Values vs. References

- Does the following `swap` method work? Why or why not?
  ```java
  public static void main(String[] args) {
    int a = 7;
    int b = 35;
    // swap a with b?
    swap(a, b);
    System.out.println(a + " " + b);
  }
  
  public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
  }
  ```

Assignment and References

In Java, assignment (a = b) always copies the value stored by variable a to the value stored by variable b.

Example:
```java
int a = 5;
int b = 6;
a = b;     // value of a is now 6
```

When a and b are reference types (arrays or objects), the value that gets copied is a memory address.

Example:
```java
int[] a = {5};  // a's value is mem. loc. 0x0034FACE
int[] b = {6};  // b's value is mem. loc. 0x0034BEFF
a = b;          // a's value is mem. loc. 0x0034BEFF
```

More on references

When setting the contents of an array or the fields of an object, Java first goes to the memory location specified by the variable and then sets the appropriate value.

Example:
```java
int[] a = {1,2};
int[] b = {3,4};
a[0] = 10;
a = b;
a[0] = 100;
b = a;
```

We'll work out what's happening in memory on the blackboard.
Midterm Problem

Suppose the following class has been defined:

```java
class IntRef {
    public int data;
    public IntRef(int data) {
        this.data = data;
    }
}
```

What does the following code print out when run?

```java
IntRef var1 = new IntRef(1);
IntRef var2 = new IntRef(2);
IntRef var3 = new IntRef(3);
var2 = var1;
var2.data = 4;
var1 = var3;
var2.data = 5;
var1.data = 6;
System.out.println("1: "+ var1.data);
System.out.println("2: "+ var2.data);
System.out.println("3: "+ var3.data);
```

Method calls vs. assignment

- Calling a method is equivalent to assigning the method parameters to their given arguments:

```java
public static void main(String[] args) {
    int a = 7;
    int b = 35;
    // swap a with b? (Doesn’t work!)
    swap(a, b);
    System.out.println(a + " " + b);
}
```

```java
public static void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}
```

```java
public static void main(String[] args) {
    int[] a = {7};
    int[] b = {35};
    // swap a with b? (Doesn’t work!)
    swap(a, b);
    System.out.println(a[0] + " " + b[0]);
}
```
Method calls vs. assignment

- Calling a method is equivalent to assigning the method parameters to their given arguments:

  ```java
  public static void main(String[] args) {
      int[] a = {7};
      int[] b = {35};
      // swap a with b? (Works!)
      int[] x = a;
      int[] y = b;
      int temp = x[0];
      x[0] = y[0];
      y[0] = temp;
      System.out.println(a[0] + " " + b[0]);
  }
  ```

Value semantics

- **value semantics**: Behavior where values are copied when assigned to each other or passed as parameters.
  - When one primitive is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

  ```java
  int x = 5;
  int y = x;  // x = 5, y = 5
  y = 17;    // x = 5, y = 17
  x = 8;    // x = 8, y = 17
  ```

Reference semantics

- **reference semantics**: Behavior where variables actually store the address of an object in memory.
  - When one reference variable is assigned to another, the object is **not** copied; both variables refer to the same object.

  ```java
  int[] a1 = {4, 5, 2, 12, 14, 14, 9};
  int[] a2 = a1;    // refers to same array as a1
  a2[0] = 7;
  System.out.println(a1[0]);  // 7
  ```

Review Question

- What does the following print?

  ```java
  public static void main(String[] args) {
      String a = "AAA";
      String b = "BBB";
      String x = a;
      String y = b;
      String temp = x;
      x = y;
      y = temp;
      System.out.println(a + " " + b);
      System.out.println(x + " " + y);
  }
  ```
References to same type

- What would happen if we had a class that declared one of its own type as a field?

```java
public class Strange {
    private String name;
    private Strange other;
}
```

- Will this compile?
  - If so, what is the behavior of the `other` field? What can it do?
  - If not, why not? What is the error and the reasoning behind it?

