GCIS-124
Software Development & Problem Solving

5.1: Abstract Data Types

RIT | Golisano College of Computing and Information Sciences

SUN	MON (2/12)	TUE	WED (2/14)	THU	FRI (2/16)	SAT
Unit 4: GUIs			Midterm		Unit 5: Data Structures I	
	Assignment 4.1 Due (start of class)		Midterm Exam 1 (Units 1-3) Written Practical		Unit 4 Mini-Practicum Assignment 4.2 Due (start of class)	
SUN	MON (2/19)	TUE	WED (2/21)		FRI (2/23)	SAT
Unit 5: Data Structures I					Unit 6: Data Structures II	
			Assignment 5.1 Due (start of class)		Unit 5 Mini-Practicum Assignment 5.2 Due (start of class)	

You Are Here

5.0 Accept the Assignment

You will create a new repository at the beginning of every new unit in this course. Accept the GitHub Classroom assignment for this unit and clone the new repository to your computer.

- Your instructor will provide you with a new GitHub classroom invitation for this unit.
- Upon accepting the invitation, you will be provided with the URL to your new repository, click it to verify that your repository has been created.
- You should create a `SoftDevII` directory if you have not done so already.
    - Navigate to your `SoftDevII` directory and clone the new repository there.
- Open your repository in VSCode and make sure that you have a terminal open with the **PROBLEMS** tab visible.

Much of the content we are going to talk about today is ***review*** from the first course in this sequence.

SPOILER ALERT

Some of it will be discussed ***very quickly*** unless there are questions. If you have a question, ***please ask!***

Data Structures

You are already familiar with many data structures ***from the perspective of a user*** including arrays, queues, stacks, lists, sets, and dictionaries.

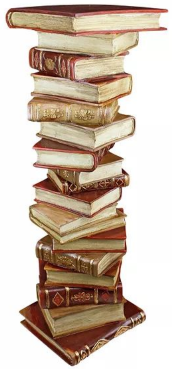
In this unit we will learn how ***different implementations*** of many of these data structures work.

We will also examine the situational ***strengths and weaknesses*** of each implementation in terms of ***complexity*** (both space and time).

- In this unit we will begin exploring ***abstract data types***.
  - An abstract data type describes the behavior of a data structure from the perspective of its user without providing any implementation details.
  - Every abstract data type (ADT) may be implemented in more than one way.
  - When solving a new problem, choosing the most efficient implementation is just as important as choosing the correct data structure.
- We will examine several different abstract data types, and implement each in multiple ways, including:
  - Queues
  - Generics
  - Lists
  - Iterators
  - The Java Collections Framework (JCF)
- Today we will specifically focus on a data structure with which you are already familiar: the ***queue***.

Abstract Data Types

- A ***data structure*** is a grouping of related data.
  - We refer to the data as ***elements***.
- Most data structures provide some common operations including ***store***, ***retrieve***, ***remove***, and ***size***.
- An ***abstract data type*** (***ADT***) defines the behavior of a data structure from the point of view of its ***user***.
  - This includes the possible values and the operations available.
- Choosing the ADT that provides operations suitable for solving a problem usually isn't enough.
  - Each ADT may be implemented in ***multiple ways***.
  - Choosing the ***correct implementation*** is also important.
  - This requires understanding the ***complexity*** of each operation under different circumstances.



Choosing the correct ***abstract data type*** involves understanding the nature and constraints of the problem being solved.

Choosing the correct ***implementation*** involves understanding which ***operations*** will be used the most, and which of the available options provides the ***most efficient implementation*** of those operations.

5.1 Identify Abstract Data Types

An ***abstract data type*** describes the capabilities of a data structure from the perspective of its user without revealing any implementation details. Read each description and write the name of the abstract data type that best fits the description.

A ***first-in, first-out*** (***FIFO***) data structure; elements are removed in the same order that they were added. Does not provide random access.	Stores ***key/value pairs***. Once added, a key can be used to retrieve its corresponding value. Keys are unique; if the same key is used more than once, the previous value is ***replaced***. Keys are not maintained in any particular order.	A ***last-in, first out*** (***LIFO***) data structure. Elements are removed in the opposite order that they were added. Does not provide random access.	Stores ***unique*** elements, meaning that duplicates are ***ignored***. Elements are not maintained in any particular order.	A ***variable length*** data structure that grows in size as elements are added, and shrinks as they are removed. Elements are maintained in the order that they are added, and can be accessed using an index.

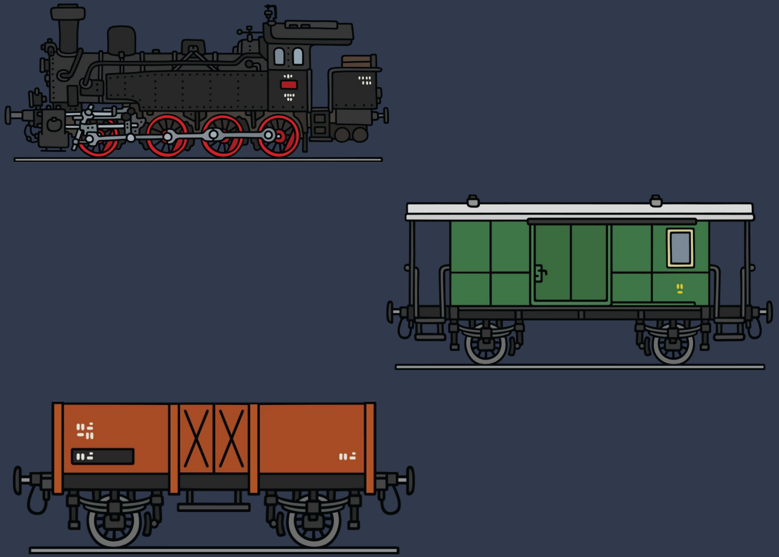
5.1 Identify Abstract Data Types

An ***abstract data type*** describes the capabilities of a data structure from the perspective of its user without revealing any implementation details. Read each description and write the name of the abstract data type that best fits the description.

A ***first-in, first-out*** (***FIFO***) data structure; elements are removed in the same order that they were added. Does not provide random access.	Stores ***key/value pairs***. Once added, a key can be used to retrieve its corresponding value. Keys are unique; if the same key is used more than once, the previous value is ***replaced***. Keys are not maintained in any particular order.	A ***last-in, first out*** (***LIFO***) data structure. Elements are removed in the opposite order that they were added. Does not provide random access.	Stores ***unique*** elements, meaning that duplicates are ***ignored***. Elements are not maintained in any particular order.	A ***variable length*** data structure that grows in size as elements are added, and shrinks as they are removed. Elements are maintained in the order that they are added, and can be accessed using an index.
QUEUE	DICTIONARY	STACK	SET	LIST

Recursive Data Structures

- By now, you all know that a ***recursive function*** is one that calls itself.
- Similarly, a ***recursive data structure*** is one that comprises a class (or classes) that has at least one field ***of its own type***.
    - These kinds of data structures can be used to build ***linked sequences***.
    - A linked sequence can be used as the basis for implementing many different data structures.
- The simplest kind of recursive data structure is the ***node***, which typically has two fields:
    - A value of some type, e.g. an int.
    - A reference to the next node in the sequence (which may be null).
- A sequence of nodes can be created by joining the nodes together - each of which refers to the next node in the sequence.
    - The first node is referred to as the ***head***.
    - The last node is referred to as the ***tail*** - the tail's next node is `null`.



Let's examine how nodes work by using an analogy that most of you should intuitively understand: ***train cars***.

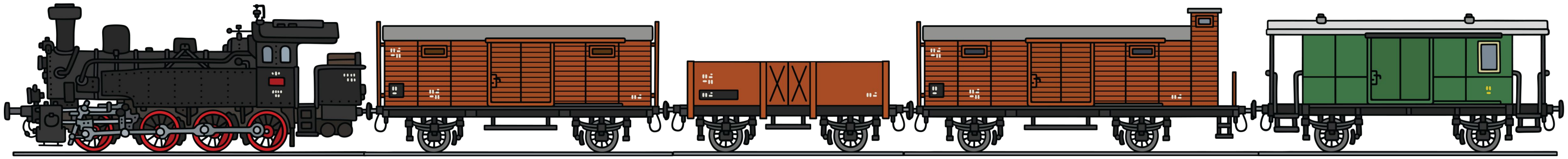
Visualizing a Linked-Sequence of Nodes

Every train car carries something of ***value*** inside of it, e.g. cargo, passengers, or the engineer.

Each train car is connected to the ***next*** car in the train.

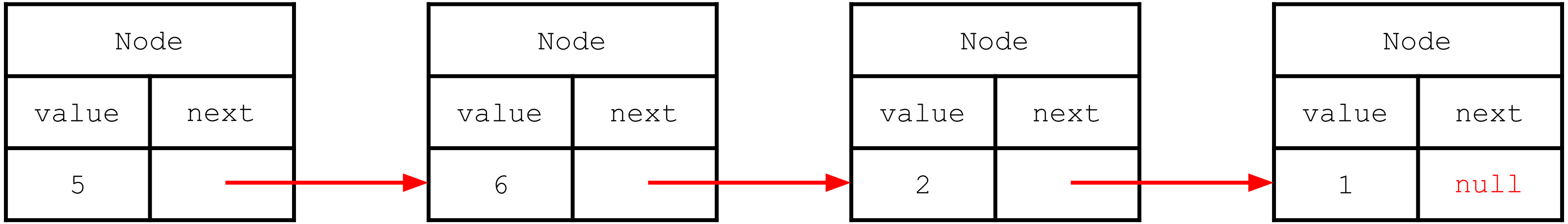
Together, the ***sequence*** of cars form the train.

The first car is the ***head*** and the last car is the ***tail*** of the sequence. The tail has no next car.



head

tail



`Node`

`value`	`next`
`5`	

`Node`

`value`	`next`
`6`	

`Node`

`value`	`next`
`2`	

`Node`

`value`	`next`
`1`	`null`

Every node has a ***value*** as well, e.g. an `int` or a `String`.

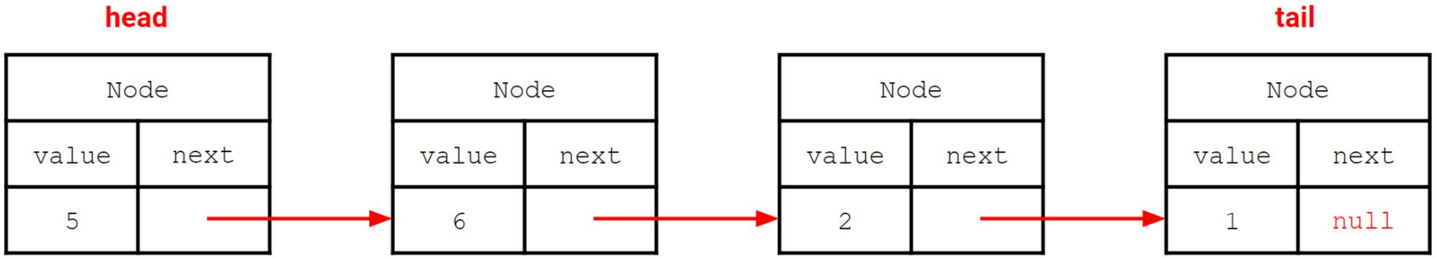
Each node has a ***reference*** to the next node in the sequence that connects them together.

Together the nodes form a ***sequence of nodes***.

The first node is the ***head*** and the last node is the ***tail*** of the sequence. The tail has no next node.

5.2 Visualizing a Sequence of Nodes

The diagram below is an example of a "box-and-pointer" visualization of the sequence of nodes containing the values `5`, `6`, `2`, and `1`. Drawing diagrams like this can help to understand how the objects in a sequence work and are connected together. Try it now!



- Using a sheet of paper and a pen or pencil (or a whiteboard!), draw a separate "box-and-pointer" diagram for-each of the following sequences. Be sure to label the head and the tail of each sequence:
    - `1`
    - `2, 3`
    - `5, 7, 11, 13`
- Now add a new value to the end of each sequence. How does the "tail" in each sequence change?

5.3 A Node Class

We will be using ***nodes*** to implement some of the abstract data types that we talk about in this unit. The first step to creating any node-based data structure is to create a `Node` class.

```
Node
-------------------------------------
- value: String
- next: Node
-------------------------------------
+ Node(value: String)
+ Node(value: String, next: Node)
+ setValue(value: String)
+ getValue(): String
+ setNext(next: Node)
+ getNext(): Node
```

- Create a new package named "`unit05.mcf`" (short for "my collections framework") and use it for all of the code that you write in this unit unless otherwise instructed.
- Create a new Java class named "`Node`".
    - Use the UML diagram to the left as a guide to implementing your `Node` class.
- The string representation of a node should match the following format: `"<value> -> <next>"`
    - e.g. `"5 -> 4 -> null"`
- Define a `main` method with an appropriate signature and use it to test your new `Node` class.

Queues

- A ***queue*** is an abstract data type that provides at least the following operations:
    - ***enqueue*** - adds an element to the ***back*** of the queue.
    - ***dequeue*** - removes an element from the ***front*** of the queue.
    - ***size*** - returns the number of elements currently in the queue.
- A queue ***may*** also provide additional operations:
    - ***front*** - returns but does not remove the element at the front of the queue.
    - ***back*** - returns but does not remove the element at the back of the queue.
    - ***is empty*** - returns `true` if the queue is empty, and `false` otherwise.
- A queue stores its elements in ***First-In, First-Out*** (***FIFO***) order.
    - The ***first*** element to be enqueued is the ***first*** element to be dequeued.
    - The ***last*** element to be enqueued is the ***last*** element to be dequeued.
- The queue abstract data type ***defines*** its behavior from the perspective of its user, but does not include ***implementation*** details.
    - What Java feature can be used to ***define*** required behavior without providing an ***implementation***?

A real-world analog for a queue is a line at the bank, a grocery store checkout, or the DMV.

People get into line as they arrive, entering at the ***back***...

The person that is ***first-in*** the line is the person that is ***first-out*** when the next employee is available.

Another way to say this is that the person that has been in line the ***longest*** is the next to leave.

When people do leave, they leave from the ***front*** of the line.

5.4 The Queue Abstract Data Type

Remember: the Queue abstract data type should ***define*** the behavior that all queues have without describing the ***implementation***. A Java interface is the ideal mechanism for defining behavior without implementing it. Create an interface for a Queue.

