

CS 5004

LECTURE 7

MORE GENERICS & HIERARCHICAL DATA

KEITH BAGLEY

SPRING 2022

AGENDA

- Java Generics
- Hierarchical Data
- Trees
- Java TreeSet & TreeMap
- Midterm Review

GENERICS

- *Generics* is the capability to parameterize types.
- We can define a class or a method with generic types that can be substituted using concrete types by the compiler.
 - For example: a generic stack class that stores the elements of a any type. From this generic class, you may create a stack object for holding strings and a stack object for holding numbers. Here, strings and numbers are concrete types that replace the generic type.

WHY GENERICS?

- Generics enables source code reuse at runtime
- Generics also support type-safety that enable errors to be detected at compile time rather than at runtime.
 - A generic class or method permits you to specify allowable types of objects that the class or method may work with.
 - If you attempt to use the class or method with an incompatible object, a compile error occurs.

GENERICICS

- Allows us to have compile-time safety and type replacement for similar algorithms
 - Parametric polymorphism rather than runtime polymorphism
- By using generics (called templates in other languages) we can have a List (or Set or whatever) of anything, by using a placeholder for an actual type used at compile-time

GENERIC TYPES

```
package java.lang;

public interface Comparable {
    public int compareTo(Object o)
}
```

(a) Prior to JDK 1.5

```
package java.lang;

public interface Comparable<T> {
    public int compareTo(T o)
}
```

(b) JDK 1.5

Runtime error

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

(a) Prior to JDK 1.5

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

(b) JDK 1.5

Improves reliability

Compile error

Generic Instantiation

EXAMPLE: ARRAYLIST

java.util.ArrayList

```
+ArrayList()  
+add(o: Object): void  
+add(index: int, o: Object): void  
+clear(): void  
+contains(o: Object): boolean  
+get(index: int): Object  
+indexOf(o: Object): int  
+isEmpty(): boolean  
+lastIndexOf(o: Object): int  
+remove(o: Object): boolean  
+size(): int  
+remove(index: int): boolean  
+set(index: int, o: Object): Object
```

(a) ArrayList before JDK 1.5

java.util.ArrayList<E>

```
+ArrayList()  
+add(o: E): void  
+add(index: int, o: E): void  
+clear(): void  
+contains(o: Object): boolean  
+get(index: int): E  
+indexOf(o: Object): int  
+isEmpty(): boolean  
+lastIndexOf(o: Object): int  
+remove(o: Object): boolean  
+size(): int  
+remove(index: int): boolean  
+set(index: int, o: E): E
```

(b) ArrayList since JDK 1.5

BEFORE GENERICS

```
// raw type using Object  
ArrayList list = new ArrayList();
```

This is *roughly* equivalent to

```
ArrayList<Object> list = new ArrayList<Object>();
```


GENERIC: NO CASTING NEEDED

```
ArrayList<Double> list = new ArrayList<>();
```

```
list.add(5.5); // 5.5 is automatically converted to new Double(5.5)
```

```
list.add(3.0); // 3.0 is automatically converted to new Double(3.0)
```

```
Double doubleObject = list.get(0); // No casting is needed
```

```
double d = list.get(1); // Automatically converted to double
```

GENERIC CLASSES

GenericStack<E>

-list: java.util.ArrayList<E>

+GenericStack()

+getSize(): int

+peek(): E

+pop(): E

+push(o: E): void

+isEmpty(): boolean

An array list to store elements.

Creates an empty stack.

Returns the number of elements in this stack.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns true if the stack is empty.

GENERICIS: SIMPLE EXAMPLE – ENVELOPE/LETTER

```
// Example of Java Generics
// Placeholder T represents any type. Allows for compile time type-safe code
public class GenericsExample<T> {

    private T data;

    public T getData(){
        return this.data;
    }

    public void setData(T data){
        this.data=data;
    }

    public static void main(String [] args) {
        GenericsExample<Dog> d = new GenericsExample<Dog>();
        d.setData(new Dog( name: "Fifi", isMale: false, age: 12));
        System.out.println(d.getData().getName() + " " + d.getData().getAge());

        /* Two different types of containers even though it shares same implementat
           Cannot put a Person in a Dog container
           // d.setData(new Person("Fannie", 22)); // <-- Error!
        */

        GenericsExample<Person> p = new GenericsExample<Person>();
        p.setData(new Person( name: "Fannie", age: 22));
        System.out.println(p.getData().getName() + " " + p.getData().getAge());
    }
}
```

Typically

- T is used as a placeholder for Type
- E as a placeholder for Element
- Etc.

COLLECTIONS WE DIDN'T TOUCH ON

- Sets
- Maps

THE SET INTERFACE

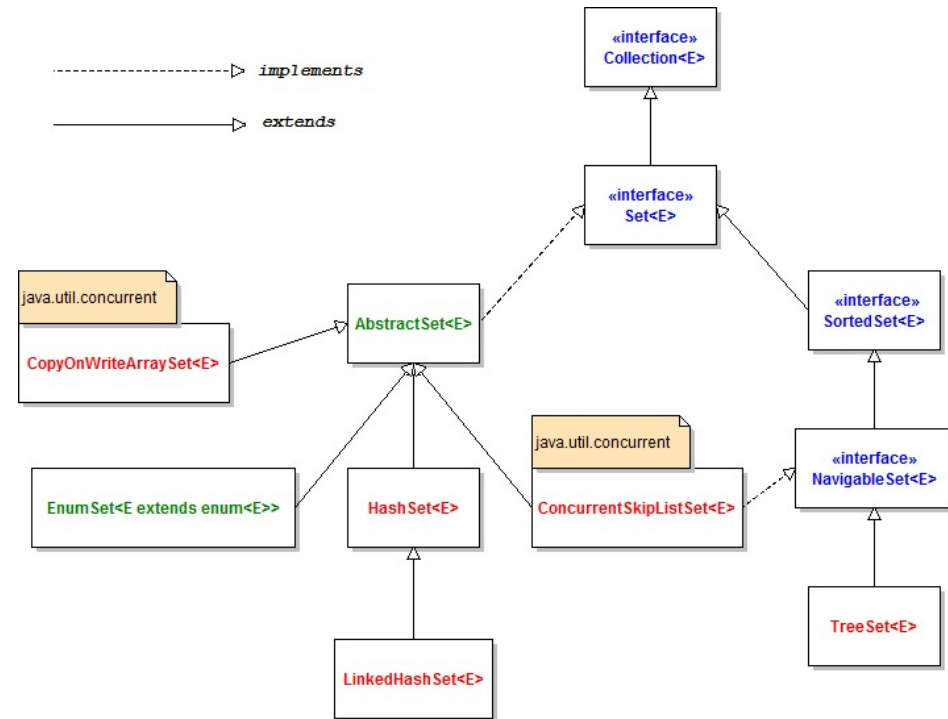
- The Set interface extends the Collection interface. It does not introduce new methods or constants, but it stipulates that an instance of Set contains no duplicate elements.
- The concrete classes that implement Set must ensure that no duplicate elements can be added to the set. That is no two elements `e1` and `e2` can be in the set such that `e1.equals(e2)` is true.

SET

- **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



SET API CLASS DIAGRAM



[Pictures credit:

<http://www.codejava.net/images/articles/javacore/collections/Set%20API%20class%20diagram.png>]