Array-Based List Review

- Array-based lists are what we've studied so far
  - `ArrayIntList`, `ArrayList`, `SortedIntList` all use arrays

- Arrays use a contiguous block of memory
  - This means all elements are adjacent to each other

```
0 1 2 3 4 5 6 7 8
6 2 5 3 7 1 4 -9 -8
```

Advantages and Disadvantages

- Advantages of array-based lists
  - random access: can get any element in the entire array quickly
    - kind of like jumping to any scene on a DVD (no fast-forwarding required)

- Disadvantages of array-based lists
  - can't insert/remove elements at the front/middle easily
    - have to shift the other elements
  - can't resize array easily
    - have to create a new, bigger array

Linked Lists

- A linked list is a type of list

- But instead of a contiguous block of memory, like this:
  - `ArrayIntList`, `ArrayList`, `SortedIntList` all use arrays
  - Arrays use a contiguous block of memory
  - This means all elements are adjacent to each other

```
0 1 2 3 4 5 6 7 9
```

- Linked list elements are scattered throughout memory:

```
8 9 7 5 6
```

- But now the elements are unordered. How do linked lists keep track of everything?
**Linked Lists**

- Each element must have a reference to the next element:

```
8 <-> 9 <-> 7 <-> 5 <-> 6
```

- Now, so long as we keep track of the first element (the front), we can keep track of all the elements.

**Linked data structures**

- All of the collections we will use and implement in this course use one of the following two underlying data structures:
  - an **array** of all elements
    - `ArrayList, Stack, HashSet, HashMap`
    ```
    42 -3 17 9
    ```
  - a set of **linked objects**, each storing one element, and one or more reference(s) to other element(s)
    - `LinkedList, TreeSet, TreeMap`
    ```
    front -> 42 <-> -3 <-> 17 <-> 9 <-> null
    ```

**Linked Lists**

- These references to the next element mean that linked lists have sequential access.

- This means that to get to elements in the middle of the list, we must first start at the front and follow all the links until we get to the middle:
  - kind of like fast-forwarding on a VHS tape
  - so getting elements from the middle/back is slow

- Linked lists also do some things well:
  - linked lists can insert elements quickly (no “shifting” needed)
  - linked lists can always add more elements (no set capacity)

- So there are tradeoffs between array lists and linked lists.

**A list node class**

```java
public class ListNode {
    int data;
    ListNode next;
}
```

- Each list node object stores:
  - one piece of integer data
  - a reference to another list node (it does NOT contain another `ListNode` object)

- ListNodes can be "linked" into chains to store a list of values:

```
42 <-> -3 <-> 17 <-> 9 <-> null
```
List node client example

```java
public class ConstructList1 {
    public static void main(String[] args) {
        ListNode list = new ListNode();
        list.data = 42;
        list.next = new ListNode();
        list.next.data = -3;
        list.next.next = new ListNode();
        list.next.next.data = 17;
        list.next.next.next = null;
        System.out.println(list.data + " "+ list.next.data
                + " "+ list.next.next.data);
            // 42 -3 17
    }
}
```

List node w/ constructor

```java
public class ListNode {
    int data;
    ListNode next;
    public ListNode(int data) {
        this.data = data;
        this.next = null;
    }
    public ListNode(int data, ListNode next) {
        this.data = data;
        this.next = next;
    }
}
```

- Exercise: Modify the previous client to use these constructors.

List node client example

- Two possible solutions:
  `ListNode list = new ListNode(42, new ListNode(-3, new ListNode(17)));`
  `ListNode list = new ListNode(17);
   list = new ListNode(-3, list);
   list = new ListNode(42, list);`

  - NOTE: In the second solution, the nodes are added in reverse order!

- List creation is still somewhat tedious. More on that next time.

