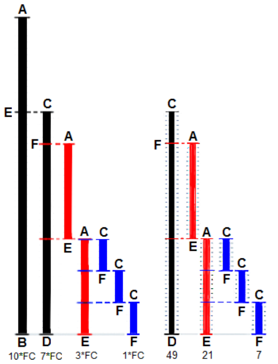
+

Class Examples
Basic Data Structures

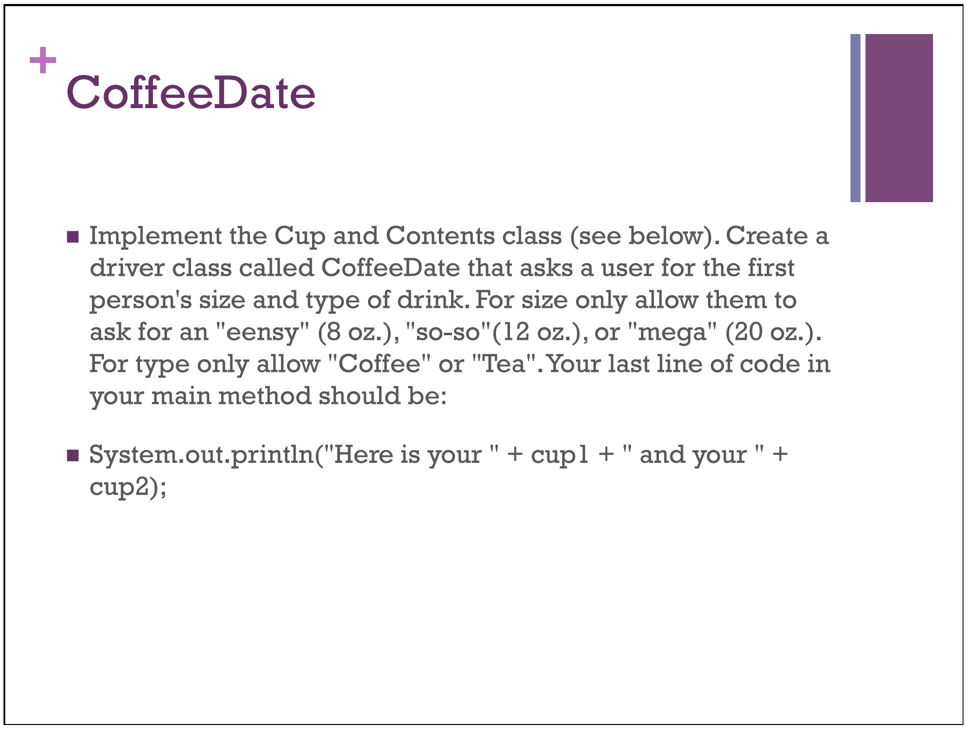
+

Fraction Class (gcd)

- In order to reduce a fraction, we first need to find the Greatest Common Divisor
- Euclid's method is a recursive way to solve this:
  - Base case: gcd(a,0) is a
  - Recursive case: gcd(a,b) is gcd(b,a mod b)
  - where $a \bmod b = a - b\left\lfloor \frac{a}{b} \right\rfloor$
- Exercise:
  - write a recursive method to do gcd
  - write an iterative method to do gcd

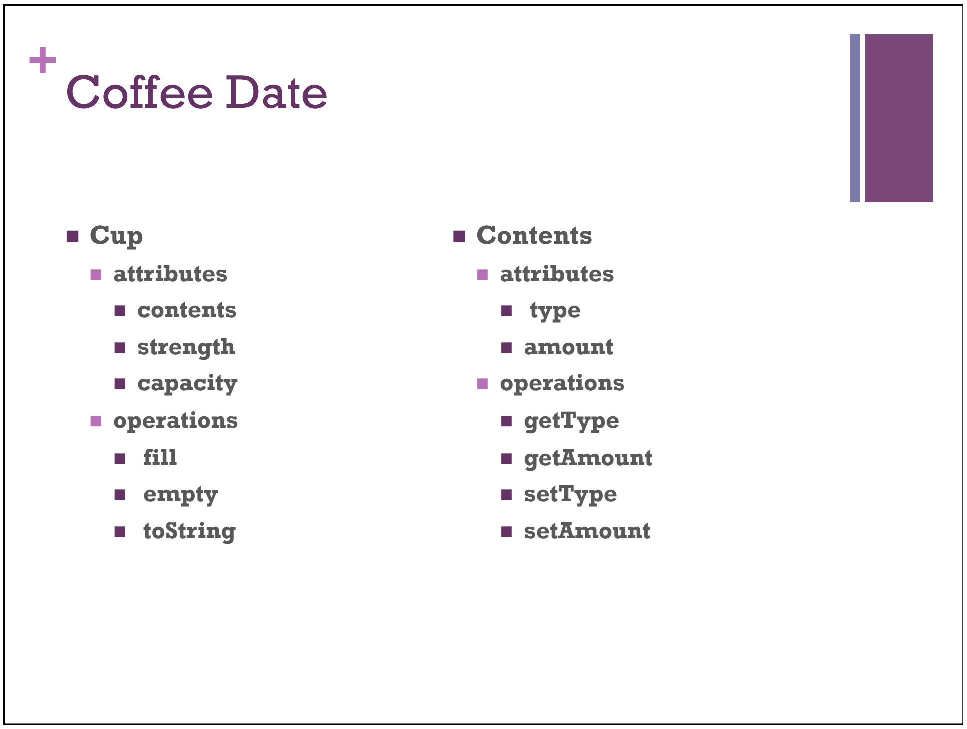


Euclid's example Nicomachus' example



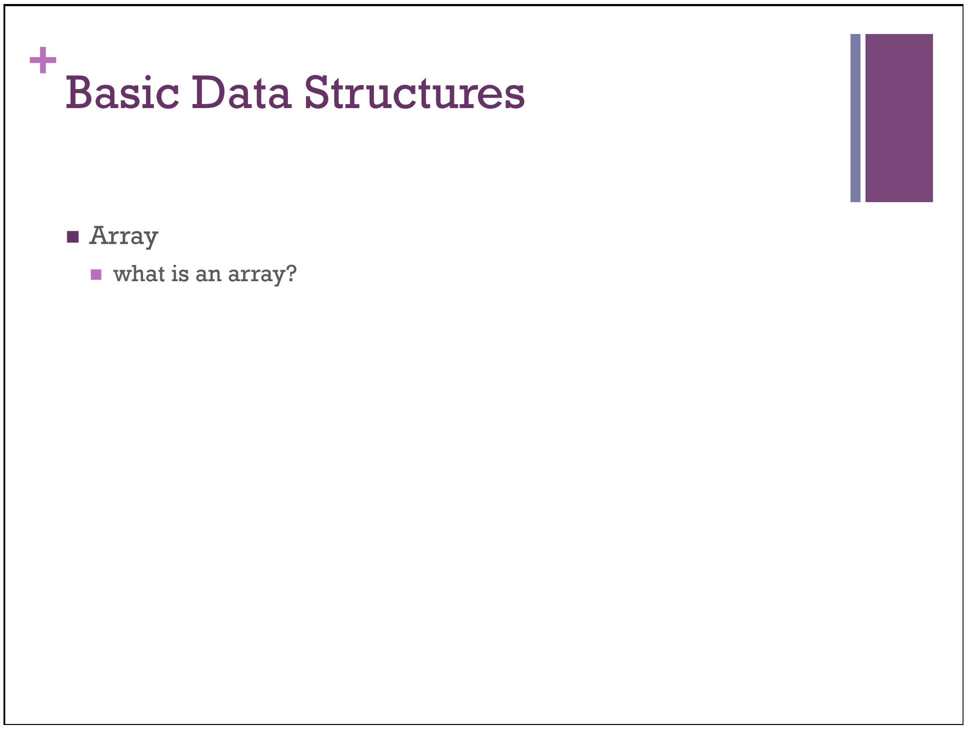
+ CoffeeDate

- Implement the Cup and Contents class (see below). Create a driver class called CoffeeDate that asks a user for the first person's size and type of drink. For size only allow them to ask for an "eensy" (8 oz.), "so-so"(12 oz.), or "mega" (20 oz.). For type only allow "Coffee" or "Tea". Your last line of code in your main method should be:
- System.out.println("Here is your " + cup1 + " and your " + cup2);