SET

- No duplicates, no index

```
import java.util.*;

public class TestHashSet {
    public static void main(String[] args) {
        // Create a hash set
        Set<String> set = new HashSet<>();

        // Add strings to the set
        set.add("London");
        set.add("Paris");
        set.add("New York");
        set.add("San Francisco");
        set.add("Beijing");
        set.add("New York");

        System.out.println(set);

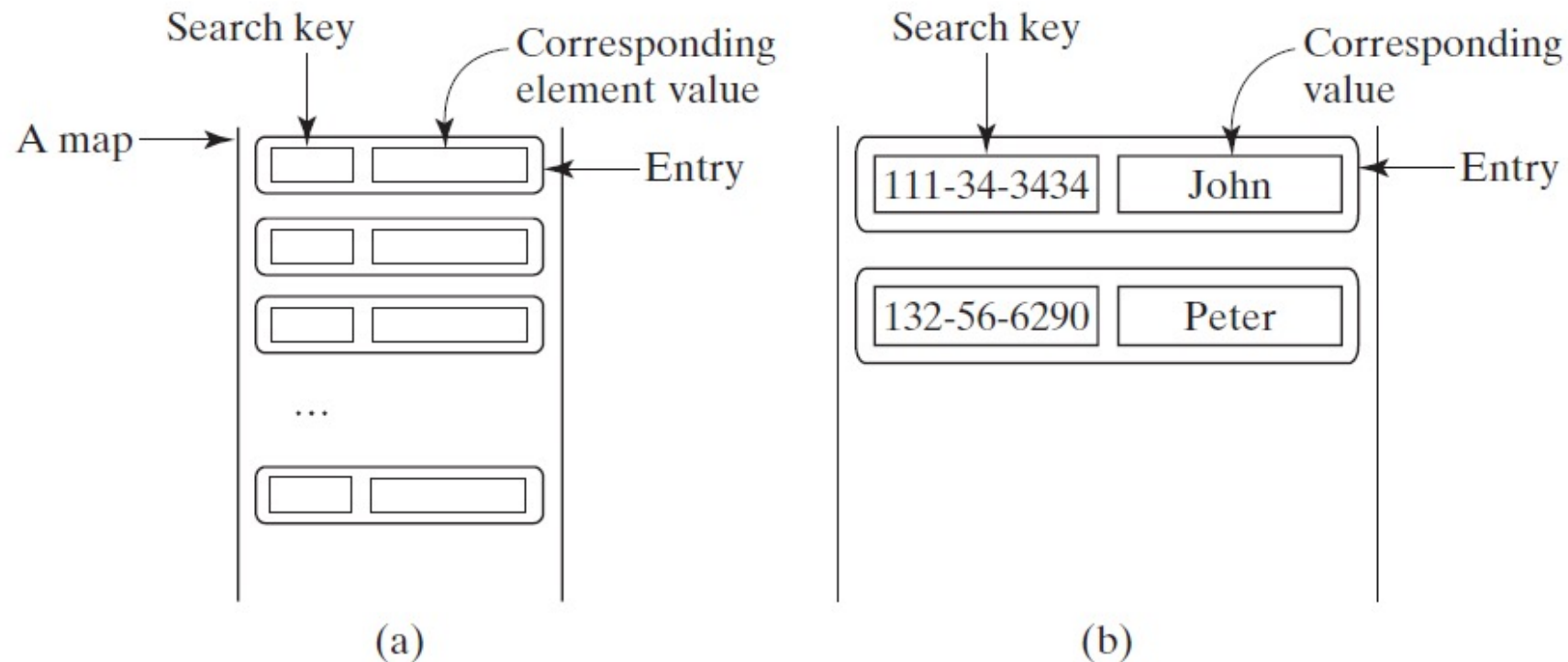
        // Display the elements in the hash set
        for (String s: set) {
            System.out.print(s.toUpperCase() + " ");
        }

        // Process the elements using a forEach method
        System.out.println();
        set.forEach(e -> System.out.print(e.toLowerCase() + " "));
    }
}
```


THE MAP INTERFACE

The Map interface maps keys to the elements. The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.

*** Similar to a **Python Dictionary**



HASHMAP & LINKEDHASHMAP

The HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping.

LinkedHashMap extends HashMap with a linked list implementation that supports an ordering of the entries in the map. The entries in a HashMap are not ordered, but the entries in a LinkedHashMap can be retrieved in the order in which they were inserted into the map (known as the insertion order), or the order in which they were last accessed, from least recently accessed to most recently (access order).

HASHMAP

- A HashMap is a concrete class

```
import java.util.HashMap;
public class Lingua {
    private HashMap<String, String> dictionary;

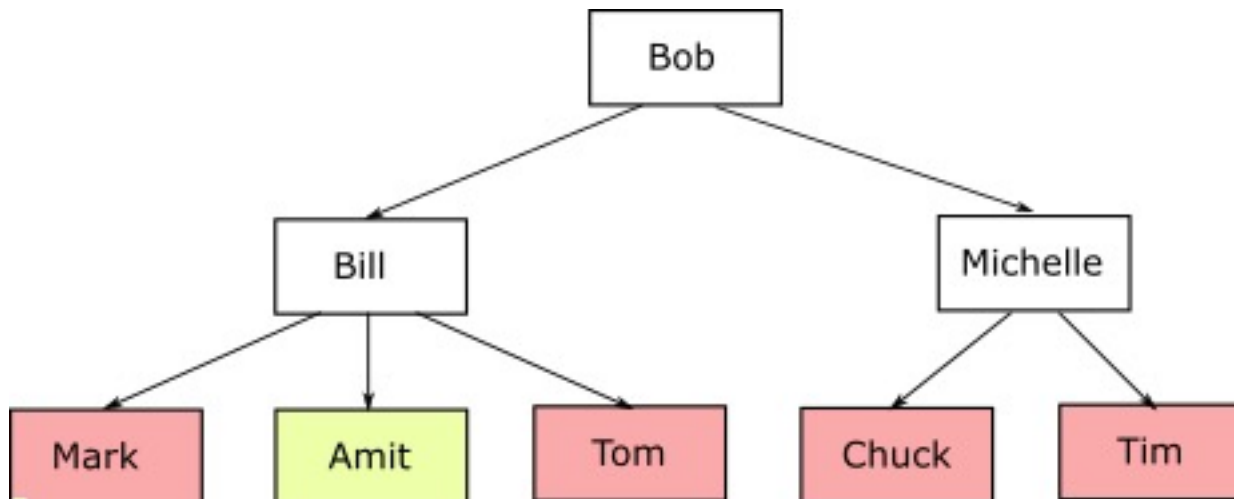
    public Lingua() {
        dictionary = new HashMap<String, String>();
        dictionary.put("Bonjour", "Hello");
        dictionary.put("Goodbye", "Au revoir");
        dictionary.put("Ca va?", "How's it going?");
    }

    public HashMap<String, String> getDictionary() { return dictionary;}

    public static void main(String [] args) {
        Lingua lingo = new Lingua(); // make a new instance
        String [] words = {"Bonjour", "Goodbye", "Ca va?"};
        for (String each : words ) {
            System.out.println("When someone from France says: " + each);
            System.out.println("In English we say: " + lingo.getDictionary().get(each));
        }
    }
}
```

HIERARCHICAL DATA

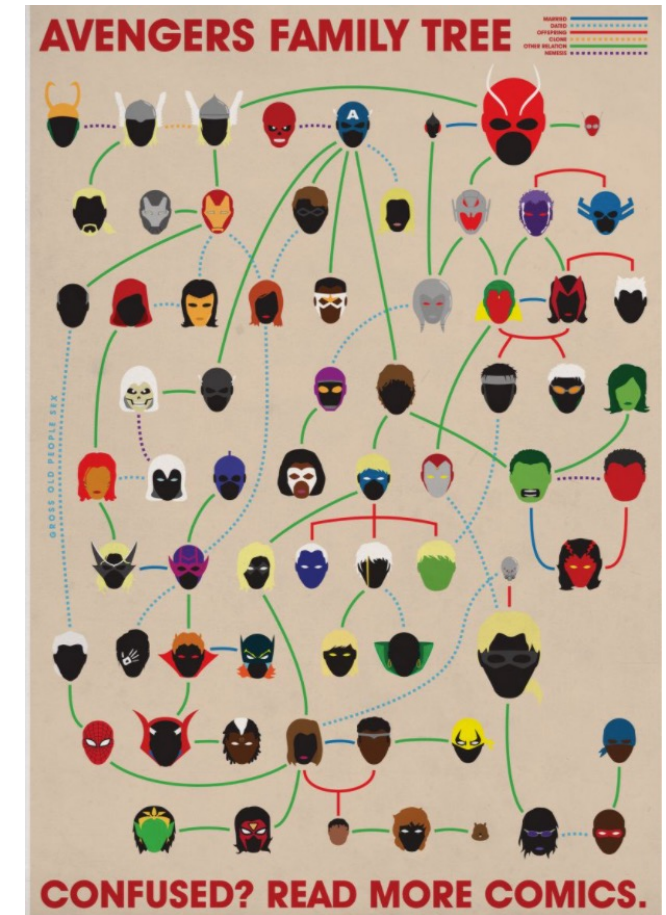
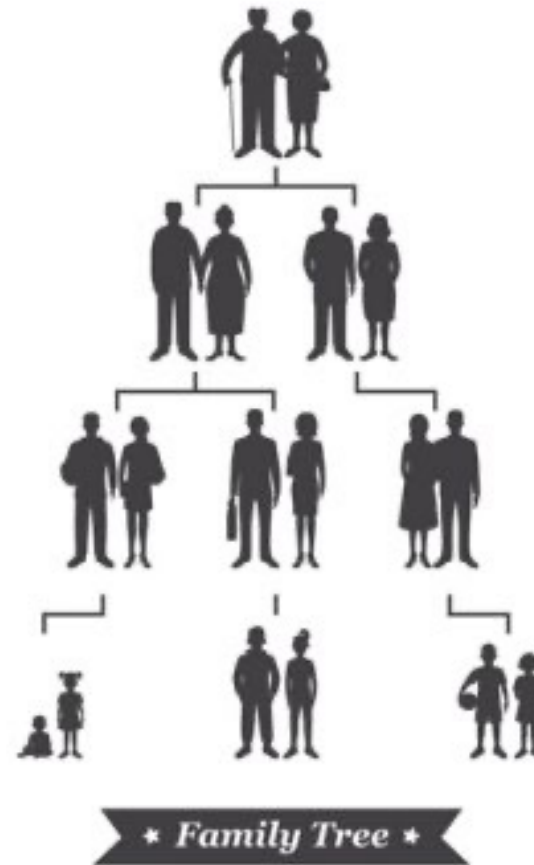
- Hierarchical data structures are non-linear, tree-like structures
 - Trees are the typical structures, but also could include non-root-based hierarchies



