CS 5004: OBJECT ORIENTED DESIGN AND ANALYSIS

SPRING 2024

LECTURE 4

Tamara Bonaci
t.bonaci@northeastern.edu

Northeastern University
Khoury College of Computer Sciences

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

ADMINISTRIVIA

AGENDA

- Review
  - Inheritance
  - Interfaces
  - Abstract classes
- Enumerations
- Client vs. Implementer View
- Abstract Data Type (ADT)
- Arrays in Java
- Lists in Java
- Java Collections Framework
  - Stacks
  - Queues
  - Deques
  - Sets

REVIEW

CS 5004, SPRING 2024 – LECTURE 4

REVIEW: COMPOSITION

- Composition - set of classes connected by an ‘has-a’ relationships
- ‘Has-a’ relationship – a relationship where one class can use the functionality of another class by using an instance of that class
- Example 1:
  - Every person has a name
  - Every person has a date of birth
- Example 2:
  - Every vehicle has a make
  - Every vehicle has a model
  - Every vehicle has a manufacturing year

REVIEW: INHERITANCE AND "IS A" RELATIONSHIP

- Inheritance - set of classes connected by an 'is-a' relationships
- 'Is-a' relationship - hierarchical connection where one category can be treated as a specialized version of another
  - Example 1:
    - Every student is a person
    - Every ALIGN student is a student
  - Example 2:
    - Every pepper is a vegetable
    - Every bell pepper is a pepper
    - Every banana pepper is a pepper

REVIEW: EVERYTHING IS AN OBJECT IN JAVA

- `public class Object` - the root of the class hierarchy
  - Every class has Object as a superclass
  - All objects inherit public methods of `Object`

`protected Object clone()`	Creates and returns a copy of this object.
`Boolean equals (Object obj)`	Indicates whether some other object is "equal to" this one.
`protected void finalize()`	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class<?> getClass()`	Returns the runtime class of this Object.
`int hashCode()`	Returns a hash code value for the object.
`void notify()`	Wakes up a single thread that is waiting on this object's monitor.
`String toString()`	Returns a string representation of the object.

REVIEW: WHAT IS AN INTERFACE?

A set of ***method declarations***—a template for what a class can do

- Cannot be *instantiated* – no constructor
- Does not actually implement the methods it declares
- All methods are **public** by default
- Can contain only static fields

Classes can ***implement*** interfaces

- Classes fill in the implementation details of methods declared in an interface
- One class can implement multiple interfaces
  - …but *extend* only one super class.

REVIEW: WHEN IS AN INTERFACE USEFUL?

- Whenever you can imagine a “category” of classes that must have some common behavior
- AND implementation of common behavior needs to look different for each some/each of the classes

REVIEW: CONCRETE VS. ABSTRACT CLASSES

Concrete classes

- **Fully** implemented
- If implementing an interface, **must** implement all interface methods!
- Instantiated directly

Abstract classes

- **Partially** implemented
- If implementing an interface, **don't have to** implement all interface methods.
- Can't be instantiated directly.

REVIEW: WHEN TO USE AN ABSTRACT CLASS?

Instead of (or as well as) as an interface:

- When you want to provide *some* implementation details common to multiple potential subclasses.

Instead of a concrete class:

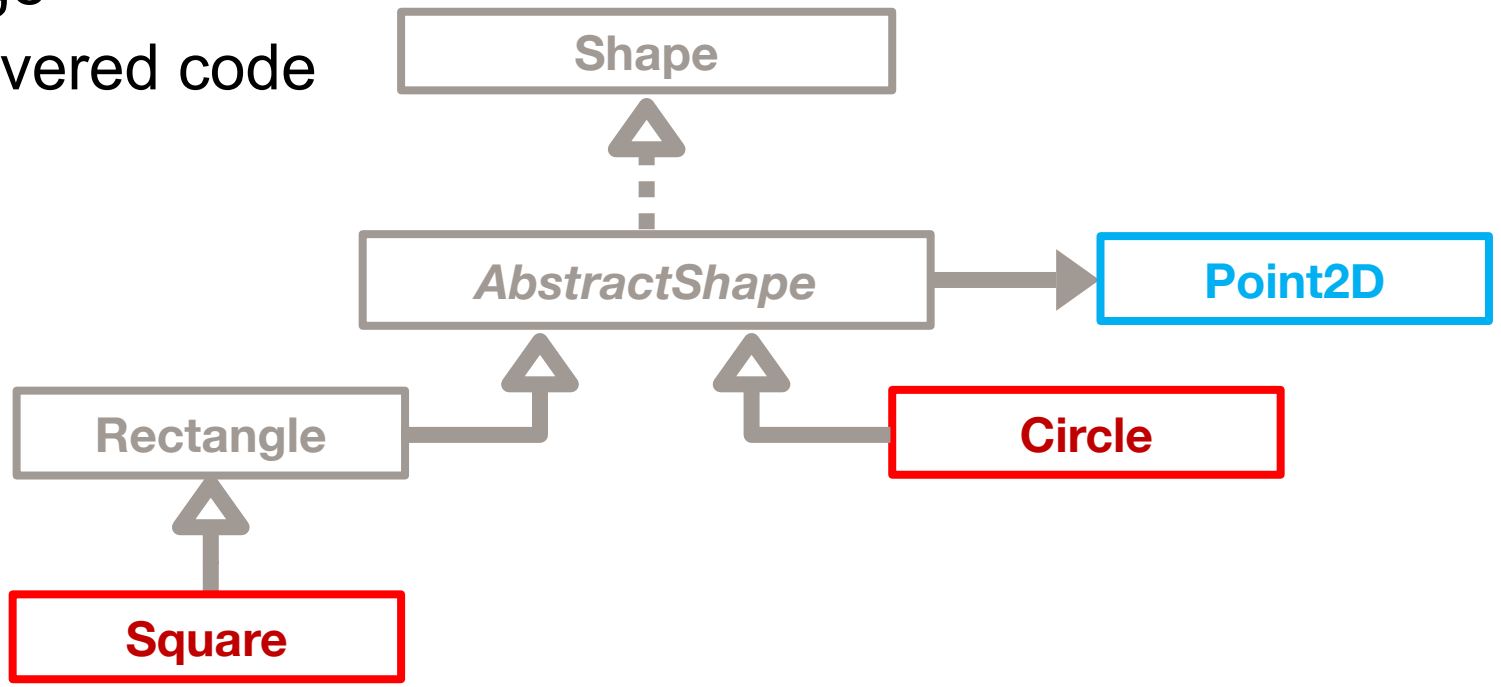
- When you don't want users to instantiate the class directly.

SOME CLARIFICATIONS

- Question 1: do we always need to have abstract classes and interfaces?
  - No, only if they're useful
  - But if in doubt, feel free to include them – you can't have too much of the good stuff ☺

SOME CLARIFICATIONS

- Question 2: how do we test abstract classes?
  - Write tests for concrete classes that don't have subclasses (including inherited methods)
  - Check Jacoco coverage
  - Add tests for any uncovered code



If `SquareTest` covers `Shape` methods `draw()`, `area()`, `resize()`, these methods will not have to be tested for parent classes

ENUMERATIONS

CS 5004, SPRING 2024 – LECTURE 4

REPRESENTING DATA THAT HAVE FINITE, SPECIFIC VALUES

- Example: we designed a class `Book`, and now we want to add information about the format in which we can buy it (hardcover, paperback, kindle)

- Question: how to represent this information in the Book class?

- Answer: let's use enumerated types

```
public enum TypeOfBook{HARDCOVER, PAPERBACK, KINDLE}
```

WHAT IS AN ENUMERATION?

- Enumeration - a way to represent a set of finite constants
- Represented as an `enum` data type
- What should be an **`enum`**?
  - Days of the week
  - Directions (N, S, E, W)

- **What shouldn't be an `enum`:**
  - Anything that is not finite
  - Anything that could be described as a "type of" something else
  - Anything that has properties/behaviors associated with it

BASIC ENUM STRUCTURE

- `Enum` data types are created in their own files (like a class), with keyword `enum`
- Each field is named in ALL CAPS (because they're always constant)
- Fields are separated by commas, and they don't have data types
- Fields are also not set to equal anything

```
public  enum DayOfWeek
{
MONDAY, TUESDAY,
WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY,
SUNDAY
}
```

USING AN ENUM

- Variables can have an enum data type
- We set the value of an enum variable using:

```
<EnumType>  varName = <EnumType>.<Field>
```

```
DayOfWeek mon = DayOfWeek.MONDAY;
```

THE SWITCH STATEMENT

- An alternative to if-else if-else
- Neater (less typing)
- Only works with enums and a handful of other data types (incl. String)

```
switch(id) {
  case value-one: //is id==value-one?
     [do something 1]
     break;
  case value-two: //is id==value-two?
     [do something 2]
     break;
  ...
  default: //none of the above
     [do something-none-of-the-above]
 }
```

GOOD OOD PRACTICES

CS 5004, SPRING 2024 – LECTURE 4

HOW DO WE REPRESENT ..X..?

- When X is something descriptive, e.g., color, animal species, day of week?

- Do I make X:
  - a `String` field in a class?
  - an `enum` field in a class?
  - a class with its own properties and methods?

HOW DO WE REPRESENT ..X..?

- When X is something descriptive, e.g., color, animal species, day of week?
- Do I make X:
  - a `String` field in a class?
  - an `enum` field in a class?
  - a class with its own properties and methods?

- Factors to consider:
  - Is there a finite and fairly small set of possible values?
  - Is X for information only?
  - ...or are their additional properties/behaviors dependent on the value of X?

IS THERE A FINITE SMALL SET OF POSSIBLE VALUES?

NO - e.g., a person's name, a book title → Use a String field in another class

```
public class  Name {
   private String firstName;
   private String lastName ;

   public Name (String firstName,  String lastName) { ...}
}
```

IS THERE A FINITE SMALL SET OF POSSIBLE VALUES?

YES – e.g., vehicle color, pet species, day of week

→ String field is not a great choice (error prone)
→ Maybe an enum field (if set is fairly small)
→ Maybe a class

More information needed!

- Is X for information only?
- ...or are their additional properties/behaviors dependent on the value of X?

ARE PROPERTIES/BEHAVIORS DEPENDENT ON THE VALUE OF X?

- Might depend on specific situation
  - NO – e.g., vehicle color, day of week (much of the time)
  - An enum field is possibly acceptable

  - YES – e.g., pet species
  - An enum field is NOT the OOD choice
  - A class (or sub class) is usually the most appropriate OOD choice

WOULD YOU DESCRIBE X AS A TYPE OF SOMETHING?

If yes, X should be a class, not a String or an enum!

CLIENT VS. IMPLEMENTER VIEW

CS 5004, SPRING 2024 – LECTURE 4

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

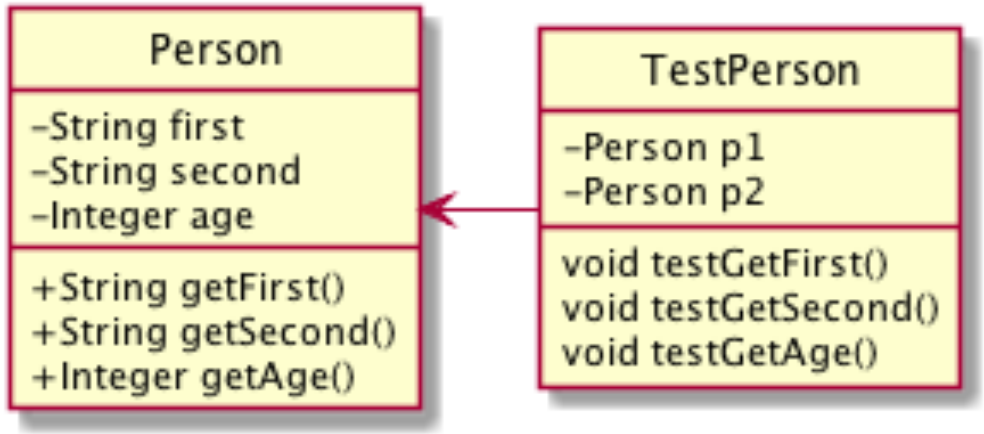
- A piece of code (class, method) that relies on **public** elements of some other class *K*

Implementer

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

CLIENT AND IMPLEMENTER

- So, what is a client and what is an implementor?



