Lecture 9

Lists I

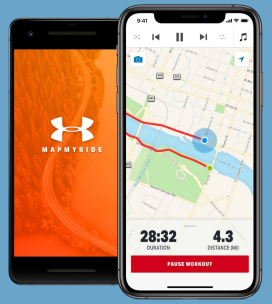
- Conversations at Software Company
- Interfaces and Inheritance
- `List` and Friends

Conversations at a Software Company

Conversations at a Software Company

mapmyride

We need a data structure for storing a bicycle ride. We want to add (time, location) pairs and compute speeds.



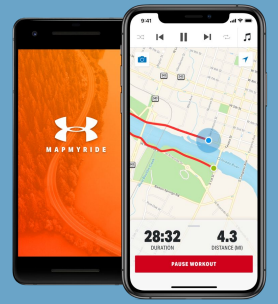
OK, my team will design an interface.

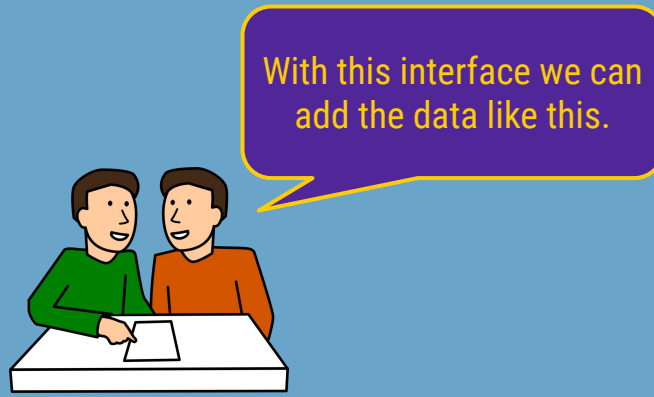
There could be GPS errors, so we should include `delete`.

Maybe we can inherit from the `Vector` class?

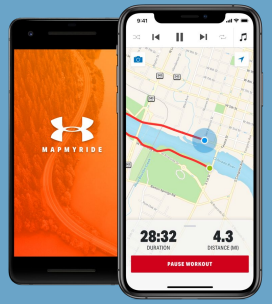
Conversations at a Software Company

mapmyride



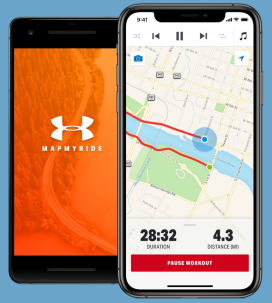


Conversations at a Software Company



Conversations at a Software Company

mapmyride



Can you improve the efficiency of `delete`?

We need a faster `delete`.

Let's switch to a doubly linked list.

Interfaces and Inheritance

and Abstract Classes

Interfaces

An *interface* provides a list of methods, but no specific implementation for these methods.

When a class implements an interface, it promises to implement these methods unless it is Abstract.

Java keywords: `interface` and `implements`.

Benefits of interfaces:

- Two classes can be used in the same way (i.e., if they implement the same interface).
- A class can be used in different ways (i.e., if it implements several interfaces).
- Similar modularity benefits as classes (i.e., know the use without the implementation, and the implementation can be changed, etc.)

```
public interface List<E> extends Structure<E>
{
    // Determine size of list.
    public int size();

    // Determine if list is empty.
    public boolean isEmpty();

    // Remove all elements of list.
    public void clear();

    // Add a value to the head of the
    public void addFirst(E value);

    // Add a value to tail of list.
    public void addLast(E value);

    // Fetch first element of list.
    public E getFirst();

    // Fetch last element of list.
    public E getLast();

    // Remove a value from first ele
    public E removeFirst();

    // Remove last value from list.
    public E removeLast();

    // Remove a value from list.  At
    // will be removed.
    public E remove(E value);

    // Add an object to tail of list.
    public void add(E value);

    // Removes value from tail of list.
    public E remove();

    // Retrieves value from tail of list.
    public E get();

    // Check to see if a value is in list.
    public boolean contains(E value);

    // Determine first location of a value in list.
    public int indexOf(E value);

    // Determine last location of a value in list.
    public int lastIndexOf(E value);

    // Get value at location i.
    public E get(int i);

    // Set value stored at location i to object o, returning old v
    public E set(int i, E o);

    // Insert value at location.
    public void add(int i, E o);

    // Remove and return value at location i.
    public E remove(int i);

    // Construct an iterator to traverse elements of list
    // from head to tail, in order.
    public Iterator<E> iterator();
```

The `List` interface from the `structure5` package. It extends the `Structure` interface.

