CS 5004: Lecture 4

Northeastern University, Spring 2021

At the start of every lecture

1. Pull the latest code from the lecture-code repo
2. Open the Evening_lectures folder
3. Copy this week's folder somewhere else
   - So you can edit it without causing GitHub conflicts
4. Open the code:
   1. Find the build.gradle file in the folder called LectureX
   2. Double click it to open the project

Agenda

- Built-in collections
- Abstract Data Types
- List ADT deep dive

Java Collections

Collections

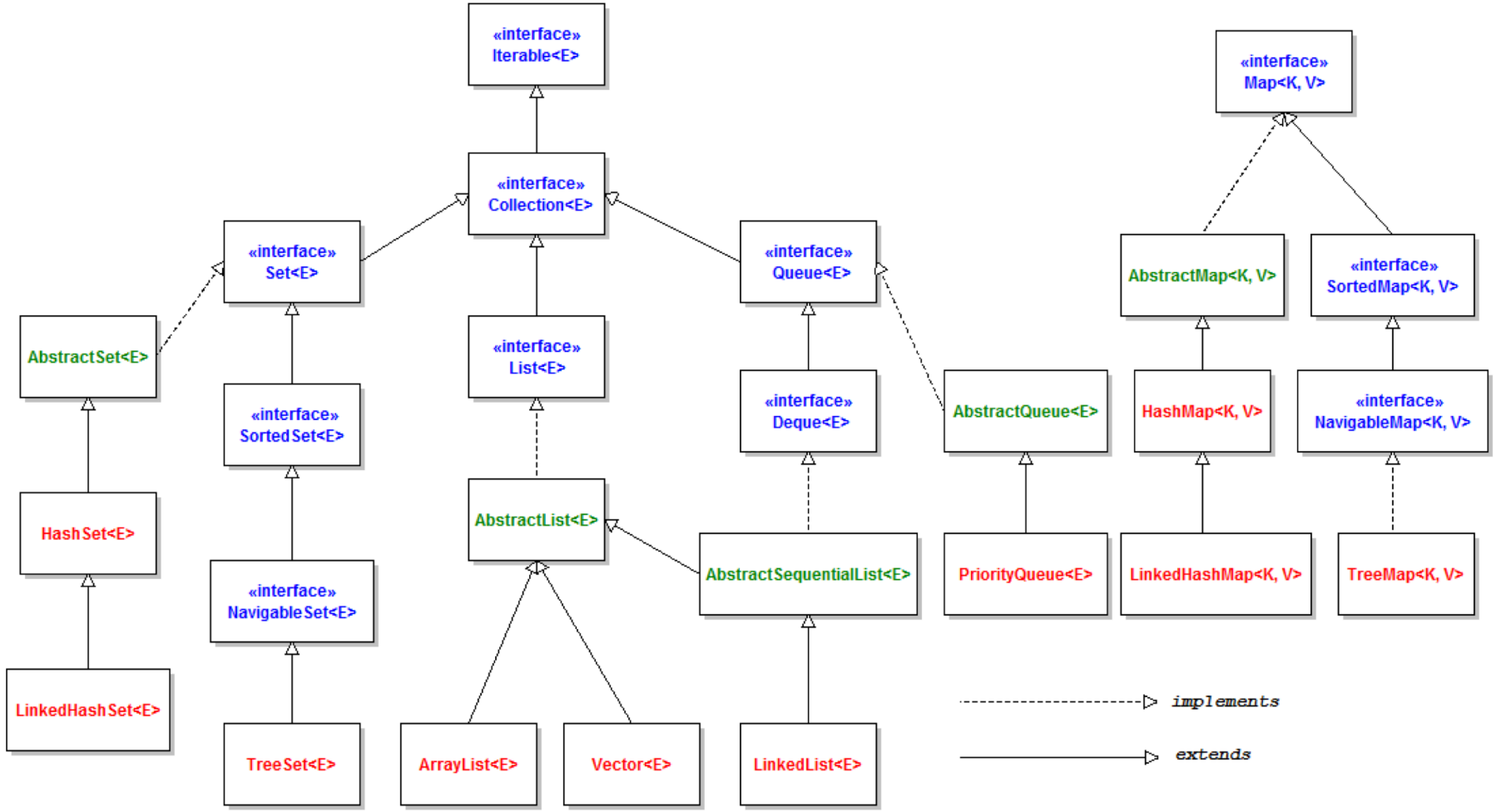
E.g. List, Stack, Queue, Sets…

- Store a group of objects
- Stored objects called **elements**

Typical operations:

- `add()`
- `remove()`
- `clear()`
- `size()`
- `contains()`

Java Collections API

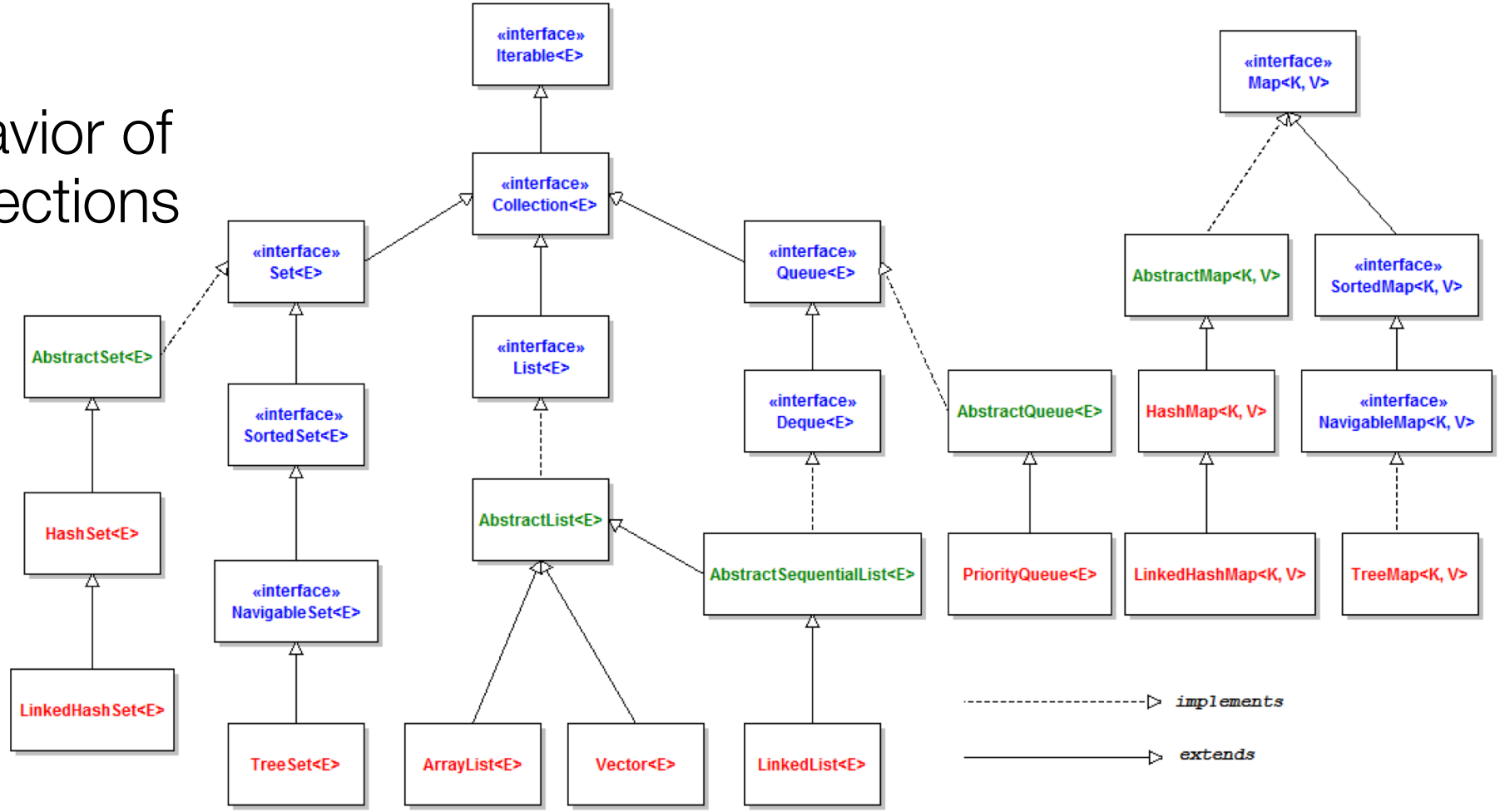


[Picture credit: http://www.codejava.net]

Java Collections API

Interfaces

- Define behavior of various collections

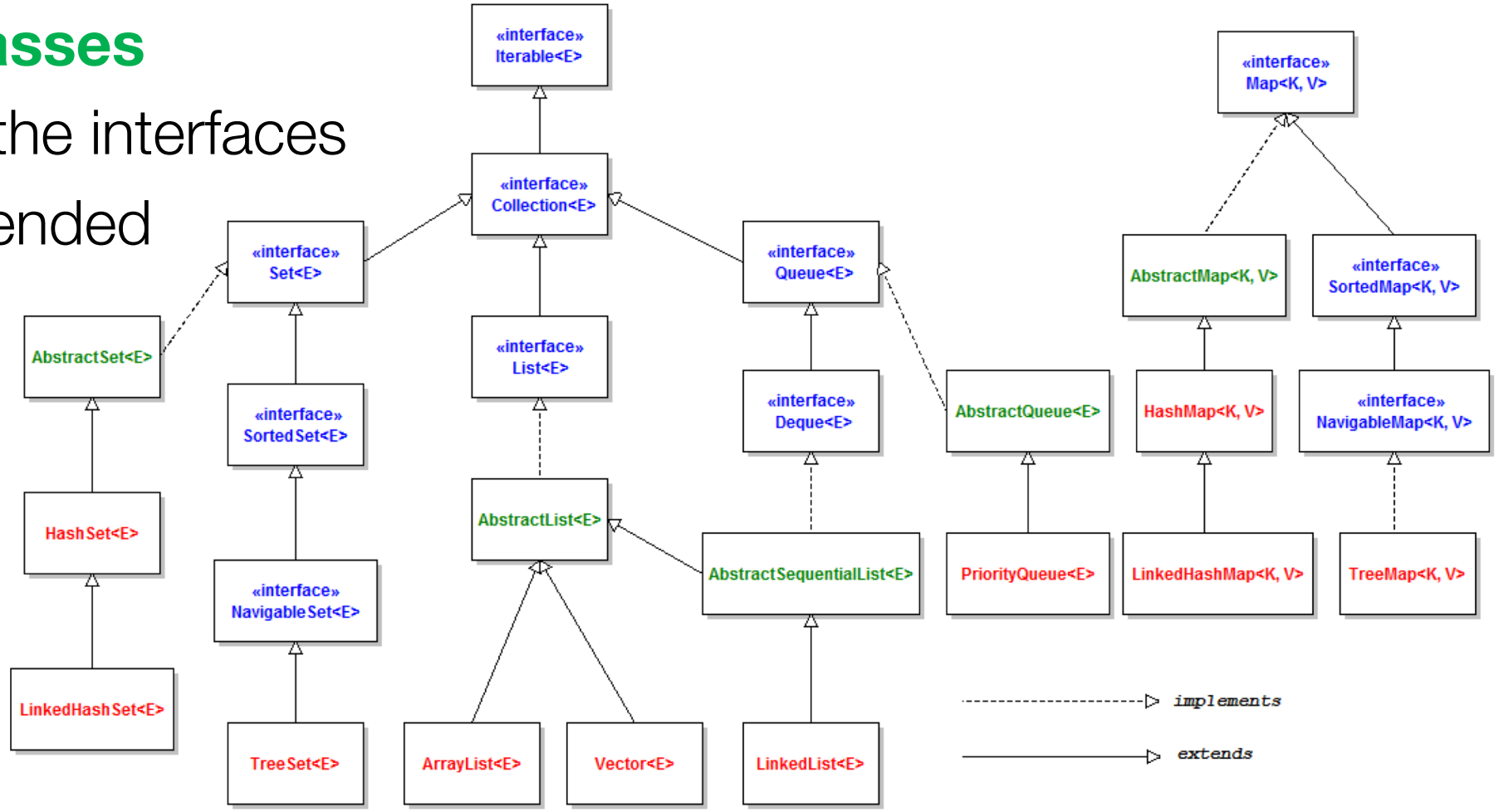


[Picture credit: http://www.codejava.net]