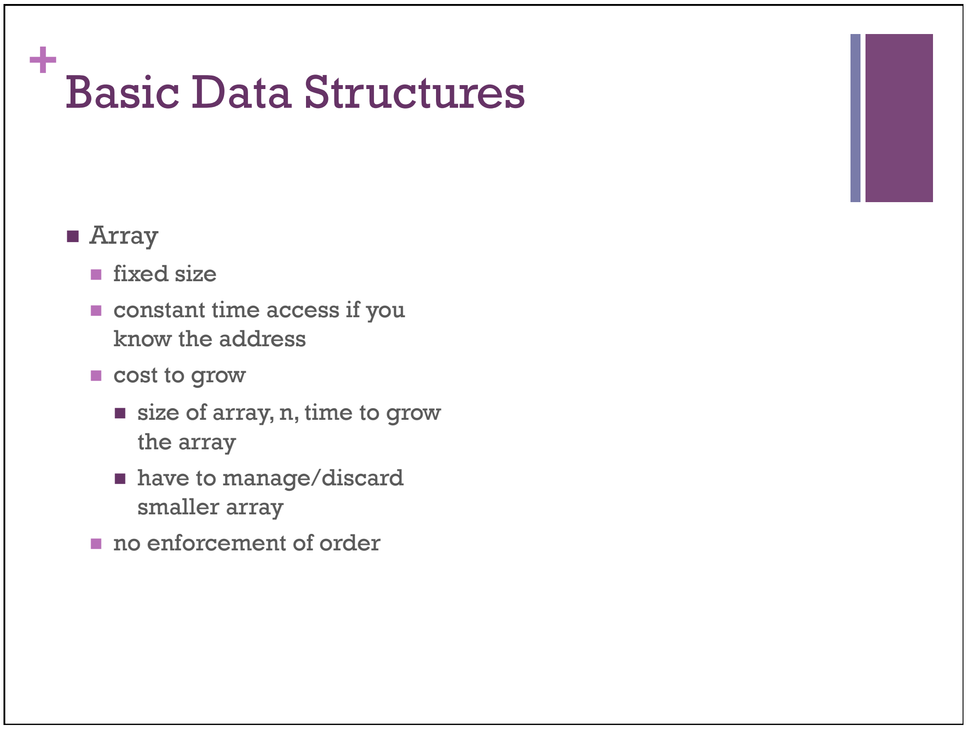
+ Coffee Date

- **Cup**
  - **attributes**
    - **contents**
    - **strength**
    - **capacity**
  - **operations**
    - **fill**
    - **empty**
    - **toString**
- **Contents**
  - **attributes**
    - **type**
    - **amount**
  - **operations**
    - **getType**
    - **getAmount**
    - **setType**
    - **setAmount**



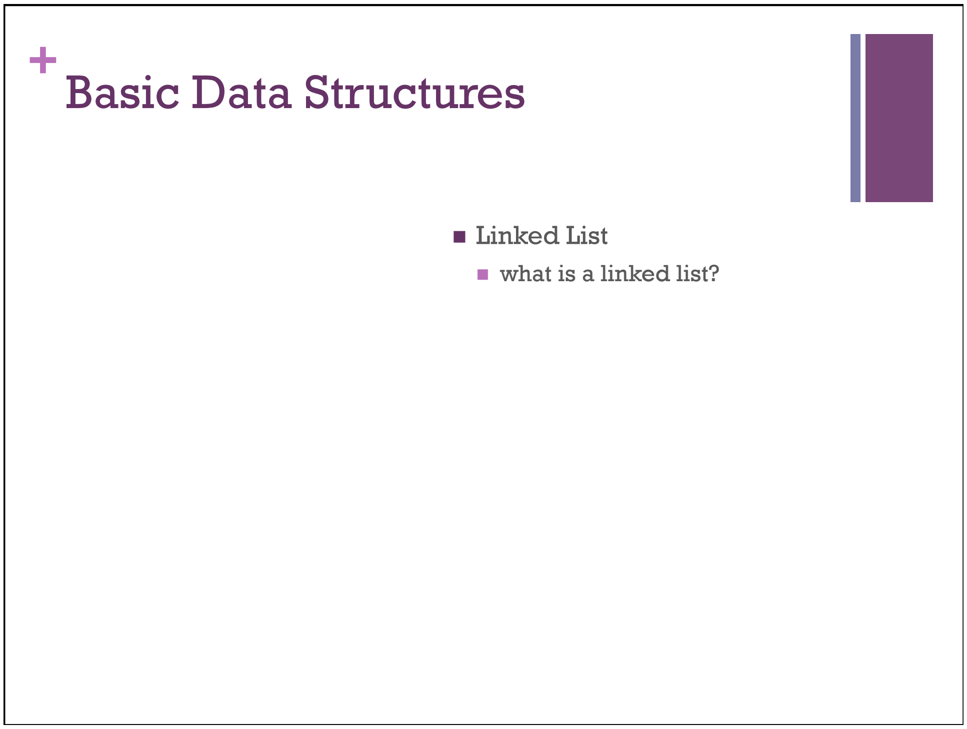
+ Basic Data Structures

- Array
  - what is an array?



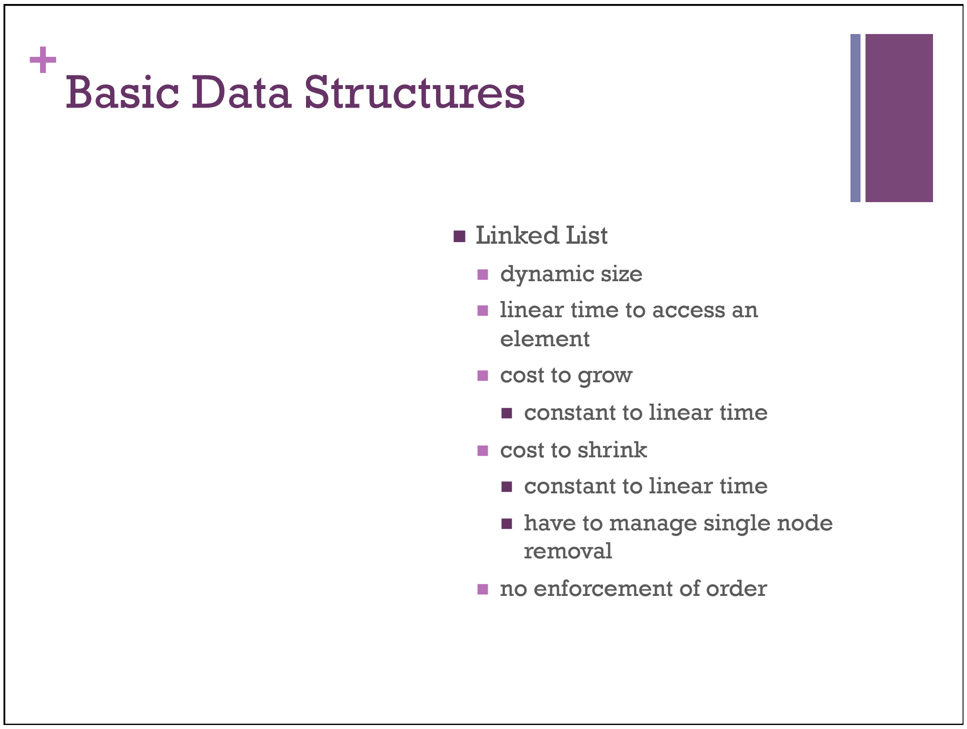
+ Basic Data Structures

- Array
  - fixed size
  - constant time access if you know the address
  - cost to grow
    - size of array, n, time to grow the array
    - have to manage/discard smaller array
  - no enforcement of order



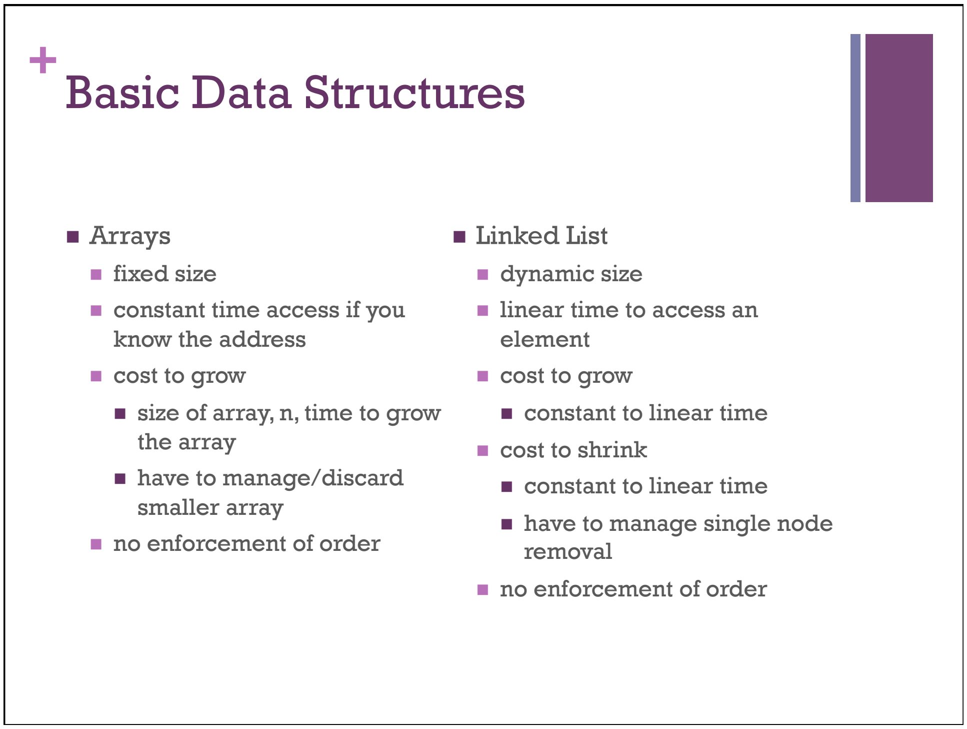
+ Basic Data Structures

- Linked List
  - what is a linked list?



