COMP2402
Abstract Data Types and Algorithms

# Course Introduction

## Your Instructor

# Robert Collier


**Email Address**

robert.collier@scs.carleton.ca


**Office Hours**

Mondays

10:00 – 12:00

Herzberg Laboratories, Room 5326

**Research Interests**

Computer Science Education, Discrete Optimization

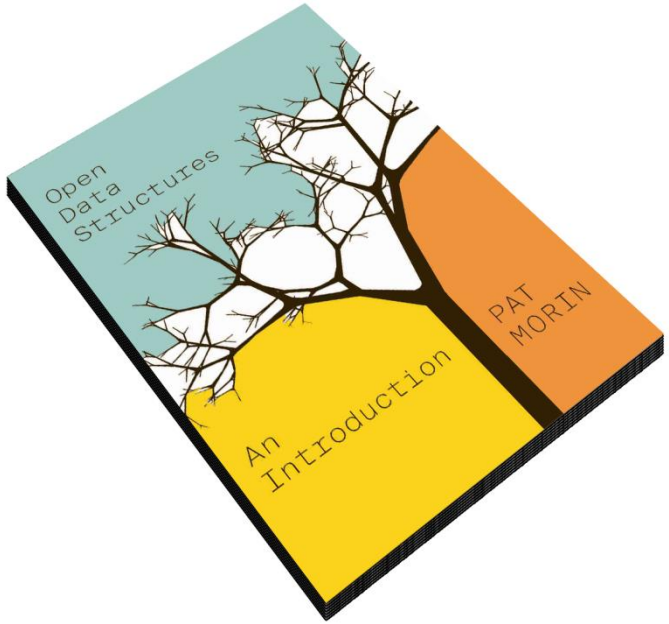
Evolutionary Computation, Artificial Intelligence