- `Person` is the implementation of the concept of a person
- `PersonTest` is a client of `Person`

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**

ABSTRACT DATA TYPE

CS 5004, SPRING 2024 – LECTURE 4

EXAMPLE: ARE THESE CLASSES THE SAME?

```
public class Point1 {
   private Float x;
   private Float y;
}
```

```
public class Point2 {
   private Float r;
   private Float theta;
}
```

- How are these classes different and similar?
  - Different: cannot replace one with the other in the program
  - Same: both classes implement a concept of a 2D point
- Goal of the ADT methodology is to express the sameness
  - Clients depend only on the concept of 2D point

ABSTRACT DATA TYPE (ADT)

- Model that describes data by specifying the operations that we can perform on them

- ADTs are written from the view of the client
  - We need to capture the clients' expectations in terms of the operations on the ADT

- For each operation, we need to 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

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

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

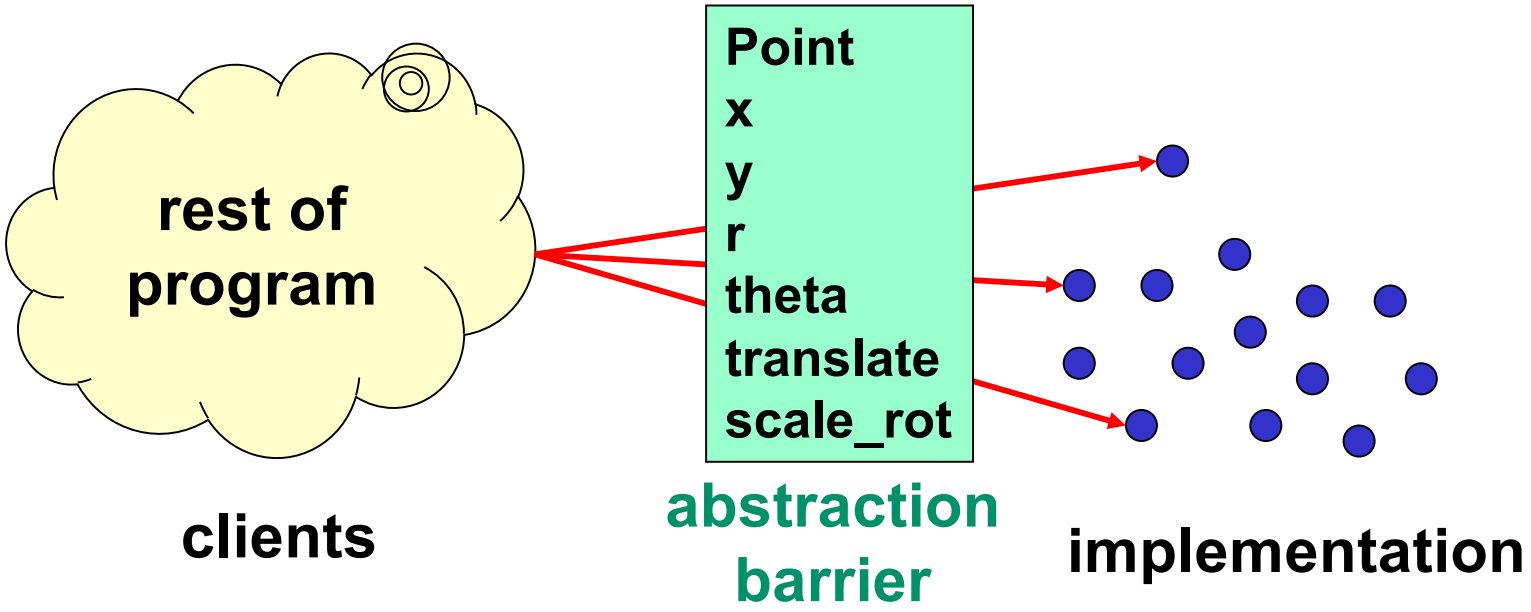
The docs (API):
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html
(Google “Java 11 String”)

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

ADT = OBJECT + OPERATIONS



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

SPECIFYING A DATA ABSTRACTION

Purpose of a specification:

- Planning
- Documenting
- *No implementation just yet*

SPECIFYING A DATA ABSTRACTION

- A collection of procedural abstractions
  - Not a collection of procedures

- An abstract state
  - Not the (concrete) representation in terms of fields, objects, …
  - "Does not exists", but used to specify operations
  - Concrete state, not part of the specification, implements the abstract state

- Each operation described in terms of "creating", "observing", "producing" or "mutating"
  - No operations other than those in the specification

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!*

CONCEPT OF A 2D POINT AS AN ADT

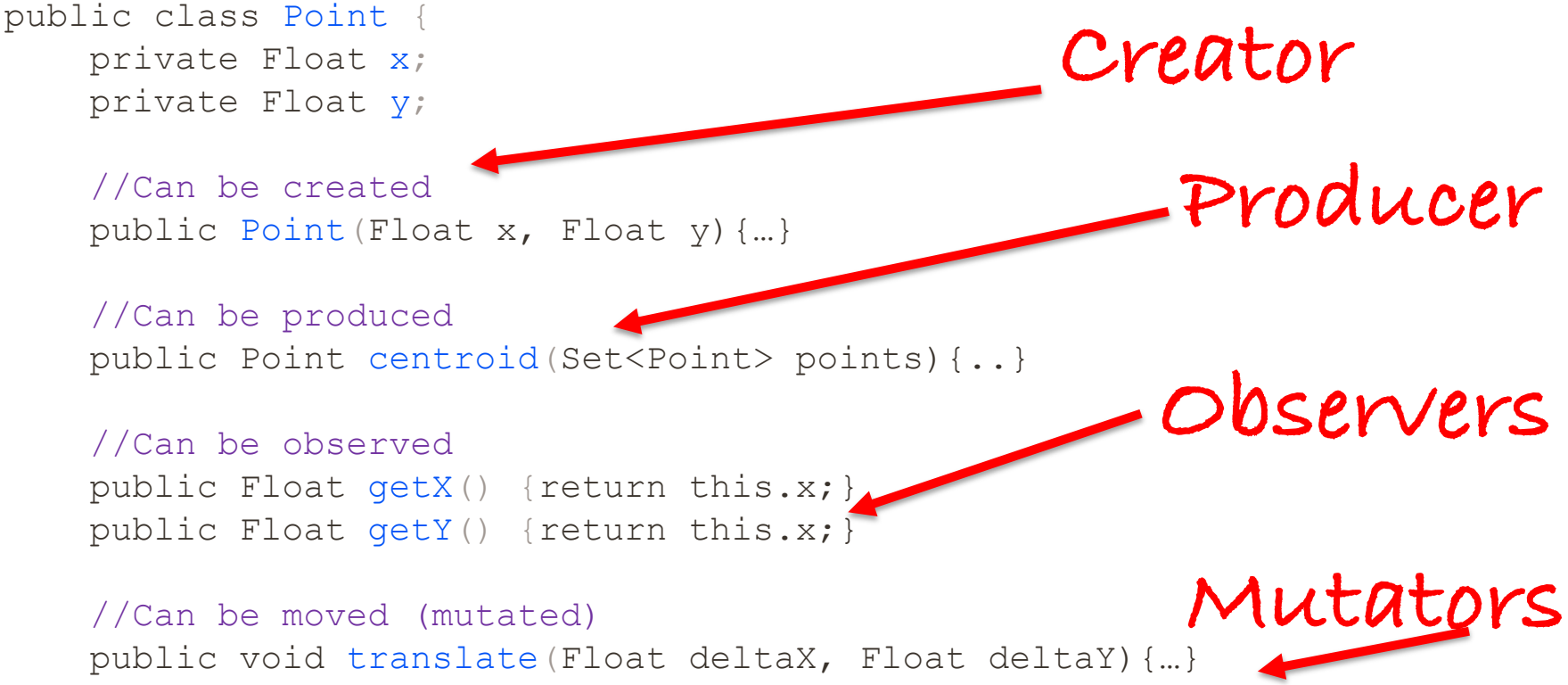
```
public class Point {
    private Float x;
    private Float y;

    //Can be created
    public Point(Float x, Float y){…}

    //Can be produced
    public Point centroid(Set<Point> points){..}

    //Can be observed
    public Float getX() {return this.x;}
    public Float getY() {return this.x;}

    //Can be moved (mutated)
    public void translate(Float deltaX, Float deltaY){…}
    public void scaleAndRotate(Float deltaX, Float deltaY){…}
}
```



WHY DO WE NEED ADTS?

- Organizing and manipulating data is pervasive
  - Inventing and describing algorithms is less common
- Start your design by designing data structures
  - How will relevant data be organized?
  - What operations will clients be permitted to do on the data?
- Potential problems with choosing data abstraction:
  - Decisions about data structures often made too early
  - Duplication of effort in creating derived data
  - Very hard to change key data structures (modularity!)

BENEFITS OF ADT

- If clients “respect” (are “forced to respect”) data abstractions:
  - We can delay decisions on how ADT is implemented
  - We can fix bugs by changing how ADT is implemented
  - We can change algorithms:
    - For performance
    - For security
    - In general or in specialized situations

- We talk about “abstraction barrier”
  - A good thing to have, and not to cross

ABSTRACT DATA TYPE - SUMMARY

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

- Clients care about the ADT
- Developers care about the data structure (it's memory consumption, speed, correctness) and how the inner workings of the data structure faithfully implement the ADT

IMPLEMENTAING AN ADT IN JAVA

CS 5004, SPRING 2024 – LECTURE 4

IMPLEMENTING AN ADT

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

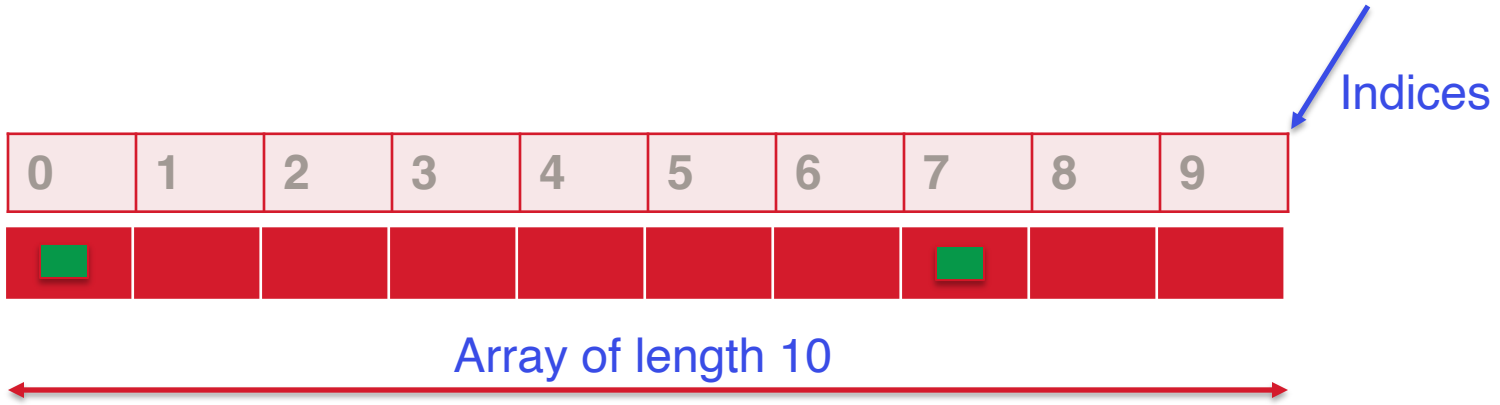
- Choose an underlying data structure:
  - An array
  - Something else…
- Write an interface if appropriate
  - Useful if there may be multiple different implementations

ARRAYS IN JAVA

CS 5004, SPRING 2024 – LECTURE 4

ARRAYS IN JAVA

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



ARRAYS OF OBJECTS

- An array of objects is created just like an array of primitive type data items

- Example:

```
Dancer[] dancers = new Dancer[5]; //Dancer is a user-defined class
```

- `Dancers` contains five memory spaces in which the address of five `Dancer` objects can be stored
- The `Dancer` objects have to be instantiated using the constructor of the Dancer class, and their references should be assigned to the array elements

MULTIDIMENSIONAL ARRAYS

- Multidimensional arrays (jagged arrays) - arrays of arrays where each element of an array holds a reference to another array
- Created by appending one set of square brackets ([]) per dimension

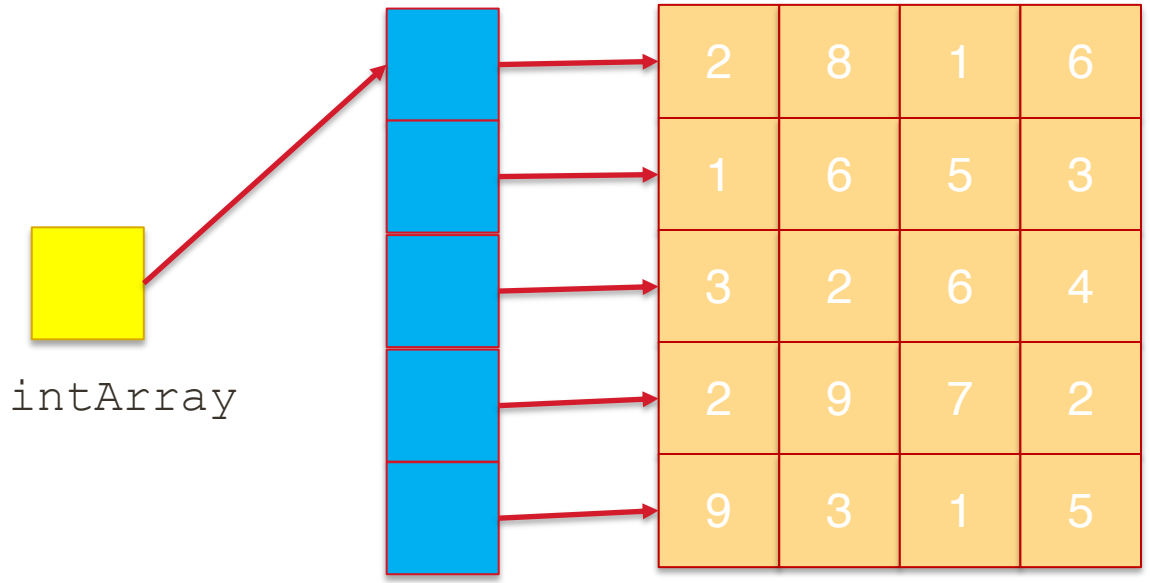
```
int[][] intArray = new int[10][20]; //a 2D array
or matrix

int[][][] intArray = new int[10][20][10]; //a 3D
array
```