Linked node problem 1

- What set of statements turns this picture:
  `list.next.next = new ListNode(30);`

- Into this?
  `list.next.next = new ListNode(30);`
Linked node problem 2

• What set of statements turns this picture:

    list → data: 10 → next: list → data: 20 → next:

• Into this?

    list → data: 30 → next: 10 → next: 20 → next:

list = new ListNode(30, list);

Linked node problem 3

• What set of statements turns this picture:

    list1 → data: 10 → next: list1 → data: 20 → next:

    list2 → data: 30 → next: list2 → data: 40 → next:

• Into this?

    list1 → data: 10 → next: list1 → data: 30 → next: list1 → data: 20 → next:

    list2 → data: 40 → next:

list1 = new ListNode(10, list1);
list2 = new ListNode(30, list2);

Two possible solutions:
ListNode temp = list1.next;
list1.next = list2;
list2 = list2.next;
list1.next.next = temp;

ListNode temp = list2.next;
list2.next = list1.next;
list1.next = list2;
list2 = temp;

Linked node problem 3

• Two possible solutions:
ListNode temp = list1.next;
list1.next = list2;
list2 = list2.next;
list1.next.next = temp;

ListNode temp = list2.next;
list2.next = list1.next;
list1.next = list2;
list2 = temp;

Summary: A list node class

public class ListNode {
    int data;
    ListNode next;
}

• Each list node object stores:
  - one piece of integer data
  - a reference to another list node (it does NOT contain another ListNode object)

• ListNode objects can be "linked" into chains to store a list of values:

• Exercise: Write a method to find length of a linked list, given a reference to its first node.
References vs. objects

\[ \text{variable} = \text{value}; \]

A variable (left side of =) is an arrow (the base of an arrow)
A value (right side of =) is an object (a box; what an arrow points at)

- For the list at right:
  - \text{a.next} = \text{value};
    means to adjust where ① points
  - \text{variable} = \text{a.next};
    means to make variable point at ②

Reassigning references

- when you say:
  - \text{a.next} = \text{b.next};
- you are saying:
  - "Make the variable \text{a.next} refer to the same value as \text{b.next}.
  - Or, "Make \text{a.next} point to the same place that \text{b.next} points."

Basic Linked List Questions

- Suppose you have two variables of type \text{ListNode} named \text{p} and \text{q}. Consider the following situation:

  \text{p} \quad \text{data next} \quad \text{data next}
  \quad 2 \quad 4
  
  \text{q} \quad \text{data next} \quad \text{data next}
  \quad 3 \quad 9

- How many variables of type \text{ListNode} are there?
- How many \text{ListNode} objects are there?
Basic Linked List Questions

- How many ListNode objects are there?
  - 4, circled in green

Linked node question

- Suppose we have a long chain of list nodes:
  
  - We don't know exactly how long the chain is.

- How would we print the data values in all the nodes?

Traversing a list correctly

- The correct way to print every value in the list:

  ```java
  ListNode current = list;
  while (current != null) {
    System.out.println(current.data);
    current = current.next; // move to next node
  }
  
  - Changing current does not damage the list.

Linked list vs. array

- Algorithm to print list values:

  ```java
  ListNode front = ...;
  while (current != null) {
    System.out.println(current.data);
    current = current.next;
  }
  
  Similar to array code:

  ```java
  int[] a = ...;
  int i = 0;
  while (i < a.length) {
    System.out.println(a[i]);
    i++;
  }
  ```
### Relationship to Array Code

- A table explaining this relationship:

<table>
<thead>
<tr>
<th>Description</th>
<th>Array Code</th>
<th>Linked List Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>go to front of list</td>
<td>int i = 0; ListNode current = front;</td>
<td>1. current = front;</td>
</tr>
<tr>
<td>continue?</td>
<td>i &lt; size current != null</td>
<td>2. if (i &lt; size &amp;&amp; current != null) {</td>
</tr>
<tr>
<td>get current value</td>
<td>elementData[i] current.data</td>
<td>3. current = current.next;</td>
</tr>
<tr>
<td>go to next element</td>
<td>i++; current = current.next</td>
<td></td>
</tr>
</tbody>
</table>

- This may be helpful if you are comfortable with arrays

### For Loops

- Of course, we usually write the array code in a for loop:

```java
for (int i = 0; i < size; i++) {
    System.out.println(elementData[i]);
}
```

- And we can still do this with the linked list code:

```java
for (ListNode current = front; current != null; current = current.next) {
    System.out.println(current.data);
}
```

- Whether you use a for loop or a while loop to traverse the linked list is up to you.