Organization hierarchy

TREES – (NON COMPUTING)

- A Tree is a visualization instrument to help show a hierarchy of related elements



Avengers family tree source: <https://ifanboy.com/articles/the-avengers-family-tree/>

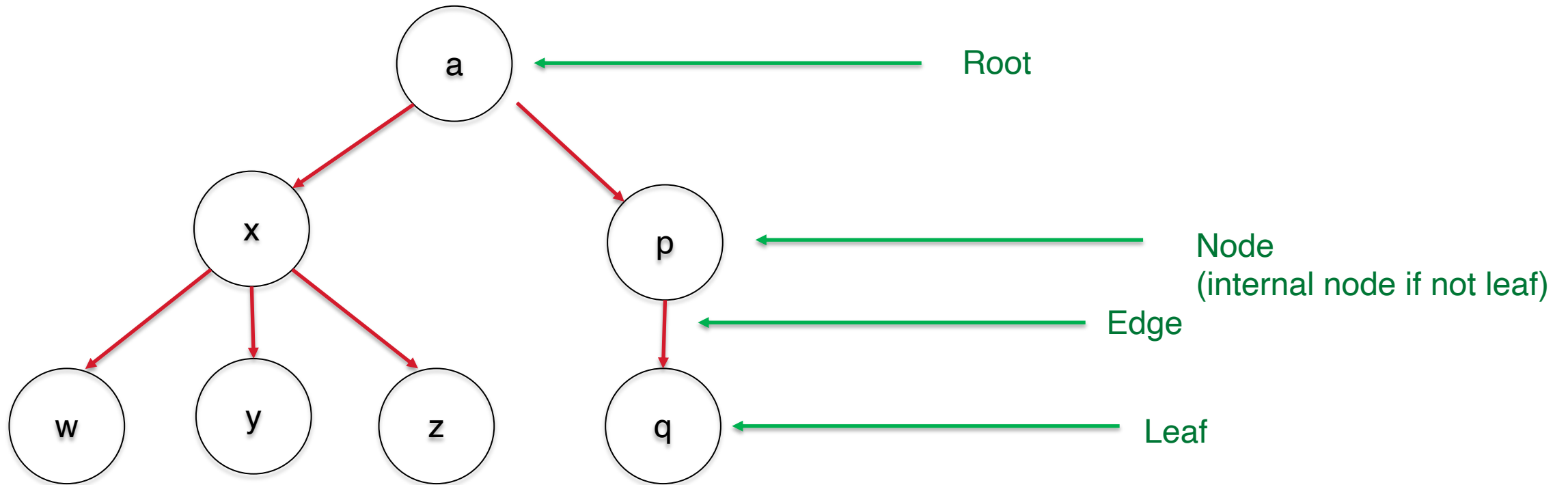
TREES (COMPUTATIONAL)

- A Tree is an abstract data type that stores elements hierarchically.
 - Useful for representing parent-child relationships
- With the exception of the top element (root) each element in a tree has a parent element and zero or more children elements
- Common uses:
 - Storage as hierarchy. File systems, document structures (pdf, html, etc.)
 - Searching. Trees can facilitate fast search/retrieval (BST).
 - Parsing, Language processing

TYPES OF TREES

- Two broad categories:
 - General Trees
 - No constraints on the number of children (degree)
 - N-ary (or M-ary) Trees
 - No more than n (or m) children per parent