MULTIDIMENSIONAL ARRAYS

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

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



MULTIDIMENSIONAL ARRAYS

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

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

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

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

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

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

IMPLEMENTING A COLLECTION ADT WITH AN ARRAY

CS 5004, SPRING 2024 – LECTURE 4

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

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

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

Client view:

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

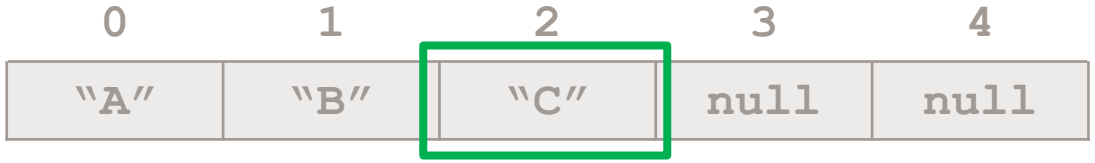
Check that `this.size < this.items.length`
→ `this.items[this.size] = newItem`
→ `this.size++` (`this.size` is now 2)

LISTOFSTRINGS WITH AN ARRAY: ADDING

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

Underlying data structure:

- String array, length 5



Client view:

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

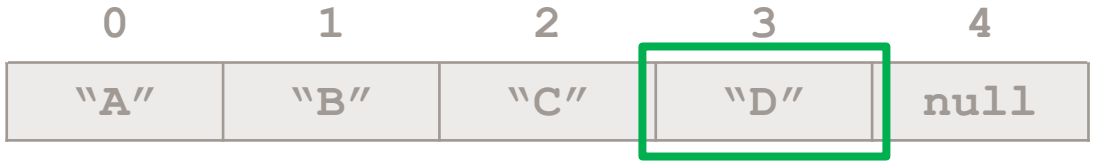
Check that `this.size < this.items.length`
→ `this.items[this.size] = newItem`
→ `this.size++` (`this.size` is now 3)

LISTOFSTRINGS WITH AN ARRAY: ADDING

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

Underlying data structure:

- String array, length 5



Client view:

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

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

→ `this.items[this.size] = newItem`

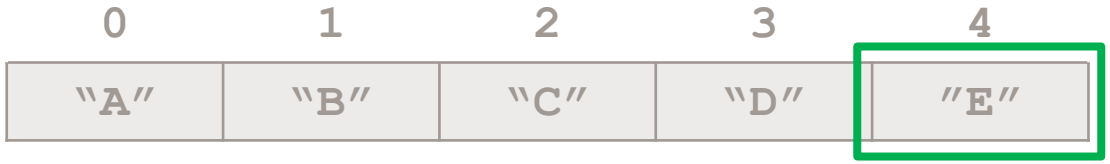
→ `this.size++` (`this.size` is now 4)

LISTOFSTRINGS WITH AN ARRAY: ADDING

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

Underlying data structure:

- String array, length 5



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

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:

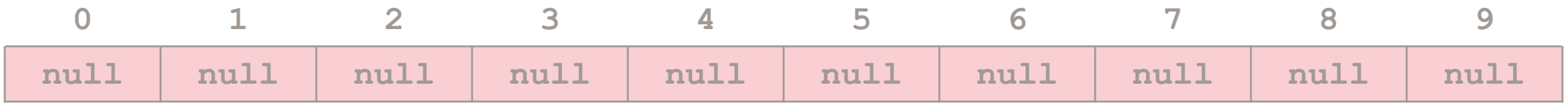
- String array, length 5

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`



LISTOFSTRINGS WITH AN ARRAY: ADDING

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

Underlying data structure:

- String array, length 10

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`

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

LISTOFSTRINGS WITH AN ARRAY: ADDING

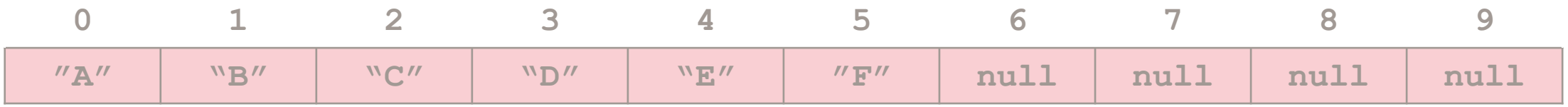
E.g., `list.add("F");`

Underlying data structure:

- String array, length 10

Client view:

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



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

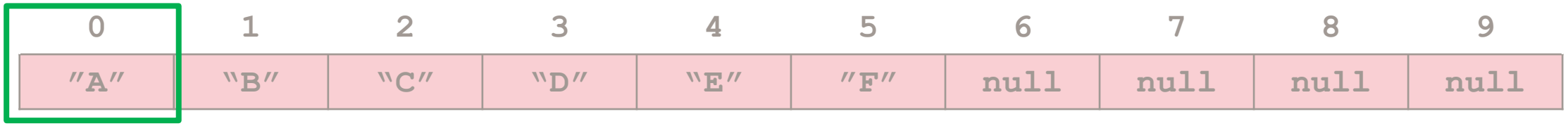
E.g., `list.get(0);`

Underlying data structure:

- String array, length 10

Client view:

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



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

→ `return this.items[index]`

LISTOFSTRINGS WITH AN ARRAY: GET AN ITEM

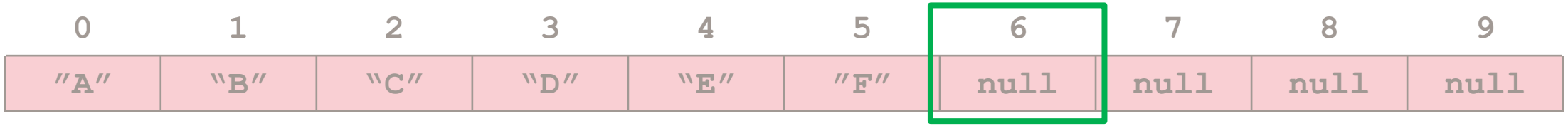
E.g., `list.get(6);`

Underlying data structure:

- String array, length 10

Client view:

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

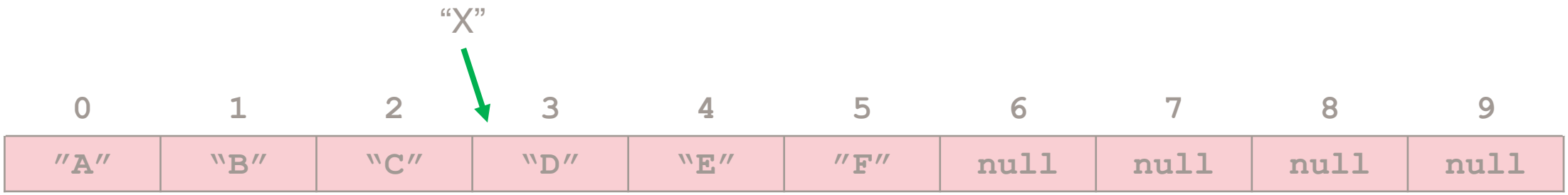


Check that `index >= 0 && index < this.size`

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

LISTOFSTRINGS WITH AN ARRAY: INSERT AT INDEX

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



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

LISTOFSTRINGS WITH AN ARRAY: INSERT AT INDEX

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

LISTOFSTRINGS WITH AN ARRAY: INSERT AT INDEX

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

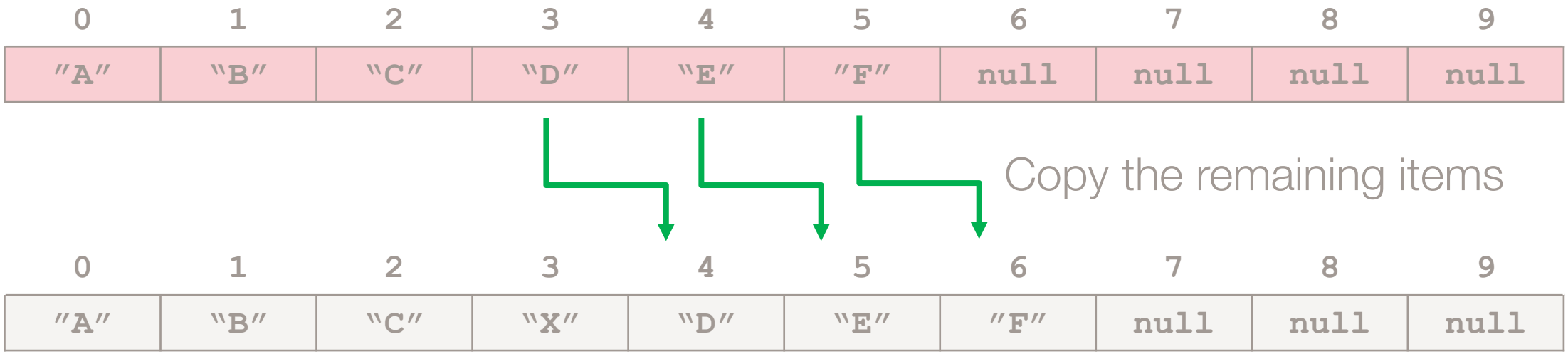
0	1	2	3	4	5	6	7	8	9
"A"	"B"	"C"	"D"	"E"	"F"	null	null	null	null

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)`



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

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

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.

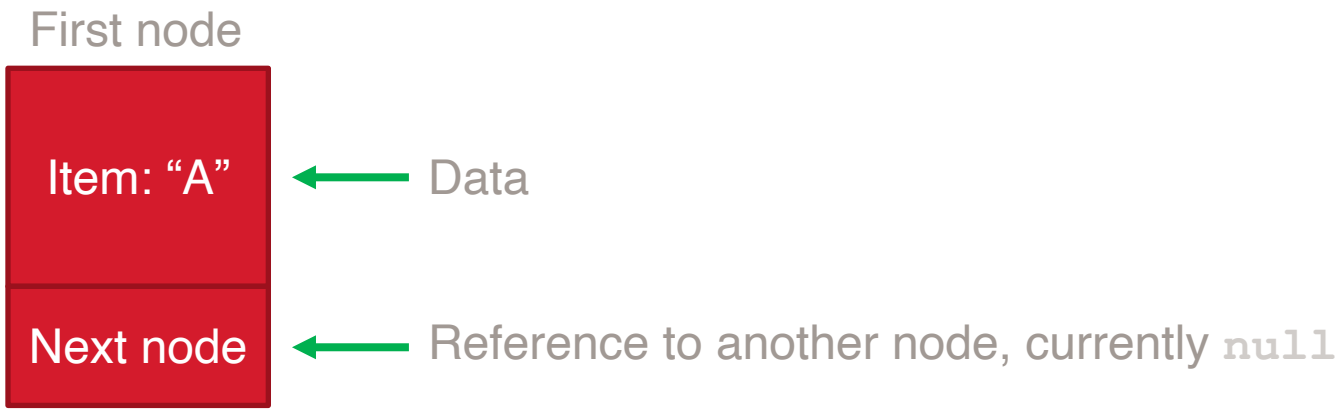
- Items in the list are commonly referred to as “nodes”

Two ways to implement:

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

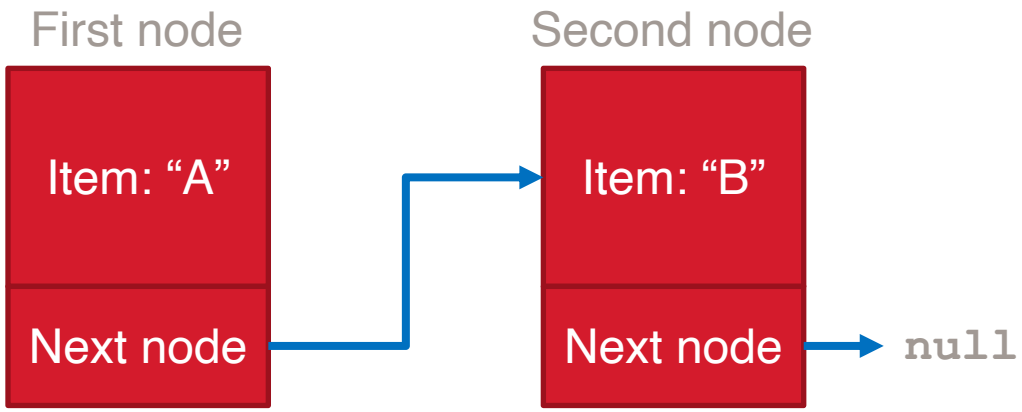
INTRODUCING THE LINKED LIST

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



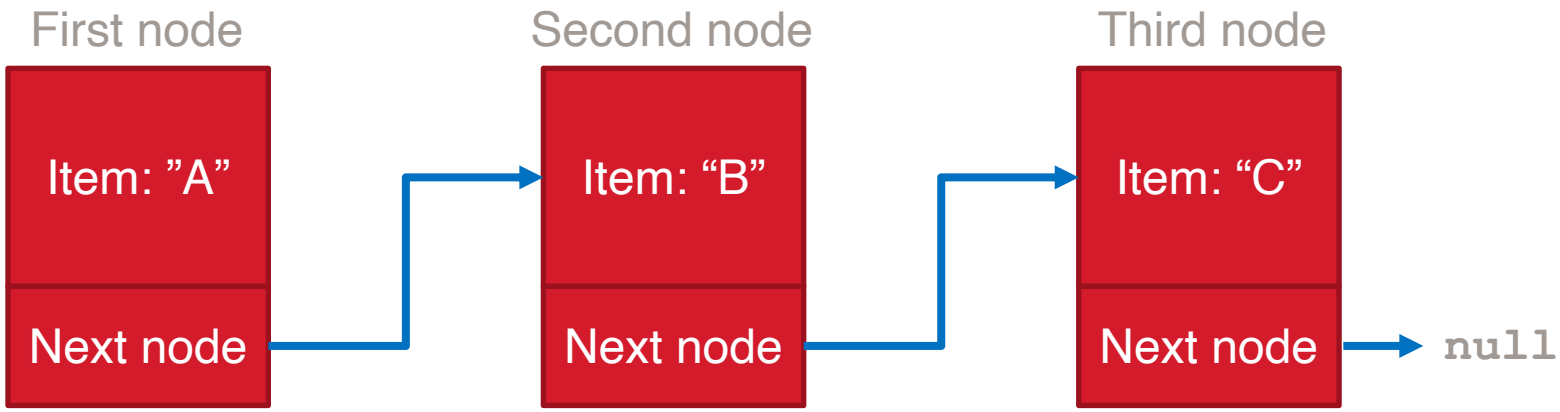
INTRODUCING THE LINKED LIST

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;`

Client view:

- An empty list
  - No items
  - Size 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`

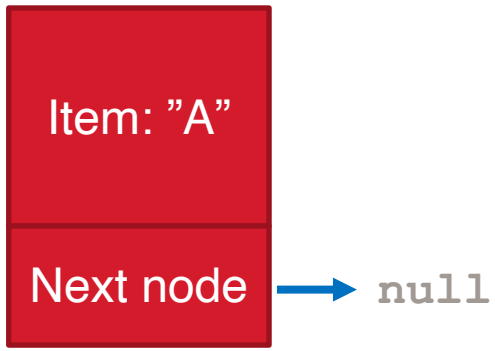
```
this.head = newNode;
```

LISTOFSTRINGS WITH A LINKED LIST: ADD ITEM

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

Underlying data structure:

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



Client view:

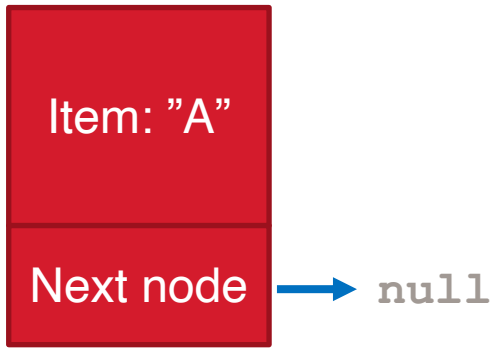
- List contents: "A"
- Size 1

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");`

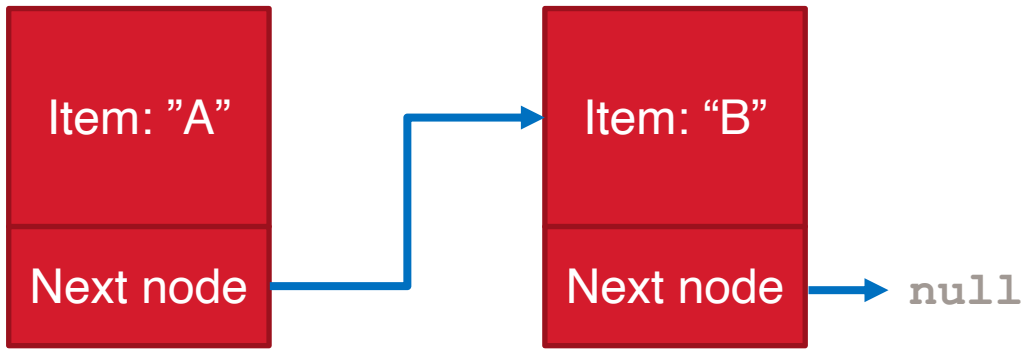
Underlying data structure:

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

Client view:

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

- `this.head` =

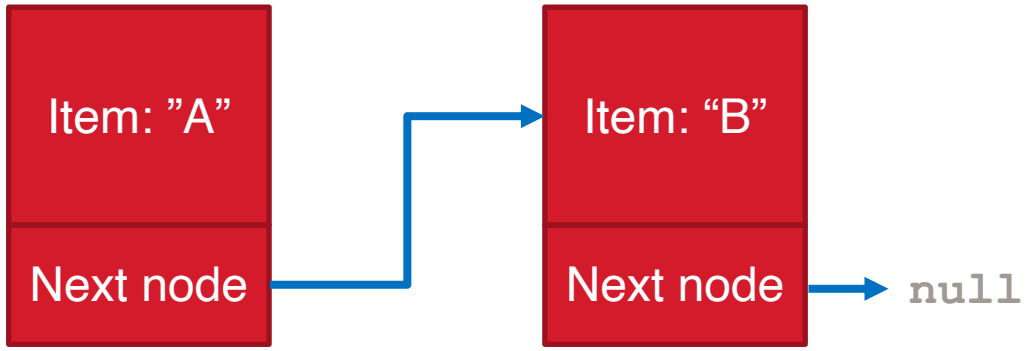


LISTOFSTRINGS WITH A LINKED LIST: GET ITEM

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

Underlying data structure:

- `this.numNodes = 2;`
- `this.head =`



Client view:

- Expected return: “A”

LISTOFSTRINGS WITH A LINKED LIST: GET ITEM

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

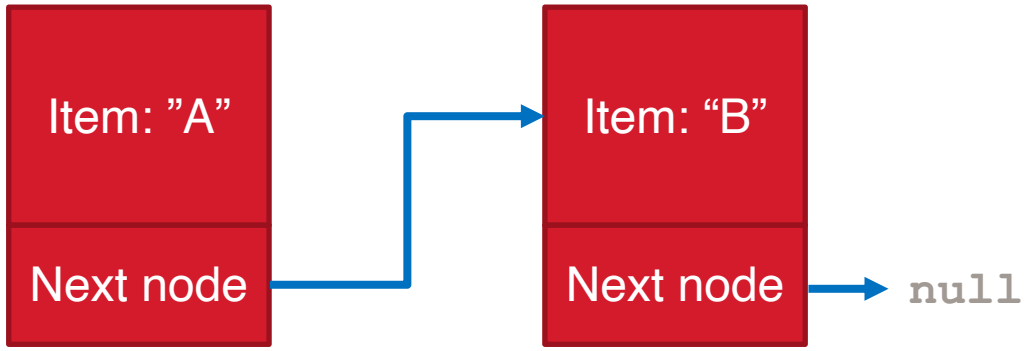
Underlying data structure:

- `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

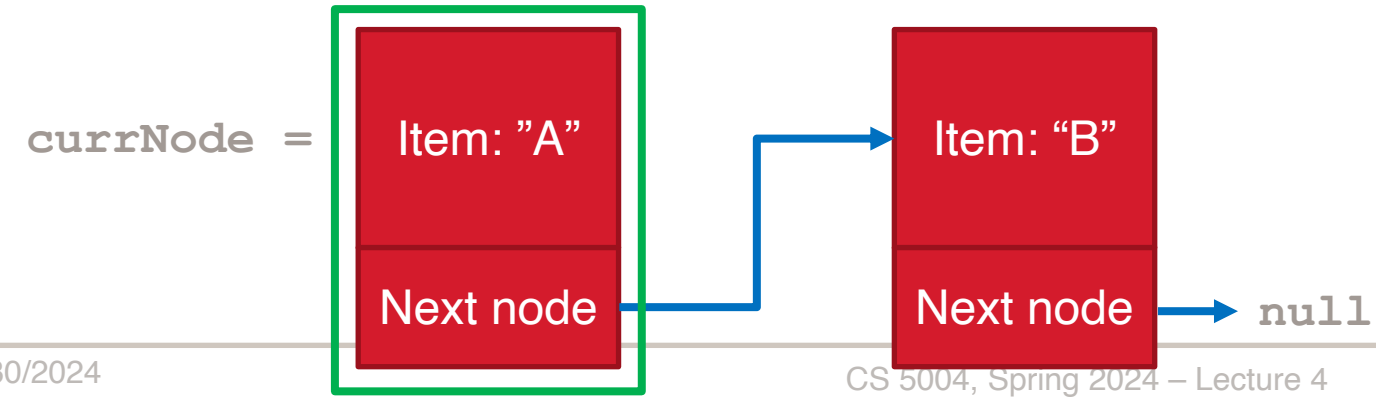
E.g., `list.get(0);`

```
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();
```

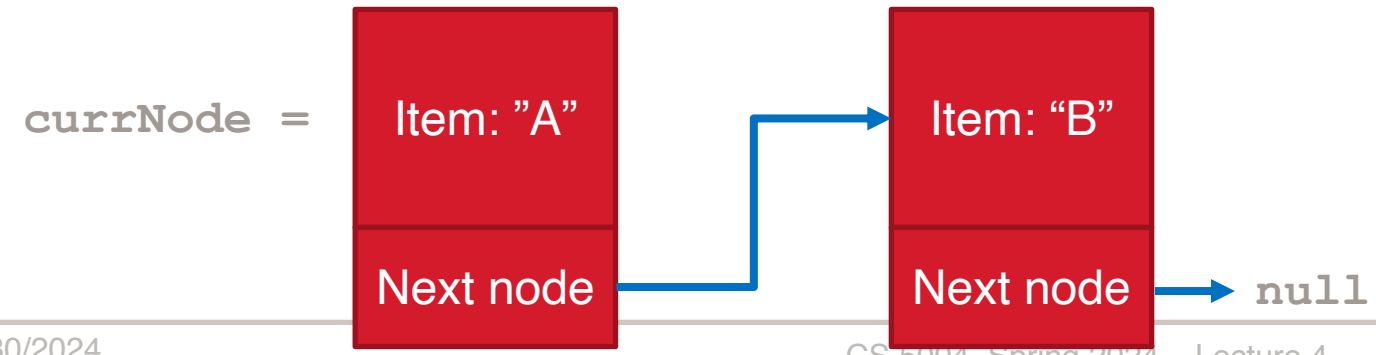


LISTOFSTRINGS WITH A LINKED LIST: GET ITEM

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

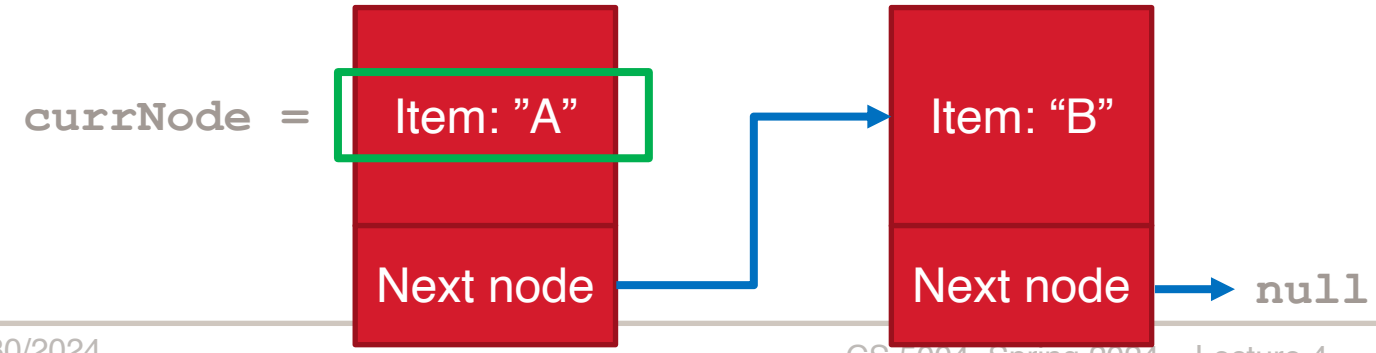


LISTOFSTRINGS WITH A LINKED LIST: GET ITEM

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

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

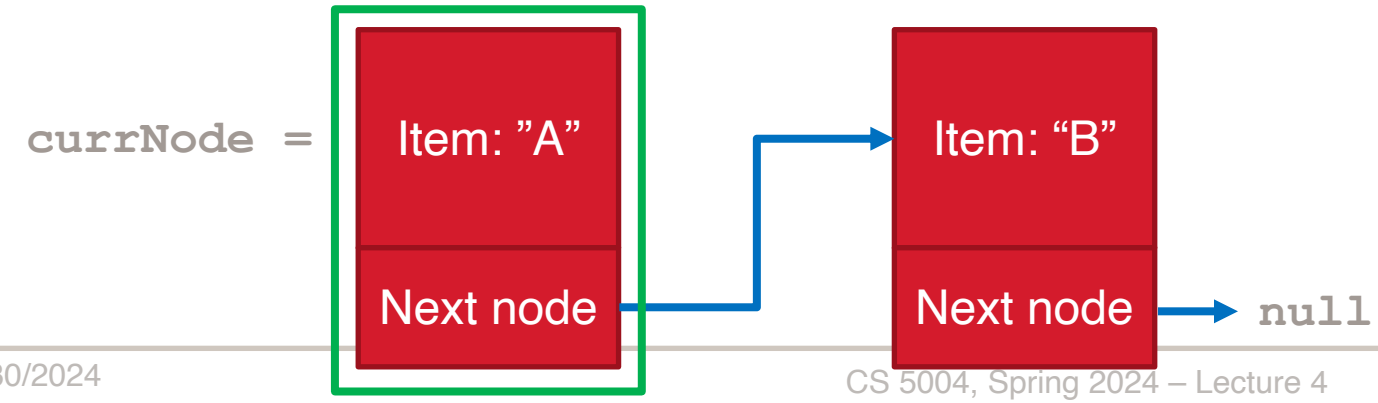
Return the item stored in the current node