<<interface>>
Queue

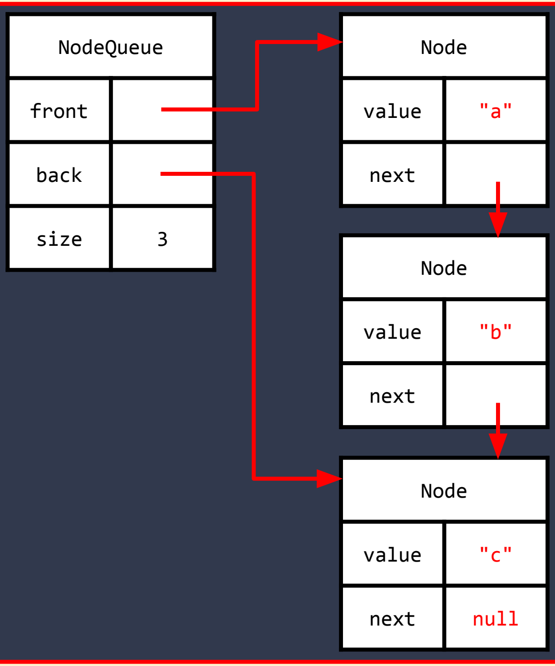
+ enqueue(value: String)
+ dequeue(): String
+ size(): int

Use the UML diagram as a guide to create an interface to represent a Queue.

- Use the UML to the left as a guide to create a new Java interface named "`Queue`" in the mcf package.

Node-Based Queues

A ***node-based queue*** uses linked nodes to store values. The ***oldest*** node is the ***front***...



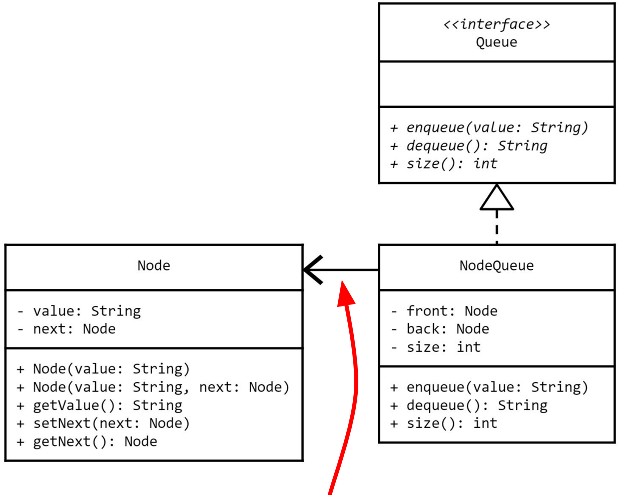
...and the ***newest*** node is the ***back***.

Any remaining values added in between form a ***linked sequence*** between the front and back.

- One possible implementation of a queue is built using a ***linked sequence***.
- You should recall that a ***linked sequence*** is defined as:
    - The ***empty sequence***, which contains no values and is represented as `null`.
    - At least one ***node*** that contains:
        - A ***value*** of some type, e.g. an `String`.
        - A reference to the ***next*** node in the sequence, which may be `null`.
- A ***node-based queue*** keeps track of three values.
    - The node at the ***front*** of the queue.
    - The node at the ***back*** of the queue.
    - The ***size*** of the queue - the total number of nodes including the front and the back.
- There are some special conditions that need to be considered:
    - If the queue is empty, ***both*** the front and back are `null`.
    - If there is only ***one*** value in the queue, ***both*** the front and back refer to the ***same*** node.

5.5 A Node-Based Queue

Our first implementation of a Queue will use Nodes to store the values that are added to the Queue. Begin your implementation of a Node-based Queue.



<<interface>>
Queue

+ enqueue(value: String)
+ dequeue(): String
+ size(): int

Node

- value: String
- next: Node

+ Node(value: String)
+ Node(value: String, next: Node)
+ getValue(): String
+ setNext(next: Node)
+ getNext(): Node

NodeQueue

- front: Node
- back: Node
- size: int

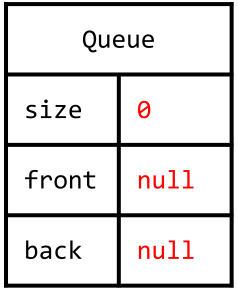
+ enqueue(value: String)
+ dequeue(): String
+ size(): int

In UML, an ***association*** indicates that one class ***uses*** another in some way, e.g. as a field. This is indicated with a solid line ending in an open arrow.

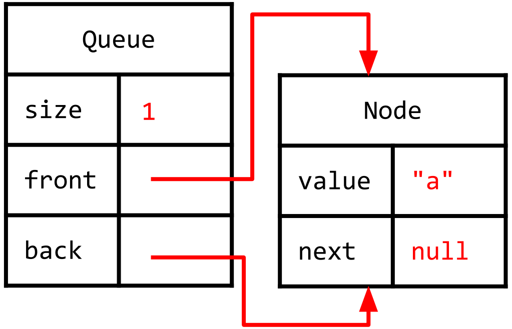
- Create a new Java class named "`NodeQueue`".
  - Use the UML to the left as a guide to begin implementing your `NodeQueue`.
  - For now focus on ***fields*** and ***constructors***.
  - Implement the `size` method.
  - ***Stub-out*** any other methods required by the `Queue` interface.
- Implement a toString() method that returns a string in the format `"<size>, <front>"`
  - e.g. `"2, 5 -> 4 -> null"`.
- Define a `main` method with the appropriate signature and use it to create an instance of the new class and print it to standard output.

A Closer Look at Enqueuing

An ***empty*** queue has a ***size*** of 0, and both the ***front*** and ***back*** are `null`.

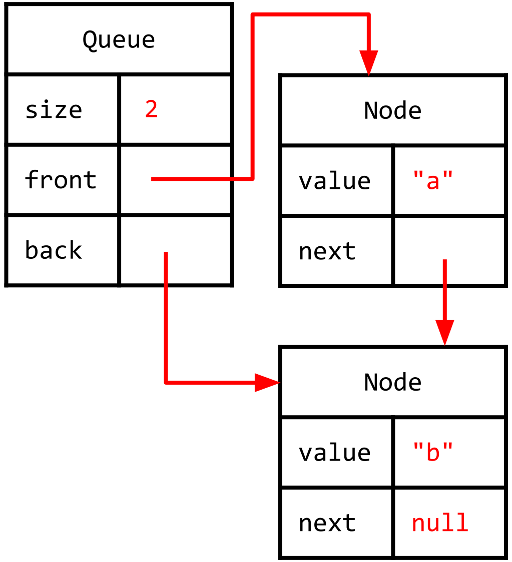


When the ***first*** value is ***enqueued***, a new ***node*** is created to hold the value.



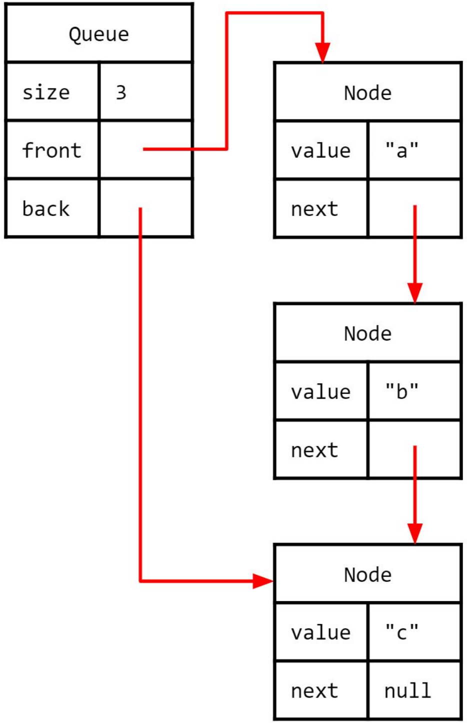
Both the front ***and*** the back are changed to refer to the new node, and ***size*** is incremented.

As each new value is enqueued, a new node is created and becomes the ***new back*** of the queue.



5.6 Enqueue Using Nodes

The first essential functionality that we will implement on the Node-based Queue is the enqueue method, which adds new elements to the back of the queue.



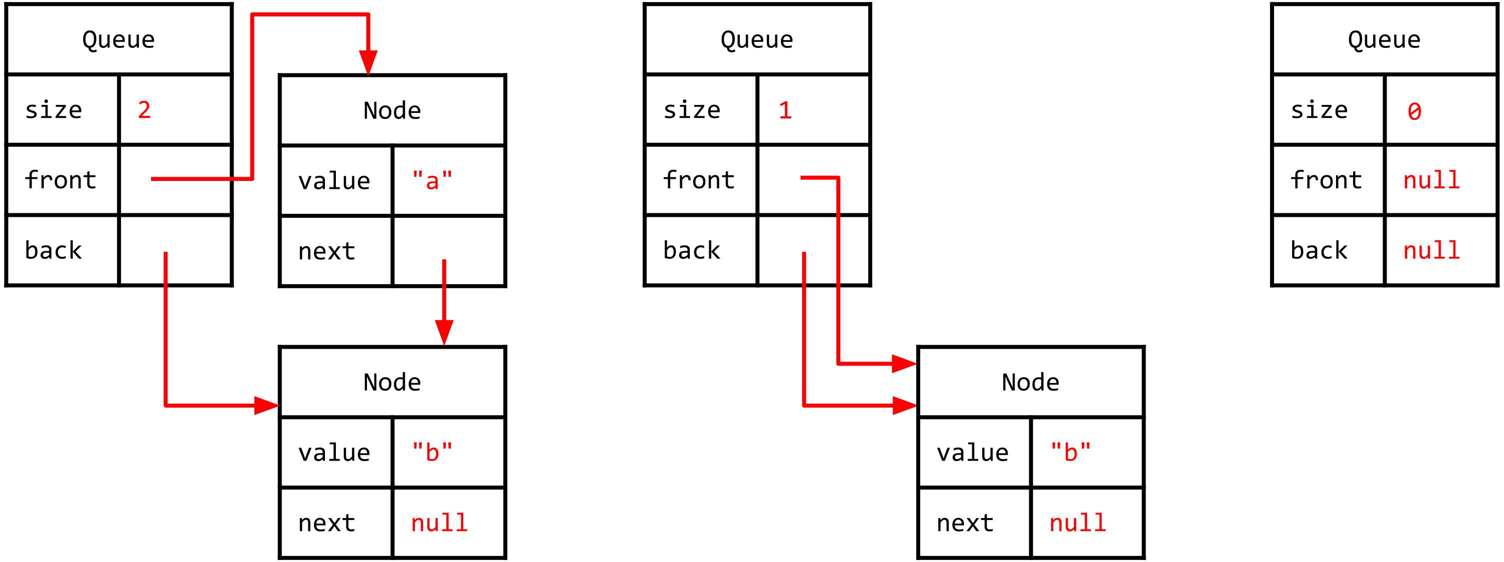
- Open your `NodeQueue` class and implement the `enqueue` ***method***.
    - Make a ***new*** `Node` with the value that is being enqueued..
    - If the `front` is `null`, the new `Node` becomes the ***new*** `front` ***and*** `back` of the queue.
    - Otherwise, the new `Node` is the ***new*** `back` of the queue.
    - Don't forget to ***increment*** `size`!
- Use `main` to enqueue a few values and print the queue.

A Closer Look at Dequeuing

When a value is ***dequeued***, it is always removed from the ***front*** of the queue.

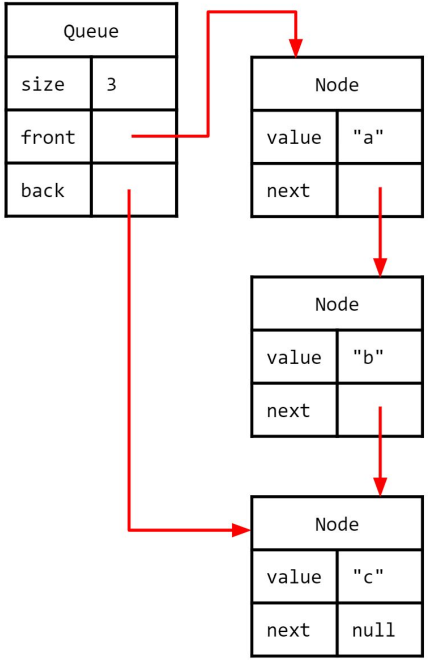
The ***new*** front is changed to the ***old*** front's next node, and ***size*** is decremented.

If the front's next node is `null`, then the queue is empty; ***both*** the front ***and*** back are set to `null`.



5.7 Dequeue Using Nodes

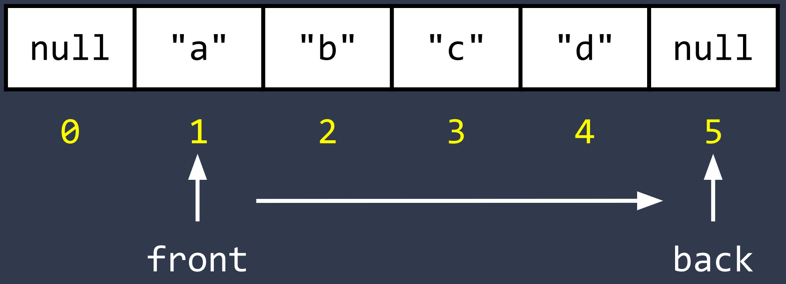
We can add new elements to the queue, which is great! But we can't actually remove anything from the queue, which is not so great. Finish implementing the Node-based Queue by added a dequeue method.



- Open your `NodeQueue` class and implement the `dequeue` ***method***.
    - Save the `front` node's ***value*** in a ***temporary variable***.
    - Make the ***new*** `front` the ***old*** `front`'s ***next*** node.
    - If the new `front` is `null`, ***set*** `back` ***to*** `null` as well!
    - Don't forget to ***decrement*** `size`!
    - ***Return*** the temporary variable.
- Use `main` to dequeue a few values and print them out. Print your queue as well.

Array-Based Queues

An ***array-based queue*** uses an ***array*** to store the elements that are added to the queue.



null	"a"	"b"	"c"	"d"	null
0	1	2	3	4	5

Fields are used to keep track of the ***front*** and ***back*** index in the array.

