CSCI 132:
Basic Data Structures and Algorithms

Lessons Learned so far + Importing Linked Lists

Reese Pearsall

Spring 2024

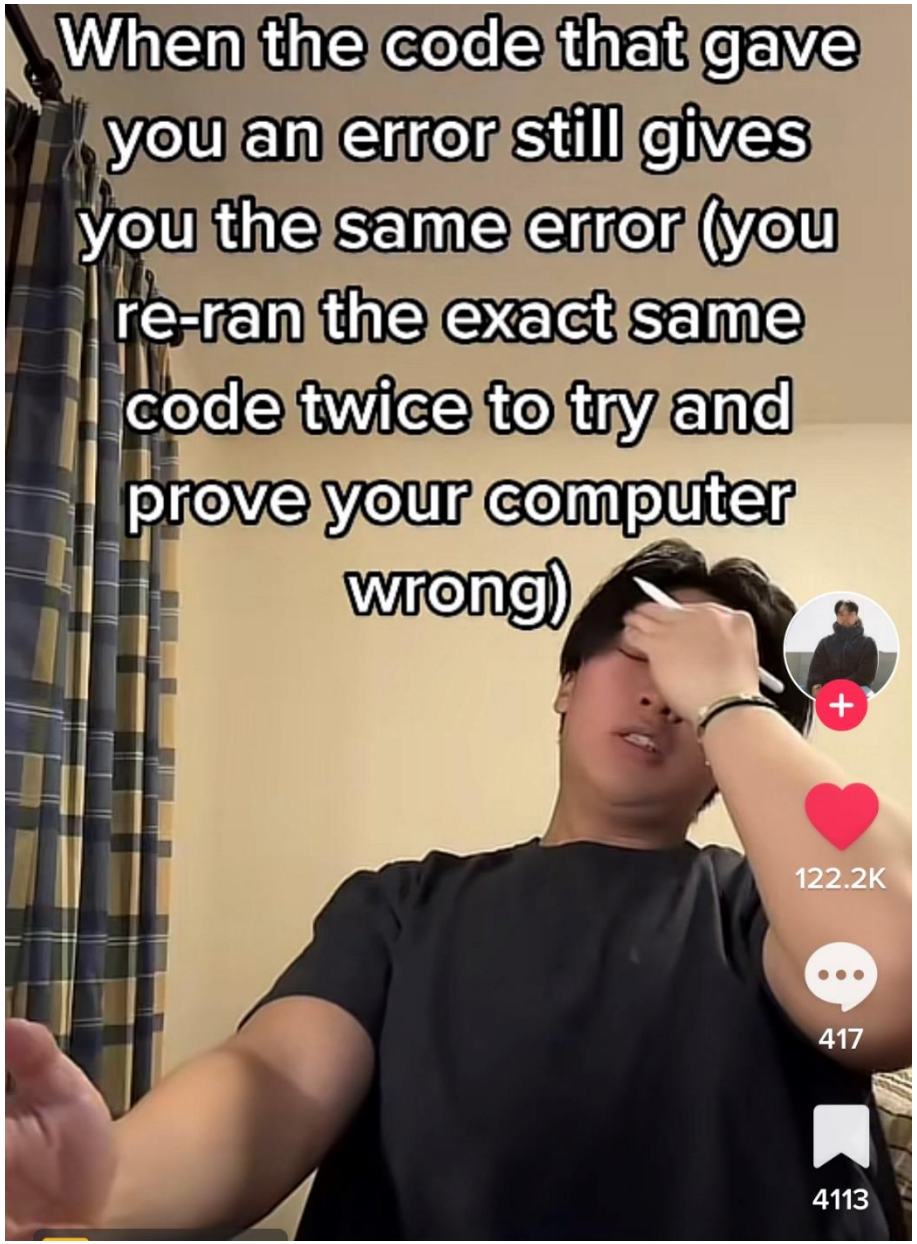
Announcements

Come get your midterm exam

- Exam average was 83
- Don’t stress if you didn’t do well
- Make sure I calculated your score correctly

Lab 7 due tomorrow

Class Registration

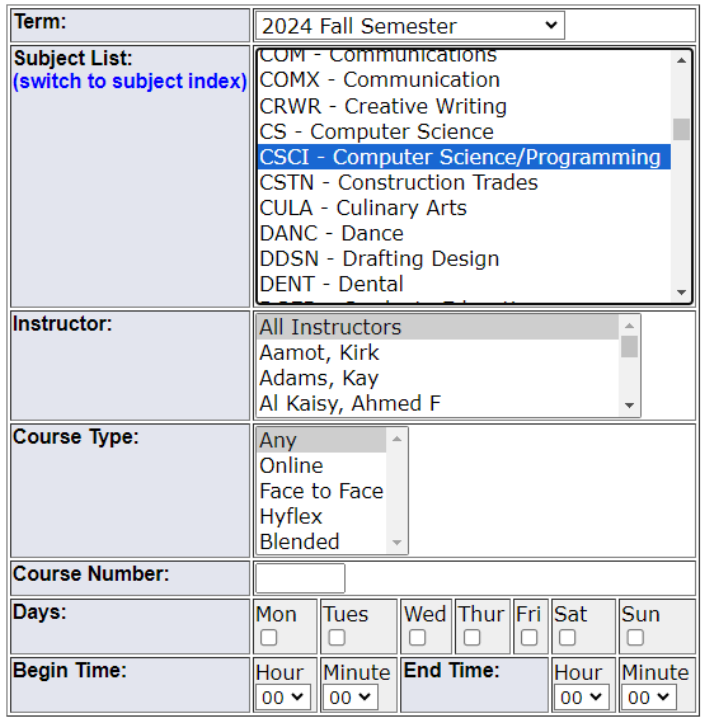


Lab 7

Class Registration

Next Class to Register for:

CSCI 232- Data Structures and Algorithms
(Offered in Summer 2024 and Fall 2024)



Other classes that may be of interest:

CS 145 – Web Design
CSCI 112- Programming in C

CSCI 215 – Social and Ethical issues in Computer Science
CSCI 204 – Multimedia Dev Methods (Game Design)
CSCI 291- Introduction to Data Science (NEW)

Resources Available to you

Student Success Center - Spring 2024

Tutoring Schedule - Barnard Hall 259: Monday, January 22nd - Friday, May 3rd

Schedule	Monday	Tuesday	Wednesday	Thursday	Friday
8:00 a.m.					
9:00 a.m.			Alex Ellingsen	Muzhou Chen	Kaden Bach
10:00 a.m.	Sultan Yarylgassimov	Ruby Martin Katie Harmon		Muzhou Chen	Gerard Shu Fuhnwi
11:00 a.m.	Sultan Yarylgassimov	Riley Slater Sage James	Jack Ruder	Nicholas Addotey Ryan Johnson	Gerard Shu Fuhnwi
Noon	Asibul Islam Shahnaj Mou	Riley Slater	Jack Ruder	Nicholas Addotey	Jared Matury Matthew Phillips
1:10 p.m.	Jake Rivers	Joshua Bowen	Karishma Rahman	Muhammad Bhatti	
2:10 p.m.	Angelo Porcello Gideon Popoola	Racquel Bowen Muhammad Arju	Gideon Popoola Karishma Rahman	Nishu Nath Muhammad Bhatti	
3:10 p.m.	Angelo Porcello Brayden Miller	Muhammad Arju Justin Mau	Shama Maganur Fatima Ododo	Nishu Nath	
4:10 p.m.		Justin Mau	Shama Maganur Fatima Ododo		

CS Tutoring Center: Barnard Hall 259

Teaching Assistants/Graders

- Section 005- Fatima Ododo
- Email: fatima.ododo@student.montana.edu
- Office Hours: Wednesdays 3:10 - 5:10 PM in Barnard Hall 259

