CS 5004

# LECTURE 6

# REVIEW OF HOF & GENERICS; RECURSIVE DATA STRUCTURES

KEITH BAGLEY

SPRING 2022

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 Check-in
- Review of Java Collections & HOFs
- Recursive Data Structures - Lists!
- Accumulators & Tail Recursion
- Examination: IShapeList Accumulator
- Breakout: Accumulators
- Filtering & Sorting (Our version of Higher Order Functions)
- Open Q & A

# CHECK-IN

- Thank you!

# CHECK-IN

- Coaching Self-Eval
- We’re still on-pace to cover the required material
  - 1 less lab than originally planned
  - Perhaps 1 less homework
  - Will likely drop the code walks (as discussed first week of class)
    - Our labs are a bit different than Online & Vancouver so you get the “collaborative experience” more than they do on a weekly basis anyway

# REVIEW: COMPARETO()

- The compareTo() method for comparing ordering sequence of objects. We get this method from Object (like toString()) and can override it if we want
  - Less than, equals, greater than
  - CompareTo method must return negative number if current object is less than other object, positive number if current object is greater than other object and zero if both objects are equal to each other.
  - CompareTo must be in consistent with equals method e.g. if two objects are equal via equals() , there compareTo() must return zero

# REVIEW: COMPARETO()

- Example from Module 3 code from Amit & John
- Allows us to “order” objects sequentially based on the criteria WE set.

```
/**
 * Created by ashesh on 1/26/2017.
 */
public abstract class AbstractShape implements Shape {
  protected Point2D reference;

  public AbstractShape(Point2D reference) {
    this.reference = reference;
  }

  @Override
  public double distanceFromOrigin() {
    return reference.distToOrigin();
  }


  @Override
  public int compareTo(Shape s) {
    double areaThis = this.area();
    double areaOther = s.area();

    if (areaThis < areaOther) {
      return -1;
    } else if (areaOther < areaThis) {
      return 1;
    } else {
      return 0;
    }
  }
}
```

# REVIEW: COMPARATORS

- In a similar approach as compareTo, we can create general-purpose comparators to help us sort elements in a Collection
- Comparators realize the functional interface
  - The functional interface allows us to create classes that have a single method and use them as a “function object”
  - In this case, comparators implement one method: compare()
- A comparator is a separate class that decouples comparison algorithms from the class(es) they are used on.
  - Composition and delegation to allow for flexibility. Allows us to specify different comparison strategies based on need.

# REVIEW: COMPARATORS

```
1  import java.util.Comparator;
2  class ShapeSizeComparator implements Comparator<IShape> {
3    public int compare(IShape shape1, IShape shape2) {
4      if (shape1.getArea() == shape2.getArea())
5        return 0;
6      else if (shape1.getArea() > shape2.getArea())
7        return 1;
8      else
9        return -1;
10   }
11 }
```

What if we want to use the Perimeter rather than Area for the sort?
Here is the original based on Area

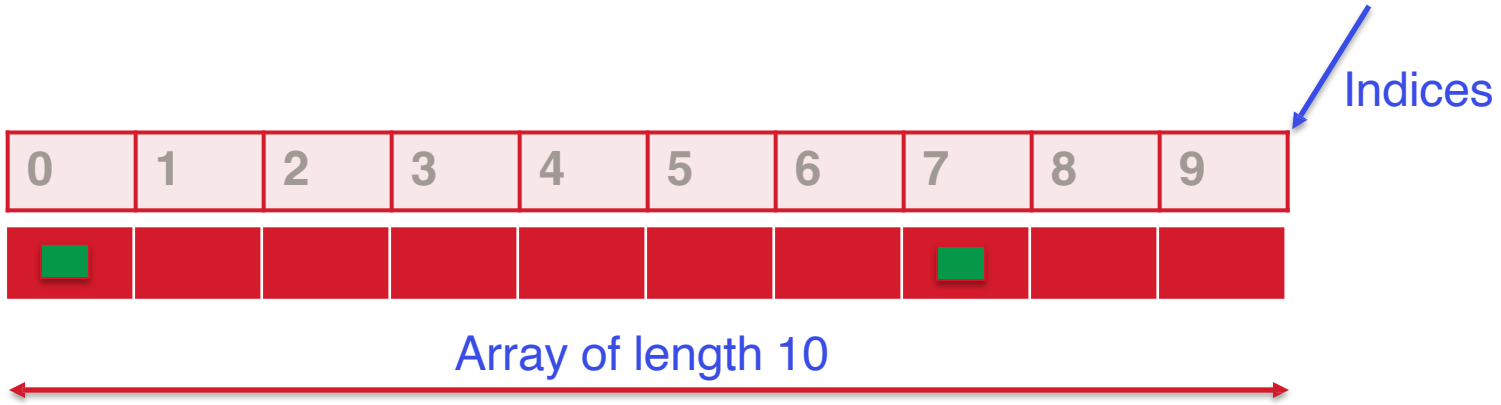
# REVIEW: COMPARATORS

```
3   import java.util.Comparator;
4   import shapes.IShape;
5
6   public class ShapePerimeterComparator implements Comparator<IShape> {
7   @   public int compare(IShape shape1, IShape shape2) {
8           // alternate approach
9           return Double.compare(shape1.getPerimeter(), shape2.getPerimeter());
10      }
11  }
```

Here is a second one based on Perimeter. Depending on the situation, we can select either one.

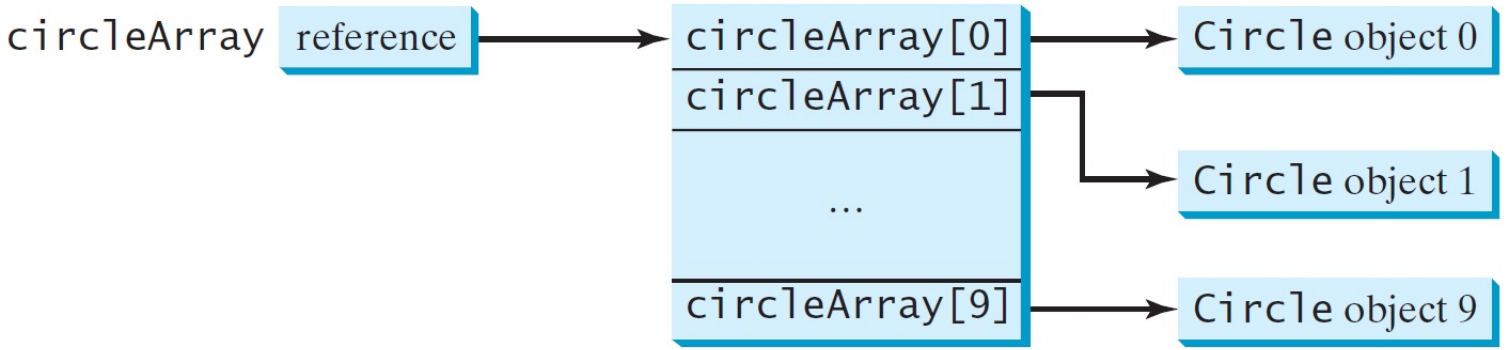
# REVIEW: ARRAYS IN JAVA

- Array - container object that holds a fixed number of values of a single type
  - The length of an array is established when the array is created, and it is fixed after creation
  - Items in an array are called elements, and each element is accessed by its numerical index