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();
```

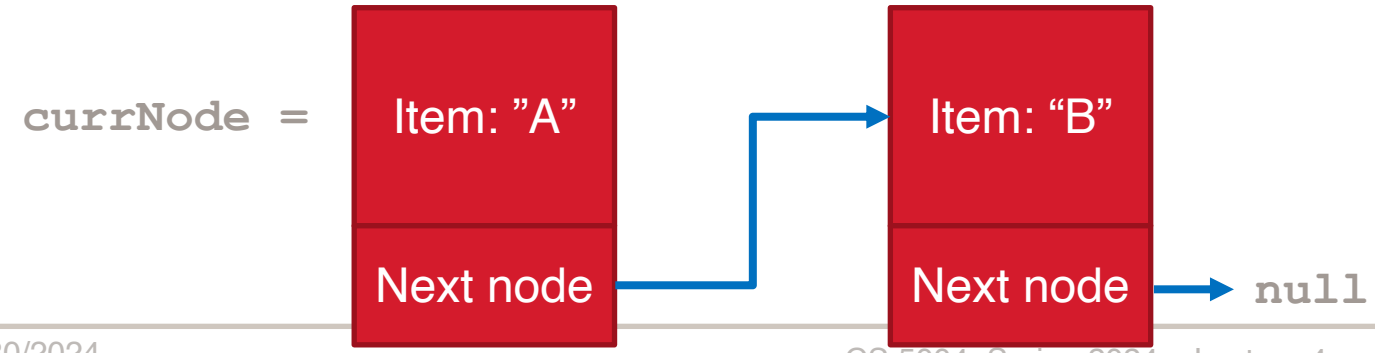


LISTOFSTRINGS WITH A LINKED LIST: GET ITEM

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

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

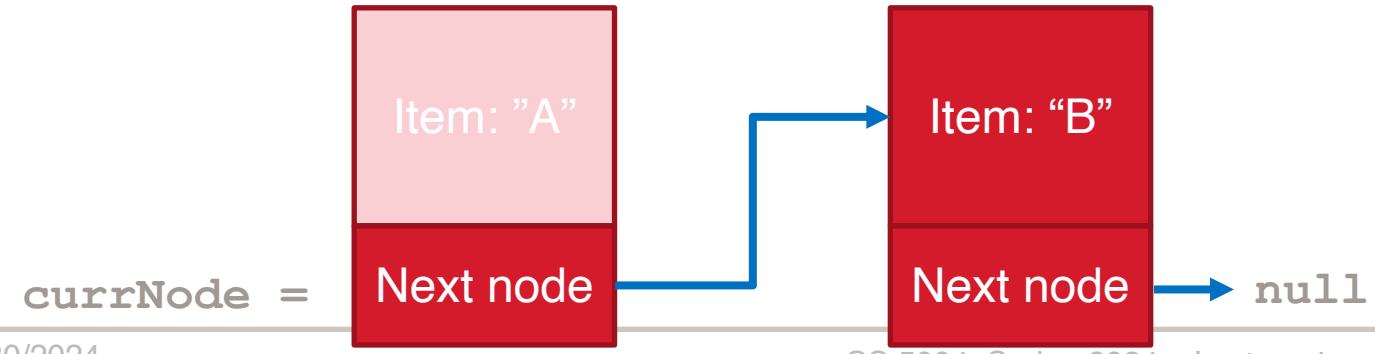
i < index →
Step into the while loop



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();
```

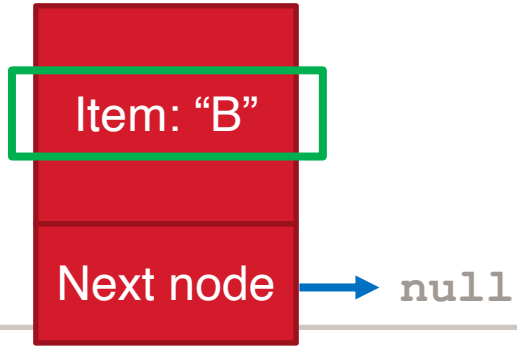


LISTOFSTRINGS WITH A LINKED LIST: GET ITEM

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

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

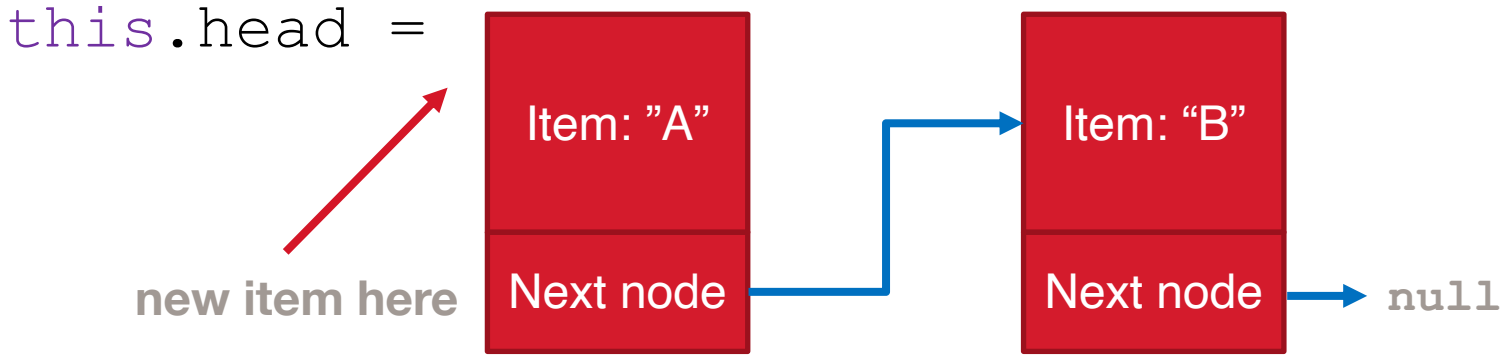
Return the item stored in the current node



currNode =

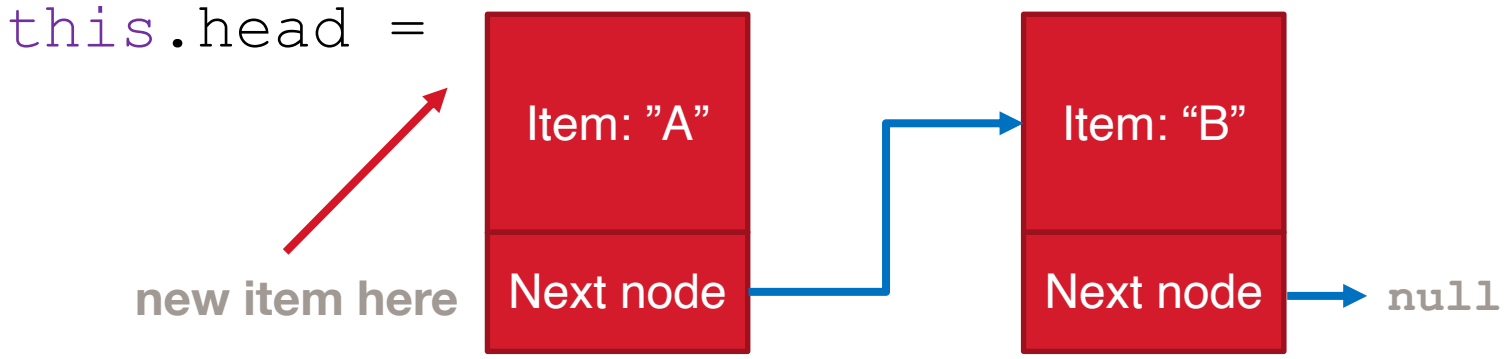
LISTOFSTRINGS WITH A LINKED LIST: INSERT ITEM

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



LISTOFSTRINGS WITH A LINKED LIST: INSERT ITEM

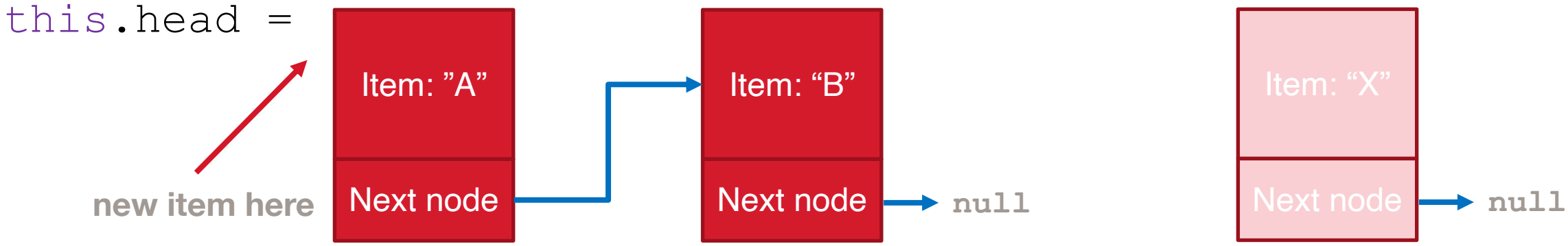
E.g., `list.insert("X", 0);`



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

LISTOFSTRINGS WITH A LINKED LIST: INSERT ITEM

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



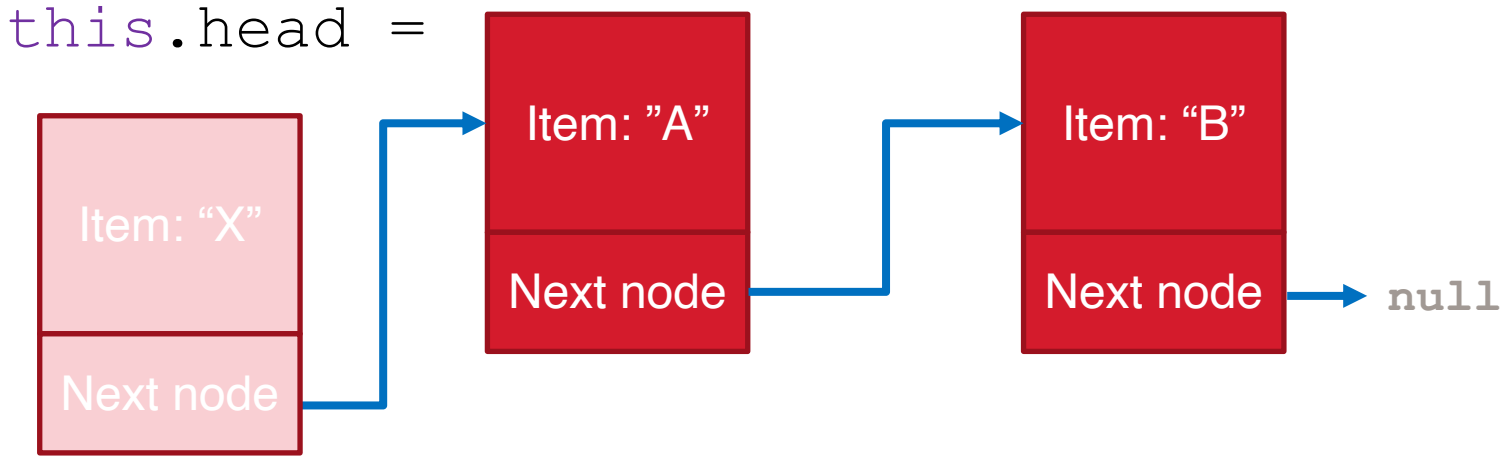
Check that `index >= 0 && index < this.numNodes`