- Section 004- Shama Maganur
- Email: shama.maganur@gmail.com
- Office Hours: Wednesdays 3:10 - 5:10 PM in Barnard Hall 259

- Section 003- Shama Maganur
- Email: shama.maganur@gmail.com
- Office Hours: Wednesdays 3:10 - 5:10 PM in Barnard Hall 259

- Section 006- Fatima Ododo
- Email: fatima.ododo@student.montana.edu
- Office Hours: Wednesdays 3:10 - 5:10 PM in Barnard Hall 259

TA Office Hours and Email

Smarty Cats Tutoring

Reese Pearsall
Email: reese.pearsall@montana.edu
Office Hours: Monday and Wednesday 1:00 - 2:00 PM, Tuesday and Friday 12:10 - 1:00 PM
Office: Barnard Hall 361
Discord: @reese_p

My email, Discord, office hours

Big-O

Big-O notation is a way to describe the running-time/time complexity of an algorithm regarding the number of operations that are executed in the algorithm (in relation to some input **`n`**)

- Focus on worst-case scenario, and how the algorithm scales as **`n`** gets really big

Big-O

Big-O notation is a way to describe the running-time/time complexity of an algorithm regarding the number of operations that are executed in the algorithm (in relation to some input **`n`**)

- Focus on worst-case scenario, and how the algorithm scales as **`n`** gets really big

A very powerful computer and a very weak computer running the same algorithm will both execute the same number of operations (the speed at which they execute these operations will be different)

Takeaway: the asymptotic running time (the big-o running time) will be the same for each computer

Big-O

Big-O notation is a way to describe the running-time/time complexity of an algorithm regarding the number of operations that are executed in the algorithm (in relation to some input **`n`**)

- Focus on worst-case scenario, and how the algorithm scales as **`n`** gets really big

To find the total running time of an algorithm, we calculate the running-time of each operation in the algorithm and then add everything together

- In Big-O, we can drop non-dominant factors and multiplicative constants (coefficients)

O(n) + O(n) + O(n): Total running time = O(3n) ∈ **O(n)** **Where n is _____**

Big-O

```
public int print_array(int[] array) {
     for(int i = 0; i < array.length; i++)
{
          System.out.println(array[i]);
     }
}
```

O(n)

```
public int print_array(int[] array) {
     for(int i = 0; i < array.length; i++)
{
          System.out.println(array[i]);
     }
}
```

O(n)

The same algorithm will have the same asymptotic running time no matter what the computer's hardware is

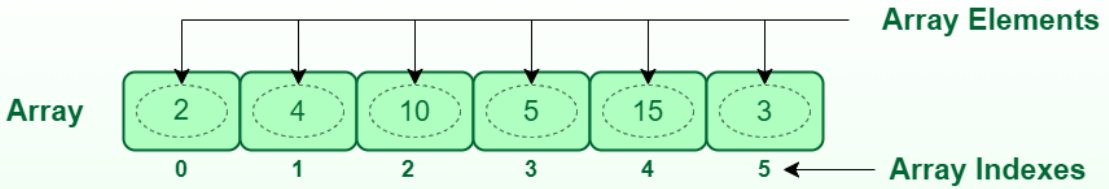
```
public int print_array(int[] array) {
     for(int i = 0; i < array.length; i++)
{
          System.out.println(array[i]);
     }
}
```

O(n)

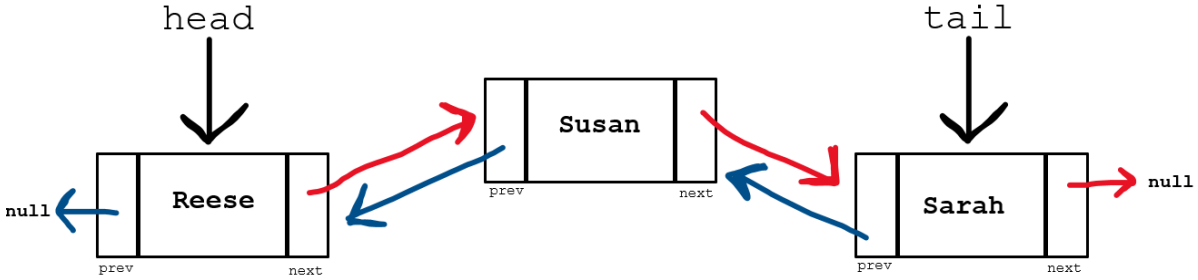
```
public int print_array(int[] array) {
     for(int i = 0; i < array.length; i++)
{
          System.out.println(array[i]);
}
```

O(n)

Data Structures so far:

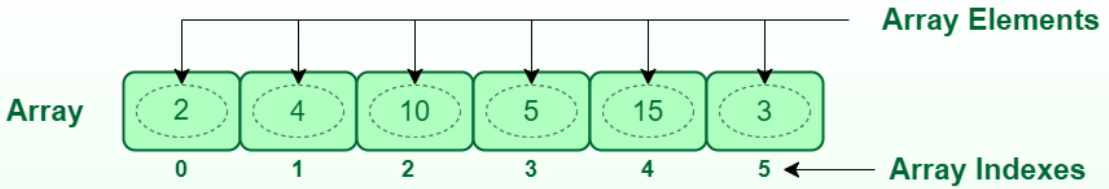


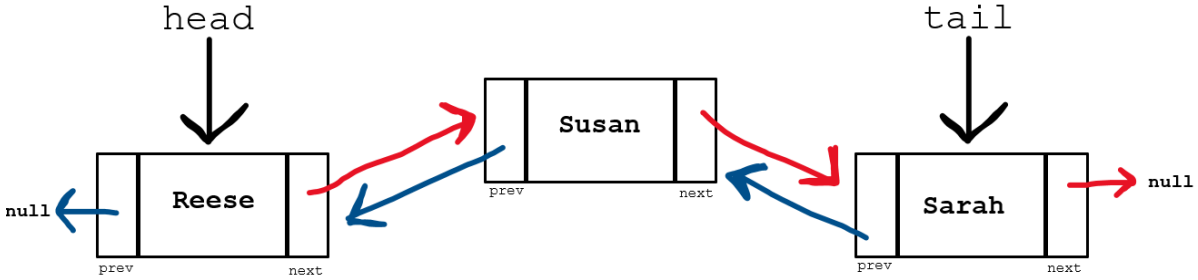
ArrayLists (Arrays)



Linked Lists

Data Structures so far:





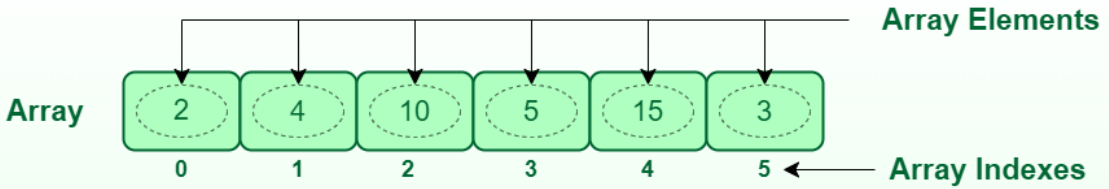
ArrayLists (Arrays)