Java Collections API

Abstract classes

- Implement the interfaces
- Can be extended to create specialized collections

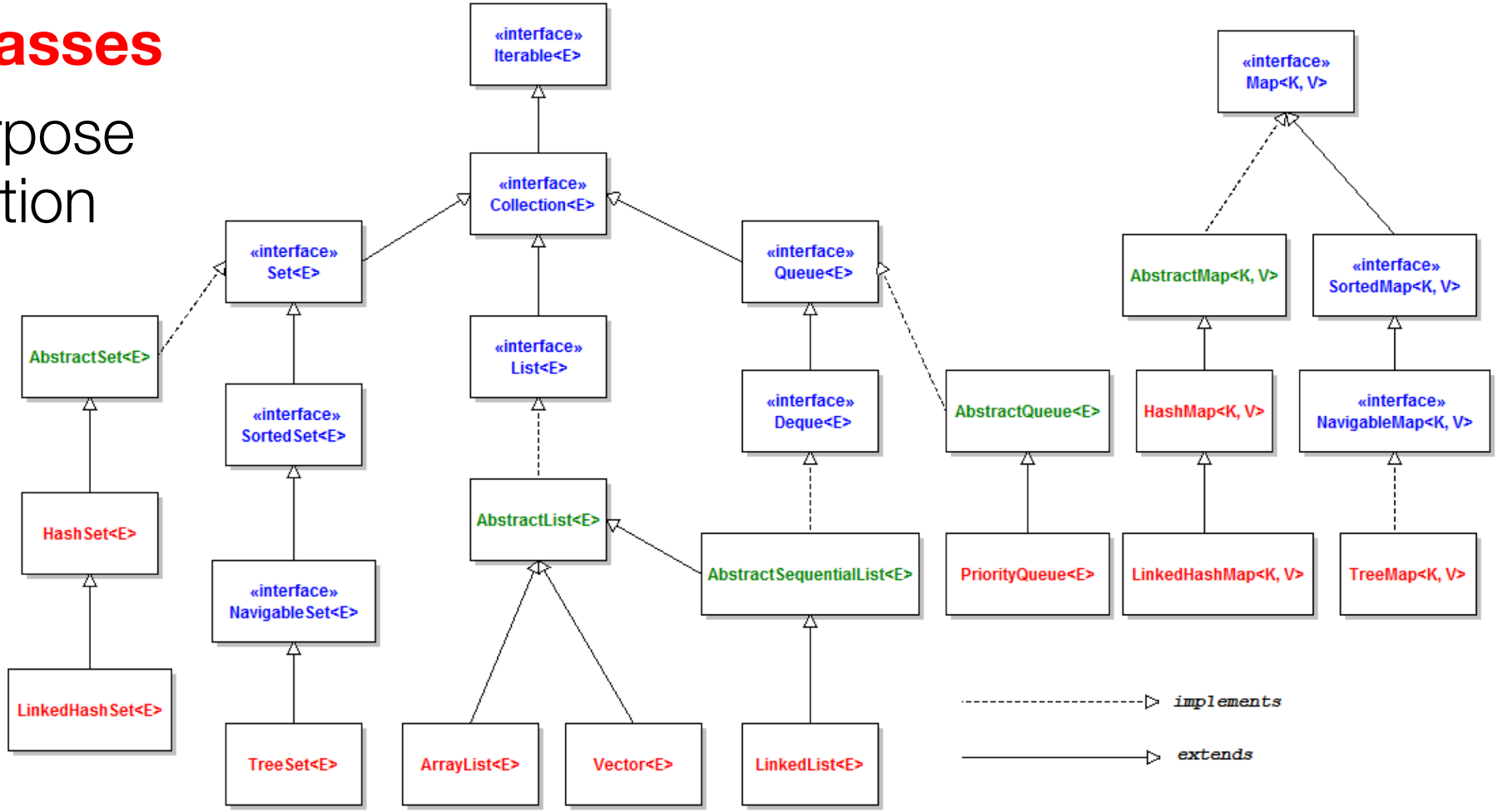


[Picture credit: http://www.codejava.net]

Java Collections API

Concrete classes

- General purpose implementation



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

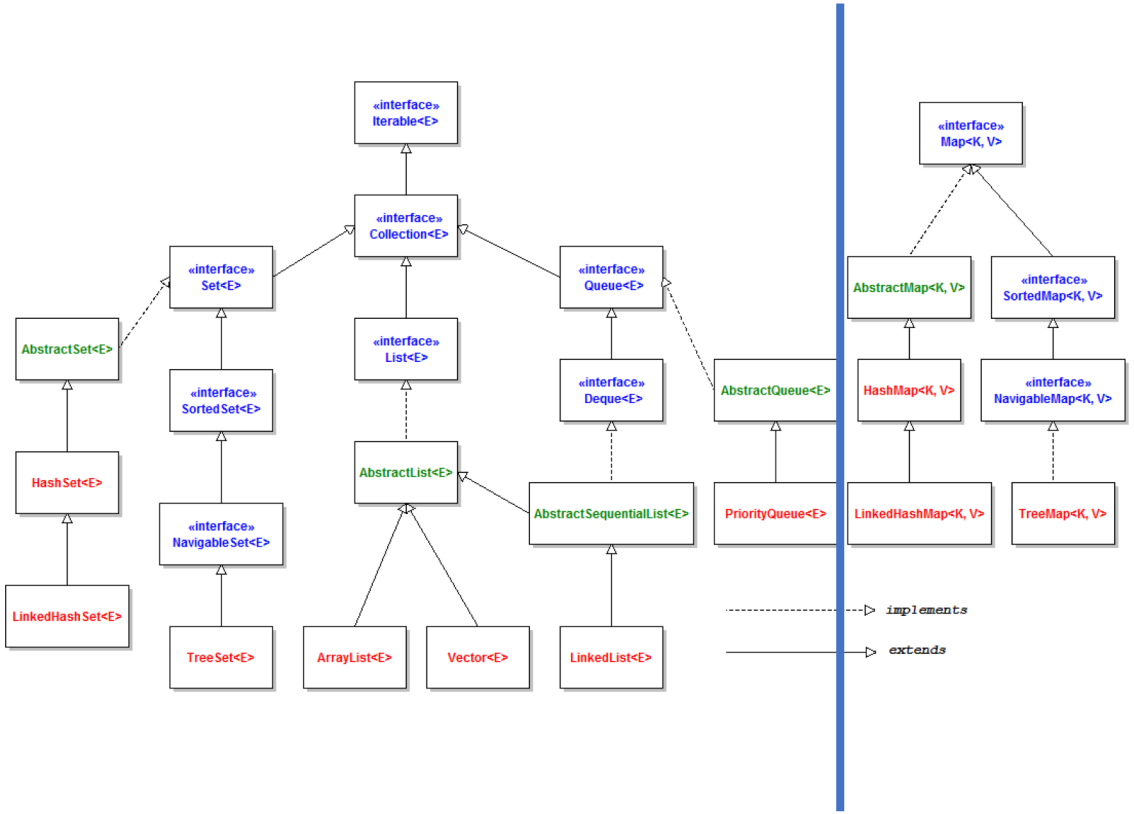
Java Collections Framework - Goals

Provide tested, easy to use, easy to extend implementations of common data structures in order to:

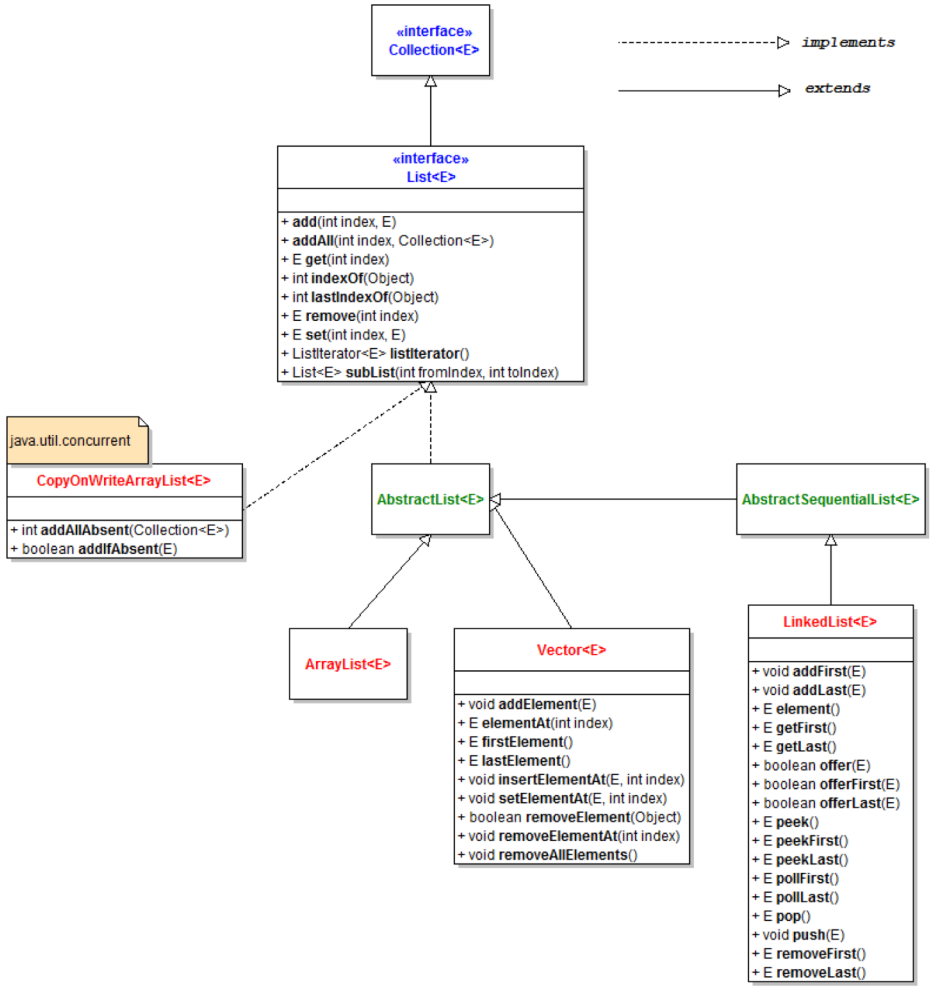
- Reduce programming effort
- Increase flexibility
- Improve program quality

Java Collections Framework

- Part of `java.util` package
- Interface `Collection<E>`
  - root interface on left side
    - Extends interface `Iterable<T>`
      - We'll come back to this...
    - Extended by three interfaces:
      - `List<E>`
      - `Set<E>`
      - `Queue<E>`



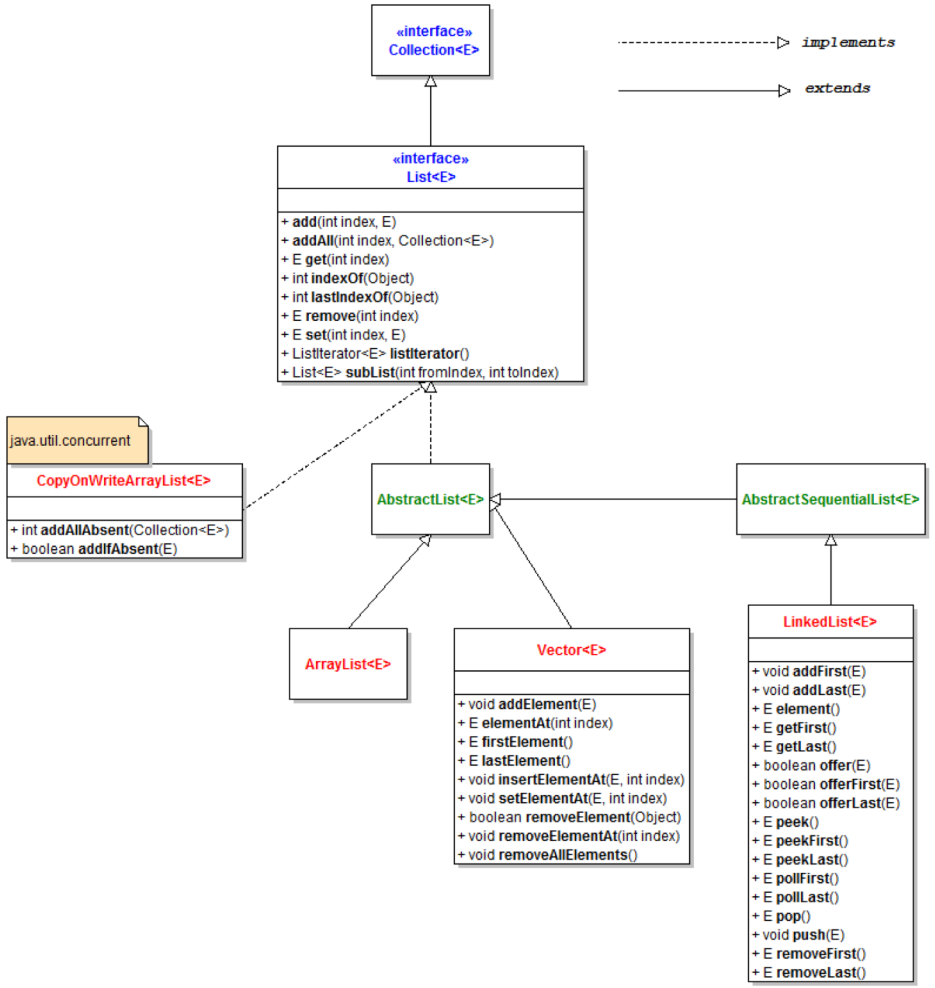
Java's List API



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

- `List<E>` - the base interface
- Abstract subclasses
  - `AbstractList<E>`
  - `AbstractSequentialList<E>`
- Main concrete classes:
  - `ArrayList<E>`
  - `LinkedList<E>`

Java's List API



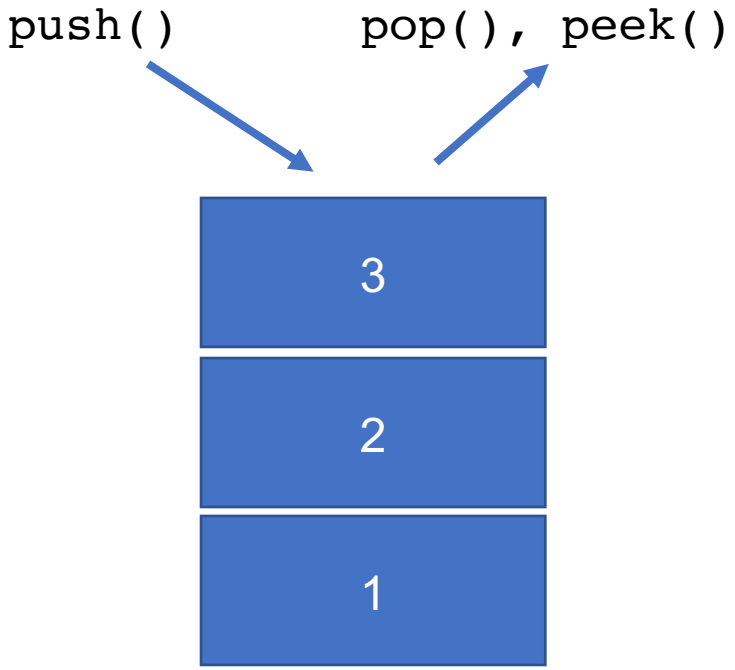
[Picture credit: http://www.codejava.net]

Primary methods:

- `E get(int index);`
- `E set(int index, E newValue);`
- `void add(int index, E x);`
- `void remove(int index);`
- `ListIterator<E> listIterator();`