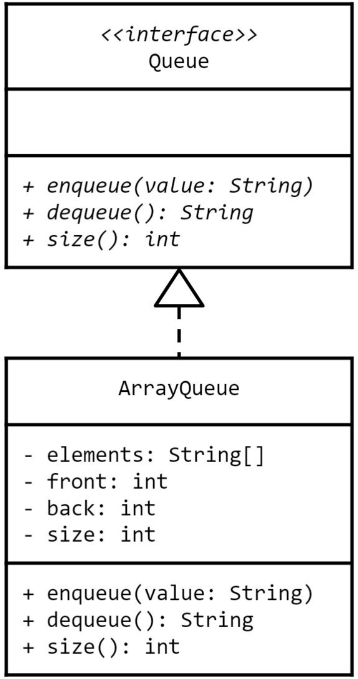
Each time an element is ***enqueued*** or ***dequeued***, one of the indexes is incremented; ***the "head" chases the "tail."***

When either index reaches the end of the array, it will need to ***wrap around*** to the beginning.

- The ***queue*** abstract data type ***defines*** the following behavior:
  - Elements are ***enqueued*** at the ***back***.
  - Elements are ***dequeued*** from the ***front***.
  - The order in which elements are added is maintained - a queue is a ***FIFO*** data structure.
- Most data structures can be implemented in more than one way.
  - The above description does not include any information about the ***implementation*** of the queue.
  - So far we have implemented a queue using a ***linked sequence*** of ***nodes***, but that is only one possible implementation of a queue.
- Another possible implementation stores elements in a ***circular array***.
  - Separate indexes for the ***front*** and ***back*** of the queue are saved as fields.
  - As elements are ***enqueued***, the back index is incremented.
  - As elements are ***dequeued***, the front index is incremented.
- From the perspective of the user, both implementations work ***exactly the same***.
  - The implementation details are ***hidden*** behind the `Queue` interface.
  - Code written to use the interface will work with ***either implementation***.

5.8 An Array-Based Queue

We've just finished implementing a Node-based Queue, but most abstract data types can be implemented in more than one way. Different implementations may offer performance benefits (and drawbacks) in some situations. Let's try implementing a Queue using an array.



<<interface>>
Queue

+ enqueue(value: String)
+ dequeue(): String
+ size(): int

ArrayQueue

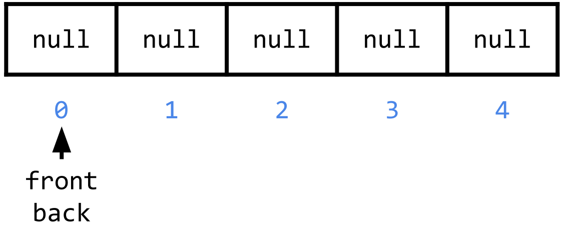
`- elements: String[]`
`- front: int`
`- back: int`
`- size: int`

`+ enqueue(value: String)`
`+ dequeue(): String`
`+ size(): int`

- Create a new Java class named "`ArrayQueue`".
  - Use the UML to the left as a guide to begin implementing your `ArrayQueue`.
  - For now focus on ***fields*** and ***constructors***.
  - Implement the `size` method.
  - ***Stub-out*** any other methods required by the `Queue` interface.
- The string representation of an ArrayQueue should match the format "`<size>, <array>`".
  - ***Hint***: use `Arrays.toString(array)`
  - e.g. "`3, [3, 2, 7, 0, 0, 0]`"
- Define a `main` method and use it to test your queue.

A Closer Look at Enqueuing

As you know, an ***array*** is created to be some fixed size, and Java fills it with a ***default value***, e.g. `null`.

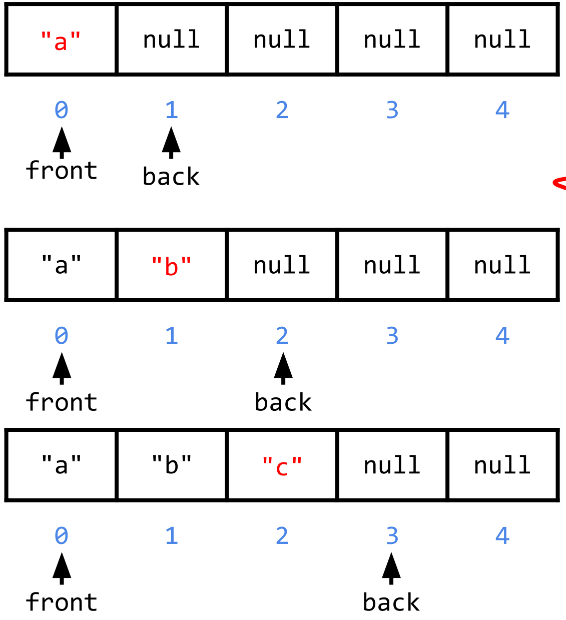


null	null	null	null	null
0	1	2	3	4

Initially both ***front*** and ***back*** indexes point to the same index: `0`.

The ***size*** starts at `0` and is incremented each time an element is ***enqueued***.

When a new element is ***enqueued***, it is added at the ***back*** index, and the ***back*** index is incremented.

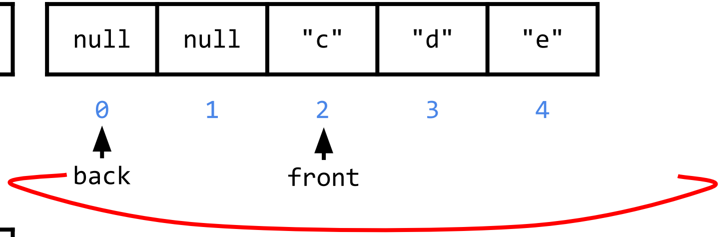


"a"	null	null	null	null
0	1	2	3	4

"a"	"b"	null	null	null
0	1	2	3	4

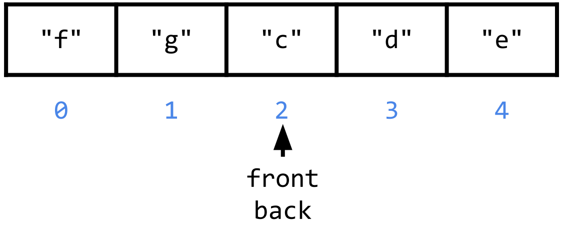
"a"	"b"	"c"	null	null
0	1	2	3	4

If after incrementing the ***back*** index is equal to the array length, it ***wraps around*** to the beginning.



null	null	"c"	"d"	"e"
0	1	2	3	4

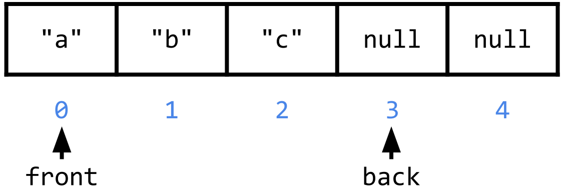
If the ***size*** is equal to the ***length*** before an enqueue, the ***array is full*** and will need to be ***resized***.



"f"	"g"	"c"	"d"	"e"
0	1	2	3	4

5.9 Enqueue Using an Array

Although enqueuing into an Array-based Queue can be fairly straightforward in most cases, some edge cases can be relatively complicated. Try implementing enqueue using an array.

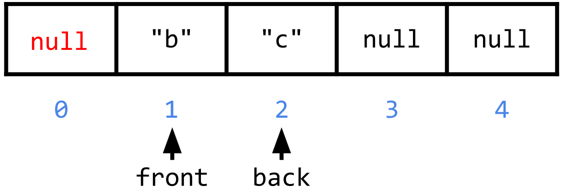


"a"	"b"	"c"	null	null
0	1	2	3	4
front			back	

- Open your `ArrayQueue` class and implement the `enqueue` ***method***.
    - ***Add*** the new element to the `back` ***index***.
    - ***Increment*** the `back` index. Be sure to ***wrap around*** if necessary.
        - ***Hint***: use modulo with the length of the array.
    - Don't forget to ***increment*** `size`.
    - For now ***do not worry*** about resizing (yet).
- ***Compile*** your class.
- ***Don't worry*** about error handling at this point.
    - What potential error(s) may occur?
    - How might you handle the error(s) if they do occur?
- Use `main` to enqueue a few values and print the queue.

A Closer Look at Dequeuing

When ***dequeuing***, the element at the ***front*** index is returned and the front index is ***incremented***.

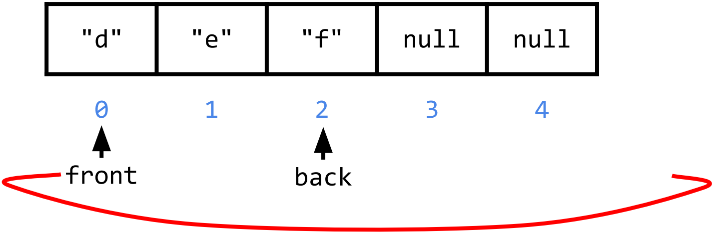


null	"b"	"c"	null	null
0	1	2	3	4
	front	back		

It is important to set the value at the ***front*** index to `null` ***before*** the index is incremented.

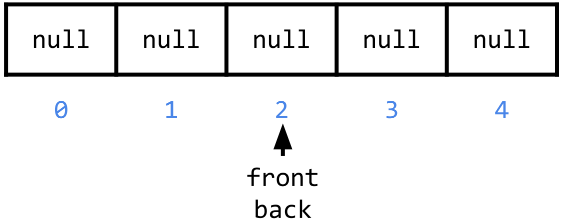
This is because keeping a reference to the value will prevent it from being ***garbage collected***.

If after incrementing the ***front*** index is equal to the array length, it ***wraps around*** to the beginning.



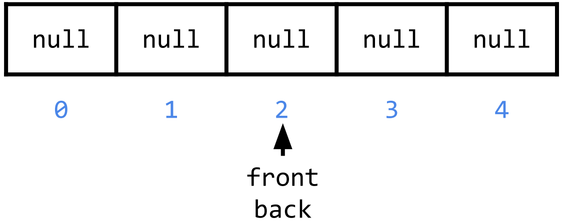
"d"	"e"	"f"	null	null
0	1	2	3	4
front		back		

If the `size` is `0` after a dequeue, the queue is ***empty***.



null	null	null	null	null
0	1	2	3	4
		front back		

There are at least two ways to handle calling ***dequeue*** on an ***empty queue***...



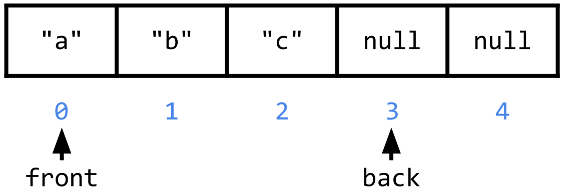
null	null	null	null	null
0	1	2	3	4
		front back		

The easiest is to simply ***return*** `null`. But this obfuscates the problem because it is possible to ***enqueue*** a `null` value.

Alternatively, an ***exception*** may be thrown.

5.10 Dequeue Using an Array

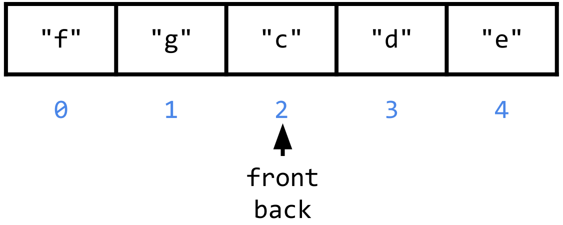
Dequeuing from an Array-based Queue is also usually straightforward and uses logic similar to the enqueue method for edge cases. Try implementing the dequeue method on your queue now.



- Open your `ArrayQueue` class and implement the `dequeue` ***method***.
  - Save the element at the `front` ***index*** in a ***temporary variable***.
  - Set the value at the `front` index to `null`. Why?
  - ***Increment*** the `front` index. Be sure to ***wrap around*** if necessary.
  - Don't forget to ***decrement*** `size`!
  - ***Return*** the temporary variable.
- ***Compile*** your class.
- ***Don't worry*** about error handling at this point.
  - What potential error(s) may occur?
  - How might you handle the error(s) if they do occur?
- Use `main` to dequeue a few values and print them out. Print your queue as well.

A Closer Look at Resizing

If ***before enqueuing*** the ***size*** is equal to the ***length*** of the array, then the ***array is full***.



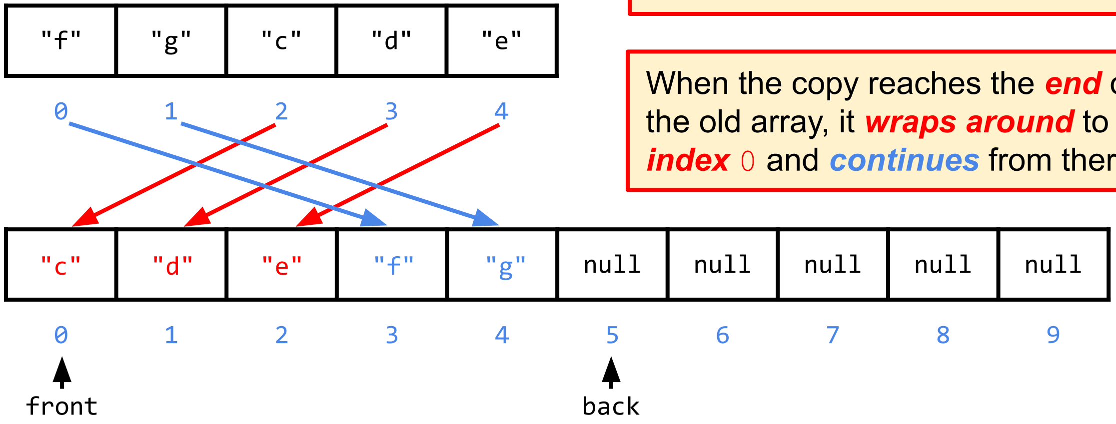
At this point, the array will need to be ***resized*** in order to make room for additional elements.

This will require making a ***new***, ***bigger array***, and ***copying*** the elements into it.

First, a new, bigger array is created (usually ***double the size*** of the current array).

Then, beginning at the ***front*** index in the ***old*** array, elements are copied into the ***new*** array beginning at ***index*** `0`.

