CS 5004: OBJECT ORIENTED DESIGN AND ANALYSIS

SPRING 2022

LECTURE 6

Divya Chaudhary
d.chaudhary@northeastern.edu

Northeastern University
Khoury College of
Computer Sciences

AGENDA

- Midterm review
- Review
  - Subtype polymorphism
  - Ad hoc polymorphism
  - Static and dynamic binding
- Recursive data structures
- Recursive linked list
- Stack implementation using a recursive linked list
- Immutable stack implementation using a recursive linked list

MIDTERM LOGISTICS AND REVIEW

CS 5004, SPRING 2022– LECTURE 6

MIDTERM LOGISTICS

- Midterm:
  - Opens next Tuesday, March 8th, 2022- 10pm
  - Closes on Saturday, March 12th, 2022- 11:59pm
  - You can choose to take midterm any time during that week
- Remote computer-based exam
  - Multiple choice questions on Canvas

  - We have next lecture on March 8th.
- No lab next week

MIDTERM LOGISTICS

- Remote computer-based exam:
  - Please use your own computers to individually work on the exam problems
  - Please submit your solutions by pushing them to your individual repos in our GitHub course organization
  - Please use the same Gradle project as the one we using throughout this course
  - Please follow code organization and naming convention similar to the conventions we followed in the course

MIDTERM TOPICS

- Types of data
- Classes and objects
- Documentation
- Writing methods
- Unit testing
- Exceptions
- Enumerations and switch statements
- Inheritance
- Interfaces and abstract classes
- Abstract Data Type
- Lists
- Arrays
- Recursive data structures – not emphasized

EXPECTATIONS – CHECK THE EXAM DOCUMENT

- Javadoc – include only if specifically asked for
  - Short description
  - `@param, @return @throws`

- `equals(), hashCode(), toString()`
  - Don’t need to write (unless specifically asked for)

EXPECTATIONS – CHECK THE EXAM DOCUMENT

- UML diagrams – only provide if explicitly required

- Pay attention to:
  - Modifiers for fields and methods (public, private, protected)
  - Types of relationships between objects (arrows)

- Don't include in UMLs:
  - Constructors
  - Getters
  - constants

EXPECTATIONS – CHECK THE EXAM DOCUMENT

- Exceptions

- No need to implement custom exceptions - you can assume the implementation exists
  - e.g., throw new `SpecialException();`
  - We don't expect you to write out the `SpecialException` class
- It should be clear when the exception is thrown

EXAM FORMAT

- Focus on:
  - Reading and understanding Java code
  - Documenting and testing Java code
  - Object-oriented design

GENERAL TEST-TAKING TIPS

- Read the questions before starting
- Take time to think and plan your approach before jumping in

TIPS FOR SUCCESS IN THIS EXAM

CS5004 Object-Oriented Design

not

CS5004 Programming in Java

TIPS FOR SUCCESS IN THIS EXAM

CS5004 **Object-Oriented Design – follow OOD principles**

not

CS5004 Programming in Java

TIPS FOR SUCCESS IN THIS EXAM

Apply OOD principles:

- Encapsulation
- Abstraction
- Information hiding
- Polymorphism
- Inheritance

Solutions that do not use OOD will earn negligible points, even if functionally correct

TIPS FOR SUCCESS IN THIS EXAM

Apply OOD principles:

- Encapsulation
- Abstraction
- Information hiding
- Polymorphism
- Inheritance

In practical terms:

- Use inheritance
  - **interfaces**
  - abstract classes
- Plan before doing
- The OOD solution is not always the easiest one to write
  - Often the easiest one to use

TIPS FOR SUCCESS IN THIS EXAM

Practice applying OOD principles

- Code to an interface
- Use inheritance
- If you haven’t applied or aren’t comfortable with a specific concept, find a way to practice it
  - Rewrite older assignments to incorporate the concept

REVIEW

CS 5004, SPRING 2022– LECTURE 6

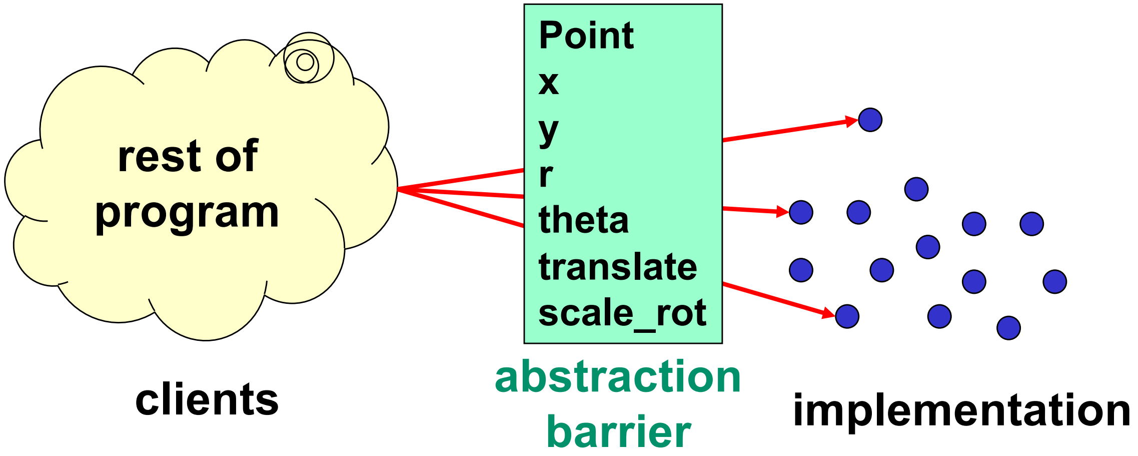
REVIEW: ABSTRACT DATA TYPE

- Abstract Data Type (ADT) - model that describes data by specifying the operations that we can perform on them

- Clients care about the ADT

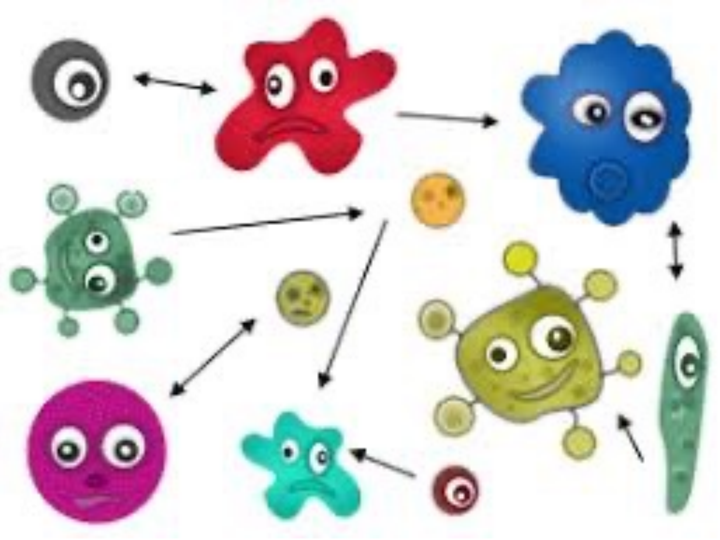
- For each operation, we describe:
  - The expected inputs, and any conditions that need to hold for our inputs and/or our ADT
  - The expected outputs and any conditions that need to hold for our output and/or our ADT
  - Invariants about our ADT

REVIEW: ADT = OBJECT + OPERATIONS



- Implementation is hidden
- The only operations on objects of the type are those provided by the abstraction

REVIEW: POLYMORPHISM



[Pictures credit: http://www.thewindowsclub.com/polymorphic-virus]

Polymorphism – the ability to define different classes and methods as having the same name but taking different data types

REVIEW: POLYMORPHISM

Polymorphism – the ability to define different classes and methods as having the same name but taking different data types

The ability of one **object** to be viewed/used as different **types**.

- Object = an *instance* of a *class* (i.e. a variable)

- Type = a *data type*
  - A **`class`** name
  - An **`abstract class`** name
  - An **`interface`** name

REVIEW: SUBTYPE POLYMORPHISM

Made possible by **inheritance**.

- Every object will have multiple types
- An object is an **`instanceof`** its runtime type
- An object is an **`instanceof`** every type its runtime type inherits from

REVIEW: COMPILE TIME AND RUN-TIME

- Java programs have two distinct phases in their lifetimes:
  - Compile time (static time) - refers to source code, and the point in time when the source code is being compiled by the Java compiler (think of a compiler as a translator)
  - Run time (dynamic time) – refers to when the code is being evaluated (or executed or run)by the Java Virtual Machine (JVM)

REVIEW: COMPILE TIME AND RUN-TIME DATA TYPE

```
Person emily = new Person();
Singer adele = new Singer();
Person flora = new Singer();
```

- Static (compile time) type – the declared type of a reference variable. Used by a compiler to check syntax

- Dynamic (run time) type – the type of an object that the reference variable currently refers to (it can change as the program execution progresses)

REVIEW: STATIC BINDING

- References have a type
  - (they refer to instances of a particular Java class)

- Objects have a type
  - Instances of a particular Java class
  - Instances of all of their super-class

- Static binding done by the compiler (when it can determine the type of an object)
- Method calls are bound to their implementation during the compilation

REVIEW: DYNAMIC BINDING

- Achieved at runtime
  - Data type of an object cannot be determined at compile time
  - JVM (not the compiler) binds a method call to its implementation

- Instances of a sub-class can be treated as if they were an instance of the parent class
  - Therefore the compiler doesn't know its type, just its base type

REVIEW: DYNAMIC BINDING

- Whenever a reference refers to an interface or a base class, methods are dynamically bound
  - Method implementation determined at runtime

- Polymorphism and dynamic binding are inter-connected, and represent a powerful feature of OO design

- Allow the creation of “frameworks”
  - Applications that are implemented around interfaces, but are customised by plugging in different implementations of those interfaces
  - Very extensible

REVIEW: CASTING

- Casting - a Java language feature that allows us to alter the compile-time type of a variable
- The runtime type is not altered because of a cast
- We can explicitly cast to a compile-time type using (T) o
- We can also implicitly cast using subtype polymorphism

- Types of casts:
- Upcasting - when we cast from a subclass to a superclass (or interface) (since we are moving up in the class hierarchy)
- Downcasting – when we cast from a superclass (or interface) to a subclass (every time we are moving down the class hierarchy)
  - Down casts are dangerous
  - We have to write code to ensure that our down cast is safe

REVIEW: OVERLOADING AND AD HOC POLYMORPHISM

- Overloading allows us to create methods that share the same method name but differ in their signature

- Ad hoc polymorphism – another name for function and operator overloading

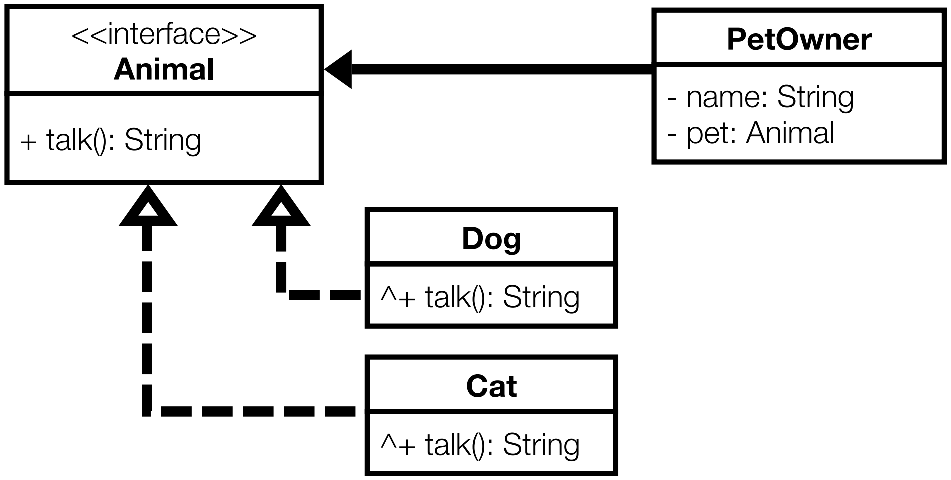
- Ad hoc polymorphism – a type of polymorphism where a polymorphic functions can be applied to arguments of different types
  - Polymorphic (overloaded) function can denote a number of distinct and potentially heterogeneous implementations, depending on the type of argument(s) to which it is applied

OVERLOADING AND AD HOC POLYMORPHISM

- Overloading allows us to create methods that share the same method name but differ in their signature
- Ad hoc polymorphism – another name for function and operator overloading
- With ad hoc polymorphism, it is a compiler or an interpreter binds (dispatches) methods - ensures that the right method is called

REVIEW: SUBTYPE POLYMORPHISM - EXAMPLE

```
Cat cat; Dog dog;

dog instanceof Dog
cat instanceof Cat
dog instanceof Animal
cat instanceof Animal
```



REVIEW: SUBTYPE POLYMORPHISM - EXAMPLE

```
public PetOwner(String name, Animal pet) {
  this.name = name;
  this.pet = pet;
}
```

We can do this because:

`Cat` is a **subtype** of `Animal`

```
PetOwner owner = new PetOwner(“Darth Vader”, new Cat(“Mittens”));

owner.getPet().talk();
```

REVIEW: SUBTYPE POLYMORPHISM - EXAMPLE

```
public PetOwner(String name, Animal pet) {
  this.name = name;
  this.pet = pet;
}

PetOwner owner = new PetOwner(“Darth Vader”, new Cat(“Mittens”));

owner.getPet().talk();
```

An example of **dynamic dispatch**.