Java's Stack class

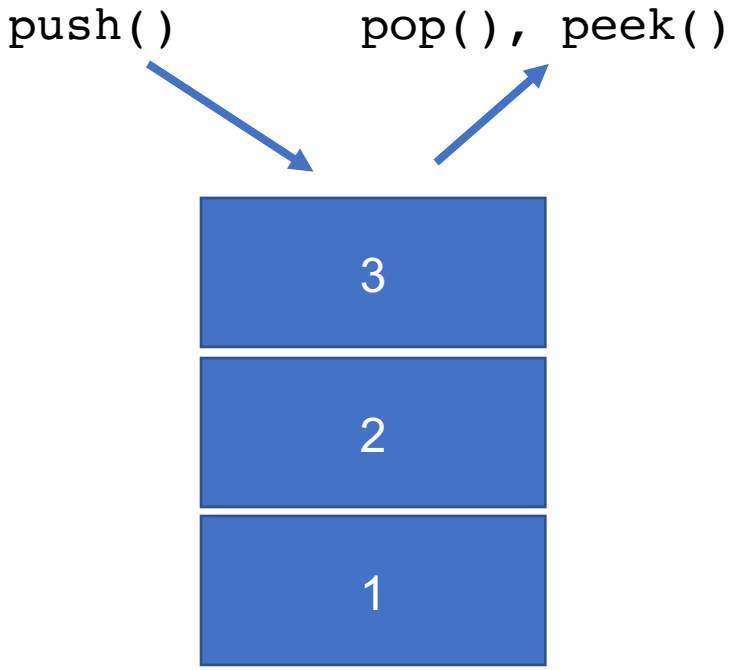
- Stack – LIFO collection
- Inherits List (via other classes)
  - Restricts insertions and deletions to only one position, the top (or end) of the list



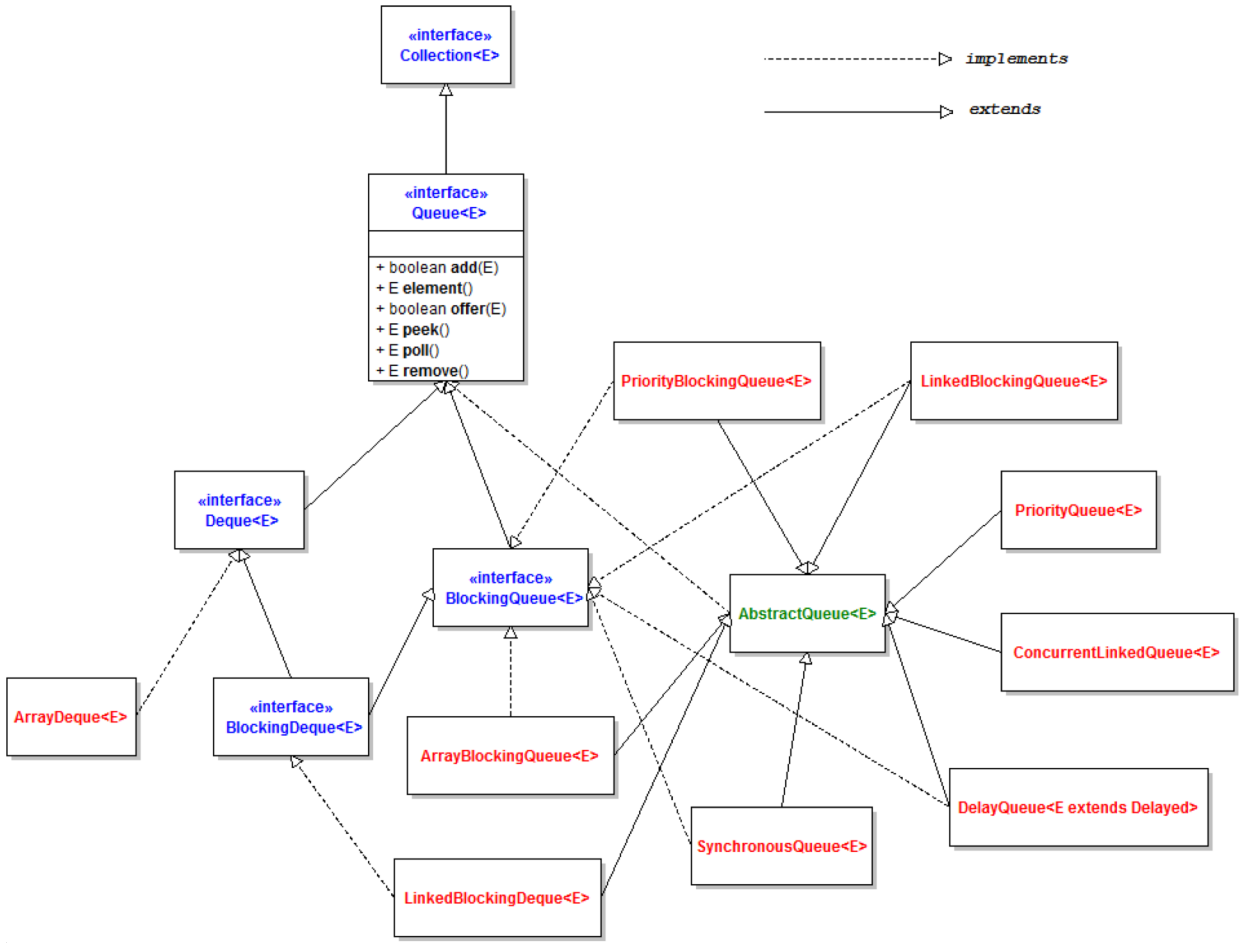
Java's Stack class

Methods:

- `push(E value);`
- `pop();`
- `peek();`
- `size();`
- `isEmpty();`



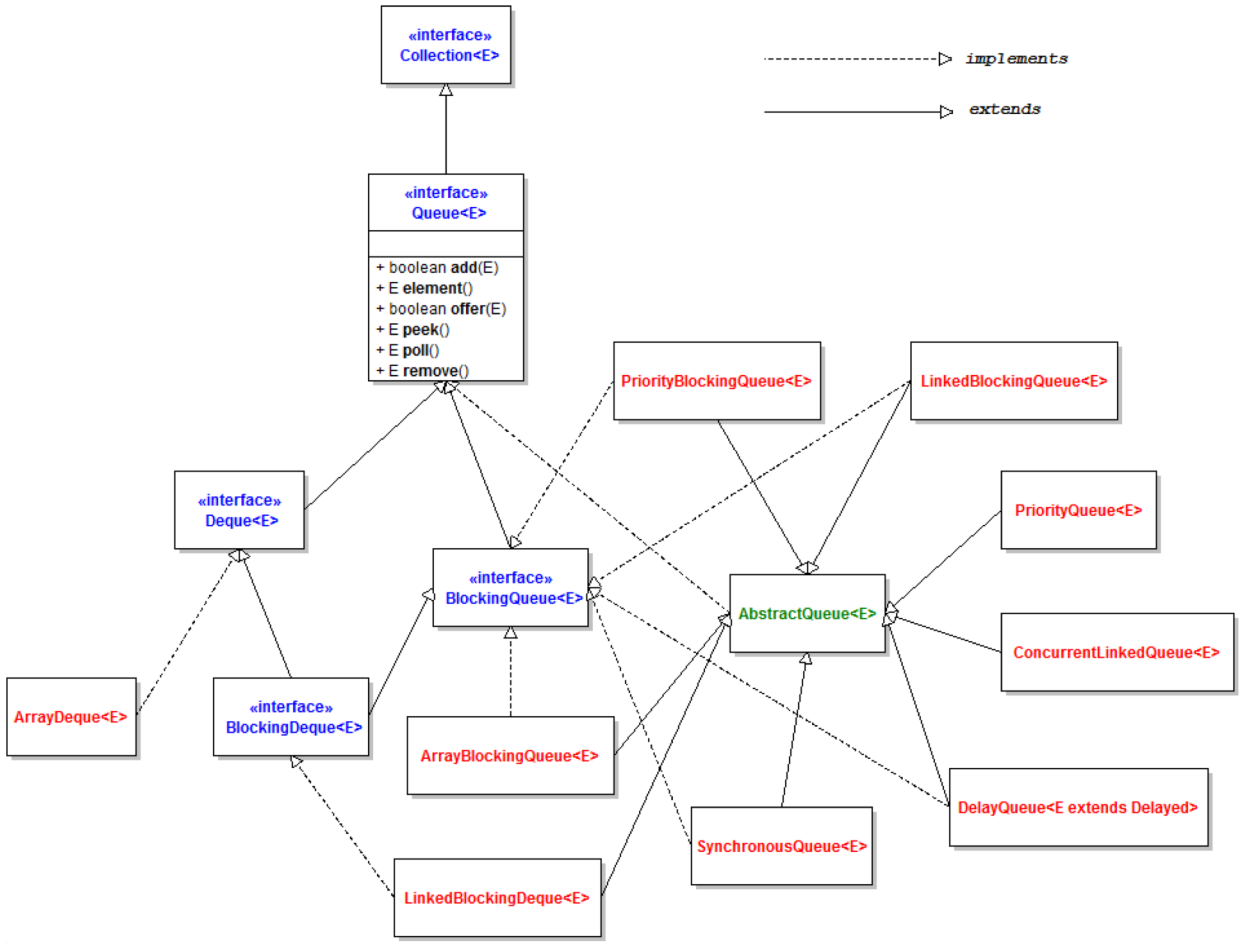
Java's Queue API



- `Queue<E>` - base interface
- Lots of inheriting implementations
- First-in-first-out data structure
  - Add to the back
  - Remove from the front

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

Java's Queue API

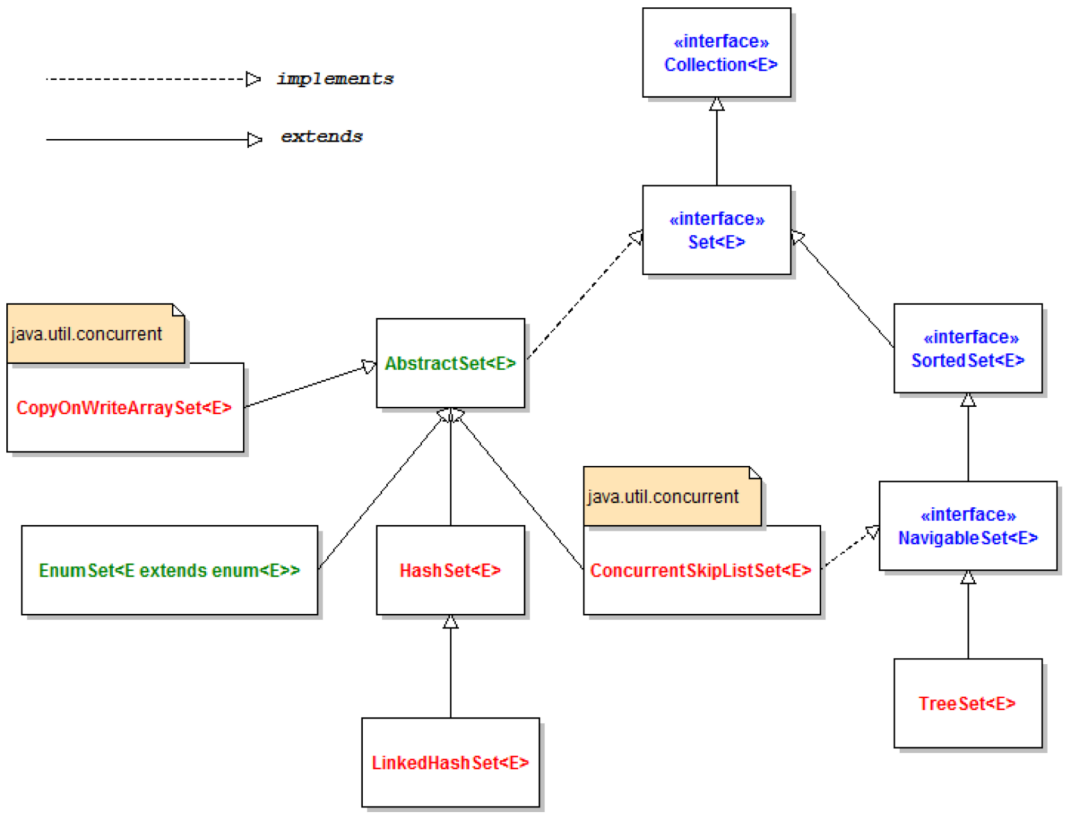


Methods

- `add(E value);`
- `remove();`
- `peek();`
- `size();`
- `isEmpty();`

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

Java's Set API



Set: an unordered collection of unique values

Can perform the following efficiently:

- `add(E value);`
- `remove();`
- `contains(E value);`

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

Java's Set API

- `Set<E>` - base interface
- Implemented by `HashSet` and `TreeSet` classes

Methods

- `add(E value)`
- `contains(E value)`
- `remove(E value)`
- `clear()`
- `size()`
- `isEmpty()`

Java's Set API

`HashSet<E>`

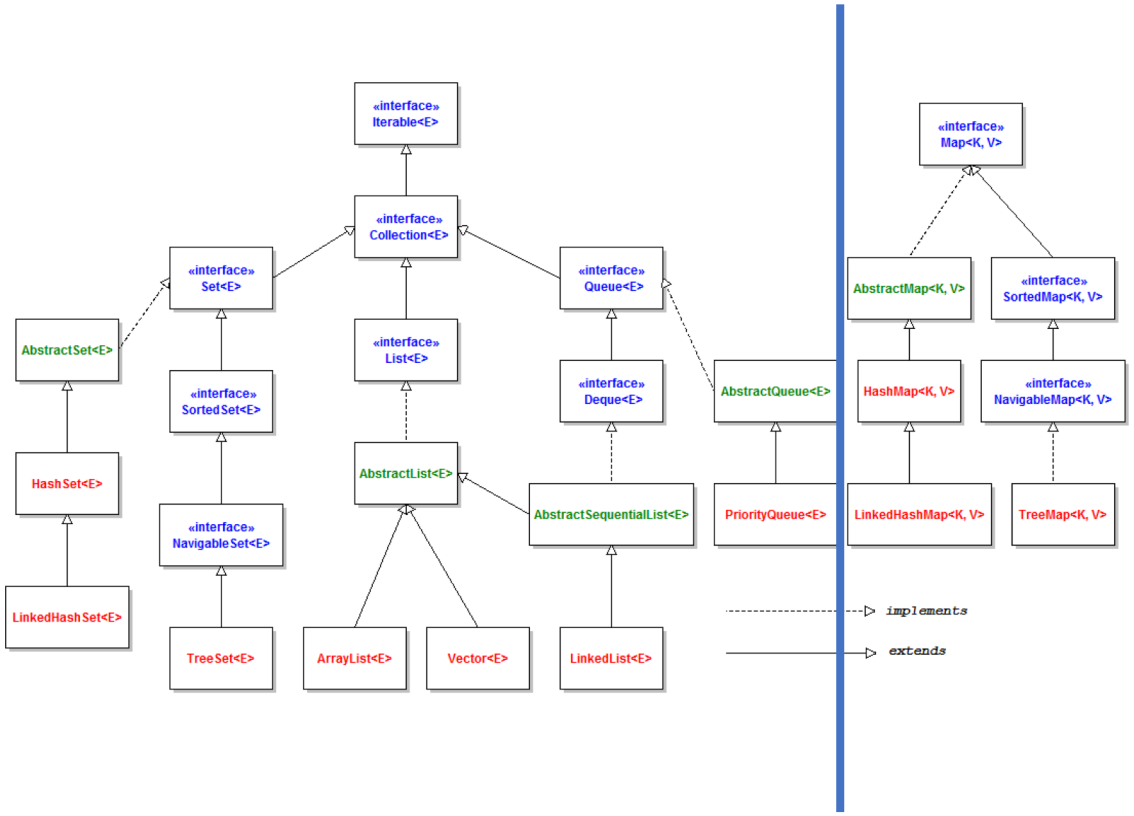
- Implemented using a hash table array
- Very fast: O(1) for all operations
- Elements stored in unpredictable order

`TreeSet<E>`

- Implemented using a **binary search tree**
- Pretty fast: O(log n) for all operations
- Elements stored in sorted order

Maps in Java Collections Framework

- Part of the `java.util` package
- Interface `Map<K,V>` - the root interface on the right side of the hierarchy



Maps

Store a set of unique keys and a collection of values—each key is associated with one value e.g.