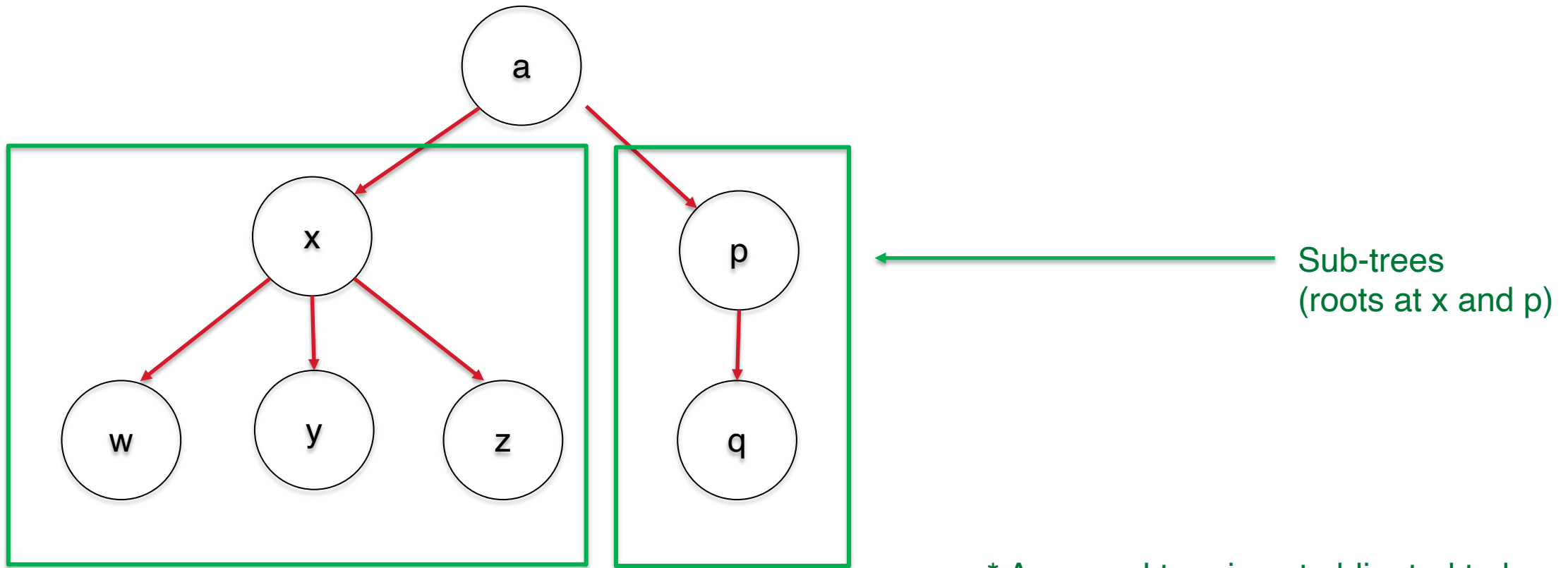
GENERAL TREES



* A Leaf is a node that has a parent, but no children

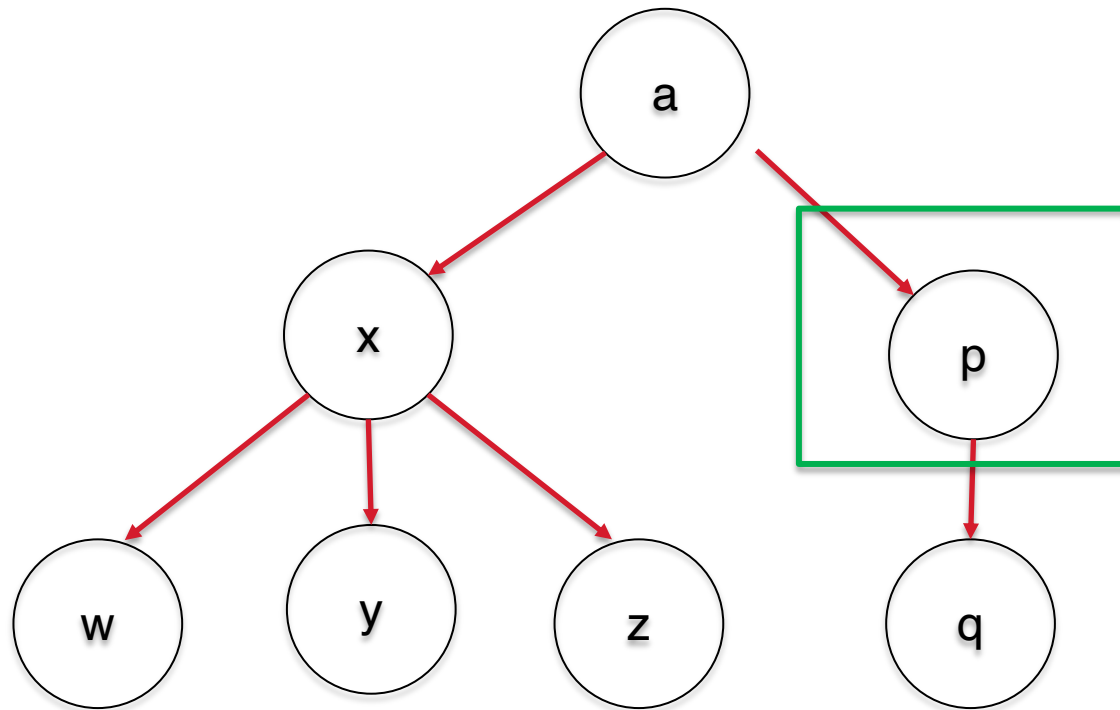
Content inspired by Prof Paul Weiss, CUNY

GENERAL TREES - SUBTREES



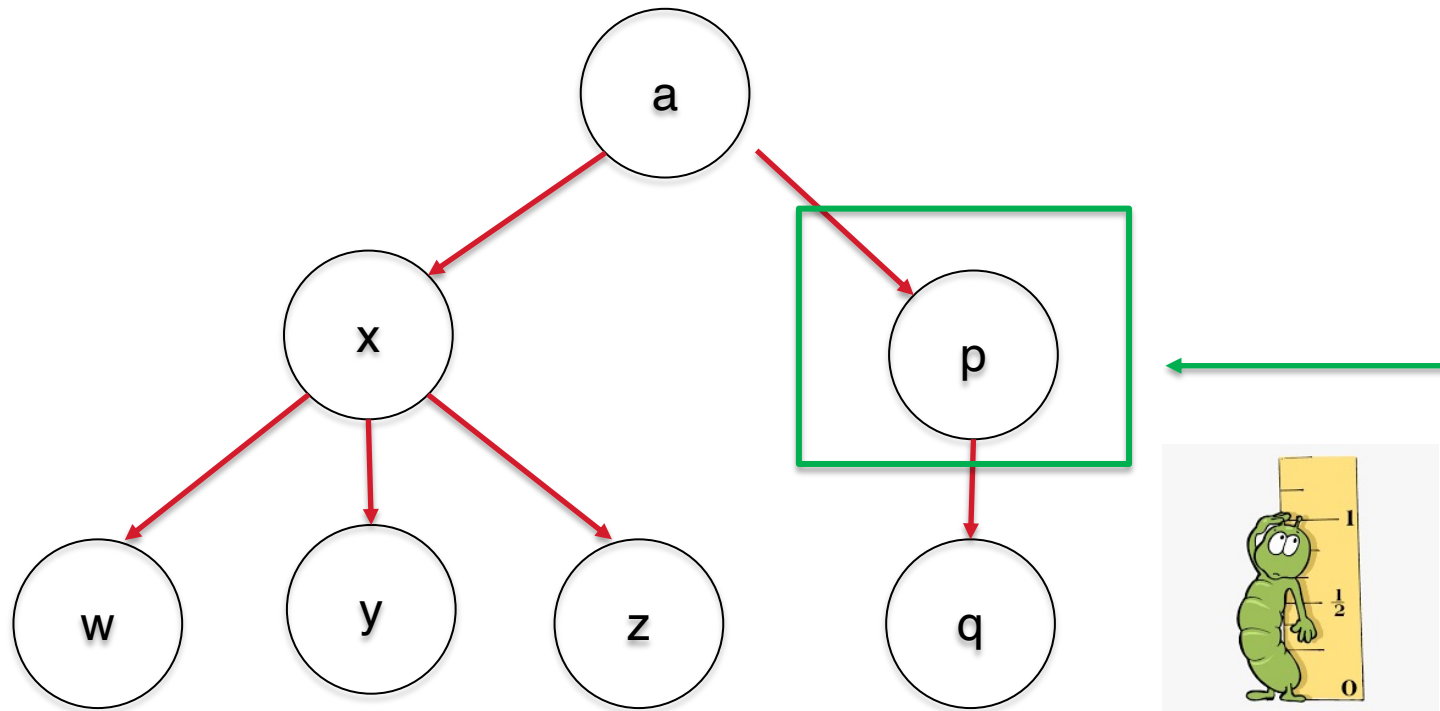
* A general tree is not obligated to have any sub-trees

GENERAL TREES - DEGREE



Degree of a node = number of children.
(p is degree 1; x is degree 3)

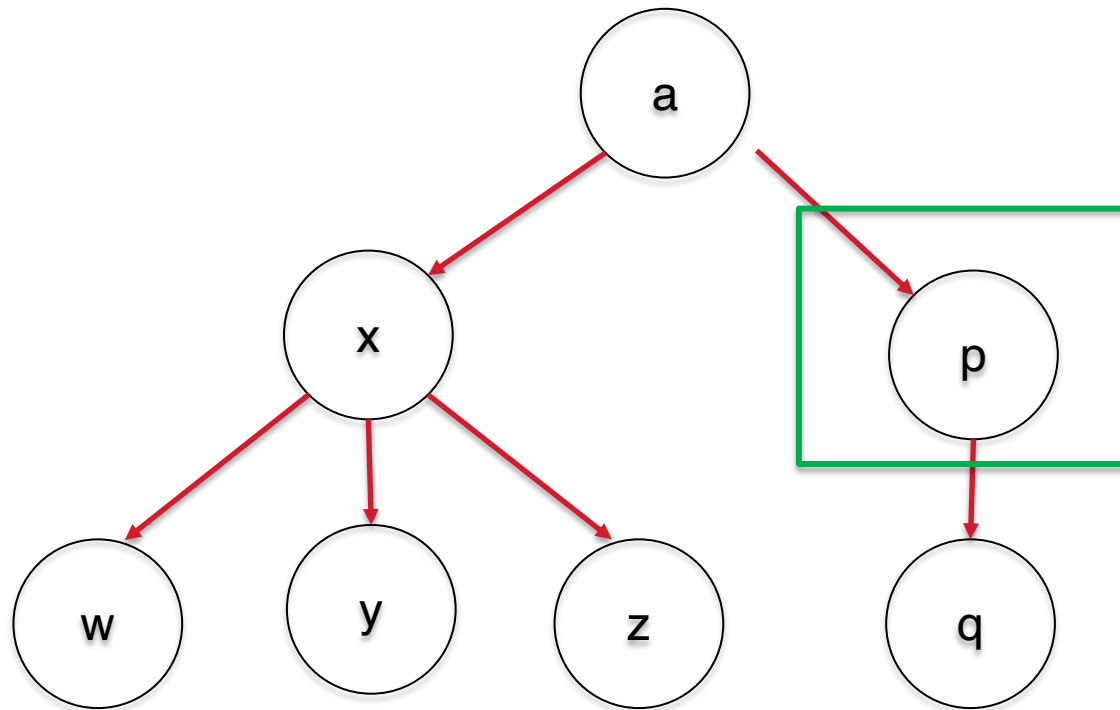
GENERAL TREES – HEIGHT



Height of a node is the length of the longest path from the node to any of its descendants (children, grandchildren, etc.). Leaf nodes have height 0

Height of p and x is 1. Height of the Tree is 2 (root a)

GENERAL TREES – DEPTH



Depth or Level of a node is the length of the path from the root to the node.