### A LinkedIntList class

- Let's write a collection class named `LinkedIntList`.
  - Has the same methods as `ArrayIntList`:
    - `add`, `add`, `get`, `indexOf`, `remove`, `size`, `toString`
  - The list is internally implemented as a chain of linked nodes
    - The `LinkedIntList` keeps a reference to its `front` as a field
    - `null` is the end of the list; a `null` front signifies an empty list

```java
public class LinkedIntList {
    private ListNode front;

    public LinkedIntList() {
        front = null;
    }

    methods go here
}
```

### LinkedIntList class v1

```java
public class LinkedIntList {
    private ListNode front;
    public LinkedIntList() {
        front = null;
    }
}
```
Implementing `add`

// Adds the given value to the end of the list.
public void add(int value) {
    ...
}

- How do we add a new node to the end of a list?
- Does it matter what the list’s contents are before the add?

---

Adding to an empty list

- Before adding 20:

  front =

- After:

  front =

  - We must create a new node and attach it to the list.

Adding to non-empty list

- Before adding value 20 to end of list:

  front =

- After:

  front =

  element 0
Don't fall off the edge!

• To add/remove from a list, you must modify the `next` reference of the node before the place you want to change.

  - Where should `current` be pointing, to add 20 at the end?
  - What loop test will stop us at this place in the list?

The add method

```java
// Adds the given value to the end of the list.
public void add(int value) {
    if (front == null) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to the end of an existing list
        ListNode current = front;
        while (current.next != null) {
            current = current.next;
        }
        current.next = new ListNode(value);
    }
}
```

Implementing get

```java
// Returns value in list at given index.
public int get(int index) {
    ...
}
```

- Exercise: Implement the `get` method.

The get method

```java
// Returns value in list at given index.
// Precondition: 0 <= index < size()
public int get(int index) {
    ListNode current = front;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```
Implementing `add` (2)

```java
// Inserts the given value at the given index.
public void add(int index, int value) {
    ...
}

- Exercise: Implement the two-parameter `add` method.
```

The `add` method (2)

```java
// Inserts the given value at the given index.
// Precondition: 0 <= index <= size()
public void add(int index, int value) {
    if (index == 0) {
        // adding to an empty list
        front = new ListNode(value, front);
    } else {
        // inserting into an existing list
        ListNode current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current.next;
        }
        current.next = new ListNode(value, current.next);
    }
}
```

Implementing `get`

```java
// Returns value in list at given index.
public int get(int index) {
    ...
}

- Exercise: Implement the `get` method.
```

The `get` method

```java
// Returns value in list at given index.
// Precondition: 0 <= index < size()
public int get(int index) {
    ListNode current = front;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```
Warm-up questions

- What is the difference between a `LinkedIntList` and a `ListNode`?
- What is the difference between an empty list and a null list?
  - How do you create each one?
- What effect does this code have on a `LinkedIntList`?
  ```java
  ListNode current = front;
  current = null;
  ```
- Implement `remove()`, which removes the item at the head of a linked list. Recall: front references the start of the list.

Warm-up answers

- A list consists of 0 to many node objects.
  - Each node holds a single data element value.
- null list: `LinkedIntList list = null;`
  empty list: `LinkedIntList list = new LinkedIntList();`
- The code doesn’t change the list.
  You can change a list only in one of the following two ways:
  - Modify its `front` field value.
  - Modify the `next` reference of a node in the list.
- `void remove() { front = front.next; }

Implementing `remove`

```java
// Removes and returns the list's first value.
public int remove() {
    ...
}
```

- How do we remove the front node from a list?
- Does it matter what the list's contents are before the remove?