# REVIEW: ARRAY OF OBJECTS, CONT.

```
Circle[] circleArray = new Circle[10];
```

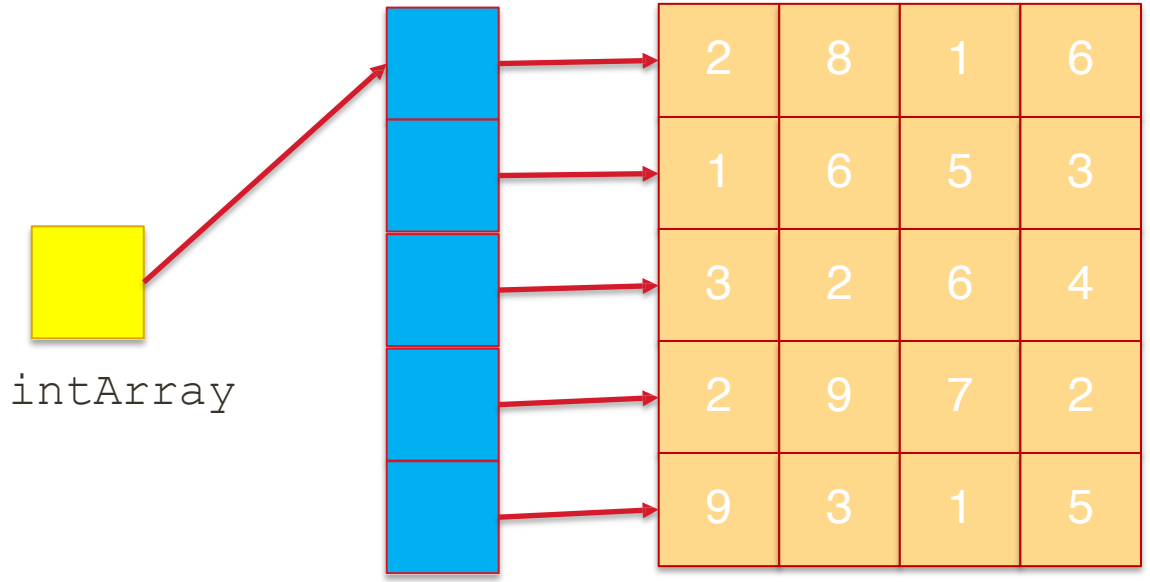



```
Circle [] circles = new Circle[10];
circles[0] = new Circle( radius: 2);
System.out.println(circles[0]);
```

# REVIEW: MULTIDIMENSIONAL ARRAYS

- Multidimensional arrays - arrays of arrays where each element of an array holds a reference to another array

```
int[][] intArray = new int[5][4]; //a 2D array or matrix
```



# REVIEW: MULTIDIMENSIONAL ARRAYS

- Multidimensional arrays (jagged arrays) - arrays of arrays where each element of an array holds a reference to another array

```
int[][] intArray = new int[5][4]; //a 2D array or
matrix
```

The above is really equivalent to a 3-step process:

```
// create the single reference intArray (yellow square)
int [][] intArray;

// create the array of references (blue squares)
intArray = new int[5][];

// create the second level of arrays (red squares)
for (int i=0; i < 5 ; i++)
    intArray[i] = new int[4]; // create arrays of integers
```

# REVIEW: PREDICATES

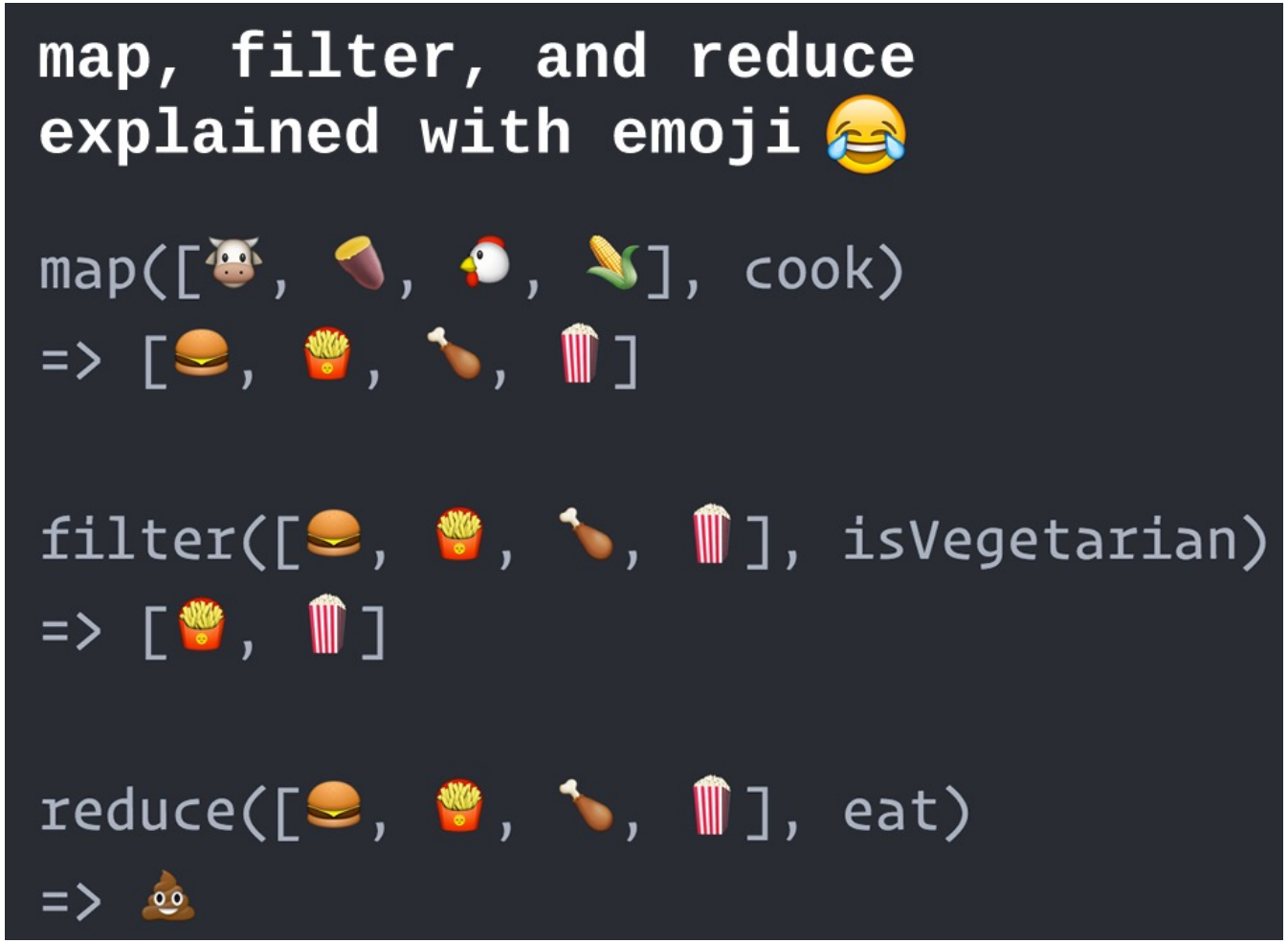
- A predicate is a function that returns true or false based on a 1-parameter input value
- We can use predicates (and comparators) in expressions for higher-order functions to help process data more flexibly and efficiently
  - More on HOF's a bit later today
- The **Functional Interface** PREDICATE is defined in the *java.util.function* package. The main method we need to implement is the test() method

