Week1:

De Morgan's Law

- **De Morgan's Law**:
  Rules used to *negate* or *reverse* boolean expressions.
  - Useful when you want the opposite of a known boolean test.

Original Expression	Negated Expression	De Morgan
a && b	!(a && b)	!a || !b
a || b	!(a || b)	!a && !b

- Example:

Original Code	Negated Code then De Morgan

`if (x == 7 && y > 3) {`
`  ...`
`}`
	`if (x != 7 || y <= 3) {`
	`  ...`
	`}`

3

```
int x = 10;
while(x > 0 ) ;
{
    x--;
}
```

-That semicolon makes the while loop do nothing.

The For Each Loop

```
type[ ] arrayName = {1stvalue, 2ndvalue, 3rdvalue};

for (type value : arrayName) {
  // do something with "each" value or element of the array
  System.out.println(value);
}
```

-ACTUAL FOR EACH

Week2:

-Which of the following is the correct syntax to indicate that class A is a subclass of B?

-`public class A extends B {`

-import BigDecimal object type to avoid roundoff error

-import Objects to check and throw error for null checks (Objects.requireNonNull)

-You can declare final in Overloading constructor to have it NEVER CHANGE

-StringBuffer: Multithreading synced/safe version of StringBuilder

https://www.javatpoint.com/difference-between-stringbuffer-and-stringbuilder#:~:text=The%20String%20class%20is%20an,and%20StringBuilder%20classes%20are%20mutable.&text=StringBuffer%20is%20synchronized%20i.e.%20thread,synchronized%20i.e.%20not%20thread%20safe.

-Inheritance is building a class from another class.

Inheritance

- **inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior.
  - a way to group related classes
  - a way to share code between two or more classes

- One class can *extend* another, absorbing its data/behavior.
  - **superclass**: The parent class that is being extended.
  - **subclass**: The child class that extends the superclass and inherits its behavior.
    - Subclass gets a copy of every field and method from superclass

-Extends means using all from superclass to subclass.

Inheritance syntax

```
public class name extends superclass {
```

– Example:

```
public class Secretary extends Employee {
    ...
}
```

- By extending `Employee`, each `Secretary` object now:
  - receives a `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` method automatically
  - can be treated as an `Employee` by client code (seen later)

Overriding methods

- **override**: To write a new version of a method in a subclass that replaces the superclass's version.
  - No special syntax required to override a superclass method.
    Just write a new version of it in the subclass.

```
public class Lawyer extends Employee {
    // overrides getVacationForm method in Employee class
    public String getVacationForm() {
        return "pink";
    }
    ...
}
```

-@Override doesn't need to be there, Java does it auto. Hard to read code w/o. Also, can't override final methods.

Calling overridden methods

- Subclasses can call overridden methods with `super`

```
super.method(parameters)
```

– Example:

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.0;
    }
    ...
}
```

– Exercise: Modify `Lawyer` and `Marketer` to use `super`.

-Calls the superclass/parent's method that was overridden with subclass/child

Problem with constructors

- Now that we've added the constructor to the `Employee` class, our subclasses do not compile. The error:

```
Lawyer.java:2: cannot find symbol
symbol  : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
       ^
```

  - The short explanation: Once we write a constructor (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well.

-Subclass needs

-When a subclass object is created, Java wants to call the constructor of the superclass.

-So to create a subclass Lawyer object of superclass Employee:

```
public Lawyer (int years) {
    super(years);    //This will call the super's Overloading Constructor and pass in years to assign as a field. This MUST be 1ST statement.
}
```

-Without this, Java will just use Default Constructor

-Protected Level of Visibility

-protected int years;

-years is accessible by class and child.

-DANGEROUS!

-Polymorphism

```
public class EmployeeTests{
    public static void main(String [] args){
        Employee emp = new Employee(2);
        Lawyer lawyer = new Lawyer(3);