# Required Textbook

## Open Data Structures in Java

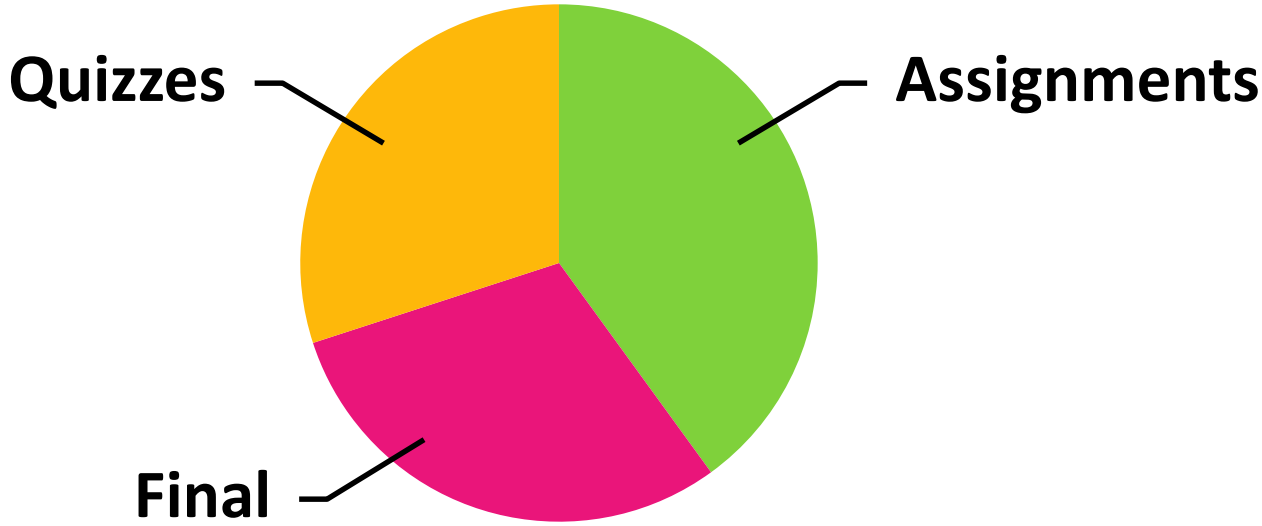
**by Pat Morin**

**http://opendatastructures.org/**



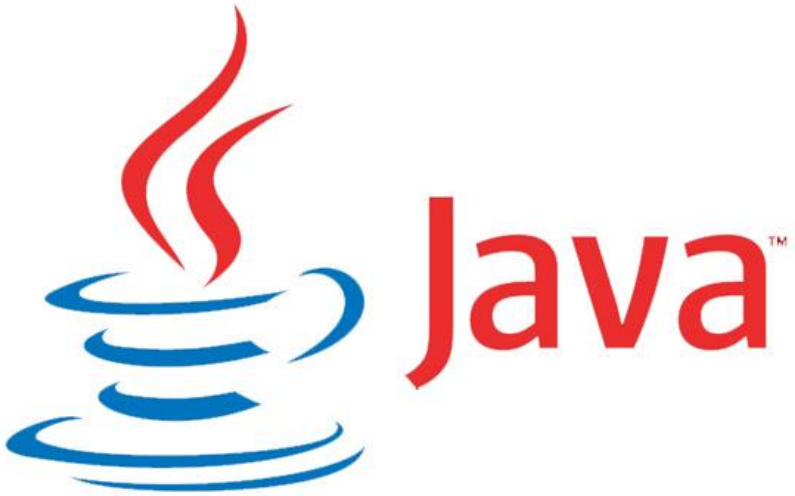
# Assessment Details

- **Quizzes** 30 %
  - In-Class
  - Two (2) × 15.0%
- **Assignments** 40 %
  - Four (4) × 10.0%
- **Final** 30 %
  - Registrar Scheduled
  - Date / Location TBA



# Abstract Data Types / Algorithms

the Programming Language of Instruction is Java




it is an Object-Oriented Language

*C++ is also an Object-Oriented Language, and would also be a Suitable option, but it Lacks the Platform Independence and Garbage Collection of Java*

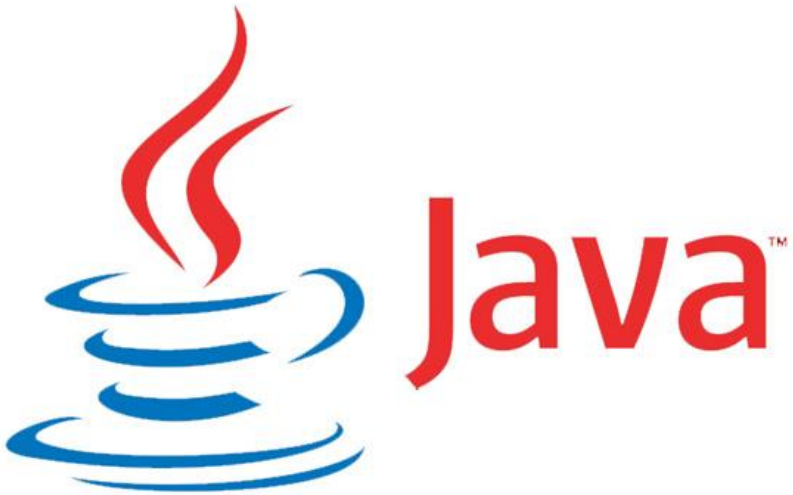
COMP2402
Abstract Data Types and Algorithms

# Data Structures and Abstract Data Types

# Reading Assignment

## The Java™ Tutorials
## "Generics"

**docs.oracle.com/javase/tutorial/java/generics/**



# Operational Definition of a Data Structure

A Data Structure...

# Operational Definition of a Data Structure

A Data Structure...

...is a Systematic Approach...

...for Storing and Accessing Data...

# Simple Example

**Suppose you want to Store (and also Manipulate) a Collection of Numbers**

**What would you Need the Storage Structure to Do?**

***sorting the values by some key would be nice, but ignore that for now...***

# Simple Example

**Suppose you want to Store (and also Manipulate) a Collection of Numbers**

**What would you Need the Storage Structure to Do?**

*sorting the values by some key would be nice, but ignore that for now...*

**Store Each Number**

**Remove Numbers from Storage**

**Insert Numbers into Storage**

**Can an Array Perform these Operations?**

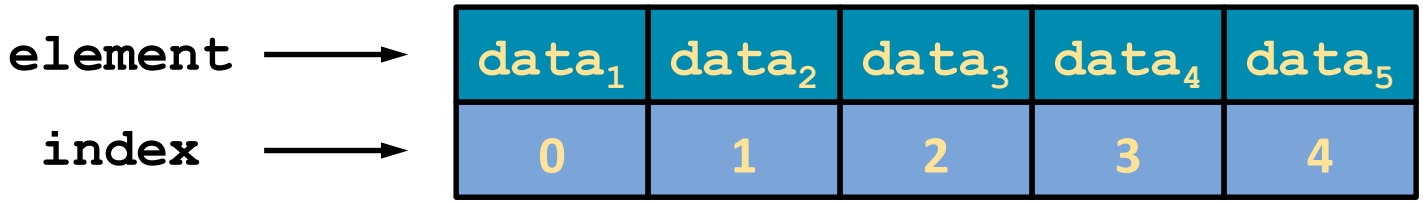
**The Answers are Yes, Yes, and Conditionally Yes**