# REVIEW: PREDICATES & LAMBDA

- Since Predicates are short test functions, we can often write them as lambda expressions rather than creating entire separate classes
  - Lambda expressions are “anonymous functions”
  - Used extensively in functional programming languages

```
List<IShape> result = shapeList.stream().filter(iShape -> iShape.getArea() < 20).collect(Collectors.toList());
```

# REVIEW: HIGHER ORDER FUNCTIONS




Content from: http://www.globalnerdy.com/wordpress/wp-content/uploads/2016/06/map-filter-reduce-in-emoji-1.png

Courtesy of Joey Devilla

# REVIEW: HIGHER ORDER FUNCTIONS

- ***Filter***: *Given a list, create a sublist of things in the original list that satisfy a given condition*. This *List => List* operation is called a **filter**. One may think of a filter as a generic operation on a list, and returns a sublist of the same type. It takes the general form
  - List filter(List input, Predicate p)
- ***Fold*** is of the form *List => value*. A *fold* operation starts with an initial value, an algorithm to combine this initial value with elements in the list and return the combination. Examples of fold include counting the number of books in the list, calculating the total price of all books, determining if the list contains a book by Rowling, etc.

# REVIEW: HIGHER ORDER FUNCTIONS

- ***Map***: A *map* operation takes in a list and returns a list of the same size, but possibly of a different type. The function that a *map* applies to each element of the list can be thought of as mapping the element to another element

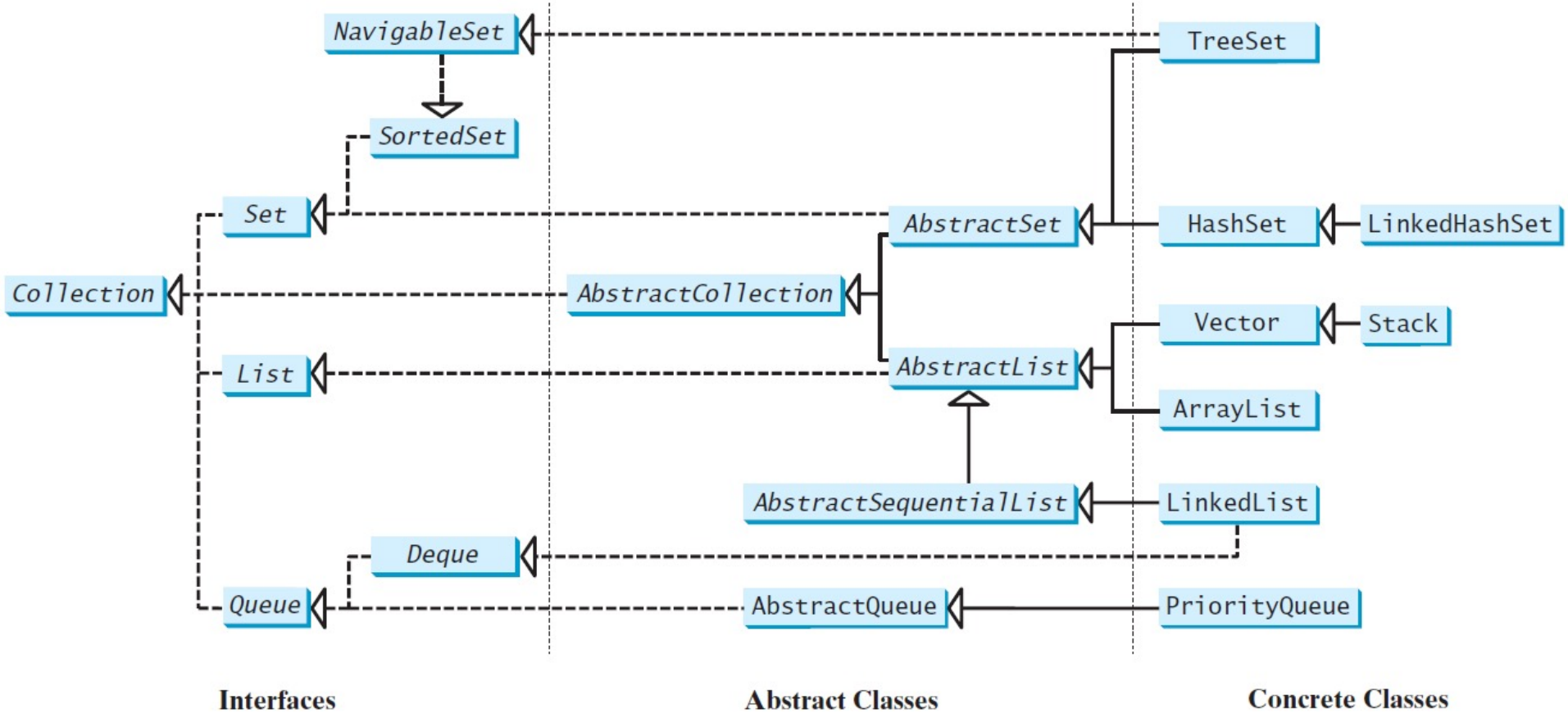
# HOF EXAMPLE: TOTAL THE AREAS OF OUR SHAPES

- We can map from Shapes to double values (Double)
  - Then use reduce (fold) to calculate a single value
  - We'll write this code later today after we cover lists!

```
System.out.println("---- Using List Map and Reduce to get a single value ----");

// map and reduce. First map the shapes 1-1 to their areas
List<Double> areas = shapeList.stream()
        .map(s->s.getArea())
        .collect(Collectors.toList());
// Now we have a list of areas (Double). Add them together to get a single value
Double total = areas.stream().reduce( identity: 0.0, (one, two) -> one + two);
System.out.println("Total =  " + total);
}
```