- Won’t know which implementation of talk() until runtime.

REVIEW: SUBTYPE POLYMORPHISM - EXAMPLE

Equals method takes **any Object** as the parameter.

- All Java classes inherit Object therefore, all are `instanceof` `Object`

```java
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Node node = (Node) o;
    return Objects.equals(getItem(), node.getItem()) &&
            Objects.equals(getNextNode(), node.getNextNode());
}
```

REVIEW: SUBTYPE POLYMORPHISM - EXAMPLE

While an object is being viewed as a base/super class, **can't access subclass functionality**.

- **Cast** to get access to that functionality

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Node node = (Node) o;
    return Objects.equals(getItem(), node.getItem()) &&
            Objects.equals(getNextNode(), node.getNextNode());
}
```

REVIEW: SUBTYPE POLYMORPHISM - EXAMPLE

Without the cast:

- compile time error
- class Object has no methods getItem or getNextNode.

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    ~~Node node = (Node) o;~~
    return Objects.equals(getItem(), node.getItem()) &&
            Objects.equals(getNextNode(), node.getNextNode());
}
```

MAIN METHOD

CS 5004, SPRING 2022– LECTURE 6

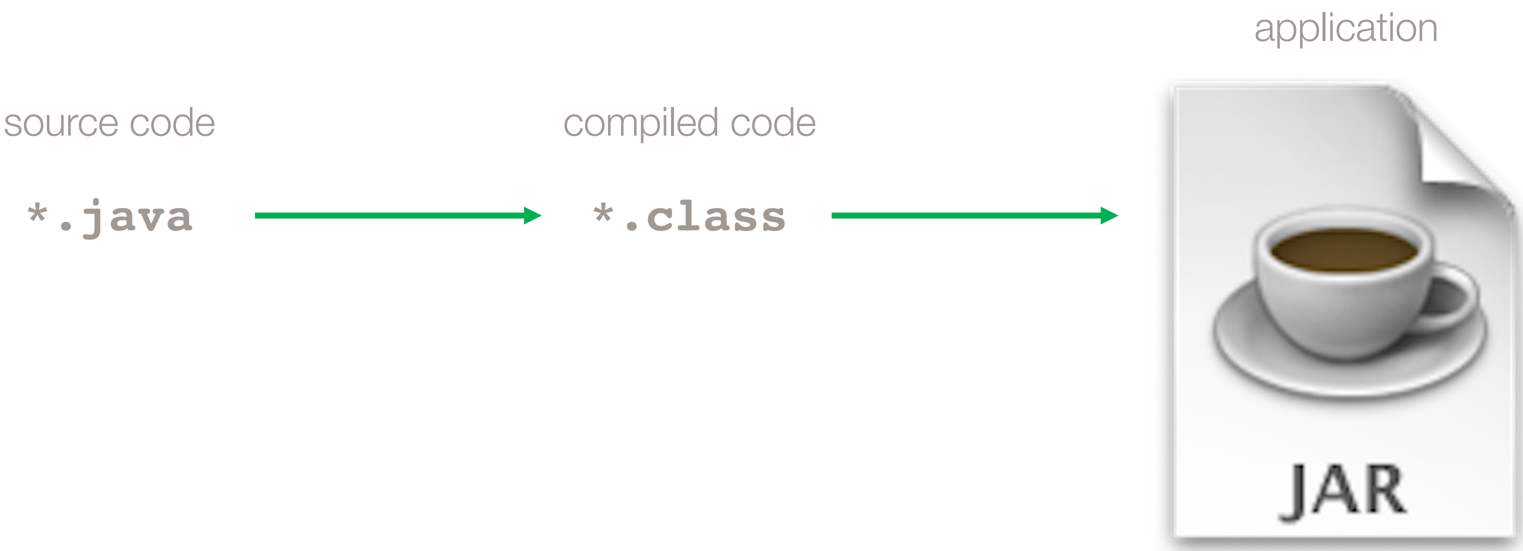
JAVA APPLICATIONS

Applications written in Java:

- Many Android phone apps
- IntelliJ
- Other IDEs e.g. Eclipse
- Original version of Minecraft
- Many more

JAVA APPLICATIONS

*.jar = a runnable Java application



source code

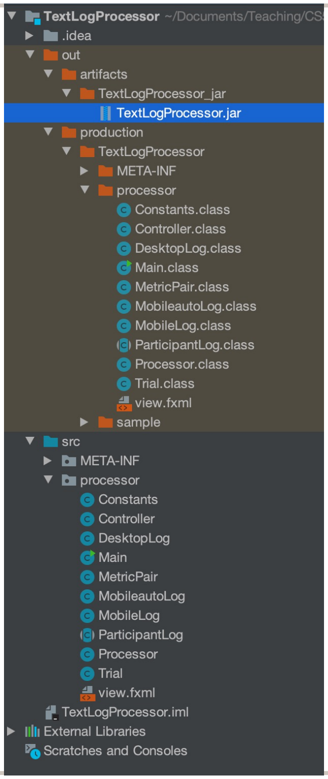
`*.java`

compiled code

`*.class`

application

EXAMPLE JAVA APPLICATION

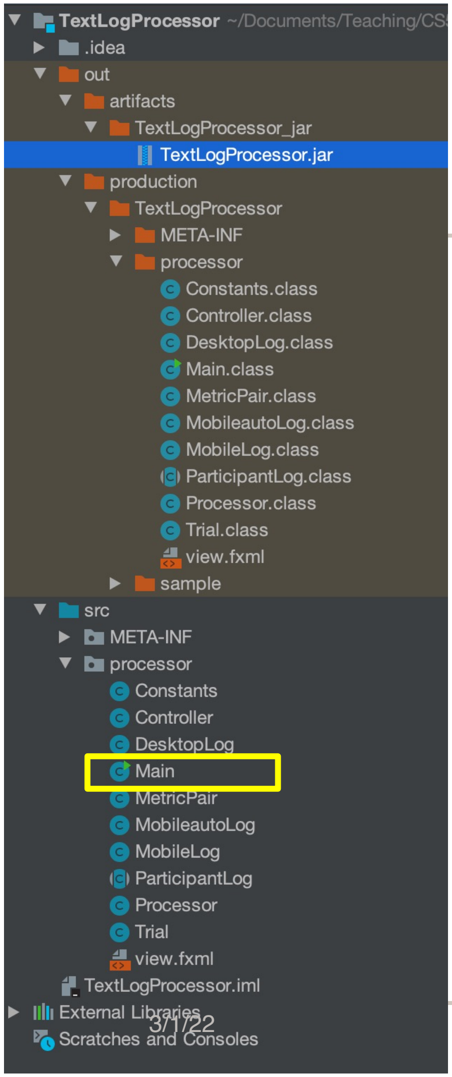


```
▼ TextLogProcessor ~/Documents/Teaching/CS
  ▶ .idea
  ▼ out
    ▼ artifacts
      ▼ TextLogProcessor_jar
          TextLogProcessor.jar
    ▼ production
      ▼ TextLogProcessor
        ▶ META-INF
        ▼ processor
            Constants.class
            Controller.class
            DesktopLog.class
            Main.class
            MetricPair.class
            MobileautoLog.class
            MobileLog.class
            ParticipantLog.class
            Processor.class
            Trial.class
            view.fxml
        ▶ sample
  ▼ src
    ▶ META-INF
    ▼ processor
        Constants
        Controller
        DesktopLog
        Main
        MetricPair
        MobileautoLog
        MobileLog
        ParticipantLog
        Processor
        Trial
        view.fxml
    TextLogProcessor.iml
▶ External Libraries
  Scratches and Consoles
```

← application

compiled code

source code



```
▼ TextLogProcessor ~/Documents/Teaching/CS
  ▶ .idea
  ▼ out
    ▼ artifacts
      ▼ TextLogProcessor_jar
          TextLogProcessor.jar
    ▼ production
      ▼ TextLogProcessor
        ▶ META-INF
        ▼ processor
            Constants.class
            Controller.class
            DesktopLog.class
            Main.class
            MetricPair.class
            MobileautoLog.class
            MobileLog.class
            ParticipantLog.class
            Processor.class
            Trial.class
            view.fxml
        ▶ sample
  ▼ src
    ▶ META-INF
    ▼ processor
        Constants
        Controller
        DesktopLog
        Main
        MetricPair
        MobileautoLog
        MobileLog
        ParticipantLog
        Processor
        Trial
        view.fxml
    TextLogProcessor.iml
▶ External Libraries
  Scratches and Consoles
```

EXAMPLE JAVA APPLICATION

Before the application can be created, the program needs an **entry point**

- A *single class* that runs when the application is run
- Handles input from user, if any
  - GUI
  - command line
- Coordinates the ”business logic”
- Handles output to user, if any
  - GUI
  - command line

DEFINING AN APPLICATION ENTRY POINT

- Some class, often called **`Main`**
- Must have a **`main`** method
  - Signature must be exactly as follows…

```
public static void main(String[] args) {
  // business logic here
}
```

DEFINING AN APPLICATION ENTRY POINT

- Some class, often called **`Main`**
- Must have a **`main`** method
  - Signature must be exactly as follows…

```
public static void main(String[] args) {
  // business logic here
}
```

Command line arguments passed in here

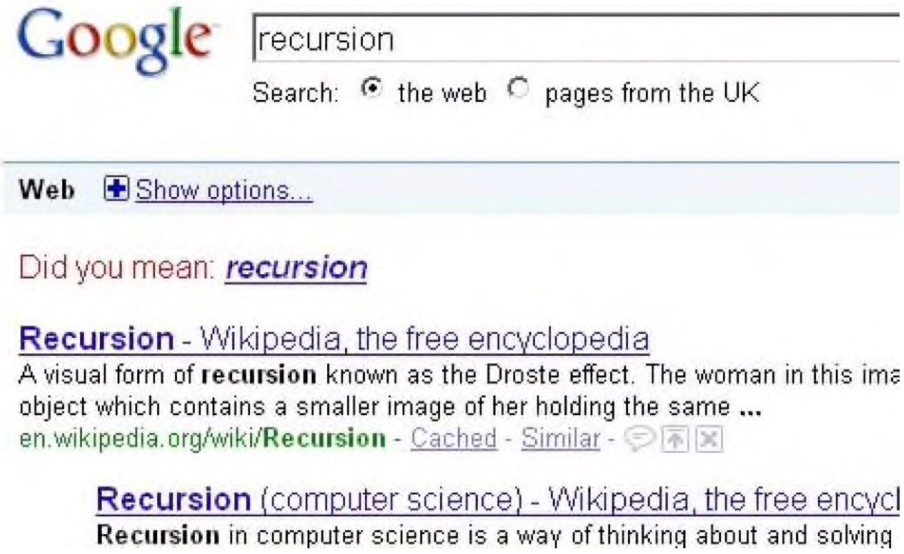
`java MyApp hello 9000`

`args` will equal **`[“hello”, “9000”];`**

RECURSIVE DATA STRUCTURES

CS 5004, SPRING 2022– LECTURE 6

RECURSION



recursion

Search: ● the web ○ pages from the UK

Web Show options...

