CS 5004: OBJECT ORIENTED DESIGN AND ANALYSIS

SPRING 2021

LECTURE 9

DIVYA CHAUDHARY

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ T 617.373.2462 ▪ khoury.northeastern.edu

AGENDA

- Course logistics
  - Review – Java CLI, Java I/0, regular expressions
- Java Collections Framework
  - Interfaces Iterator, Comparable and Comparator
  - Maps
  - Introduction to hashing
  - Nested classes
  - Binary and binary search trees

REVIEW

CS 5004, SPRING 2022 – LECTURE 9

REVIEW: INTERACTIVE PROGRAMS

- Interactive programs – programs where a user interacts with a program by providing an input into the console, that a program then reads and uses for execution

- Interactive programs can (sometimes) be challenging
  - Computers and users think in very different way
  - Users misbehave
  - Users are malicious

REVIEW: ACCESSING COMMAND LINE ARGUMENTS

- Question: How to pass information into a program before we run it?
- Answer: Use command–line arguments to `main()` method
- A command-line argument (input) is the information that directly follows the program name on the command line when we call the program to execute it
- Command-line arguments are stored as `Strings` in the String array `args` passed to `main()`
- All command-line arguments are passed as `Strings` → you need to convert them into a valid format

REVIEW: I/O STREAMS

- Concept of an I/O stream – a communication channel ("pipe") between a source and a destination that allows us to create a flow of data

- I/O Stream has:
  - An input source (a file, another program, device)
  - An output destination (file, another program, device)
  - The kind of data streamed (bytes, ints, objects)

REVIEW: BUFFERED READER/WRITER

- `BufferedReader` – reads text from a character-input stream, buffering characters to provide efficient reading of characters, arrays, and lines
- Buffer size may be specified, or the default size may be used
- (the default is typically large enough for most purposes)

- Why use `BufferReader`?
  - In general, each read request causes a corresponding read request to be made of the underlying character or byte stream
  - These operations may be costly → each invocation of `read()` or `readLine()` in a `FileReader` or `InputFileReader` causes bytes to be read from the file, converted into characters, and then returned
  - `BufferReader` increases efficiency by buffering the input from the specified file

REVIEW: REGULAR EXPRESSIONS

- Regular expression (RE) – a sequence of characters used to describe some set of strings (pattern)
- Example: RE “class” used inside a document to search for the word “class”

- Defined over some alphabet Σ
  - For programming languages, alphabet is usually ASCII or Unicode
- In other words, regular expression re describes the set of strings, called a language L(re), over the elements of some alphabet Σ

REVIEW: REGULAR EXPRESSIONS IN JAVA

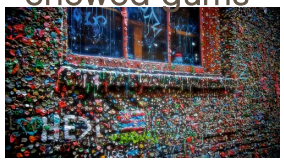
- `Java.util.regex.Pattern`
  - A regular expression, specified as a string, first compiled into an instance of class `Pattern`
  - The resulting pattern can then be used to create a `Matcher` object that can match arbitrary character sequences against the regular expression
- `Java.util.regex.Matcher`
  - Can be used to perform three different kinds of match operations:
    - `matches()` - method attempts to match the entire input sequence against the pattern
    - `lookingAt()` - method attempts to match the input sequence, starting at the beginning, against the pattern
    - `find()` - method scans the input sequence looking for the next subsequence that matches the pattern

JAVA COLLECTIONS FRAMEWORK

CS 5004, SPRING 2022 – LECTURE 9

ONCE AGAIN: DATA COLLECTIONS

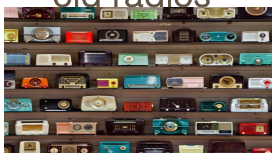
Collection of chewed gums



Collection of pens

Collection of cassette tapes

Collection of old radios



What is a data collection?

Shoes collection

Star wars

Cars collection

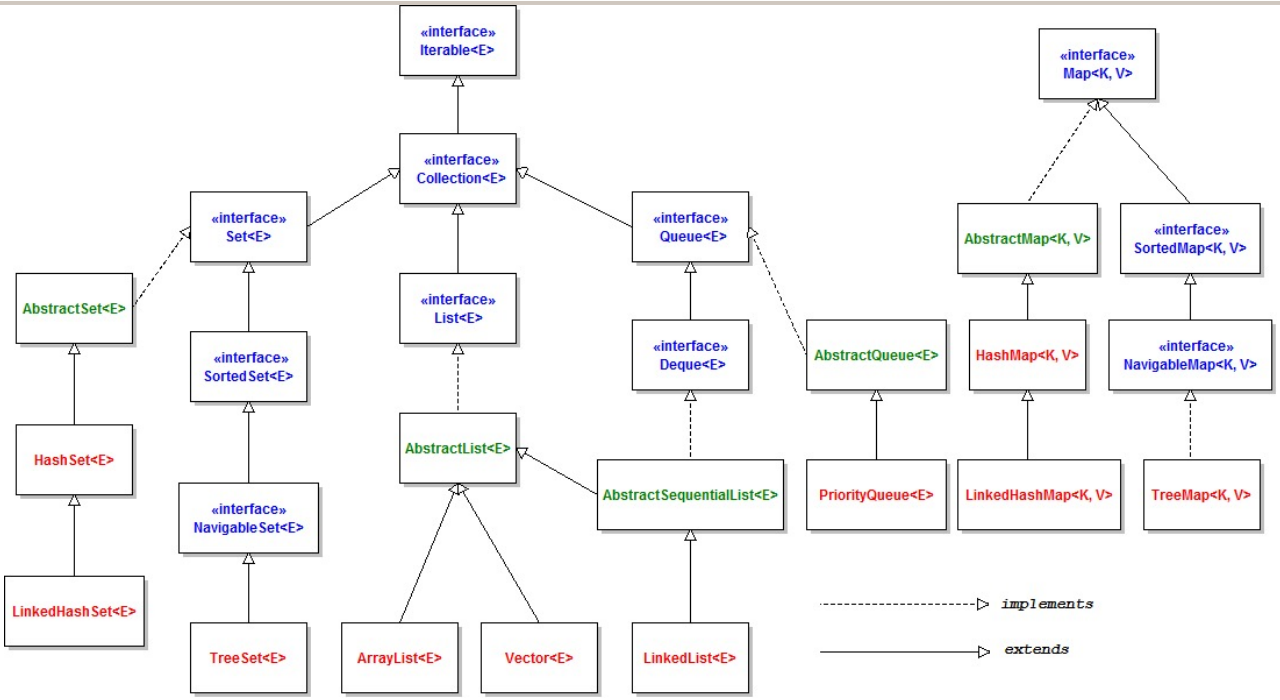
[Pictures credit: http://www.smosh.com/smosh-pit/articles/19-epic-collections-strange-things]