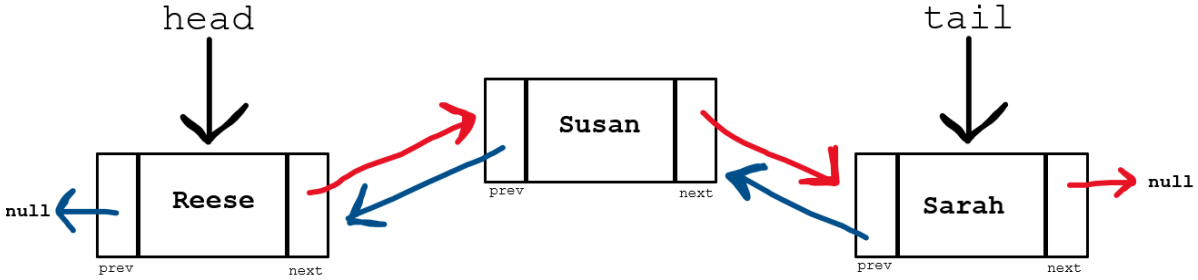
Can hold one data type

Linked Lists

~~Can hold multiple data types~~

Data Structures so far:





ArrayLists (Arrays)

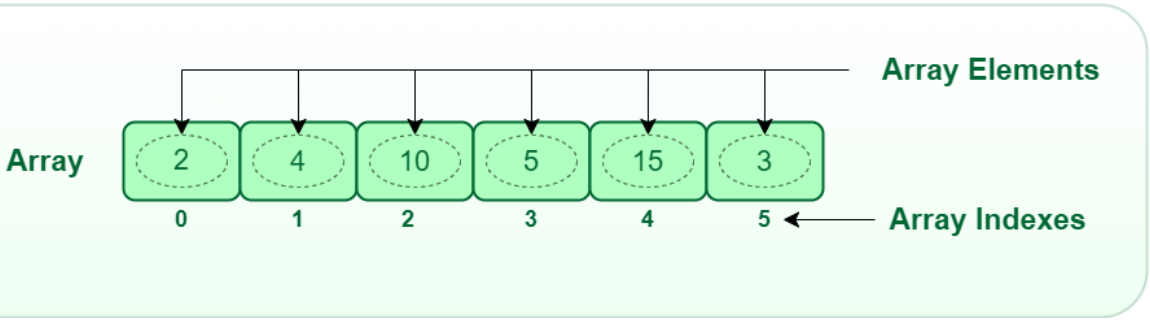
Can hold one data type

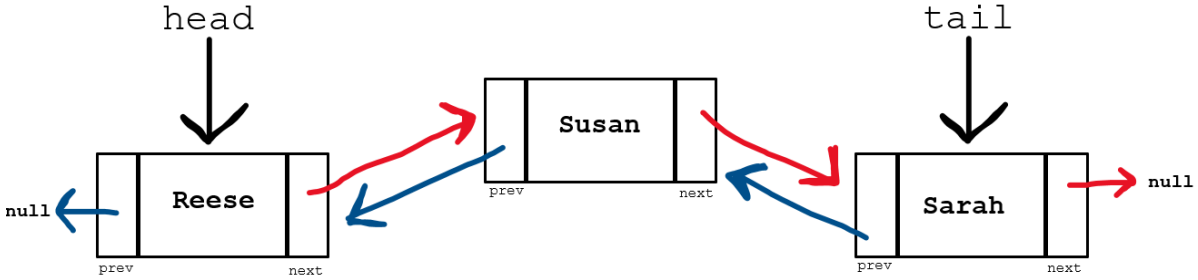
- Can also store objects, which allow for multiple data types

Linked Lists

Nodes in the linked list can hold multiple data types

Data Structures so far:





ArrayLists (Arrays)

Can hold one data type

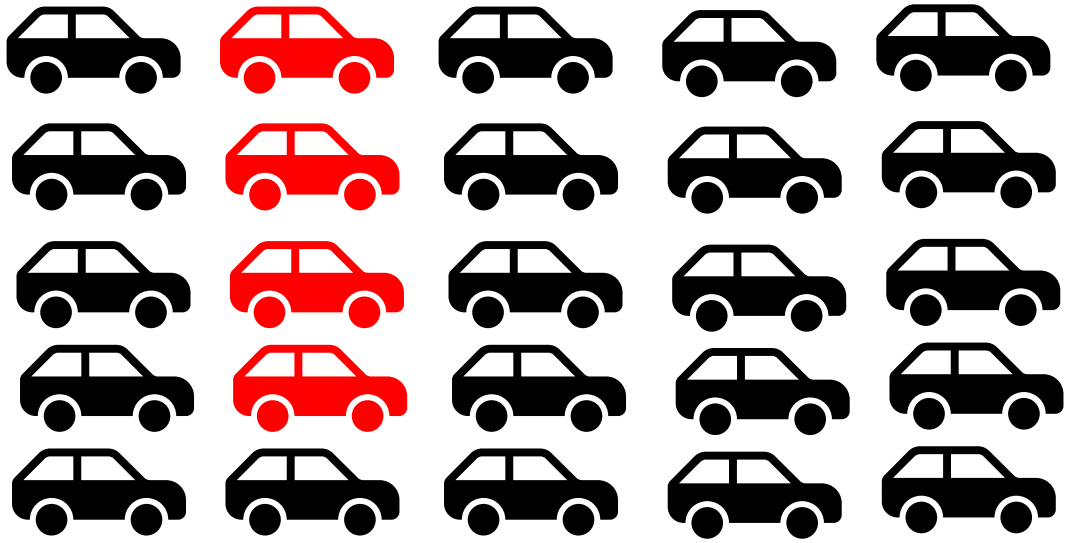
- Can also store objects, which allow for multiple data types

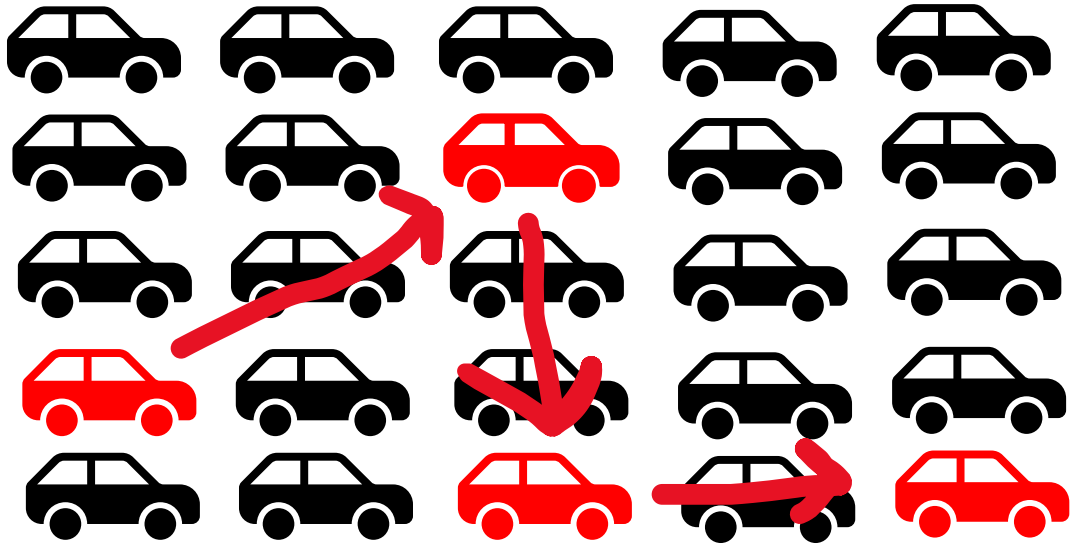
Entire array is stored at a **contiguous** spot in memory

Linked Lists

Nodes in the linked list can hold multiple data types

Linked list nodes are stored at **non-contiguous** spots in memory





Can hold one data type

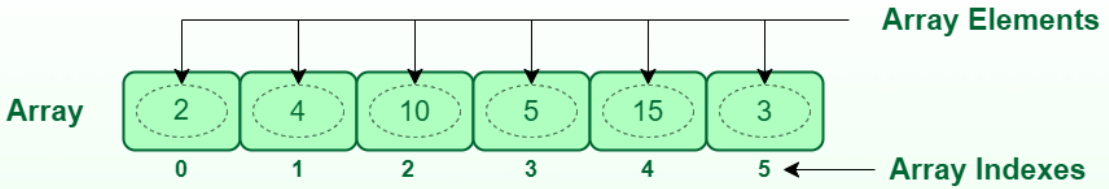
- Can also store objects, which allow for multiple data types