        Employee anEmployee;
        anEmployee = lawyer;//perfectly legal and normal, this is polymorphism

    }


}
```

```
public class Vehicle {...}
public class Car extends Vehicle {...}
public class SUV extends Car {...}
```

Which of the following are legal statements?

a. ☑ Car c = new SUV();
b. ☐ SUV s = new Car();
c. ☑ Vehicle v = new SUV();
d. ☑ SUV s = new SUV();
e. ☑ Vehicle v = new Car();
f. ☐ Car c = new Vehicle();

-Overriding Fields

Inheritance and fields

- Try to give lawyers $5000 for each year at the company:

```
public class Lawyer extends Employee {
    ...
    public double getSalary() {
        return super.getSalary() + 5000 * years;
    }
    ...
}
```

- Does not work; the error is the following:

```
Lawyer.java:7: years has private access in Employee
        return super.getSalary() + 5000 * years;
                                          ^
```

- Private fields cannot be directly accessed from subclasses.
  - One reason: So that subclassing can't break encapsulation.
  - How can we get around this limitation?

-Weakness of inheritance is that calling subclass methods from a subclass reference of a superclass variable doesn't work.
-TypeCast can fix.

```
public class EmployeeTests{
    public static void main(String [] args){
        Employee emp = new Employee(2);
        Lawyer lawyer = new Lawyer(3);

        Employee anEmployee;
        anEmployee = lawyer;//perfectly legal and normal, this is polymorphism
        System.out.println(anEmployee.getVacationDays());
        if (anEmployee instanceof Lawyer)
            ((Lawyer)anEmployee).sue();

    }


}
```

-Better InstanceOf

```
(theOtherEmployee.getClass().getSimpleName() == "Employee")
```

```
public boolean equals(Object theObject) {
   //if (theObject instanceof Clock) { //Alt instanceof
   if (theObject.getClass().getSimpleName().equalsIgnoreCase("Clock")) {
```

-Object class's equals() method or == Sucks at comparing objects, since it compares if the object is referring to the same point in memory.
-Interfaces can not have instance fields.
-WWe can pass as param a Secretary (subclass to Employee) into methods that takes type Employee.
-Interface: List

ArrayList methods (10.1)

`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

ArrayList methods 2

`addAll(`**list**`)` `addAll(`**index**`, `**list**`)`	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
`contains(`**value**`)`	returns true if given value is found somewhere in this list
`containsAll(`**list**`)`	returns true if this list contains every element from given list
`equals(`**list**`)`	returns true if given other list contains the same elements
`iterator()` `listIterator()`	returns an object used to examine the contents of the list (seen later)
`lastIndexOf(`**value**`)`	returns last index value is found in list (-1 if not found)
`remove(`**value**`)`	finds and removes the given value from this list
`removeAll(`**list**`)`	removes any elements found in the given list from this list
`retainAll(`**list**`)`	removes any elements *not* found in given list from this list
`subList(`**from**`, `**to**`)`	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
`toArray()`	returns the elements in this list as an array

```
ArrayList<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");
```

ArrayList vs. array

- construction

```
String[] names = new String[5];
ArrayList<String> list = new ArrayList<String>();
```

- storing a value

```
names[0] = "Jessica";
list.add("Jessica");
```

- retrieving a value

```
String s = names[0];
String s = list.get(0);
```

ArrayList vs. array 2

- doing something to each value that starts with "B"

```
for (int i = 0; i < names.length; i++) {
    if (names[i].startsWith("B")) { ... }
}
```

```
for (int i = 0; i < list.size(); i++) {
    if (list.get(i).startsWith("B")) { ... }
}
```

- seeing whether the value "Benson" is found

```
for (int i = 0; i < names.length; i++) {
    if (names[i].equals("Benson")) { ... }
}
```

```
if (list.contains("Benson")) { ... }
```

-i-- because remove() shift everything else left.

