaps switch to the other later on.

# LECTURE 8:
# DATA STRUCTURES

## PART 4: INTERFACES AND SUBTYPING

CS 2110
Fall 2021

# Interface vs. Implementation

# Interface vs. Implementation

I don’t care.

Just give me coffee.

Images:
http://www.globalvendinggroup.com/products/National-637-Coffee-Machine.html,
https://en.wikipedia.org/wiki/Coffee_vending_machine#/media/File:Mechanisms_inside_a_coffee_vending_machine.jpg

# Interface vs. Implementation

**Interface:** the operations of an ADT

- □ What you see in documentation web pages
- □ Method names and specifications
- □ Abstract from details: what to do, not how to do it
- □ Java syntax: `interface`

**Implementation:** the code for a data structure

- □ What you see in source code
- □ Fields and method bodies
- □ Provide the details: how to do operation
- □ Java syntax: `class`

Could be many implementations of an interface

e.g. List: ArrayList, LinkedList

# LinkedList vs. ArrayList

Both support the same operations: `prepend, get`

Always an engineering tradeoff: choose efficient data structure for operations of concern

But, some clients won't care about the different efficiency as long as they get the operations they want…

| | `LinkedList` | `ArrayList` |
|---|---|---|
| `prepend` | constant time | linear time |
| `get` | linear time | constant time |

# Interfaces and Subtyping

**Recall:** if S is a subtype of T, then anywhere a T is expected, an S can be used

**Recall:** if SC extends C, then SC is a subtype of C

**New:** if C implements I, then C is a subtype of I

Examples:

□

```
List<T> lst=  new LinkedList<>();
```

□

```
void m(List<T> lst) { … }
m(new ArrayList<>());
```

Compile-time reference rule:
Only operations allowed on lst:
prepend, get, size

# Example

Change implementation?
Change new-expression

Create ArrayList object, store it in a List variable

```
int size= 50000;
List<Integer> al= new ArrayList<>(size);


long start= System.currentTimeMillis();
for (int k= 0; k < size; k= k + 1)    al.prepend(k);
long time= System.currentTimeMillis() - start;


System.out.println("Time for " + size + " prepends: " +
                                        time + " milliseconds");
```

Current time, in milliseconds*

Elapsed time to prepend size values

* Time since midnight, January 1, 1970 UTC

Demo in Eclipse

# Java List, ArrayList, LinkedList

Java has in package java.util

interface List<T>

class ArrayList<T> implements List<T>

class LinkedList<T> implements List<T>

class Stack<T> implements List<T>

class Vector<T> implements List<T>

and more

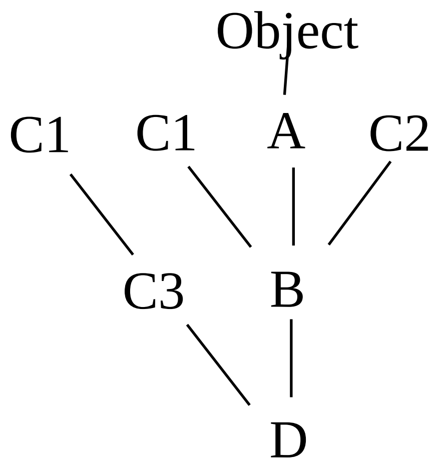
# Your task: Learn about casting

Homework, to complete your introduction to interfaces.

JavaHyperText, upper navigation bar,
item Abstract classes and interfaces

1. Read about Three other components of an interface
2. Study 3.75-minute video on Casting to learn about casting

and drawing objects of classes that implement interfaces



www.cs.cornell.edu/courses/JavaAndDS/abstractInterface/01ai.html

# LECTURE 8:
# DATA STRUCTURES

## PART 5: ABSTRACT CLASSES AND METHODS

CS 2110
Fall 2021

# Abstract Classes and Methods

Make a class abstract so that it cannot be instantiated —objects of the class cannot be created.

Make a method in an abstract class (or interface abstract so it must eventually be overridden —defined in a subclass.

Watch the first, short, JHT tutorial on Abstract classes and interfaces:

https://www.cs.cornell.edu/courses/JavaAndDS/abstractInterface/01ai.html

# Your Turn: Read in JavaHyperText

- data structure
- list, linked list, doubly-linked list
- abstract data type
- interface, implements
- abstract method, abstract class