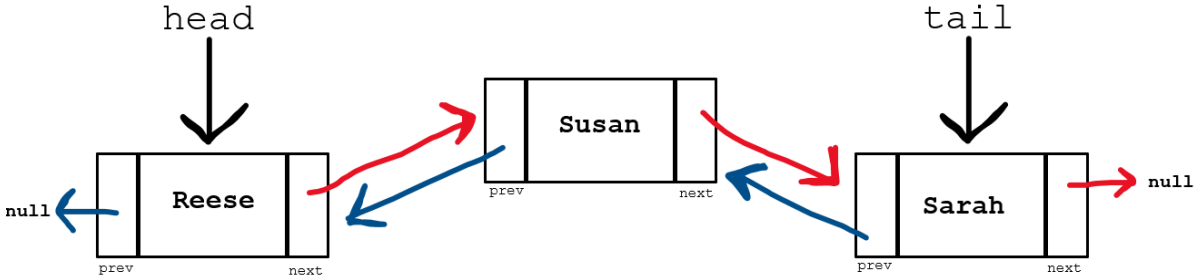
Nodes in the linked list can hold multiple data types

Entire array is stored at a **contiguous** spot in memory

Linked list nodes are stored at **non-contiguous** spots in memory

Data Structures so far:





ArrayLists (Arrays)

Objects in the ArrayLists can hold multiple data types

Entire array is stored at a **contiguous** spot in memory

Elements are ordered by index

(Easy to go directly to array spot)

Linked Lists

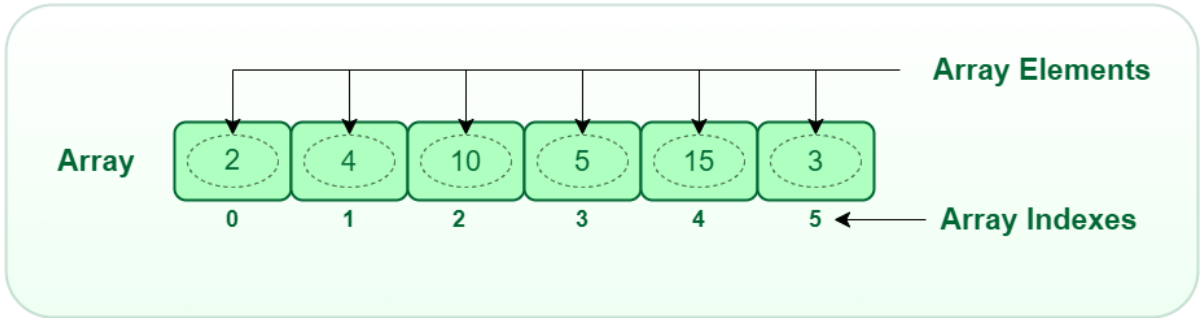
Nodes in the linked list can hold multiple data types

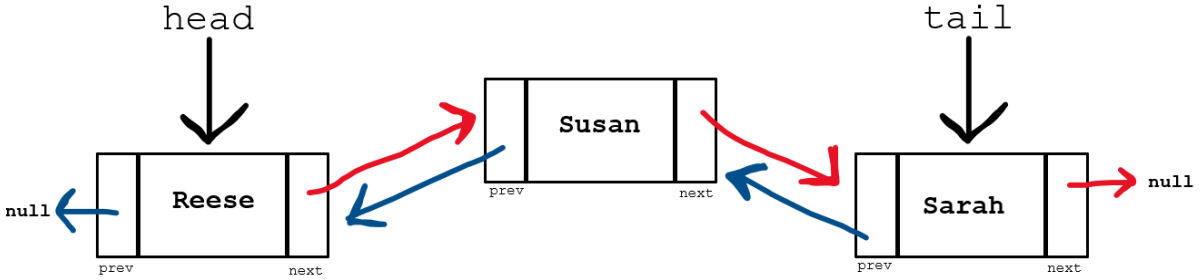
Linked list nodes are stored at **non-contiguous** spots in memory

Elements are ordered by their next pointer

(Must traverse from the head to reach node)

Data Structures so far:





ArrayLists (Arrays)

Can hold one data type

- Can also store objects, which allow for multiple data types

Entire array is stored at a **contiguous** spot in memory

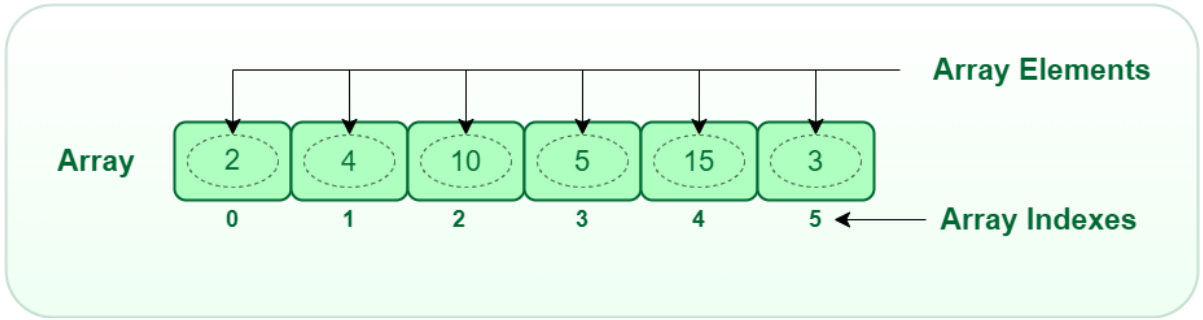
Linked Lists

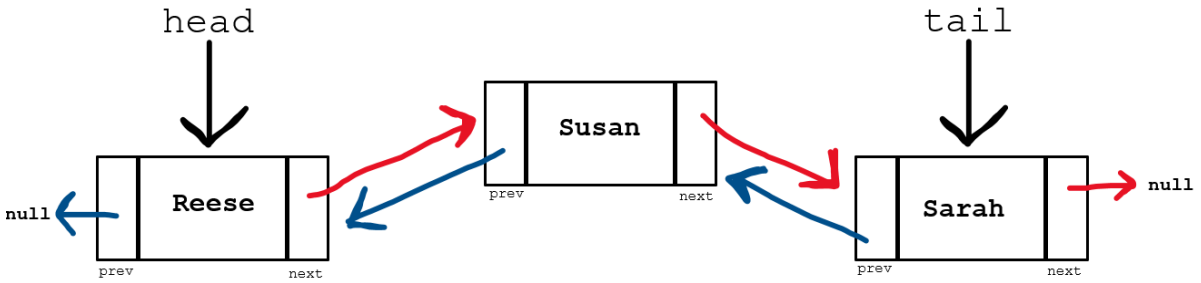
Nodes in the linked list can hold multiple data types

Linked list nodes are stored at **non-contiguous** spots in memory

Traversing a linked list requires more work than traversing an array

Data Structures so far:





ArrayLists (Arrays)

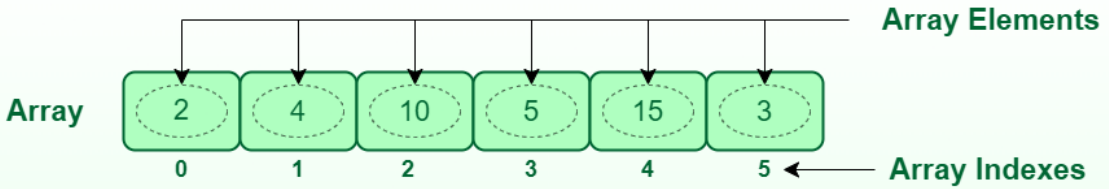
Can add new elements to data structure (resizable)