Depth of x & p is 1

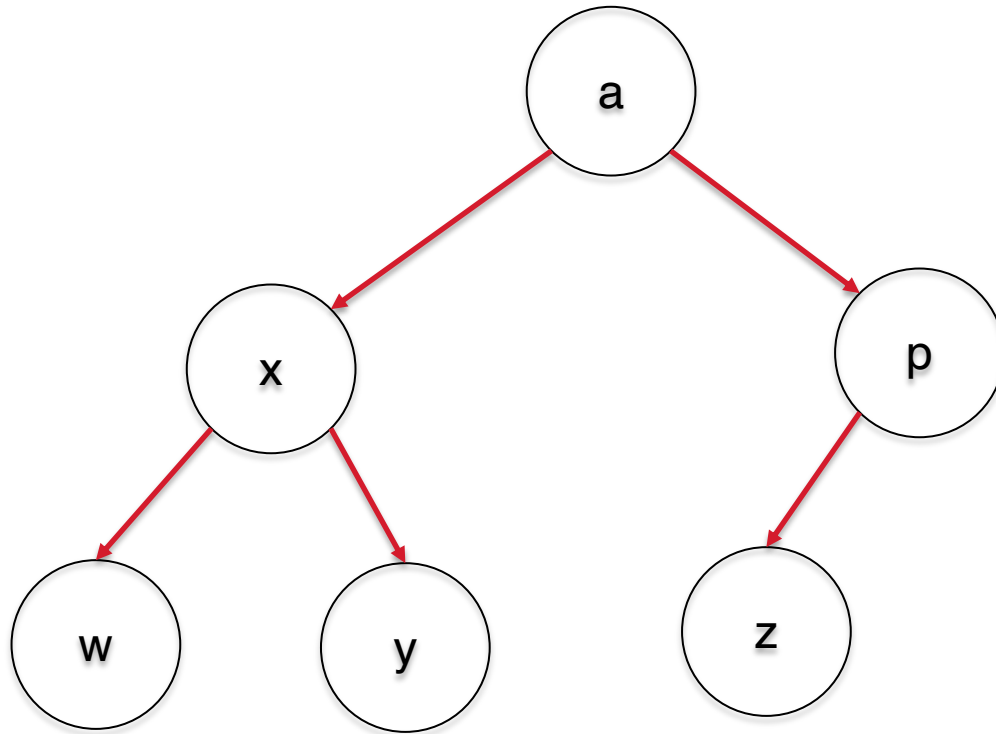
Depth of w, y, z, q is 2

The root has depth 0

N-ARY (OR M-ARY) TREES

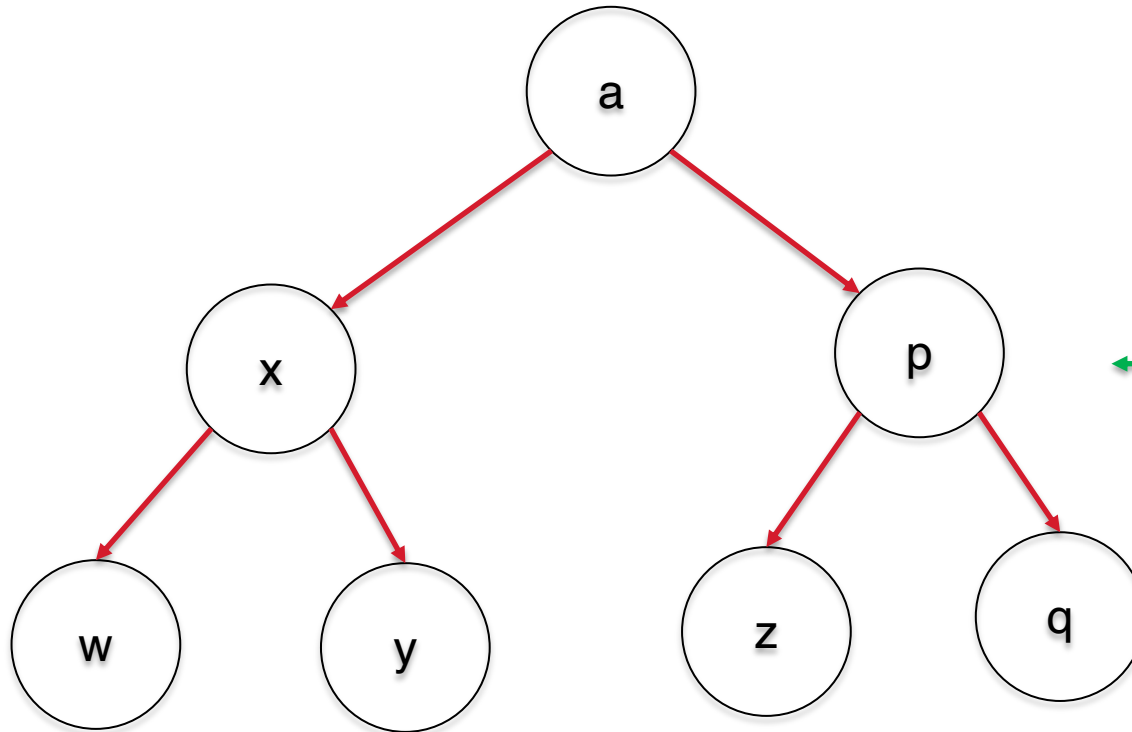
- N-ary Trees are constrained: the degree of any node is bounded by n
 - The degree for any node can never be greater than n
- Ternary Trees are 3-ary trees
- Binary Trees are 2-ary trees
 - Binary trees also have the quality of being able to distinguish the root and left and right subtree (which is also binary)

BINARY TREE



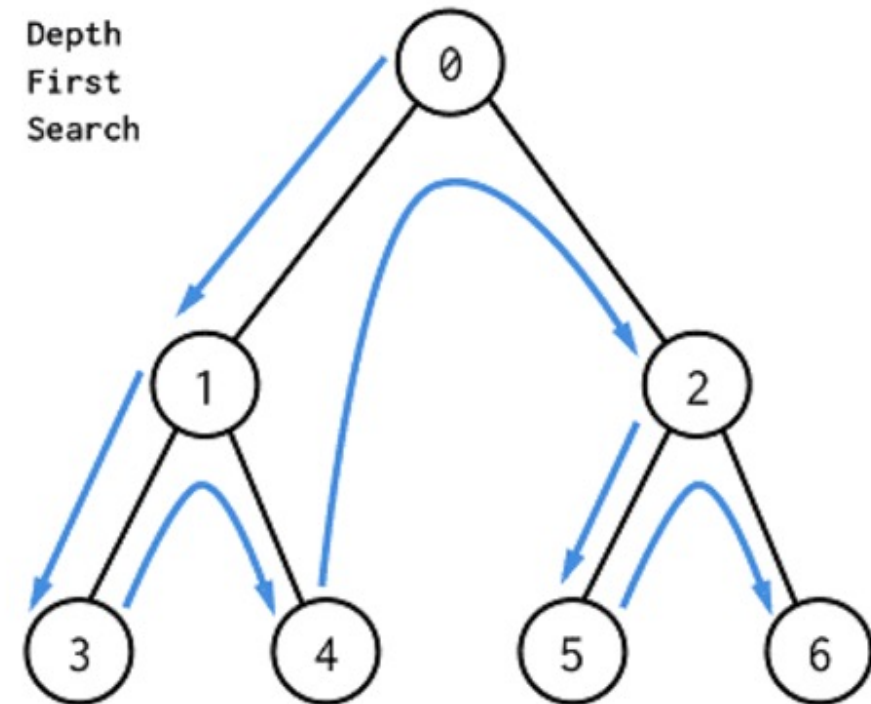
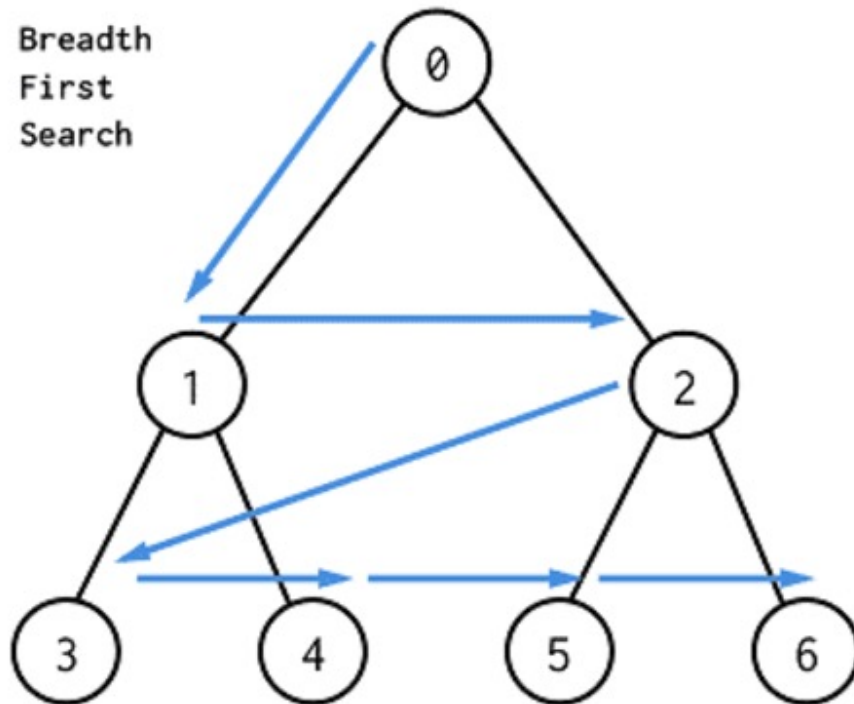
← **Complete** binary tree has every level, except the last, is filled and all nodes in the last level are filled as far left as they can be.