```
// Removes all plural words from the given list.
public static void removePlural(ArrayList<String> theList)
{
    for (int i = 0; i < theList.size(); i++) {
        String str = theList.get(i);
        if (str.endsWith("s")) {
            theList.remove(i);
            i--;
        }
    }
}
```

-Try these statements

-Catch Exception if error

-Finally, do these statements

-public void bruh() throws Exception { //The method that calls this method must CATCH the Exception or else explodes.

if (urmumfat) {

throw new Exception(“fat like eart”);

}

}

-Collection Interface

-We have Array, but they kinda suck compared to the others.

-There are list, bag, stack, queue, set, map, graph

-Scope/Visibility

```
visibility in Java (most to least)
-public
-protected (child classes and classes in current package)
-package (sometimes called package private sometimes package public)
-private
```

-ArrayList limitations

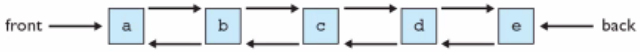
-Accessing is super fast because it's stored one after another in RAM, zip boop & quickly grab with index.

-Adding to an ArrayList in the middle of it, shifting everything out of the way is slow.

-LinkedList

Linked list

- **linked list**: a list implemented using a linked sequence of values
  - each value is stored in a small object called a **node**, which also contains references to its neighbor nodes
  - the list keeps a reference to the first and/or last node
  - in Java, represented by the class `LinkedList`



-There is Singular and Doubly Linked

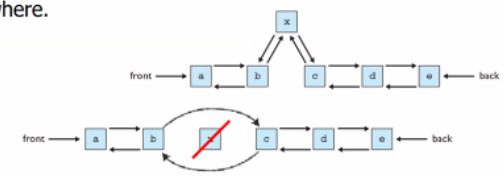
-Singular is like a one way street.

-Doubly Linked (In this slide), where its a two way street type beat. In this pic, e has a link back to a.

-LinkedList is really fast to remove and add in middle cases as the the links just refer to new one instead of shifting everything. The slow is finding the indexes.

Linked list performance

- To add, remove, get a value at a given index:
  - The list must advance through the list to the node just before the one with the proper index.
  - Example: To add a new value to the list, the list creates a new node, walks along its existing node links to the proper index, and attaches it to the nodes that should precede and follow it.
  - This is very fast when adding to the front or back of the list (because the list contains references to these places), but slow elsewhere.



20

-Going from one node to another be links is slower than ArrayList, as the nodes are not ordered in RAM continuously.

-Iterating LinkedList

A particularly slow idiom

```
List<String> list = new LinkedList<String>();
// ... (put a lot of data into the list)

// print every element of linked list
for (int i = 0; i < list.size(); i++) {
    Object element = list.get(i);
    System.out.println(i + ": " + element);
}
```

- This code executes a slow operation (`get`) every pass through a loop that runs many times
  - this code will take prohibitively long to run for large data sizes

-get method is slow because it starts at 0 and walks 1 by 1 to finish.

`Iterator` interface in `java.util`

every collection has an `iterator()` method that returns an iterator over its elements

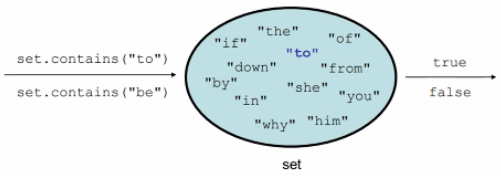
```
Set<String> set = new HashSet<String>();
...
Iterator<String> itr = set.iterator();