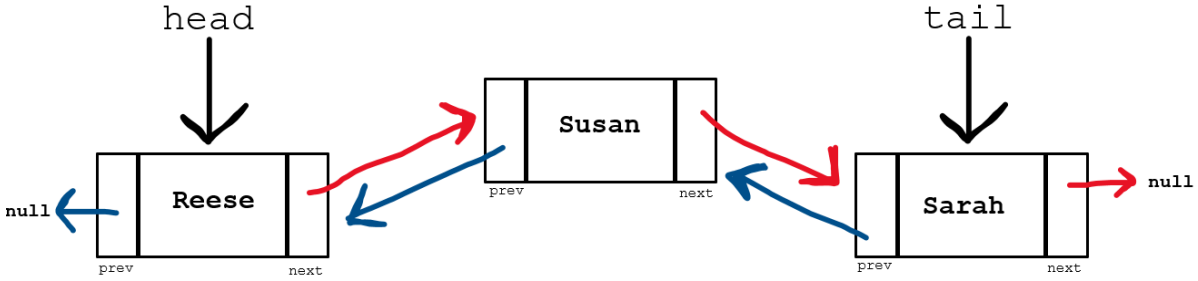
Linked Lists

Can add new elements to data structure (resizable)

Both data structures can grow dynamically, and new elements can be added, but they way they add new elements is **drastically** different

Data Structures so far:





ArrayLists (Arrays)

Linked Lists

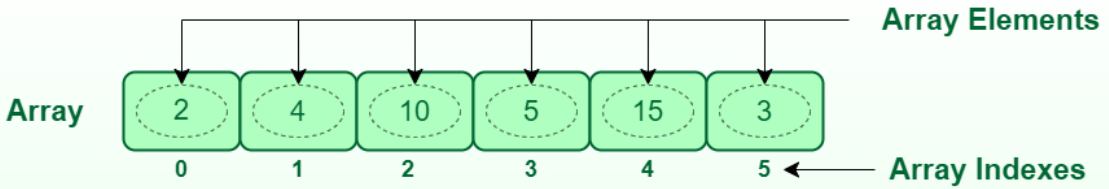
```
int[] newArray = new int[myArray.length + 1];
for(int i = 0; i < myArray.length; i++) {
            newArray[i] = myArray[i];
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

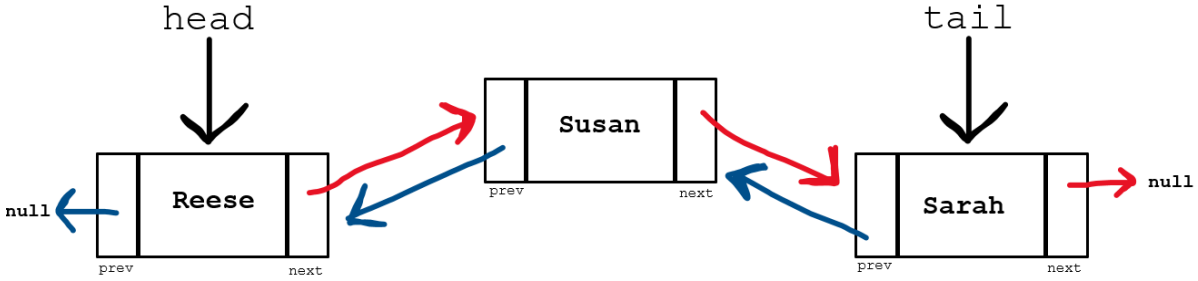
Create a brand-new array, copy everything over from old array

```
public void addToFront(Node newNode) {
    if(head == null) {
        head = newNode;
    }
    else {
        newNode.setNext(head);
        head = newNode;
    }
}
```

Update pointers

Data Structures so far:





ArrayLists (Arrays)

Linked Lists

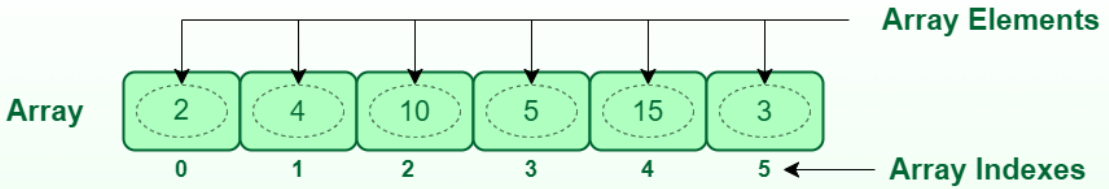
```
int[] newArray = new int[myArray.length + 1];
for(int i = 0; i < myArray.length; i++) {
            newArray[i] = myArray[i];
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

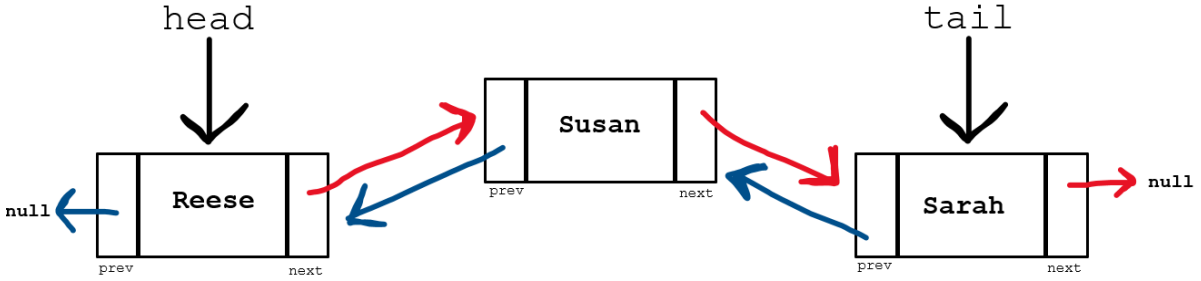
Create a brand-new array, copy everything over from old array O(n)

```
public void addToFront(Node newNode) {
    if(head == null) {
        head = newNode;
    }
    else {
        newNode.setNext(head);
        head = newNode;
    }
}
```

Update pointers O(1)

Data Structures so far:





ArrayLists (Arrays)