Did you mean: *recursion*

Recursion - Wikipedia, the free encyclopedia
A visual form of **recursion** known as the Droste effect. The woman in this ima
object which contains a smaller image of her holding the same **...**
en.wikipedia.org/wiki/**Recursion** - Cached - Similar -

Recursion (computer science) - Wikipedia, the free encycl
Recursion in computer science is a way of thinking about and solving

[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 ALGORITHM

- 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

- Recursive data structure - a data structure partially composed of smaller or simpler instances of the same data structure

- Just like recursive functions, recursive structures have:
  - Base case
  - Recursive case

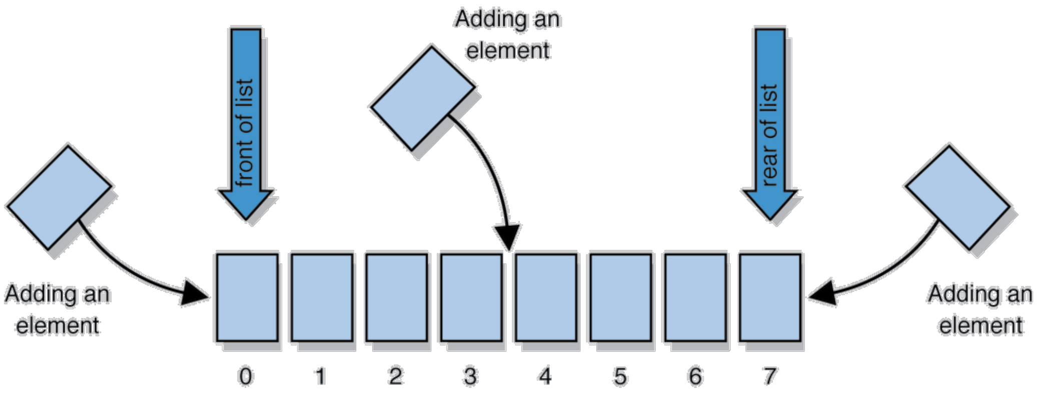
RECURSIVE LINKED LIST

CS 5004, SPRING 2022– LECTURE 6

REVIEW: LIST ADT

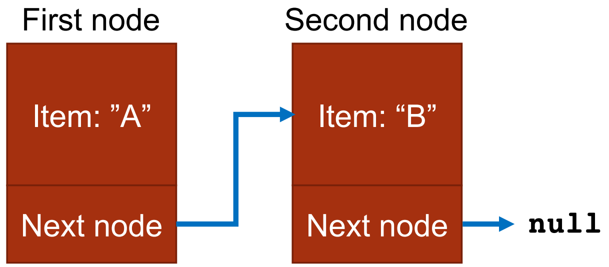
- Note: a List ADT and a linked list are not the same – List ADT can be implemented *using* a linked list

- List ADT – an ordered collection (also known as a sequence)