# Array Terminology

an Array is a Container of Values with the Same Type

each Element is Accessed by a Numerical Index

the Length (i.e., number of elements) is Fixed at Creation



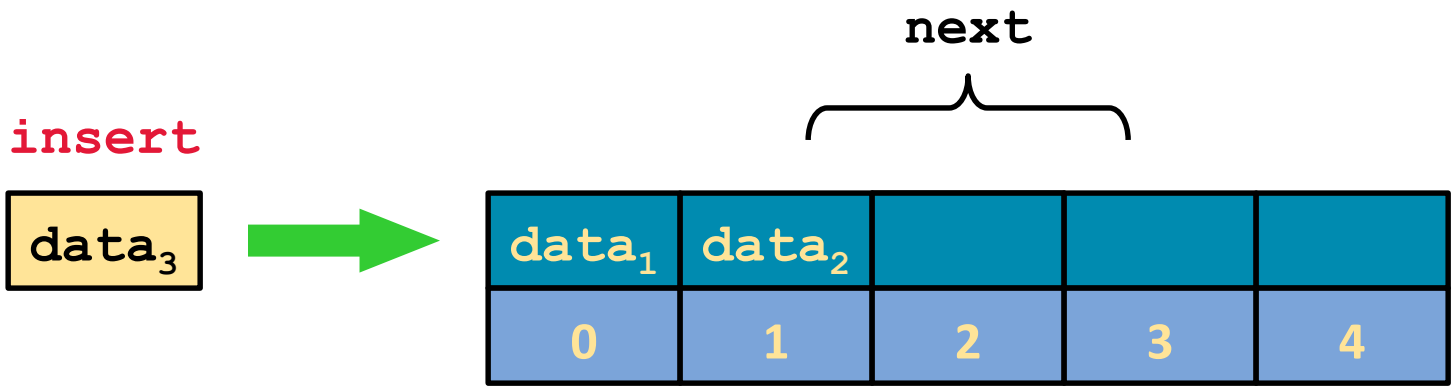
# Array Fundamentals

```
int   length = 5;  // fixed length
int   next   = 0;
int[] array = new int[length];

// insert element at the end of the array
public void insert(int data) {
    array[next] = data;
    next++;

    // any problems ?
```

*what if the array "capacity" has already been reached?*



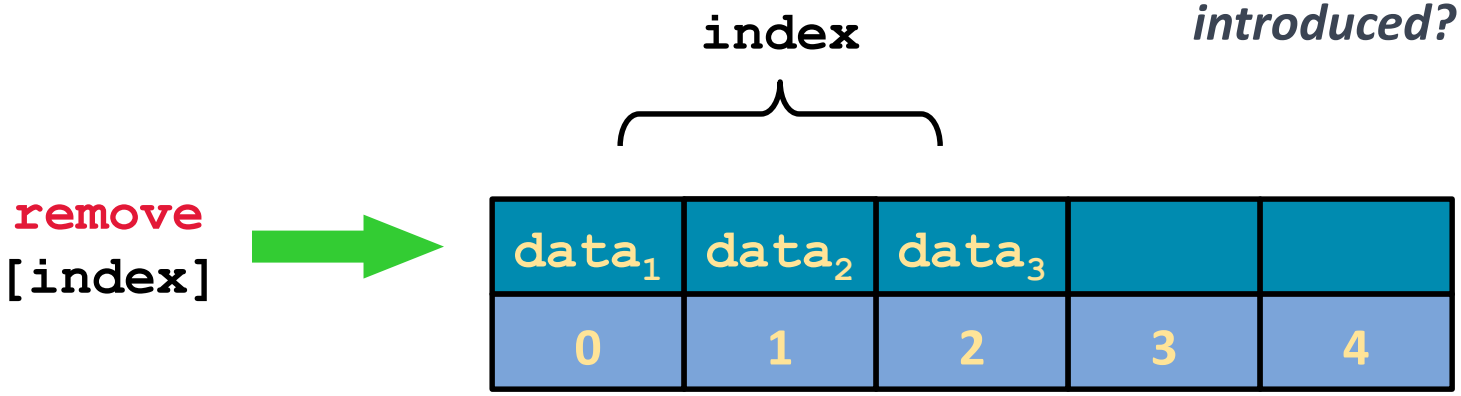
# Array Fundamentals

```
int   length = 5;  // fixed length
int   next = 0;
int[] array = new int[length];

// remove element at specified index
public void remove(int index) {
   array[index] = -1;

   // any problems ?
```