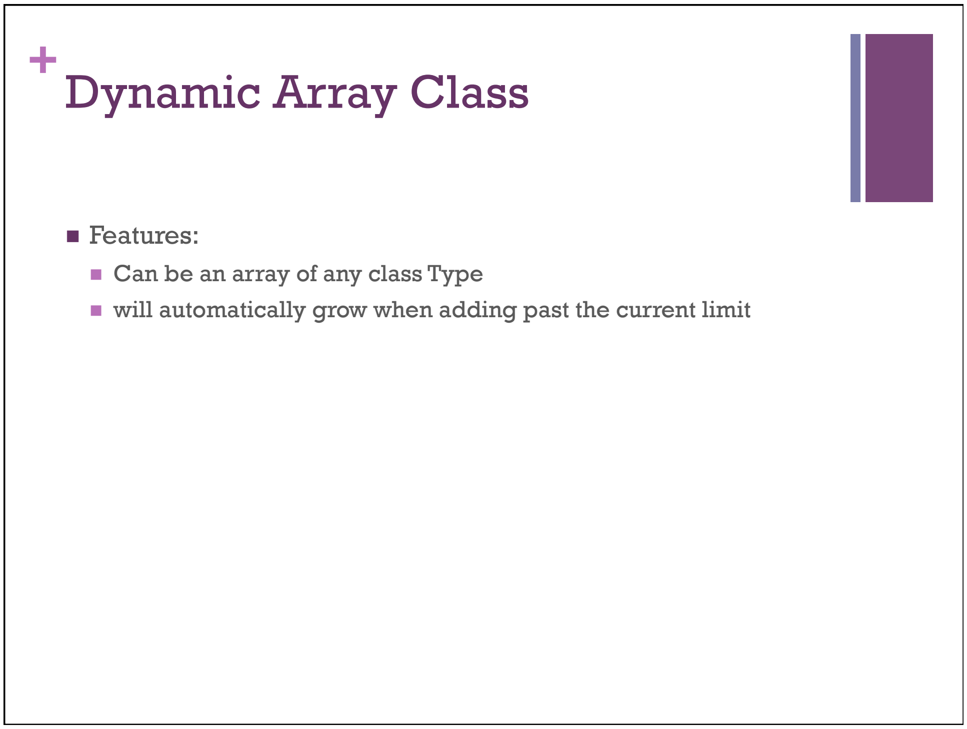
+ Basic Data Structures

- Linked List
  - dynamic size
  - linear time to access an element
  - cost to grow
    - constant to linear time
  - cost to shrink
    - constant to linear time
    - have to manage single node removal
  - no enforcement of order



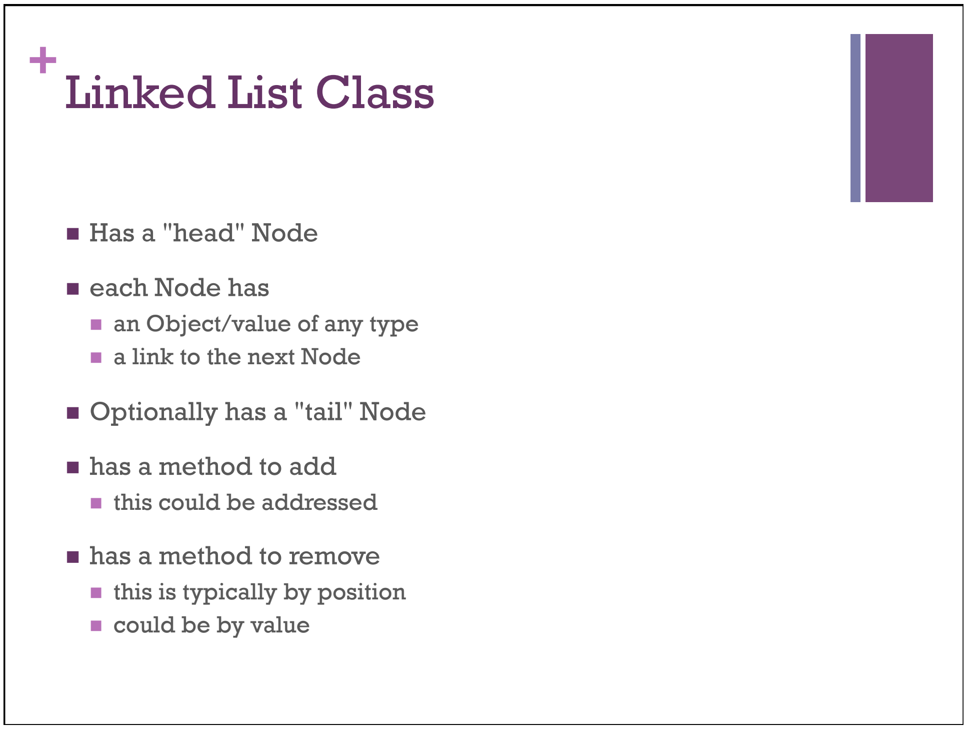
Basic Data Structures

- Arrays
  - fixed size
  - constant time access if you know the address
  - cost to grow
    - size of array, n, time to grow the array
    - have to manage/discard smaller array
  - no enforcement of order
- Linked List
  - dynamic size
  - linear time to access an element
  - cost to grow
    - constant to linear time
  - cost to shrink
    - constant to linear time
    - have to manage single node removal
  - no enforcement of order



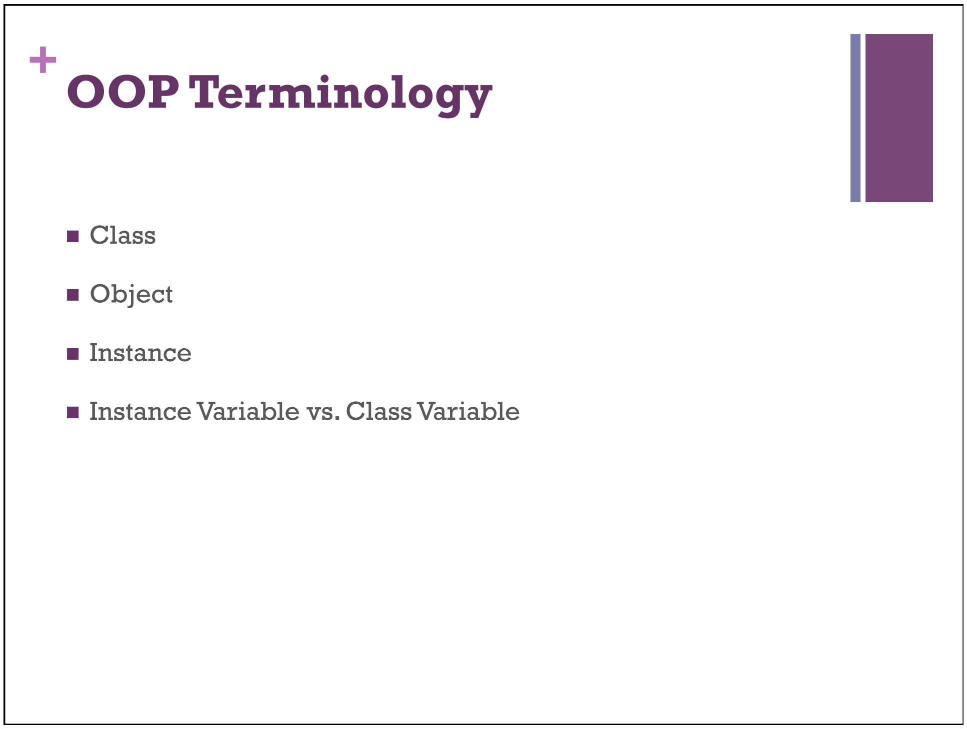
Dynamic Array Class

- Features:
  - Can be an array of any class Type
  - will automatically grow when adding past the current limit



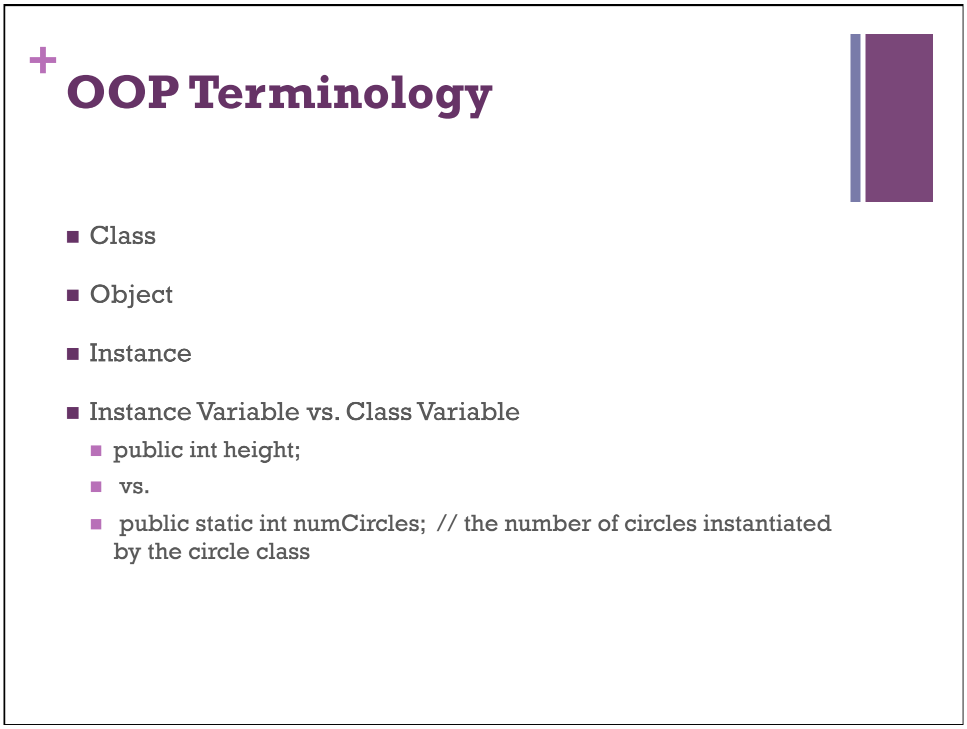
+ Linked List Class

- Has a "head" Node
- each Node has
  - an Object/value of any type
  - a link to the next Node
- Optionally has a "tail" Node
- has a method to add
  - this could be addressed
- has a method to remove
  - this is typically by position
  - could be by value