...
List<Integer> list = new LinkedList<Integer>();
...
Iterator<Integer> itr = list.iterator();
```

-With an dedicated iterator, the iterator works like a Scanner, moving from one to the next but staying at the finish.

-Set (Essentially a Collection without index/order)

Sets (11.2)

- **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - add, remove, search (contains)
  - We don't think of a set as having indexes; we just add things to the set in general and don't worry about order



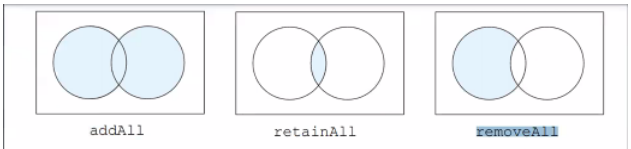
-HashSet & TreeSet (& LinkedHashSet)

Set implementation

- in Java, sets are represented by `Set` interface in `java.util`
- `Set` is implemented by `HashSet` and `TreeSet` classes
  - `HashSet`: implemented using a "hash table" array; very fast: **O(1)** for all operations; elements are stored in unpredictable order
  - `TreeSet`: implemented using a "binary search tree"; pretty fast: **O(log N)** for all operations; elements are stored in sorted order
  - `LinkedHashSet`: **O(1)** but stores in order of insertion

-O(n) is Constant Time Complexity, the higher the number inside the () is how long it takes.

-Some methods

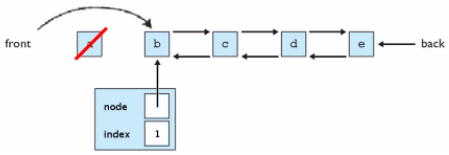


`addAll(`**collection**`)`	adds all elements from the given collection to this set
`containsAll(`**coll**`)`	returns `true` if this set contains every element from given set
`equals(`**set**`)`	returns `true` if given other set contains the same elements
`iterator()`	returns an object used to examine set's contents *(seen later)*
`removeAll(`**coll**`)`	removes all elements in the given collection from this set
`retainAll(`**coll**`)`	removes elements *not* found in given collection from this set
`toArray()`	returns an array of the elements in this set

-Remove

Iterator's remove

- The `remove` removes the last value that was returned by a call to `next`
  - in other words, it deletes the element just *before* the iterator's current node



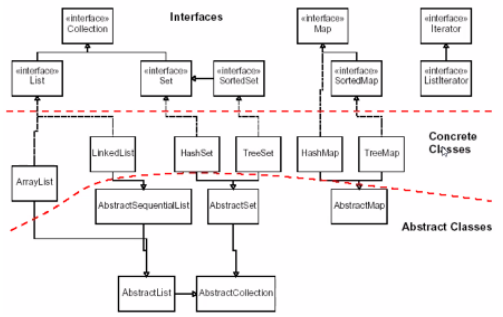
-It exist, but volatile, especially with multi thread multiple iterators.

-List ADT

-Java has a List abstract data type.

-Collection framework

Java collections framework



-Sets More

-TreeSet, HashSet, LinkedHashSet

- `HashSet` : elements are stored in an unpredictable order

```
Set<String> names = new HashSet<String>();
names.add("Jake");
names.add("Robert");
names.add("Marisa");
names.add("Kasey");
System.out.println(names);
// [Kasey, Robert, Jake, Marisa]
```

- `TreeSet` : elements are stored in their "natural" sorted order

```
Set<String> names = new TreeSet<String>();
...
// [Jake, Kasey, Marisa, Robert]
```

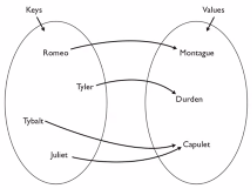
- `LinkedHashSet` : elements stored in order of insertion

```
Set<String> names = new LinkedHashSet<String>();
...
// [Jake, Robert, Marisa, Kasey]
```

-Map ADT

The Map ADT

- **map**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.
  - a.k.a. "dictionary", "associative array", "hash"
- basic map operations:
  - **put**(*key*, *value*): Adds a mapping from a key to a value.
  - **get**(*key*): Retrieves the value mapped to the key.
  - **remove**(*key*): Removes the given key and its mapped value.



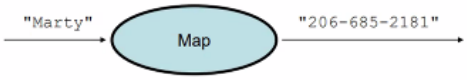
`myMap.get("Juliet")` returns `"Capulet"`

-Map (element corresponding) Vs Set (existence)

Maps vs. sets

- A set is like a map from elements to `boolean` values.
  - *Set: Is "Marty" found in the set? (true/false)*

  - *Map: What is "Marty" 's phone number?*



- in Java, maps are represented by `Map` interface in `java.util`
- `Map` is implemented by the `HashMap` and `TreeMap` classes
  - `HashMap`: implemented using an array called a "hash table"; extremely fast: **O(1)** ; keys are stored in unpredictable order
  - `TreeMap`: implemented as a linked "binary tree" structure; very fast: **O(log N)** ; keys are stored in sorted order
  - A map requires 2 type parameters: one for keys, one for values.