BINARY TREE



Full binary tree has as many possible nodes for its height. Every node that's not a leaf node has two children

TREES: SEARCHING – BFS VS. DFS



Shown here: Pre-order DFS

Image from Dev Community at: <https://dev.to/danimal92/difference-between-depth-first-search-and-breadth-first-search-6om>

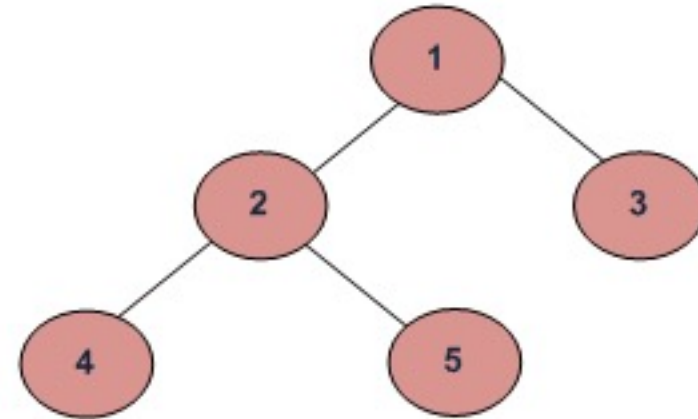
DEPTH-FIRST: PRE-ORDER, IN-ORDER, POST-ORDER

Unlike linear data structures, trees can be traversed in different ways. DFS has multiple options.

Depth First Traversals:

- (a) **Inorder** (Left, Root, Right) : 4 2 5 1 3
- (b) **Preorder** (Root, Left, Right) : 1 2 4 5 3
- (c) **Postorder** (Left, Right, Root) : 4 5 2 3 1

And of course, **Breadth First**: 1 2 3 4 5



<https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/>

DEPTH-FIRST: PRE-ORDER, IN-ORDER, POST-ORDER

Another Example

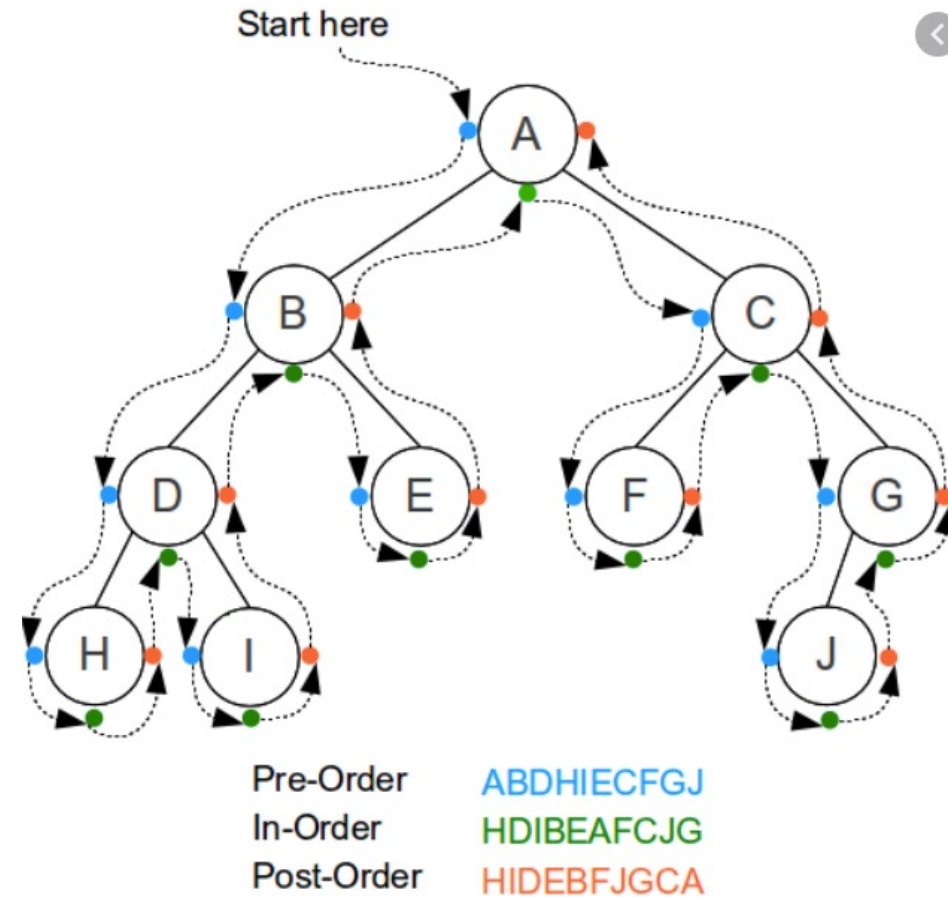


Image courtesy of <https://thenafi36.wordpress.com/2014/09/26/uva-536-tree-recover/>

JAVA DOESN'T HAVE A COLLECTIONS TREE...

- But we can use the MutableTreeNode from the Swing library. It implements the tree structure used to represent things like folders, directories, etc.

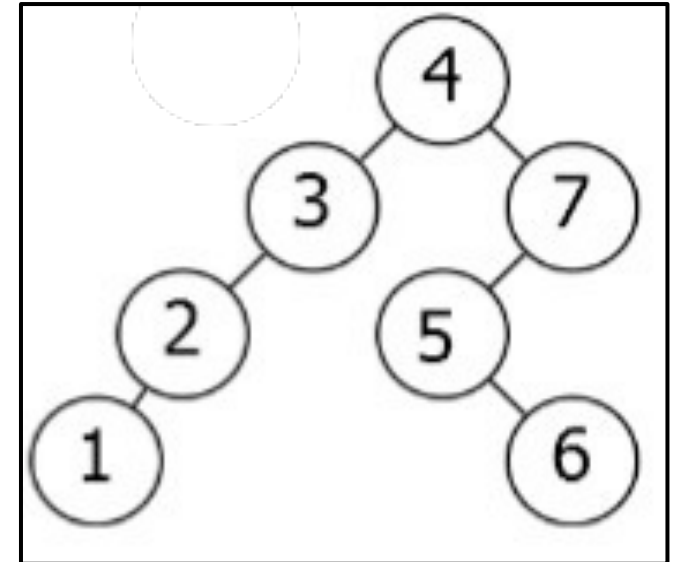
```
DefaultMutableTreeNode root = new DefaultMutableTreeNode( userObject: "Root");
DefaultMutableTreeNode mercury = new DefaultMutableTreeNode( userObject: "Mercury");
root.add(mercury);
DefaultMutableTreeNode venus = new DefaultMutableTreeNode( userObject: "Venus");
DefaultMutableTreeNode moon = new DefaultMutableTreeNode( userObject: "Moon 1");
venus.add(moon);
venus.add(new DefaultMutableTreeNode( userObject: "Moon 2"));

root.add(venus);
DefaultMutableTreeNode mars = new DefaultMutableTreeNode( userObject: "Mars");
root.add(mars);

System.out.println(root.getChildCount());
```

BINARY SEARCH TREE (BST)

- Binary Search Tree(BST) are Binary Trees with a constraint.
 - Each node contains elements lesser than itself on the left side and elements greater than itself on the right side.
 - The left subtree of a particular node contains elements that are smaller in value than the current node.
 - The right subtree of a particular node contains elements that are greater than the value of the current node.