ONCE AGAIN: DATA COLLECTIONS

- Data collection - an object used to store data (think *data structures*)
  - Stored objects called elements
  - Some typically operations:
    - `add()`
    - `remove()`
    - `clear()`
    - `size()`
    - `contains()`

- **Data collections we already know:** Lists (ArrayList, LinkedList), Stack, Queue, Deques, Sets

ONCE AGAIN: JAVA COLLECTIONS FRAMEWORK



[Pictures credit: http://www.codejava.net]

ONCE AGAIN: JAVA COLLECTIONS FRAMEWORK

- Under the Java Collections Framework, we have:
  - Interfaces that define the behavior of various data collections
  - Abstract classes that implement the interface(s) of the collection framework, that can then be extended to created a specialized data collection
  - Concrete classes that provide a general-purpose implementation of the interface(s)

ONCE AGAIN: JAVA COLLECTIONS FRAMEWORK

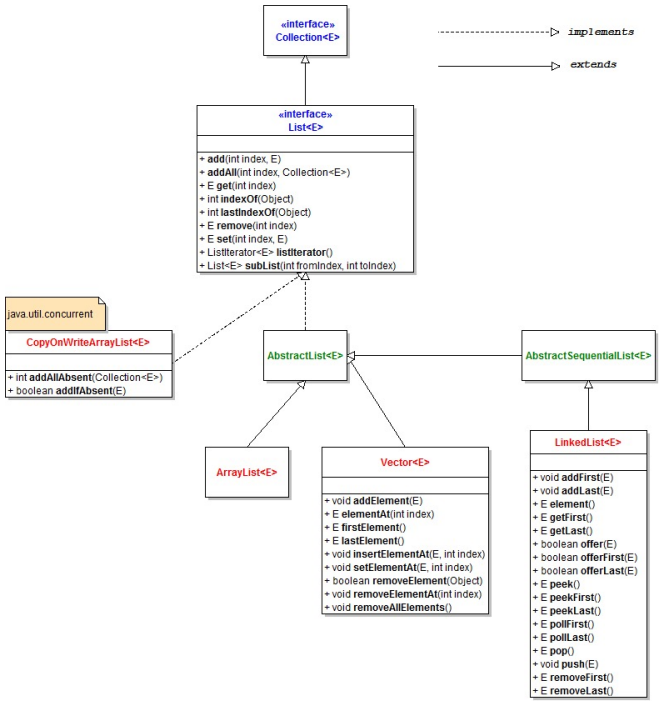
- Goals of the Java Collections Framework:
  - Reduce programming effort by providing the most common data structures
  - Provide a set of types that are easy to use and extend
  - Provide flexibility through defining a standard set of interfaces for collections to implement
  - Improve program quality through the use and reuse of tested implementations of common data structures

ONCE AGAIN: JAVA COLLECTIONS FRAMEWORK

- Part of the `java.util` package

- Interface `Collection<E>`:
  - Root interface in the collection hierarchy
  - Extended by four interfaces:
    - `List<E>`
    - `Set<E>`
    - `Queue<E>`
    - `Map<K,V>`
  - Extends interface `Iterable<T>`

JAVA LIST API



[Pictures credit: http://www.codejava.net]

- `List<E>` - the base interface
- Abstract subclasses:
  - `AbstractList<E>`
  - `AbstractSequentialList<E>`
- Concrete classes:
  - `ArrayList<E>`
  - `LinkedList<E>`
  - `Vector<E>` (legacy collection)
  - `CopyOnWriteArrayList<E>` (class under java.util.concurrent package)
- Main methods:
  - `E get(int index);`
  - `E set(int index, E newValue);`
  - `Void add(int index, E x);`
  - `Void remove(int index);`
  - `ListIterator<E> listIterator();`

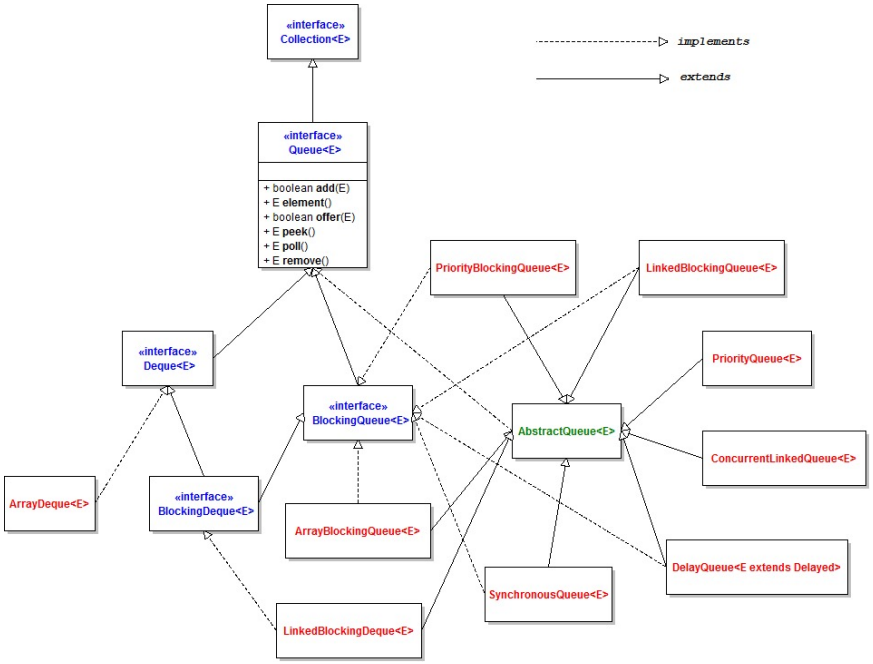
JAVA CLASS STACK

`Stack <E> ()`	Object constructor – constructs a new stack with elements of type E
`push(value)`	Places given value on top of the stack
`pop()`	Removes top value from the stack, and returns it. Throws `EmptyStackException` if the stack is empty.
`peek()`	Returns top value from the stack without removing it. Throws EmptyStackException if the stack is empty.
`size()`	Returns the number of elements on the stack.
`isEmpty()`	Returns true if the stack is empty.