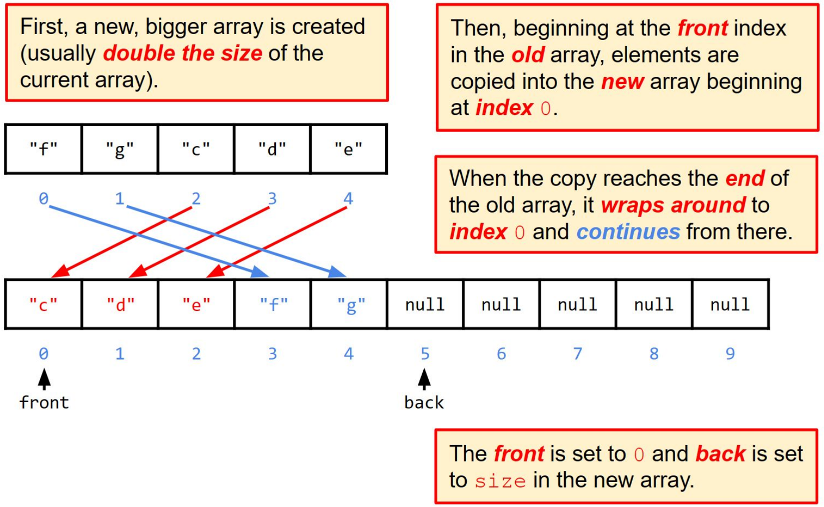
When the copy reaches the ***end*** of the old array, it ***wraps around*** to ***index*** `0` and ***continues*** from there.



The ***front*** is set to `0` and ***back*** is set to `size` in the new array.

5.11 Resizing

A Node-based Queue never "fills up" because we can always add another node to the back. But arrays are fixed-size, and so we may eventually run out of space. The solution is to make a new, larger array and copy the elements into it.



First, a new, bigger array is created (usually ***double the size*** of the current array).

Then, beginning at the ***front*** index in the ***old*** array, elements are copied into the ***new*** array beginning at ***index*** `0`.

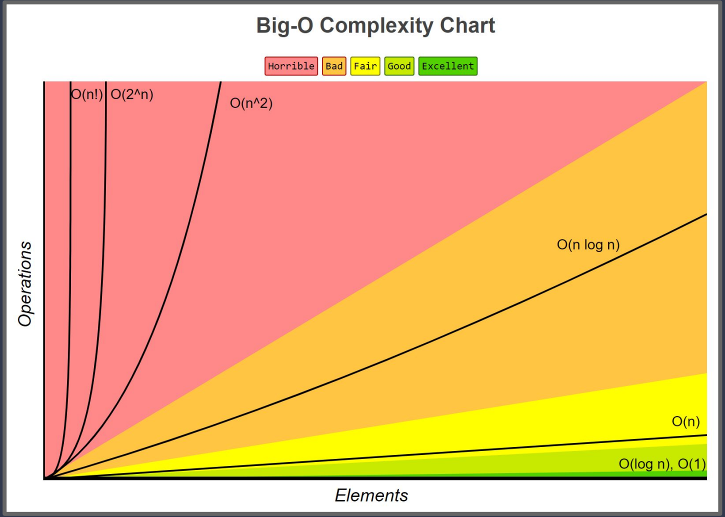
When the copy reaches the ***end*** of the old array, it ***wraps around*** to ***index*** `0` and ***continues*** from there.

The ***front*** is set to `0` and ***back*** is set to `size` in the new array.

- Open your `ArrayQueue` class and navigate to the `enqueue` ***method*** and modify it to resize the array ***if the array is full***.
  - Make a new array that is double the size.
  - Use a loop to copy beginning ***from the*** `front` ***index*** in the old array and ***into index*** `0` in the new array.
    - ***Hint***: The `destination` index starts at `0` and ends at `size-1`
    - ***Hint***: The `source` index is `(front + destination) % size`
  - Don't forget to set `front` to `0` and `back` to `size`!
- ***Compile*** your class.

Queue Complexity

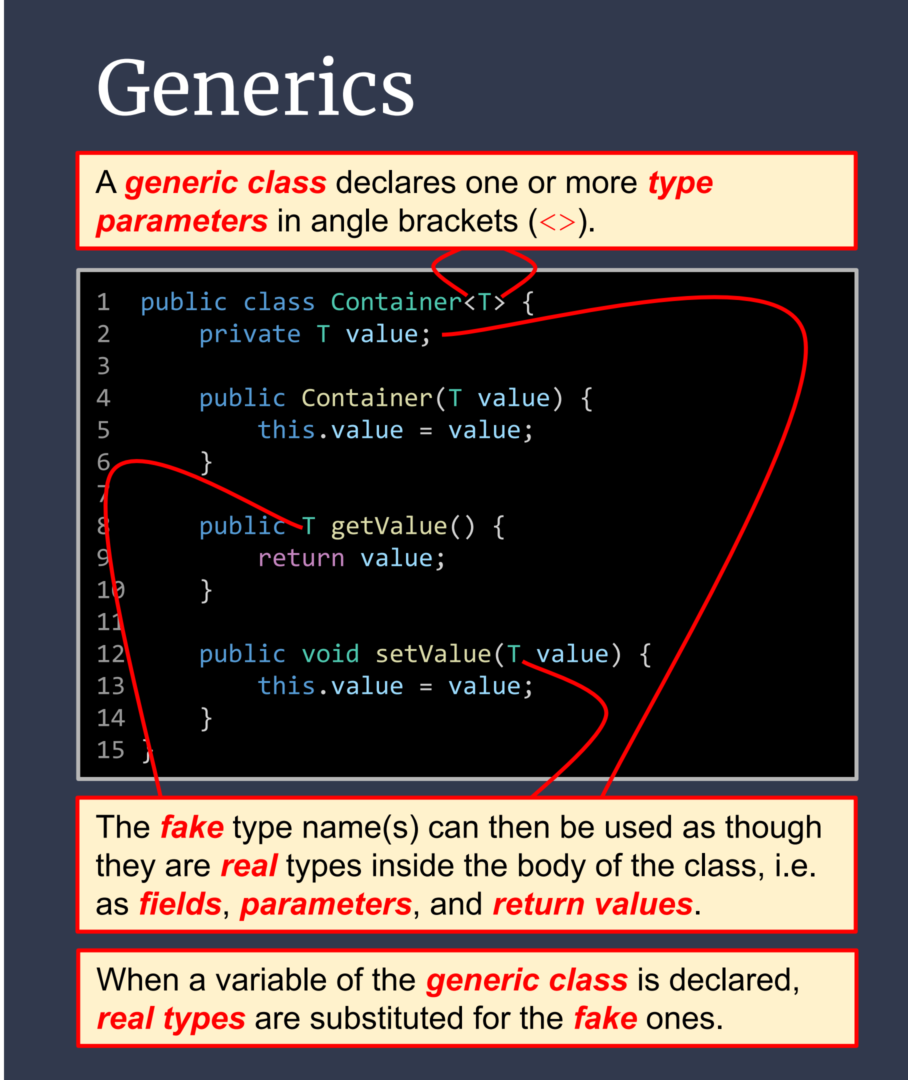
- Today, we have discussed two different implementations of the ***queue*** abstract data type: ***node-based*** and ***array-based***.
  - From the perspective of the user, there is no difference between the ***behavior*** of the two implementations.
  - Choosing which implementation to use requires a deep understanding of the strengths and weaknesses of each under different circumstances.
- The expected ***time complexity*** of all operations on ***both*** queue implementations is ***constant time*** (***O(C)***).
  - In fact, this is also the ***worst case*** time complexity for ***all*** operations on a ***node-based*** queue.
- The ***worst case*** time complexity for ***enqueue*** on an ***array-based*** queue occurs when the ***array is full***.
  - This requires that the elements be ***copied*** into a new array; a ***linear time*** operation (***O(n)***).
  - ***Doubling*** the size of the array each time it is resized will ensure that the copy operation is ***rare***.
  - If the maximum number of elements is known, the array may be ***pre-allocated*** to avoid copying entirely.
- The ***space complexity*** of ***both*** implementations is different.
  - Creating a ***node object*** for every element in a node-based queue uses more memory than an array.
  - On the other hand, the ***array-based*** queue ***usually*** over-allocates its array, and so uses more memory.



Generally, when there is a tradeoff between time complexity (***speed***) and space complexity (***memory***), we prefer time complexity.

That being said, there is so little difference between the time complexity of both implementations (except in rare circumstances) that ***either implementation is appropriate*** for most problems.

- The queue interface and implementations that we wrote last time only work with `String`s.
    - If we wanted a queue that worked with `double`s, or `int`s, or `Goat`s, we would need to make a new interface and implementation by basically ***copy/pasting***, and doing a ***search & replace*** with the new type.
- It turns out that Java has a feature that does essentially the same thing without having to write a new class each time: ***generics***.
- A ***generic type*** declares one or more ***type parameters*** between ***angle brackets*** (<>).
    - `public class Foo<Bar, Bel> {...}`
    - `public interface Operation <N>{ ... }`
    - The type parameters should ***never*** use the name of a real type, e.g. `String`, `Scanner`, etc.
    - The type parameter can be used inside the class as though it is a real type, e.g. as ***variables***, ***parameters***, ***return types***, etc.
- When a variable of the generic class is declared, a ***real type*** is specified in place of each type parameter, e.g. `Operation<String> myClass;`
    - The ***Java compiler*** makes sure that ***only*** the type(s) specified when the variable is declared are used with ***this instance*** of the generic class.



Generics

A ***generic class*** declares one or more ***type parameters*** in angle brackets (<>).

```
public class Container<T> {
    private T value;

    public Container(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

The ***fake*** type name(s) can then be used as though they are ***real*** types inside the body of the class, i.e. as ***fields***, ***parameters***, and ***return values***.

When a variable of the ***generic class*** is declared, ***real types*** are substituted for the ***fake*** ones.

A Closer Look at Generics

A ***generic type*** declares one or more ***type parameters*** with ***fake*** type names.

```java
public class Container<T> {
    private T value;

    public Container(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

The ***fake*** type names can then be used in the class as though they are a ***real type***, i.e. as ***fields***, ***parameters***, or ***return values***.

When a variable of the generic type is declared and/or instantiated, a ***real type*** is specified for-each type parameter...

Note that the ***diamond operator*** (<>) can be used on the ***right*** side of an assignment (the types are ***inferred*** from the ***left***).

```java
Container<String> c = new Container<>("abc");


c.setValue("def");
c.setValue(123);
```

From that point forward, only the specified type can be used ***with that specific instance*** of the generic type.

Trying to use a ***different type*** with the ***same instance*** will cause a ***compiler error***.

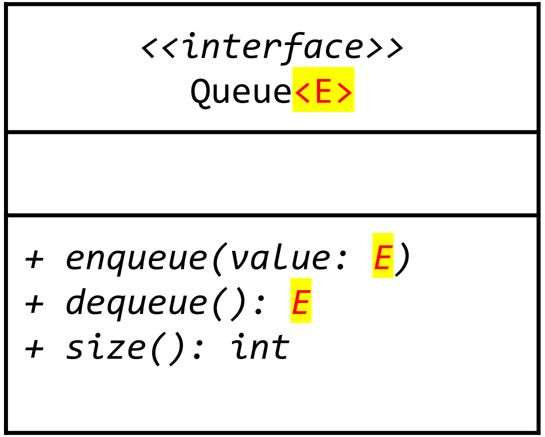
When ***each instance*** is created with a real type, it is as though the Java compiler does a ***search and replace***...

```java
public class Container< String > {
  private  String  value;

  public Container( String value) {
     this.value = value;
  }

  public  String  getValue() {
        return value;
    }

  public void setValue( String value) {
     this.value = value;
  }
}
```

...and the ***type parameter*** is replaced with the ***real type*** wherever it occurs in the class.

5.12 Making Queue Work With any Type

The `Queue` interface that we used to implement the node and array-based queues only works with strings. Thankfully, Java's ***generics*** syntax allows us to define classes and methods that will work with *any* type. Let's modify our existing classes to use generics instead of strings.



- Open the `Queue` interface and convert it into a ***generic type***.
    - Add a ***type parameter*** to the interface declaration.
        - While you can use any fake name that you'd like, it is customary to use `E` (short for ***element***) as the name of the type parameter in a generic data structure.
    - Replace instances of the `String` type with the type parameter where appropriate.
- ***Compile*** your interface to make sure that it is syntactically correct.
    - Note that your queue implementations will no longer compile unless they are updated!

Combining Generics

A generic type may use ***its own type parameter*** when declaring ***variables of a generic type*** rather than specifying some real type.

```java
public class Warehouse<T> {
    private Container<T> container;

    public Warehouse(T value) {
        container = new Container<>(value);
    }

    public Container<T> getContainer() {
        return container;
    }
}
```

When a ***real type*** is provided for the type parameter, the same type is ***propagated*** to the variables in the class that use the same type parameter.

```java
Warehouse<String> wh = new Warehouse<>("abc");
Container<String> c = wh.getContainer();
String value = c.getValue();
```

- When a variable of a generic type is declared within the body of a class, it is ***usually*** the case that a ***real type*** must be provided for its ***type parameter(s)***.
    - e.g. `Container<String> container;`
- One common exception to this rule is ***within the body*** of a generic type.
    - The generic type may use ***its own type parameter*** when declaring variables, fields, parameters, or return values of a generic type.
    - When a real type is provided for ***its*** own type parameter, it is ***propagated*** to any variables within the class that use the ***same*** type parameter.
- Currently our `Node` implementation only works with `String`s.
    - If we are to make a `NodeQueue` that can hold any type of element, we will need to make the `Node` class generic as well.
    - As usual, this will involve adding a type parameter to the `Node` class declaration, and we will replace all references to the `String` type with the type parameter.
    - Different nodes in the same linked sequence will always store the same type of values, and so we will want to use the same type parameter for the `next Node`.