```
// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();
```

-Methods

`Map` methods

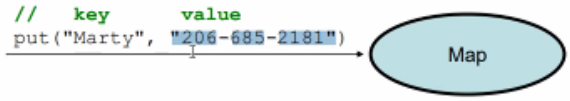
`put(`**key**`, `**value**`)`	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
`get(`**key**`)`	returns the value mapped to the given key (`null` if not found)
`containsKey(`**key**`)`	returns `true` if the map contains a mapping for the given key
`remove(`**key**`)`	removes any existing mapping for the given key
`clear()`	removes all key/value pairs from the map
`size()`	returns the number of key/value pairs in the map
`isEmpty()`	returns `true` if the map's size is 0
`toString()`	returns a string such as `"{a=90, d=60, c=70}"`

`keySet()`	returns a set of all keys in the map
`values()`	returns a collection of all values in the map
`putAll(`**map**`)`	adds all key/value pairs from the given map to this map
`equals(`**map**`)`	returns `true` if given map has the same mappings as this one

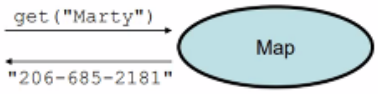
-Using Maps

Using maps

- A map allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).



  - Later, we can supply only the key and get back the related value:
    Allows us to ask: *What is Marty's phone number?*



keySet and values

- `keySet` method returns a `Set` of all keys in the map
  - can loop over the keys in a foreach loop
  - can get each key's associated value by calling `get` on the map

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Marty", 19);
ages.put("Geneva", 2);  // ages.keySet() returns Set<String>
ages.put("Vicki", 57);
for (String name : ages.keySet()) {           // Geneva -> 2
    int age = ages.get(name);                 // Marty -> 19
    System.out.println(name + " -> " + age);  // Vicki -> 57
}
```

- `values` method returns a collection of all values in the map
  - can loop over the values in a foreach loop
  - no easy way to get from a value to its associated key(s)

-Reversal

Proper map reversal

- Really each GPA maps to a *collection* of people.

```
Map<Double, Set<String>> taGpa =
        new HashMap<Double, Set<String>>();
taGpa.put(3.6, new TreeSet<String>());
taGpa.get(3.6).add("Jared");
taGpa.put(4.0, new TreeSet<String>());
taGpa.get(4.0).add("Alyssa");
taGpa.put(2.9, new TreeSet<String>());
taGpa.get(2.9).add("Steve");
taGpa.get(3.6).add("Stef");
taGpa.get(2.9).add("Rob");
...
System.out.println("Who got a 3.6? " +
                   taGpa.get(3.6));    // [Jared, Stef]
```

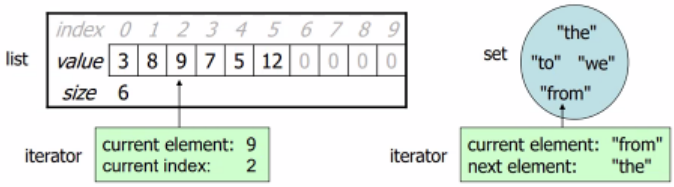
OR

```
Double[] gpas = {3.6, 4.0, 2.9};
String[][] nameArray = {{"Jared", "Stef"},
                        {"Alyssa"},
                        {"Steve", "Rob"}};
Map<Double, Set<String>> taGpa2 = new HashMap<Double,
                                      Set<String>>();
for (int i = 0; i < nameArray.length; i++){
  Set<String> names = new TreeSet<String>();
  for (int k = 0; k < nameArray[i].length; k++) {
     names.add(nameArray[i][k]);
  }
  taGpa2.put(gpas[i],names);
}
System.out.println("Who got a 3.6? " + taGpa2.get(3.6));
```

. . .