- Example:

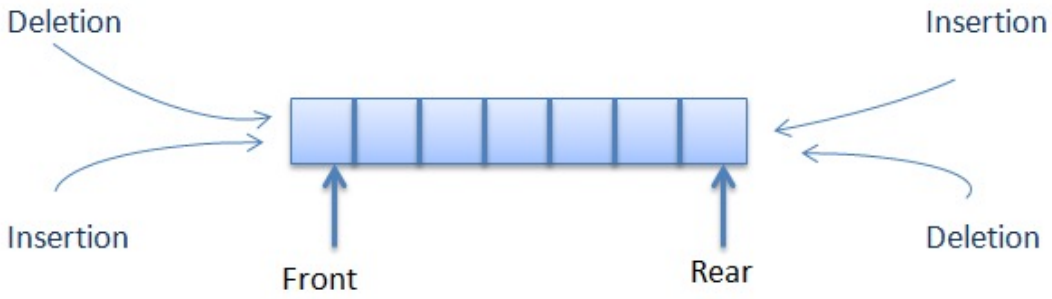
```
Stack<String> s = new Stack<String>();
s.push("Hello");
s.push("PDP");
S.push("Fall 2017"); //bottom ["Hello", "PDP", "Fall 2017"] top
System.out.println(s.pop()); //Fall 2017
```

CLASS DIAGRAM OF QUEUE API



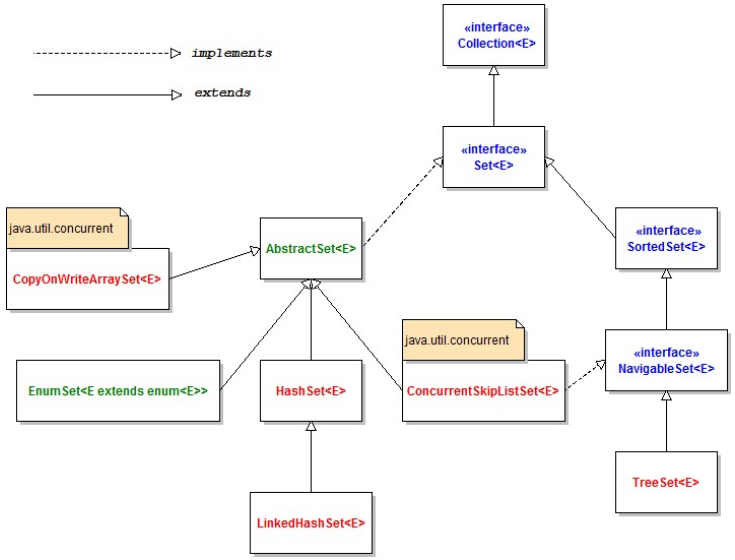
[Pictures credit: http://www.codejava.net/java-core/collections/class-diagram-of-queue-api]

DEQUE



[Pictures credit: http://www.java2novice.com/data-structures-in-java/queue/double-ended-queue/]

CLASS DIAGRAM OF SET API



[Pictures credit: http://www.codejava.net/images/articles/javacore/collections/Set%20API%20class%20diagram.png]

ITERATING OVER A DATA COLLECTION

CS 5004, SPRING 2022 – LECTURE 9

INTERFACE ITERABLE<T>

- Super-interface for interface `Collection<T>`
- Implementing interface `Iterable<T>` allows an object to be traversed using the for each loop
- Every object that implements `Iterable<T>` must provide a method `Iterator iterator()`

Modifier and Type	Method and Description
`default void`	**`forEach(Consumer<? super T> action)`** Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception.
`Iterator<T>`	**`iterator()`** Returns an iterator over elements of type T.
`default Spliterator<T>`	**`spliterator()`** Creates a **`Spliterator`** over the elements described by this `Iterable`.

INTERFACE ITERABLE<T> AND FOREACH LOOP

- Super-interface for interface `Collection<T>`
  - Implementing interface `Iterable<T>` allows an object to be traversed using the foreach loop!
  - `Foreach loop -` performs some specified action for each element of the interface `Iterable`, until:
    - All elements have been processed or
    - The action throws an exception
  - Actions are performed in the order of iteration (unless otherwise specified by the implementing class)
  - Exceptions thrown by the action are relayed to the caller

INTERFACE ITERABLE<T> AND ITERATOR<E>

- Super-interface for interface `Collection<T>`
- Every object that implements `Iterable<T>` must provide a method `Iterator iterator()!`

- Interface `Iterator` – an iterator over a collection

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove():
}
```

- Iterator `remove()` method – removes the last item returned by method `next()`

DIRECT USE OF ITERATOR<E>

- Foreach loop relies on an iterator
- But we can use an iterator directly, by calling method implemented by that iterator

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove():
}
```

- Careful when an iterator is used directly (not via a for each loop)→ if you make any structural changes to a collection being iterated (add, remove, clear), the iterator is no longer valid (`ConcurrentModificationException` thrown)

ORDERING OF OBJECTS

CS 5004, SPRING 2022 – LECTURE 9

COMPARING OBJECTS

- Problem: How do we compare `String` in some list of `Strings`?
  - Operators like < and > do not work with `String` objects
  - But we do think of `Strings` as having an alphabetical ordering
- Natural ordering: Rules governing the relative placement of all values of a given type
- Comparison function: Code that, when given two values A and B of a given type, decides their relative ordering:

  - A < B, A == B, A > B

METHOD COMPARETO()

- A standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method
  - Example: in the `String` class, there is a method:

    ```
    public int compareTo(String other)
    ```

- A call of A.`compareTo(B)` will return:

  a value < 0 if A comes "before" B in the ordering

  a value > 0 if A comes "after" B in the ordering

  0 if A and B are considered "equal" in the ordering

USING METHOD COMPARETO

- `compareTo` can be used as a test in an if statement

```
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) {   // true
    ...
}
```

Primitives	Objects
`if (a < b) { ...`	`if (a.compareTo(b) < 0) { ...`
`if (a <= b) { ...`	`if (a.compareTo(b) <= 0) { ...`
`if (a == b) { ...`	`if (a.compareTo(b) == 0) { ...`
`if (a != b) { ...`	`if (a.compareTo(b) != 0) { ...`
`if (a >= b) { ...`	`if (a.compareTo(b) >= 0) { ...`
`if (a > b) { ...`	`if (a.compareTo(b) > 0) { ...`

METHOD COMPARE TO AND JAVA COLLECTIONS

- We can use an array or list of `Strings` with Java's included binary search method because it calls `compareTo` internally

```
String[] a = {"al", "bob", "cari", "dan", "mike"};
int index = Arrays.binarySearch(a, "dan");  // 3
```

- Java's `TreeSet/Map` use compareTo internally for ordering

```
Set<String> set = new TreeSet<String>();
for (String s : a) {
    set.add(s);
}
System.out.println(s);
// [al, bob, cari, dan, mike]
```

ORDERING OUR OWN DATA TYPES

- Problem: we cannot binary search or make a `TreeSet/Map` of arbitrary types, because Java doesn't know how to order the elements
- Example: the program compiles, but crashes when we run it

```
Set<HtmlTag> tags = new TreeSet<HtmlTag>();
tags.add(new HtmlTag("body", true));
tags.add(new HtmlTag("b", false));
…
Exception in thread "main" java.lang.ClassCastException
        at java.util.TreeSet.add(TreeSet.java:238)
```

INTERFACE COMPARABLE - TEMPLATE

```
public class name implements Comparable<name> {

    ...

    public int compareTo(name other) {
        ...
    }
}
```

INTERFACE COMPARABLE - EXAMPLE

```
public class Point implements Comparable<Point> {
    private int x;
    private int y;
    ...


    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;    // same x, smaller y
        } else if (y > other.y) {
            return 1;     // same x, larger y
        } else {
            return 0;     // same x and same y
        }
    }
}
```

INTERFACE COMPARATOR

- Problem: we may want to be able to order instance of some classes by more then one property (for example, by first name and by last name). What can we do?
- Solution: use interface `Comparator<T>` - a comparison function, which imposes a total ordering on some collection of objects
  - Can be passed to a sort method, to allow precise control over the sort order
  - Can also be used to control the order of certain data structures (tree sets and tree maps)
  - Can also be used provide an ordering for collections of objects that don't have a natural ordering

METHOD COMPARE (T O1, T 02)

- `int compare(T o1, T o2)` - compares its two arguments for order and returns:
  - A negative integer if the first argument is less than the second
  - Zero, if the first argument is equal to the second
  - A positive integer if the first argument is greater than the second
- The implementor must also ensure that the relation is transitive:

  ```
  ((compare(x, y) > 0) && (compare(y, z) > 0)) implies compare(x, z) > 0
  ```

- Not strictly required, but good rule to follow: `compare()` method should be consistent with `equals()` method:

  ```
  (compare(x, y)==0) == (x.equals(y))
  ```

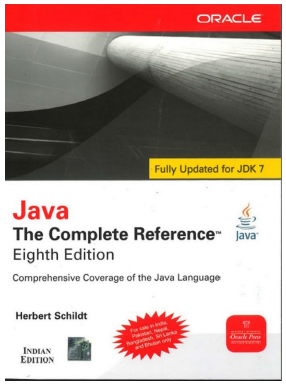
  (Any comparator that violates this condition should clearly indicate this fact. The recommended language is "`Note: this comparator imposes orderings that are inconsistent with equals.`") (don't do this!)

MAP – ANOTHER USEFUL DATA COLLECTION

CS 5004, SPRING 2022 – LECTURE 9

MAPS

- Write a program to count the number of occurrences of every unique word in a large text file (e.g. *Java Reference Manual*)



[Pictures credit: https://images-na.ssl-images-amazon.com/images/I/61rKnDmww9L.jpg]

MAPS

- Write a program that stores, modifies and retrieves:
  - Assignment grades for every student in this College
  - Financial information for every client of some bank
  - Browsing history for every user of some search engine
  - Searches and transactions for every user of some online retailer
  - Activity and likes of every user of some online platform

- Question: What do these records have in common?
- The way we think about them → every data sample has a unique user → unique ID (key)
- What is the appropriate data collection for this data?
- Maps

MAPS

- Map – a data collection that holds a set of unique *keys* and a collection of *values*, where each key is associated with one value
- Also known as:
  - 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

MAP IMPLEMENTATIONS

- In Java, maps are represented by `Map` type in `java.util`
- Map is implemented by the `HashMap` and `TreeMap` classes
  - `HashMap` - implemented using 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
  - `LinkedHashMap` - O(1)
    - Keys are stored in order of insertion

MAP IMPLEMENTATIONS

- Map requires 2 types of parameters:
  - One for keys
  - One for values

- Example:

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

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 AND VALUES

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

- Example:

```
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
}
```

METHODS KEYSET AND VALUES

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

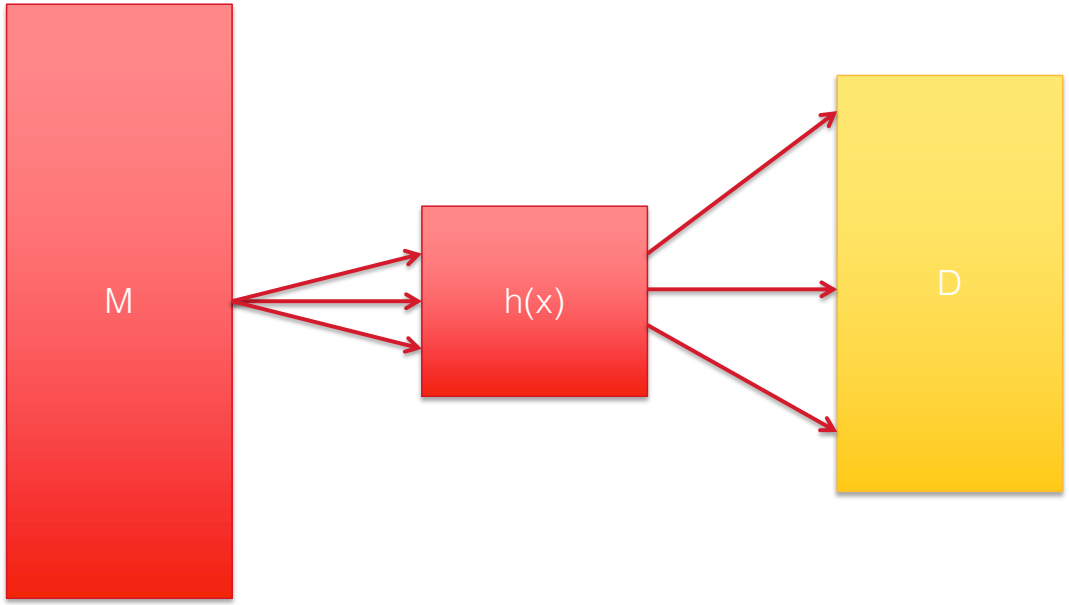
`keySet()`	returns a set of all keys in the map
`values()`	returns a collection of all values in the map
`putAll(`**map**`)`	adds all key/value pairs from the given map to this map
`equals(`**map**`)`	returns `true` if given map has the same mappings as this one

INTRODUCTION TO HASHING

CS 5004, SPRING 2022 – LECTURE 9

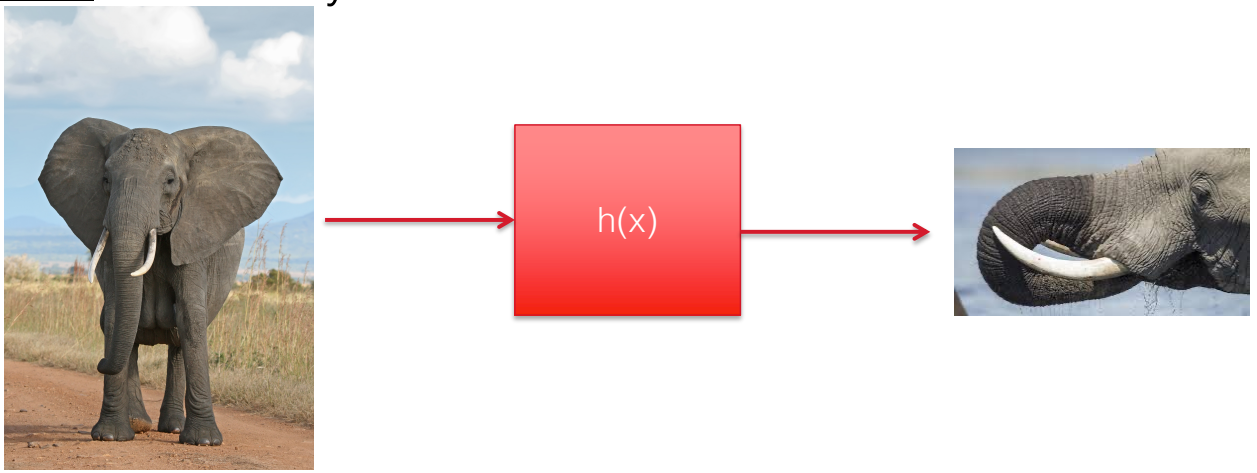
INTRODUCTION TO HASHING

- Mimicking a buddy – pseudo-random mapping using a hash function h(x)



INTRODUCTION TO HASHING

- Hash function:
  - Maps search space M into target space D
  - Map data of arbitrary size onto data of a **fixed size**



[Pictures credit: https://upload.wikimedia.org/wikipedia/commons/3/37/African_Bush_Elephant.jpg
https://aos.iacpublishinglabs.com/question/aq/1400px-788px/what-are-elephant-tusks-used-for_3c174bec-bd85-4fab-a617-7a1a33f14c62.jpg?domain=cx.aos.ask.com]

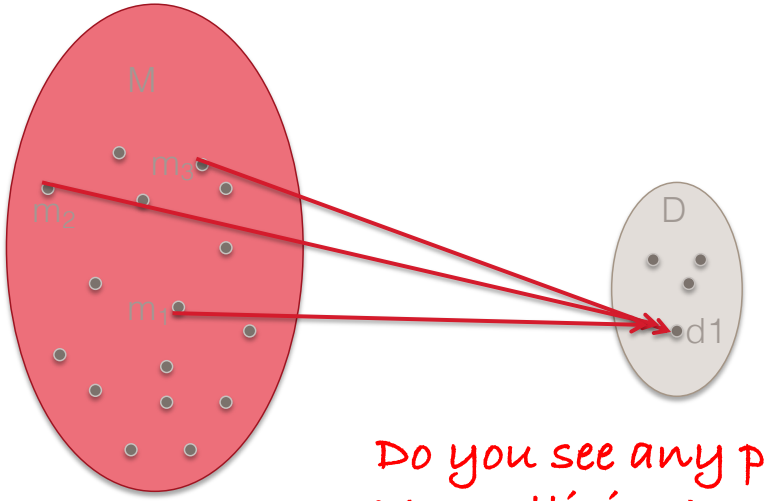
INTRODUCTION TO HASH FUNCTIONS

- Hash function – Map data of arbitrary size onto data of a fixed size

- Desired properties of hash functions:
  - Repeatability:
    - For every x in D, it should always be h(x) = h(x)
  - Equally distributed:
    - For some y, z in D, P(h(y)) = P(h(z))
  - Constant-time execution: h(x) = O(1)

PROBLEMS WITH HASH FUNCTIONS

- Hash function – can be thought of as a "lossy compression function", since |M| << |D|



Do you see any problems here?
Yes, collision!