LINKED LIST

Sequential* version



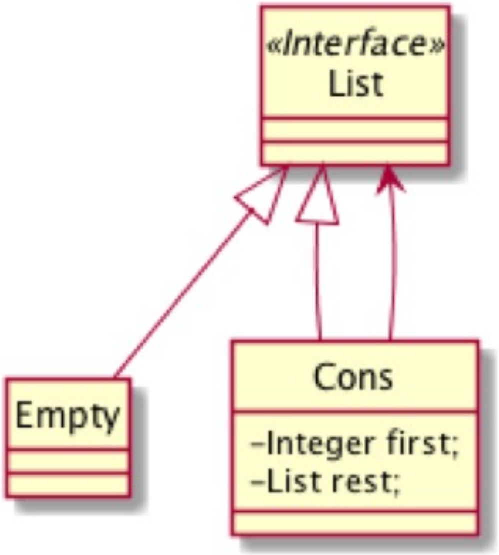
```java
public class Node {
  private DataType item;
  private Node next;

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

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

LISTS AS RECURSIVE DATA STRUCTURES

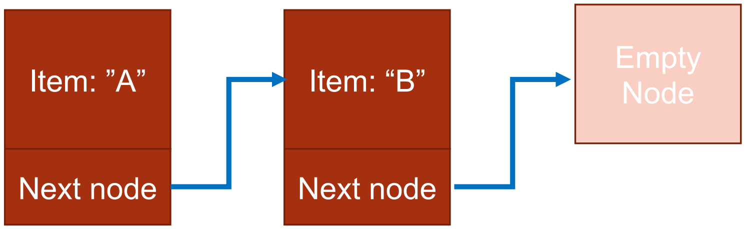
- List – an ordered collection (also known as a sequence)

- A linked list is either:
  - Null (base case)
  - A node whose next field references a list



LISTS AS RECURSIVE DATA STRUCTURES

Recursive version



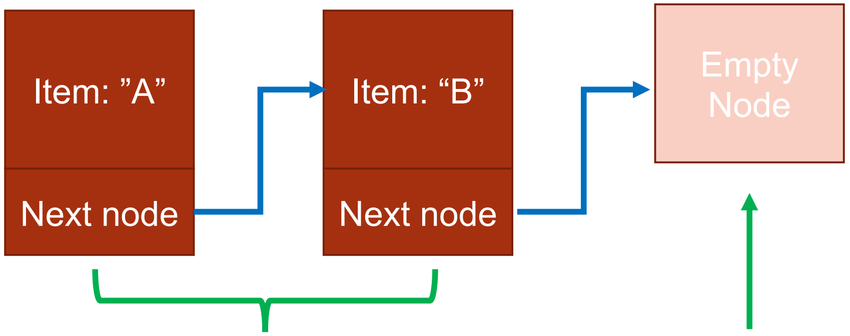
Replace null at end of list with special type of Node

Recursive data structure
A data structure partially composed of smaller or simpler instances of the same data structure.
Just like recursive functions, recursive structures have:

- Base case
- Recursive cases

LISTS AS RECURSIVE DATA STRUCTURES

Recursive version



Item: ”A”

Next node

Item: “B”

Next node

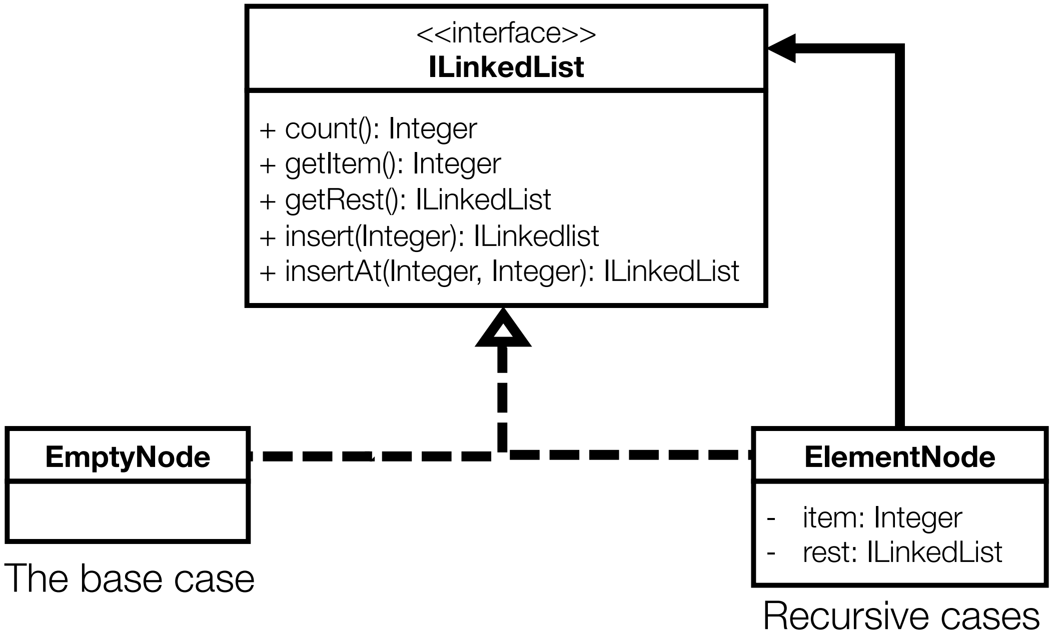
Empty Node

Replace null at end of list with special type of Node

Recursive case
Non-empty list

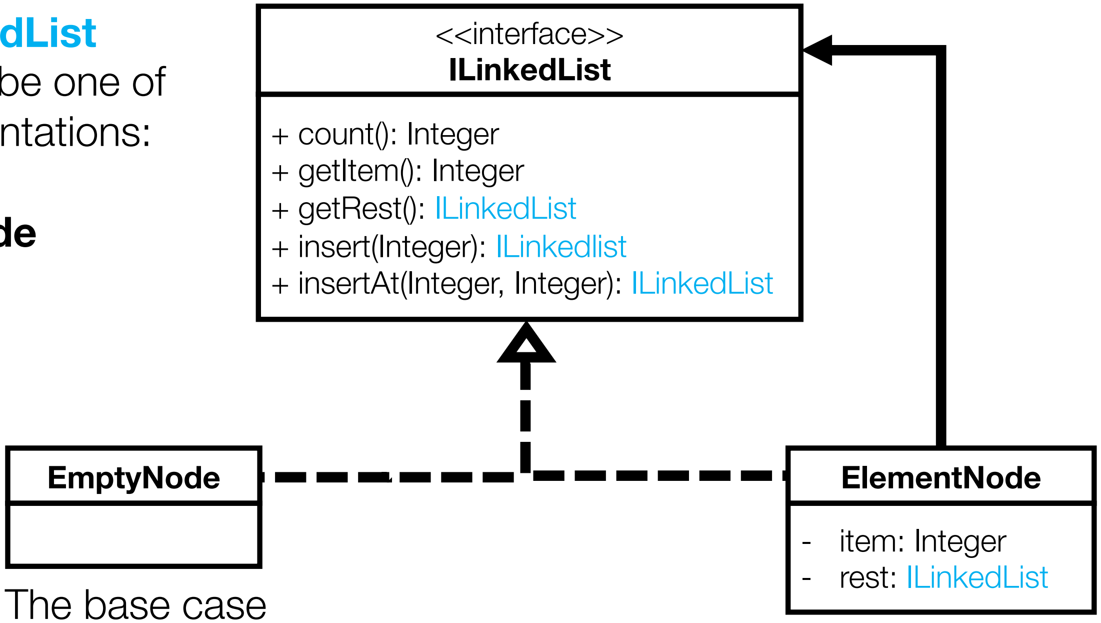
Base case
An empty list

RECURSIVE LINKED LIST IMPLEMENTATION



RECURSIVE LINKED LIST IMPLEMENTATION

Every **ILinkedList** returned will be one of two implementations:
EmptyNode
ElementNode



<<interface>>
ILinkedList

+ count(): Integer
+ getItem(): Integer
+ getRest(): ILinkedList
+ insert(Integer): ILinkedlist
+ insertAt(Integer, Integer): ILinkedList

EmptyNode

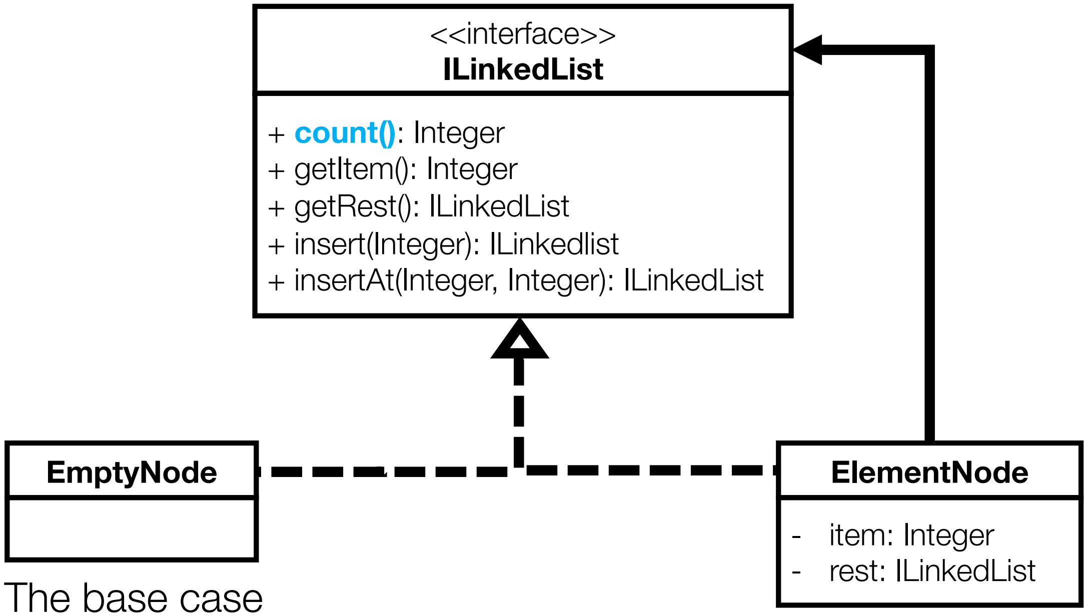
ElementNode

- item: Integer
- rest: ILinkedList

The base case

Recursive cases

RECURSIVE LINKED LIST IMPLEMENTATION



<<interface>>
ILinkedList

+ **count()**: Integer
+ getItem(): Integer
+ getRest(): ILinkedList
+ insert(Integer): ILinkedlist
+ insertAt(Integer, Integer): ILinkedList

EmptyNode

ElementNode

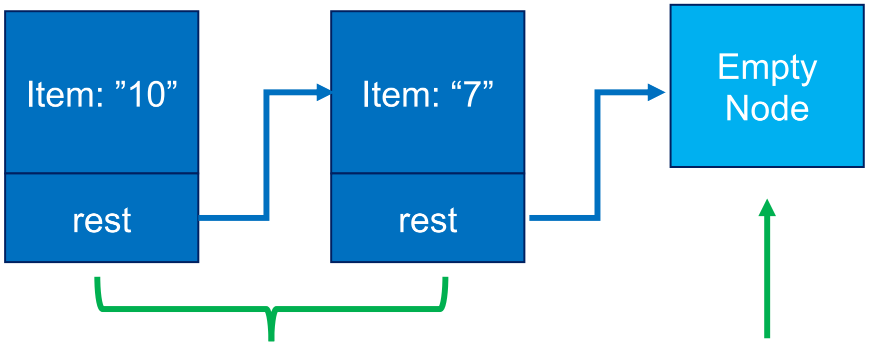
- item: Integer
- rest: ILinkedList

The base case

Recursive cases

RECURSIVE LINKED LIST: COUNT()

Where to start?

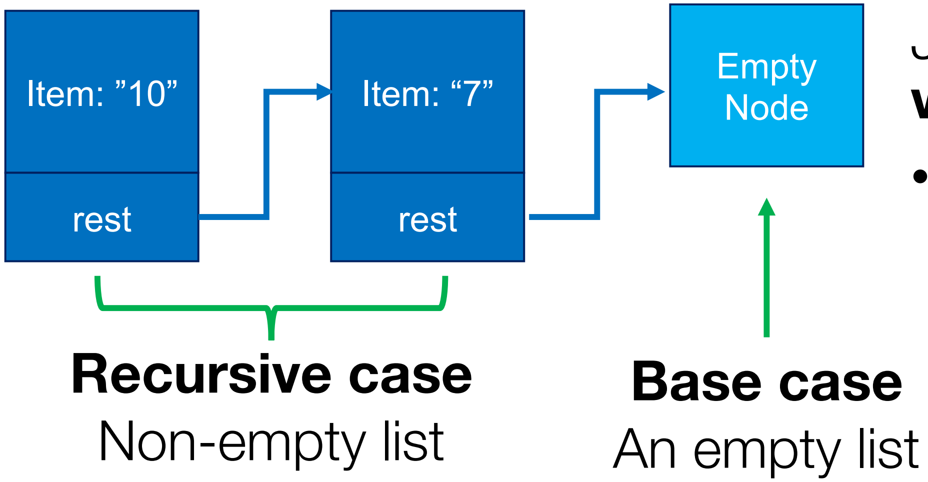


Recursive case
Non-empty list

Base case
An empty list

RECURSIVE LINKED LIST: COUNT()

Where to start?



Recursive case
Non-empty list

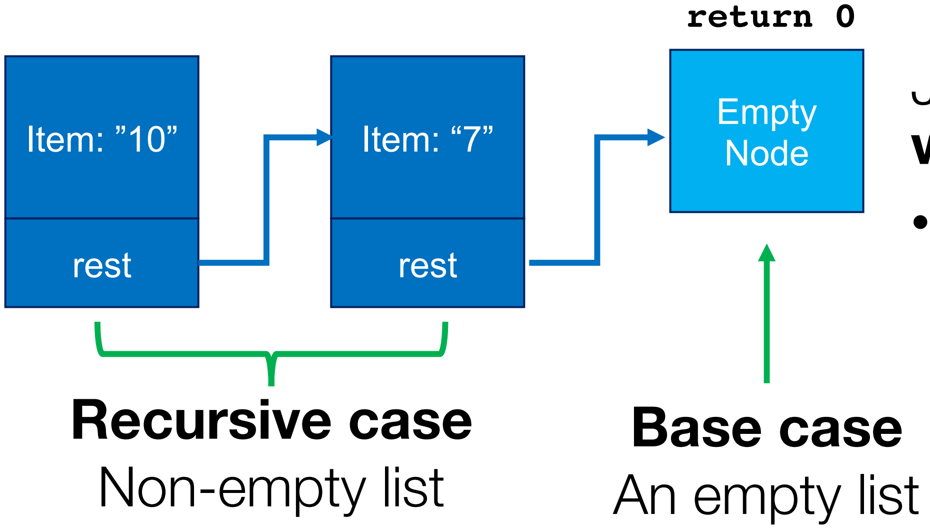
Base case
An empty list

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?

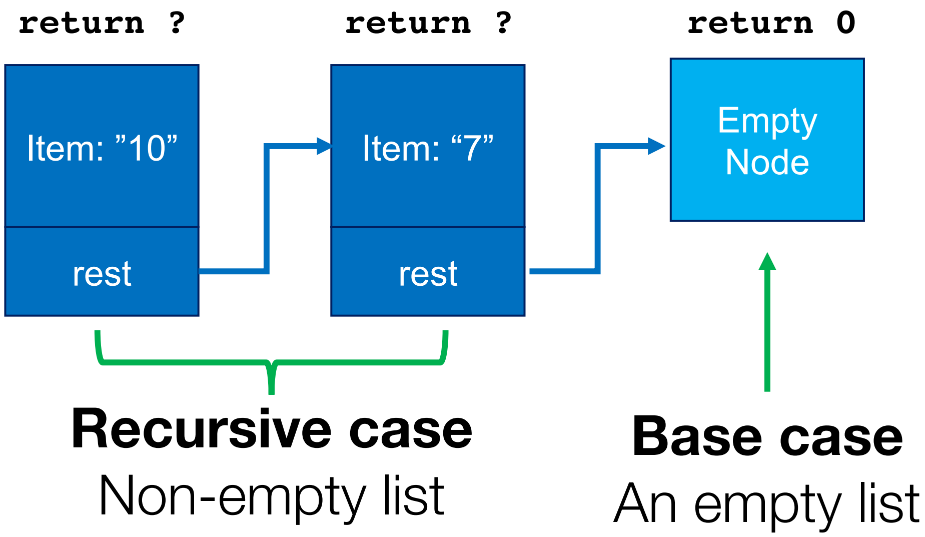


Just like a recursive function, **start with the base case**

- 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 case

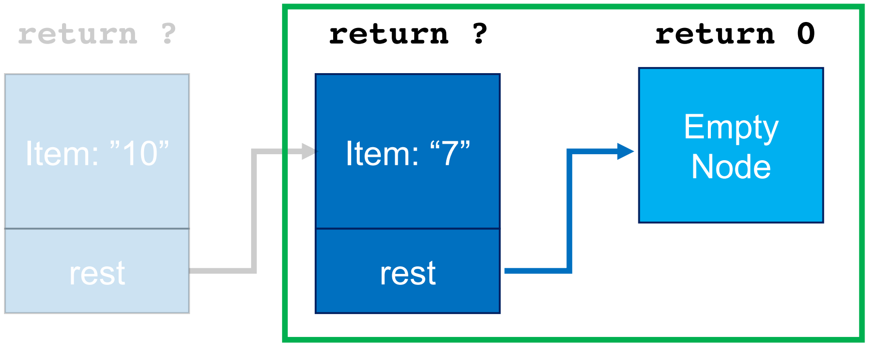