- Key – student; Value – assignment grade
- Key – bank account id; Value – account balance
- Key – platform username; Value – user activity

Maps

Also known as:

- Dictionary
- Associative array
- Hash

Basic operations:

- `put(K key, V value)`
- `get(K key)`
- `remove(K key)`

Java's Map API

- `Map<K, V>` - base interface in `java.util`
- Implemented by `HashMap` and `TreeMap` classes

Java's Map API

`HashMap<K, V>`

- Implemented using a hash table
- Extremely fast: O(1)
- Keys stored in unpredictable order

`TreeMap<K, V>`

- Implemented as a binary search tree
- Very fast: O(log N)
- Keys stored in sorted order

`Map<K, V>` implementations

- Interface `Map<K, V>` requires 2 type parameters:
  - K – for keys
  - V – for values

Example: maps from String keys to Integer values

```
Map<String, Integer> ages = new HashMap<String, Integer>();
```

`Map<K, V>` primary methods

- **`put(K key, V value)`** – adds a mapping from the given key to the given value. If the key exists, replace its value with the new value
- **`get(K key)`** – returns the value mapped to the given key (null if not found)
- **`containsKey(K key)`** – returns true if the key is in the map

`Map<K, V>` primary methods

- `remove(K key)` – remove the key and its associated value
- `clear()` – remove all key/value pairs
- `size()` – returns the number of key/value pairs in the map
- `isEmpty()` – returns true if the map's size is 0

Iterating through a `Map<K, V>`

Unlike other collections, Maps are not directly iterable

- Iterate through keys or values instead
- Get all keys as a Set using method **`keySet()`**

Example: Iteration with `keySet()`

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

Example: Iteration with `keySet()`

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

Contains String keys and Integer values

Example: Iteration with `keySet()`

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

Put some key/value pairs into the Map

Example: Iteration with `keySet()`

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

Returns a `Set<String>` containing all the keys in the Map

Example: Iteration with `keySet()`

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

For-each loop iterates through each String key in `ages.keyset()`

Example: Iteration with `keySet()`

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

Get the value mapped to a specific key

`Map<K, V>` - get all values

Method **`values()`** returns a collection of all values in the map

But, no easy way to get from a value to its associated key(s)

Java Collections wrap up

- Suggested reading: Java documentation for built-in collections covered today
- We'll come back to hashing, trees, and graphs

Abstract Data Types

Client and implementer

Client

- A piece of *code* (class, method) that relies on **public** elements of some other class *K*

Implementer

- The code inside *K* that has access to **private** information in *K*.

Client and implementer

Client

- A piece of code (class, method) that relies on **public** elements of some other class *K*
- *CreditCard is a client of Charge (lecture 3 paymentmethods example)*

Implementer

- The code inside *K* that has access to **private** information in *K*.
- *Methods in Charge are implementers*

Client and implementer

So far, you've written client code and implementer code at the same time.

In the real world

- Client code may be written by someone else
- At a later date
- For some specific purpose you don't know about yet

Designing with future clients in mind

- Use interfaces to ensure consistency
- Try to anticipate future clients' needs:
  - What functionality should your code provide?
- Make use of **abstract data types**

Data types vs. abstract data types

Data type

- A set of values
- A set of operations on those values

Abstract data type

- Special data type
- Internal representation of data is hidden from clients

Built-in ADTs in Java

- String
- Integer
- Stack
- etc…

Using an ADT: the API

The ways a client can interact with an ADT are specified in an **Application Programming Interface (API)**

- An API = ”contract” between the ADT and the client
- Guarantees the type of data and operations the ADT will support
- *Does not reveal implementation details*

Example: String ADT

The docs (API):
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html

(Google “Java 11 String”)

Key: The details of the underlying data structure are not revealed.

Creating ADTs in Java

- Use classes
- Separate specification in an interface and/or abstract class
  - Not required but usually a good idea
- Design:
  - How data will be organized
  - What client operations will be permitted

Specifying an ADT

Purpose of a specification:

- Planning
- Documenting
- *No implementation just yet*

Specifying an ADT

Specification must include:

- **Overview**
- Abstract state
- Operation specifications

A description of what the class is for. Also specify whether it's mutable / immutable.

Specifying an ADT

Specification must include:

- Overview
- **Abstract state**
- Operation specifications

Describe key fields without implementation details. E.g. a "Person has a first and last name."

Specifying an ADT

Specification must include:

- Overview
- Abstract state
- **Operation specifications**

Specifications of every operation (i.e. method) to be supported by the ADT.

Specifying an ADT

Operations are categorized into:

- Creators
- Observers
- Producers
- Mutators

Specifying an ADT

Operations are categorized into:

- **Creators**
- Observers
- Producers
- Mutators

Methods that create a new object i.e. constructors

Specifying an ADT

Operations are categorized into:

- Creators
- **Observers**
- Producers
- Mutators

Methods that return values or information about an object *without changing any values*. E.g. getters.

Specifying an ADT

Operations are categorized into:

- Creators
- Observers
- **Producers**
- Mutators

Methods that create and return a new object of the ADT. Mostly for *immutable* ADTs.

Specifying an ADT

Operations are categorized into:

- Creators
- Observers
- Producers
- **Mutators**

Methods that change the value of a field or otherwise modify an ADT object. E.g. setters.
Mutable ADTs only!

Exercise: ADT specification example

Open up Java 11's Color docs:

https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/java/awt/Color.html

(Google "Java 11 Color")

Identify specification components:

- Overview & abstract state
- Creators
- Observers
- Producers – if any
- Mutators – if any

Writing ADT specifications

ADTs are specified from the **client's** point of view:

- What will the client expect this ADT to do?

For each operation, describe:

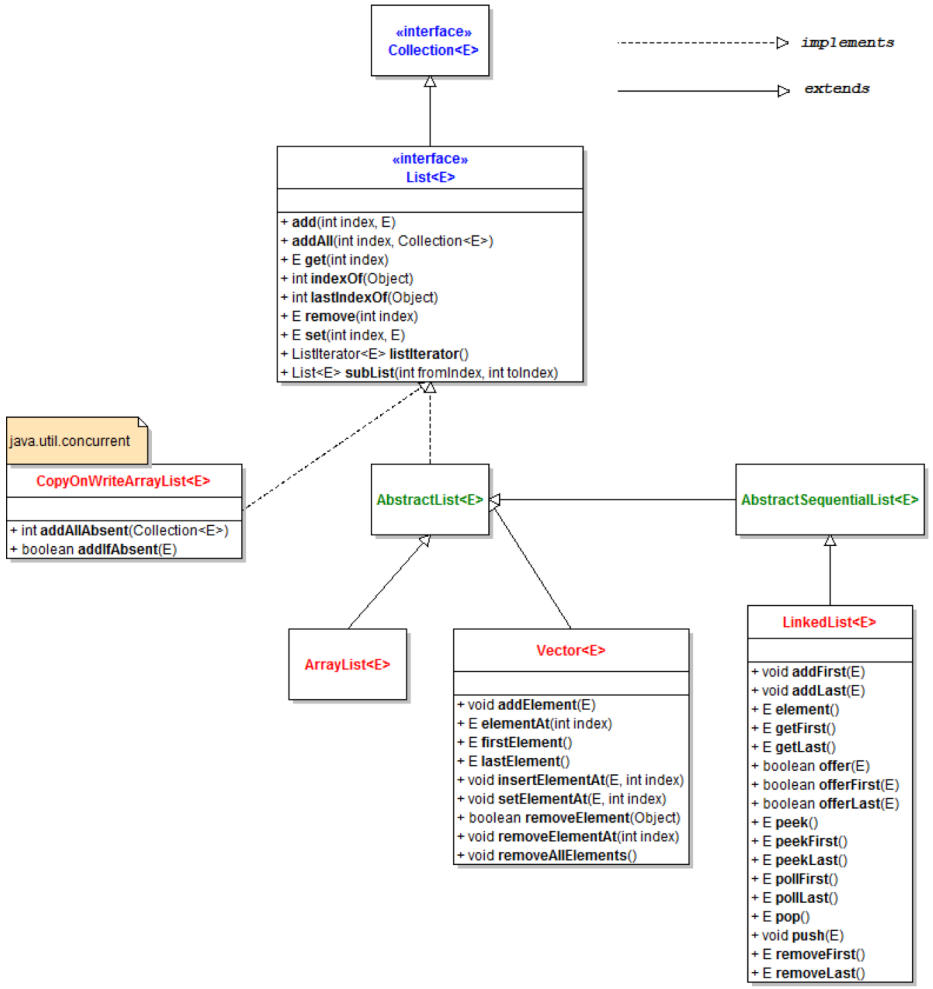
- Expected inputs and outputs.
- Any required conditions (e.g. valid input range)

ADTs and the OOD principles

- Encapsulation
- **Abstraction**
- **Information hiding**
- Polymorphism
- Inheritance

List ADT deep dive

Remember this? Java's List API



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

- `List<E>` - the base interface
- Abstract subclasses
  - `AbstractList<E>`
  - `AbstractSequentialList<E>`
- Main concrete classes:
  - `ArrayList<E>`
  - `LinkedList<E>`

Specifying a List ADT

- Overview
- Abstract state
- Operation specifications

Specifying a List ADT

- **Overview & abstract state**
  A List represents an ordered, mutable collection of objects that have the same data type. A List expands as items are added.
- **Operation specifications**
  - **`create`** an empty list (creator)
  - **`add`** an object to the end of the list (mutator)
  - **`insert`** an item into the list (mutator)
    - Throws IndexOutOfBoundsException.
  - **`get`** an object at a specific index (observer)
    - Throws IndexOutOfBoundsException.
  - get the **`number of items`** in the list (observer)

Implementing an ADT

When the ADT is a ***collection*** of some sort:

- Choose an underlying data structure:
  - An array
  - Something else...
- Write an interface if appropriate
  - Useful if there may be multiple different implementations
  - Example: lecture-code > evening_section > Lecture 4 > adts

Implementing an ADT

When the ADT is a ***collection*** of some sort:

- Choose an underlying data structure:
  - **An array**
  - Something else...
- Write an interface if appropriate
  - Useful if there may be multiple different implementations
  - Example: lecture-code > evening_section > Lecture 4 > adts

Implementing a collection ADT with an array

Considerations:

- Arrays must have a specified size → how big should it be?
- What happens if the array is bigger than the number of items in the collection?
- What happens if the array fills up?

Implementing a collection ADT with an array

```
public class SomeCollection implements ISomeCollection {
  private DataType[] items;
  private int size;
  private int NUM_SLOTS = 10;

  public SomeCollection() {
    this.items = new DataType[NUM_SLOTS];
    this.size = 0;
  }

  // Implement ADT methods here...
}
```

Implementing a collection ADT with an array

```
public class SomeCollection implements ISomeCollection {
  private DataType[] items;
  private int size;
  private int NUM_SLOTS = 10;

  public SomeCollection() {
    this.items = new DataType[NUM_SLOTS];
    this.size = 0;
  }

  // Implement ADT methods here...
}
```

An array to store the items in the collection.

- Replace `DataType` with the appropriate data type...