→ 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);`



If `index == 0`

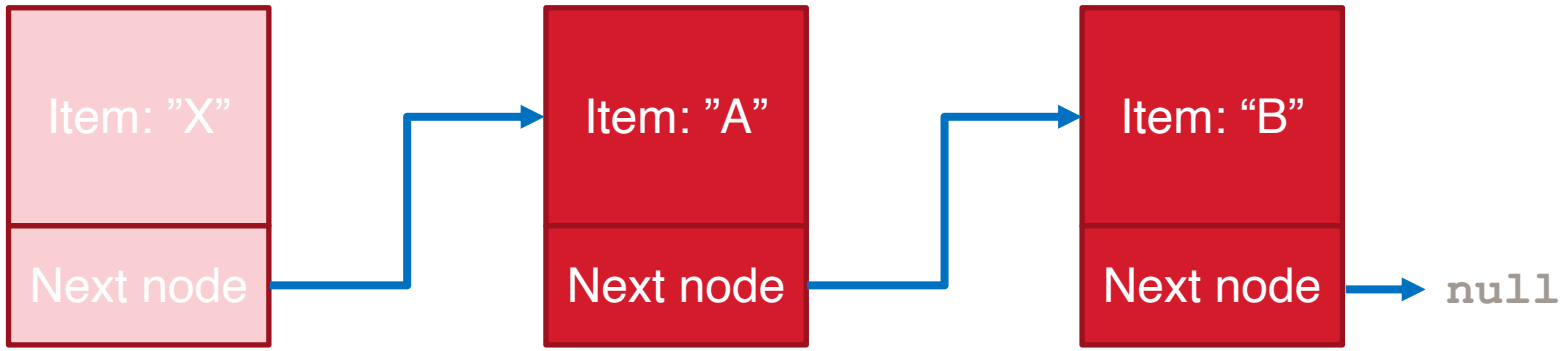
→ Set the new Node's **`nextNode`** to **`this.head`**;

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

LISTOFSTRINGS WITH A LINKED LIST: INSERT ITEM

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

`this.head =`



If `index == 0`

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

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

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

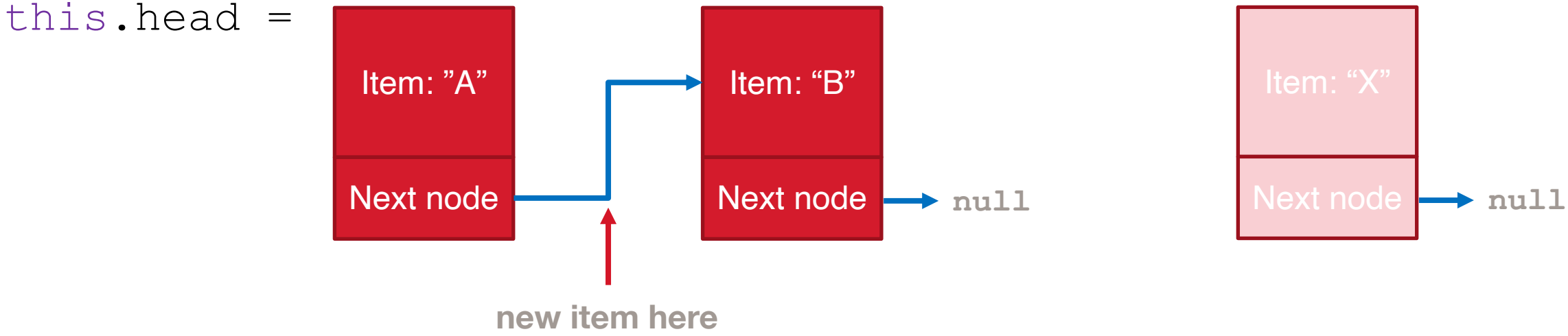
```
this.head = newNode;
```

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);`



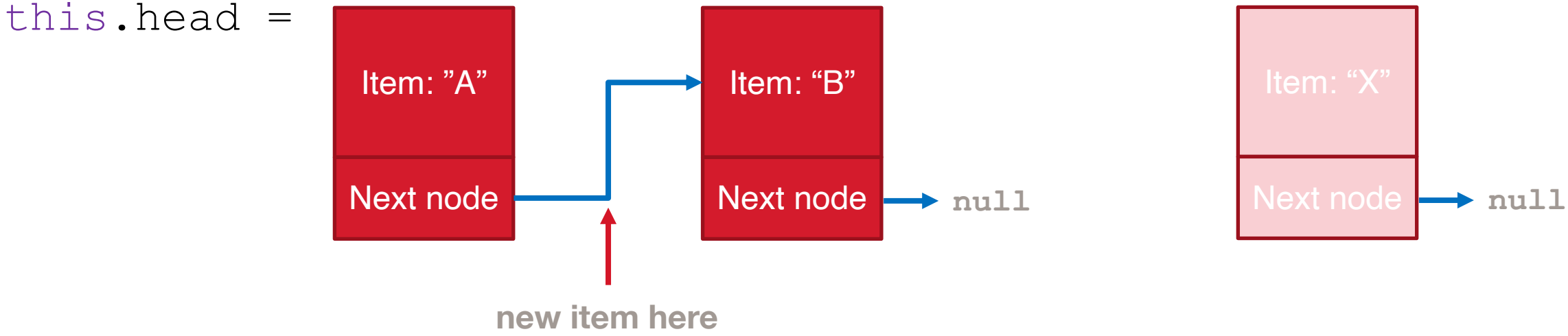
Check that `index >= 0 && index < this.numNodes`

→ 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);`

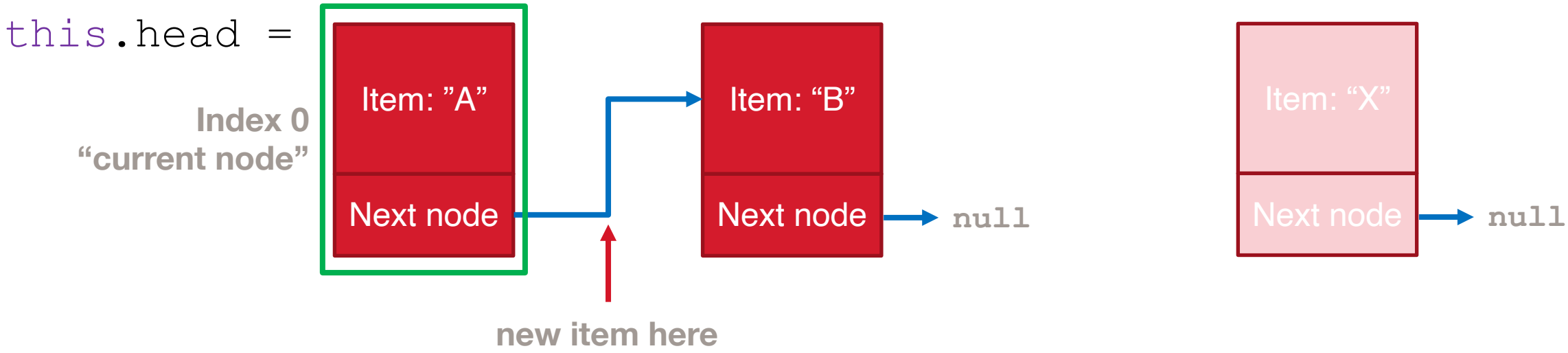


If `index > 0`

→ 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);`

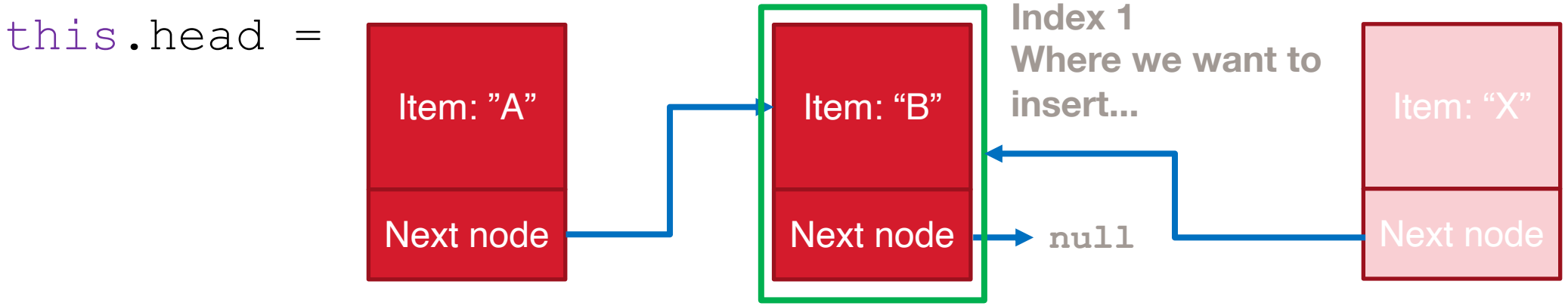


If `index > 0`

→ 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);`



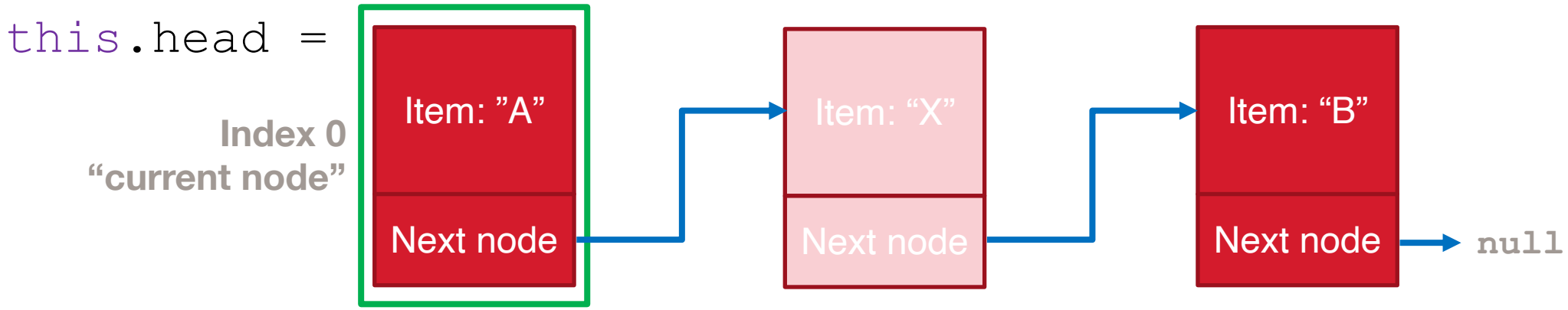
If `index > 0`

→ 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

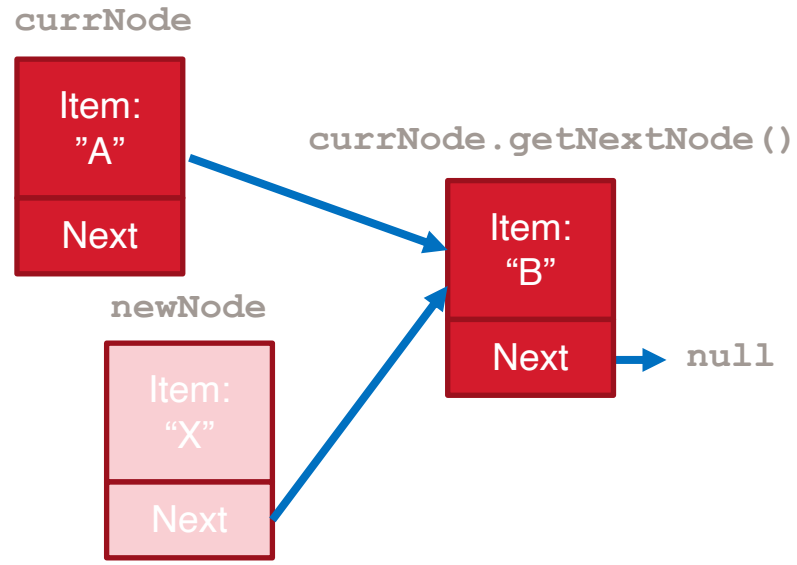
LISTOFSTRINGS WITH A LINKED LIST: INSERT ITEM

```
//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 de curi = 0;
  NorNode = 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

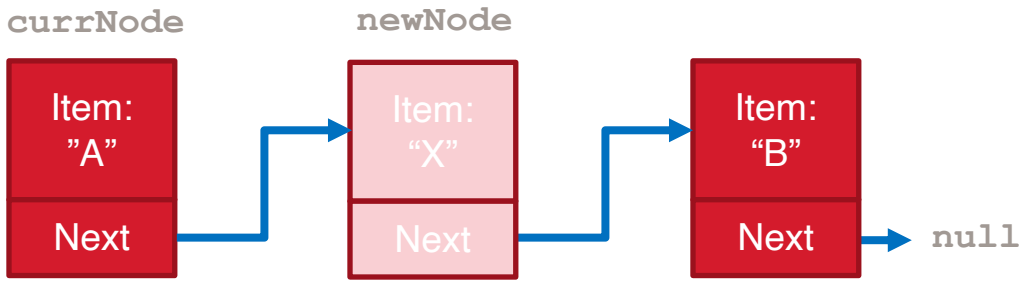
```
//Inside the insert method (after validating index)
Node newNode = new Node(item, null);
if (index == 0) {
  newNode.setNextNode(this.head);
  this.head = newNode;
}
else {
  int i = 0;
  Node currNode = this.head;
  while (i < index - 1) {
    i++;
    currNode = currNode.getNextNode();
  }
  newNode.setNextNode(currNode.getNextNode());
  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