Removing from a list

- Before removing element at index 1:
  ![Diagram of a list with elements 0, 42, -3, 20, and 42 next to elements 0, 1, and 2]
- After:
  ![Diagram of a list with elements 0, 42, 20, and 42 next to elements 0 and 1]
Removing from the front

- Before removing element at index 0:

  - Front: \(42\) → \(-3\) → \(20\)
  - After removal:
    - Front: \(-3\) → \(20\)

- After:

  - Front: \(-3\) → \(20\)

Removing the only element

- Before:
  - Front: \(20\)
  - After removal:
    - Front: \(\) → \(\) → \(\)
    - We must change the front field to store `null` instead of a node.
    - Do we need a special case to handle this?

Exercise

- Write a method `addSorted` that accepts an integer value as a parameter and adds that value to a sorted list in sorted order.
  - Before `addSorted(17)`:
    - Front: \(\) → \(-4\) → \(8\) → \(22\)
  - After `addSorted(17)`:
    - Front: \(\) → \(-4\) → \(8\) → \(17\) → \(22\)
The common case

- Adding to the middle of a list:
  \texttt{addSorted(17)}

\begin{itemize}
  \item [The data] \begin{array}{c}
  \text{element 0} \\
  \text{element 1} \\
  \text{element 2}
  \end{array}
  \begin{array}{c}
  \text{front} = \\
  \text{-4} \\
  \text{8} \\
  \text{22}
  \end{array}
\end{itemize}

- Which references must be changed?
- What sort of loop do we need?
- When should the loop stop?

Key idea: peeking ahead

- Corrected version of the loop:
  \texttt{ListNode current = front;}
  while (current.next.data < value) {
    current = current.next;
  }

\begin{itemize}
  \item [The data] \begin{array}{c}
  \text{element 0} \\
  \text{element 1} \\
  \text{element 2}
  \end{array}
  \begin{array}{c}
  \text{front} = \\
  \text{-4} \\
  \text{8} \\
  \text{22}
  \end{array}
\end{itemize}

- This time the loop stops in the right place.

First attempt

- An incorrect loop:
  \texttt{ListNode current = front;}
  while (current.data < value) {
    current = current.next;
  }

\begin{itemize}
  \item [The data] \begin{array}{c}
  \text{element 0} \\
  \text{element 1} \\
  \text{element 2}
  \end{array}
  \begin{array}{c}
  \text{front} = \\
  \text{-4} \\
  \text{8} \\
  \text{22}
  \end{array}
\end{itemize}

- What is wrong with this code?
  - The loop stops too late to affect the list in the right way.

Another case to handle

- Adding to the end of a list:
  \texttt{addSorted(42)}

\begin{itemize}
  \item [The data] \begin{array}{c}
  \text{element 0} \\
  \text{element 1} \\
  \text{element 2}
  \end{array}
  \begin{array}{c}
  \text{front} = \\
  \text{-4} \\
  \text{8} \\
  \text{22}
  \end{array}
\end{itemize}

\textit{Exception in thread "main": java.lang.NullPointerException}

- Why does our code crash?
- What can we change to fix this case?
Multiple loop tests

• A correction to our loop:
  ListNode current = front;
  while (current.next != null &&
         current.next.data < value) {
    current = current.next;
  }

  – We must check for a next of null before we check its .data.

Handling the front

• Another correction to our code:
  if (value <= front.data) {
    // insert at front of list
    front = new ListNode(value, front);
  } else {
    // insert in middle of list
    ListNode current = front;
    while (current.next != null &&
           current.next.data < value) {
      current = current.next;
    }
  }

  – Does our code now handle every possible case?

Third case to handle

• Adding to the front of a list:
  addSorted(-10)

  – What will our code do in this case?
  – What can we change to fix it?

Fourth case to handle

• Adding to (the front of) an empty list:
  addSorted(42)

  – What will our code do in this case?
  – What can we change to fix it?
Final version of code

// Adds given value to list in sorted order.
// Precondition: Existing elements are sorted
public void addSorted(int value) {
    if (front == null || value <= front.data) {
        // insert at front of list
        front = new ListNode(value, front);
    } else {
        // insert in middle of list
        ListNode current = front;
        while (current.next != null &&
               current.next.data < value) {
            current = current.next;
        }
    }
}

Linked vs. array lists

• We have implemented two collection classes:
  - ArrayIntList
    ```
    index   0  1  2  3
    value   42 -3 17  9
    ```
  - LinkedIntList
    ```
    front
    data   next
    data   data
    42     -3
    17     -
    9
    ```
  - They have similar behavior, implemented in different ways. We should be able to treat them the same way in client code.