Implementing a collection ADT with an array

```
public class SomeCollection implements ISomeCollection {
  private DataType[] items;
  private int size;
  private int NUM_SLOTS = 10;

  public SomeCollection() {
    this.items = new DataType[NUM_SLOTS];
    this.size = 0;
  }

  // Implement ADT methods here...
}
```

The number of items in the collection

- Not necessarily the same as `items.length`

Implementing a collection ADT with an array

```
public class SomeCollection implements ISomeCollection {
  private DataType[] items;
  private int size;
  private int NUM_SLOTS = 10;

  public SomeCollection() {
    this.items = new DataType[NUM_SLOTS];
    this.size = 0;
  }

  // Implement ADT methods here...
}
```

The initial length of the `items` array

- Also used when resizing

ListOfStrings with an array: Instantiating

```
ListOfStrings list = new ListOfStrings();
```

Underlying data structure:

- String array, length 5

0	1	2	3	4
null	null	null	null	null

Client view:

- An empty list
  - No items
  - Size 0

ListOfStrings with an array: Adding

E.g. `list.add("A");`

Underlying data structure:

- String array, length 5

0	1	2	3	4
"A"	null	null	null	null

Client view:

- In the list: "A"
- Size 1

Check that `this.size < this.items.length`
→ `this.items[this.size] = newItem`
→ `this.size++` (`this.size` is now 1)

ListOfStrings with an array: Adding

E.g. `list.add("B");`

Underlying data structure:

- String array, length 5

0	1	2	3	4
"A"	"B"	null	null	null

Client view:

- In the list: "A", "B"
- Size 2

Check that `this.size < this.items.length`
→ `this.items[this.size] = newItem`
→ `this.size++` (`this.size` is now 2)

ListOfStrings with an array: Adding

E.g. `list.add("C");`

Underlying data structure:

- String array, length 5

0	1	2	3	4
"A"	"B"	"C"	null	null

Client view:

- In the list: "A", "B", "C"
- Size 3

Check that `this.size < this.items.length`
→ `this.items[this.size] = newItem`
→ `this.size++` (`this.size` is now 3)

ListOfStrings with an array: Adding

E.g. `list.add("D");`

Underlying data structure:

- String array, length 5

0	1	2	3	4
"A"	"B"	"C"	"D"	null

Client view:

- In the list: "A", "B", "C", "D"
- Size 4

Check that `this.size < this.items.length`
→ `this.items[this.size] = newItem`
→ `this.size++` (`this.size` is now 4)

ListOfStrings with an array: Adding

E.g. `list.add("E");`

Underlying data structure:

- String array, length 5

0	1	2	3	4
"A"	"B"	"C"	"D"	"E"

Client view:

- In the list: "A", "B", "C", "D", "E"
- Size 5

Check that `this.size < this.items.length`
→ `this.items[this.size] = newItem`
→ `this.size++` (`this.size` is now 5)

ListOfStrings with an array: Adding

E.g. `list.add("F");`

Underlying data structure:

- String array, length 5

0	1	2	3	4
"A"	"B"	"C"	"D"	"E"

Client view:

- In the list: "A", "B", "C", "D", "E"
- Size 5

Check that `this.size < this.items.length` → false

ListOfStrings with an array: Adding

E.g. `list.add("F");`

Underlying data structure:

- String array, length 5

0	1	2	3	4
"A"	"B"	"C"	"D"	"E"

Client view:

- In the list: "A", "B", "C", "D", "E"
- Size 5

Check that `this.size < this.items.length` → false

- Make a new array of size `this.size + NUM_SLOTS`

0	1	2	3	4	5	6	7	8	9
null	null	null	null	null	null	null	null	null	null

ListOfStrings with an array: Adding

E.g. `list.add("F");`

Underlying data structure:

- String array, length 10

0	1	2	3	4
"A"	"B"	"C"	"D"	"E"

Client view:

- In the list: "A", "B", "C", "D", "E"
- Size 5

Check that `this.size < this.items.length` → false

- Make a new array of size `this.size + NUM_SLOTS`

0	1	2	3	4	5	6	7	8	9
"A"	"B"	"C"	"D"	"E"	null	null	null	null	null

- Copy the contents of `this.items`, set `this.items` to the new array

ListOfStrings with an array: Adding

E.g. `list.add("F");`

Underlying data structure:

- String array, length 10

Client view:

- "A", "B", "C", "D", "E", "F"
- Size 6

0	1	2	3	4	5	6	7	8	9
"A"	"B"	"C"	"D"	"E"	"F"	null	null	null	null

Walkthrough: List ADT implementation

...with an array as the underlying data structure

GitHub > lecture-code > Evening > Lecture4 > adts > ListOfStrings

- Constructor
- The **`add`** ADT method

Aside: choosing the underlying array size

```
private int NUM_SLOTS = 10;
...
this.items = new String[NUM_SLOTS];
```

Tradeoff:

- Arrays with a lot of empty slots waste space.
- When the array fills, resizing takes time and space.

ListOfStrings with an array: Get an item

E.g. `list.get(0);`

Underlying data structure:

- String array, length 10

Client view:

- “A”, “B”, “C”, “D”, “E”, “F”
- Index 0 should return “A”

0	1	2	3	4	5	6	7	8	9
"A"	“B”	“C”	“D”	“E”	"F"	null	null	null	null

Check that `index >= 0 && index < this.size`

→ `return this.items[index]`

ListOfStrings with an array: Get an item

E.g. `list.get(6);`

Underlying data structure:

- String array, length 10

Client view:

- “A”, “B”, “C”, “D”, “E”, “F”
- Index 6 is out of bounds

0	1	2	3	4	5	6	7	8	9
"A"	“B”	“C”	“D”	“E”	"F"	null	null	null	null

Check that `index >= 0 && index < this.size`

→ `false` so `throw new IndexOutOfBoundsException()`

Break (10 mins)

Exercise: implement `insert`

- We'll walk through the logic before you start coding...

ListOfStrings with an array: Insert at index

E.g. `list.insert("X", 3)`

"X"

0	1	2	3	4	5	6	7	8	9
"A"	"B"	"C"	"D"	"E"	"F"	null	null	null	null

Create a new array with the same* length (or resize if too small)

0	1	2	3	4	5	6	7	8	9
null	null	null	null	null	null	null	null	null	null

ListOfStrings with an array: Insert at index

E.g. `list.insert("X", 3)`

0	1	2	3	4	5	6	7	8	9
"A"	"B"	"C"	"D"	"E"	"F"	null	null	null	null

Copy items **before** the insert index

0	1	2	3	4	5	6	7	8	9
"A"	"B"	"C"	null	null	null	null	null	null	null

ListOfStrings with an array: Insert at index

E.g. `list.insert("X", 3)`

0	1	2	3	4	5	6	7	8	9
"A"	"B"	"C"	"D"	"E"	"F"	null	null	null	null

Add the new item at the given index

0	1	2	3	4	5	6	7	8	9
"A"	"B"	"C"	"X"	null	null	null	null	null	null

ListOfStrings with an array: Insert at index

E.g. `list.insert("X", 3)`

0	1	2	3	4	5	6	7	8	9
"A"	"B"	"C"	"D"	"E"	"F"	null	null	null	null

Copy the remaining items

0	1	2	3	4	5	6	7	8	9
"A"	"B"	"C"	"X"	"D"	"E"	"F"	null	null	null

Don't forget to increase `this.size`!

Exercise: implement `insert`

- Test cases are provided
- We'll look at a solution afterwards

Implementing an ADT

When the ADT is a ***collection*** of some sort:

- Choose an underlying data structure:
  - An array
  - **Something else...**
- Write an interface if appropriate
  - Useful if there may be multiple different implementations
  - Example: lecture-code > evening_section > Lecture 4 > adts

Collection ADT implementation – underlying data structure

Array

- Pro: Built-in data structure
- Con: Fixed size
  - What do we do when we run out of slots

Something else?

- A custom data structure
- …can we use something that doesn't need resizing?

Introducing the linked list

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

Introducing the linked list

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

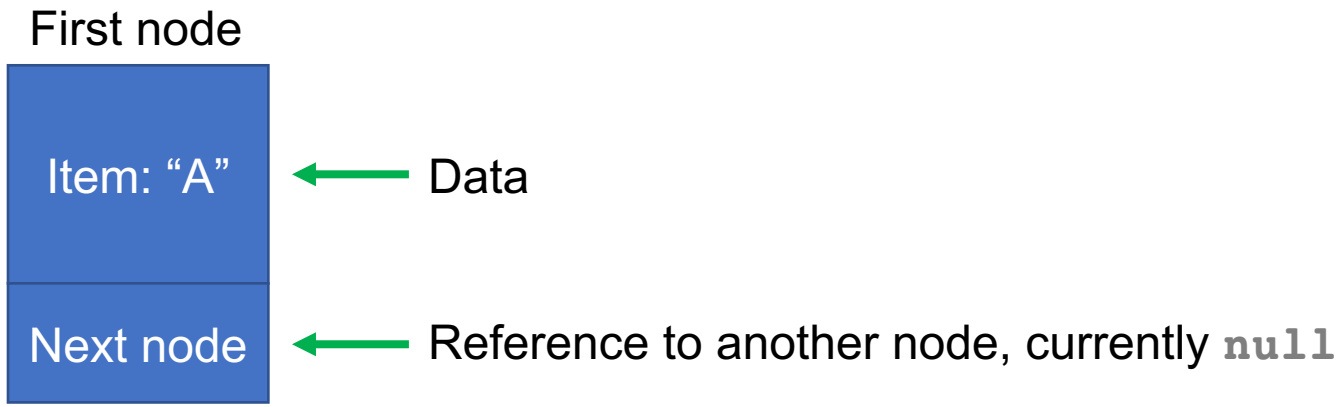
- Items in the list are commonly referred to as “nodes”

Two ways to implement:

- Sequential (today)
- Recursive (a couple of weeks)

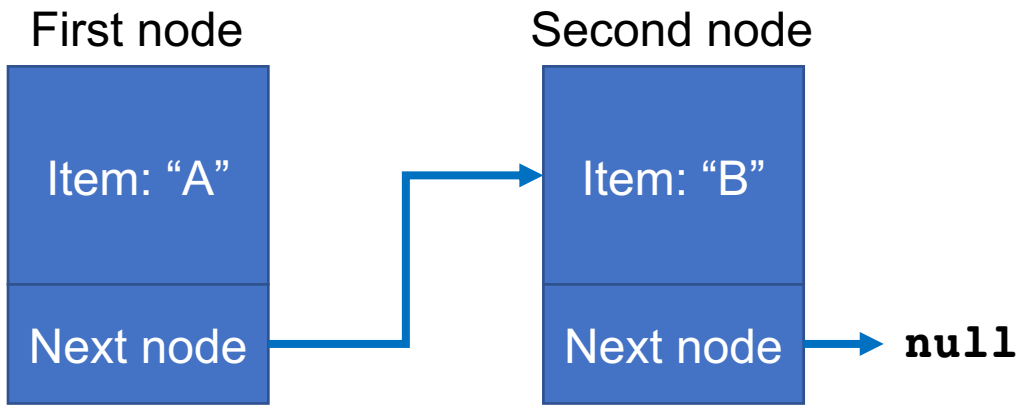
Introducing the linked list

We start with a single node, contains some data and a reference to another node.



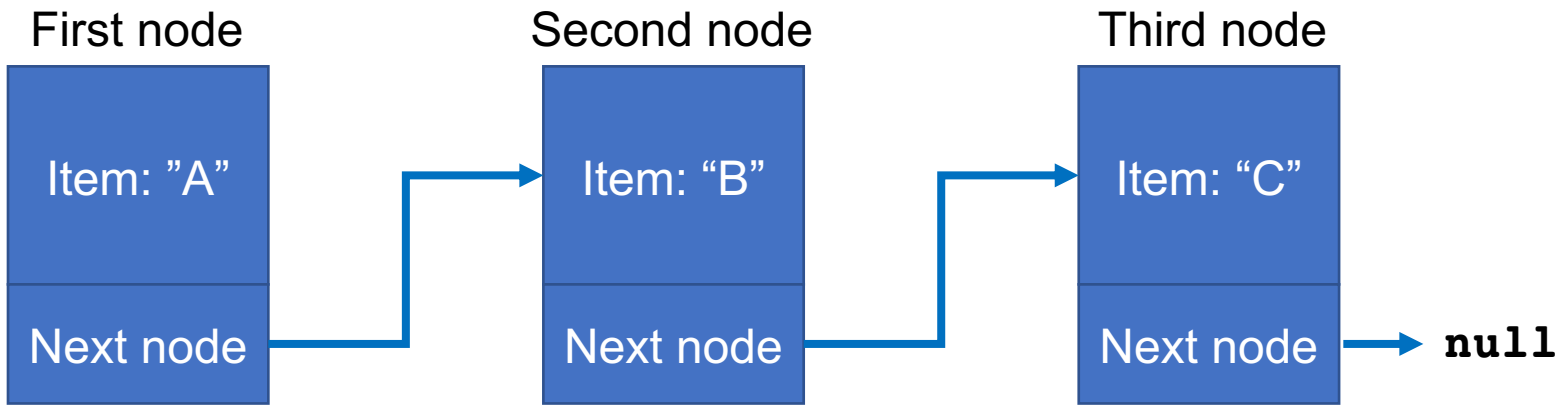
Introducing the linked list

Adding a second node…



Introducing the linked list

Adding a third node…



Implementing a collection ADT with a linked list

```
public class Node {
  private DataType item;
  private Node nextNode;

  public Node(DataType item, Node nextNode) {
    this.item = item;
    this.nextNode = nextNode;
  }
  // Implement getters, setters, equals, hashCode, toString
}
```

Implementing a collection ADT with a linked list

```
public class SomeCollection implements ISomeCollection {
  private Node head;
  private int numNodes;

  public SomeCollection() {
    this.head = null;
    this.numNodes = 0;
  }
  // Implement ADT methods here...
}
```

ListOfStrings with a linked list: Instantiating

```
ListOfStrings list = new ListOfStrings();
```

Underlying data structure:

- `this.numNodes = 0;`
- `this.head = null;`

Client view:

- An empty list
  - No items
  - Size 0

ListOfStrings with a linked list: Add item

E.g. `list.add("A");`

Underlying data structure:

- `this.numNodes = 0;`

- `this.head = null;`

  If the head is null → make a new Node containing the item

  ```
  Node newNode = new Node(item, null);
  ```

  ...and set `this.head` to be the new `Node`

  ```
  this.head = newNode;
  ```

Client view:

- An empty list
  - No items
  - Size 0

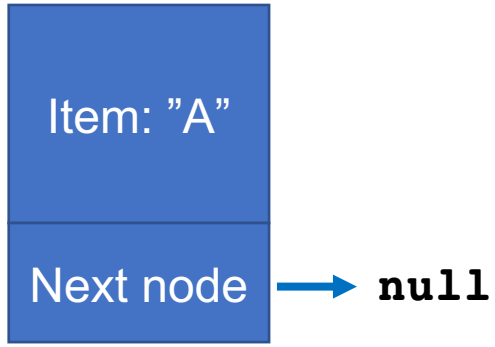
ListOfStrings with a linked list: Add item

E.g. `list.add("A");`

Underlying data structure:

- `this.numNodes++;` (now 1)

- `this.head =`



Client view:

- List contents: "A"
- Size 1

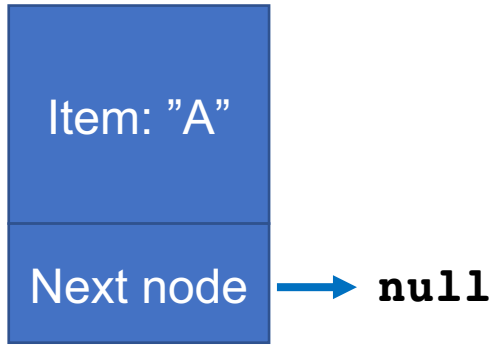
ListOfStrings with a linked list: Add item

E.g. `list.add("B");`

Underlying data structure:

- `this.numNodes = 1;`

- `this.head =`



Client view:

- List contents: "A"
- Size 1

If the head is NOT null → make a new Node containing the item

- Starting from `this.head`, check each Node's `nextNode` until you find one that's `null`.
- Set the Node's `nextNode` to point to the new Node

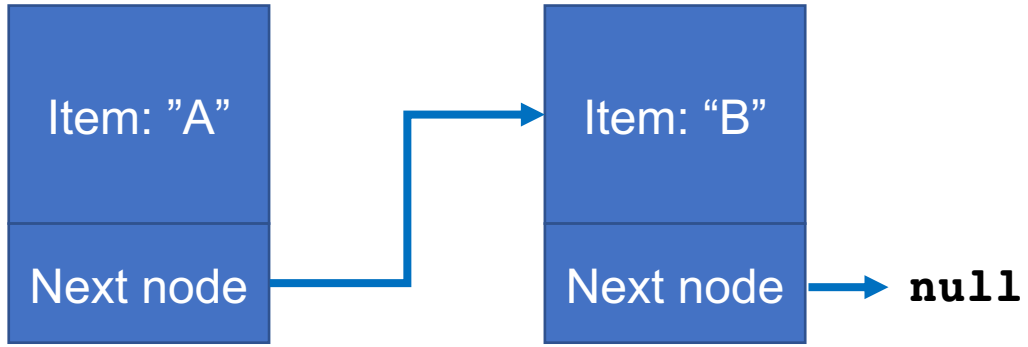
ListOfStrings with a linked list: Add item

E.g. `list.add("B");`

Underlying data structure:

- `this.numNodes++;` (now 2)

- `this.head =`



Client view:

- List contents: "A", "B"
- Size 2

Walkthrough: List ADT implementation

...with a linked list as the underlying data structure

GitHub > lecture-code > Lecture4 > adts > LinkedListOfStrings

- Constructor
- The **`add`** ADT method

ListOfStrings with a linked list: Get item

E.g. `list.get(0);`

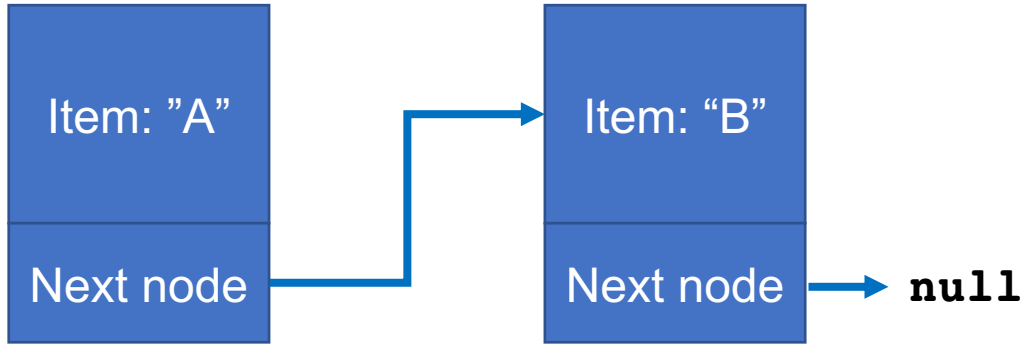
Underlying data structure:

- `this.numNodes = 2;`

Client view:

- Expected return: “A”

- `this.head =`



ListOfStrings with a linked list: Get item

E.g. `list.get(0);`

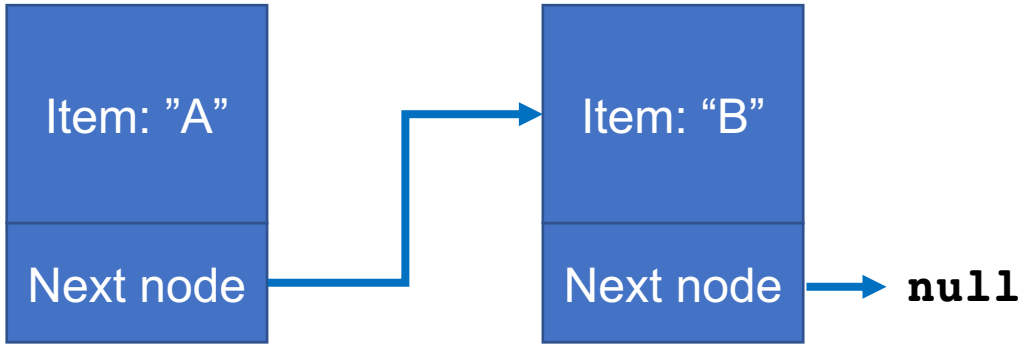
Underlying data structure:

- `this.numNodes = 2;`

Client view:

- Expected return: “A”

- `this.head =`



Check that `index >= 0 && index < this.numNodes`

→ Use a loop to step through each Node

ListOfStrings with a linked list: Get item

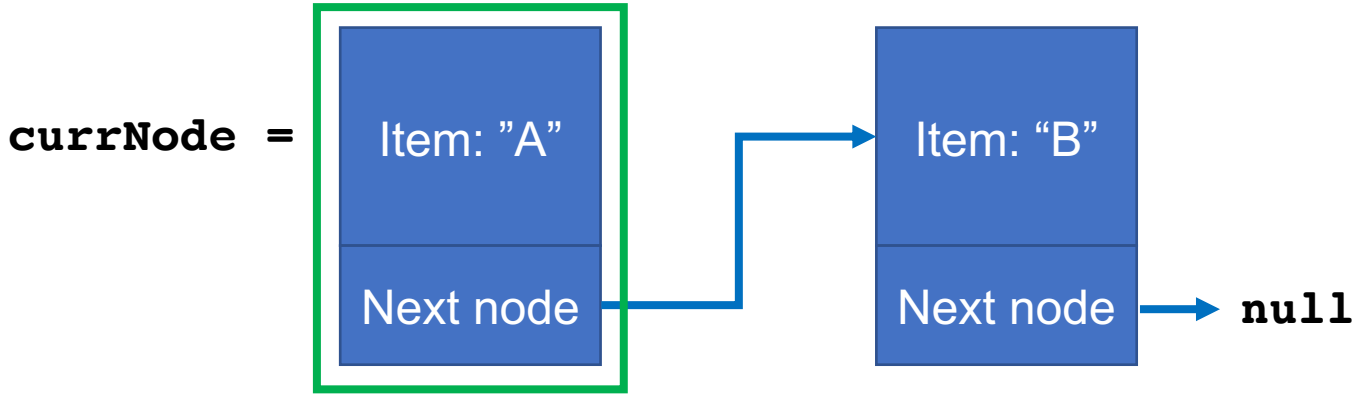
E.g. `list.get(0);`

```
Node currNode = this.head;

int i = 0;
while (i < index) {
    i++;
    currNode = currNode.getNextNode();
}
return currNode.getItem();
```

ListOfStrings with a linked list: Get item

E.g. `list.get(0);`

```
Node currNode = this.head;
int i = 0;
while (i < index) {
    i++;
    currNode = currNode.getNextNode();
}
return currNode.getItem();
```

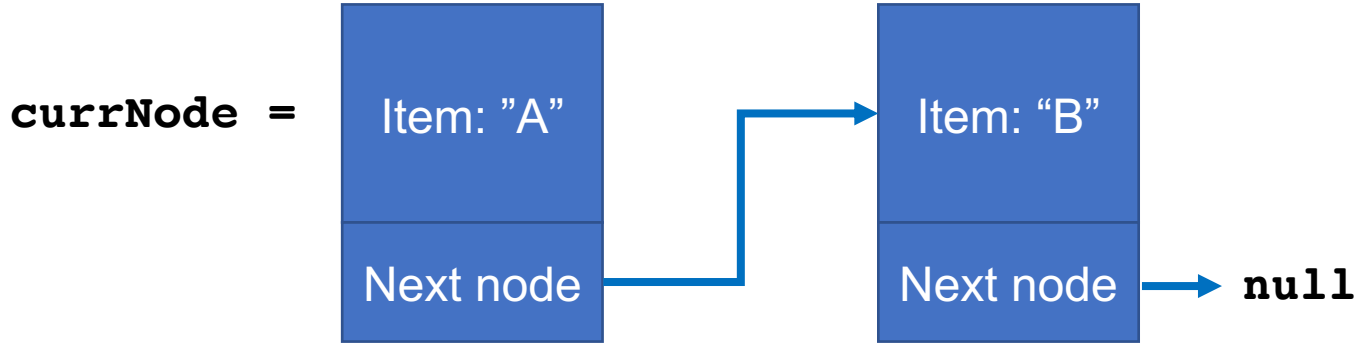


ListOfStrings with a linked list: Get item

E.g. `list.get(0);`

```
Node currNode = this.head;
int i = 0;
while (i < index) {
    i++;
    currNode = currNode.getNextNode();
}
return currNode.getItem();
```

Because the index is 0, the while loop is skipped

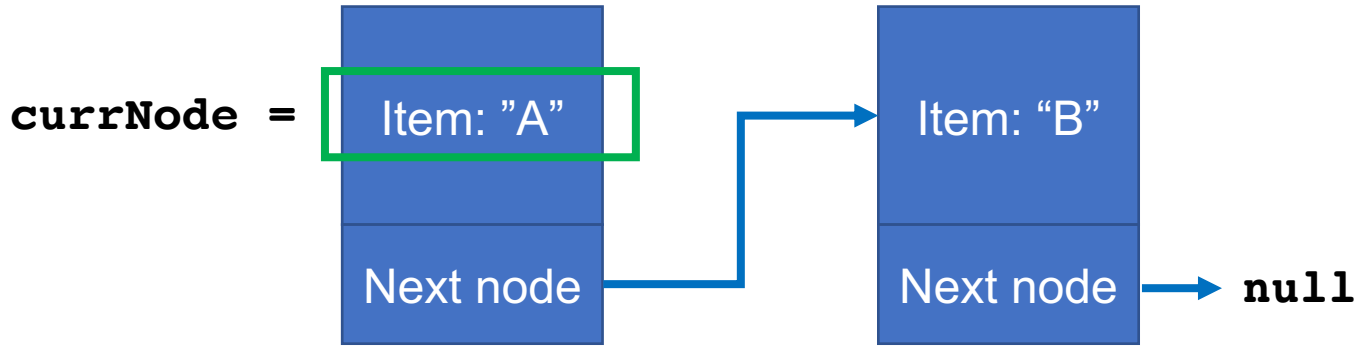


ListOfStrings with a linked list: Get item

E.g. `list.get(0);`

```
Node currNode = this.head;
int i = 0;
while (i < index) {
    i++;
    currNode = currNode.getNextNode();
}
return currNode.getItem();
```

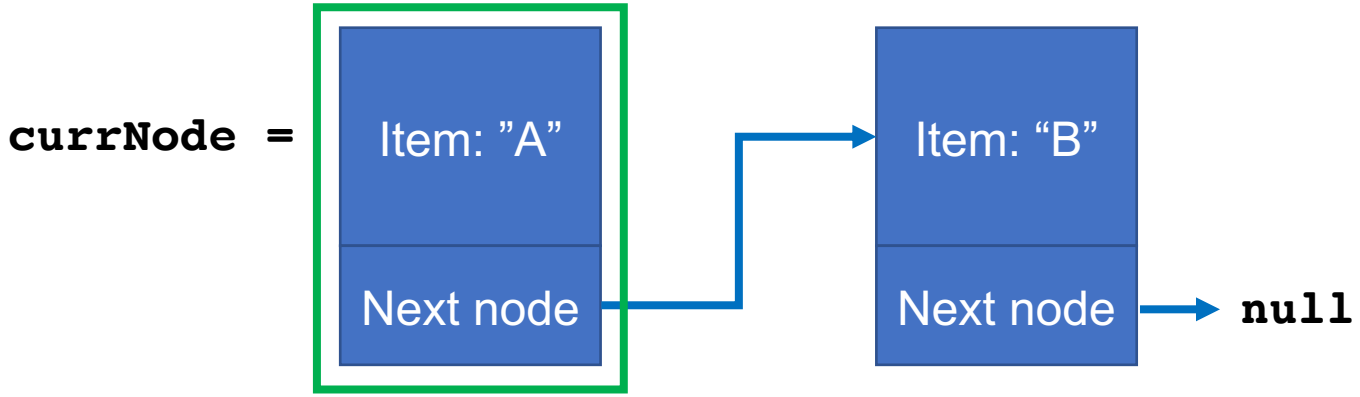
Return the item stored in the current node



ListOfStrings with a linked list: Get item

E.g. `list.get(1);`

```
Node currNode = this.head;
int i = 0;
while (i < index) {
    i++;
    currNode = currNode.getNextNode();
}
return currNode.getItem();
```

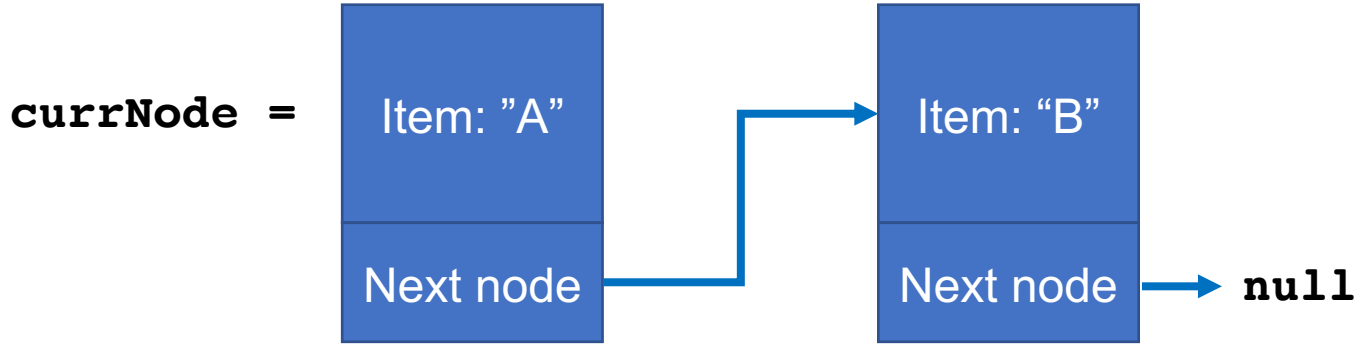


ListOfStrings with a linked list: Get item

E.g. `list.get(1);`

```
Node currNode = this.head;
int i = 0;
while (i < index) {
    i++;
    currNode = currNode.getNextNode();
}
return currNode.getItem();
```

