Advanced Topics

- In this lecture we will discuss
  - LinkedLists
  - Maps
  - Java Collections Framework
  - Streams and lambda expressions
  - Other topics as we have time

Your future in CS

I used to include this on my slides, but since these slides have changed - going to just leave it up here for every notebook. I get a lot of questions about more programming courses, the concentrations, and minors in computer science. Here is a brief reminder.

CS 165 – Next Course In Sequence, also consider CS 220 (math and stats especially)

- CO Jobs Report 2021 – 77% of *all* new jobs in Colorado require programming
- 60% of all STEM jobs requires *advanced* (200-300 level)
- 31% of all Bachelor of Arts degree titled jobs also required coding skills
- 2016 Report found on average jobs that require coding skills paid $22,000 more

- Concentrations in CS:

  - Computer science has a number of concentrations.
    - General concentration is the most flexible, and even allows students to double major or minor pretty easily.
    - Software Engineering
    - Computing Systems
    - Human Centered Computing
    - Networks and Security
    - Artificial Intelligence
    - Computer Science Education.
  - Minors:
    - Minor in Computer Science - choose your own adventure minor
    - Minor in Machine Learning - popular with stats/math, and engineering
    - Minor in Bioinformatics - Biology + Computer Science

Lists

There are two well known list types in java ArrayLists and LinkedLists

ArrayLists

- Underlining structure is an array
- which means
  - Access time to an element can be $O(1)$ as you have direct indexing.
  - Finding an element is $O(n)$ because you have to go through every element
  - What happens when the Array is full?
    - Let's see.

In [1]:

```
public class MyArrayList {
    Object[] values = new Object[10]; // default start
    int size = 0;

    public void add(Object obj) {
        if(size >= values.length) doubleValues(); // oh no, it is full!
        values[size++] = obj; // store at the end
    }

    public void insert(int index, Object obj) {
        if(index >= size) throw new IndexOutOfBoundsException();
        if(size >= values.length) doubleValues(); // just in case
        // ok, to insert we have to move every value 'down' the list!
        System.out.print("Shifting - ");
        for(int i = size-1; i >=index; i--) {
            System.out.print(".");
            values[i+1] = values[i];
        }
        System.out.println();
        values[index] = obj;
        size++;
    }

    private void doubleValues() {
        System.out.println("Running double");
        Object[] values2 = new Object[values.length*2];
        for(int i = 0; i < values.length; i++) {
            values2[i] = values[i]; // copies every object into the new list
        }
        values = values2; // resets the first variable to the new array created
        System.gc(); //forces the garbage collector to look for memory to free
    }
    public String toString() {
        return Arrays.toString(values);
    }

}
```

In [2]:

```
MyArrayList list = new MyArrayList();
System.out.println(list);
for(int i = 0; i < 11; i++) {
    list.add(new Integer(i));
}
System.out.println(list);
for(int i = 0; i < 1000; i++) {
    list.add(new Integer(i));
}
```

```
[null, null, null, null, null, null, null, null, null, null]
Running double
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, null, null, null, null, null, null, null, null, nu
ll]
Running double
Running double
Running double
Running double
Running double
Running double
```

In [3]:

```
MyArrayList list = new MyArrayList();
System.out.println(list);
for(int i = 0; i < 8; i++) {
    list.add(new Integer(i));
}
System.out.println(list);
list.insert(7, new Integer(-7));
System.out.println(list);
list.insert(0, new Integer(-10)); // it has to move all of them down one!
System.out.println(list);
list.insert(0, new Integer(-1));
System.out.println(list);
```