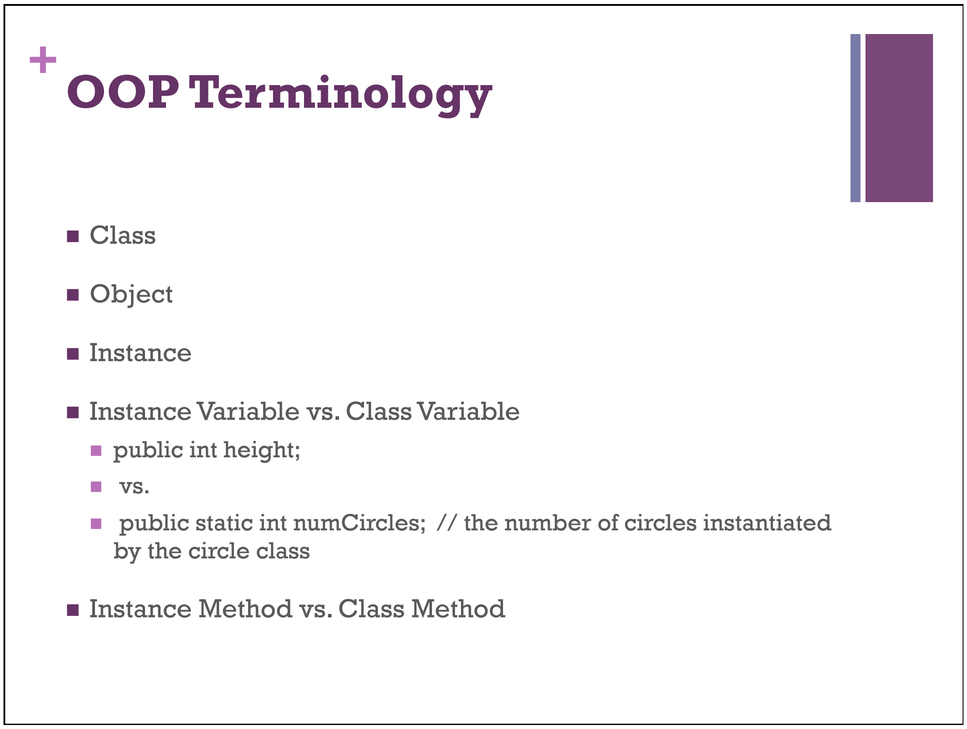
+ OOP Terminology

- Class
- Object
- Instance
- Instance Variable vs. Class Variable



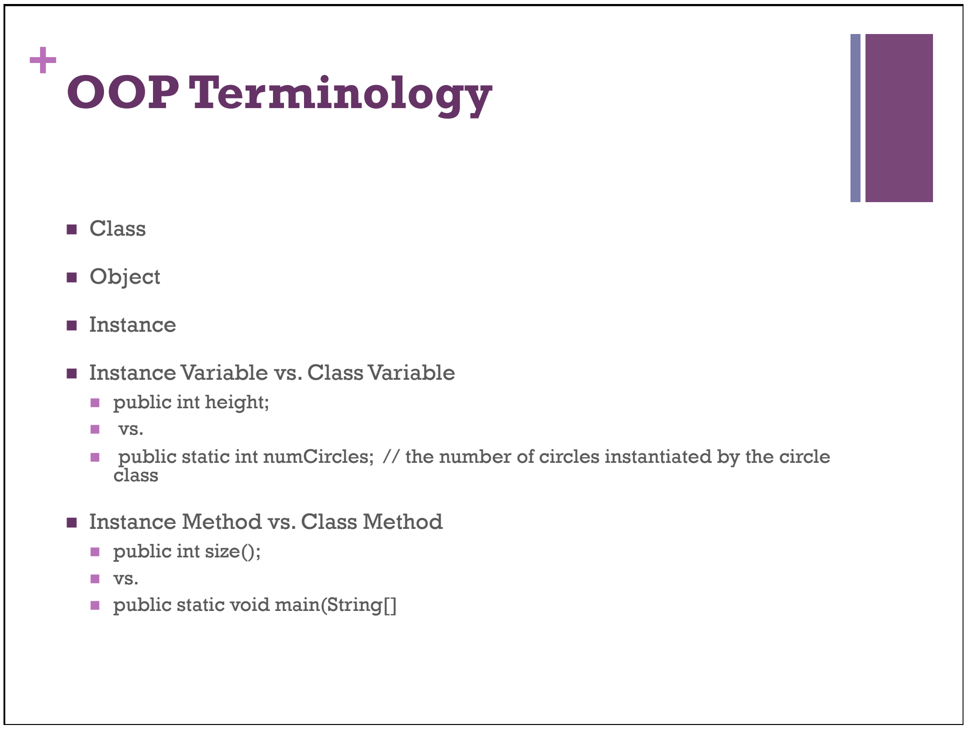
+
OOP Terminology

- Class
- Object
- Instance
- Instance Variable vs. Class Variable
  - public int height;
  - vs.
  - public static int numCircles; // the number of circles instantiated by the circle class



+
OOP Terminology

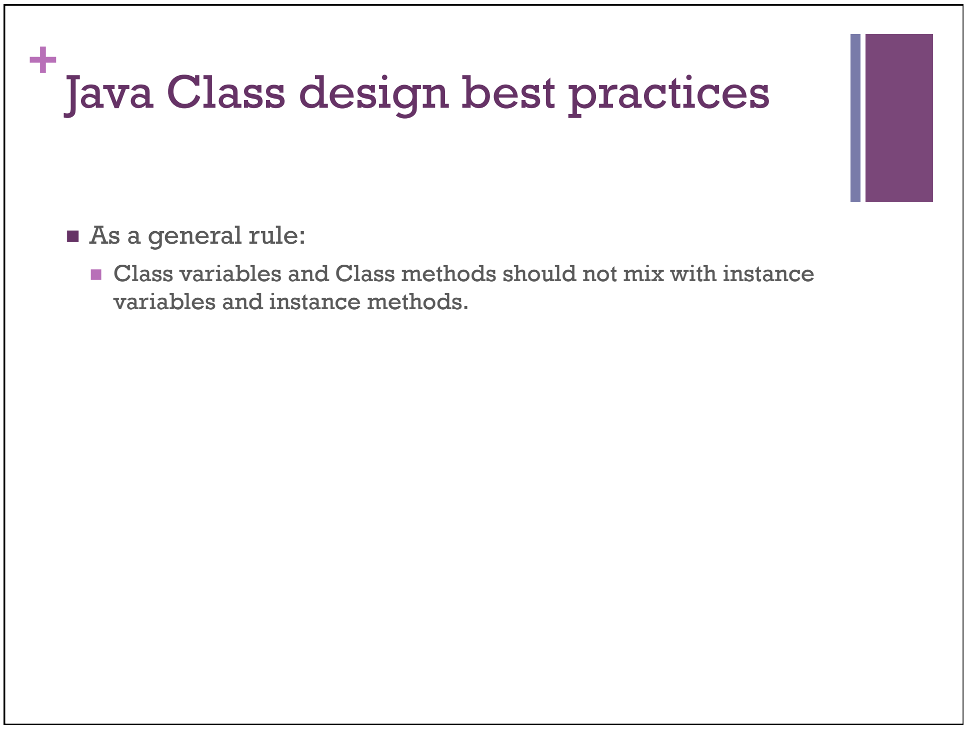
- Class
- Object
- Instance
- Instance Variable vs. Class Variable
  - public int height;
  - vs.
  - public static int numCircles; // the number of circles instantiated by the circle class
- Instance Method vs. Class Method



+

OOP Terminology

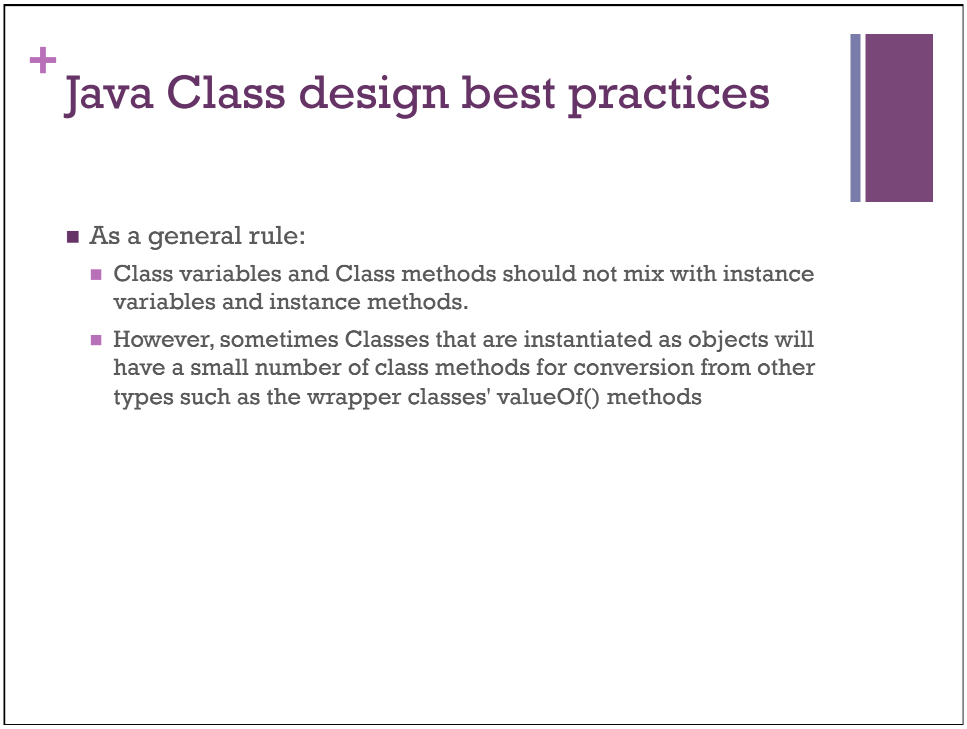
- Class
- Object
- Instance
- Instance Variable vs. Class Variable
  - public int height;
  - vs.
  - public static int numCircles; // the number of circles instantiated by the circle class
- Instance Method vs. Class Method
  - public int size();
  - vs.
  - public static void main(String[]