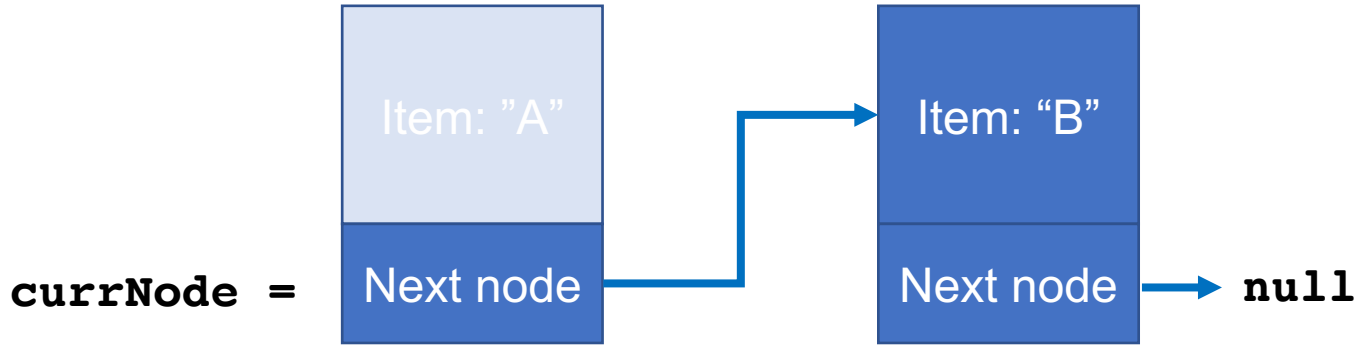
i < index →
Step into the while loop



ListOfStrings with a linked list: Get item

E.g. `list.get(1);`

```
Node currNode = this.head;

int i = 0;
while (i < index) {
    i++;
    currNode = currNode.getNextNode();
}
return currNode.getItem();
```

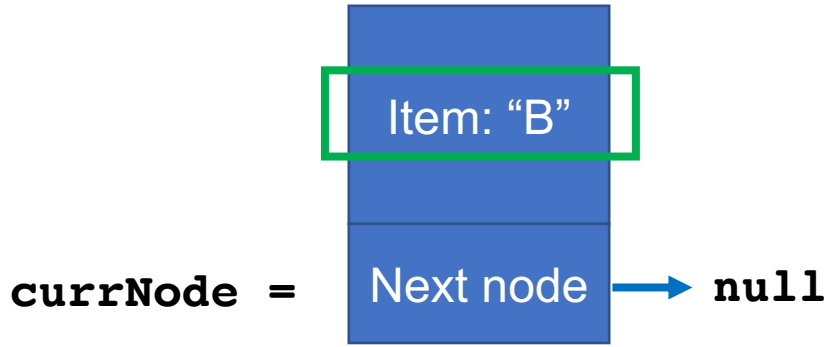


ListOfStrings with a linked list: Get item

E.g. `list.get(1);`

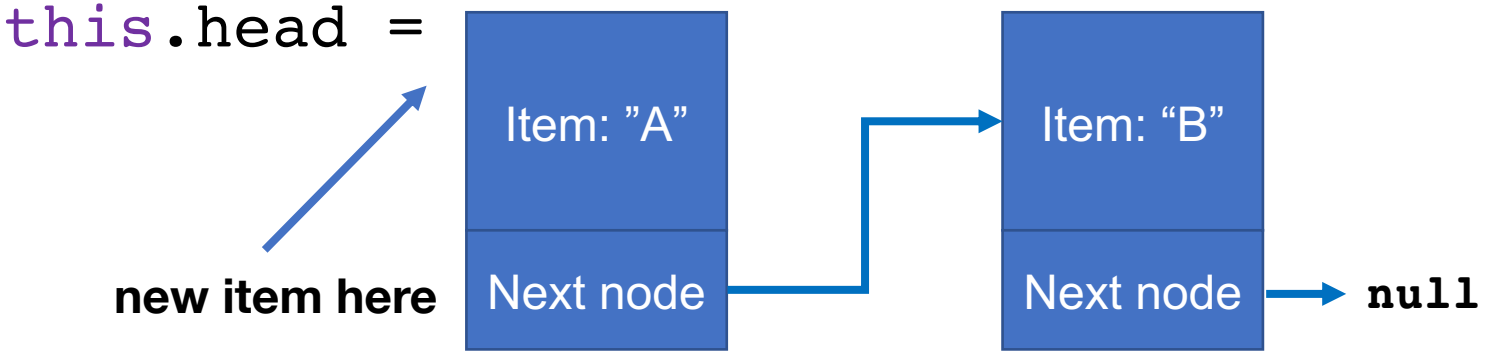
```
Node currNode = this.head;

int i = 0;
while (i < index) {
    i++;
    currNode = currNode.getNextNode();
}
return currNode.getItem();
```

Return the item stored in the current node



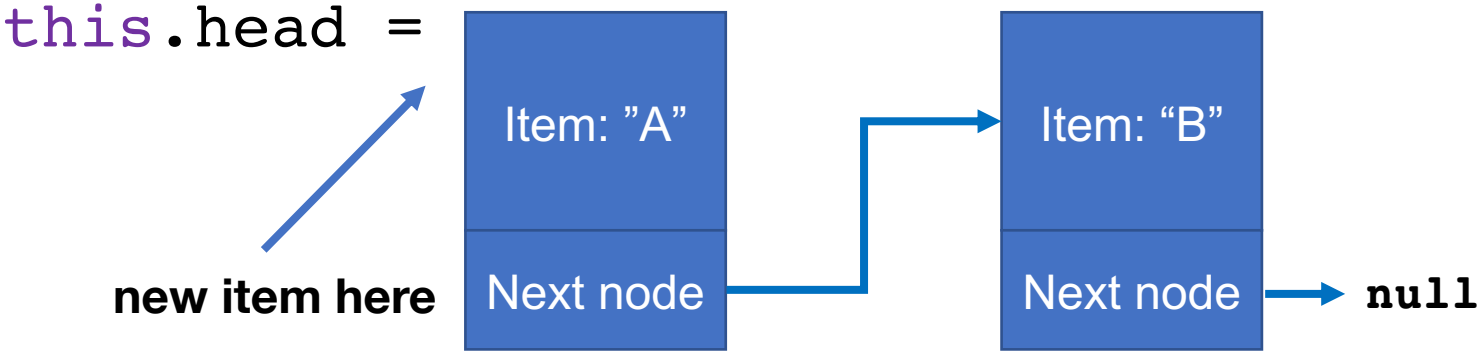
ListOfStrings with a linked list: Insert item

E.g. `list.insert("X", 0);`



ListOfStrings with a linked list: Insert item

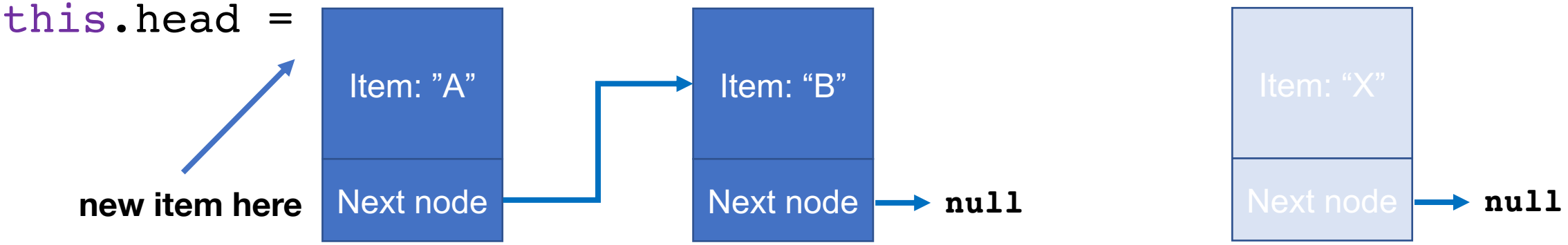
E.g. `list.insert("X", 0);`



Check that `index >= 0 && index < this.numNodes`

ListOfStrings with a linked list: Insert item

E.g. `list.insert("X", 0);`



Check that `index >= 0 && index < this.numNodes`

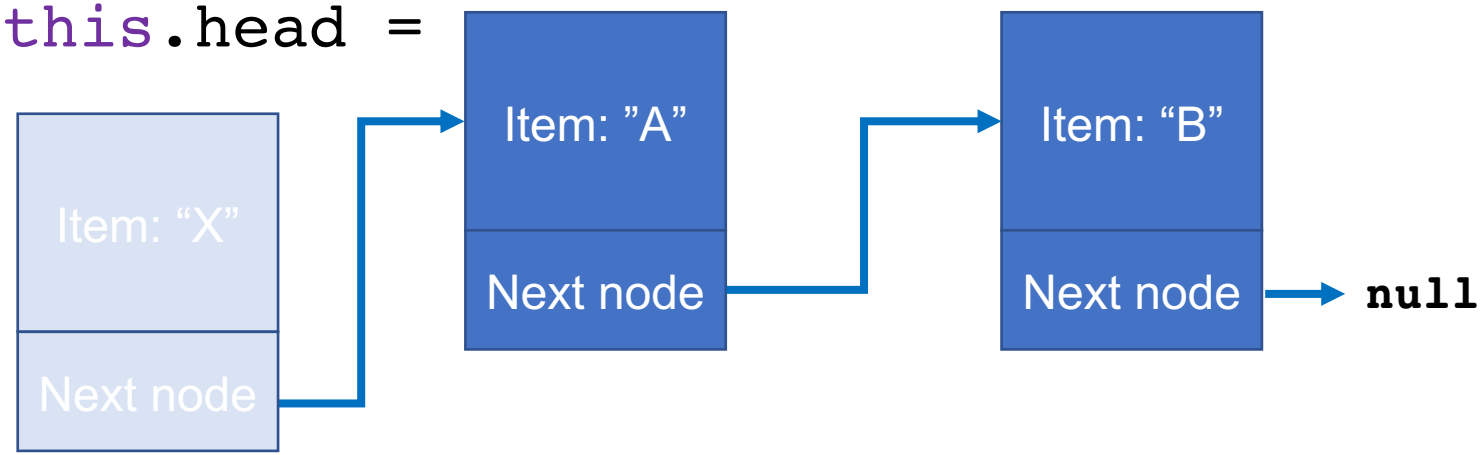
→ Create a new Node containing the item:

```
Node newNode = new Node("X", null);
```

ListOfStrings with a linked list: Insert item

E.g. `list.insert("X", 0);`

`this.head =`



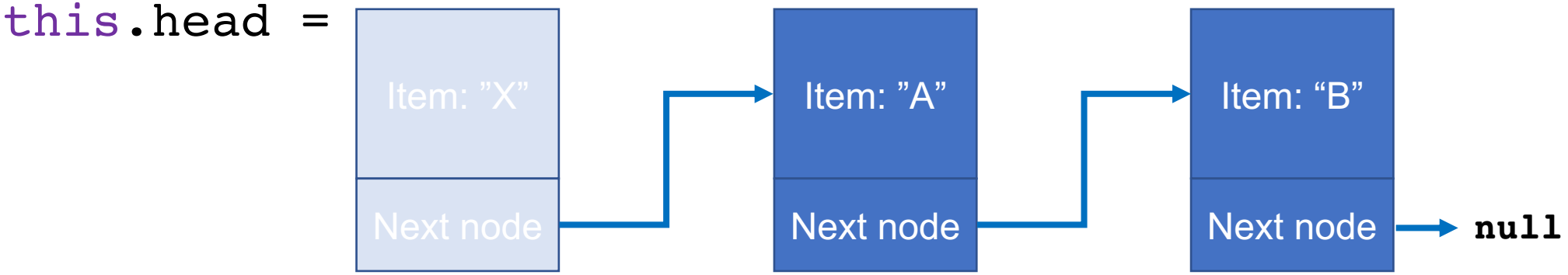
If `index == 0`

→ Set the new Node's **`nextNode`** to **`this.head`**;

`newNode.setNextNode(this.head);`

ListOfStrings with a linked list: Insert item

E.g. `list.insert("X", 0);`



If `index == 0`

→ Set the new Node's **`nextNode`** to **`this.head`**;

```
newNode.setNextNode(this.head);
```

→ Set **`this.head`** to **`nextNode`**

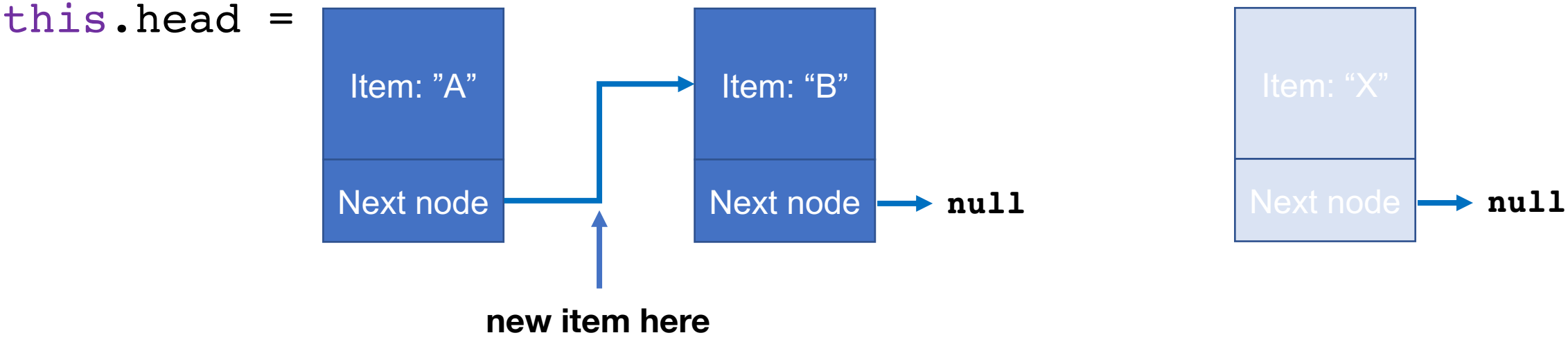
```
this.head = newNode;
```

ListOfStrings with a linked list: Insert item

```
//Inside the insert method (after validating index)
Node newNode = new Node(item, null);
if (index == 0) {
  newNode.setNextNode(this.head);
  this.head = newNode;
}
```