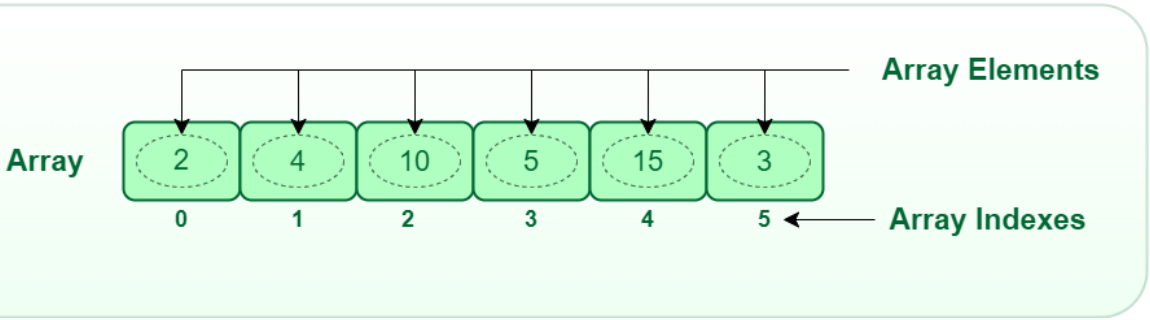
Linked Lists

```
int[] newArray = new int[myArray.length + 1];
for(int i = 0; i < myArray.length; i++) {
            newArray[i] = myArray[i];
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

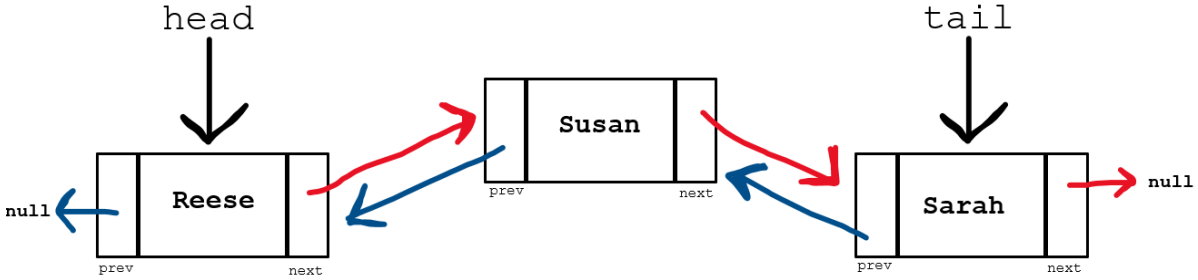
```
public void addToFront(Node newNode) {
    if(head == null) {
        head = newNode;
    }
    else {
        newNode.setNext(head);
        head = newNode;
    }
}
```

Takeaway: Adding a new element to an ArrayList requires much more work than adding a new element to a Linked List

Data Structures so far:



ArrayLists (Arrays)



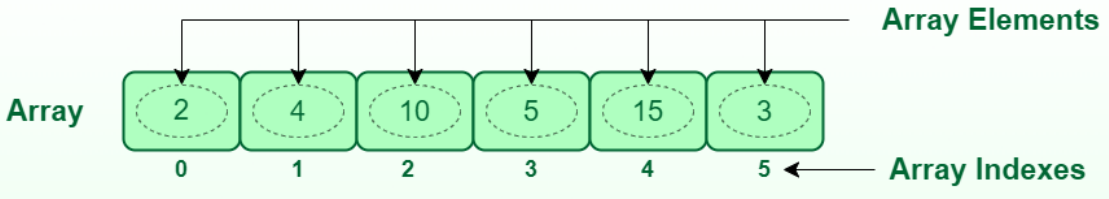
Linked Lists

Arrays are generally much easier to sort than Nodes in a Linked List

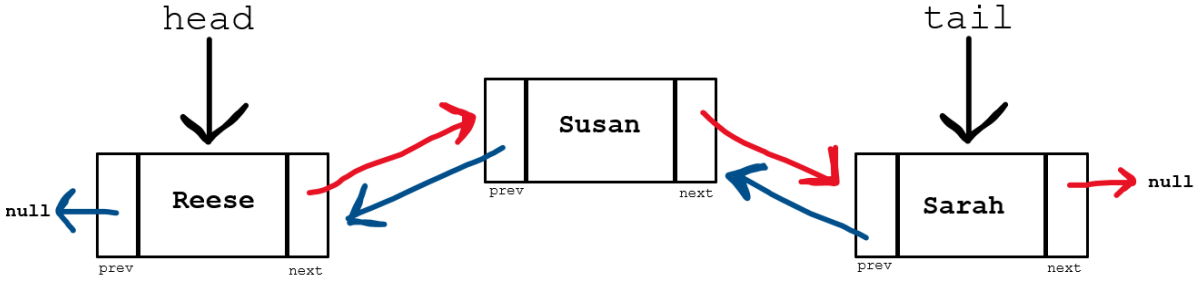
If you are constantly needing to add new elements to the data structure, using a Linked List requires much less work in the long run

Arrays are more memory efficient (adding is not very memory efficient though)

Data Structures so far:



ArrayLists (Arrays)



Linked Lists

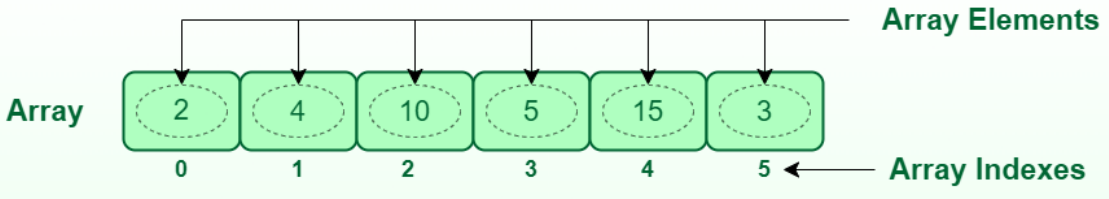
When to use each data structure?

It depends on ***how you are using your data*** and ***if you know how much data you have***

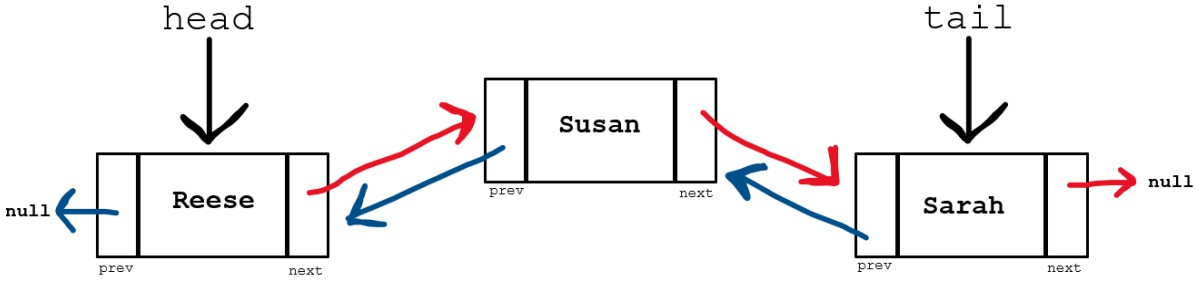
If you don't know how much data you need to store, or if you are constantly needing to add new elements to the data structure → **Linked Lists**

If you know how much data you need to store, and if you can add all your data at once → **Arrays/ArrayLists**

Data Structures so far:



ArrayLists (Arrays)



Linked Lists

These two data structures are implementations of a **List** Abstract Data Type (ADT)

ADT is a class whose behavior is defined by a set of operations and how a user interacts with it.

A list data type must be able to **get** an element, **add** an element, **remove** an element, etc
→ How they do these operations is up to the subclass (LL and AL)

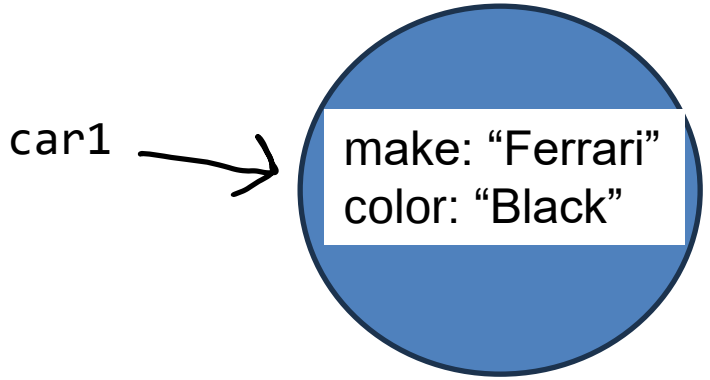
As programmers, we use handy methods that were written by other people that allows us to use these data structures

References