5.13 Making Nodes Work With any Type (too)

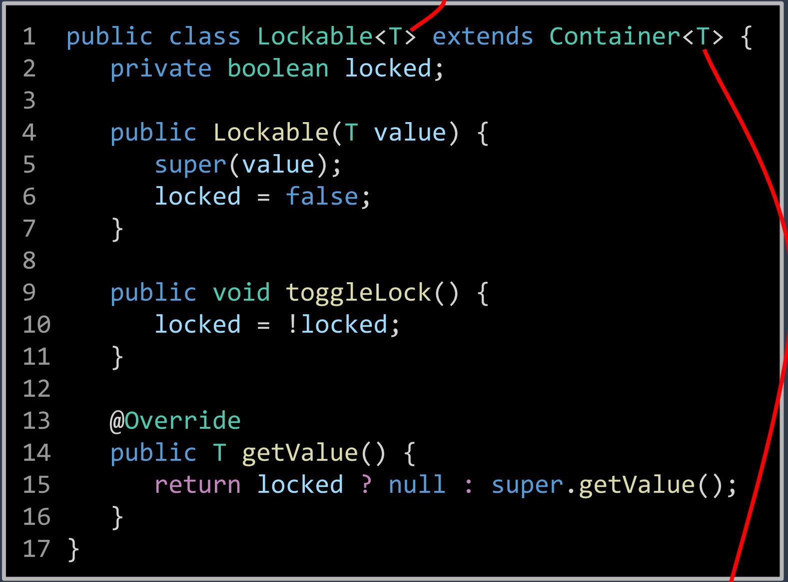
The `Node` class can only store *string* values. If we want to use `Node` to store *any* kind of value, we'll need to make it ***generic***. Then we'll be able to use it to implement a generic, Node-based Queue.

```
Node<E>
---------------------------------------
- E: value
- next: Node<E>
---------------------------------------
+ Node(value: E)
+ Node(value: E, next: Node<E>)
+ getValue(): E
+ setNext(next: Node<E>)
+ getNext(): Node<E>
```

- Open the `Node` class and modify it into a ***generic type*** that can store any kind of value (not just strings).
  - Add a ***type parameter*** to the class declaration.
    - Replace instances of the `String` type with the type parameter where appropriate.
  - Note that a nodes in a ***linked-sequence*** will all be of the ***same type***, and so the reference to the next node should also be generic, i.e. `private Node<E> next`
  - The compiler will make sure that two nodes can only be linked together if the ***real types*** match.
- Use your `main` method to create and print a few nodes that use types other than strings.
  - What happens when you use a primitive type like `int`?
  - What happened to your `NodeQueue`?

Extending Generics

A ***type parameter*** cannot be used within the body of a class unless it is declared as part of the class.



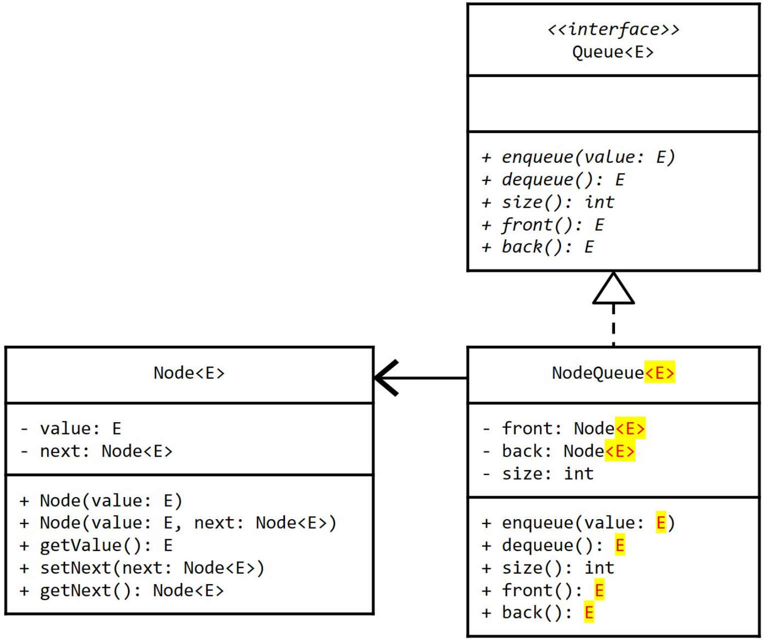
```
public class Lockable<T> extends Container<T> {
   private boolean locked;

   public Lockable(T value) {
      super(value);
      locked = false;
   }

   public void toggleLock() {
      locked = !locked;
   }

   @Override
   public T getValue() {
      return locked ? null : super.getValue();
   }
}
```

The ***same*** type parameter(s) can then be used when ***extending*** or ***implementing*** another generic type.

- Type parameters ***must*** be declared along with the ***class*** or ***interface declaration*** in a generic type, i.e.
    - `public class Foo<T> {}`
    - `public interface Queue<E> {}`
- Trying to specify a type parameter anywhere else will cause a ***compiler error***, e.g.
    - `public class A extends B<T> {}`
- When extending/implementing a generic type, one option is to make the new class generic as well by ***declaring a type parameter*** and using the same type parameter with the superclass or interface, e.g.
    - `public class A<T> extends B<T> {}`
- A second option is to make the new class work ***only with a specific real type*** by specifying that type with the superclass or interface, e.g.
    - `public class A extends B<String> {}`
- While the second option is ***easier***, you will lose the flexibility that generics provide in the subclass.
    - In this example, class `A` will ***not*** be generic; it will ***only work with strings***.

5.14 Making `NodeQueue` Generic

`Node` is the fundamental building block of a Node-based queue. Now that it's generic, we can refactor the `NodeQueue` to be generic as well.



<<interface>>
Queue<E>

+ enqueue(value: E)
+ dequeue(): E
+ size(): int
+ front(): E
+ back(): E

Node<E>

- value: E
- next: Node<E>

+ Node(value: E)
+ Node(value: E, next: Node<E>)
+ getValue(): E
+ setNext(next: Node<E>)
+ getNext(): Node<E>

NodeQueue<E>

- front: Node<E>
- back: Node<E>
- size: int

+ enqueue(value: E)
+ dequeue(): E
+ size(): int
+ front(): E
+ back(): E

- Open the `NodeQueue` class and modify it into a ***generic type*** that can store any kind of value, not just integers.
  - Add a type parameter to the class declaration.
  - Use the ***same*** type parameter when implementing the generic `Queue` interface.
  - Modify all of the `Node` references to use the ***same*** type parameter.
  - Replace instances of the `String` type with the type parameter.
- Use `main` to test your updated `NodeQueue`.

- Primitive types ***cannot*** be used as the ***real type*** for a generic ***type parameter***.
    - Generics ***only*** work with ***reference types***.
- So what do you do if you need a data structure that stores integers or booleans?
- Thankfully, Java provides a set of ***wrapper classes***, each of which represents a different one of the eight primitive types.
    - In general, the name of the wrapper class is the ***capitalized***, ***fully-spelled-out name*** of the primitive, e.g. `Integer` instead of `int` and `Character` instead of `char`.
- If a generic is needed to work with a primitive type, the corresponding wrapper class should be used in place of the type parameter, e.g.
    - `Queue<Integer> intQueue;`
- Once you create an instance of a generic using one of the wrapper classes, you can use the generic with the corresponding primitive type.
    - Java will seamlessly and transparently translate from the primitive type to its wrapper class. This is called ***autoboxing***.
    - Translating back from the wrapper to the primitive is called ***unboxing***.

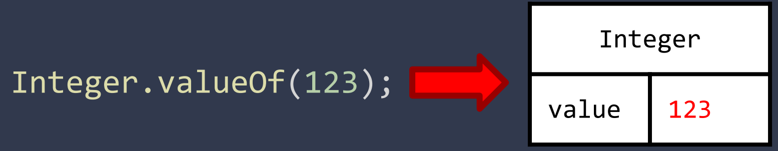
Autoboxing

(and Unboxing)

If one of the Java ***wrapper classes*** is used to create an instance of a generic type...

```
Queue<Integer> intQueue = new NodeQueue<>();

intQueue.enqueue(123);
intQueue.enqueue(456);
```

...Java will ***autobox*** primitive values of the corresponding type by transparently creating a ***new instance*** of the wrapper class with the ***same value*** to use in place of the primitive.



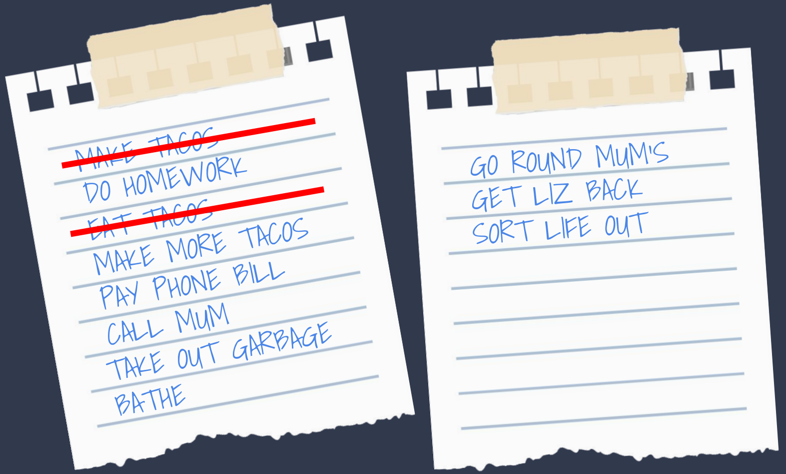
```
Integer.valueOf(123);
```

The same process also works in reverse; the primitive value is retrieved from inside the wrapper. This is called ***unboxing***.

```
int value = intQueue.dequeue();
```

Lists

A real-world analog for a list is a ***to-do list***.



The size of the list ***dynamically changes*** as new tasks are added or old tasks are completed.

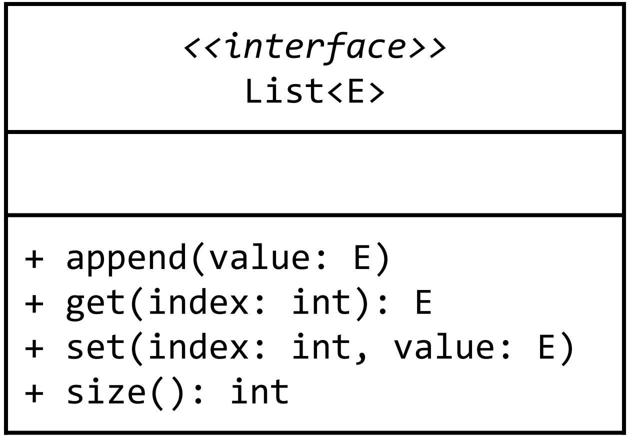
If the first page is filled up, ***additional space can be allocated*** so that more items can be added to the list.

- A ***list*** is an abstract data type that provides at least the following operations:
  - ***append*** - adds a new value to the end of the list.
  - ***get*** - returns the value at a specific index in the list.
  - ***set*** - changes the value at a specific index in the list.
  - ***size*** - returns the number of elements currently in the list.
- A list ***may*** also provide additional operations.
  - ***insert*** - inserts a new element at a specific index somewhere in the list. Elements after the specified index are ***shifted*** to the ***right***.
  - ***remove*** - removes an element from a specific index somewhere in the list. Elements after the specified index are ***shifted*** to the ***left***.
- A list maintains elements in the order in which they are added.
- A list most closely resembles a ***dynamically sized array***.
  - The size ***grows*** as elements are added and ***shrinks*** as elements are removed.
  - However, Java does not support ***operator overloading***, and so square brackets (`[]`) cannot be used.
- The above description ***defines*** the list abstract data type, but does not include ***implementation details***.

5.15 The List Abstract Data Type

PROBLEMS 37

The List abstract data type ***defines*** the behavior that all lists must provide, but doesn't ***implement*** any of that behavior. Once again, a Java interface is the perfect way to represent an ADT. This time, let's make the List work with values of any type by making it generic right from the start.



```
<<interface>>
List<E>

+ append(value: E)
+ get(index: int): E
+ set(index: int, value: E)
+ size(): int
```

- Create a new Java interface named "`List`".
    - Use the UML diagram to the left as a guide when implementing your `List` interface.
- Now is a good time to make sure that you don't have any ***problems*** in your project.
    - If you do have any errors or warnings that you are not sure how to fix, please raise your hand now!

Array Lists

Using ***arrays*** and ***type parameters*** together can feel a little ***kludgy***. Trying to create an array using the type parameter ***won't work***...

```
T[] things = new T[10]; // compiler error
```

The ~~hack~~ ***workaround*** is to create an Object array, which can then be used to store ***any*** kind of value (including the type parameter).

```
Object[] things = new Object[10];
things[0] = anyValue; // OK!
```

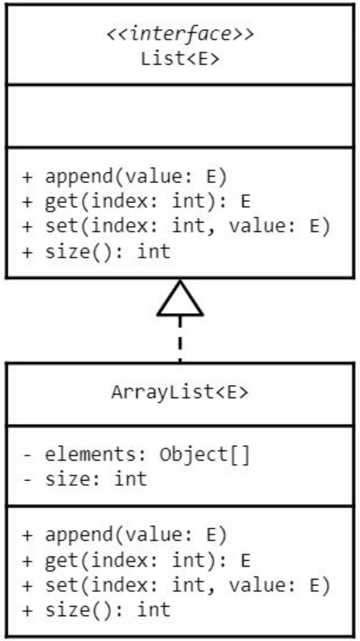
Of course this means that a ***cast*** is needed to change the type values in the array from Object to ***any other type*** (including the type parameter).

```
Object obj = things[5];
T thing = (T)obj;
```

- One possible implementation of a list is built using an ***array***.
    - The array is ***over-allocated***, meaning that it is created to be large enough to hold a non-zero number of elements even when the list is empty.
    - The ***size*** of the list indicates the number of elements currently stored in the array, which is initially 0.
    - The ***capacity*** of the array indicates that maximum number of elements that can be stored before the array is full.
- Unfortunately, in Java an array cannot be created using a type parameter.
    - e.g. `E[] e = new E[5];` will cause a ***compiler error***.
- The ***workaround*** for this is to create an `Object` ***array*** to store the elements.
    - Remember that `Object` is the parent of ***all*** reference types in the Java language, and so ***any*** type can be stored in an `Object` array.
- This means that the `get` method will need to ***cast*** the value into the correct type before it is returned.
    - e.g. `E value = (E)elements[index];`
- Otherwise the only challenge is that the array needs to be ***resized*** when ***size*** is equal to its ***capacity***.
    - Thankfully, this is pretty easy to do!

5.16 An Array-Based List

One possible implementation of the List abstract data type uses an array to store the elements in the list. The array is ***over-allocated*** to start, so that there is room to add a few elements before worrying about needing to resize (we'll handle that later).



- Create a new Java class named "`ArrayList`".
  - Use the UML to the left as a guide to begin implementing your `ArrayList`.
  - At first, focus on the ***fields*** and a ***constructor***.
    - In the constructor, create an `Object` ***array*** large enough to hold ***2 values***.
  - You should be able to easily implement the `size`, `get`, and `set` methods.
    - ***Do not*** worry about validating the `index`.
- The string representation of the `ArrayList` should match the format `"size, <array>"`.
  - For example: `"3, [2, 3, 4, 0, 0, 0]"`
  - ***Hint***: use `Arrays.toString(array)`
- Define a `main` method and use it to create an instance of your class and print it to standard output.