# REVIEW: JAVA COLLECTIONS API




[Picture credit: Liang 2020

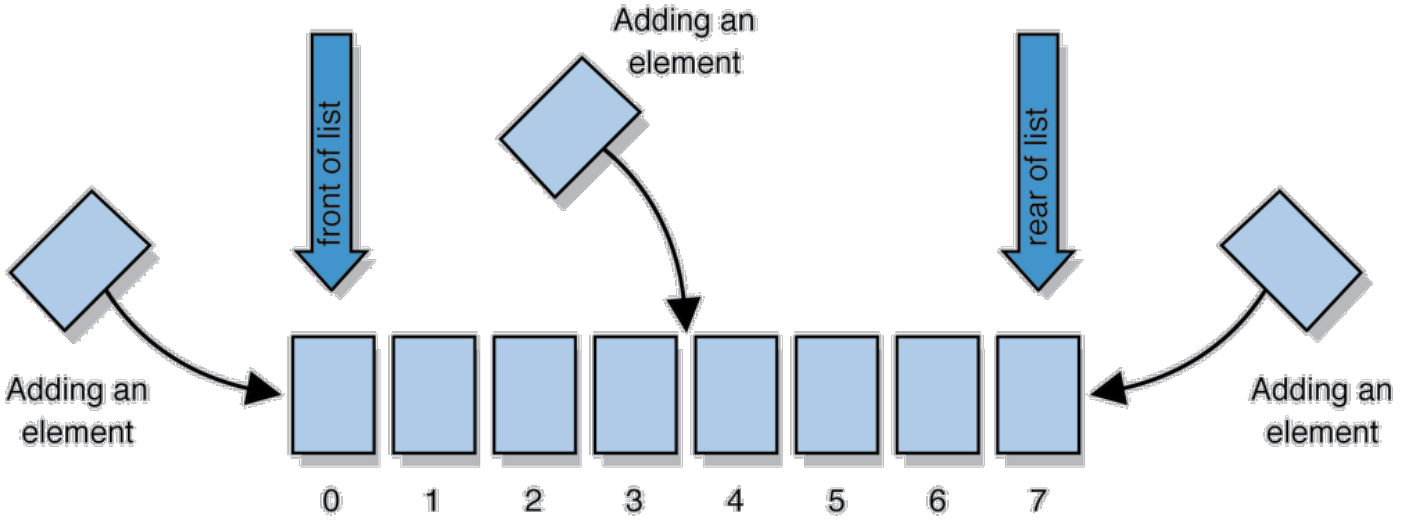
# JAVA COLLECTIONS FRAMEWORK

- 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>`

# LISTS

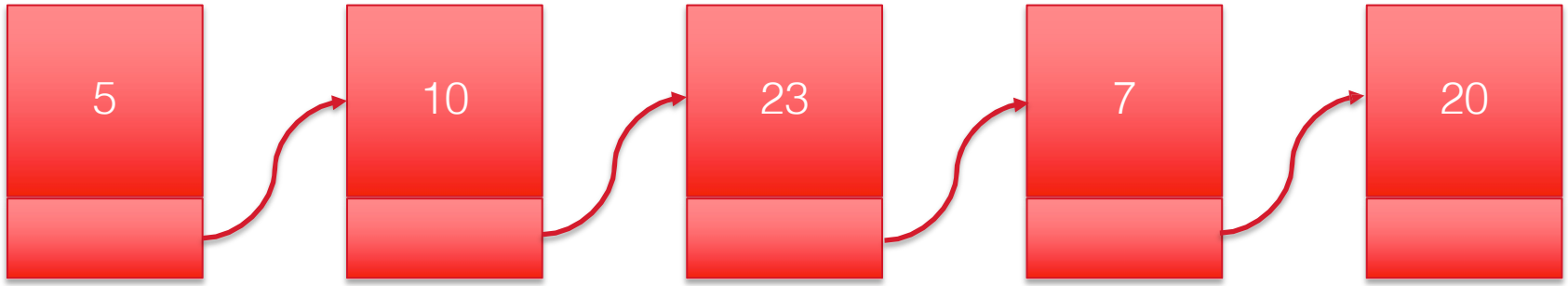
- List – an ordered collection (also known as a sequence)
  - A user controls where in the list each element is inserted
  - A user can access elements by their integer index (position in the list), and search for elements in the list
- Size - the number of elements in the list
- List allows for elements to be added to the front, to the back, or arbitrary



# LISTS AND LINKED DATA STRUCTURES

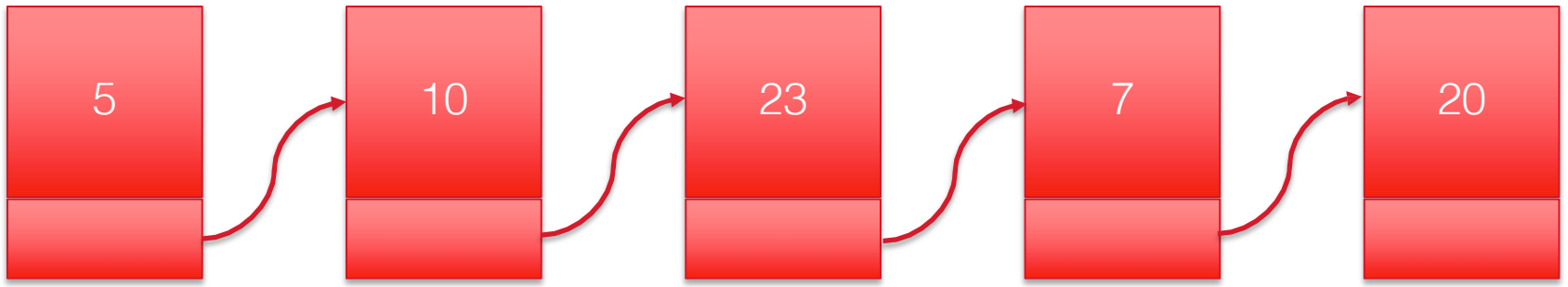
- Lists use one of the following underlying structures:
  - An array of elements


  - A collection of linked objects, each storing one element, and one or more references to other elements



# LINKED LIST

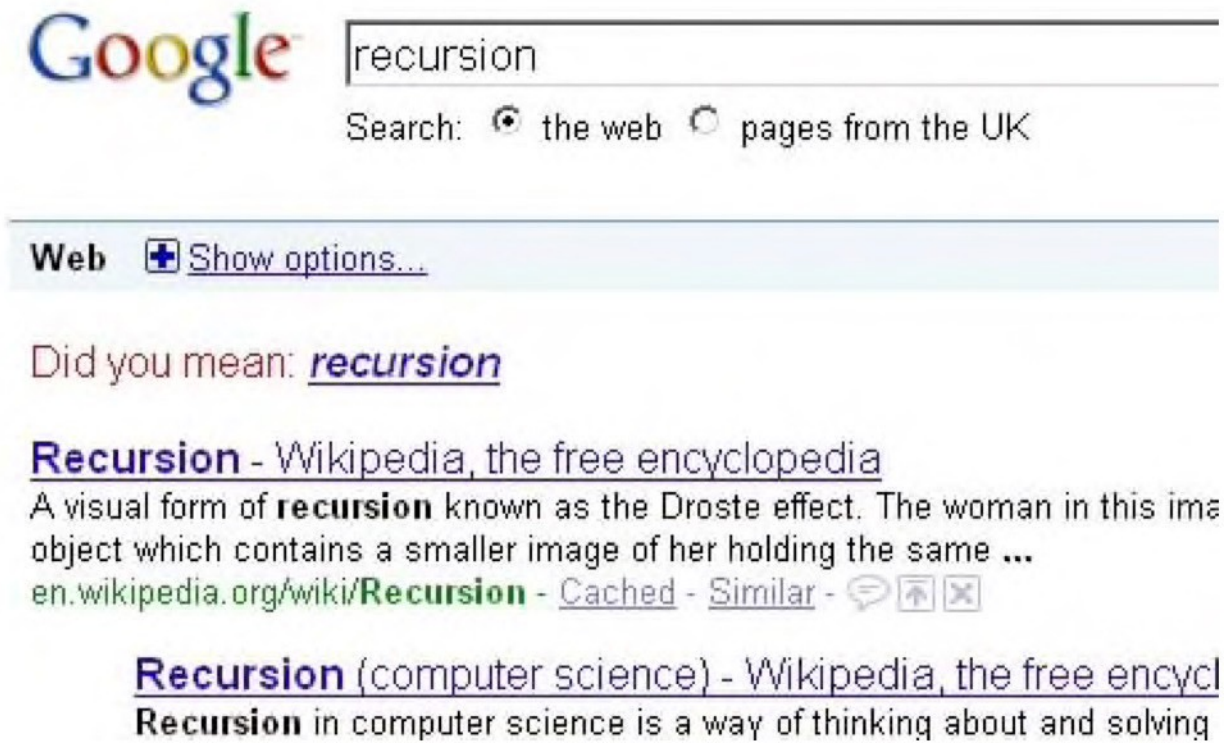
- A collection of linked objects, each storing one element, and one or more references to other elements