```
Car car1 = new Car(“Ferrari”,”Black”);
```

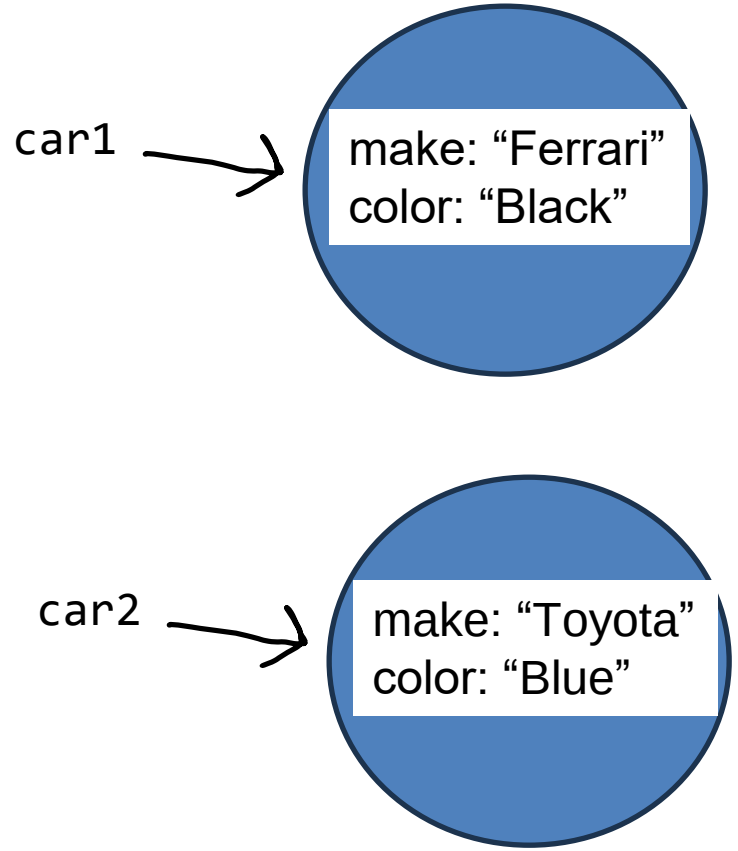
References

```
Car car1 = new Car(“Ferrari”,”Black”);
```



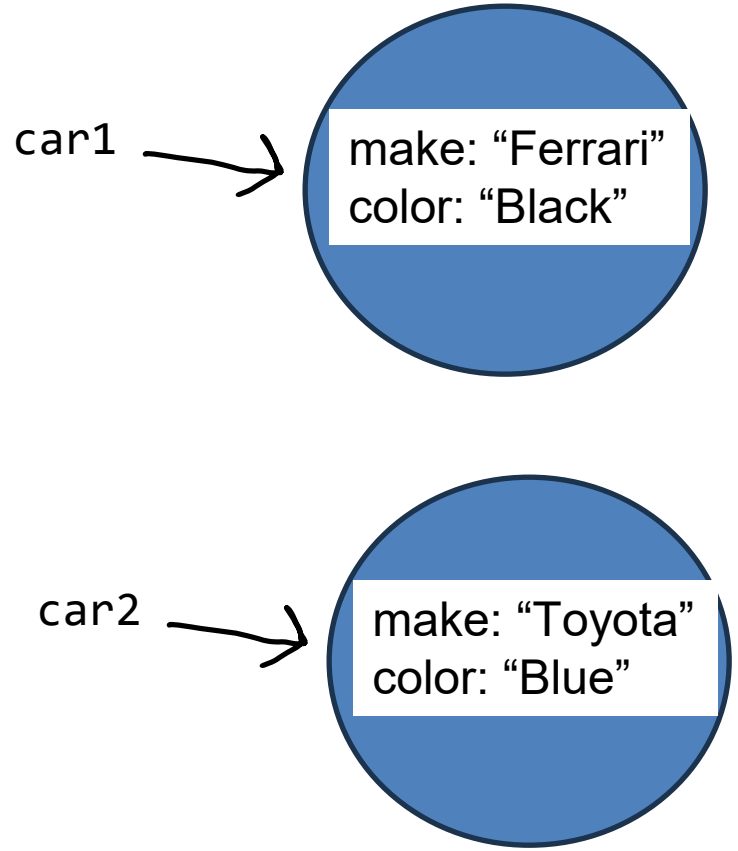
References

```
Car car1 = new Car(“Ferrari”,”Black”);
Car car2 = new Car(“Toyota”,”Blue”);
```



References

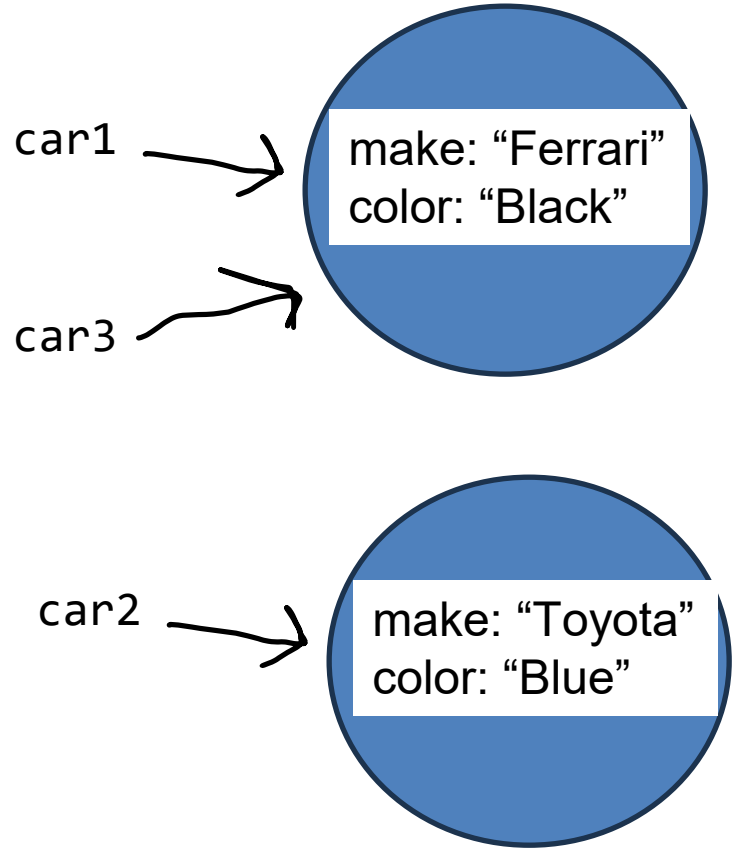
```
Car car1 = new Car(“Ferrari”,”Black”);
Car car2 = new Car(“Toyota”,”Blue”);
Car car3 = car1;
```



References

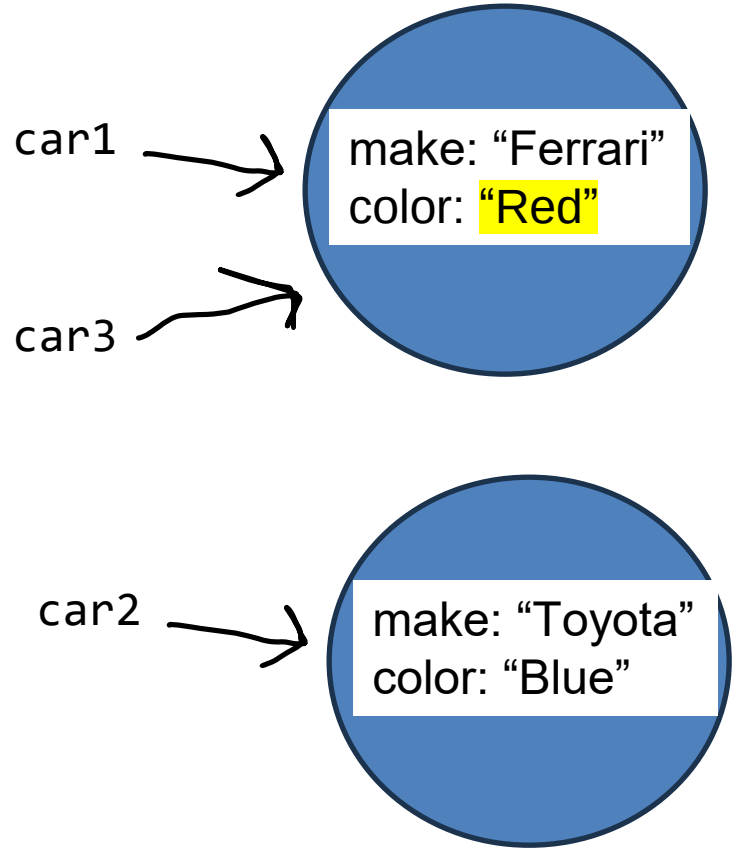
```
Car car1 = new Car(“Ferrari”,”Black”);
Car car2 = new Car(“Toyota”,”Blue”);
Car car3 = car1;
```

The new keyword is not used, so a new object is **not** created. Instead, it will point to the same object that car1 is pointing to



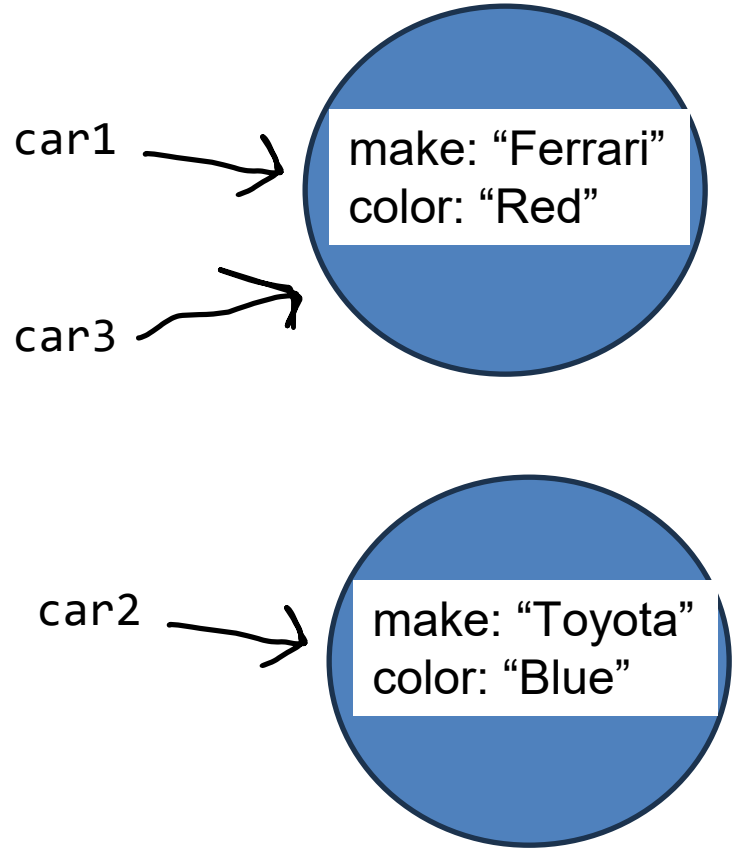
References