-Map Iterator

- **iterator**: An object that allows a client to traverse the elements of any collection.
  - Remembers a position, and lets you:
    - get the element at that position
    - advance to the next position
    - remove the element at that position



-Ex:

```
Set<Integer> scores = new TreeSet<Integer>();
scores.add(94);
scores.add(38);   // Kim
scores.add(87);
scores.add(43);   // Marty
scores.add(72);
...

Iterator<Integer> itr = scores.iterator();
while (itr.hasNext()) {
    int score = itr.next();

    System.out.println("The score is " + score);

    // eliminate any failing grades
    if (score < 60) {
        itr.remove();
    }
}
System.out.println(scores);  // [72, 87, 94]
```

```
Map<String, Integer> scores = new TreeMap<String, Integer>();
scores.put("Kim", 38);
scores.put("Lisa", 94);
scores.put("Roy", 87);
scores.put("Marty", 43);
scores.put("Marisa", 72);
...

Iterator<String> itr = scores.keySet().iterator();
while (itr.hasNext()) {
    String name = itr.next();
    int score = scores.get(name);
    System.out.println(name + " got " + score);

    // eliminate any failing students
    if (score < 60) {
        itr.remove();      // removes name and score
    }
}
System.out.println(scores);  // {Lisa=94, Marisa=72, Roy=87}
```

-Map and Tallying

Maps and tallying

- a map can be thought of as generalization of a tallying array
  - the "index" (key) doesn't have to be an `int`

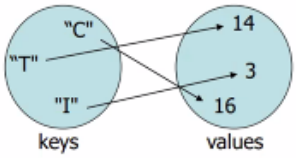
- recall previous tallying examples from CSE 142
  - count digits: `22092310907`

index	0	1	2	3	4	5	6	7	8	9
value	3	1	3	0	0	0	0	1	0	2

```
// (C)Clinton, (T)Trump, (I)ndependent
```

  - count votes: "TCCCCTTCCCTCCTCCTCTTCITCTTITCTTITT"

key	"C"	"T"	"I"
value	16	14	3




```
import java.util.Iterator;
import java.util.Map;
import java.util.TreeMap;
public class Election {
  public static void main(String[] theArgs) {
    String votes = "TCCCCTTCCCTCCTCCTCTTCITCTTITCTTITT";
    Map<Character, Integer> results = getVotes(votes);
    displayVotes(results);
  }
  public static Map<Character, Integer> getVotes(String theVoteString) {
    Map<Character, Integer> tally = new TreeMap<Character, Integer>();
    for (int i = 0; i < theVoteString.length(); i++) {
      Character candidate = theVoteString.charAt(i);
      if (tally.containsKey(candidate)) {
        tally.put(candidate, tally.get(candidate) + 1);
      } else {
        tally.put(candidate, 1);
      }
    }
    return tally;
  }
  public static void displayVotes(Map<Character,
                                      Integer> theMap) {
    Iterator<Character> itr = theMap.keySet().iterator();
    while (itr.hasNext()) {
      Character key = itr.next();
      int count = theMap.get(key);
      System.out.println(key + " has " + count + " votes!");
    }
  }
}
```

-Hash = Fast. That what prof said to do for coding interview.
-Sequential/Linear Search (Using this for list that are already sorted is bad.)