# WHAT IF WE ROLLED OUR OWN?

- First rule: D.R.Y.
  - We're not going to advocate creating your own collection classes, except when you can't find an existing collection that is suitable
    - E.g.: Certain graphs and trees might not be implemented
- This section is to fulfill the need for you to understand – conceptually – HOW things work "under the covers" so you have parity with your peers in direct-entry Programming Design Paradigms (CS5010)

# RECURSION




[Pictures credit: http://www.telegraph.co.uk/technology/google/6201814/Google-easter-eggs-15-best-hidden-jokes.html]

# RECURSION

- Recursion – an operation defined in terms of itself
- Solving a problem recursively means solving smaller occurrences of the same problem
- Recursive programming – an object consist of methods that call themselves to solve some problem

- Can you think of some examples of recursions and recursive programs?

# RECURSIVE ALGORITHMS

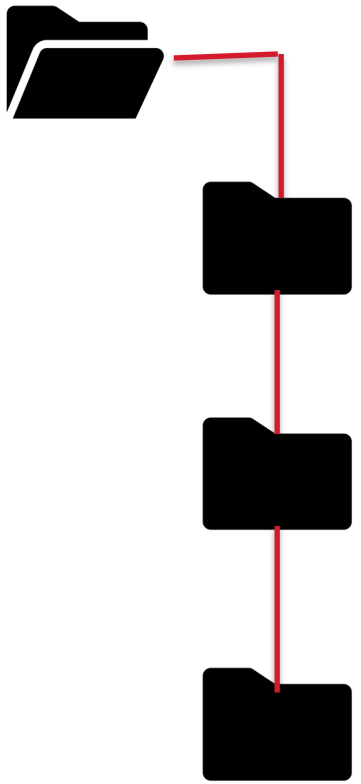
- Every recursive algorithm consists of:
  - Base case – at least one simple occurrence of the problem that can be answered directly
  - Recursive case - more complex occurrence that cannot be directly answered, but can be described in terms of smaller occurrences of the same problem

- A crucial part of recursive programming is identifying these cases

# RECURSIVE DATA STRUCTURES

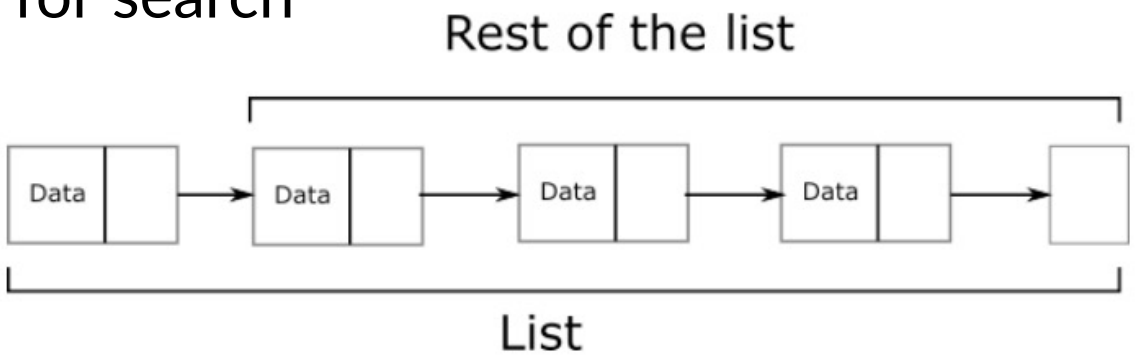
- We can represent some complex forms of data recursively such that an element in our structure refers to another element of the same type, and so forth

- Just like recursive functions, recursive structures can be thought of as having a:
  - Base case
  - Recursive case



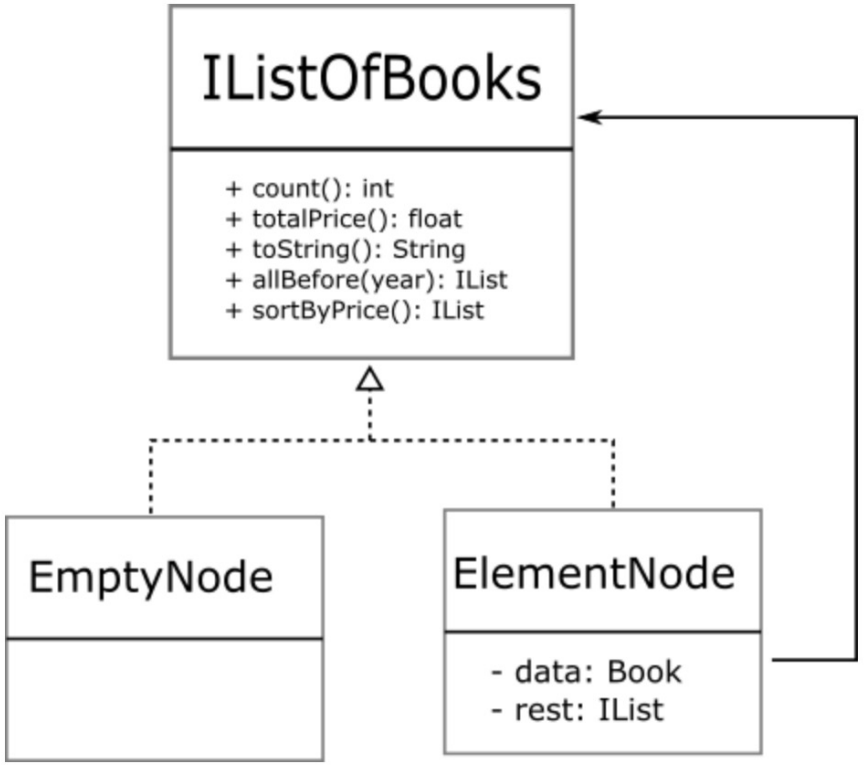
# LINKED LISTS

- You have now used the Java List<>, and you are familiar with the concept of lists from other programming languages
- There are several ways of implementing a list; one approach is to create what is called a linked list
- A linked list is made of individual nodes. A node in the list contains the data (in our case, the book) and points to the next node in the list (the link).
  - The last node in such a list is often empty, signifying the end of the list
- Linked lists are sequential and O(n) for search