A Closer Look at Appending

Conveniently, the `size` field indicates both the number of elements in the list ***and*** the index at which the next element should be appended.

null	null	null	null	null
0	1	2	3	4

size=0

For example, consider an array-based list onto which ***3 elements*** have been appended...

"a"	"b"	"c"	null	null
0	1	2	3	4

size=3

The challenge occurs when the ***size*** is equal to the ***capacity*** of the array; the array must be ***resized***.

"a"	"b"	"c"	"d"	"e"
0	1	2	3	4

size=5

"a"	"b"	"c"	"d"	"e"	null	null	null	null	null
0	1	2	3	4	5	6	7	8	9

size=5

The process is much more straightforward than resizing a queue. First, create an array that is ***double the size***...

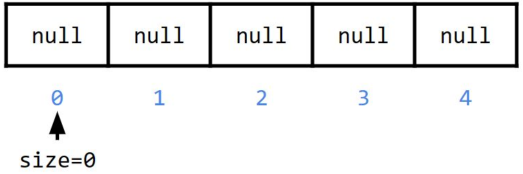
...and then ***copy*** each element into the ***same index*** in the new array. There is no need to move any values around and the size ***doesn't change***.

`Arrays.copyOf(orig, size)` is a static method that returns a ***new array*** of the specified `size` with all of the elements ***copied*** from `orig`.

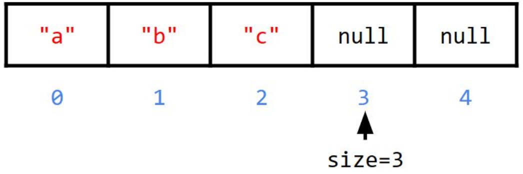
5.17 A Basic Implementation of `append`

The basic implementation of the append method is pretty straightforward in an Array-based list: simply add the new value at the index to which size refers. Things get a little trickier if the array is full, but we'll worry about that in the next activity!

Conveniently, the `size` field indicates both the number of elements in the list ***and*** the index at which the next element should be appended.



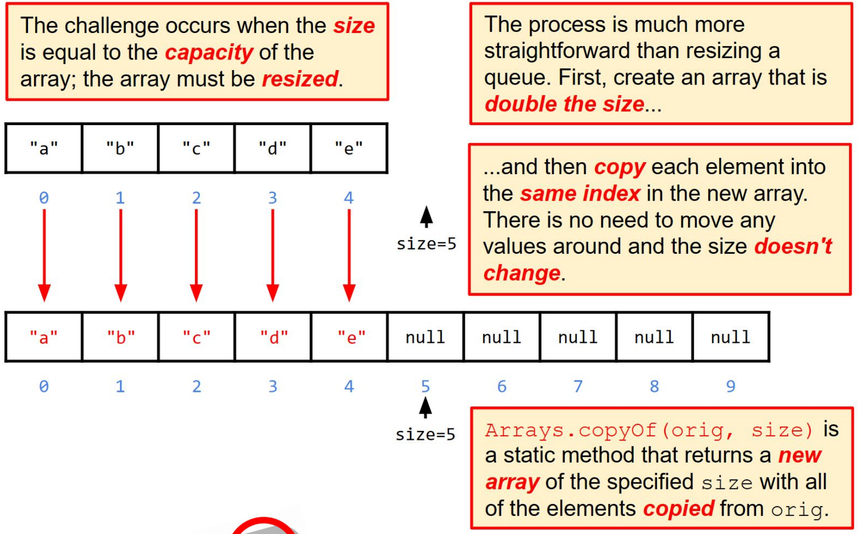
For example, consider an array-based list onto which ***3 elements*** have been appended...



- Open your `ArrayList` class and navigate to the stubbed-out `append` method.
    - Implement the method so that it adds a new value to the index to which size currently points.
    - Don't forget to increment size!
    - Don't worry about resizing the array just yet.
- Update the `main` method to append a few values to your `ArrayList` and print it to standard output.
    - Don't add more values that the array can hold!

5.18 Resizing the Array in an Array-Based List

When the ***size*** of the ArrayList is equal to the ***capacity*** of its array, that means that the array is full! The only way to add more elements to the list is to create a new, bigger array to hold all of the elements. The existing elements will need to be copied into the new array before the new element is added.



The challenge occurs when the ***size*** is equal to the ***capacity*** of the array; the array must be ***resized***.

The process is much more straightforward than resizing a queue. First, create an array that is ***double the size***...

...and then ***copy*** each element into the ***same index*** in the new array. There is no need to move any values around and the size ***doesn't change***.

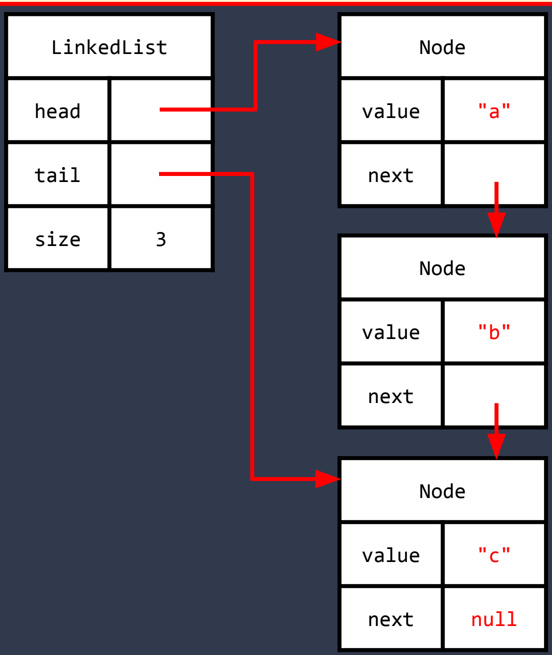
`Arrays.copyOf(orig, size)` is a static method that returns a ***new array*** of the specified `size` with all of the elements ***copied*** from `orig`.

- Open the `ArrayList` and navigate to the `append` method.
    - ***Before*** adding the new element, check to see if the ***size*** of the list is equal to the ***capacity*** of its array.
    - If so, use `Arrays.copyOf(original, newSize)` to create a bigger copy of the array ***first***.
    - Then add the element as you would normally.
- Update your `main` method to append enough values to cause the array to be resized.
    - Print the `ArrayList` to standard output!

PROBLEMS 37

Linked Lists

A ***linked list*** uses is very similar to a ***node-based queue***.

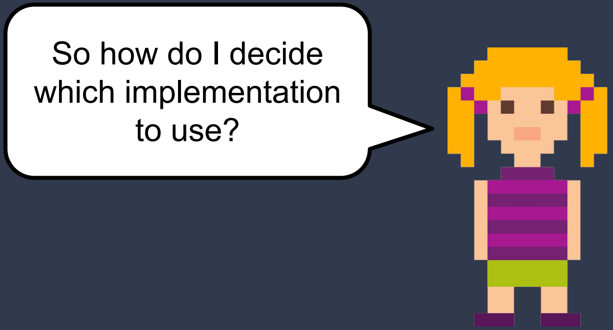


The primary difference is that a ***linked list*** access by index, meaning that the ***get*** and ***set*** methods need to ***search*** the linked sequence for the node at a specific index.

- Another possible implementation of the list ADT is built using a ***linked-sequence***.
  - This type of list is called a ***linked list***.
- You should recall that a ***linked sequence*** is defined as:
  - The ***empty sequence***, which contains no values and is represented as `null`.
  - At least one ***node*** that contains:
    - A ***value*** of some type, e.g. an `String`.
    - A reference to the ***next*** node in the sequence, which may be `null`.
- A ***linked list*** keeps track of three values:
  - The ***first*** node in the list, which is also called the ***head***.
  - The ***last*** node in the list, which is also called the ***tail***.
  - The ***size*** of the list - the total number of nodes including the head and the tail.
- There are some special conditions that need to be considered:
  - If the list is empty, ***both*** the head and the tail are `null`.
  - If there is ***one*** value in the list, ***both*** the head and the tail refer to the ***same*** node.

List Complexity

- The ***expected*** time complexity for all basic operations on an ***array list*** is ***constant time*** (***O(C)***).
    - The notable exception is ***append***, which has a worst case time complexity of ***O(n)*** because of the copy operation, but this should be ***rare***.
- The performance of an array list suffers whenever the elements are ***shifted*** in the array.
    - The shift is a ***linear time*** (***O(n)***) operation and is necessary whenever an element is ***inserted into*** or ***removed from*** the middle of the array (we did not implement these operations).
- The ***expected*** time complexity for a ***linked list*** is very different!
    - The performance of ***append*** method will ***always*** be ***constant time***.
    - The ***get***, ***set***, ***insert***, and ***remove*** operations will also run in ***constant time*** if performed at the ***head*** or ***tail*** of the list.
- The performance of a linked list suffers whenever the list has to be ***searched***.
    - This is a ***linear time*** (***O(n)***) operation and is necessary whenever ***get***, ***set***, ***insert*** or ***delete*** is performed at any index in the middle of the list.



That depends, if you...	- Always Append at the End - Never Insert or remove in the middle - Need Random Access - Know the maximum number of elements	- Always Insert or Remove at head and/or tail - Don't need random access - Don't know the maximum number of elements
...then choose:	Array List	Linked List

Understanding which operations are needed ***the most frequently*** will help you choose the most efficient implementation to solve your problem.

Java's `for`-each Loop

- So far, most (if not all) of the `for` ***loops*** we have written have used Java's ***"classic"*** or ***"C-Style"*** `for` loop syntax.
    - ***Initialize*** a counting variable, evaluate a ***boolean expression***, ***modify*** the variable.
    - `for(int i=0; i<array.length; i++) {}`
- Java also includes a syntax that is much closer to Python's `for` loop called a `for`-***each*** loop.
    - Given some ***sequence*** like a ***list*** or an ***array***, the `for`-each loop will iterate over the elements in the sequence.
- Just like Python's version, a ***loop variable*** is assigned to the next element in the sequence ***before*** the start of each iteration.
    - Of course, because it's Java the ***type*** of the loop variable has to be declared.

The Python `for` ***loop*** assigns the next element in the specified ***sequence*** to the ***loop variable*** at the start of each ***iteration***.

```
for element in an_array:
    print(element)
```

The loop automatically terminates after the last element.

The Java `for`-***each loop*** works exactly the same way other than some small syntactic differences. The ***type*** of the loop variable must be specified...

```
for(int element : an_array) {
   System.out.println(element);
}
```

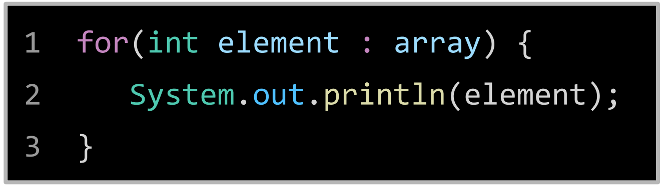
...and a ***colon*** (`:`) is used instead of the keyword "`in`".

5.19

Using for-each to Iterate Over an Array

Java's `for`-each loop is a convenient way of iterating through the elements in a data structure without needing to understand how the elements are stored or organized. It works naturally with Java arrays, iterating through the elements in order by index. Try using a `for`-each loop now to iterate over an array of strings.

The `for-each` ***loop*** in Java works exactly like the for loop in Python, with some slight syntactic differences.



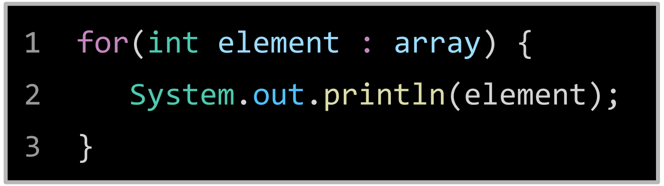
```
for(int element : array) {
    System.out.println(element);
}
```

- Create a new Java class named "`ForEach`" and define a `static` method named "`forArray`" that declares a parameter for a `String` ***array***.
  - Use a `for-each` ***loop*** to iterate over the elements in the array and print each on a separate line.
  - ***Do not*** use a "classic" `for` loop.
- Define a `main` method with the appropriate signature.
  - ***Call*** your method with a `String` array containing several values of your choice.
- ***Run*** your new class.

5.20 Using `for`-each with an `ArrayList`

The `for`-each loop in Java should work with any data structure. For example, we *should* be able to use a `for`-each loop to iterate through the elements of an `ArrayList`. Let's try it now and see what happens.

The `for-each` ***loop*** in Java works exactly like the for loop in Python, with some slight syntactic differences.



```
for(int element : array) {
    System.out.println(element);
}
```

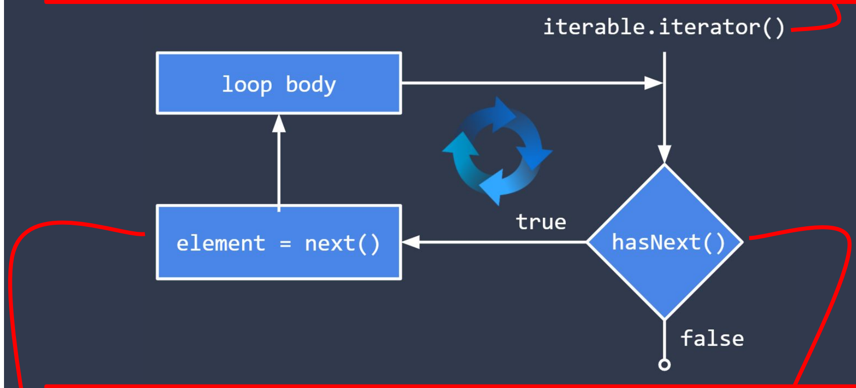
- Open your `ForEach` class and define a `static` method named "`forList`" that declares a parameter for a `List` of ***strings***.
    - Use a `for-each` ***loop*** to iterate over the elements in the list and print each on a separate line.
    - ***Do not*** use a "classic" `for` loop.
- ***Call*** `forList` from `main` with an `ArrayList` containing several `String` values of your choice.
- ***Run*** your class.
    - What happens?