- **sequential search**: Locates a target value in an array/list by examining each element from start to finish.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

i

-Binary Search (Dumb number guessing game but useful on sorted lists)

- **binary search**: Locates a target value in a *sorted* array/list by successively eliminating half of the array from consideration.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

min mid max

```
// Returns the index of an occurrence of target in a,
// or a negative number if the target is not found.
// Precondition: elements of a are in sorted order
public static int binarySearch(int[] a, int target) {
    int min = 0;
    int max = a.length - 1;

    while (min <= max) {
        int mid = (min + max) / 2;
        if (a[mid] < target) {
            min = mid + 1;
        } else if (a[mid] > target) {
            max = mid - 1;
        } else {
            return mid;    // target found
        }
    }

    return -(min + 1);     // target not found
}
```

-Comparable Interface (This code can get sort any array with Comparable Interface)

```
public static int binarySearch(Comparable[] a, Comparable target) {
    int min = 0;
    int max = a.length - 1;
    while (min <= max) {
        int mid = (min + max) / 2;
        if (a[mid].compareTo(target) < 0) {
            min = mid + 1;
        } else if (a[mid].compareTo(target) > 0) {
            max = mid - 1;
        } else {
            return mid; // target found
        }
    }

    return -(min + 1); // target not found
    }
```

-The parameter will give a warning for this code.

-Arrays Class methods

- Class `Arrays` in `java.util` has many useful array methods:

Method name	Description
`binarySearch(`**array**`,` **value**`)`	returns the index of the given value in a *sorted* array (or < 0 if not found)
`binarySearch(`**array**`,` **minIndex**`,` **maxIndex**`,` **value**`)`	returns index of given value in a *sorted* array between indexes *min* / *max* - 1 (< 0 if not found)
`copyOf(`**array**`,` **length**`)`	returns a new resized copy of an array
`equals(`**array1**`,` **array2**`)`	returns `true` if the two arrays contain same elements in the same order
`fill(`**array**`,` **value**`)`	sets every element to the given value
`sort(`**array**`)`	arranges the elements into sorted order
`toString(`**array**`)`	returns a string representing the array, such as `"[10, 30, -25, 17]"`

- Syntax: `Arrays.`**methodName**`(`**parameters**`)`

-Comparator Interface

-It allows unique ways to compare objects. Useful for things like students and ordering by name and whatever.

-Or, want to sort a string array to alphabetical order but ignore case, which normal sort will do.

Custom Ordering with Comparators

The `Comparator` interface in `java.util` describes a method for comparing two objects of a given type:

```
public interface Comparator<T> {
public int compare(T o1, T o2);
}
```

< 0 if o1 comes before o2,

0 if o1 and o2 are equivalent,

> 0 if o1 comes after o2.

Note that `Comparator` is a generic interface.

`T` will be replaced by the type of object you are actually comparing.

The `compare` method's job is to decide the relative ordering of the two given objects and return an appropriate integer:

-TO DO THIS, create a custom class implementing the Comparator interface.

-Ex:

The following `Comparator` compares `Strings`, ignoring case:

```
public class CaseInsensitiveComparator implements
                                        Comparator<String>{
  public int compare(String s1, String s2) {
    return s1.toLowerCase().compareTo(s2.toLowerCase());
  }
}
```

OR

```
public class CaseInsensitiveComparator implements
                                        Comparator<String>{
  public int compare(String s1, String s2) {
    return s1.compareToIgnoreCase(s2);
  }
}
```

```
public class DescendingComparator implements
                                        Comparator<String>{
  public int compare(String s1, String s2) {
    return s2.compareToIgnoreCase(s1);
  }
}
```

descends

-Using it by instantiating it

Sort methods are often written so that they can be called with a `Comparator` as a second parameter.

The sorting algorithm will use that comparator to order the elements of the array or list.

```
String[] strings = {"Foxtrot", "alpha", "echo", "golf",
                    "bravo", "hotel", "Charlie", "DELTA"};
Arrays.sort(strings);
System.out.println(Arrays.toString(strings));
Arrays.sort(strings, new CaseInsensitiveComparator());
System.out.println(Arrays.toString(strings));
```

Output:

```
[Charlie, DELTA, Foxtrot, alpha, bravo, echo, golf, hotel]
[alpha, bravo, Charlie, DELTA, echo, Foxtrot, golf, hotel]
```

-Runtime efficiency

-How good is speed and memory efficient?