*how do you address the "gap" that has been introduced?*

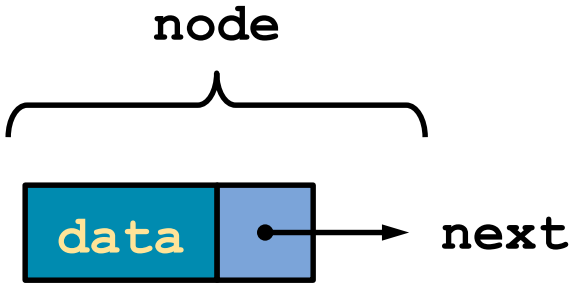


# Linked List Terminology

the Basic Unit of a Linked List is called a Node

a Node has both a Data and a Reference component

the Reference points to the Next Node in the Linked List

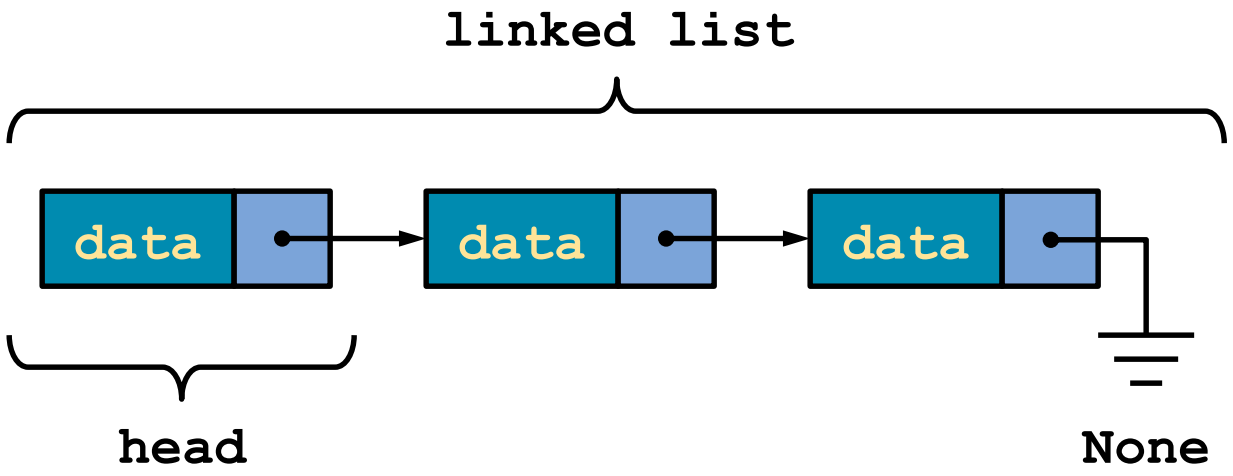


# Linked List Terminology

the Linked List is a Sequential Collection of Nodes

the First Node in the Linked List is known as the Head

the End of the List is indicated by a Reference to None



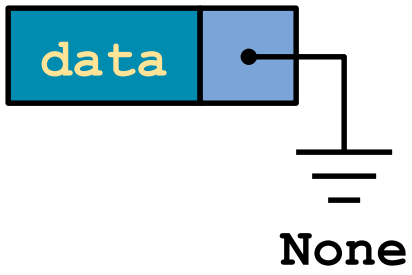
# Linked List Fundamentals

```
class Node {

    private Node next;
    private int  data;

    public Node(int data) {   // constructor
        this.data = data;
...
```

*new instances of "Node" contain whatever "data" was provided at initialization...*




*...but the reference to the "next" node in the list is initially empty*

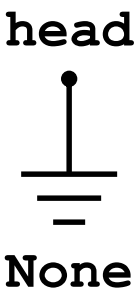
# Linked List Fundamentals

```
class LinkedList {

    private Node head;

    public LinkedList() {   // constructor
        head = null;
...
```

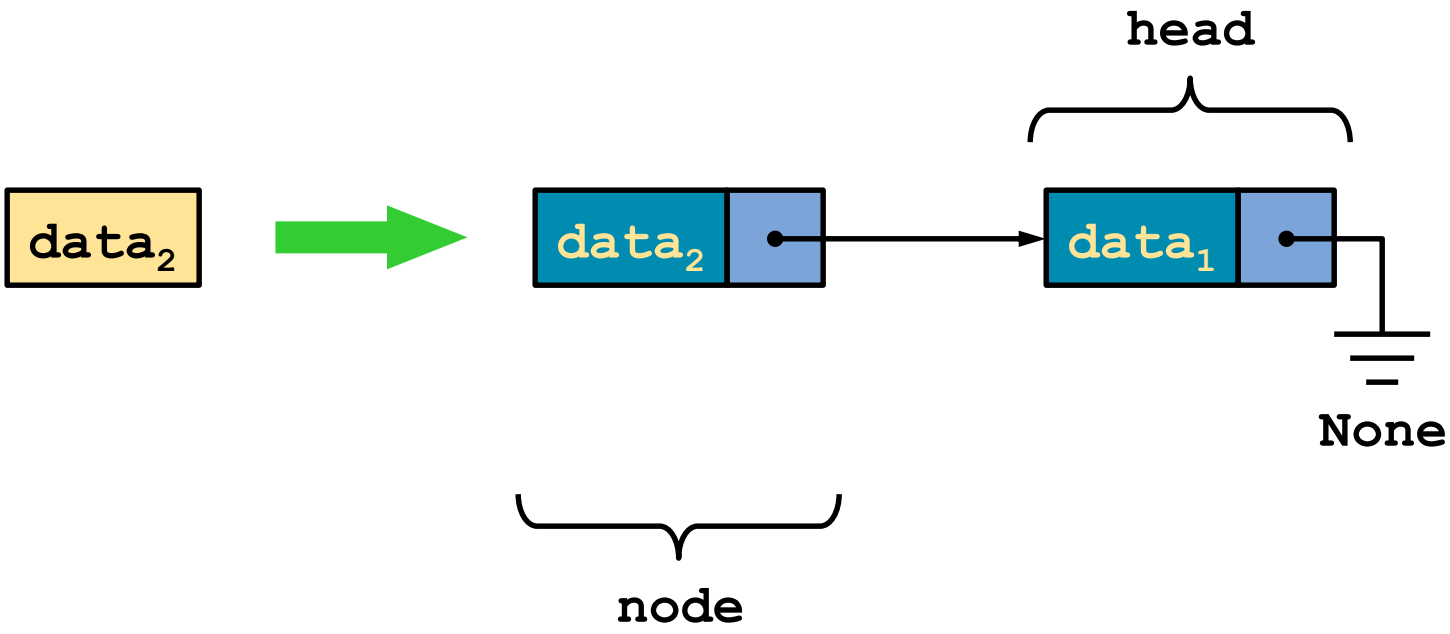
*new instances of "LinkedList" contain nothing whatsoever...*