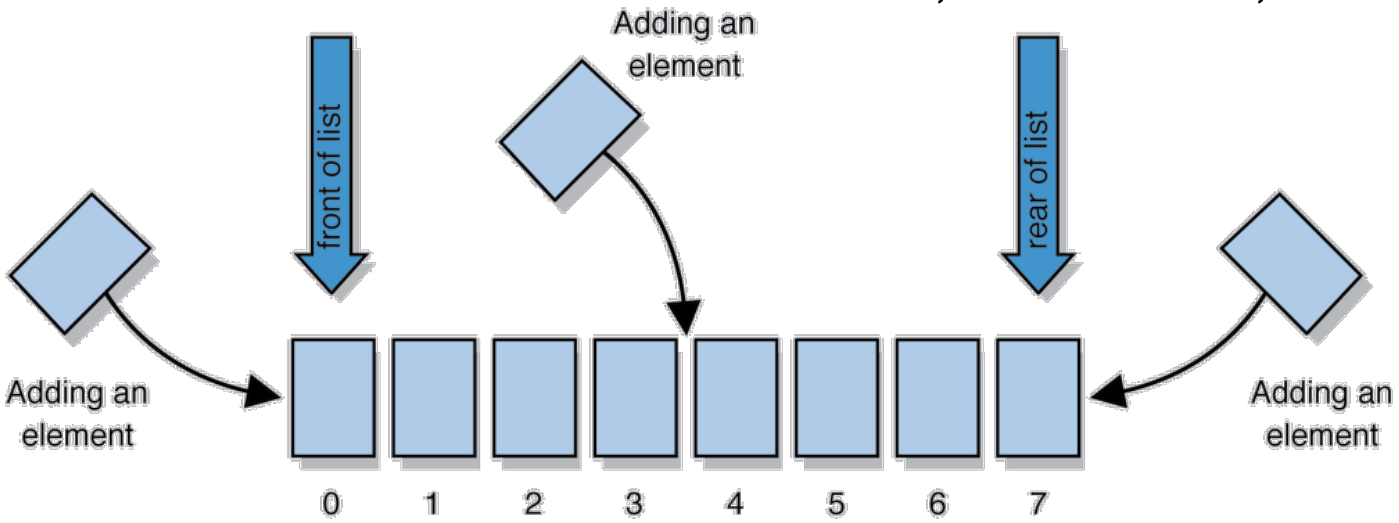
- Implementation of key methods will vary based on ADT specification

LISTS IN JAVA

CS 5004, SPRING 2024 – LECTURE 4

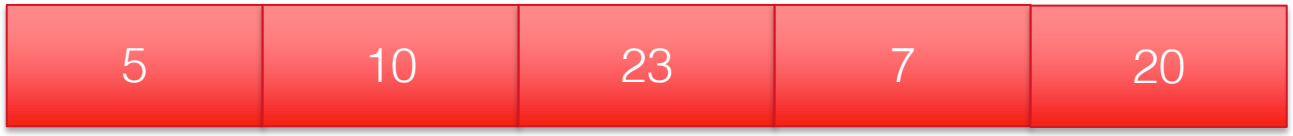
LISTS

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

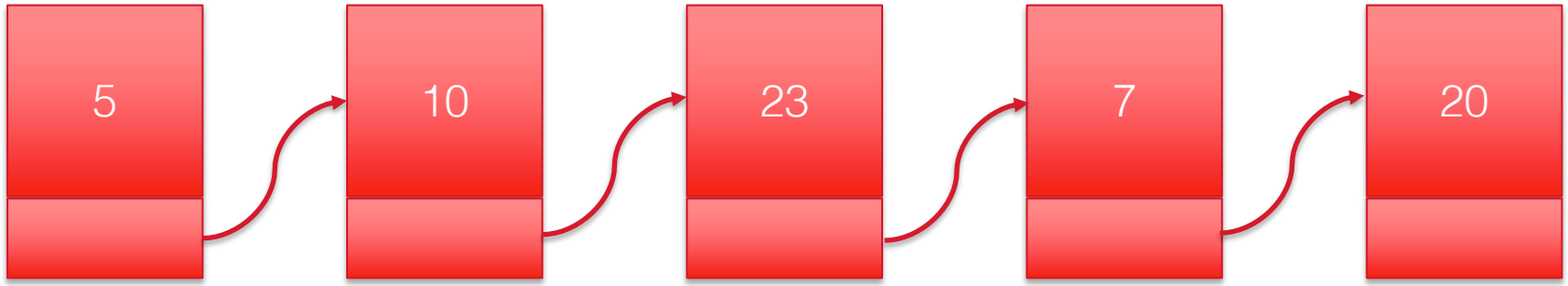


LISTS AND LINKED DATA STRUCTURES

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

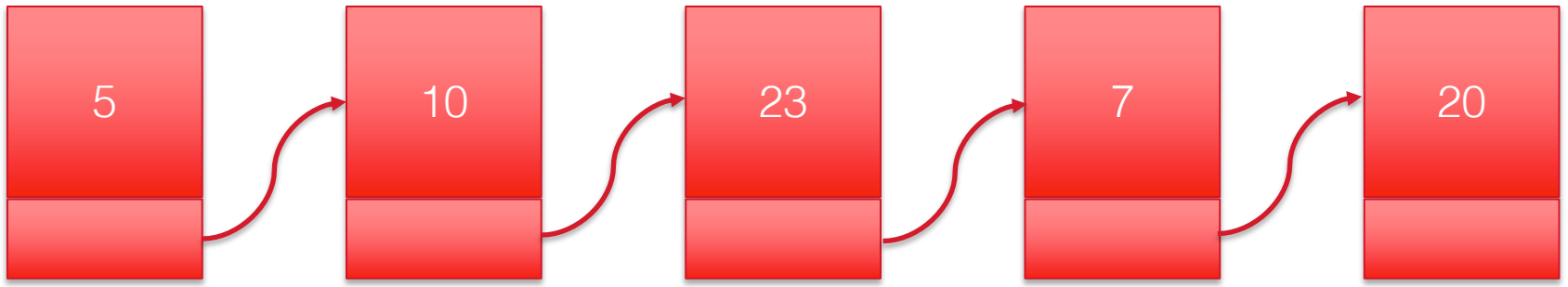


  - A collection of linked objects, each storing one element, and one or more references to other elements



LINKED LIST

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

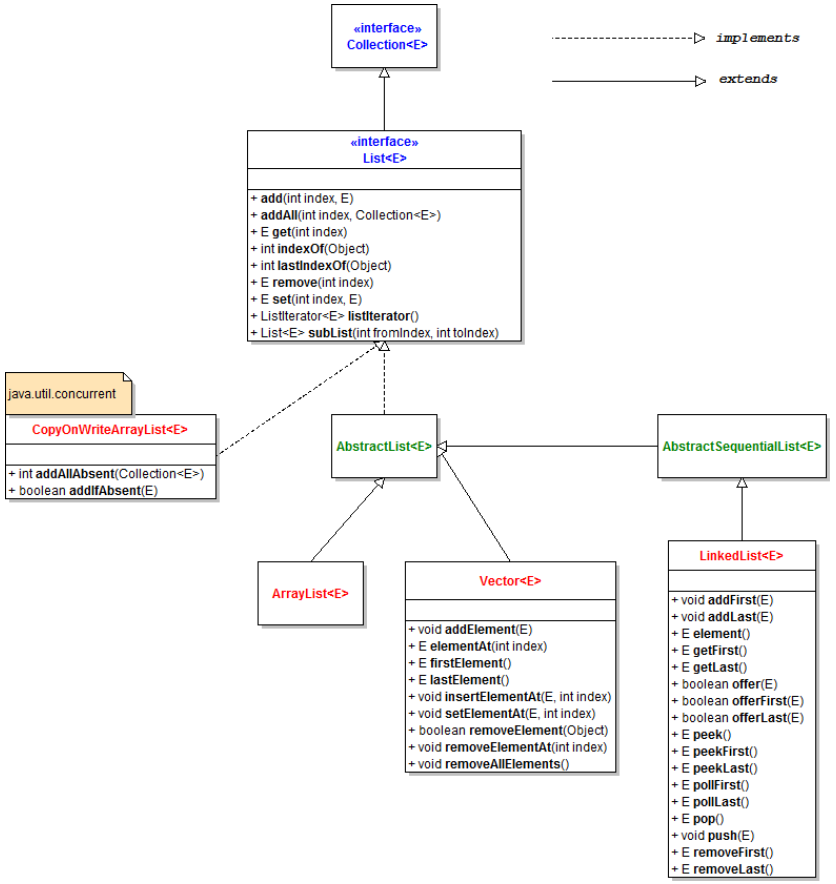


ARRAYLIST VS. LINKED LIS

- List - a collection of elements with 0-based indexes
  - Elements can be added to the front, back, or in the middle
  - Can be implemented as an ArrayList or as a LinkedList

  - What is the complexity of adding an element to the front of an:
    - ArrayList?
    - LinkedList?

JAVA LIST API



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

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

JAVA LIST API

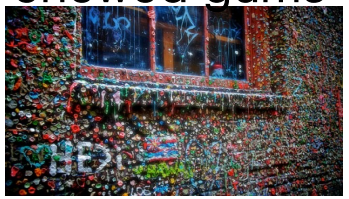
- `List<E>` - the base interface
- Concrete classes:
  - `ArrayList<E>`
  - `LinkedList<E>`
- Main methods:
  - `E get(int index);`
  - `E set(int index, E newValue);`
  - `Void add(int index, E x);`
  - `Void remove(int index);`
  - `ListIterator<E> listIterator();`

JAVA COLLECTIONS FRAMEWORK

CS 5004, SPRING 2024 – LECTURE 4

DATA COLLECTIONS

Collection of chewed gums



Collection of pens

Collection of cassette tapes

Collection of old radios

What is a data collection?

Shoes collection

Star wars collection

Cars collection

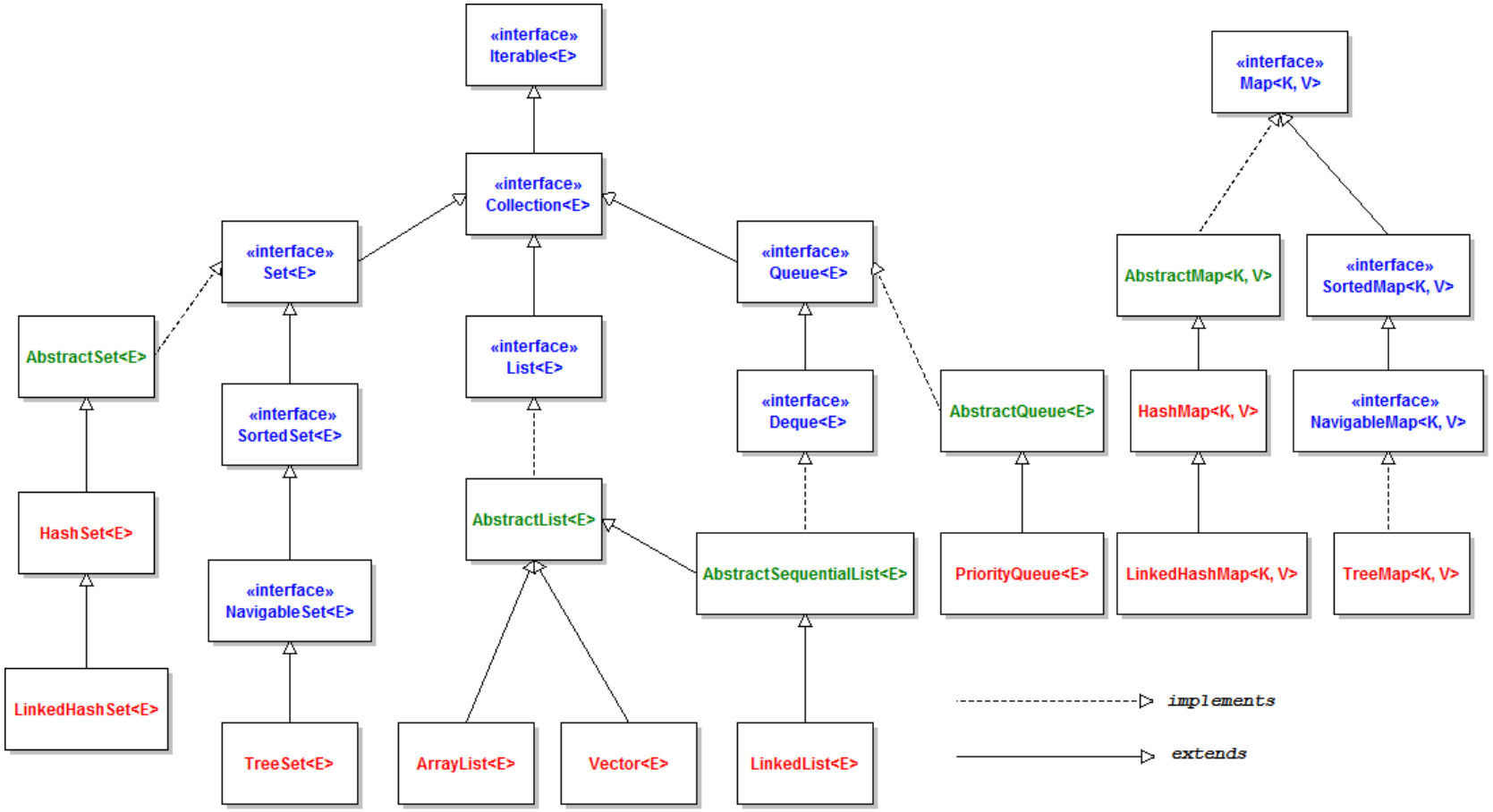
[Pictures credit: http://www.smosh.com/smosh-pit/articles/19-epic-collections-strange-things]

DATA COLLECTIONS?