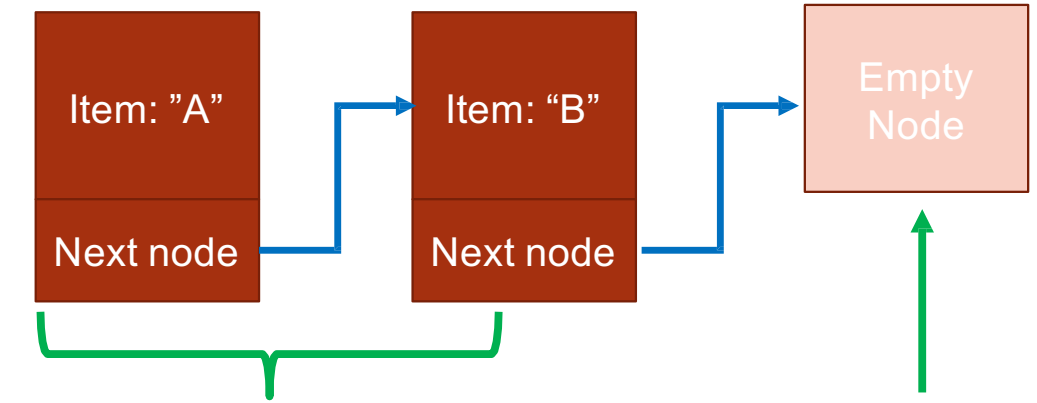
# LINKED LISTS

- Since a linked list is a self-referential data structure, a list can be thought of as a single node followed by “the rest of the list” (previous slide).
- An empty list consists simply of an empty node.
- A list can implement a protocol that handles that variety. A list can be (a) an empty node or (b) a node with an element of data and another list



# LINKED LISTS

- Linking Nodes




**Recursive case**
Non-empty list

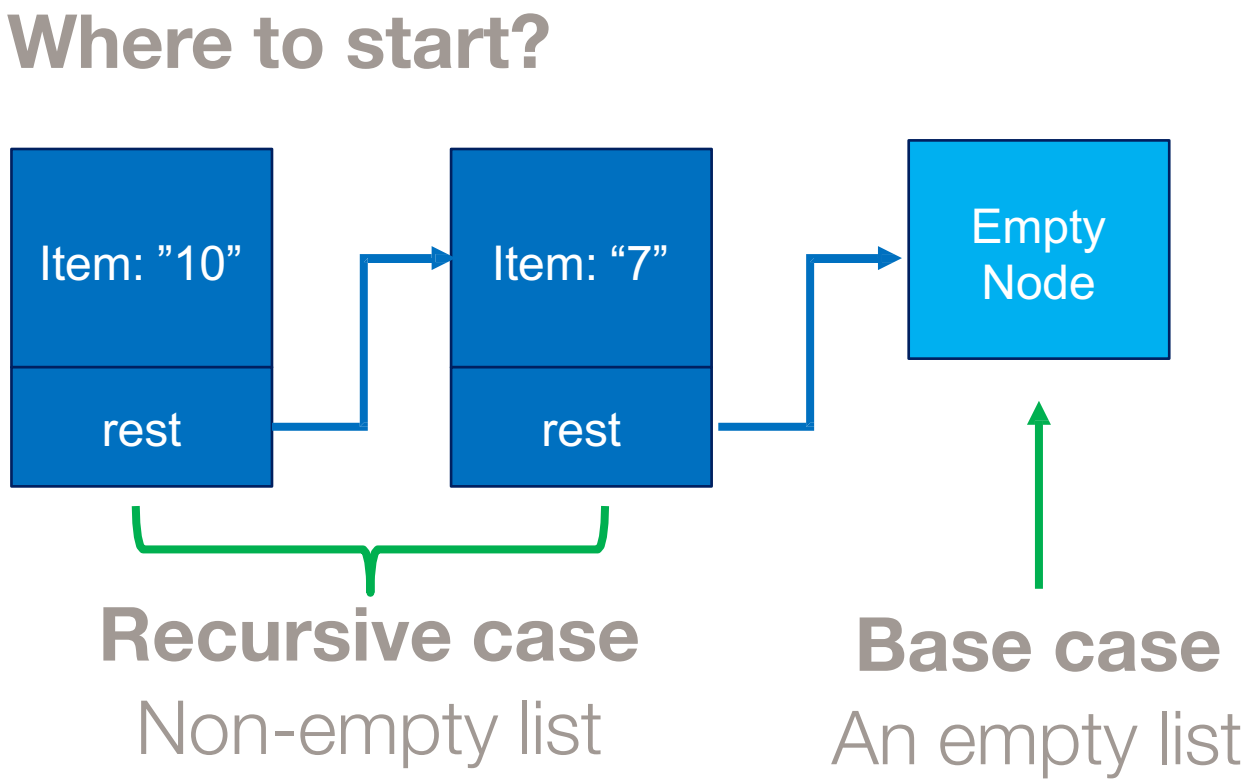
**Base case**
An empty list

```
public class ElementNode {
   private DataType item;
   private ElementNode rest;

   public ElementNode(DataType item, next) {
      ElementNode this.item =item;
      this.rest =next;
   }
   // getters, setters, etc
}
```

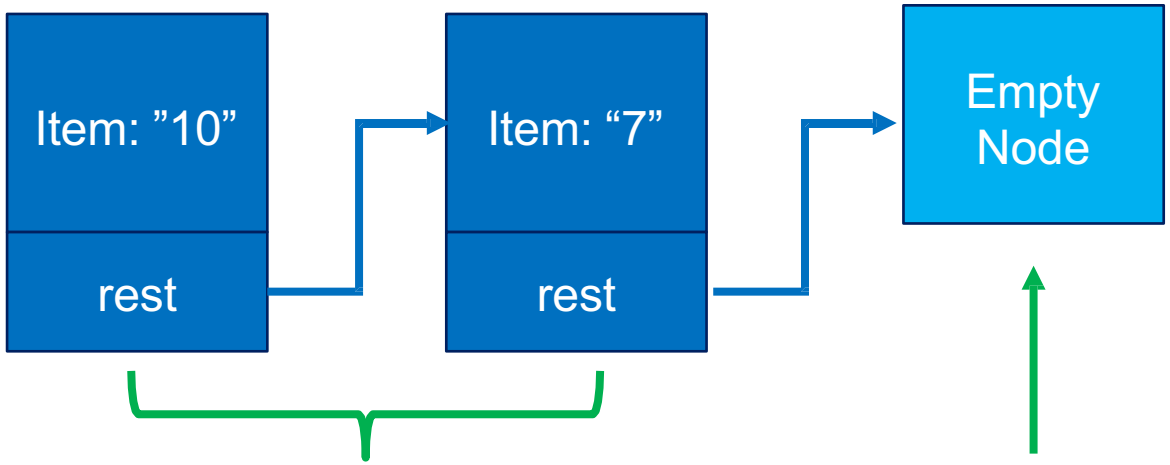
*Linked list is always a recursive structure but methods may/may not use recursion