EXAM 1 REVIEW

TERMINOLOGY REVIEW

- **Class:** a blueprint for how to make an object. Classes describe the “thing” (abstraction) we’re talking about – its:
 - **attributes** (data it owns/knows),
 - **operations** (behavior, what it knows how to do), and
 - **state and relationships** to other things
- **Object:** an “instance” (or one of the “things”) from the class
 - Objects have attributes, operations and unique identity

Other (almost) synonyms you might hear:

Instance variables -> attributes

Methods or “member functions” -> operations

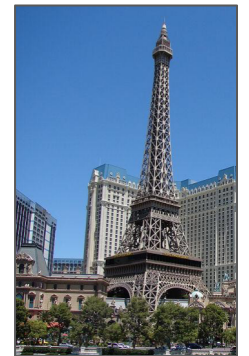
blueprint



Paris



Las Vegas



SOME TERMINOLOGY

- **Encapsulation:** an object's data is packaged together with its corresponding operations
- **Information Hiding:** uses encapsulation to emphasize the separation of external "visible" properties of an object from internal hidden properties, and restrict access by enforcing well-defined interfaces



Encapsulation



Information Hiding

Encapsulation

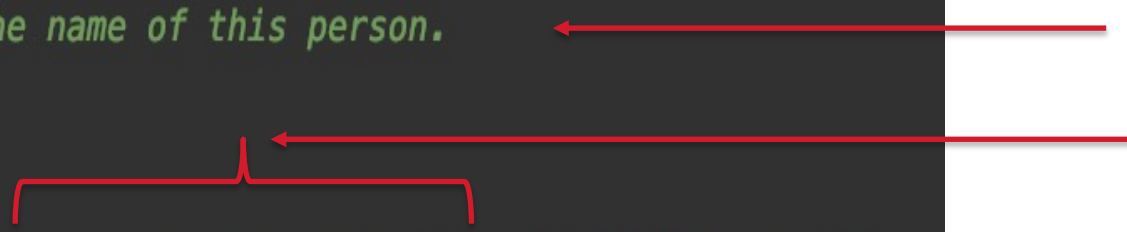
Information Hiding

```
public class Square implements Shape {  
    private String name;  
    private double length;  
  
    /**  
     * Constructor for the Square class. Create  
     *  
     * @param length the length of the square  
     */  
  
    public Square(double length) {  
        this.name = "Square";  
        this.length = length;  
    }  
}
```

METHODS

- Methods have a signature that tells users the name, parameters and return type
- Methods have a method body which defines the code that is executed when invoked
- Methods may be **overridden** or **overloaded**

```
/**
 * Change the name of this person.
 *
 */
public void changeName(String name) { this.name = name; }
```



Javadoc comments w/ purpose,
return values, parameters

Signature

- `changeName(String)` is the signature in this example

TESTING AND TEST COVERAGE

- Testing validates and verifies certain aspects of the system
 - Unit testing examines discrete elements of code smaller than the subsystem level
- Test coverage of code uses unit testing to examine the code written by developers.
 - Writing automated unit tests (JUnit/PyUnit/etc.) for each method increases our *code coverage*
 - Writing test suites for automation PLUS manual testing increases our overall test coverage for *testing requirements*
- Regardless of role, everyone should be involved in QA and testing at some level
- Test Driven Development is an iterative approach to development (and testing) that keeps code in lockstep with appropriate unit tests

HOW DO WE UNIT TEST?

- Remember, we're going to write tests for a piece of a class
 - In general, classes should do one thing well – **High Cohesion**
 - Methods should have a purpose, and focus on doing one thing for that “one thing” that the class is responsible for
- Put on your “tester/QA” hat and try to “break” the system
- **Do NOT wait** until the system is fully built to start testing

Examples

- Verify “good state” initialization after constructor
- `toString()` override returns proper value in proper format
- Values and calculations correct?
- Values and calculations correct and MUTATES instance variables properly?
- Boundary Conditions: Any constraints on values? Does the method work correctly? (e.g. with 0? Negative numbers?)
- Verify proper throw exceptions for error conditions when checked runtime errors arise? (e.g. `ArithmeticException` or `IllegalStateException` for our `SimpleFraction` class – `reciprocal()`)
- Verify all transformations work properly (e.g. `reciprocal()`)
- Verify any calculations on external objects/values work (e.g. `.multiply(otherFraction)`)
- Etc.

ENUMERATED TYPES

- An enumerated type (enum in Java) is a kind of “named constant” in which you provide a short list of possible values upon specification
 - Typically, we are not concerned about the *underlying value* how the enumeration is represented
 - We do care about the *conceptual* value
 - We may care about *sequencing* (predecessor/successor relationship)

```
public class CalendarEvent {  
    // create a set of values  
    enum DAY { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };  
  
    private DAY dayOfEvent;  
    private String eventName;  
}
```

ABSTRACTION & ABSTRACTION

- Abstraction (the concept) allows us to ignore unessential details
 - Verb: Abstraction as an activity: the *process* of thinking where unessential details are ignored, and conceptual “things” are produced to help us solve problems
 - Noun: Abstraction as an entity: The resulting “things” that are produced from the process of abstraction

INTERFACES & INHERITANCE

- Interfaces are an abstract specification for some type.
 - Interfaces specify external behavior
 - Abstract and Concrete types will implement interfaces
 - The implementing types may be conceptually unrelated to each other
 - Interfaces allow for cross-hierarchy (unbounded) polymorphism
- Inheritance allows new classes to be created from existing classes
 - The new class reuses features (attributes & operations) from the existing class (superclass aka base class aka parent class)
 - Code reuse is one focus of inheritance. Conceptual hierarchies is another
 - Subclasses can also add/extend new features to what they get from their superclasses
- Abstract Classes can opt to provide some implementation for subtypes, but cannot be instantiated

SUBTYPING VS SUBCLASSING

- Remember: protocols/contracts (Java interfaces) are TYPE specifications
 - The protocol tells us what the type does, NOT how the type does it, or what kind of internal representation it uses
 - In CS5001, we learned about Abstract Data Types (like Queues and Stacks). Recall we chose a Python List for one implementation, but that was NOT the only alternative
- Remember: classes are ONE style of implementation. In Java it is THE style of implementation. In other languages (like C) there is no such thing as a class, so there is no such thing as “subclassing”.
 - Types and subtyping is a concept that is consistent across all languages, and even hardware

SUBTYPING VS SUBCLASSING 1

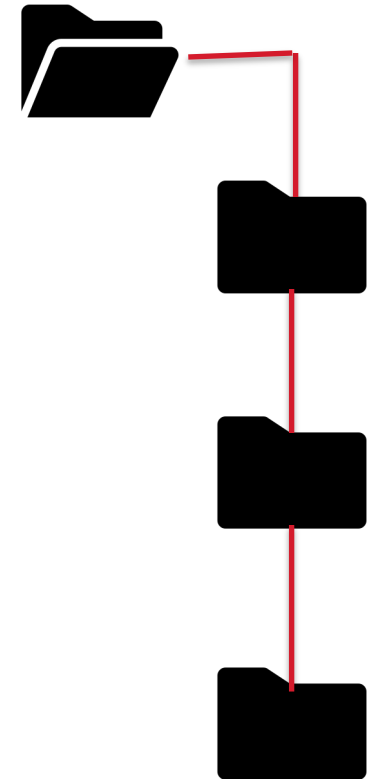
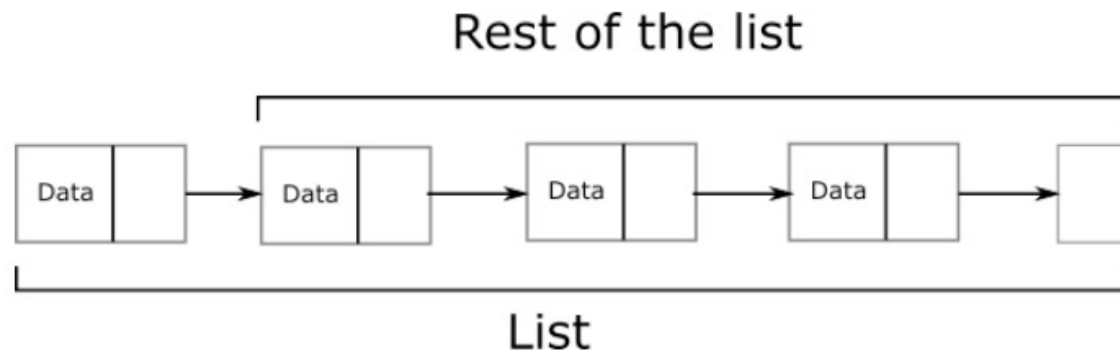
- Consider the Type: Lock
 - Simple protocol:
lock()
unlock()



Subtypes: Hardware cylinder lock, Keycard lock (hardware/software), Cyber security lock
Implementation is different. No “subclassing”. Classes are a OO software mechanism

RECURSIVE DATA STRUCTURES

- We can represent some complex forms of data recursively such that an element in our structure refers to another element of the same type, and so forth
 - This is not the same as recursive functions that you are already familiar with, but a recursive structure which is fractal-like – a repeating structure from top to bottom



PREDICATES & LAMBDA

- Lambda functions allow us to:
 - write a method that takes a (predicate) object as an argument and uses it.
 - In Java, functions cannot be passed as arguments directly, because Java treats *data* and *methods* as two different things. Therefore we enclose the function inside a (Predicate) object, and then pass it around.
- Most functional programming languages do not distinguish between data and functions. The approach in those languages is more concise than in Java.