ListOfStrings with a linked list: Insert item

E.g. `list.insert("X", 1);`



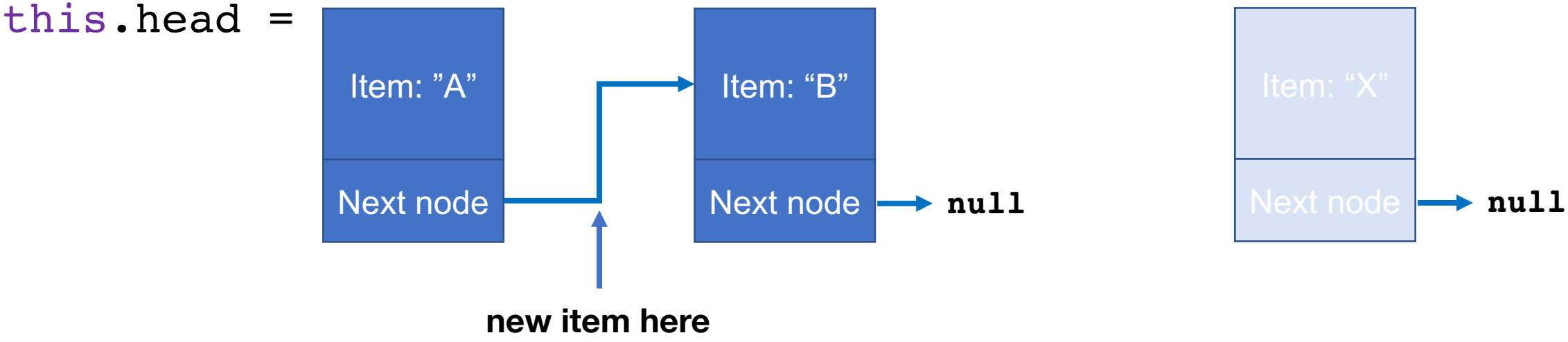
Check that `index >= 0 && index < this.numNodes`

→ Create a new Node containing the item:

```
Node newNode = new Node("X", null);
```

ListOfStrings with a linked list: Insert item

E.g. `list.insert("X", 1);`

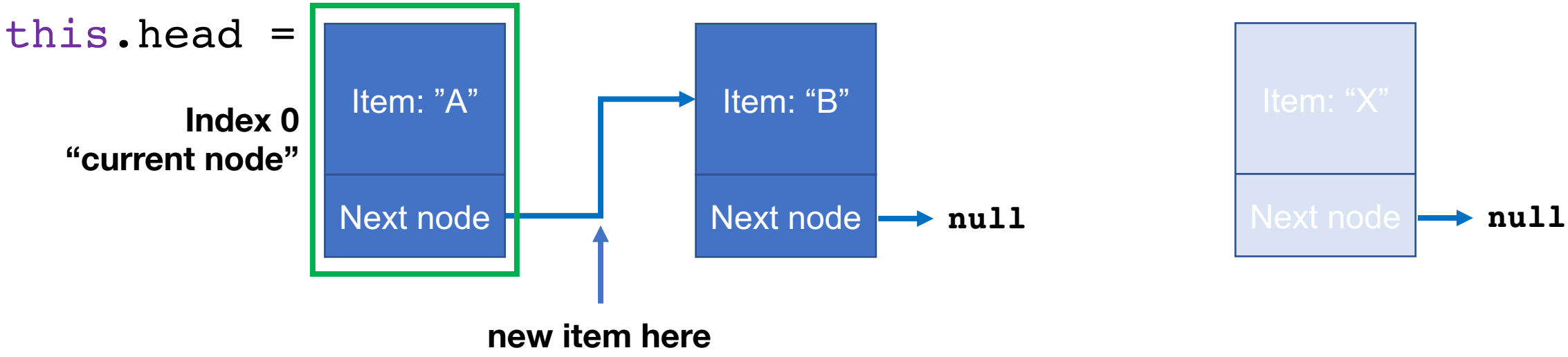


If `index > 0`

→ Starting from **`this.head`**, loop through the Nodes until get to the item at the index *before* where we want to insert

ListOfStrings with a linked list: Insert item

E.g. `list.insert("X", 1);`

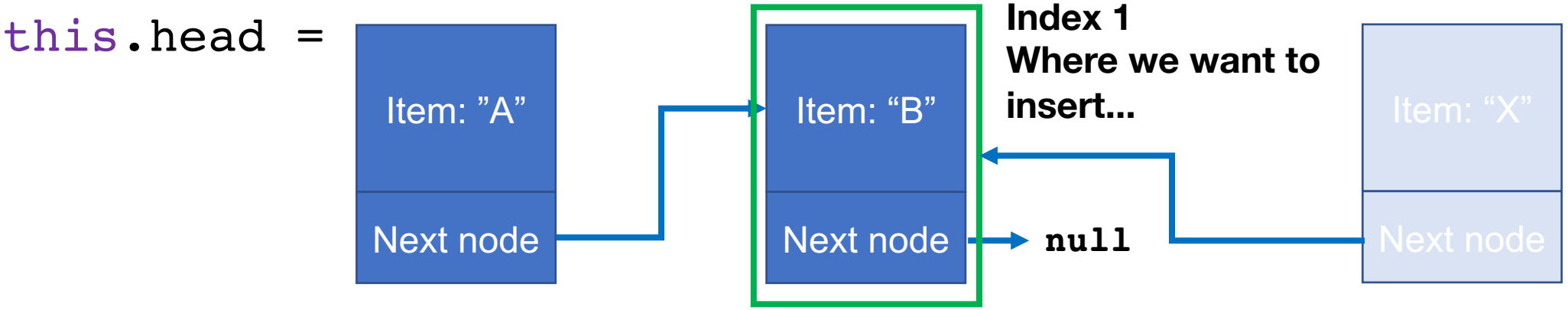


If `index > 0`

→ Starting from **`this.head`**, loop through the Nodes until get to the item at the index *before* where we want to insert

ListOfStrings with a linked list: Insert item

E.g. `list.insert("X", 1);`



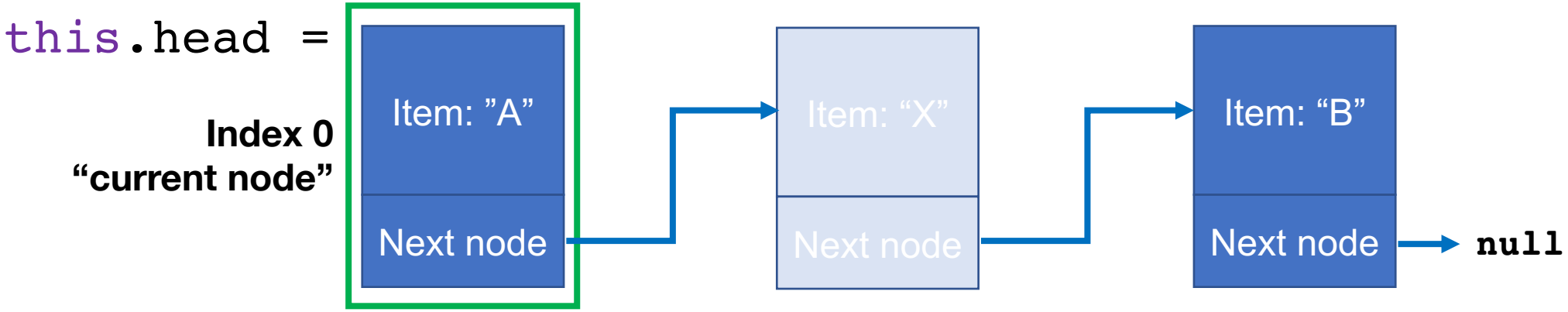
If `index > 0`

→ Starting from **`this.head`**, loop through the Nodes until get to the item at the index *before* where we want to insert

→ Set the new Node's **`nextNode`** to the *next* Node in the loop i.e. the Node currently occupying the index

ListOfStrings with a linked list: Insert item

E.g. `list.insert("X", 1);`



If `index > 0`

→ Starting from **`this.head`**, loop through the Nodes until get to the item at the index *before* where we want to insert

→ Set the new Node's **`nextNode`** to the next Node in the loop

→ Set the current Node's **`nextNode`** to the new Node

ListOfStrings with a linked list: Insert item

```
//Inside the insert method (after validating index)

Node newNode = new Node(item, null);
if (index == 0) {
  newNode.setNextNode(this.head);
  this.head = newNode;
}
else {
  int i = 0;
  Node currNode = this.head;
  while (i < index - 1) {
    i++;
    currNode = currNode.getNextNode();
  }
  newNode.setNextNode(currNode.getNextNode());
  currNode.setNextNode(newNode);
}
this.numNodes++;
```

Inserting at a *valid* index > 0

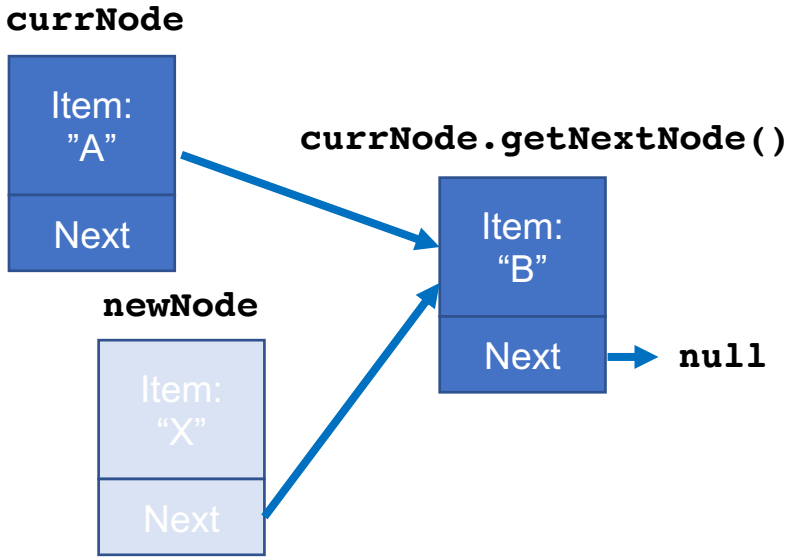
ListOfStrings with a linked list: Insert item

```
//Inside the insert method (after validating index)

Node newNode = new Node(item, null);
if (index == 0) {
  newNode.setNextNode(this.head);
  this.head = newNode;
}
else {
  int i = 0;
  Node currNode = this.head;
  while (i < index - 1) {
    i++;
    currNode = currNode.getNextNode();
  }
  newNode.setNextNode(currNode.getNextNode());
  currNode.setNextNode(newNode);
}
this.numNodes++;
```

Iterate through the nodes until we get to the one before the insert

ListOfStrings with a linked list: Insert item

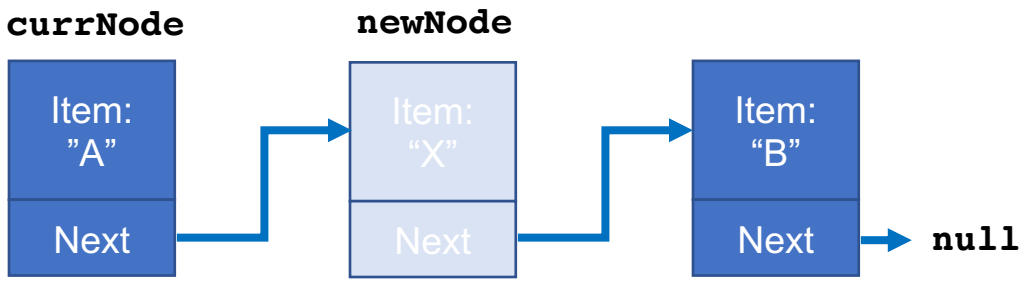
```
//Inside the insert method (after validating index)

Node newNode = new Node(item, null);
if (index == 0) {
  newNode.setNextNode(this.head);
  this.head = newNode;
}
else {
  int i = 0;
  Node currNode = this.head;
  while (i < index - 1) {
    i++;
    currNode = currNode.getNextNode();
  }
  newNode.setNextNode(currNode.getNextNode());
  currNode.setNextNode(newNode);
}
this.numNodes++;
```



Set the *new* Node's nextNode pointer to the node currently at `index`

ListOfStrings with a linked list: Insert item

```
//Inside the insert method (after validating index)

Node newNode = new Node(item, null);
if (index == 0) {
  newNode.setNextNode(this.head);
  this.head = newNode;
}
else {
  int i = 0;
  Node currNode = this.head;
  while (i < index - 1) {
    i++;
    currNode = currNode.getNextNode();
  }
  newNode.setNextNode(currNode.getNextNode());
  currNode.setNextNode(newNode);
}
this.numNodes++;
```



Set the *current* Node's nextNode pointer to the *new* Node

Implementing other collection ADTs

Still need to choose an underlying data structure, either:

- Built-in array
- Linked list (i.e. Node)
  - Encouraged

- Implementation of key methods will vary based on ADT specification

Exercise: ADT implementation

- Read through the completed linked list code: LinkedListOfStrings.java
  - Ask questions if anything is unclear!
- Run the tests: LinkedListOfStringsTest.java
- Try adding the following methods to the ADT (and implement):
  - `IListOfStrings filter(String substring);`
    `//returns a list of all the items containing substring`
  - `IListOfStrings reverse();`
    `//returns a copy of the list with items in reverse`

- *Note: for HW4, either arrays or linked lists are fine*
  - *linked lists encouraged!*