# EXAMPLE: RECURSIVE LINKED LIST OF INTEGERS: COUNT()



# RECURSIVE LINKED LIST: COUNT()

## Where to start?




**Recursive case**

Non-empty list

**Base case**

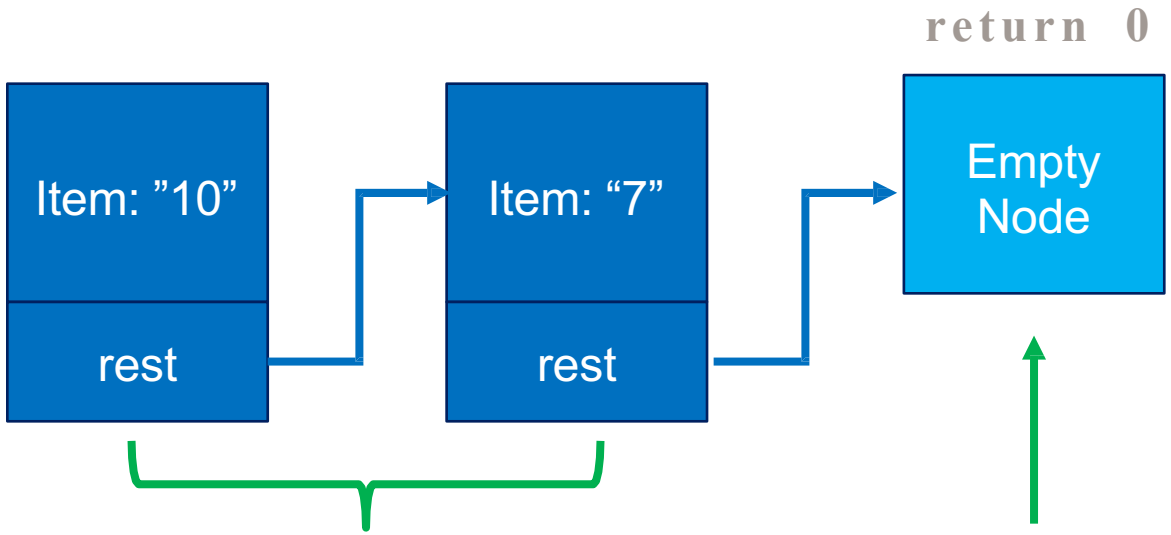
An empty list

Just like a recursive function, **start with the base case**

- What should `count()` do if the list is empty?

# RECURSIVE LINKED LIST: COUNT()

## Where to start?

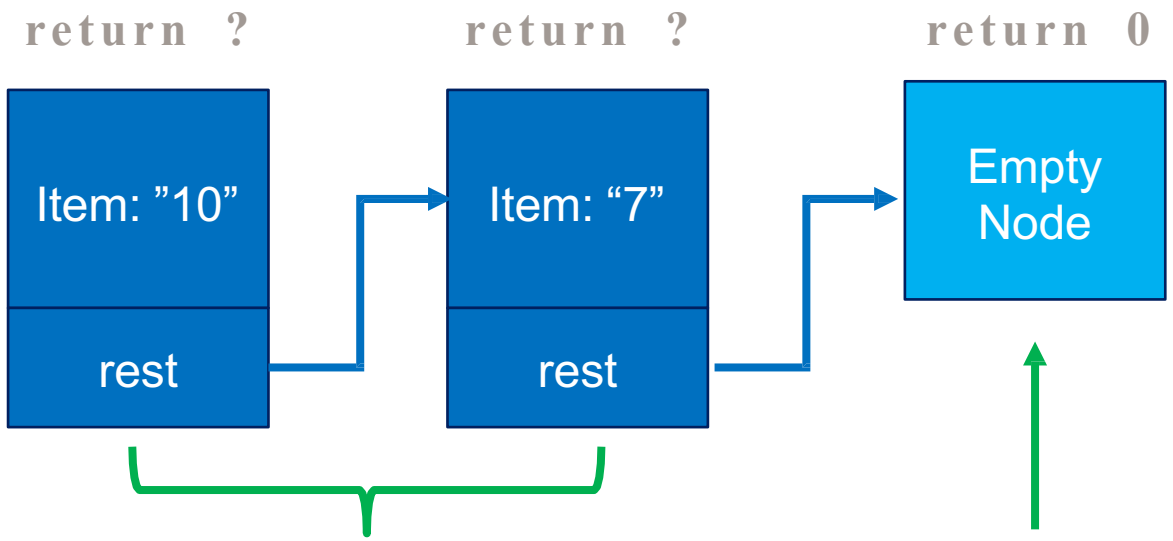



Just like a recursive function, **start with the base case**

- What should `count()` do if the list is empty?
  - An empty list has no items
  - → return 0

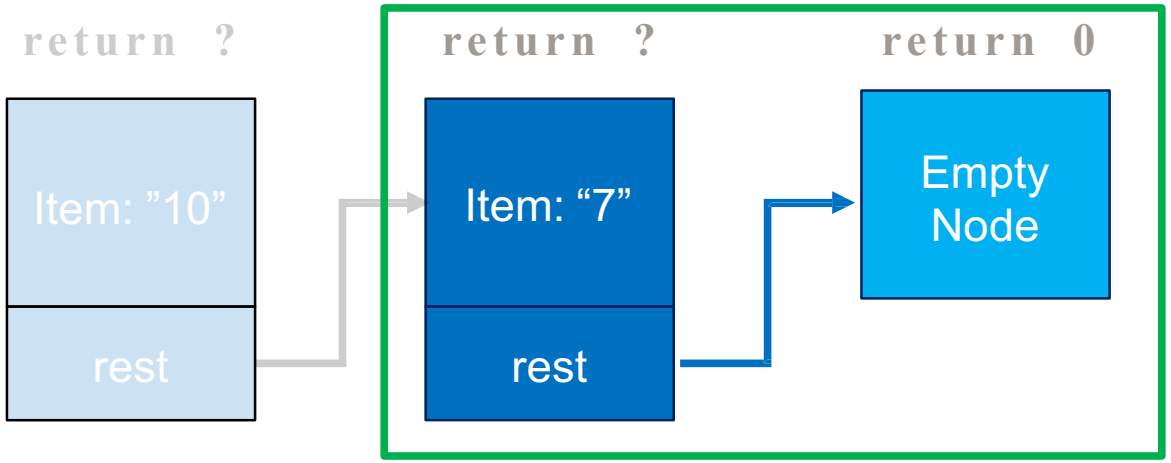
# RECURSIVE LINKED LIST: COUNT()

## What about the recursive case?



# RECURSIVE LINKED LIST: COUNT()

## What about the recursive case?




Think about the next simplest case, a list of 1.

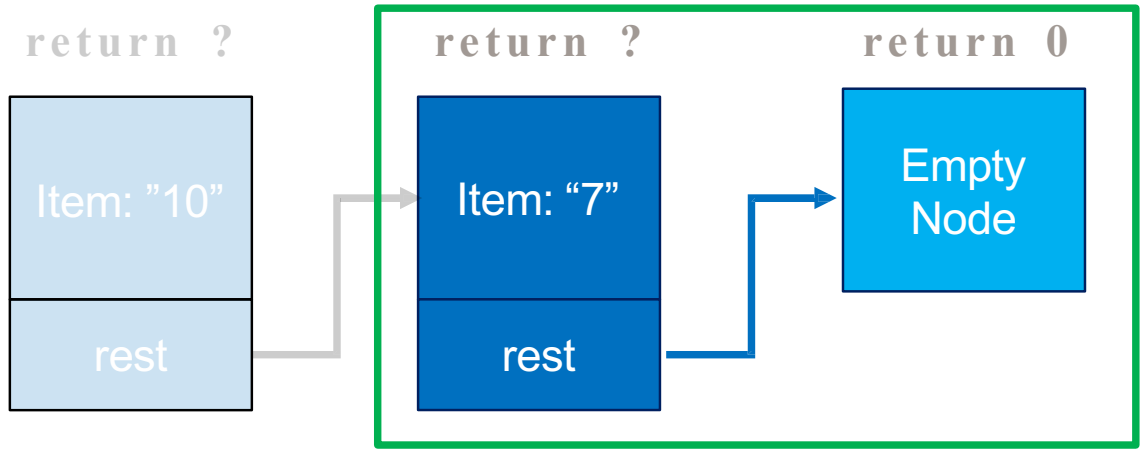
What we know:

- `this.rest.count()` is 0

The size of the list is 1 + the size of the rest of the list

# RECURSIVE LINKED LIST: COUNT()

## What about the recursive case?




Think about the next simplest case, a list of 1.