*...and new data can only be added by inserting it as part of a new "Node"*

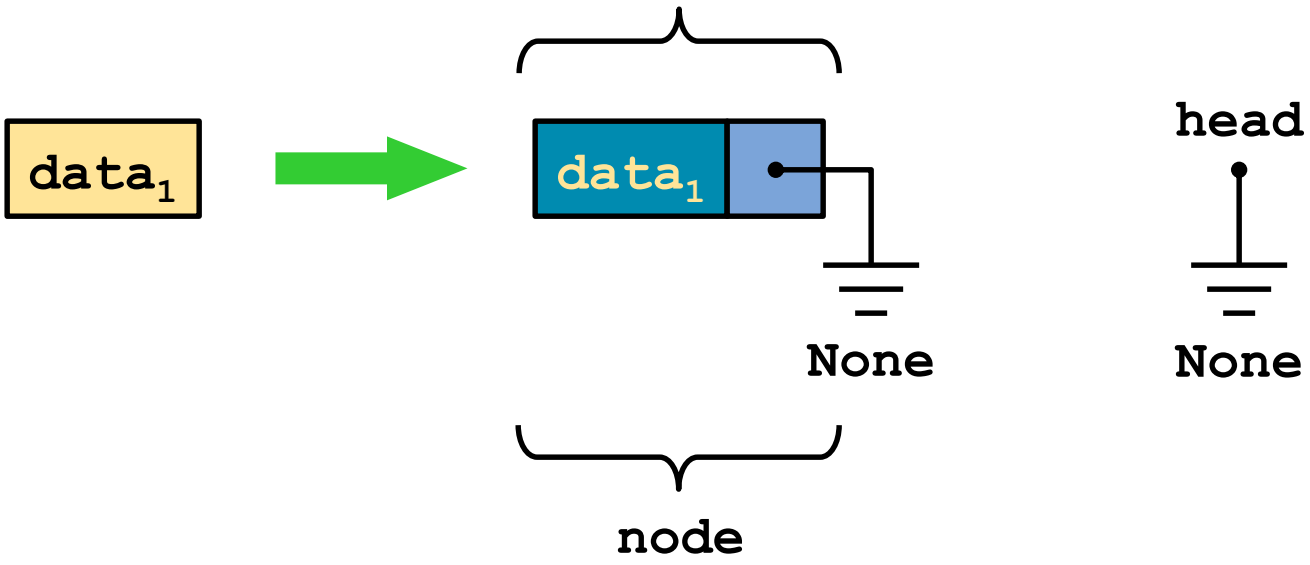
# Linked List Fundamentals

```
// insert new node at the head of the list
public void insert(int data) {
    Node data2 = new Node(data);
    data2.setNext(head);
    head = data2;
```