RESOLVING COLLISIONS

- Hash function – can be thought of as a "lossy compression function", since |M| << |D|
- Problem – collision

- Possible approaches to resolve collisions:
  - Store data in the next available space
  - Store both in the same space
  - Try a different hash
  - Resize the array

INNER AND NESTED CLASSES

CS 5004, SPRING 2022 – LECTURE 9

NESTED CLASSES

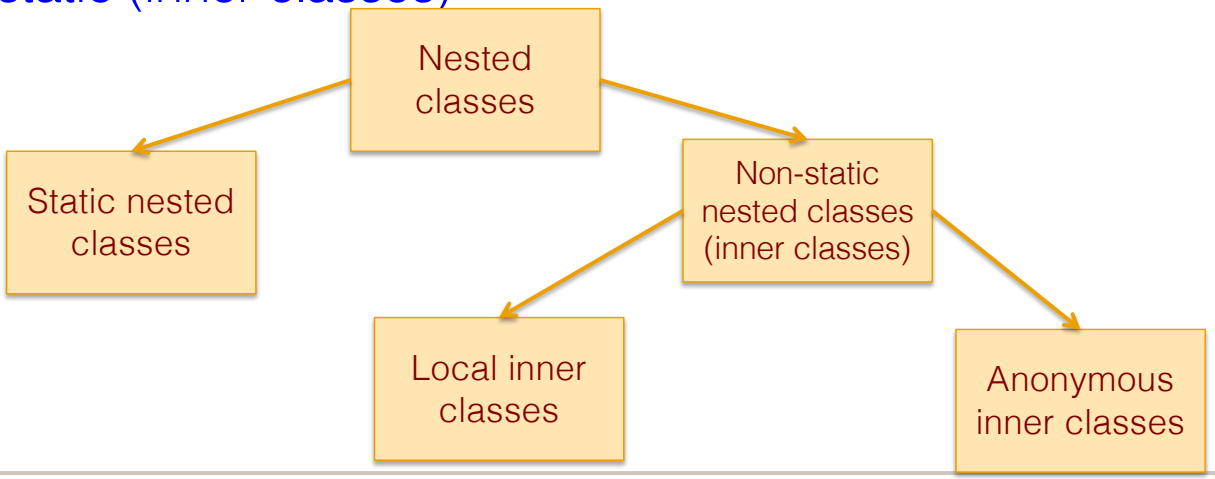
- Nested class – a class defined within some other class
- Class example: node and tree

- Some features of nested classes:
  - The scope of a nested class is bounded by the scope of its outer class (the inner class does not exist independently of its outer class)
  - A nested class is a member of its outer class
    - Can be declared private, public, protected
    - Can have access to all members (including private) of its outer class
    - Outer class does not have access to the members of the nested class

NESTED CLASSES

- Nested class – a class defined within some other class

- Nested classes can be:
  - Static classes
  - Non-static (inner classes)



STATIC NESTED CLASSES

- Behave the same way as top-level classes

```
class A {
//code for A
   static class B {
   //code for B
   }
}
```

- To access a static nested class, we need to use the name of the outer class:
  A.B b = new A.B()

INNER CLASSES

- Object of inner classes exist within an instance of the outer class

```
class X {
//code for X
   Class Y {
   //code for Y
   }
}
```

- To access an inner nested class, we need to do so through an instance of the outer class:

```
X x = new X();
X.Y y = x.new Y();
```

DIFFERENCE BETWEEN STATIC NESTED AND INNER CLASSES

- Static nested classes do not have a direct access to the non-static members of the outer class (non-static variables and methods)
  - Static class must access the non-static members of its enclosing class through an object

- Inner classes have access to all members (static and non-static, including private) of its outer class, and may refer to them directly
  - Used more frequently

WHY USE NESTED CLASSES (STATIC AND NON-STATIC)?

- A way of logically grouping classes that are only used in one place
- Increased encapsulation
- More readable and maintainable code

SHADOWING

- Shadowing – inner most declaration hides outer declaration

```
public class ShadowTest {
   public int x = 0;

   class FirstLevel {
      public int x = 1;
      void methodInFirstLevel(int x) {
         System.out.println("x = " + x);
   System.out.println("this.x = " + this.x);
         System.out.println("ShadowTest.this.x =
”ShadowTest.this.x); } }
```

SHADOWING - EXAMPLE

```
public class ShadowTest {
    public int x = 0;

    class FirstLevel {
        public int x = 1;
        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
    System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = ”ShadowTest.this.x);
    }
}

public static void main(String... args) {
    ShadowTest st = new ShadowTest();
    ShadowTest.FirstLevel fl = st.new FirstLevel();

    fl.methodInFirstLevel(23);
    }
}
```

The following is the output of this example:

x = 23

this.x = 1

ShadowTest.this.x = 0

TREES

CS 5004, SPRING 2022 – LECTURE 9

TREES

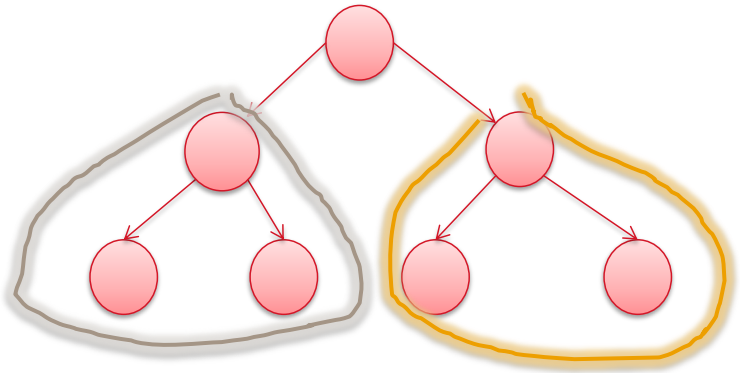
- Tree - a directed, acyclic structure of linked nodes
  - Directed - one-way links between nodes
  - Acyclic - no path wraps back around to the same node twice

- Can be defined recursively:
  - A tree is either:
    - Empty(null), or
    - A root node that contains:

      Data
      A left subtree
      A right subtree
      (The left and/or right subtree could be empty)

BINARY TREE

- Binary tree – a tree in which no node can have more than two children



BINARY TREE IMPLEMENTATION

- A basic BinaryNode object stores:
  - Data,
  - Link to the left child
  - Link to the right child
- Multiple nodes can be linked together into a larger tree

```
class BinaryNode{
//Friendly data; accessible by other package routines
   Object element;
   BinaryNode left;
   BinaryNode right;
}
```

EXAMPLE: CLASS STRINGTREENODE

StringTreeNode class

```
// A StringTreeNode object is one node in a binary tree of String
    public class StringTreeNode{
    public String data; // data stored at this node
    Public StringTreeNode left; // reference to left subtree
    Public StringTreeNode right; // reference to right subtree

    // Constructs a leaf node with the given data
    Public StringTreeNode(String data){
     this(data, null, null);
    }
    // Constructs a branch node with the given data and links
    Public StringTreeNode(String data, StringTreeNode left, StringTreeNode right){
     this.data = data;
     this.left = left;
     this.right = right;
    }
}
```

EXAMPLE: CLASS STRINGTREE

```
// An StringTree object represents an entire binary tree of String.
public class StringTree{
    private StringTreeNode root;
    //some methods
}
```

- Observations:
  - We can only talk to the `StringTree`, not to the node objects inside the tree
  - Methods of the `StringTree` create and manipulate the nodes, their data and links between them

TEMPLATE FOR TREE METHODS

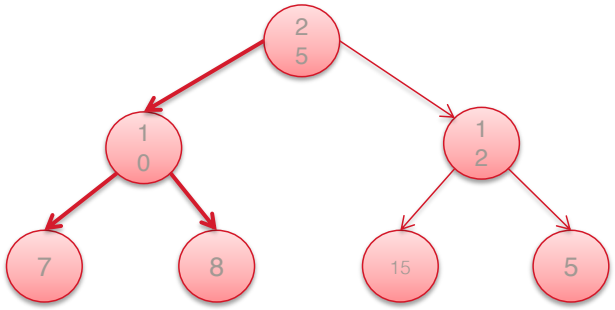
- Tree methods are often implemented recursively with a public/private pair
  - The private version accepts the root node to process

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
   public type name(parameters) {
      name(overallRoot, parameters);
   }
   private type name(IntTreeNode root, parameters) {
      ...
   }
}
```

EXAMPLE: METHOD CONTAINS()

- Add a method contains to the IntTree class that searches the tree for a given integer, returning true if it is found

- Example: If an IntTree variable tree referred to the tree below, the following calls would have these results:
  - `tree.contains(25)` → `true`
  - `tree.contains(12)` → `true`
  - `tree.contains(4)` → `false`
  - `tree.contains(77)` → `false`



EXAMPLE: METHOD CONTAINS()

```
// Returns whether this tree contains the given integer
public boolean contains(int value){
    return contains(overallRoot, value);
}
private boolean contains(IntTreeNode node, int value){
    if (node == null){
        return false; // base case: not found here
    }else if (node.data == value){
        return true; // base case: found here
    }else{
        // recursive case: search left/right subtrees
        return contains(node.left, value) || contains(node.right,
                value);
     }
}
```

SEARCH TREE ADT

CS 5004, SPRING 2022 – LECTURE 9

REFRESHER: BINARY SEARCH

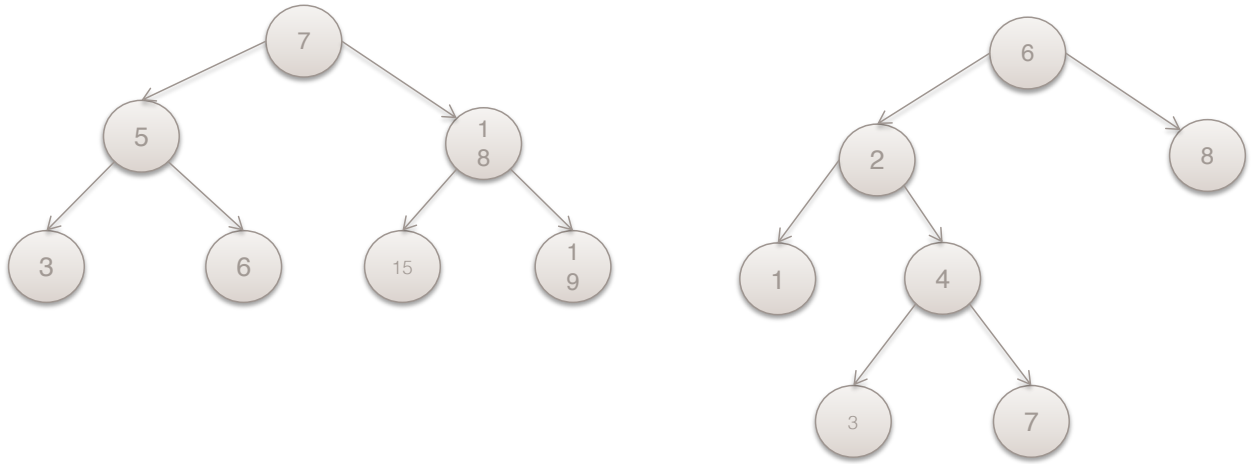
- Binary search – a search that finds a target value in a *sorted* data collection by successively eliminating half of the collection from consideration
- In the worst case, how many elements will need to examined
- Example: Find value 25 in the array below:

index	0	1	2	3	4	5	6	7	8	9	10	11
value	-5	0	6	7	13	20	25	56	78	124	203	255
	Min					Mid						Max

BINARY SEARCH TREES

- Binary search tree – a binary three that stores element in a sorted order
- Every non-empty node X of some BST has the property that:
  - Elements of X's left subtree contain data smaller than X's data
  - Elements of X's right subtree contain data greater than X's
  - X's left and right subtrees are also binary search trees

EXAMPLE: BINARY SEARCH TREE?



CLASS BINARYSEARCHTREE?