An IntList interface

// Represents a list of integers.
public interface IntList {
    public void add(int value);
    public void add(int index, int value);
    public int get(int index);
    public int indexOf(int value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, int value);
    public int size();
}

public class ArrayIntList implements IntList {
    ...}

public class LinkedIntList implements IntList {
    ...}

Redundant client code

public class ListClient {
    public static void main(String[] args) {
        ArrayIntList list1 = new ArrayIntList();
        list1.add(18);
        list1.add(27);
        list1.add(93);
        System.out.println(list1);
        list1.remove(1);
        System.out.println(list1);
        LinkedIntList list2 = new LinkedIntList();
        list2.add(18);
        list2.add(27);
        list2.add(93);
        System.out.println(list2);
        list2.remove(1);
        System.out.println(list2);
    }
}
Client code w/ interface

```java
public class ListClient {
    public static void main(String[] args) {
        IntList list1 = new ArrayIntList();
        process(list1);
        IntList list2 = new LinkedIntList();
        process(list2);
    }

    public static void process(IntList list) {
        list.add(18);
        list.add(27);
        list.add(93);
        System.out.println(list);
        list.remove(1);
        System.out.println(list);
    }
}
```

ADTs as interfaces (11.1)

- **abstract data type (ADT):** A specification of a collection of data and the operations that can be performed on it.
  - Describes what a collection does, not how it does it.

- Java’s collection framework uses interfaces to describe ADTs:
  - Collection, Deque, List, Map, Queue, Set

- An ADT can be implemented in multiple ways by classes:
  - ArrayList and LinkedList implement List
  - HashSet and TreeSet implement Set
  - LinkedList, ArrayDeque, etc. implement Queue

  *They messed up on Stack; there’s no Stack interface, just a class.*

Using ADT interfaces

When using Java’s built-in collection classes:

- It is considered good practice to always declare collection variables using the corresponding ADT interface type:
  ```java
  List<String> list = new ArrayList<String>();
  ```

- Methods that accept a collection as a parameter should also declare the parameter using the ADT interface type:
  ```java
  public void stutter(List<String> list) {
      ...
  }
  ```

Why use ADTs?

- Why would we want more than one kind of list, queue, etc.?

- Answer: Each implementation is more efficient at certain tasks.
  - ArrayList is faster for adding/removing at the end;
  - LinkedList is faster for adding/removing at the front/middle.
  - HashSet can search a huge data set for a value in short time;
  - TreeSet is slower but keeps the set of data in a sorted order.

  - You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work.
Our list classes

- We implemented the following two list classes:
  - ArrayIntList
    - index: 0, 1, 2
    - value: 42, -3, 17
  - LinkedIntList
    - front → data: 42 → data: -3 → data: 17

- Problems:
  - We should be able to treat them the same way in client code.
  - Some of their methods are implemented the same way (redundancy).
  - Linked list carries around a clunky extra node class.
  - They can store only int elements, not any type of value.
  - It is inefficient to get or remove each element of a linked list.

ADT interfaces (11.1)

- abstract data type (ADT): A specification of a collection of data and the operations that can be performed on it.
  - Describes what a collection does, not how it does it.

- Java’s collection framework describes ADTs with interfaces:
  - Collection, Deque, List, Map, Queue, Set, SortedMap

- An ADT can be implemented in multiple ways by classes:
  - ArrayList and LinkedList implement List
  - HashSet and TreeSet implement Set
  - LinkedList, ArrayDeque, etc. implement Queue

- Exercise: Create an ADT interface for the two list classes.