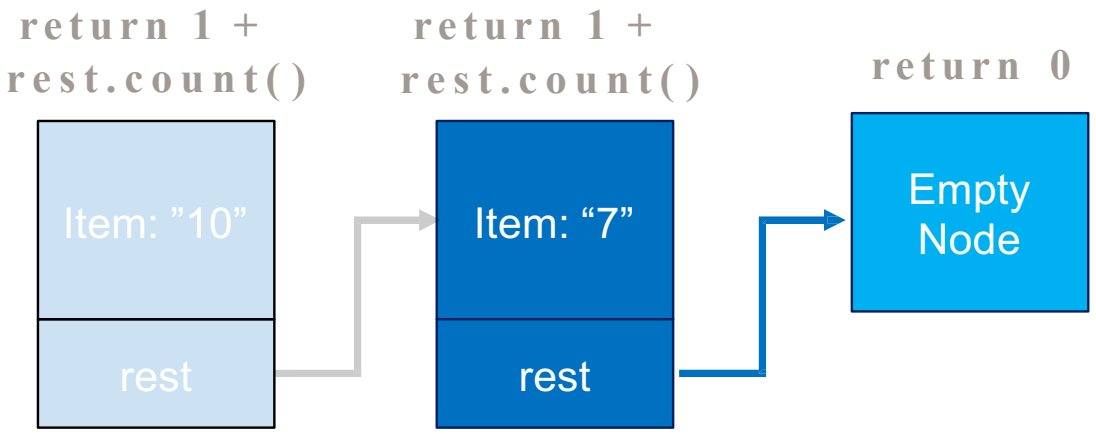
What we know:

- `this.rest.count()` is 0

So, `this.count()` should return...

`1 + this.rest.count()`

# RECURSIVE LINKED LIST: COUNT()




Think about the next simplest case, a list of 1.

What we know:

- `this.rest.count()` is 0

So, `this.count()` should return...

`1 + this.rest.count()`

# PROCESSING: TRADITIONAL VS. ACCUMULATOR

- A traditional recursive approach on our recursive structure “unravels” back-to-front
  - In other words, we wait to “add” (or process) the current node information until the return from the other, subsequent recursive calls
- Accumulator approaches process information before the recursive call is made, and does nothing after it returns
  - Called “tail-recursive”

# WHY DO YOU CARE?

- With recursion, sometimes an accumulator approach can help us deal with situations that we'd be tempted to do "Bad Things™"
  - E.g. using "type fields", instanceof, getClassName(), etc.

```
@Override
public String toString() {
  if (rest instanceof EmptyNode) {
    return (this.contents + "." + rest.toString());
  }
  if (rest instanceof PunctuationNode) {
    return (this.contents + rest.toString());
  }
  else {
    return (this.contents + " " + rest.toString());
  }
```

Traditional recursive approach. Make the call, wait to process until the return.

# ACCUMULATOR

- Approach
  - Create a helper function that takes an accumulator
  - Old base case's return value becomes initial accumulator value
- Final accumulator value becomes new base case return value

# ACCUMULATOR

- Tail recursion can assist with optimization & avoiding “out of stack space for recursive calls” dilemma.
  - Accumulator is a pattern that can assist when decision-making requires knowledge of state NOW and we are unable to “see” what’s coming in future recursive calls

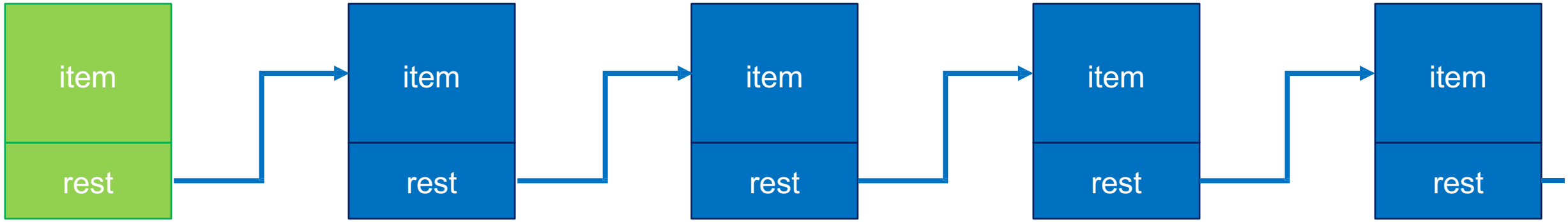
```
@Override
public String toStringHelper(String acc) {
  return rest.toStringHelper(acc + " " + this.contents);
}

/**
 * returns string of a sentence
 *
 * @return string of a sentence
 */
public String toString() {
  return rest.toStringHelper(this.contents);
}
```

Accumulator (tail recursion)
Process in the current node first (as part of the parameter value), THEN pass down the chain

Don’t need to know what’s next;
We make decisions based on what was given to us as the parameter

# INSERT(INTEGER): ILINKEDLIST




Create a new ElementNode containing the Integer, put it at the beginning

# INSERT(INTEGER): ILINKEDLIST

- Insert at the head of the list so doesn't need to be recursive
- BUT, still need to tackle insert for both node types
  - Head is list with contents > ElementNode
  - Head is empty list > Empty Node

# INSERT(INTEGER): ILINKEDLIST

## EmptyNode.java

```
public ILinkedList insert(Integer item)
{
   return new ElementNode(item, this);
}
```

# INSERT(INTEGER): ILINKEDLIST

## EmptyNode.java

```
public ILinkedList insert(Integer item)
{
  return new ElementNode(item, this);
}
```

## ElementNode.java

```
public ILinkedList insert(Integer item)
{
  return new ElementNode(item, this);
}
```

# INSERT(INTEGER): ILINKEDLIST

## EmptyNode.java

```
public ILinkedList insert(Integer item)
{
   return new ElementNode(item, this);
}
```

## ElementNode.java

```
public ILinkedList insert(Integer item)
{
   return new ElementNode(item, this);
}
```

**this** represents the current "head" of the List

# HOW DOES JAVA KNOW WHICH VERSION TO CALL?

## Dynamic dispatch

- If the list that calls insert is an **ElementNode**, Java will call the **ElementNode insert** implementation
- If the list that calls insert is an **EmptyNode**, Java will call the **EmptyNode insert** implementation

# GENERICS

# GENERICS

- 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
- 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
- We’ll cover this more next time!

# GENERICS

```java
// 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.