+

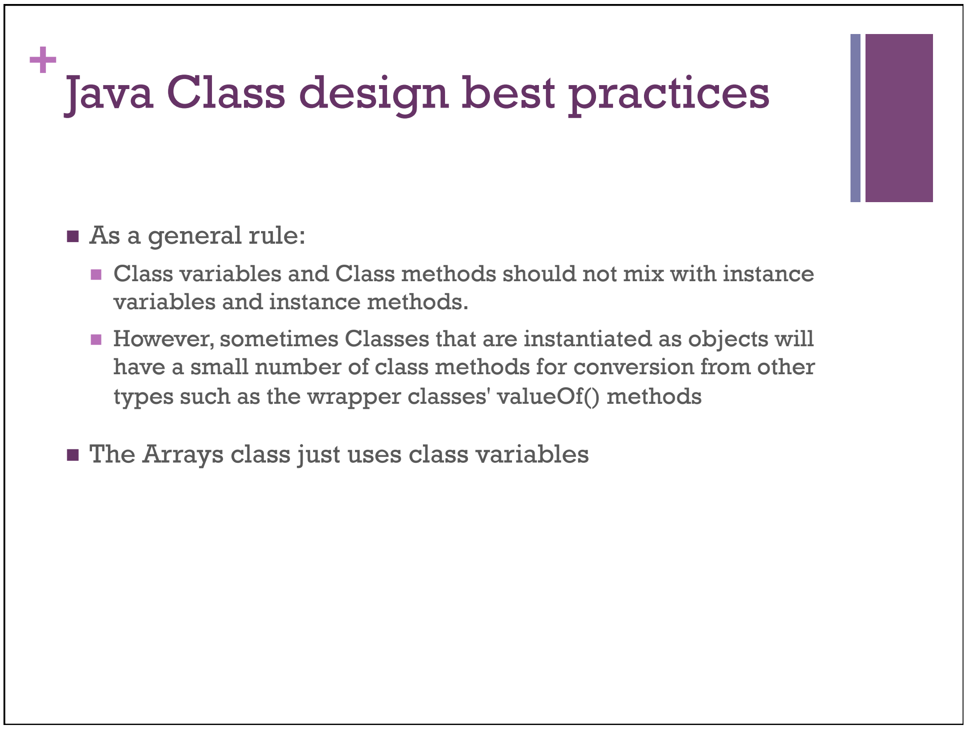
Java Class design best practices

- As a general rule:
  - Class variables and Class methods should not mix with instance variables and instance methods.



+ Java Class design best practices

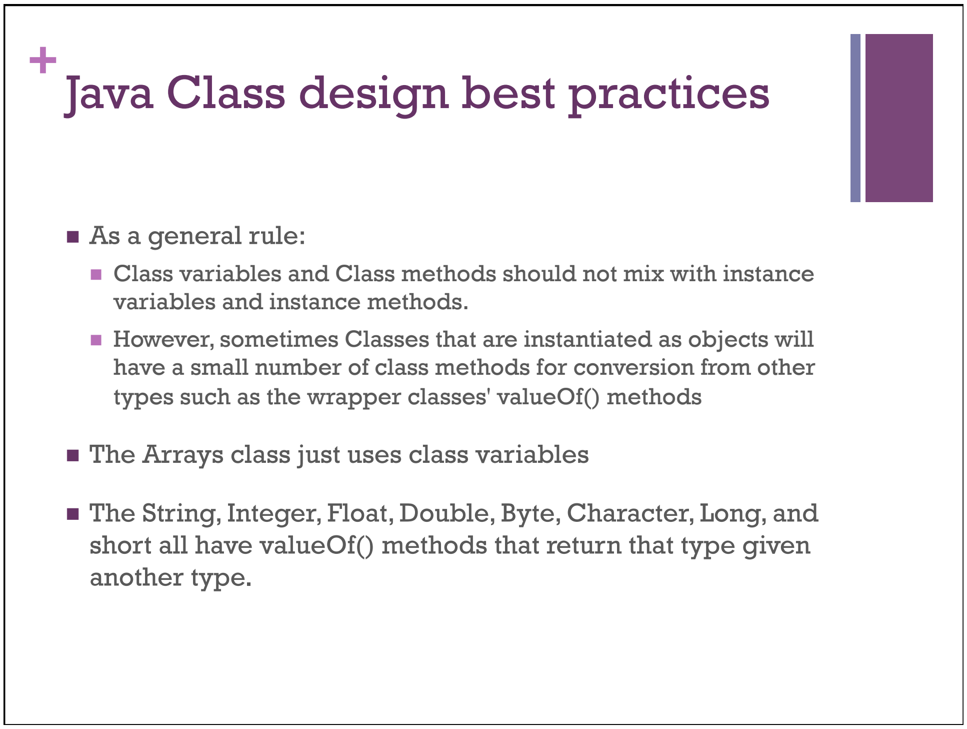
- As a general rule:
  - Class variables and Class methods should not mix with instance variables and instance methods.
  - However, sometimes Classes that are instantiated as objects will have a small number of class methods for conversion from other types such as the wrapper classes' valueOf() methods



+ Java Class design best practices

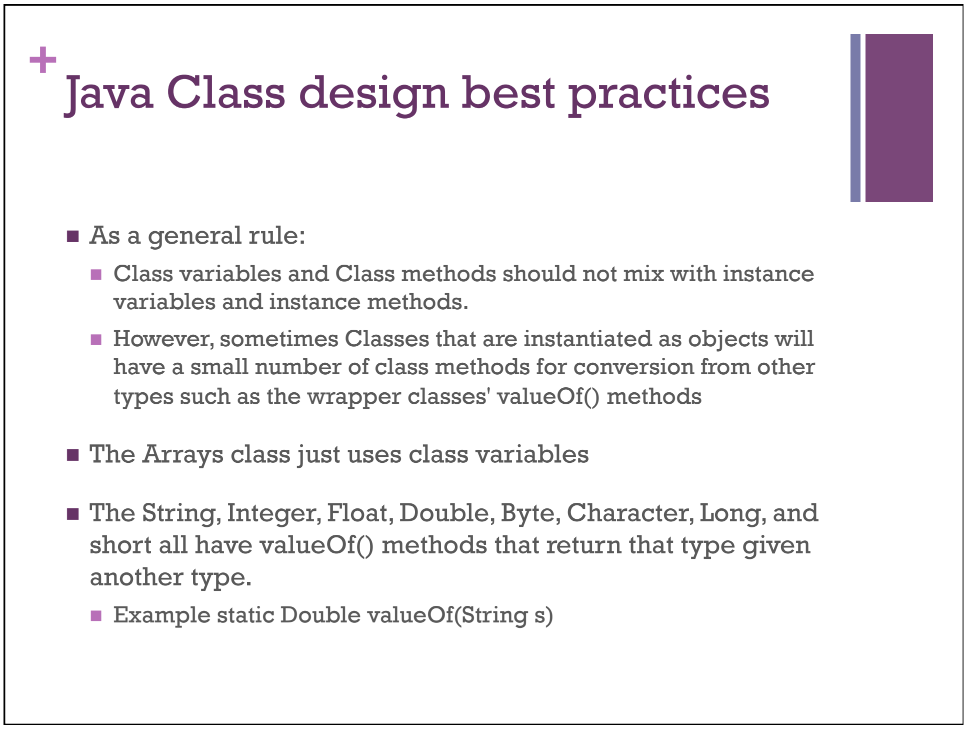
- As a general rule:
  - Class variables and Class methods should not mix with instance variables and instance methods.
  - However, sometimes Classes that are instantiated as objects will have a small number of class methods for conversion from other types such as the wrapper classes' valueOf() methods