An IntList interface (16.4)

// Represents a list of integers.
public interface IntList {
    public void add(int value);
    public void add(int index, int value);
    public int get(int index);
    public int indexOf(int value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, int value);
    public int size();
}

public class ArrayIntList implements IntList {
    ...
}

public class LinkedIntList implements IntList {
    ...
}

Our list classes

- We have implemented the following two list collection classes:
  - ArrayIntList
    - index: 0, 1, 2
    - value: 42, -3, 17
  - LinkedIntList
    - front → data: 42 → data: -3 → data: 17

- Problems:
  - We should be able to treat them the same way in client code.
  - Some methods are implemented the same way (redundancy).
  - Linked list carries around a clunky extra node class.
  - They can store only int elements, not any type of value.
  - It is inefficient to get or remove each element of a linked list.
Common code

- Notice that some of the methods are implemented the same way in both the array and linked list classes.
  - `add(value)`
  - `contains`
  - `isEmpty`

- Should we change our interface to a class? Why / why not?
  - How can we capture this common behavior?

Abstract class syntax

```java
// declaring an abstract class
class AbstractIntList implements IntList {  
  public void add(int value) { add(size(), value); }
  public boolean contains(int value) { return indexOf(value) >= 0; }
  public boolean isEmpty() { return size() == 0; }
}
```

- A class can be `abstract` even if it has no abstract methods
- You can create variables (but not objects) of the abstract type
- Exercise: Introduce an abstract class into the list hierarchy.

An abstract list class

```java
// Superclass with common code for a list of integers.
class AbstractIntList implements IntList {  
  public boolean contains(int value) { return indexOf(value) >= 0; }
  public boolean isEmpty() { return size() == 0; }
}
```

```java
public class ArrayIntList extends AbstractIntList { ...  
  public class LinkedIntList extends AbstractIntList { ... }
}
```

Our list classes

- We have implemented the following two list collection classes:
  - `ArrayIntList`
  - `LinkedIntList`

- Problems:
  - We should be able to treat them the same way in client code.
  - Some of their methods are implemented the same way (redundancy).
  - Linked list carries around a clunky extra node class.
  - They can store only `int` elements, not any type of value.
  - It is inefficient to get or remove each element of a linked list.
Linked list iterator

- The following code is particularly slow on linked lists:
  ```java
  List<Integer> list = new LinkedList<Integer>();
  ...
  for (int i = 0; i < list.size(); i++) {
    int value = list.get(i);
    if (value % 2 == 1) {
      list.remove(i);
    }
  }
  ```

  - Why?
  - What can we do to improve the runtime?

Iterators (11.1)

- **iterator**: An object that allows a client to traverse the elements of a collection, regardless of its implementation.
  - Remembers a position within a collection, and allows you to:
    * get the element at that position
    * advance to the next position
    * (possibly) remove or change the element at that position
  - Benefit: A common way to examine any collection's elements.

  ![Diagram of iterator](image)

- **hasNext()**: returns `true` if there are more elements to examine
- **next()**: returns the next element from the collection (throws a `NoSuchElementException` if there are none left to examine)
- **remove()**: removes from the collection the last value returned by `next()` (throws `IllegalStateException` if you have not called `next()` yet)

- every provided collection has an `iterator` method
  ```java
  Set<String> set = new HashSet<String>();
  ...
  Iterator<String> itr = set.iterator();
  ...
  ```

- Exercise: Write iterators for our array list and linked list.
  - You don't need to support the `remove` operation.

for-each loop and Iterable

- Java's collections can be iterated using a "for-each" loop:
  ```java
  List<String> list = new LinkedList<String>();
  ...
  for (String s : list) {
    System.out.println(s);
  }
  ```

  - Our collections do not work in this way.

- To fix this, your list must implement the `Iterable` interface.
  ```java
  public interface Iterable<E> {
    public Iterator<E> iterator();
  }
  ```
Our list classes

• We have implemented the following two list collection classes:
  - ArrayIntList
    - index: 0, 1, 2
    - value: 42, -3, 17
  - LinkedIntList
    - data
      - front: 42
      - next: -3
    - data
      - next: 17

• Problems:
  • We should be able to treat them the same way in client code.
  • Some of their methods are implemented the same way (redundancy).
  • Linked list carries around a clunky extra node class.
  • They can store only int elements, not any type of value.
  • It is inefficient to get or remove each element of a linked list.