Non-empty list

Base case

An empty list

RECURSIVE LINKED LIST: COUNT()

What about the recursive case?



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

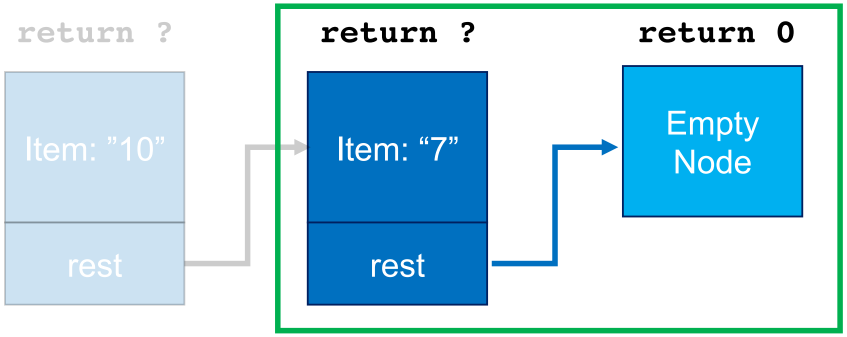
What we know:

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

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.

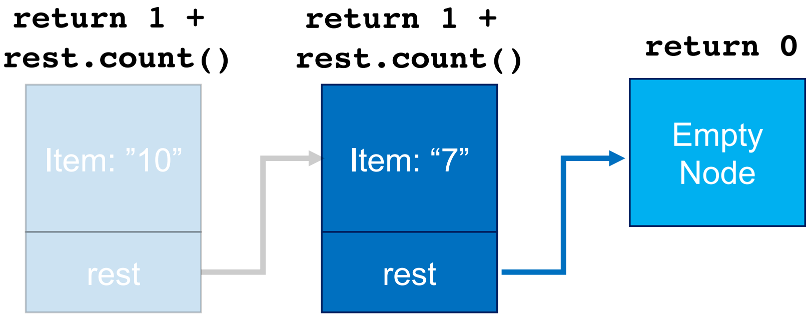
What we know:

- **`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.

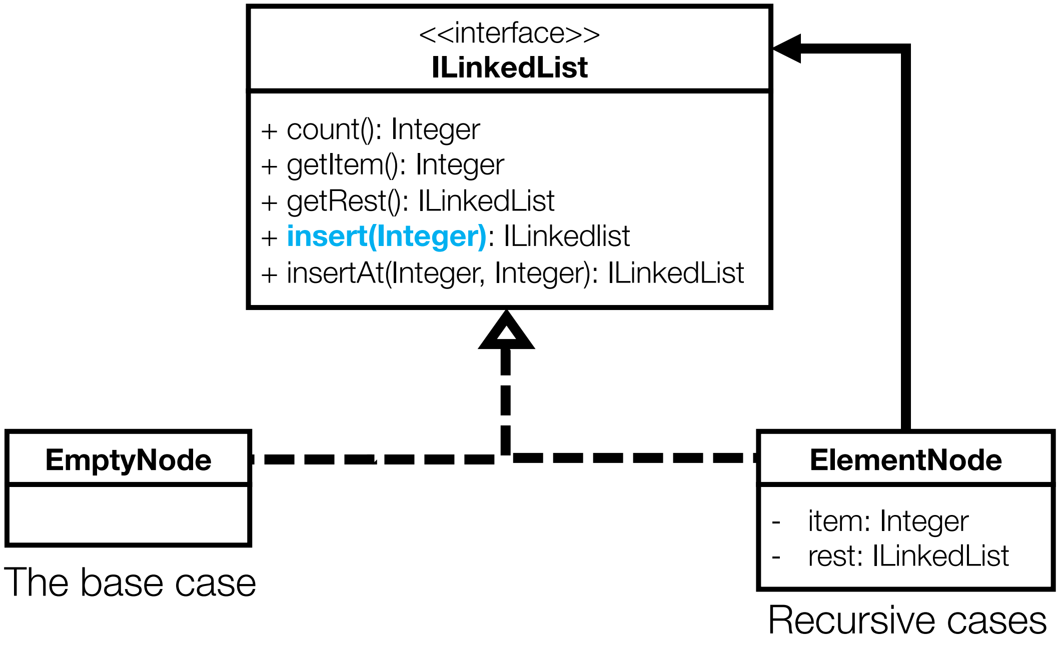
What we know:

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

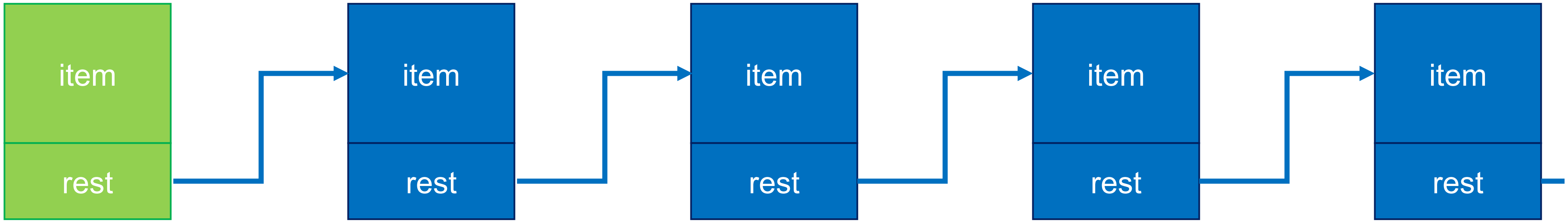
So, **`this.count()`** should return...

`1 + this.rest.count()`

RECURSIVE LINKED LIST IMPLEMENTATION



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

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

ElementNode.java

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

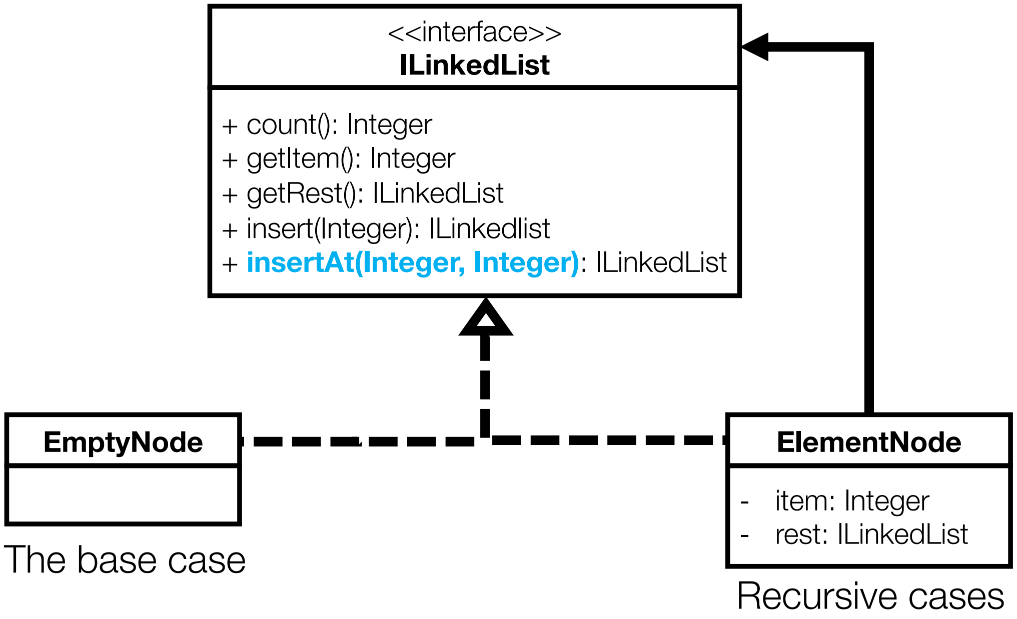
`this` **represents the current "head" of the List**

HOW DOES JAVA KNOW WHICH VERSION TO CALL?

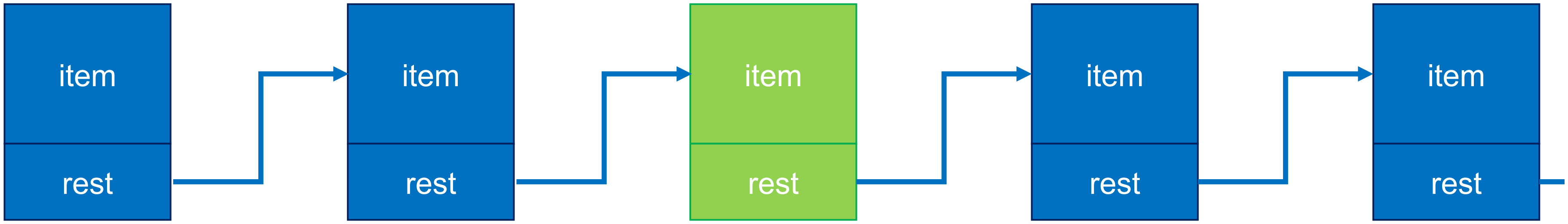
Dynamic dispatch

- 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

RECURSIVE LINKED LIST IMPLEMENTATION



INSERTAT(INTEGER, INTEGER): ILINKEDLIST



Create a new ElementNode containing the Integer, put it at index e.g. 2

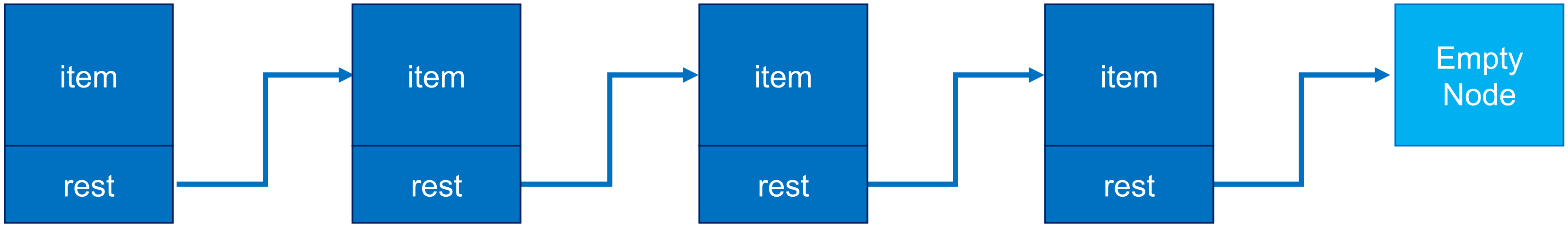
INSERTAT(INTEGER, INTEGER): ILINKEDLIST

- Insert at given index
- Will need to recursively check nodes to find the right index
- Also need to check index is in bounds

INSERTAT(INTEGER, INTEGER): ILINKEDLIST

- Insert at given index
- Will need to recursively check nodes to find the right index
- Also need to check index is in bounds

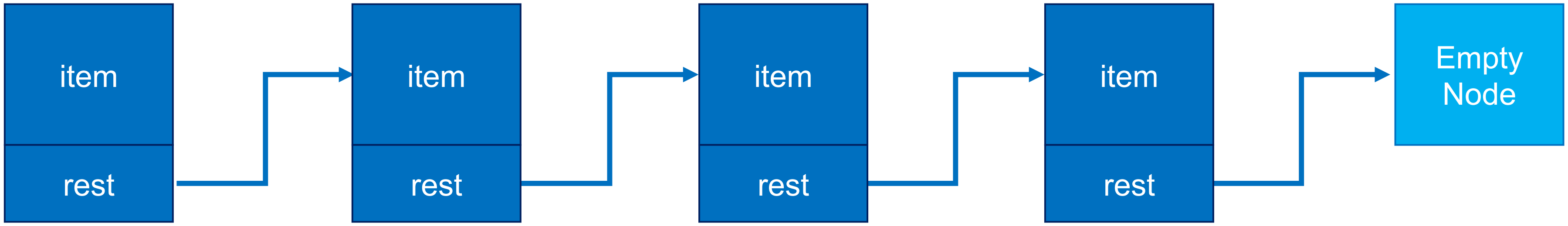
EXAMPLE: INSERT AT INDEX 2



Start at node 0

index = 2

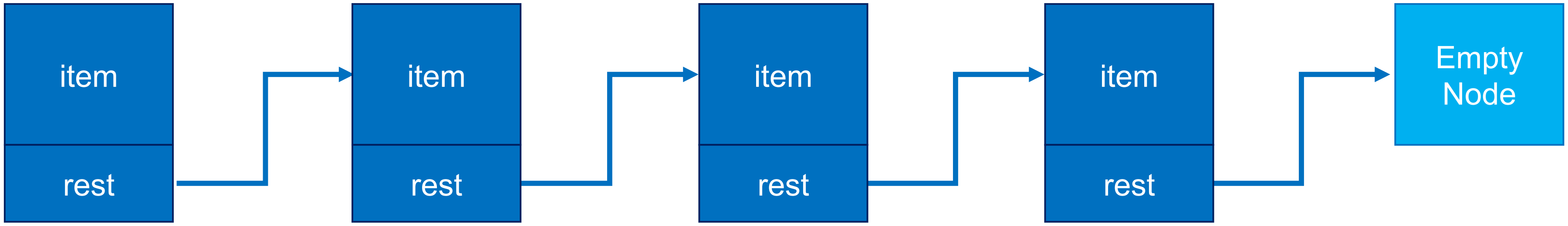
EXAMPLE: INSERT AT INDEX 2



Go to node 1, subtract 1 from index

index = 1

EXAMPLE: INSERT AT INDEX 2

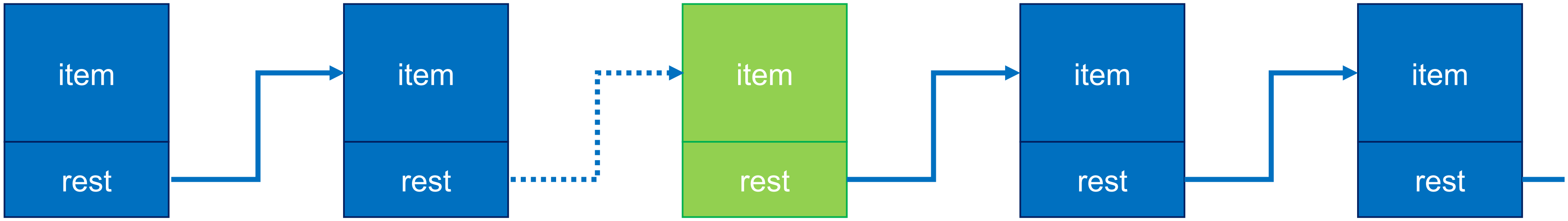


Go to node 2, subtract 1 from index

Index = 0

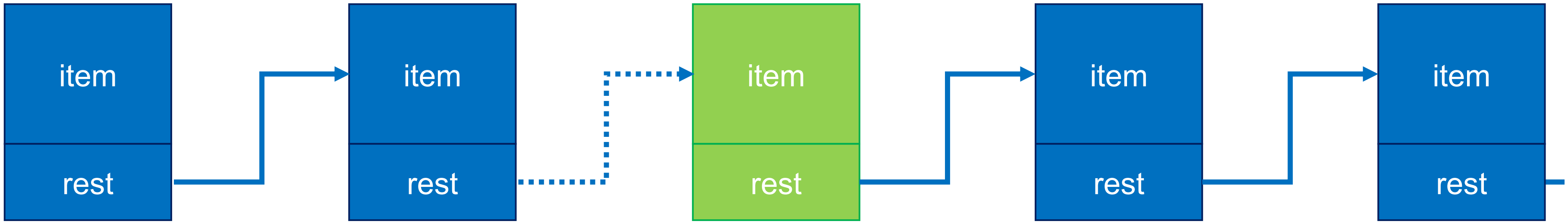
EXAMPLE: INSERT AT INDEX 2

-



Create new node, set its next node to point to node already at node 2…

EXAMPLE: INSERT AT INDEX 2



Create new node, set its next node to point to node already at node 2…

Connect to previous node (index 1)

INSERTAT IMPLEMENTATION – BASE CASE

- Start with the base case – existing list is empty
- Two cases:
  - index = 0, same as insert()
  - index is out of range > throw exception

INSERTAT IMPLEMENTATION – BASE CASE