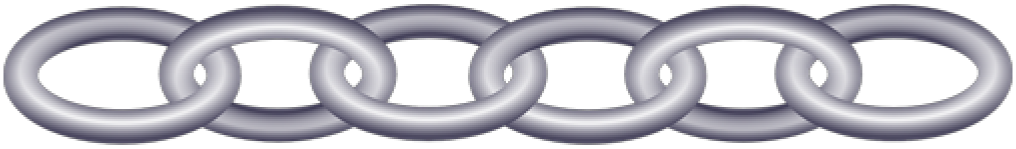
```
[null, null, null, null, null, null, null, null, null, null]
[0, 1, 2, 3, 4, 5, 6, 7, null, null]
Shifting - .
[0, 1, 2, 3, 4, 5, 6, -7, 7, null]
Shifting - .........
[-10, 0, 1, 2, 3, 4, 5, 6, -7, 7]
Running double
Shifting - ..........
[-1, -10, 0, 1, 2, 3, 4, 5, 6, -7, 7, null, null, null, null, null, null, null, null,
null]
```

- Inserting at the start -- is expensive!
    - Especially if you are at a size max
- Inserting at the end is cheap, unless you are at a size max

LIFO and FIFO

- Last in first out (LIFO) - also known as a `stack`
    - Very common process for algorithms
    - Works well with an ArrayList, as you are always adding and inserting at an
- First in First Out (FIFO) - also known as a `queue`
    - ArrayList are in the worst case scenario as they are always shifting values!

LinkedList



- Another common type of list
- The idea
    - Each "node" contains a value
    - And then a reference to the next node in the list
        - Think of a chain!
    - We also keep pointers to the front and end of the chain
        - Adding something at the beginning means
            - Create a new node
            - Set the value to the node
            - Set next node to the be start of the list
            - set the start of the list to the new node
            - cost? $O(1)$
        - Adding at the end?
            - Same pattern, but with the last node!
            - cost? $O(1)$
        - However, accessing a node that isn't the beginning or the end?
            - cost? $O(n)$
            - As you have to traverse the entire list to find it!
- You will implement this in CS 165

LinkedList or ArrayList?

- If you need quick access to the elements, and you are not constantly removing and adding elements
    - ArrayList
- If you need to constantly remove and add elements at the start or end
    - LinkedList
- How do you pick which in your code?
    - Use the **power** of inheritance and polymorphism

In [4]:

```java
public class MyStack<T> {   // the T is called a 'generic', meaning i can define the t
    private List<T> list = new ArrayList<>();
    public void push(T value) {
        list.add(0, value);
    }
    public T pop() {
        return list.remove(0);
    }
}
```

In [5]:

```java
import java.time.Instant;
```

```
MyStack<Integer> stack = new MyStack<>();  // it now uses Integer in place of T

Instant start = java.time.Instant.now();
for(int i = 0; i < 1000000; i++)
    stack.push(new Integer(i));
Instant end = java.time.Instant.now();
System.out.println("ArrayList Version Done: " + java.time.Duration.between(start, end)
```

```
ArrayList Version Done: 116163
```

In [6]:

```
public class MyStack<T> {
    private List<T> list = new LinkedList<>(); // notice I only have to change this or
    public void push(T value) {
        list.add(0, value);
    }
    public T pop() {
        return list.remove(0);
    }
}
```

In [7]:

```
// now using the LinkedList Version

MyStack<Integer> stack = new MyStack<>();  // it now uses Integer in place of T

Instant start = java.time.Instant.now();
for(int i = 0; i < 1000000; i++)
    stack.push(new Integer(i));
Instant end = java.time.Instant.now();
System.out.println("LinkedList Version Done: " + java.time.Duration.between(start, enc
```

```
LinkedList Version Done: 217
```