Iterable & Iterator

- In Python, the `for` loop ***does not*** automatically work with any kind of data structure.
  - The special `__get__` and/or `__iter__` methods must to be implemented in the data structure class.
- The same is true in Java - a data structures ***will not automatically work*** with a `for-each` loop.
- The `java.lang.Iterable`interface defines the `iterator()` method that returns an instance of the `java.util.Iterator`interface.
- An `Iterator` returns ***each element*** in a data structure in some ***sequential order***. It defines basic methods that the `for-each` loop uses to ***control*** iteration.
  - `hasNext()` - returns true if there is at least one element in the sequence that the Iterator has not yet returned.
  - `next()` - returns the next element in the sequence, e.g. `E e = iterator.next()`
- An `Iterator` is guaranteed to return each element in the sequence ***exactly once***.
- In Java, a data structure ***must implement*** the `Iterable` interface and ***return*** a working `Iterator` in order to work with a `for-each` loop.

```
for(E element : iterable) {
    // loop body
}
```

Given some `iterable` object, the `for-each` loop first calls its `iterator()` method to get its `Iterator`...

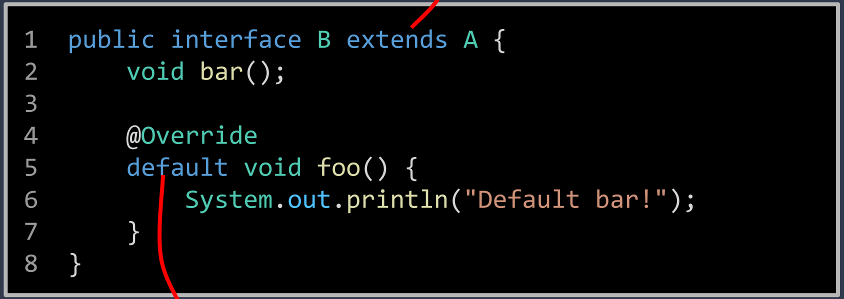


At the start of each iteration, it calls `hasNext()` on the `Iterator` to determine if there are any more elements.

If `hasNext()` returns `true`, the `next()` method is used to assign the next element to the ***loop variable***.

Extending Interfaces

One interface may ***extend*** another to inherit its methods and establish a ***inheritance relationship***, i.e. `B` will work with code written for `A`.



```
public interface B extends A {
    void bar();

    @Override
    default void foo() {
        System.out.println("Default bar!");
    }
}
```

A Java interface may use the `default` ***modifier*** on a method to provide a default implementation of that method.

A class that implements the interface ***may*** or ***may not*** provide an implementation for any default methods. If no implementation is provided by the class, the default version in the interface is used.

- In Java, one interface may ***extend*** another interface.
    - e.g. `public interface B extends A {}`
    - In this example, the interface `B` will ***inherit*** all of the abstract methods defined in interface `A`.
    - Note that ***both*** `A` and `B` must be interfaces; an interface ***cannot*** extend a class.
- Under normal circumstances, any class that implements an interface must provide a concrete implementation of ***all*** of the abstract methods defined by the interface.
    - This includes any methods that are inherited from some other interface!
- We'd like to modify our `List` interface so that it ***extends*** `Iterable`, but this introduces a ***problem***: any class that currently implements `List` ***will no longer compile***!
    - ***Neither*** `ArrayList` ***nor*** `LinkedList` currently provides an implementation of the `iterator()` method, which would be required if `List` extends `Iterable`.
- Java provides a mechanism to circumvent this problem: `default` ***interface methods***.
    - An interface may provide a `default` ***implementation*** of one or more of its methods.
    - If a class implements the interface but ***does not*** provide an implementation of a default method, the default implementation is used by...well, ***default***.

A Closer Look at Default Methods

In Java, one interface may ***extend*** another interface to ***inherit*** the methods defined in the other interface.

The child interface ***may*** add its own methods, and may also provide ***default implementations*** of any inherited methods by using the `default` ***modifier***.

An interface may also define its ***own*** default methods as well.

Default methods are typically used when adding ***new*** methods to an ***existing*** interface to avoid causing compilation errors in existing classes.

A class that implements the interface ***must*** implement any abstract methods and ***may*** provide its own implementations of any default methods.

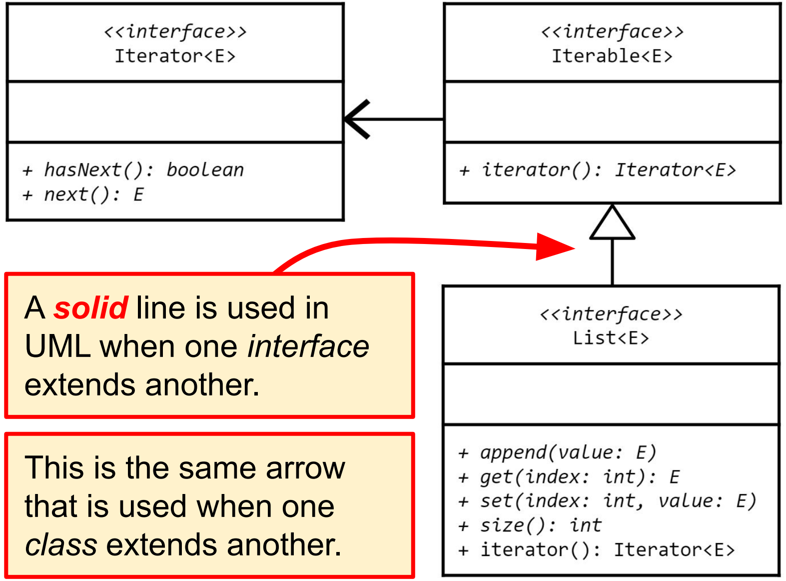
```java
public interface A {
    void foo();
}
```

```java
public interface B extends A {
    void bar();

    default void foo() {
        System.out.println("Default foo!");
    }

    default void bell() {
        System.out.println("Default bell!");
    }
}
```

```java
public class C implements B {
    @Override
    public void bar() {
        System.out.println("Concrete bar!");

    }
}
```

5.21 An Iterable List

Ideally, *any* implementation of `List` should be iterable and thus usable with a `for`-each loop. The best way to ensure this is to modify the `List` interface to ***extend*** the `Iterable` interface. The existing implementation(s) of `List` (e.g. `ArrayList`) will break if we don't add a ***default*** implementation of the required method!



A ***solid*** line is used in UML when one *interface* extends another.

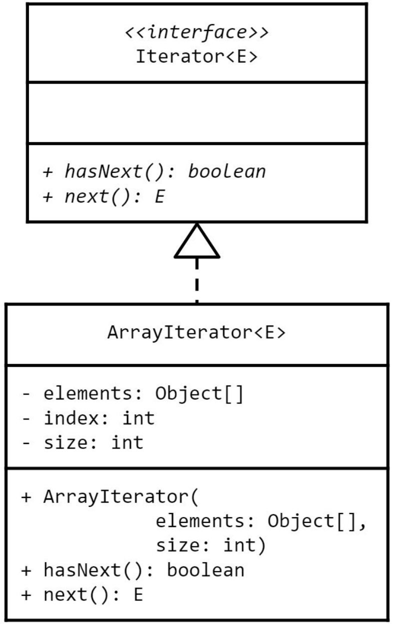
This is the same arrow that is used when one *class* extends another.

- Open the `List` interface and modify it so that it ***extends*** `Iterable`.
  - You will need to ***import*** `java.util.Iterator`.
  - Add a ***default implementation*** of the `iterator()` method that throws an `UnsupportedOperationException`
- Your `ForEach` class should no longer have any errors! That's a good thing!
  - But what happens when you try to run it now?

5.22 An Array Iterator

PROBLEMS 37

Trying to use the `ArrayList` with a `for`-each loop will throw an exception (because the default implementation in the `List` interface is being used). The first step towards fixing that problem is to implement an `Iterator` that can iterate through the elements in an array.



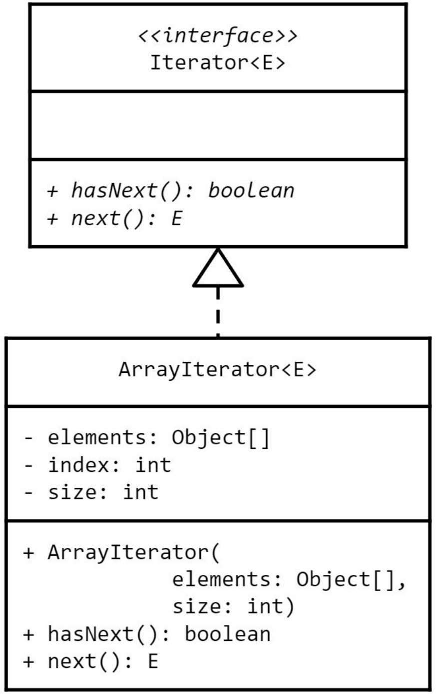
```
<<interface>>
Iterator<E>

+ hasNext(): boolean
+ next(): E
```

```
ArrayIterator<E>

- elements: Object[]
- index: int
- size: int

+ ArrayIterator(
          elements: Object[],
          size: int)
+ hasNext(): boolean
+ next(): E
```

- Create a new class named "`ArrayIterator`".
  - Use the UML class diagram to the left as a guide to implement your iterator.
  - Use `index` to keep track of which element should be returned by the `next()` method.
  - ***Note*** that the array may only be ***partially full***! Use `size` to determine when the `index` has passed the last element.
  - Print a message to standard output in each method including that the name of the method and the value being returned.
- Rename the provided `ArrayIteratorTest.txt` to `.java` and use it to test your `ArrayIterator`.
  - If you need help getting it to work, please raise your hand!

5.23 An Iterable `ArrayList`

Now that we have an `ArrayIterator`, we can implement the `iterator()` method on `ArrayList` so that it no longer uses the default implementation to throw an exception!



<<interface>>
Iterator<E>

+ hasNext(): boolean
+ next(): E

ArrayIterator<E>

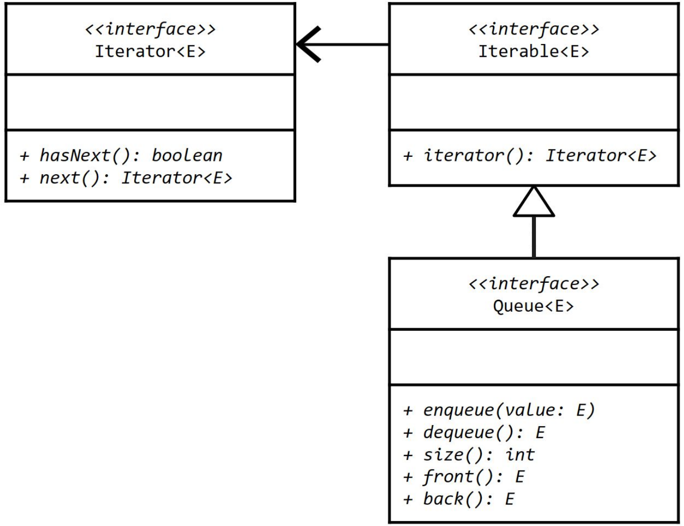
- elements: Object[]
- index: int
- size: int

+ ArrayIterator(
    elements: Object[],
    size: int)
+ hasNext(): boolean
+ next(): E

- Open your `ArrayList` class and ***override*** the `iterator()` method inherited from `Iterable` and `List`.
    - Print a message to standard output indicating that the method has been called.
    - ***Return*** a new `ArrayIterator` using the list's array of elements and its current size.
- Try running the `ForEach` class again.
    - What happens now?
    - Do you see how the `next()` and `hasNext()` methods are used to control the `for`-each loop?

5.24 An Iterable Queue

Technically speaking, a `Queue` only provides access to the elements at the front and back (and not to the elements somewhere in the middle). Still, it can be a useful exercise to practice making another data structure iterable so that it will work with a `for`-each loop. Let's try it now!



<<interface>>
Iterator<E>

+ hasNext(): boolean
+ next(): Iterator<E>

<<interface>>
Iterable<E>

+ iterator(): Iterator<E>

<<interface>>
Queue<E>

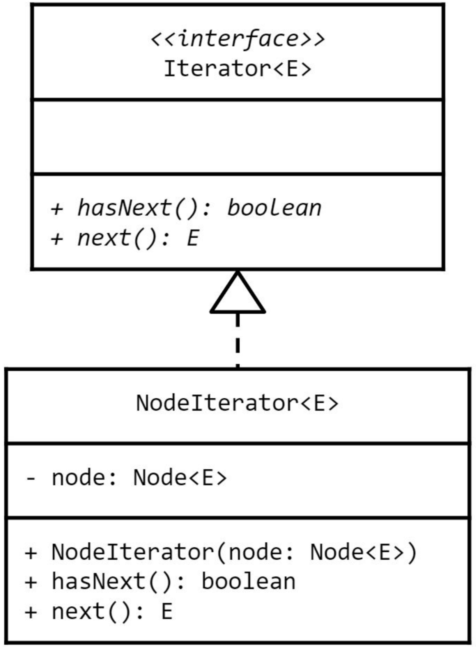
+ enqueue(value: E)
+ dequeue(): E
+ size(): int
+ front(): E
+ back(): E

- Open the `Queue` interface and modify it so that it extends Iterable.
- Write a default implementation of the `iterator` method that throws an `UnsupportedOperationException`
    - You will once again need to import the java.util.Iterator interface so that you can use it as a return type.
- Open the `ForEach` class and define a method named "`forQueue`" that declares a parameter for a `Queue`.
    - Uses a `for`-each loop to iterate through the elements and print them to standard output.
- In main, create a `NodeQueue` and enqueue a few elements before passing it into the `forQueue` method.
    - What happens?

5.25 An Iterator of Nodes

PROBLEMS 37

The ArrayIterator will not work with a NodeQueue because it stores its elements using nodes rather than an array. We'll need to implement a new Iterator that can iterate through a series of linked nodes.