```
public ILinkedList insertAt(Integer item, Integer index)
                               throws IndexOutOfBoundsException {
  if (!index.equals(0)) {
    throw new IndexOutOfBoundsException();
  } else {
    return new ElementNode(item, this);
  }
}
```

INSERTAT IMPLEMENTATION – BASE CASE

```
public ILinkedList insertAt(Integer item, Integer index)
                                   throws IndexOutOfBoundsException {
   if (!index.equals(0)) {
     throw new IndexOutOfBoundsException();
   } else {
     return new ElementNode(item, this);
   }
}
```

Index out of range

INSERTAT IMPLEMENTATION – BASE CASE

```
public ILinkedList insertAt(Integer item, Integer index)
                                  throws IndexOutOfBoundsException {
  if (!index.equals(0)) {
    throw new IndexOutOfBoundsException();
  } else {
    return new ElementNode(item, this);
  }
}
```

Insert here…

INSERTAT IMPLEMENTATION – RECURSIVE CASE

- Three cases:
  - Index is out of range → throw exception
  - This is the index we want to insert at → insert here
  - This is NOT the index we want to insert at → check next node

INSERTAT IMPLEMENTATION – RECURSIVE CASE

```
public ILinkedList insertAt(Integer item, Integer index)
                             throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                            this.rest.insertAt(item, index – 1));
  }
}
```

INSERTAT IMPLEMENTATION – RECURSIVE CASE

```java
public ILinkedList insertAt(Integer item, Integer index)
                throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
               this.rest.insertAt(item, index – 1));
  }
}
```

Index out of range

INSERTAT IMPLEMENTATION – RECURSIVE CASE

```
public ILinkedList insertAt(Integer item, Integer index)
                throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
               this.rest.insertAt(item, index – 1));
  }
}
```

This is NOT the index we want to insert at

INSERTAT IMPLEMENTATION – RECURSIVE CASE

```java
public ILinkedList insertAt(Integer item, Integer index)
                throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                this.rest.insertAt(item, index – 1));
  }
}
```

Recursive call

Reduce index

INSERTAT IMPLEMENTATION – RECURSIVE CASE

```
public ILinkedList insertAt(Integer item, Integer index)
                    throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                   this.rest.insertAt(item, index – 1));
  }
}
```

This is the index we want to insert at

INSERTAT IMPLEMENTATION – RECURSIVE CASE

```
public ILinkedList insertAt(Integer item, Integer index)
                throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                this.rest.insertAt(item, index – 1));
  }
}
```

Copy the contents of this node
to maintain link from previous node

INSERTAT IMPLEMENTATION – RECURSIVE CASE

```
public ILinkedList insertAt(Integer item, Integer index)
                throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
               this.rest.insertAt(item, index – 1));
  }
}
```

Return new node with new item to the previous recursive call

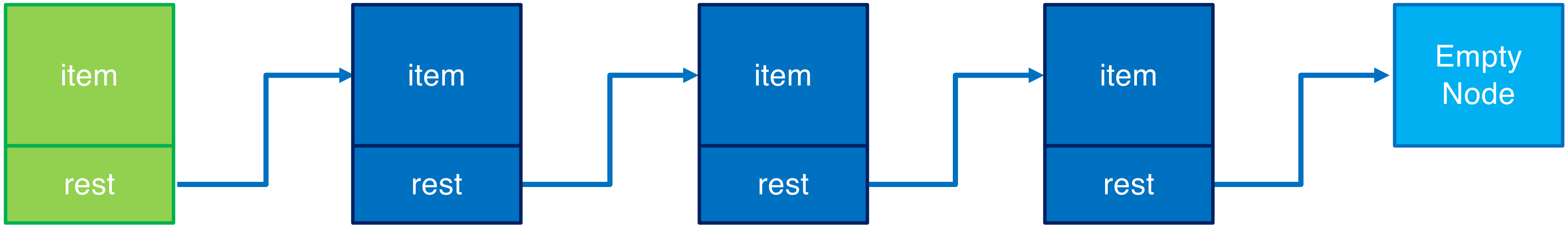
INSERTAT IMPLEMENTATION – RECURSIVE CASE

```
public ILinkedList insertAt(Integer item, Integer index)
                throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
               this.rest.insertAt(item, index – 1));
  }
}
```

Point “rest” to the copy node

EFFICIENCY

Linked List is very efficient when you are only adding/removing from the front



Accessing/inserting/removing anywhere else is less efficient

- You have to traverse the list each time

INSERTAT EFFICIENCY – RECURSIVE CASE

```
public ILinkedList insertAt(Integer item, Integer index)
                throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                     this.rest.insertAt(item, index – 1));
  }
}
```

Do we really need to do this?

INSERTAT EFFICIENCY – RECURSIVE CASE

Pro

- Prevents list traversal if the index is invalid
- `list.insertAt(100, -1);`

Con

- If the index is valid, the check is repeated every node
- `list.insertAt(100, list.count());`

```
if (index > this.count() || index < 0) {
  throw new IndexOutOfBoundsException();
}
```

INSERTAT EFFICIENCY – RECURSIVE CASE

Pro

- Prevents list traversal if the index is invalid
- **`list.insertAt(100, -1);`**
- Design choice: if you think invalid inserts will be more common, keep the check

Con

- If the index is valid, the check is repeated every node
- **`list.insertAt(100, list.count());`**

```
if (index > this.count() || index < 0) {
  throw new IndexOutOfBoundsException();
}
```

INSERTAT EFFICIENCY – RECURSIVE CASE

Pro

- Prevents list traversal if the index is invalid
- **`list.insertAt(100, -1);`**
- Design choice: if you think invalid inserts will be more common, keep the check

Con

- If the index is valid, the check is repeated every node
- **`list.insertAt(100, list.count());`**
- Design choice: if you think valid inserts will be more common, can we remove the check?

```
if (index > this.count() || index < 0) {
  throw new IndexOutOfBoundsException();
}
```

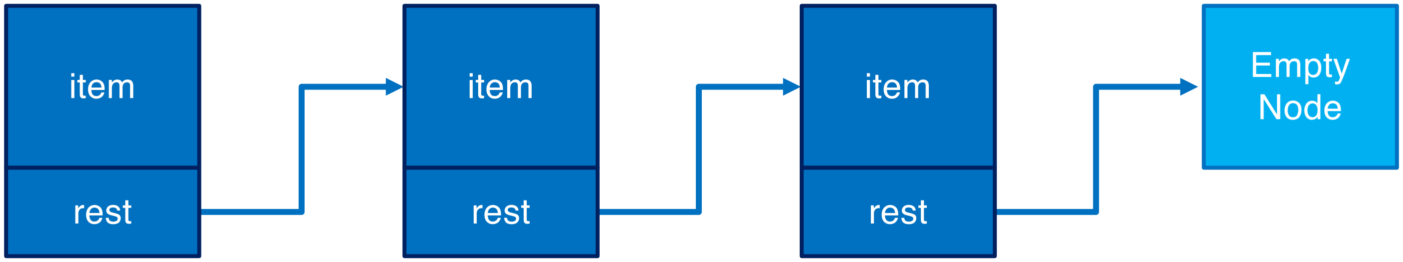
INSERTAT RECURSIVE CASE – ANOTHER APPROACH

```
public ILinkedList insertAt(Integer item, Integer index)
                throws IndexOutOfBoundsException {
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
               this.rest.insertAt(item, index – 1));
  }
}
```

We can remove the index check

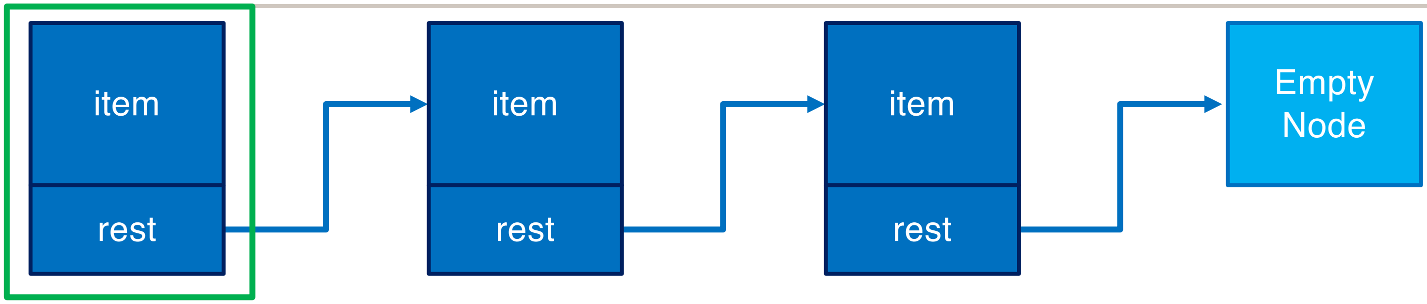
EmptyNode will catch an invalid index

EXAMPLE: INSERTAT(100, -1);



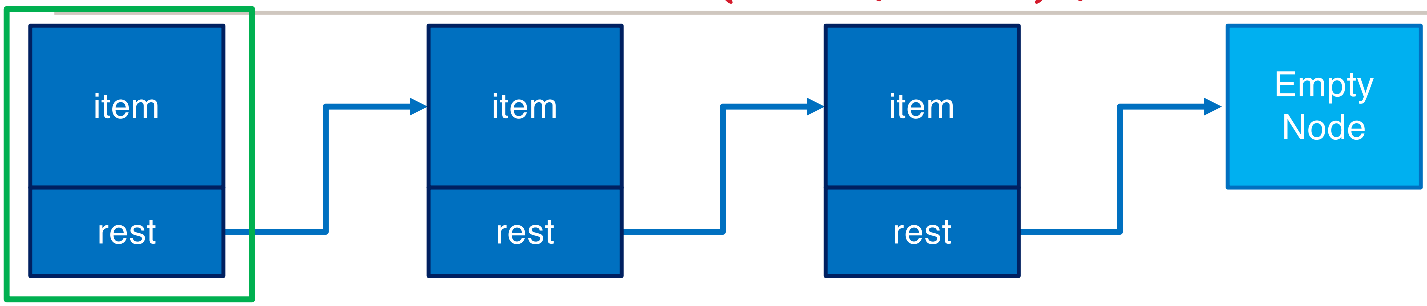
```
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index-1));
  }
}
```

EXAMPLE: INSERTAT(100, -1);



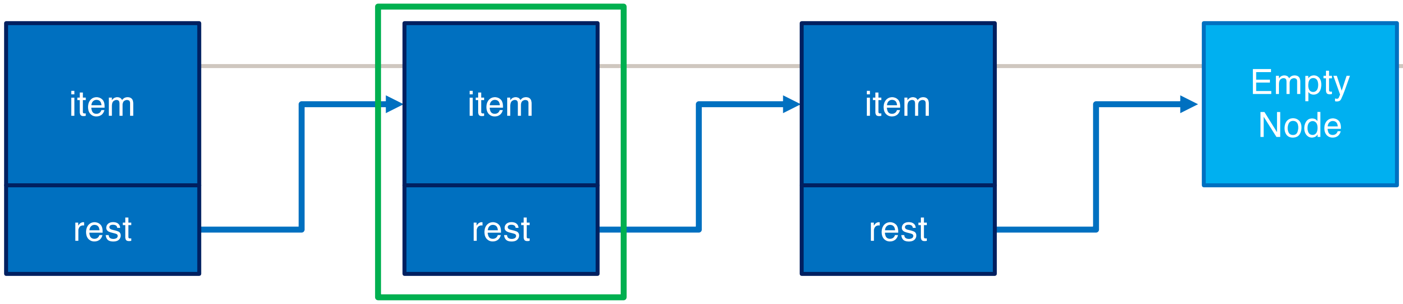
```
 index: -1
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index–1));
  }
}
```

EXAMPLE: INSERTAT(100, -1);