EXAMPLE: PREDICATES & LAMBDA

And after we've connected the nodes, we want to filter

```
@Test
public void testFilterLambda() {
    int year = 11;
    // Lambda is an anonymous function. Here is the same as the above. Less verbose since
    // test() is the only method in the class.
    // Lambda can be constructed parameters -> body_of_function (open/close braces not needed for
    // 1-liners. Also note that the compiler can infer some types. If it has trouble, the first
    // line could have been written:
    //         IListOfDog youngList = node3.filter((Dog dog) -> dog.getAge() < year);
    IListOfDog youngList = node3.filter(dog -> dog.getAge() < year);
    assertEquals( expected: "(Rex)", youngList.toString());
}
```

COMPARATORS

- In a similar way, we can create general-purpose comparators to help us sort elements in a Collection

```
@Test
public void testSortComparator() {
    IListOfDog sortedList = node3.sort(new DogAgeComparator());
    assertEquals( expected: "(Rex)(Fifi)(Nero)", sortedList.toString());

    // *** -> Class Exercise: Write a DogName Comparator so this will work:
    // IListOfDog sortedList1 = node3.sort(new DogNameComparator());
    // assertEquals("(Fifi)(Nero)(Rex)", sortedList1.toString());

    IListOfDog sortedList3 = node3.sort(new DogDescendingAgeComparator());
    assertEquals( expected: "(Nero)(Fifi)(Rex)", sortedList3.toString());
}
```

JAVA COLLECTIONS & GENERICS

- Although we are looking “under the hood” to understand how collections like lists are designed and implemented, Java provides many of these collections as generic types that we can use “out of the box”
- Java is strongly typed and generally “type safe”, so unlike Python (which uses dynamic typing) Java uses the concept of generics to be able to reuse source (not .class/.obj) implementation across types
- By using generics (called templates in other languages) we can have a List (or Set or whatever) of anything, by using a placeholder for an actual type used at compile-time

GENERICS

```
// Example of Java Generics
// Placeholder T represents any type. Allows for compile time type-safe code
public class GenericsExample<T> {

    private T data;

    public T getData(){
        return this.data;
    }

    public void setData(T data){
        this.data=data;
    }

    public static void main(String [] args) {
        GenericsExample<Dog> d = new GenericsExample<Dog>();
        d.setData(new Dog( name: "Fifi", isMale: false, age: 12));
        System.out.println(d.getData().getName() + " " + d.getData().getAge());

        /* Two different types of containers even though it shares same implementat
           Cannot put a Person in a Dog container
           // d.setData(new Person("Fannie", 22)); // <-- Error!
        */

        GenericsExample<Person> p = new GenericsExample<Person>();
        p.setData(new Person( name: "Fannie", age: 22));
        System.out.println(p.getData().getName() + " " + p.getData().getAge());
    }
}
```

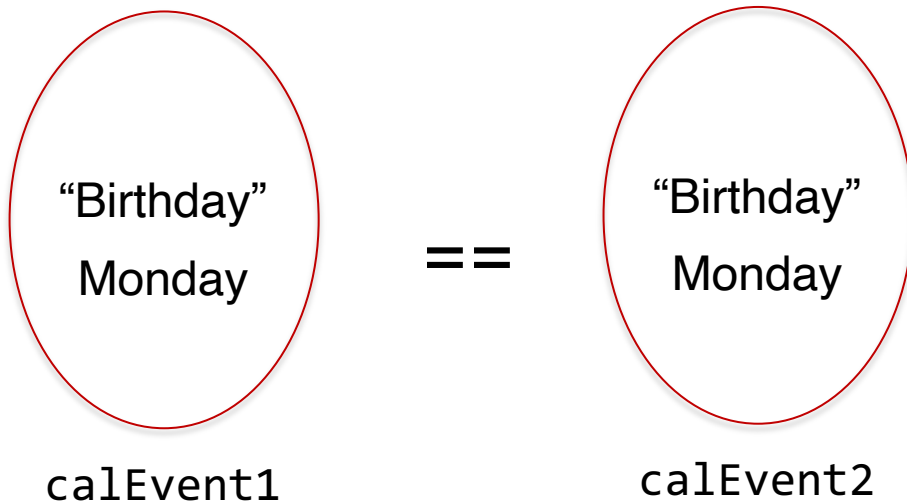
Typically

- T is used as a placeholder for Type
- E as a placeholder for Element
- Etc.

REVIEW: EQUALITY AND COMPARISON

- Remember objects have
 - Behavior
 - Attributes
 - State
 - Unique Identity
- When we compare objects, we might use any of the last three to determine what we mean when we say “equal to” in English

REVIEW: EQUALITY AND COMPARISON: ==

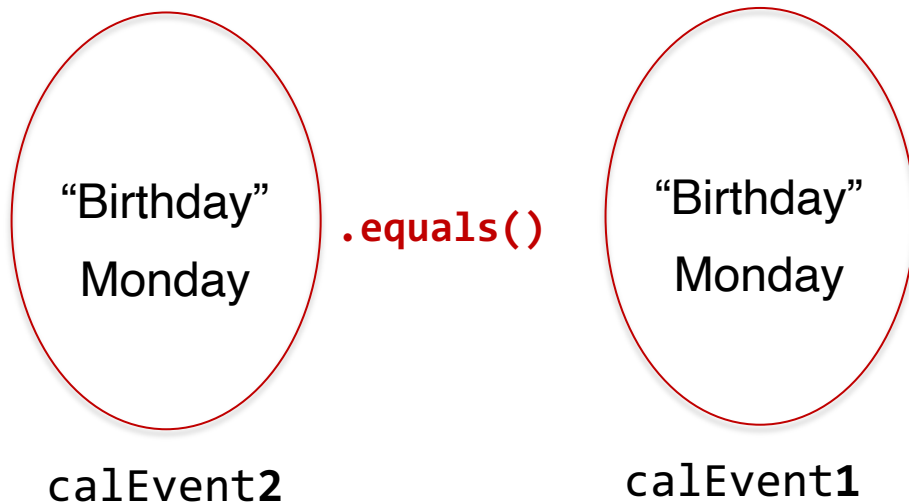


`==` works for basic types like `int`, but probably NOT what you intend for objects

`==` essentially is asking: "is this the same object", not
Are these equal

```
calEvent1 = new CalendarEvent(DAY.MONDAY, "Birthday");  
calEvent2 = new CalendarEvent(DAY.MONDAY, "Birthday");  
if (calEvent1 == calEvent2) // false
```

EQUALITY AND COMPARISON:



Equality (or sameness) in math should adhere to some basic properties that makes it rigorous and unambiguous. They are:

Reflexivity: A is equal to A (every object is equal to itself).

Symmetry: If A is equal to B, then B is equal to A.

Transitivity: If A is equal to B and B is equal to C, then A is equal to C.

```
calEvent1 = new CalendarEvent(DAY.MONDAY, "Birthday");
calEvent2 = new CalendarEvent(DAY.MONDAY, "Birthday");
calEvent1.equals(calEvent1) // reflexivity
(calEvent1.equals(calEvent2) == (calEvent2.equals(calEvent1)) // symmetry
```


HOW TO PREPARE FOR THE MIDTERM

- Review lecture notes
 - Modules on Canvas, your personal notes
- Review homework assignments & Labs
 - Make sure you understand how you got to the key parts of your solution (e.g. implementing Chesspiece interface, refactoring Paycheck etc.)
- Scan through code already written by you
- Work smaller problems for practice

TAKING THE MIDTERM

- Breathe!
 - You've got this – you can do it!
- If something is unclear, ask for clarification
- Only resources allowed: Your excellent brain and IntelliJ
 - No phone-a-friend, closed book, closed notes, no internet (other than for Canvas), no external sites (Stack Overflow, etc.)
- Time is money – if you get stuck, move to the next question and then circle back
 - You don't want to run out of time before trying all of the problems

AFTER THE MIDTERM

- Breathe!
 - You earned a breather!
- It's only halftime, no matter how you think you did – enjoy your Spring Break and come back ready to keep grindin'! 😊

FINAL WORDS FROM YOUR POSSE

“Woof!”

(Roughly translated: “You can do this!”)



Q & A