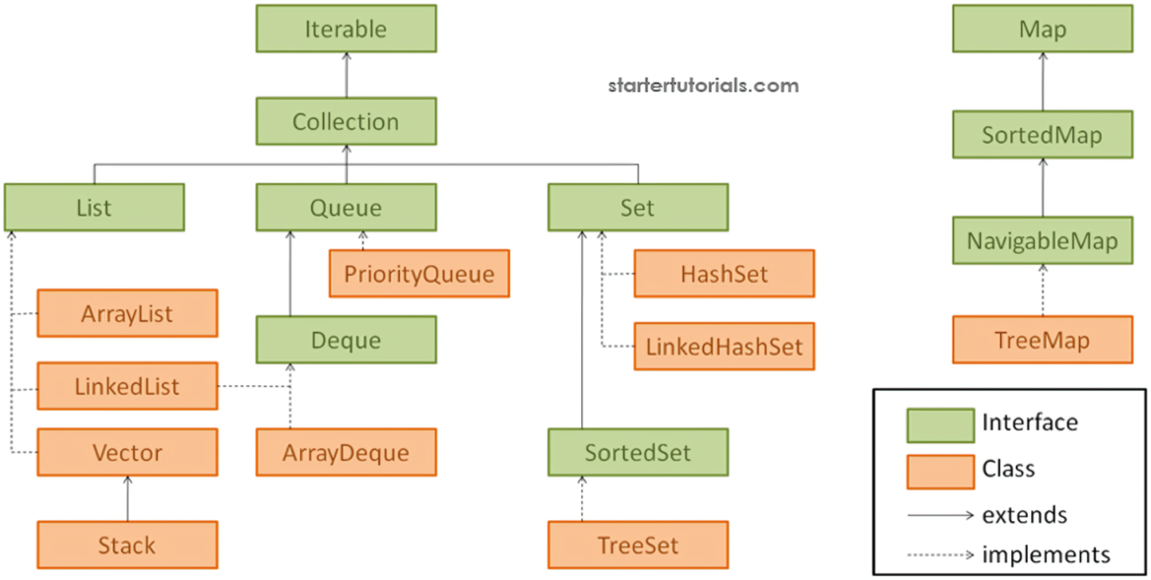
Time Difference

In the example above - 1 minute and 56 seconds, compared to 0.4s!!

Knowing which data structure to use matters!

Collections Framework

- Java SDK has a massive framework
- A very popular piece is the collection framework
    - Both `ArrayList` and `LinkedList` are implemented for you
    - They both implement `List`
- They also have a lot more features and classes



- For the most part we 'declare' as the interfaces
- Initialize as the classes we need specifically
- `List`
  - We are already using, notice Vector and Stack
- `Queue`
  - Interface that focuses on LIFO and FIFO style commands
  - Queues often determine order of actions (like who gets the internet next on a router!)
- `Set`
  - Like a mathematical set
  - Order does not matter
  - Unique values only
  - You must override `.equals()` and `.getHash()` if you store your objects in a set
- `SortedSet`
  - Like set, but an order is kept
  - You must implement `Comparable` and override `.equals()` for it to work!
- `Map`
  - Let's talk about this!
  - And we will use a class missing from the diagram, a HashMap

Maps

- So far to access an element in an list, we need an index from $0..n$
  - What if we could name indices?
  - Based on the name, we could pull elements directly
- Stated another name
  - What if we could **map** index names to values
  - We have that in most languages, in Java it is `HashMap` or `TreeMap` (if you need the values sorted)

In [8]:

```
Map<String, String> contacts = new HashMap<>();
contacts.put("awonder", "awonder@wonderland.colostate.edu");
contacts.put("queen", "redqueen@wonderland.colostate.edu");
contacts.put("hatter", "madhatter@wonderland.colostate.edu");
System.out.println(contacts.get("queen"));
```

```
redqueen@wonderland.colostate.edu
```

Notice that we can access the value by the "key". We can also change the value by the key.

In [9]:

```
contacts.put("queen", "offwiththeirhead@wonderland.colostate.edu");

System.out.println(contacts.get("queen"));
```

```
offwiththeirhead@wonderland.colostate.edu
```

In [10]:

```
System.out.println(contacts); // notice the 'order' isn't kept!
```

```
{awonder=awonder@wonderland.colostate.edu, hatter=madhatter@wonderland.colostate.edu,
queen=offwiththeirhead@wonderland.colostate.edu}
```

We can also get the keys as a **Set** with `.keySet()` and can get the values as a **Collection** with `.values()`

> Discussion - why Set for keys and Collection for values?