```
public class BinarySearchTree <T extends Comparable <? super T>> {
    private static class BinaryNode<T> {
    //build binary node}

    public BinaryNode<T> root;

    public BinarySearchTree() {
        root = null;
    }

    public void makeEmpty() {
        root = null;
    }

    public boolean isEmpty()  {
        return root == null;
    }
```

CLASS BINARYSEARCHTREE?

```
public class BinarySearchTree <T extends Comparable <? super T>> {
    private static class BinaryNode<T> {
    //build binary node}

    public boolean contains(T x) {
        return contains(x, root);
    }

    public T findMin() {
        if(isEmpty()) throw new NullPointerException();
        return findMin(root).element;
    }

    public T findMax() {
        if(isEmpty()) throw new NullPointerException();
        return findMax(root).element;
    }
```

CLASS BINARYSEARCHTREE?

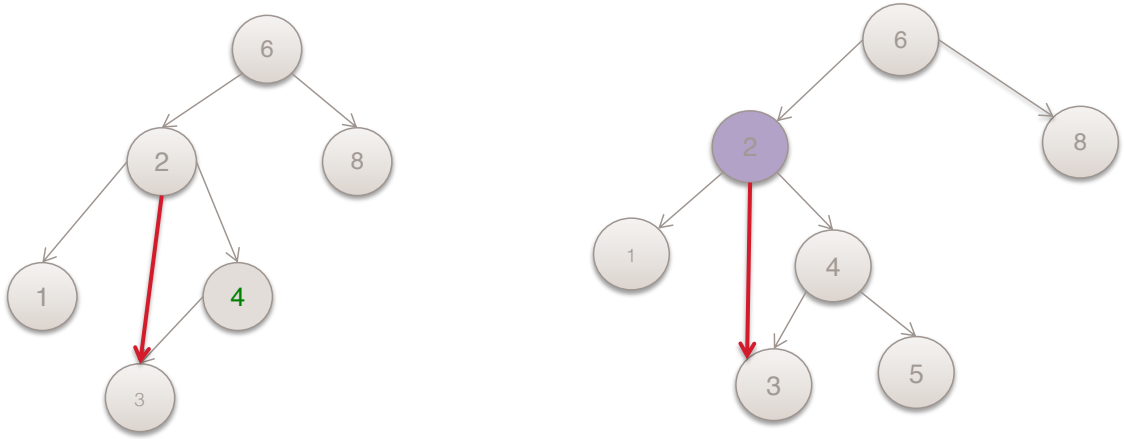
```
public class BinarySearchTree <T extends Comparable <? super T>> {
    private static class BinaryNode<T> {
    //build binary node}

    public void insert(T x) {
        root = insert(x, root);
    }

    public void remove(T x) {
        root = remove(x, root);
    }
```

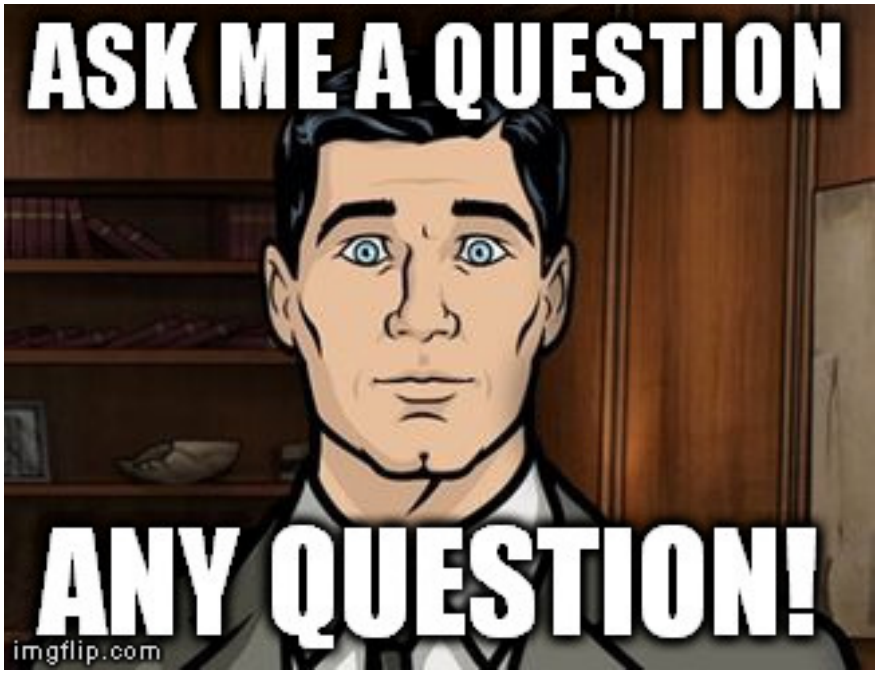
METHOD PRIVATE REMOVE(T NODE, T ROOT)

- Deletion – the hardest operation
- Once the node to delete has been found, we need to consider several possibilities:
  - Node is a leaf – can be deleted immediately
  - Node has one child – can be deleted after its parent adjusts a link to bypass it
  - Node has two children – replace data of that node with the smallest data of the right subtree



- Lazy deletion – when an element should be deleted, it is left in the tree, and merely marked as being deleted

YOUR QUESTIONS



[Meme credit: imgflip.com]

REFERENCES AND READING MATERIAL

- Mark Allen Weiss, Data Structures and Algorithm Analysis in Java, chapters 1, 2, 3, 4, 5, 6, 7, 9, 10, 12
- Oracle, java.util Class Collections, [Online] http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html
- Oracle, Java Tutorials Collections, [Online] https://docs.oracle.com/javase/tutorial/collections/
- Vogella, Java Collections – Tutorial, [Online] http://www.vogella.com/tutorials/JavaCollections/article.html
- Oracle, Java Tutorials, Nested Classes, [Online] https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
- Princeton, Introduction to Programming in Java, Recursion, [Online] http://introcs.cs.princeton.edu/java/23recursion/
- Java Tutorials, Sets, [Online], https://docs.oracle.com/javase/7/docs/api/java/util/Set.html
- Java Tutorials, Interface Map, [Online], https://docs.oracle.com/javase/8/docs/api/java/util/Map.html
- Graphs in Computer Science, [Online] http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/Graphs.html