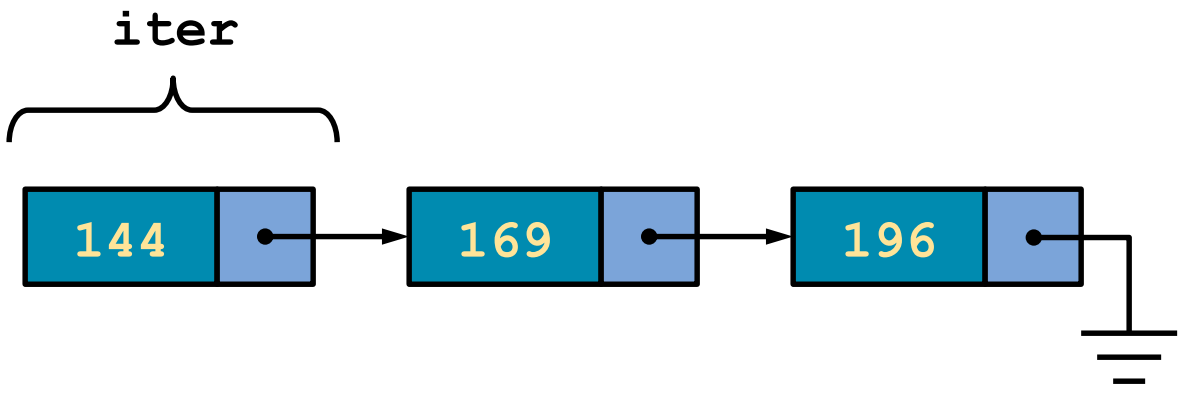
# Linked List Fundamentals

```
// insert new node at the head of the list
public void insert(int data) {
    Node data2 = new Node(data);
    data2.setNext(head);
    head = data2;
```



# Linked List Traversal

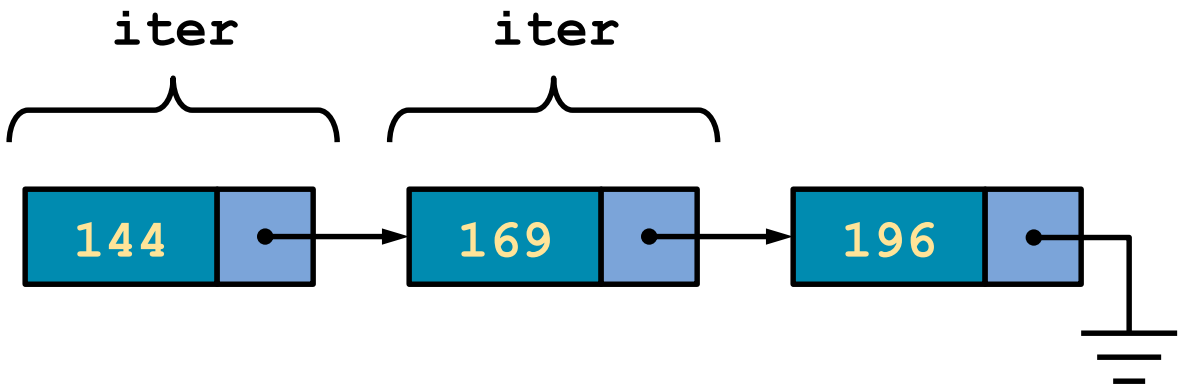
```
// traverse the list
public void traverse() {
    Node iter = head;
    while (iter != null) {
        // visit this node
        iter = iter.getNext();
    }
```




text = ""

# Linked List Traversal

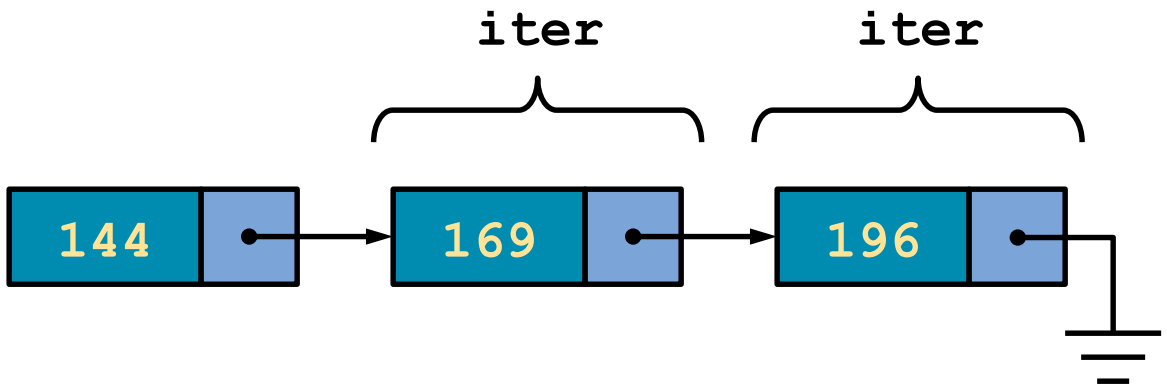
```
// traverse the list
public void traverse() {
    Node iter = head;
    while (iter != null) {
        // visit this node
        iter = iter.getNext();
    }
```




text = "[144] "