- Data collection - an object used to store data (think data structures)
  - Stored objects called elements
  - Some typically operations:
    - `add()`
    - `remove()`
    - `clear()`
    - `size()`
    - `contains()`
- Some examples: ArrayList, LinkedList, Stack, Queue, Maps, Sets, Trees

Why do we need different data collections?

JAVA COLLECTIONS API



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

JAVA COLLECTIONS FRAMEWORK

- Part of the `java.util` package

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

STACK ADT

CS 5004, SPRING 2024 – LECTURE 4

JAVA CLASS STACK

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

- Example:

```
Stack<String> s = new Stack<String>();
s.push("Hello");
s.push("PDP");
S.push("Fall 2017"); //bottom ["Hello", "CS 5004", "Spring 2020"] top
System.out.println(s.pop()); //Spring 2020
```

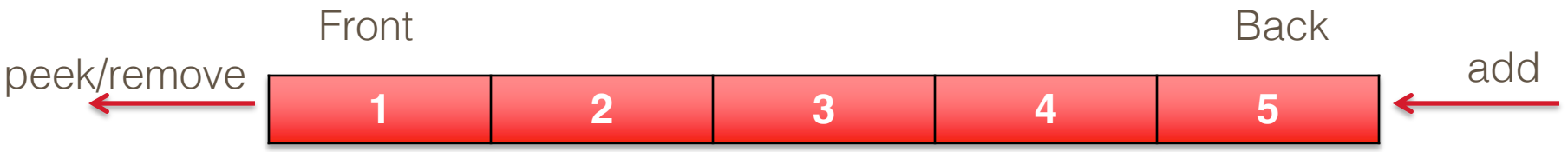
QUEUE ADT

CS 5004, SPRING 2024 – LECTURE 4

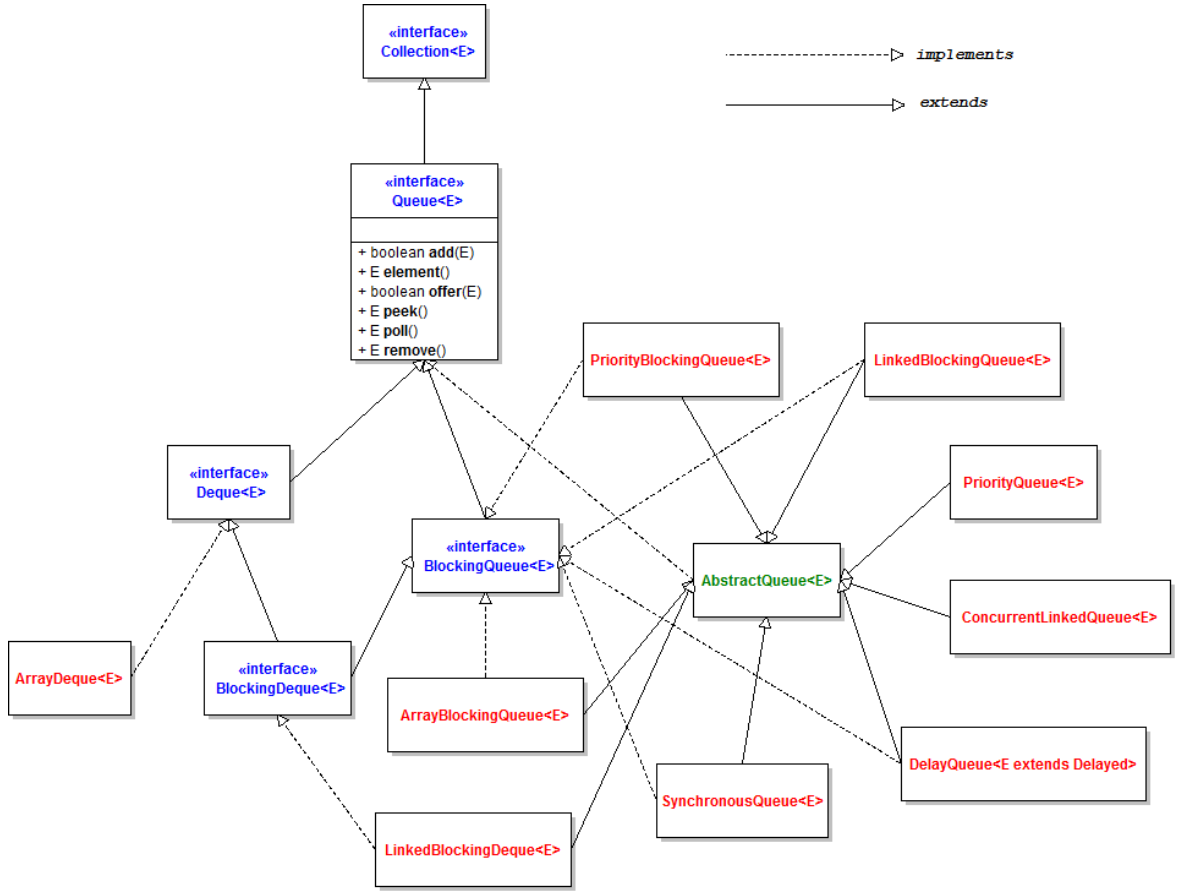
WHAT IS A QUEUE?

- Queue – a data collection that retrieves elements in the FIFO order (first in, first out)
  - Elements are stored in order of insertion, but don't have indexes
    - Client can only:
      - Add to the end of the queue,
      - Examine/remove the front of the queue

- Basic queue operations:
  - `Add (enqueue)` - add an element to the back of the queue
  - `Peek` - examine the front element
  - `Remove (dequeue)` - remove the front element



CLASS DIAGRAM OF THE QUEUE API



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

JAVA INTERFACE QUEUE

`add(value)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`peek()`	returns front value from queue without removing it; r eturns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

- Example:

```
Queue<Integer> myQueue = new LinkedList<Integer>();
myQueue.add(10);
myQueue.add(18);
myQueue.add(2017); // front [10, 18, 2017] back
System.out.println(myQueue.remove()); // 10
```

DEQUE ADT

CS 5004, SPRING 2024 – LECTURE 4

DEQUE

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

DEQUE

	First Element (Head)	
	Throws exception	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`
Remove	`removeFirst()`	`pollFirst()`
Examine	`getFirst()`	`peekFirst()`

	Last Element (Tail)	
	Throws exception	*Special value*
Insert	`addLast(e)`	`offerLast(e)`
Remove	`removeLast()`	`pollLast()`
Examine	`getLast()`	`peekLast()`

[Pictures credit:https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html

DEQUE

Queue Method	Equivalent Deque Method
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

[Pictures credit:https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html

DEQUE

Stack Method	Equivalent Deque Method
`push(e)`	`addFirst(e)`
`pop()`	`removeFirst()`
`peek()`	`peekFirst()`

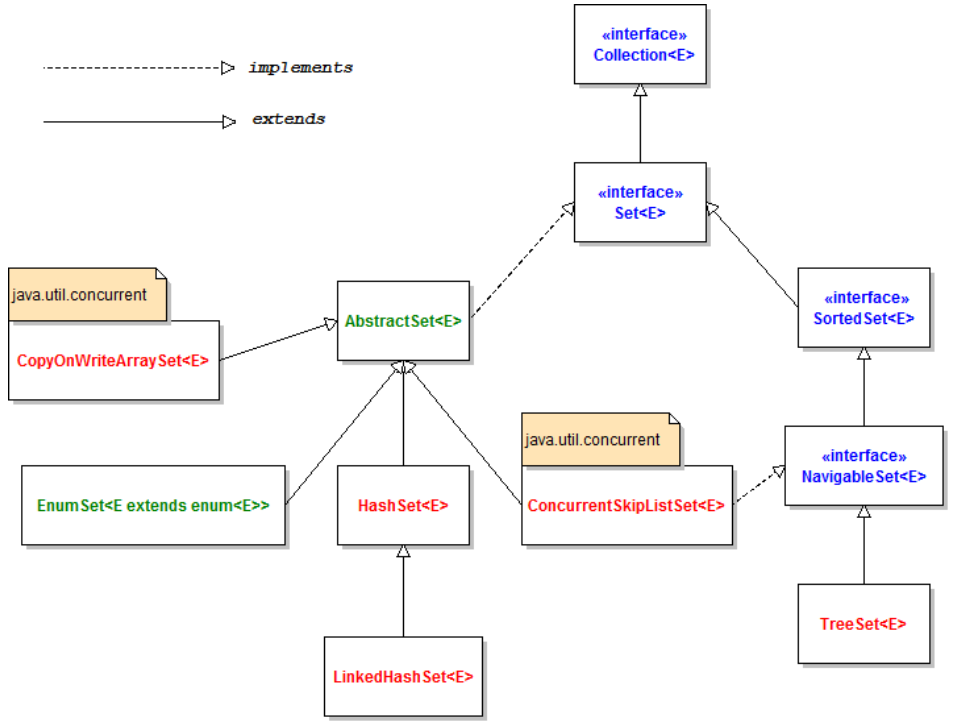
SET ADT

CS 5004, SPRING 2024 – LECTURE 4

SET

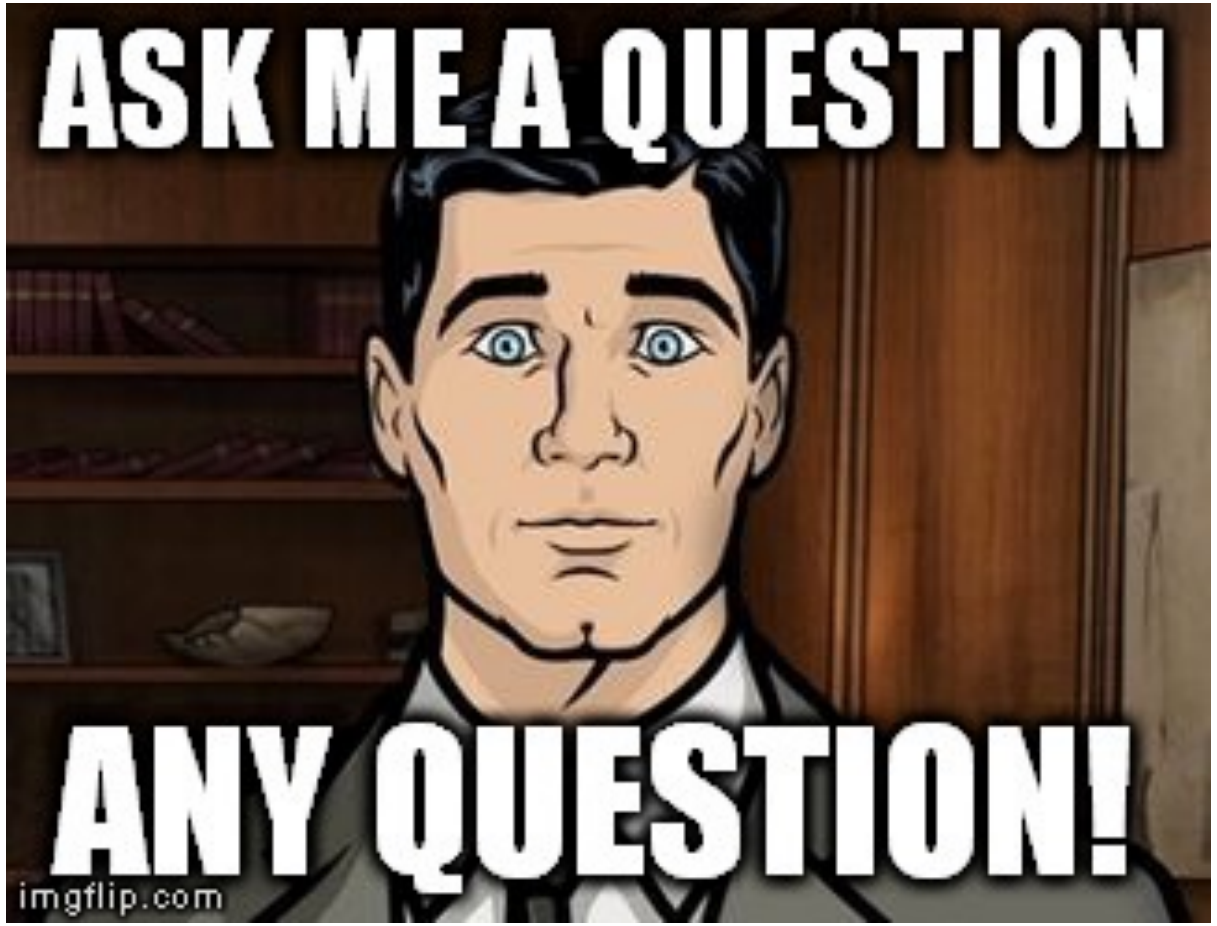
- Set - a collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - `add`,
  - `remove`,
  - `search (contains)`
- We don't think of a set as having indexes; we just add things to the set in general and don't worry about order

SET API CLASS DIAGRAM



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

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)