Inner classes

• inner class: A class defined inside of another class.
  - can be created as static or non-static
  - we will focus on standard non-static ("nested") inner classes

• usefulness:
  • inner classes are hidden from other classes (encapsulated)
  • inner objects can access/modify the fields of the outer object

Inner class syntax

// outer (enclosing) class
public class name {
  ...
  // inner (nested) class
  private class name {
    ...
  }
}

• Only this file can see the inner class or make objects of it.
• Each inner object is associated with the outer object that created it, so it can access/modify that outer object’s methods/fields.
  • If necessary, can refer to outer object as OuterClassName.this

• Exercise: Convert the linked node into an inner class.

Our list classes

• We have implemented the following two list collection classes:
  - ArrayIntList
    - index: 0, 1, 2
    - value: 42, -3, 17
  - LinkedIntList
    - data
      - front: 42
      - next: -3
    - data
      - next: 17

• Problems:
  • We should be able to treat them the same way in client code.
  • Some of their methods are implemented the same way (redundancy).
  • Linked list carries around a clunky extra node class.
  • They can store only int elements, not any type of value.
  • It is inefficient to get or remove each element of a linked list.
Type Parameters (Generics)

ArrayList<Type> name = new ArrayList<Type>();

• Recall: When constructing a java.util.ArrayList, you specify the type of elements it will contain between < and >.
  – We say that the ArrayList class accepts a type parameter, or that it is a generic class.

ArrayList<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");

Generics and arrays (15.4)

public class Foo<T> { 
  private T myField;    // ok
  
  public void method1(T param) {
    myField = new T();    // error
    T[] a = new T[10];    // error
    myField = param;      // ok
    T[] a2 = (T[]) (new Object[10]);  // ok
  }
}

• You cannot create objects or arrays of a parameterized type.
• You can create variables of that type, accept them as parameters, return them, or create arrays by casting from Object[].

Implementing generics

// a parameterized (generic) class
public class name<Type> { 
  ...
}

– By putting the Type in < >, you are demanding that any client that constructs your object must supply a type parameter.
  • You can require multiple type parameters separated by commas.

• The rest of your class's code can refer to that type by name.

• Exercise: Convert our list classes to use generics.

Generics and inner classes

public class Foo<T> { 
  private class Inner<T> {} // incorrect
  private class Inner {}   // correct
}

– If an outer class declares a type parameter, inner classes can also use that type parameter.
– Inner class should NOT redefine the type parameter. (If you do, it will create a second type parameter with the same name.)
**Array list iterator**

```java
public class ArrayList<E> extends AbstractList<E> {
    private class ArrayIterator implements Iterator<E> {
        private int index; // current position in list
        public ArrayIterator() {
            index = 0;
        }
        public boolean hasNext() {
            return index < size();
        }
        public E next() {
            index++;
            return get(index - 1);
        }
        public void remove() {
            ArrayList.this.remove(index - 1);
            index--;
        }
    }
}
```

**Linked list iterator**

```java
public class LinkedList<E> extends AbstractList<E> {
    private class LinkedIterator implements Iterator<E> {
        private ListNode current; // current position in list
        public LinkedIterator() {
            current = front;
        }
        public boolean hasNext() {
            return current != null;
        }
        public E next() {
            E result = current.data;
            current = current.next;
            return result;
        }
        public void remove() {
            // not implemented for now
            throw new UnsupportedOperationException();
        }
    }
}
```

**Final List interface (15.3, 16.5)**

```java
// Represents a list of values.
public interface List<E> extends Iterable<E> {
    public void add(E value);
    public void add(int index, E value);
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public Iterator<E> iterator();
    public void remove(int index);
    public void set(int index, E value);
    public int size();
}
```