<<interface>>
Iterator<E>

+ hasNext(): boolean
+ next(): E

NodeIterator<E>

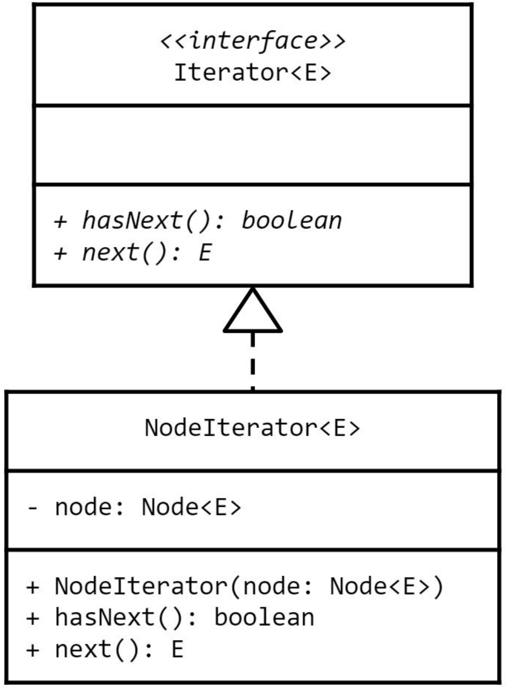
- node: Node<E>

+ NodeIterator(node: Node<E>)
+ hasNext(): boolean
+ next(): E

- Create a new class named "`NodeIterator`".
  - Use the UML class diagram to the left as a guide to implement your iterator.
  - The `hasNext()` method should return `true` as long as the iterator has not reached the end of the sequence (`node` is `null`).
  - The `next()` method should return the ***value*** in the `node` ***and*** move to the ***next node*** in the linked-sequence.
  - Print a message to standard output in each method including that the name of the method and the value being returned.
- Rename the provided `NodeIteratorTest.txt` to `.java` file and use it test your `NodeIterator`.
  - If your implementation is not working, raise your hand and ask for help!

5.26 An `Iterable` `NodeQueue`

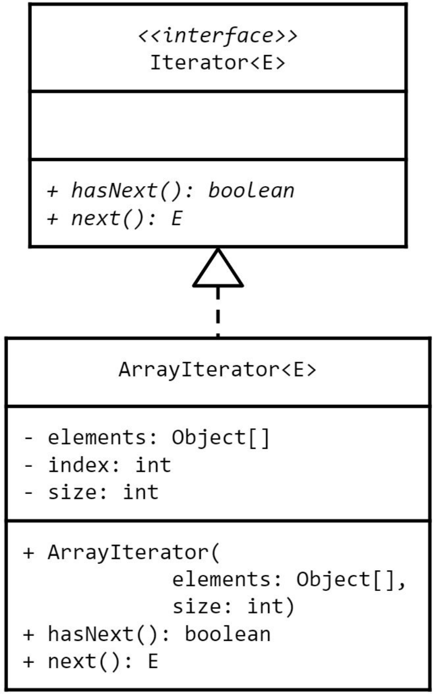
Now that we've got a working `NodeIterator`, we can update the `NodeQueue` so that it will work with a `for`-each loop without throwing an exception. Let's do that now.



- Open your `NodeQueue` class and ***override*** the `iterator()` method inherited from `Iterable` and `List`.
    - ***Return*** a `NodeIterator` that starts at the `head` of the list.
- ***Run*** your `ForEach` class.
    - What happens now?
    - Do you see how the `next()` and `hasNext()` methods are used to control the `for`-each loop?

5.27 An Iterable `ArrayQueue`

One `Queue` down, and one to go! Let's update the `ArrayQueue` so that it can also be used with a for-each loop. This may not be as simple as it seems…

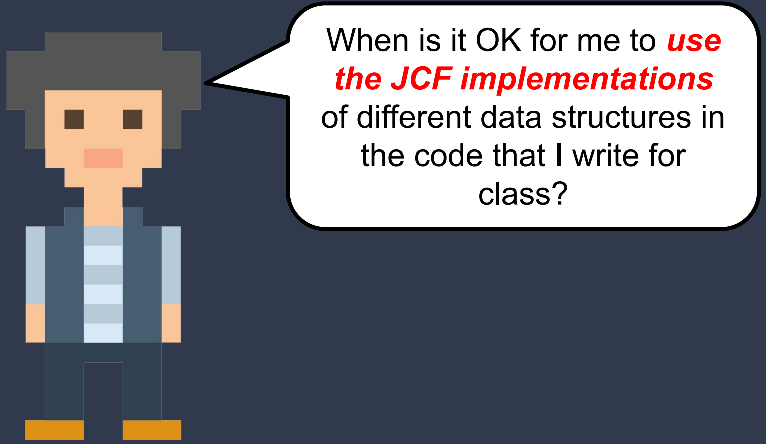


```
<<interface>>
Iterator<E>

+ hasNext(): boolean
+ next(): E
```

```
ArrayIterator<E>

- elements: Object[]
- index: int
- size: int

+ ArrayIterator(
          elements: Object[],
          size: int)
+ hasNext(): boolean
+ next(): E
```

- Open the `ArrayQueue` class and override the `iterator` method inherited from the `Queue` interface.
    - Ideally, you would be able to use it to return an `ArrayIterator`, but the current implementation of that class only iterates from index 0 to `size-1`. Will that work with an `ArrayQueue`?
    - What changes do you think you will need to make to the class?
    - Will it still work with the `ArrayList`?!
- Use the `main` method in the `ForEach` class to create an `ArrayQueue`, enqueue a few values, and pass it into the `forQueue` method.
    - What happens if you dequeue some of the values before calling the method?
    - What if the array is resized?
    - What if the front or back wrap around to the beginning?

The Java Collections Framework (JCF)



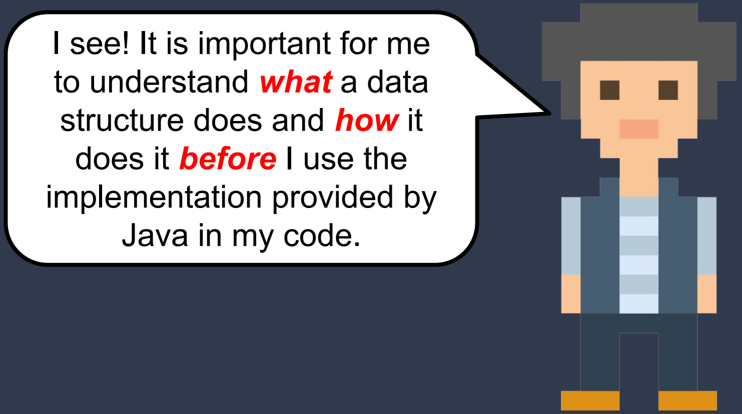
The ***Java Collections Framework*** (***JCF***) provides very flexible and powerful implementations of many different abstract data types.

The general policy for SoftDev II is that, ***once we have explored an abstract data type in class***, you are free to use the corresponding implementation in the JCF.

- A ***data structure*** is a grouping of related elements.
    - Data structures are also sometimes referred to as ***collections***.
- A ***collections framework*** is a unified architecture for representing and manipulating collections.
- All collections frameworks contain three things:
    - ***Interfaces*** defining abstract data types (ADTs).
    - ***Implementations*** of the interfaces.
    - Methods that provide implementations of ***useful algorithms*** for things like searching and sorting.
- Java's version is the ***Java Collections Framework*** (***JCF***).
- Implementing your own collections as we have done is an extremely valuable learning experience with many benefits.
    - Gain practice with ***interfaces***, ***inheritance***, and ***polymorphism***.
    - Gain a ***deeper understanding*** of how data structures work internally.
    - Learn about ***design***, ***design patterns*** and ***algorithms***.
- Practically speaking, however, the implementations provided by a collections framework like the JCF are more ***robust***, ***reliable***, ***stable***, and ***feature rich*** than those that we will build ourselves.

JCF Implementations

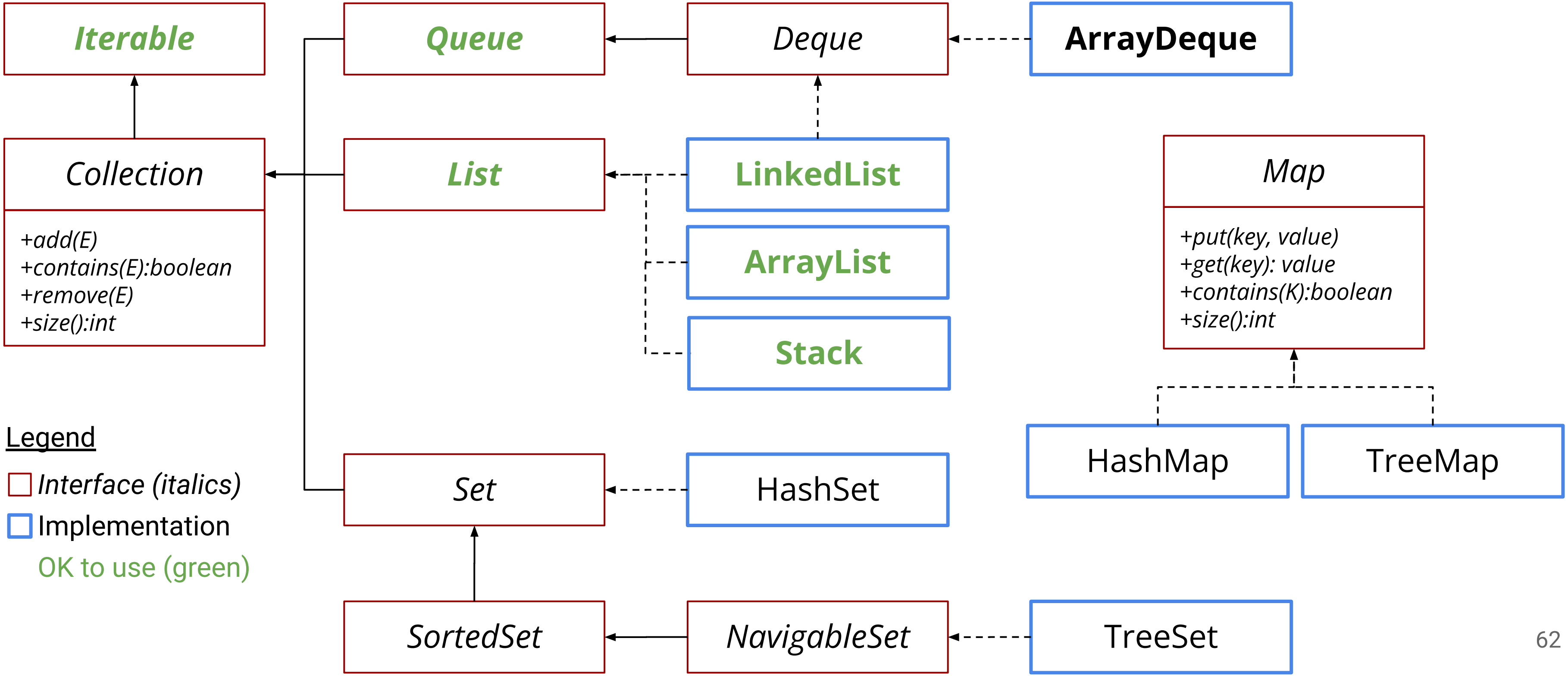
- Of course the JCF includes many interfaces and implementations of the data structures that we have looked at so far.
    - Virtually every interface and implementation in the JCF is ***generic*** and so can be used with any type.
- `java.util.List` is the ***list*** interface and defines methods for ***add***, ***get***, ***insert***, ***remove***, and ***iteration*** as well as many others.
- Not surprisingly, there are ***two*** main implementations of `List` that are commonly used.
    - `java.util.ArrayList` is ***array-based***.
    - `java.util.LinkedList` is ***node-based***.
- `java.util.Queue` is the ***queue*** interface and is, oddly enough, implemented by `LinkedList`.
    - Java's version of `Queue` is a little odd because it is ***iterable*** and provides ***random access***.
    - It also uses strange method names like ***"offer"*** and ***"poll"*** instead of "enqueue" and "dequeue."
- `java.util.Stack` is an implementation of a ***stack***.
    - It is also a little odd because it ***iterates backwards*** (from bottom to top) and provides random access.



Unless otherwise instructed, you may feel free to use ***any*** of the JCF interfaces or implementations ***here***, i.e. on your homework or practical exams.

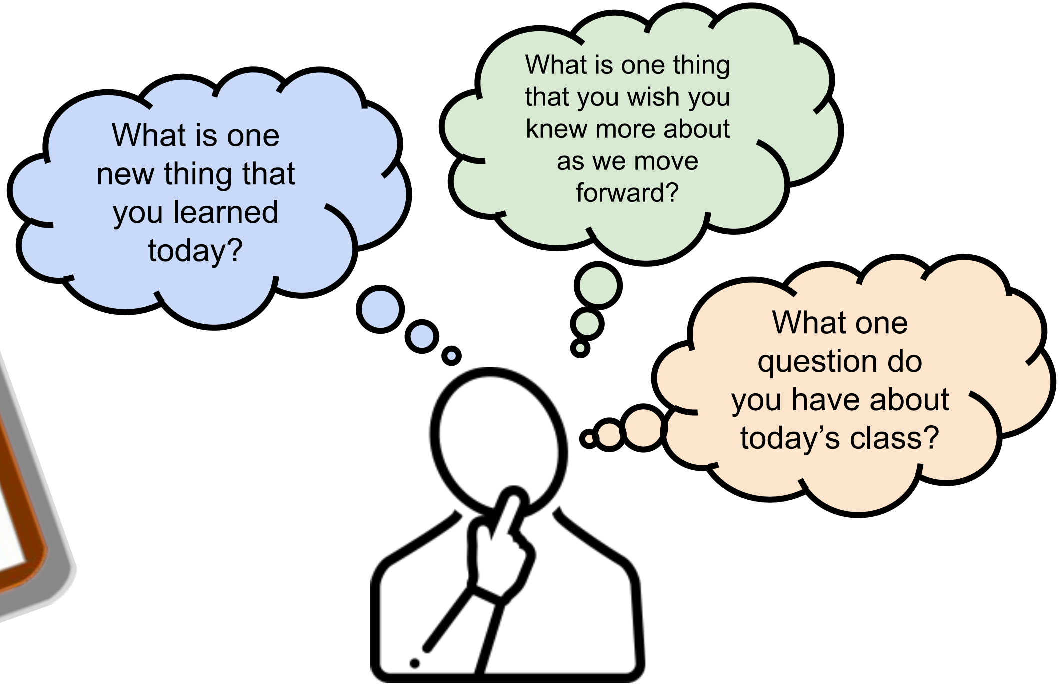
As we continue to explore additional data structures in the coming units, this list will ***gradually grow*** to include the entire JCF.

An *Incomplete* JCF Class Hierarchy



Summary & Reflection

- Abstract Data Types
  - Java Interfaces
- Nodes & Linked-Sequences
- Queues
  - Node-Based
  - Array-Based
- Lists
  - Array-based
- for-each Loops
- Iterable & Iterator
  - ArrayIterator
  - NodeIterator
- The JCF



What is one new thing that you learned today?

What is one thing that you wish you knew more about as we move forward?

What one question do you have about today's class?

Please answer the questions above in your notes for today.