# Linked List Traversal

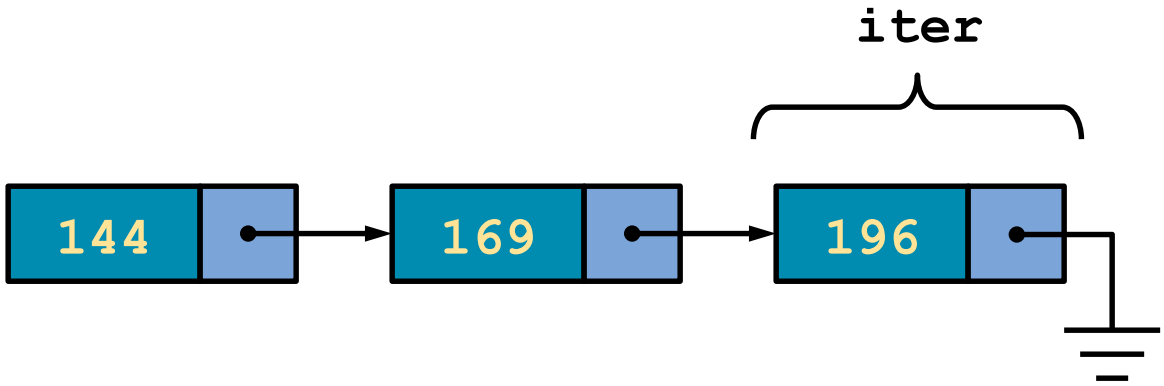
```
// traverse the list
public void traverse() {
    Node iter = head;
    while (iter != null) {
        // visit this node
        iter = iter.getNext();
    }
```




text = "[144] [169] "

# Linked List Traversal

```
// traverse the list
public void traverse() {
    Node iter = head;
    while (iter != null) {
        // visit this node
        iter = iter.getNext();
    }
```




text = "[144] [169] [196] "

# Sequential Access vs. Random Access

one of the Advantages of the Linked List over the Array

is that the Linked List is a Dynamic Data Structure

(i.e., Doesn't Have problems with Gaps or Resizing)

one of the Major Disadvantages of the Linked List

is that it is a Sequential Access Data Structure

(i.e., the References Must be Traversed in Order)

# Take Home Message 1 of 3

## On Some Tasks, Some Data Structures are More Suitable than Others.

your Application needs to Access, Insert, and Remove Data from whatever Data Structure you use for Storage

if Accessing the data is More Frequent than Inserting or Removing elements, the Array is the Better Choice

if Inserting or Removing from storage is More Frequent than Access, the Linked List is Probably Better

# Operational Definition of a Data Structure

A Data Structure...

...is a Systematic Approach...

...for Storing and Accessing Data...

# Operational Definition of a Data Structure

A Data Structure...
...is a Systematic Approach...
...for Storing and Accessing Data...
...so that it can be Used Efficiently...
for a Specific Purpose.

## Take Home Message 2 of 3

# There Must be (Better?) Alternatives to Arrays and Linked Lists.

an Application Might Use All Operations Frequently
(i.e., Accessing, Inserting, and Removing)

Neither the Array nor the Linked List can do Each Operation Efficiently so Neither might be Suitable

this Scenario is Not Contrived (e.g., a Database)
so there Must Be an Alternative

## Take Home Message 3 of 3

# What an Abstract Data Type can Do is Separate from How it is Done.

a Program could Call `get`, `set`, `insert`, or `remove` Without Knowing if it was an Array or a Linked List

this Program would Not be "Immune" to the Performance Issues (i.e., bottlenecks) noted

Nevertheless, What the Program Can Do is Independent of How the Underlying Structure Will Do It

# Interface vs. Implementation

**Interface**

***(i.e., the Abstract Data Type)***

**"What" a data structure Can Do**

**Implementation**

**"How" a data structure Will Do it**

**Good Object-Oriented Design will Separate these**

**Why?**

# Interface vs. Implementation

**Interface**

*(i.e., the Abstract Data Type)*

**"What" a data structure Can Do**

**Implementation**

**"How" a data structure Will Do it**

**Good Object-Oriented Design will Separate these**

**Why?**

**If some Code Depends Only on the Interface, you can Make Changes to the Underlying Implementation Without Changing the Dependent Code**

# Demo "Integer Linked List"

```
class Node {

      private Node next;
      private int data;

      public Node(int data) {
      ...

class LinkedList {

      private Node head;
      ...
      public void traverse() {
            Node iter = head;
            int visited;
            while (iter != null) {
                  visited = iter.get();
                  ...
```

# Demo "Naïve Generic Linked List"

```
class Node {

      private Node next;
      private Object data;

      public Node(Object data) {
      ...

class LinkedList {

      private Node head;
      ...
      public void traverse() {
            Node iter = head;
            Object visited;
            while (iter != null) {
                  visited = (Integer) iter.get();
                  ...
```

# Demo "Generic Linked List"

```
class Node<T> {

      private Node<T> next;
      private T data;

      public Node(T data) {
      ...

class LinkedList<T> {

      private Node<T> head;
      ...
      public void traverse() {
            Node<T> iter = head;
            T visited;
            while (iter != null) {
                  visited = iter.get();
                  ...
```