In [11]:

```
Set<?> keys = contacts.keySet();
Collection<?> values = contacts.values();

System.out.println(keys);
System.out.println();
System.out.println(values);
```

```
[awonder, hatter, queen]

[awonder@wonderland.colostate.edu, madhatter@wonderland.colostate.edu, offwiththeirhe
ad@wonderland.colostate.edu]
```

CS 165 == Technical Interviews

- The material you learn in CS 165 is the heart of a lot of technical interviews.
- While many of these data structures are written for you - knowing which to use when **matters**.

Going Beyond What We Teach

- Java 1.8+ introduced a lot of functional programming techniques
- Many books haven't caught up
    - This next part, you don't really learn in a class - but I felt like teaching it this year!
    - It is used in industry, so worth learning / messing around with.

Java Stream Interface

- What if we could treat our data/collection as 'stream' of information
  - We can modify that data
  - We can filter that data
  - We create new collections from that data
- While we do all that with for loops and complicated programming
  - These techniques are so common, java created the stream interface
  - Many modern languages have something similar
  - Kotlin defaults to streams as part of their collections

In [12]:

```
import  java.util.stream.*;

List<String> names = Arrays.asList("Hatter", "", "Alice", "Red Queen", "Cat", "", "Whi

System.out.println(names); // oh no, our data has empty strings in it!
// we could create a loop that removes each one, or we could do the following

List<String> filtered = names.stream().filter(name -> !name.isEmpty()).collect(Collect
System.out.println(filtered); // one line removes them.
```

```
[Hatter, , Alice, Red Queen, Cat, , White Queen]
[Hatter, Alice, Red Queen, Cat, White Queen]
```

Lambda Expressions

- What is that -> magic?
- It is actually a lambda expression
  - A method defined on the 'fly'
  - It takes the value for each item in the list, passes it into `name` as the parameter
  - It then executes the code 'returning' the answer which is a Boolean
  - If the answer is true, the value is kept
  - If the answer if false, the value is filtered out
- You can define your own, and often they are paired with interfaces.
  - This combined with using the interface allows for methods to be treated more like objects

In [13]:

```
interface Power {
    int pow(int x);
}

Power test = (x) ->  { return x * x;};

int value = test.pow(10);
System.out.println(value);

public static void myFunctional(Power power, int x) {
    System.out.println(power.pow(x));
}
myFunctional(test, 12);

myFunctional(x -> x * x * x, 12); // in this case, i wanted power 3, not power 2
```

```
100
144
1728
```

More Streams

Now that we understand lambda expressions, lets look at other things we can do with streams!

Map

Takes the values, applies the function, builds a new list based on the values.

In [14]:

```
List<Integer> numbers = Arrays.asList(2, 2, 3, 7, 3, 5, 10);

//get list of unique squares
List<Integer> pow2List = numbers.stream().map( x -> x*x).collect(Collectors.toList());

System.out.println(pow2List);

List<String> evenOdd = numbers.stream().map( x -> x%2==0 ? "Even" : "Odd").collect(Col
System.out.println(evenOdd);
```

```
[4, 4, 9, 49, 9, 25, 100]
[Even, Even, Odd, Odd, Odd, Odd, Even]
```

What if i only wanted unique values? use `.distinct()`

In [15]:

```
List<Integer> numbers = Arrays.asList(2, 2, 3, 7, 3, 5, 10);

//get list of unique values
List<Integer> pow2List = numbers.stream().map( x -> x*x).distinct().collect(Collectors

System.out.println(pow2List);
```

```
[4, 9, 49, 25, 100]
```

What if I wanted 10 random numbers in a list?

In [16]:

```
import java.util.Random;

Random rnd = new Random();
int[] random = rnd.ints(1, 21).limit(10).toArray(); // random.ints() returns an IntStr
System.out.println(Arrays.toString(random));
```

```
[3, 15, 6, 6, 9, 7, 18, 16, 18, 5]
```

In [27]:

```
int[] random = rnd.ints(1, 21).limit(20).sorted().toArray();
System.out.println("Sorted " + Arrays.toString(random));
System.out.println("Max value " + rnd.ints(1, 21).limit(20).max().getAsInt()); // thes
System.out.println("Min value " + rnd.ints(1, 21).limit(20).min().getAsInt());
```

```
Sorted [1, 3, 3, 6, 7, 7, 8, 9, 9, 10, 10, 11, 11, 11, 12, 14, 17, 19, 20, 20]
Max value 20
Min value 1
```

Also, we can run statistics pretty easily!