- The Arrays class just uses class variables



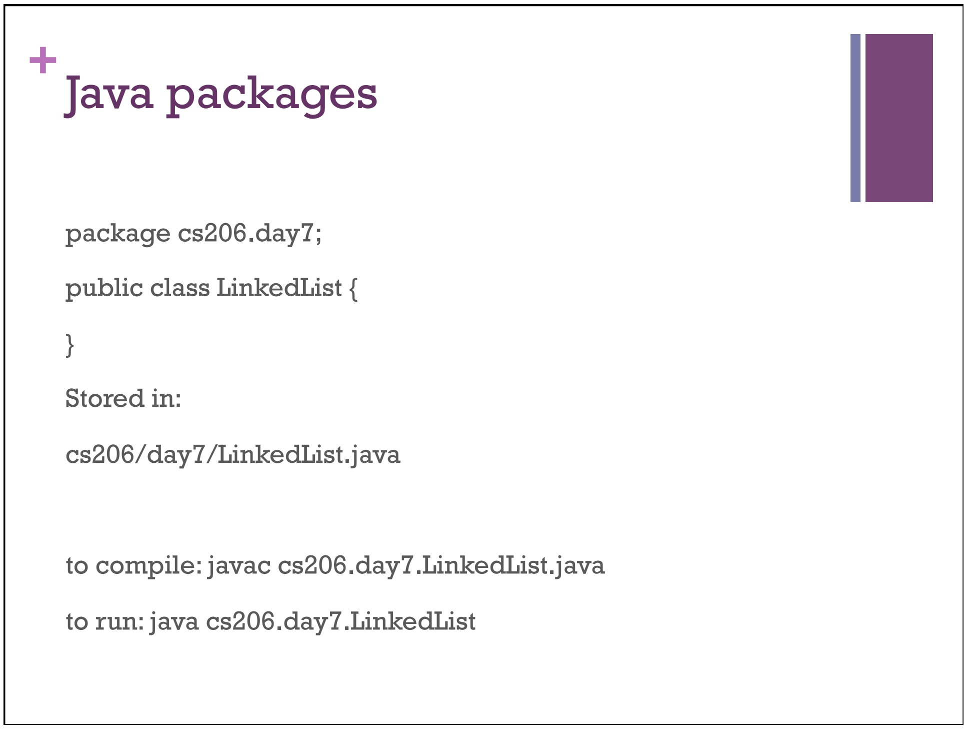
+ Java Class design best practices

- As a general rule:
  - Class variables and Class methods should not mix with instance variables and instance methods.
  - However, sometimes Classes that are instantiated as objects will have a small number of class methods for conversion from other types such as the wrapper classes' valueOf() methods
- The Arrays class just uses class variables
- The String, Integer, Float, Double, Byte, Character, Long, and short all have valueOf() methods that return that type given another type.



+ Java Class design best practices

- As a general rule:
  - Class variables and Class methods should not mix with instance variables and instance methods.
  - However, sometimes Classes that are instantiated as objects will have a small number of class methods for conversion from other types such as the wrapper classes' valueOf() methods
- The Arrays class just uses class variables
- The String, Integer, Float, Double, Byte, Character, Long, and short all have valueOf() methods that return that type given another type.
  - Example static Double valueOf(String s)



+ Java packages

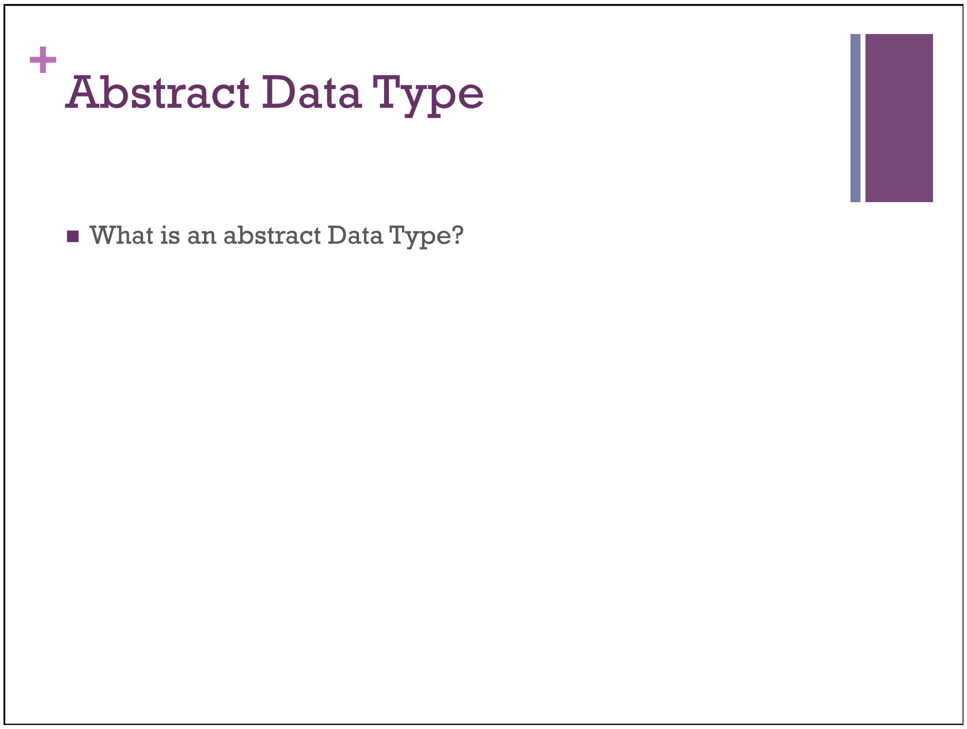
```
package cs206.day7;

public class LinkedList {

}
```

Stored in:

cs206/day7/LinkedList.java

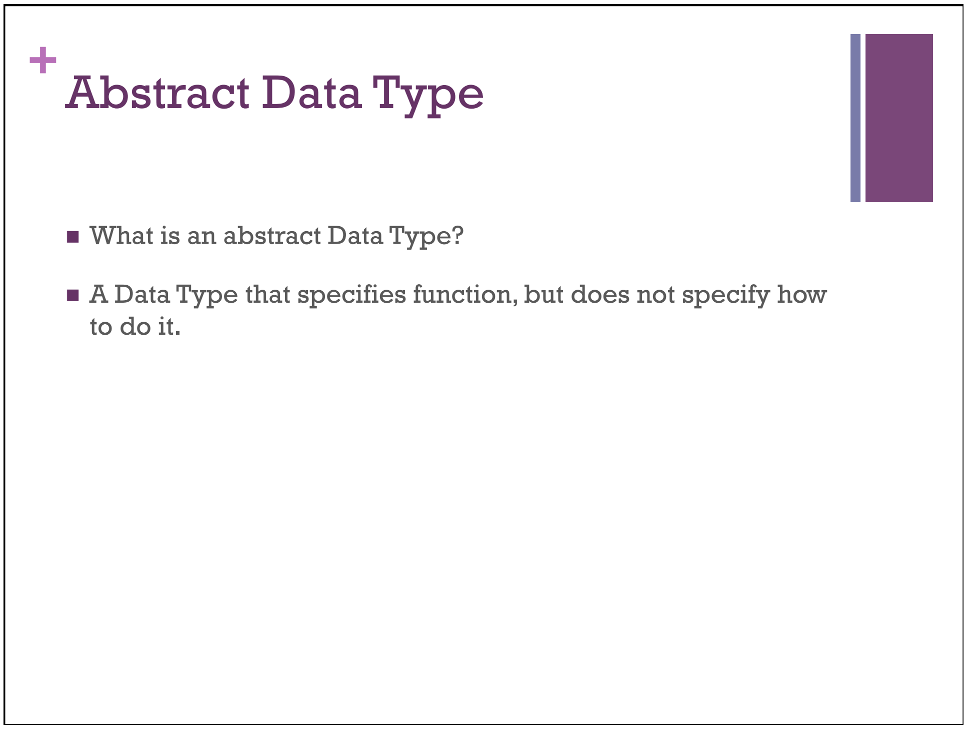
to compile: javac cs206.day7.LinkedList.java

to run: java cs206.day7.LinkedList



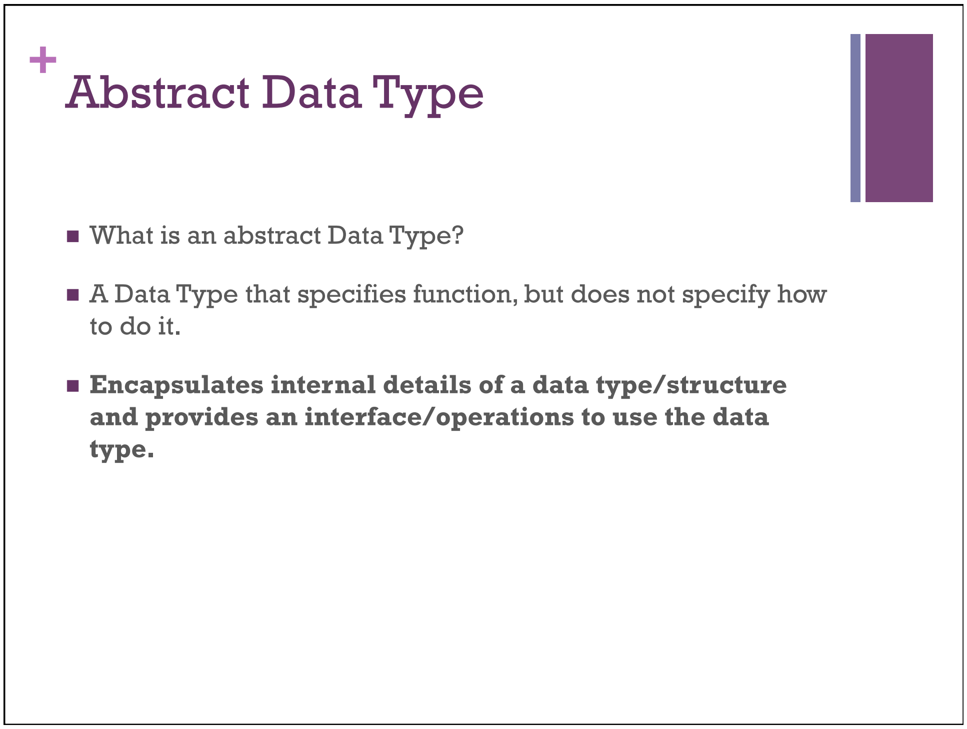
+ Abstract Data Type

- What is an abstract Data Type?