-Executing a statement / comparing is one “n”.

-Examples:

```
statement1;
statement2;
statement3;                                   } 3

for (int i = 1; i <= N; i++) {
    statement4;                               } N
}

for (int i = 1; i <= N; i++) {
    statement5;
    statement6;                               } 3N
    statement7;
}
```

4N + 3

-To find how much work in general, find the dominant term in the final express (which will grow fastest). So 4N would be the leading term.

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {            } N^2
        statement1;
    }
}

for (int i = 1; i <= N; i++) {
    statement2;
    statement3;                               } 4N
    statement4;
    statement5;
}
```

$N^2 + 4N$

-Growth Rate

- We measure runtime in proportion to the input data size, N.
  - **growth rate**: Change in runtime as N changes.

- Say an algorithm runs $\mathbf{0.4N^3 + 25N^2 + 8N + 17}$ statements.
  - Consider the runtime when N is *extremely large* .
  - We ignore constants like 25 because they are tiny next to N.
  - The highest-order term (N^3) dominates the overall runtime.
  - We say that this algorithm runs "on the order of" N^3.
  - or $\mathbf{O(N^3)}$ for short ("**Big-Oh** of N cubed")

- **complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Class	Big-Oh	If you double N, ...	Example
constant	O(1)	unchanged	10ms
logarithmic	$O(\log_2 N)$	increases slightly	175ms
linear	O(N)	doubles	3.2 sec
log-linear	$O(N \log_2 N)$	slightly more than doubles	6 sec
quadratic	$O(N^2)$	quadruples	1 min 42 sec
cubic	$O(N^3)$	multiplies by 8	55 min
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	$5 * 10^{61}$ years

-Collection Methods Efficiency

- Efficiency of various operations on different collections:

Method	ArrayList	SortedIntList	Stack	Queue
`add` (or `push`)	O(1)	O(N)	O(1)	O(1)
`add(`**`index, value`**`)`	O(N)		-	-
`indexOf`	O(N)	O(?)	-	-
`get`	O(1)	O(1)	-	-
`remove`	O(N)	O(N)	O(1)	O(1)
`set`	O(1)	O(1)	-	-
`size`	O(1)	O(1)	O(1)	O(1)

-Collection Class's Methods

`Collections` class

Method name	Description
`binarySearch(`**`list, value`**`)`	returns the index of the given value in a sorted list (< 0 if not found)
`copy(`**`listTo, listFrom`**`)`	copies **listFrom**'s elements to **listTo**
`emptyList(), emptyMap(), emptySet()`	returns a read-only collection of the given type that has no elements
`fill(`**`list, value`**`)`	sets every element in the list to have the given value
`max(`**`collection`**`), min(`**`collection`**`)`	returns largest/smallest element
`replaceAll(`**`list, old, new`**`)`	replaces an element value with another
`reverse(`**`list`**`)`	reverses the order of a list's elements
`shuffle(`**`list`**`)`	arranges elements into a random order
`sort(`**`list`**`)`	arranges elements into ascending order

-Sorting Algorithm be like

- **bogo sort**: shuffle and pray
- **bubble sort**: swap adjacent pairs that are out of order
- **selection sort**: look for the smallest element, move to front
- **insertion sort**: build an increasingly large sorted front portion
- **merge sort**: recursively divide the array in half and sort it
- **heap sort**: place the values into a sorted tree structure
- **quick sort**: recursively partition array based on a middle value

other specialized sorting algorithms:

- **bucket sort**: cluster elements into smaller groups, sort them
- **radix sort**: sort integers by last digit, then 2nd to last, then ...
- ...