In [18]:

```
List<Integer> grades = Arrays.asList(1, 4, 2, 3, 3, 3, 0);
```

```
IntSummaryStatistics stats = grades.stream().mapToInt((x) -> x).summaryStatistics();
System.out.println("Max: " + stats.getMax());
System.out.println("Min: " + stats.getMin());
System.out.println("Sum: " + stats.getSum());
System.out.println("Average: " + stats.getAverage());

// mode is still more complicated
Map<Integer, Integer> counter = new HashMap<Integer, Integer>();
for (Integer i : grades) {
    Integer j = counter.get(i);
    counter.put(i, (j == null) ? 1 : j + 1);
}
int max = Collections.max(counter.values());
List<Integer> mode = counter.entrySet().stream()
    .filter(entry -> entry.getValue() == max)
    .map(entry -> entry.getKey())
    .collect(Collectors.toList());

System.out.println("Mode: " + mode.get(0));
```

```
Max: 4
Min: 0
Sum: 16
Average: 2.2857142857142856
Mode: 3
```

Using our own objects

This is all great, but we want to use our own objects!

You can call specific methods in an object using `Class::method`.

In [21]:

```
public class Submission  {
    final String name;
    final int id;
    final int grade;

    public Submission(String name, int id, int grade) {
        this.name = name;
        this.id = id;
        this.grade = grade;
    }

    public String getName() { return name;}
    public int getID() { return id;}
    public int getGrade() { return grade;}
    public String toString() { return String.format("%s::ID=%d::grade=%d", name, id, g
}

List<Submission> assignments = new LinkedList<>();

assignments.add(new Submission("Lab 1", 1, 4));
assignments.add(new Submission("Lab 2", 2, 3));
assignments.add(new Submission("Lab 2", 2, 4));
assignments.add(new Submission("Lab 3", 3, 3));
assignments.add(new Submission("Lab 3", 3, 0));
assignments.add(new Submission("Lab 3", 3, 0));
```

```java
assignments.add(new Submission("Lab 3", 3, 4));
System.out.println(assignments);
```

```
[Lab 1::ID=1::grade=4, Lab 2::ID=2::grade=3, Lab 2::ID=2::grade=4, Lab 3::ID=3::grade
=3, Lab 3::ID=3::grade=0, Lab 3::ID=3::grade=0, Lab 3::ID=3::grade=4]
```

In [22]:

```java
Map<Integer, List<Submission>> byId = assignments.stream().collect(Collectors.grouping

System.out.println(byId);

System.out.println();
System.out.println(byId.getOrDefault(1, List.of(new Submission("Lab 1", 1, 0))));
System.out.println(byId.getOrDefault(5, List.of(new Submission("Lab 5", 5, 0))));
```

```
{1=[Lab 1::ID=1::grade=4], 2=[Lab 2::ID=2::grade=3, Lab 2::ID=2::grade=4], 3=[Lab 3::
ID=3::grade=3, Lab 3::ID=3::grade=0, Lab 3::ID=3::grade=0, Lab 3::ID=3::grade=4]}

[Lab 1::ID=1::grade=4]
[Lab 5::ID=5::grade=0]
```

Overall

- Using Collections is extremely beneficial and common in programming
- The stream interface changes how you view collections
  - Instead of objects to iterate over
  - They become objects to Query and modify - like accessing databases.
- Lambda Expressions allows for functional programming in java
  - where functions can be defined on the fly for small quick actions
- How do we figure this all out?
  - **DIVIDE-CONQUER-GLUE**
  - Write out in small stages, test, print, guess, keep expanding.

The world of computer science is just beginning. Just remember, we spend a lot of time looking things up (the Javadoc SDK website is our friend!) - that is ok! No one has it all memorized, but knowing when, how and what to modify that is important.