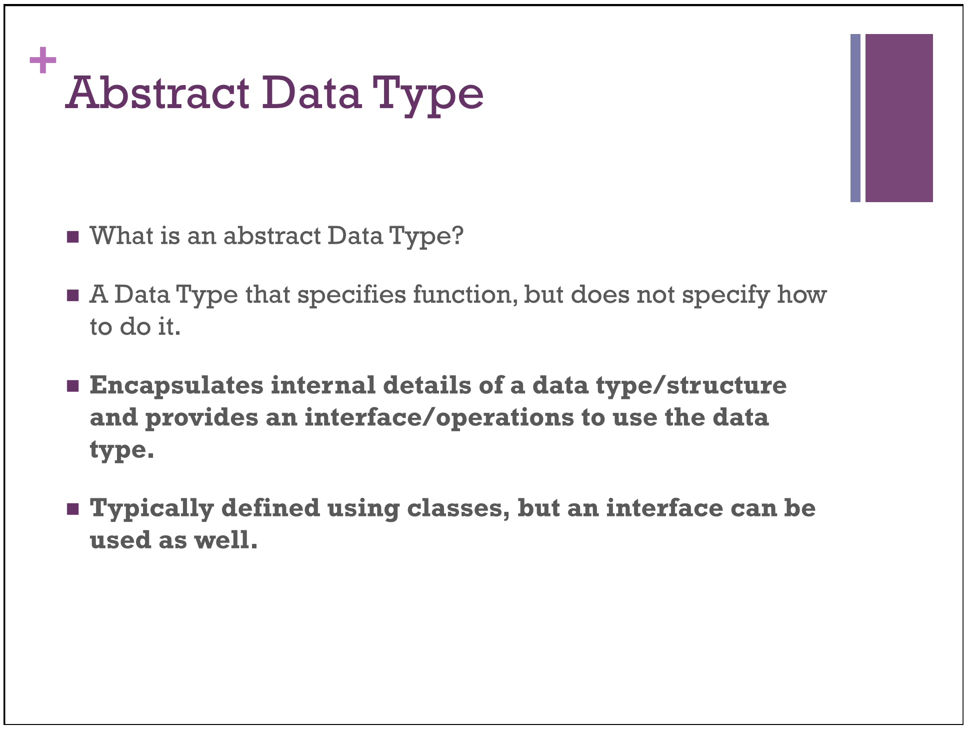
+ Abstract Data Type

- What is an abstract Data Type?
- A Data Type that specifies function, but does not specify how to do it.



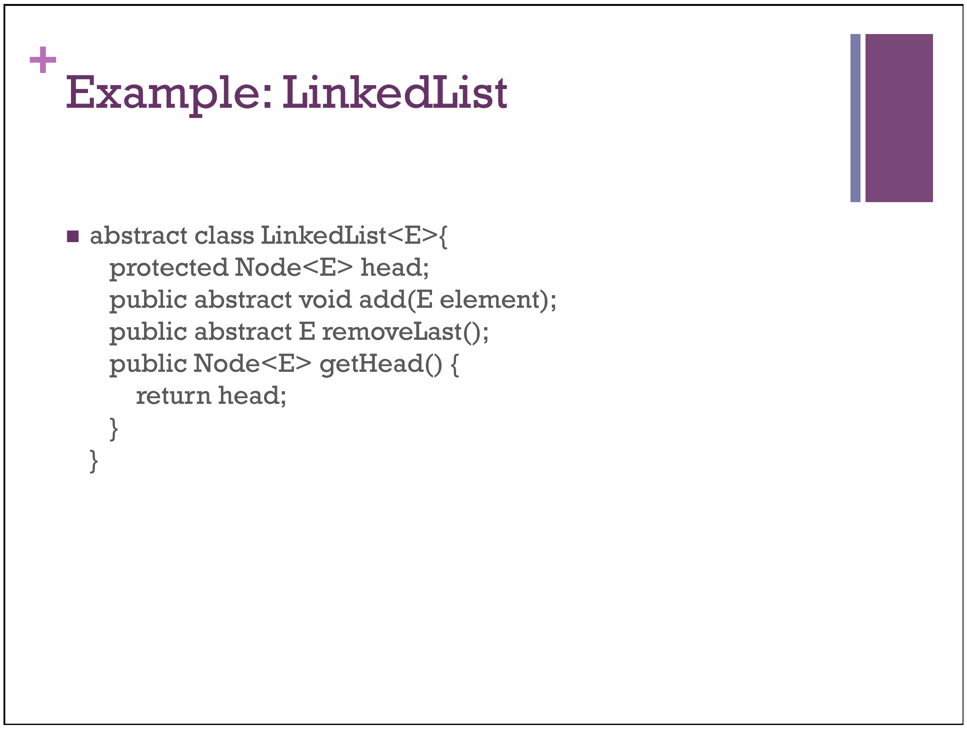
+ Abstract Data Type

- What is an abstract Data Type?
- A Data Type that specifies function, but does not specify how to do it.
- **Encapsulates internal details of a data type/structure and provides an interface/operations to use the data type.**



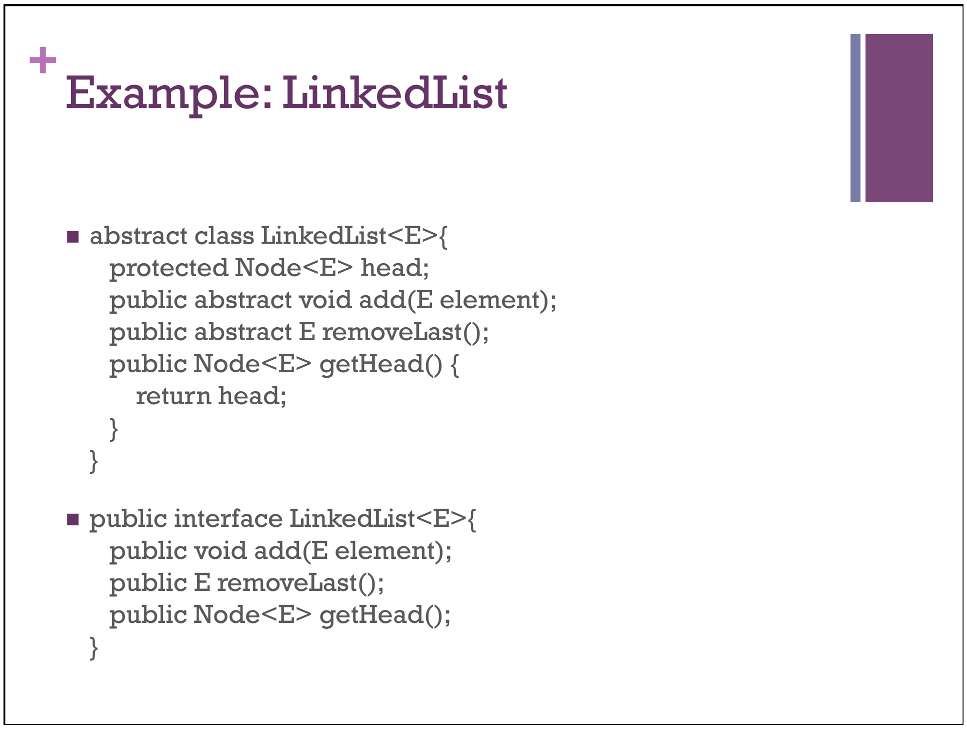
+ Abstract Data Type

- What is an abstract Data Type?
- A Data Type that specifies function, but does not specify how to do it.
- **Encapsulates internal details of a data type/structure and provides an interface/operations to use the data type.**
- **Typically defined using classes, but an interface can be used as well.**



+ Example: LinkedList

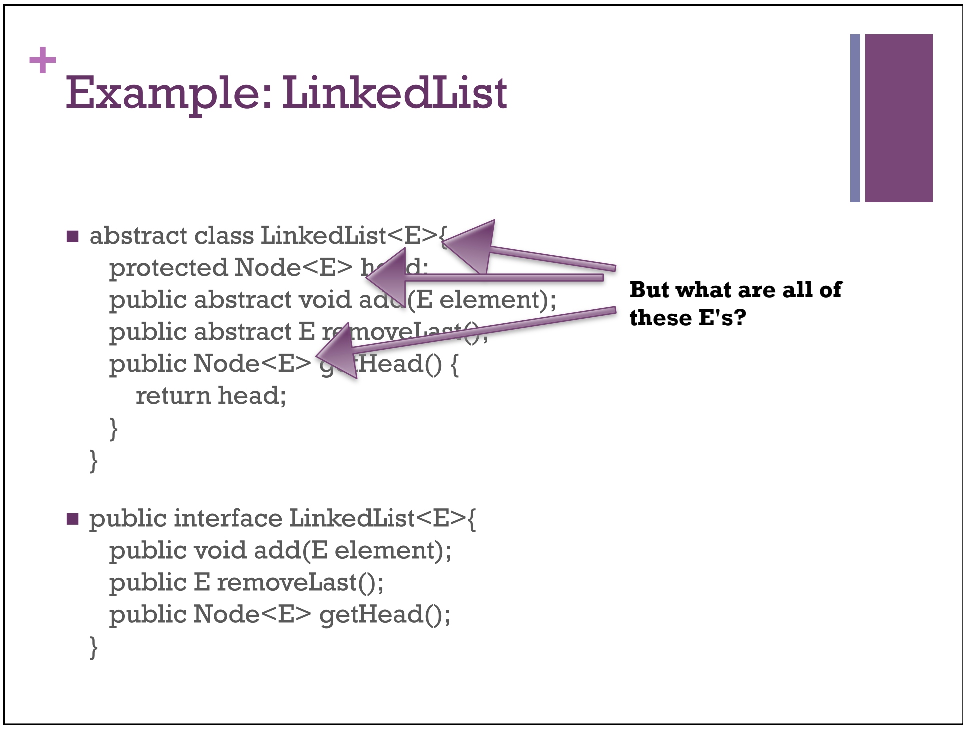
```
abstract class LinkedList<E>{
   protected Node<E> head;
   public abstract void add(E element);
   public abstract E removeLast();
   public Node<E> getHead() {
      return head;
   }
}
```