```
Car car1 = new Car(“Ferrari”,”Black”);
Car car2 = new Car(“Toyota”,”Blue”);
Car car3 = car1;

car3.set_color(“Red”);
```



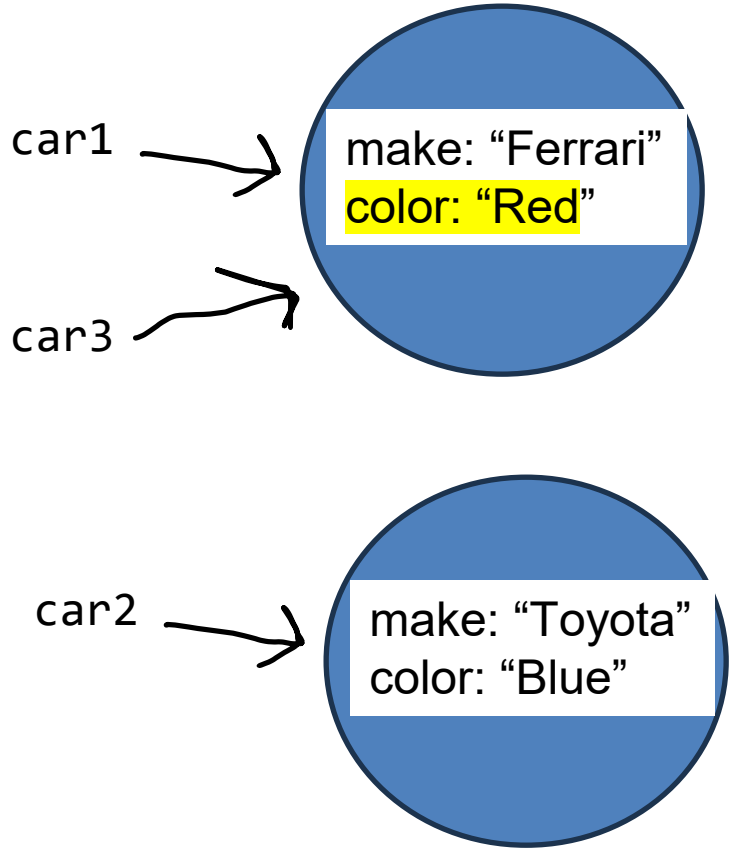
References

```
Car car1 = new Car(“Ferrari”,”Black”);
Car car2 = new Car(“Toyota”,”Blue”);
Car car3 = car1;

car3.set_color(“Red”);
System.out.println(car1.getColor());
```



```
Car car1 = new Car(“Ferrari”,”Black”);
Car car2 = new Car(“Toyota”,”Blue”);
Car car3 = car1;

car3.set_color(“Red”);
System.out.println(car1.getColor());
```

Red



Abstract Classes are restricted classes that cannot be used to create objects. To access it, it must be inherited from another class.

```
public abstract class Employee {
    ...
}
```

```
Employee e = new Employee("Sally", 4444, 123456);
```

You **cannot** create instances of an abstract class.

```
Accountant kevin = new Accountant("Kevin Malone", 4444, 42000, 'C');
```

Instead, we use objects from another class *that inherits from the abstract class*

The Linked List Class

We will no longer be writing our own Linked List class, instead we will now import the Java-provided Linked List Class

```
import java.util.LinkedList;
```

We will no longer be writing our own Linked List class, instead we will now import the Java-provided Linked List Class

```
import java.util.LinkedList;
```

```
LinkedList<String> names = new LinkedList<String>();
```

The data type the linked list will be holding

Reference variable for LL

The Linked List Class

https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Doubly-linked list implementation of the `List` and `Deque` interfaces. Implements all optional list operations, and permits all elements (including `null`).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

The **documentation** describe how the LinkedList class was implemented, and all the methods/operations we can do with the Linked List class

Methods

Modifier and Type	Method and Description
boolean	`add(E e)` Appends the specified element to the end of this list.
void	`add(int index, E element)` Inserts the specified element at the specified position in this list.
boolean	`addAll(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the sp
boolean	`addAll(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	`addFirst(E e)` Inserts the specified element at the beginning of this list.
void	`addLast(E e)` Appends the specified element to the end of this list.
void	`clear()` Removes all of the elements from this list.
Object	`clone()` Returns a shallow copy of this `LinkedList`.
boolean	`contains(Object o)` Returns `true` if this list contains the specified element.
Iterator<E>	`descendingIterator()` Returns an iterator over the elements in this deque in reverse sequential order.
E	`element()` Retrieves, but does not remove, the head (first element) of this list.
E	`get(int index)` Returns the element at the specified position in this list.
E	`getFirst()` Returns the first element in this list.

The Linked List Class

```
import java.util.LinkedList;

public class march20demo {

    public static void main(String[] args) {

        LinkedList<String> names = new LinkedList<String>();

        names.add("Reese");
        names.add("Spencer");
        names.add("Susan");

        System.out.println(names);


    }

}
```