# Demo "Generic Linked List of Integers"

```
public static void main(String[] args) {

        LinkedList<Integer> integerList =
                new LinkedList<Integer>();

                for (int i = 0; i < 5; i++) {
                        integerList.insert(i);
                        integerList.traverse();
                }

        }

}
```

# Demo "Generic Linked List of Characters"

```
public static void main(String[] args) {

        LinkedList<Character> integerList =
                new LinkedList<Character>();

                for (int i = 65; i < 70; i++) {
                        integerList.insert((char) i);
                        integerList.traverse();
                }

        }

}
```

# Java Generics Overview

Generics Allow Types to be Parameters
in the Definitions of Classes, Interfaces, or Methods

## Advantages to using Generics

Eliminates the Need for Type Casting
...as noted in the previous example

allows the Compiler to Perform Stronger Type Checking
...to be demonstrated in a moment

when Algorithms Designed to work with Collections are Generic, they can be Reused with very little effort
...and Collections are particularly relevant to this course

# Type Parameter Restrictions

Although Generics can be Used to Parameterize Type, for certain applications it Might Be Necessary to Restrict Which Types are Permitted as Parameters

the `extends` Keyword can be used for this Restriction

```
public static <T>                                    void foo(T x){
        ...
}

public static <T extends Comparable<T>> void bar(T x){
        ...
}
```

if the Latter is Passed a Type Parameter that is Not Comparable, a Compile Time Error is thrown

# Type Erasure

**Generics Provides Compile Time Type Checking;**

**the Java Compiler uses Type Erasure to Replace Generics (and Insert Type-Casting if required)**

since this is done at compile time there is no overhead at run time

## e.g., Type Erasure changes this...

```
class Node<T extends Comparable<T>> {

    private Node<T> next;
    private T data;

    public Node(T data) {
```

# Type Erasure

**Generics Provides Compile Time Type Checking;**

the **Java Compiler** uses **Type Erasure** to
**Replace Generics** (and **Insert Type-Casting** if required)
since this is done at compile time there is no overhead at run time

## ... into this (at Compile Time).

```
class Node {

        private Node next;
        private Comparable data;

        public Node(Comparable data) {
```

# Generics and Inheritance

soon we will introduce Collections, which
Accept Type Parameters and Store Multiple Values

assuming the MyContainer class has a Method add
that Inserts a new value Into the Collection

```
MyContainer<Number> foo = new MyContainer<Number>();

foo.add(new Integer(1));
foo.add(new Double(1));
```

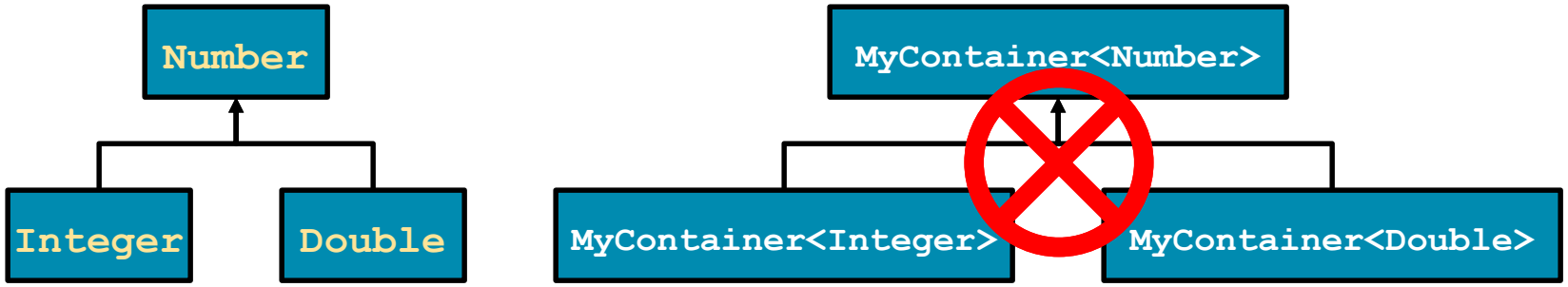
since Integer and Double are Both Subclasses of Number
Both of these Operations are Legal

# Generics and Inheritance

n.b., Generics and Inheritance can be Counterintuitive

```
public void foo(MyContainer<Number> x) {
        ...
}
```

the Argument here can be `MyContainer<Number>`,
but not `MyContainer<Integer>` or `MyContainer<Double>`*




*these are Not Subtypes of `MyContainer<Number>`

# Generics and Arrays (in Java)

## Generic Array Creation in Java is Not Permitted

```
public class Foo<E> {
    private E data[];
    public Foo(int s) {
        data = new E[s];    // not permitted
    }
}
```

If the Class is Explicitly Aware of the Type of Objects Contained (n.b., the "Class<E> c" parameter) then the newInstance Method of Array will Create the Array*

```
    public Foo(Class<E> c, int s) {
        final E[] data = (E []) Array.newInstance(c,s)
    }
```

*this is all handled by the Factory class included with the text