Example: LinkedList

```
abstract class LinkedList<E>{
   protected Node<E> head;
   public abstract void add(E element);
   public abstract E removeLast();
   public Node<E> getHead() {
      return head;
   }
}
```

```
public interface LinkedList<E>{
   public void add(E element);
   public E removeLast();
   public Node<E> getHead();
}
```

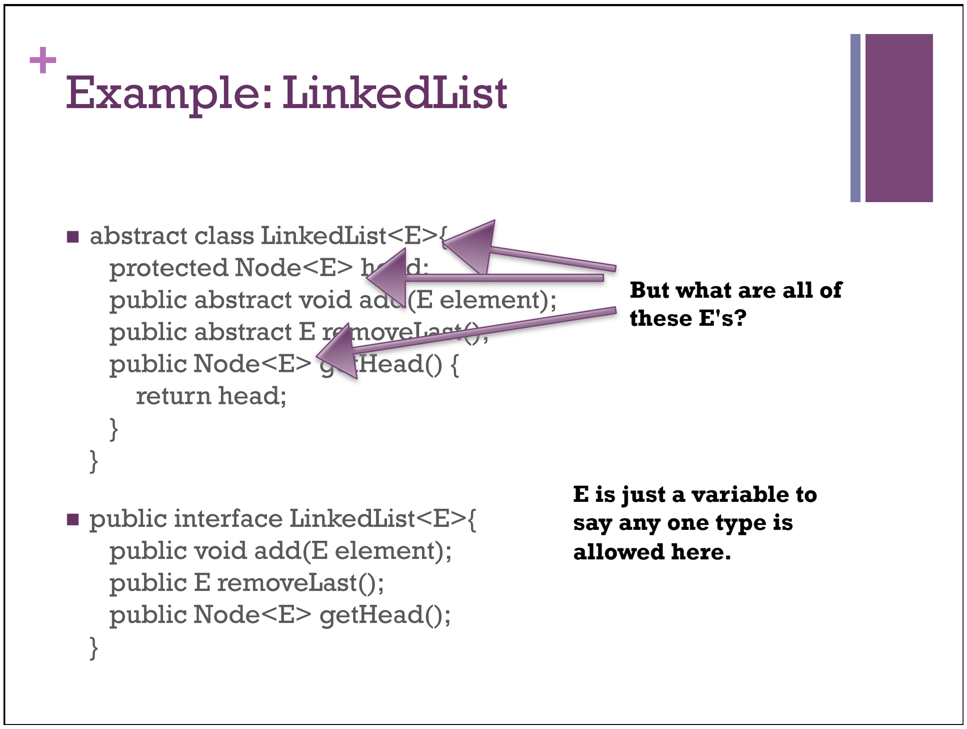


Example: LinkedList

```
abstract class LinkedList<E>{
   protected Node<E> head;
   public abstract void add(E element);
   public abstract E removeLast();
   public Node<E> getHead() {
      return head;
   }
}
```

But what are all of these E's?

```
public interface LinkedList<E>{
   public void add(E element);
   public E removeLast();
   public Node<E> getHead();
}
```



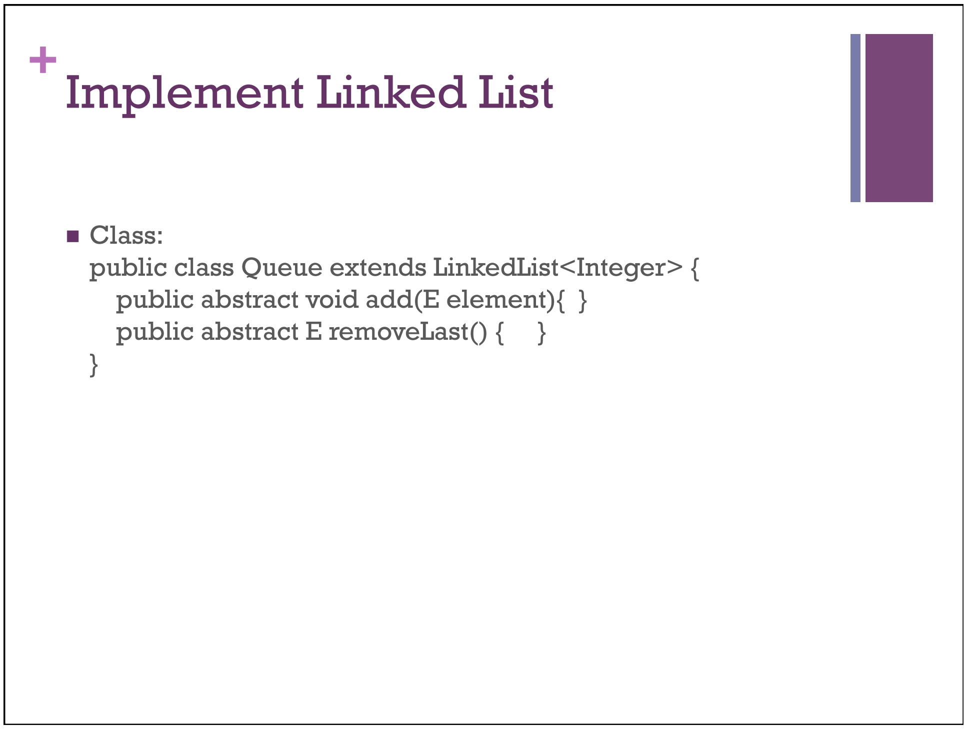
+ Example: LinkedList

-
  ```
  abstract class LinkedList<E>{
     protected Node<E> head;
     public abstract void add(E element);
     public abstract E removeLast();
     public Node<E> getHead() {
        return head;
     }
  }
  ```

But what are all of these E's?

-
  ```
  public interface LinkedList<E>{
     public void add(E element);
     public E removeLast();
     public Node<E> getHead();
  }
  ```

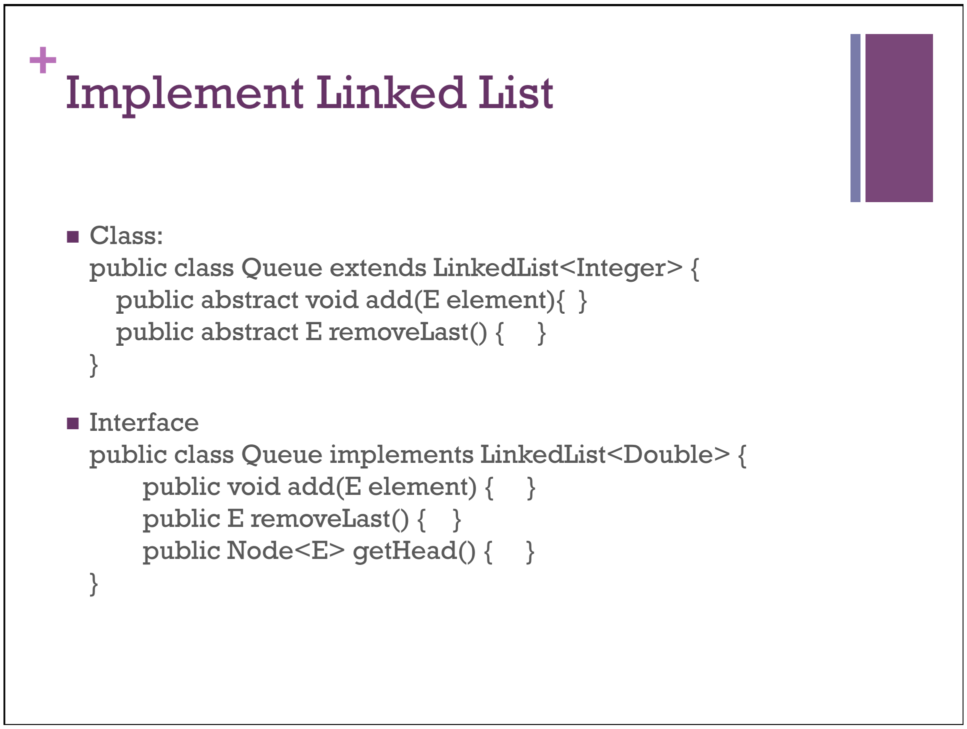
E is just a variable to say any one type is allowed here.



+ Implement Linked List

- Class:

  ```
  public class Queue extends LinkedList<Integer> {
     public abstract void add(E element){ }
     public abstract E removeLast() {    }
  }
  ```



+

Implement Linked List

- Class:

```
public class Queue extends LinkedList<Integer> {
   public abstract void add(E element){  }
   public abstract E removeLast() {     }
}
```

- Interface

```
public class Queue implements LinkedList<Double> {
      public void add(E element) {     }
      public E removeLast() {    }
      public Node<E> getHead() {     }
}
```