```
 index: -1
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index-1));
  }
}
```

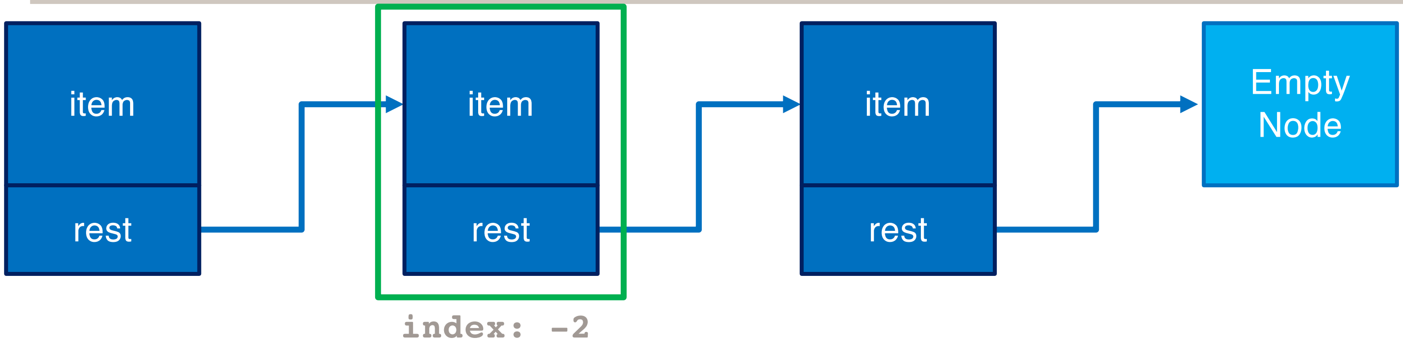
EXAMPLE: INSERTAT(100, -1);



index: -2

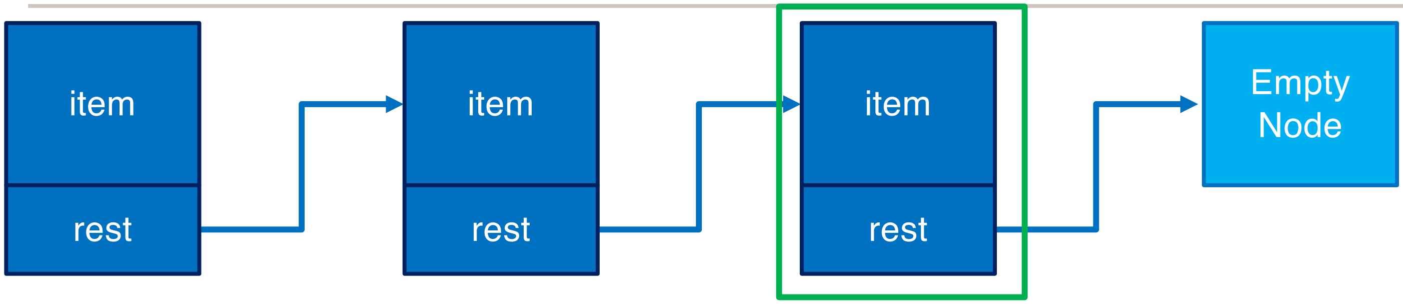
```
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index-1));
  }
}
```

EXAMPLE: INSERTAT(100, -1);



```
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index–1));
  }
}
```

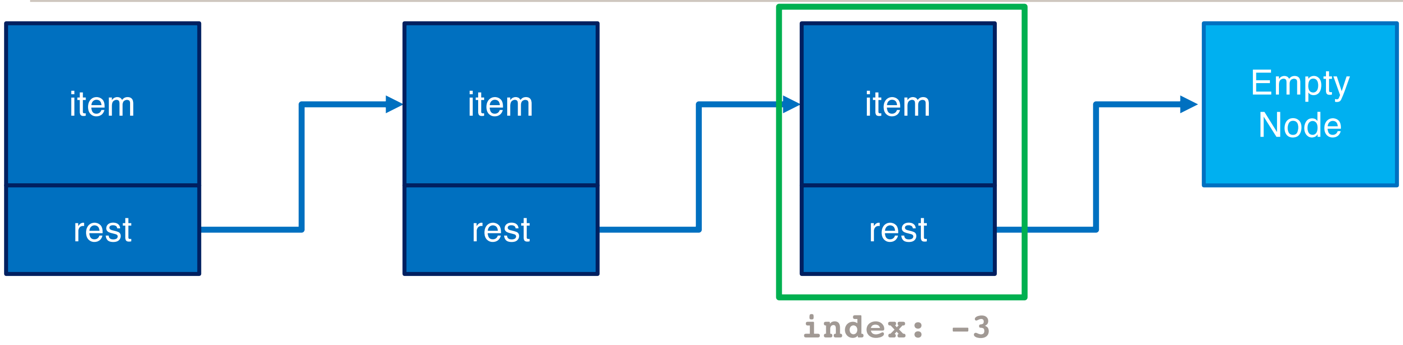
EXAMPLE: INSERTAT(100, -1);



index: -3

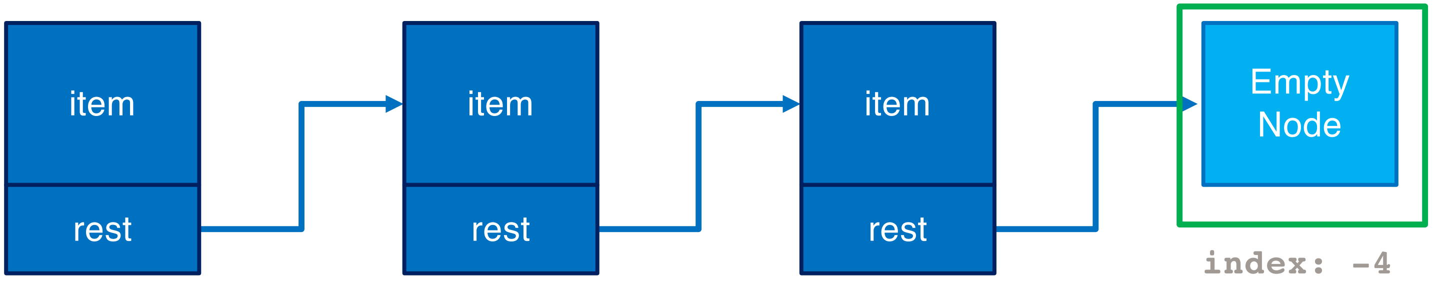
```
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index-1));
  }
}
```

EXAMPLE: INSERTAT(100, -1);



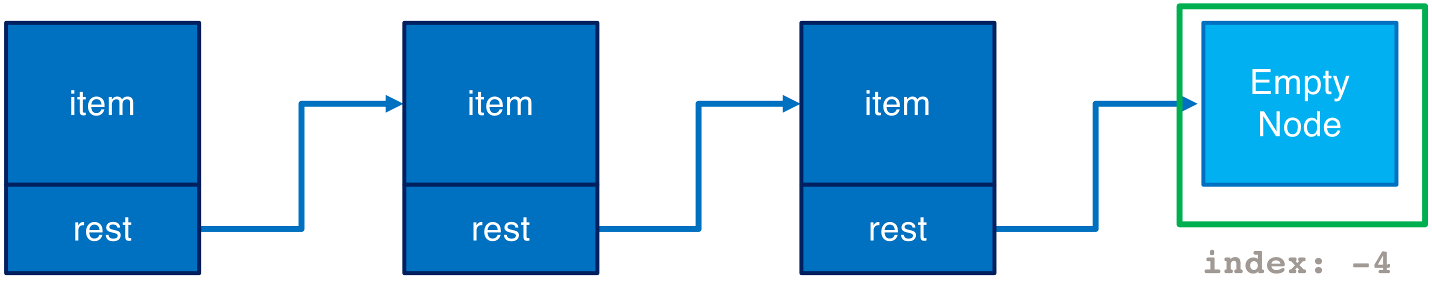
```
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index–1));
  }
}
```

EXAMPLE: INSERTAT(100, -1);



```
public ILinkedList insertAt(Integer item, Integer index) { // in EmptyNode
  if (!index.equals(0)) {
    throw new IndexOutOfBoundsException();
  } else {
    return new ElementNode(item, this);
  }
}
```

EXAMPLE: INSERTAT(100, -1);



```
public ILinkedList insertAt(Integer item, Integer index) { // in EmptyNode
  if (!index.equals(0)) {
    throw new IndexOutOfBoundsException();
  } else {
    return new ElementNode(item, this);
  }
}
```

STACK IMPLEMENTATION

CS 5004, SPRING 2022– LECTURE 6

THE MUTABLE STACK ADT

- **`void push(Integer item)`** - push an Integer on to the Stack
- **`Integer pop() throws EmptyStackException`** – returns and removes the most recently-added item.
- **`Integer top() throws EmptyStackException`** – returns the most recently-added item
- **`boolean isEmpty()`** – checks if the Stack is empty.

FIELDS AND CONSTRUCTOR(S)

```
public class Stack implements IStack {
  private ILinkedList top;

  private Stack() {
    this.top = new EmptyNode();
  }

  public static Stack createEmpty() {
    return new Stack();
  }
}
```

FIELDS AND CONSTRUCTOR(S)

```
public class Stack implements IStack {
  private ILinkedList top;

  private Stack() {
    this.top = new EmptyNode();
  }

  public static Stack createEmpty() {
    return new Stack();
  }
}
```

Underlying data structure

FIELDS AND CONSTRUCTOR(S)

```
public class Stack implements IStack {
  private ILinkedList top;

  private Stack() {
    this.top = new EmptyNode();
  }

  public static Stack createEmpty() {
    return new Stack();
  }
}
```

Why private?

- Sometimes want to prevent direct access to constructors
- Most useful for immutable
- Not necessary here (but fine)

FIELDS AND CONSTRUCTOR(S)

```
public class Stack implements IStack {
  private ILinkedList top;

  private Stack() {
    this.top = new EmptyNode();
  }

  public static Stack createEmpty() {
    return new Stack();
  }
}
```

Convenience method creates a Stack without “new”

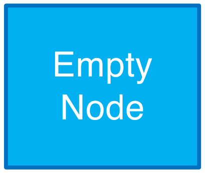
- static methods can’t go in an interface

CREATING A STACK

```
Stack aStack = Stack.createEmpty();
```

Inside **aStack**:

```
this.top =
```



Empty Node

THE MUTABLE STACK ADT

- `void push(Integer item)` - push an Integer on to the Stack
- `Integer pop() throws EmptyStackException` – returns and removes the most recently-added item.
- `Integer top() throws EmptyStackException` – returns the most recently-added item
- `boolean isEmpty()` – checks if the Stack is empty.

PUSH(INTEGER): VOID

```java
public void push(Integer item) {
  this.top = this.top.insert(item);
}
```

PUSH(INTEGER): VOID

```
public void push(Integer item) {
  this.top = this.top.insert(item);
}
```

Dynamic dispatch. One of:

`EmptyNode's insert(Integer)`

`ElementNode insert(Integer)`

The linked list is immutable so reassign `this.top`

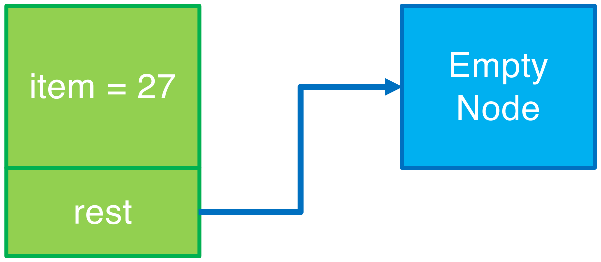
- `Stack` is mutable

PUSHING ITEMS ON TO THE STACK

```
aStack.push(27);
```

Inside **aStack**:

```
this.top =
```

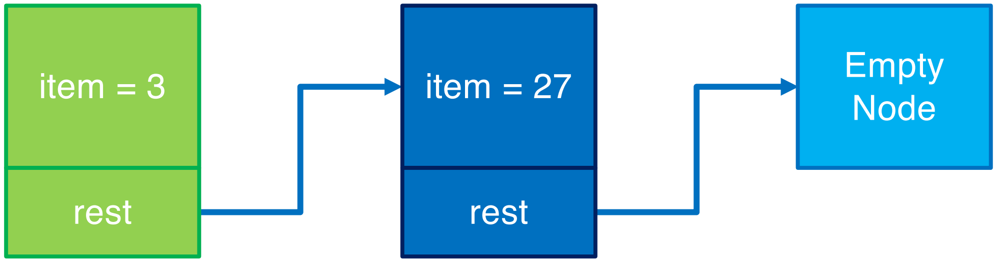


PUSHING ITEMS ON TO THE STACK

```
aStack.push(27);
aStack.push(3);
```

Inside **`aStack`**:

`this.top =`



THE MUTABLE STACK ADT

- **`void push(Integer item)`** - push an Integer on to the Stack
- **`Integer pop() throws EmptyStackException`** – returns and removes the most recently-added item.
- **`Integer top() throws EmptyStackException`** – returns the most recently-added item
- **`boolean isEmpty()`** – checks if the Stack is empty.

ISEMPTY(): BOOLEAN

```
public boolean isEmpty() {
  return this.top.count().equals(0);
}
```

ISEMPTY(): BOOLEAN

```
public boolean isEmpty() {
  return this.top.count().equals(0);
}
```

Dynamic dispatch. One of:

`EmptyNode's count()`

`ElementNode count()`

THE MUTABLE STACK ADT

- **`void push(Integer item)`** - push an Integer on to the Stack
- **`Integer pop() throws EmptyStackException`** – returns and removes the most recently-added item.
- **`Integer top() throws EmptyStackException`** – returns the most recently-added item
- **`boolean isEmpty()`** – checks if the Stack is empty.

TOP(): INTEGER

```
public Integer top() throws EmptyStackException {
  if (this.isEmpty())
    throw new EmptyStackException();
  return this.top.getItem();
}
```

TOP(): INTEGER

```
public Integer top() throws EmptyStackException {
  if (this.isEmpty())
    throw new EmptyStackException();
  return this.top.getItem();
}
```

Ensures the specification is met

TOP(): INTEGER

```
public Integer top() throws EmptyStackException {
  if (this.isEmpty())
    throw new EmptyStackException();
  return this.top.getItem();
}
```

Not necessary

- a “checked” exception
- built-in, inherits **`RunTimeException`**

TOP(): INTEGER

```
public Integer top() throws EmptyStackException {
  if (this.isEmpty())
    throw new EmptyStackException();
  return this.top.getItem();
}
```

Dynamic dispatch. One of:

`EmptyNode’s getItem()`

`ElementNode getItem()`

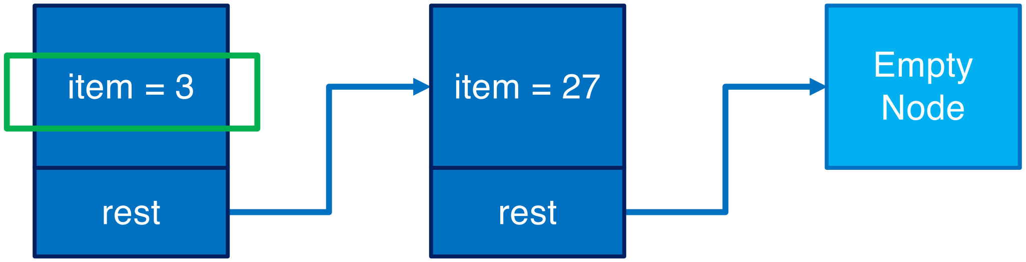
(*We* know it can’t be an EmptyNode but the compiler does not)

GETTING THE TOP ITEM

`aStack.top();` → 3

Inside **`aStack`**:

`this.top =`



THE MUTABLE STACK ADT

- `void push(Integer item)` - push an Integer on to the Stack
- `Integer pop() throws EmptyStackException` – returns and removes the most recently-added item.
- `Integer top() throws EmptyStackException` – returns the most recently-added item
- `boolean isEmpty()` – checks if the Stack is empty.

POP(): INTEGER

```
public Integer pop() throws EmptyStackException {
  Integer poppedItem = this.top();
  this.top = this.top.getRest();
  return poppedItem;
}
```

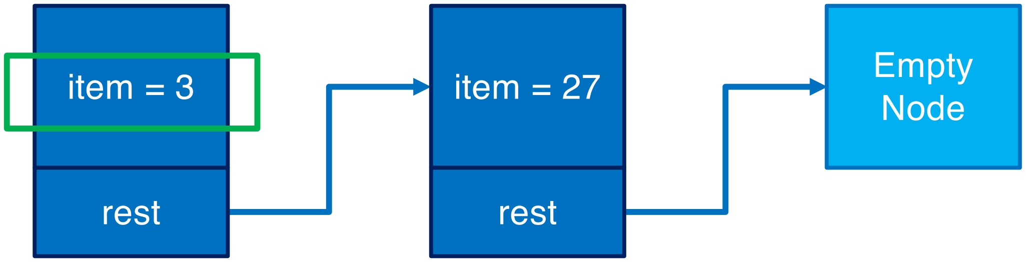
POP(): INTEGER

```
public Integer pop() throws EmptyStackException {
  Integer poppedItem = this.top();
  this.top = this.top.getRest();
  return poppedItem;
}
```

Will throw exception if empty

- Stores the return value if not

```
this.top =
```

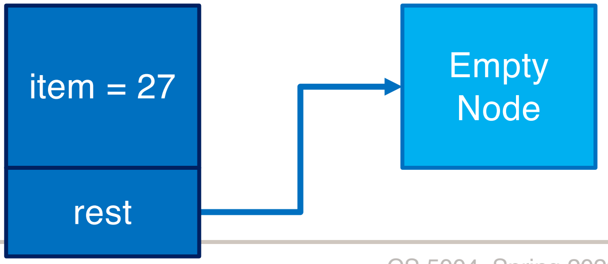


POP(): INTEGER

```
public Integer pop() throws EmptyStackException {
  Integer poppedItem = this.top();
  this.top = this.top.getRest();
  return poppedItem;
}
```

Removes the top element

```
this.top =
```



IMMUTABLE STACK IMPLEMENTATION

CS 5004, SPRING 2022– LECTURE 6

AN IMMUTABLE STACK ADT

- **`IImmutableStack createEmpty()`** – creates a new empty stack.
- **`IImmutableStack push(Integer item)`** - returns a new stack with item at the top.
- **`IImmutableStack pop() throws EmptyStackException`** – ~~returns and~~ removes the most recently-added item.
- **`Integer top() throws EmptyStackException`** – returns the most recently-added item
- **`boolean isEmpty()`** – checks if the Stack is empty.

AN IMMUTABLE STACK ADT

- `IImmutableStack createEmpty()` – creates a new empty stack.
- `IImmutableStack push(Integer item)` - returns a new stack with item at the top.
- `IImmutableStack pop() throws EmptyStackException` –removes the most recently-added item.
- `Integer top() throws EmptyStackException` – returns the most recently-added item
- `boolean isEmpty()` – checks if the Stack is empty.

FIELDS AND CONSTRUCTORS

```
public class ImmutableStack implements IImmutableStack {
  private final ILinkedList top;

  private ImmutableStack() {
    this.top = new EmptyNode();
  }

  private ImmutableStack(ILinkedList elements) {
    this.top = elements;
  }

}
```

FIELDS AND CONSTRUCTORS

```java
public class ImmutableStack implements IImmutableStack {
  private final ILinkedList top;

  private ImmutableStack() {
    this.top = new EmptyNode();
  }

  private ImmutableStack(ILinkedList elements) {
    this.top = elements;
  }

}
```

Underlying data structure

- `final` ensures immutability

FIELDS AND CONSTRUCTORS

```
public class ImmutableStack implements IImmutableStack {
  private final ILinkedList top;

  private ImmutableStack() {
    this.top = new EmptyNode();
  }

  private ImmutableStack(ILinkedList elements) {
    this.top = elements;
  }

}
```

Doesn’t *need* to be private

- emptyStack will serve as constructor

FIELDS AND CONSTRUCTORS

```
public class ImmutableStack implements IImmutableStack {
  private final ILinkedList top;

  private ImmutableStack() {
    this.top = new EmptyNode();
  }

  private ImmutableStack(ILinkedList elements) {
    this.top = elements;
  }

}
```

Definitely private → Don’t want clients to know about the underlying structure

- Need for immutable methods

CREATEEMPTY(): IIMMUTABLESTACK

Typically static, calls private/public constructor

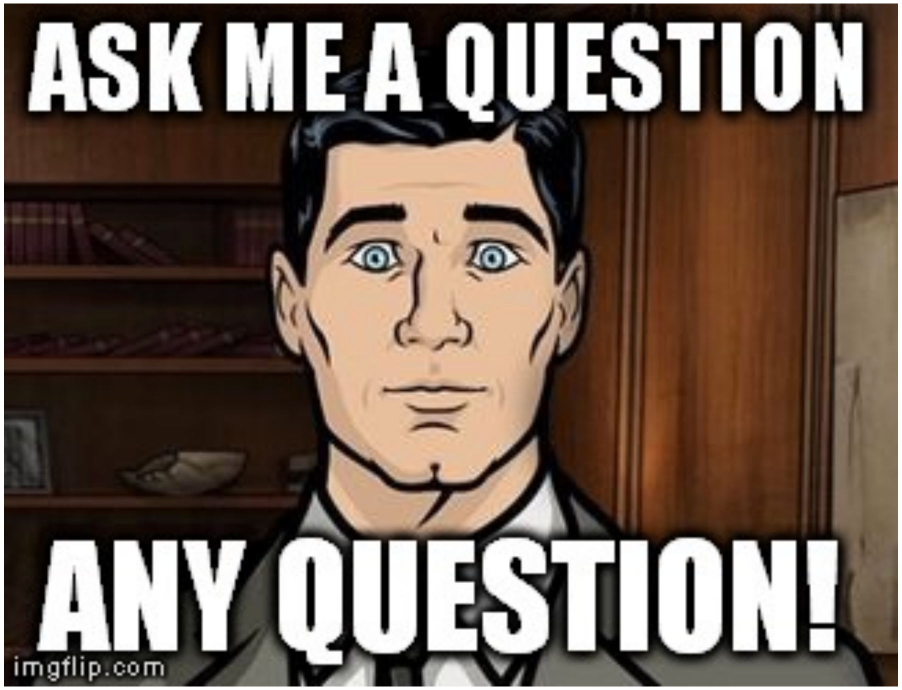
```
public static ImmutableStack createEmpty() {
  return new ImmutableStack();
}
```

ASIDE: CHOOSING ARRAY VS LINKED LIST FOR UNDERLYING DATA STRUCTURE

Rules of thumb

- Use an array when random access is important
  - i.e., access by index
  - will be faster for insert at index as well
- Use a linked list when random access/order is not important
  - faster for add/remove (doesn’t involve resizing)

YOUR QUESTIONS



[Meme credit: imgflip.com]

REFERENCES AND READING MATERIAL

- Java Getting Started (https://docs.oracle.com/javase/tutorial/getStarted/index.html)
- Object-Oriented Programming Concepts (https://docs.oracle.com/javase/tutorial/java/concepts/index.html)
- Language Basics (https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html)
- How to Design Classes (HtDC), Chapters 1-3
- JUnit: Getting Started (https://github.com/junit-team/junit4/wiki/Getting-started)
- JUnit: Assertions (https://github.com/junit-team/junit4/wiki/Assertions)
- Unit testing with JUnit: http:/www.vogella.com/tutorials/JUnit/article.html
- Java Tutorial: Interfaces and Inheritance: https://docs.oracle.com/javase/tutorial/java/IandI/index.html
- Java – Exceptions (https://www.tutorialspoint.com/java/java_exceptions.htm)
- Declare Your Own Exception (https://www.ibm.com/developerworks/community/blogs/738b7897-cd38-4f24-9f05-48dd69116837/entry/declare_your_own_java_exceptions?lang=en)
- Geeks for Geeks: Arrays in Java: https://www.geeksforgeeks.org/arrays-in-java/