Inheritance

When a class Y *extends* or *inherits* from another class X, it takes on all of its properties and methods.

- X is often called the *base class* or *parent class*.
- Y is often called the *derived class* or *child class*.

In addition, Y can add new properties and methods.
It can also change the implementation of base methods.

Java keyword: `extends`

Warning: You may start seeing inheritance everywhere!

Benefits of inheritance:

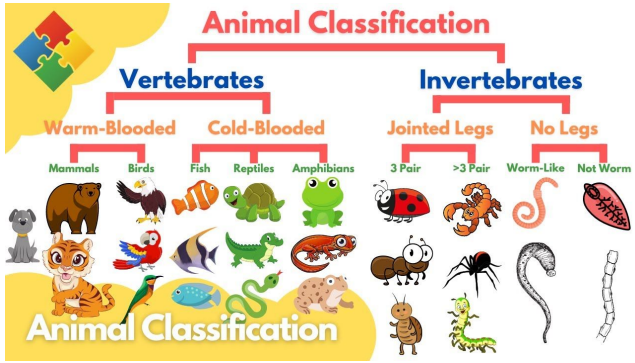
- Use different derived classes in the same way (i.e., "feed all animals" regardless of which type)
- Save time and avoid errors by implementing common behaviors once inside of base classes.

```
public class SinglyLinkedList<E> extends AbstractList<E>
{
    // The number of elements in list.
    protected int count;                    // list size

    // The head of the list.  A reference to a singly linked li
    protected Node<E> head; // ref. to first element

    // Construct an empty list.
    public SinglyLinkedList()
    {
        head = null;
        count = 0;
    }
```

`SinglyLinkedList` inherits from `AbstractList` in the `structure` package.



A frog is an amphibian, which is a cold-blooded vertebrate, which is an animal.

Abstract Classes

When designing object and class hierarchies, it is sometimes helpful to supply an abstract class.

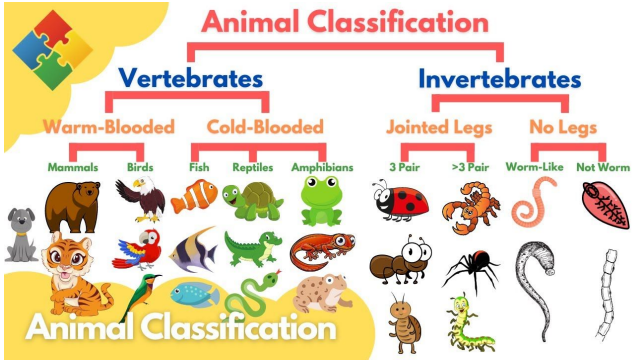
An abstract class cannot directly be “instantiated”. That is, objects of the class can’t be made with `new`.

A class is abstract if it has some unimplemented methods, or if it is specified to be abstract.

Java keyword: `abstract`

Benefits of abstract classes:

- Save time and avoid errors by implementing common behaviors once inside of abstract classes.



Vertebrates is a useful abstract classification. We cannot make a vertebrate.

```
public abstract class AbstractList<E>
    extends AbstractStructure<E> implements List<E>
{
    // Default constructor for AbstractLists
    public AbstractList()
    {
    }

    // Determine if list is empty.
    public boolean isEmpty()
    {
        return size() == 0;
    }
```

The `AbstractList` class is `abstract`. Non-abstract classes like `SinglyLinkedList` inherit from it.

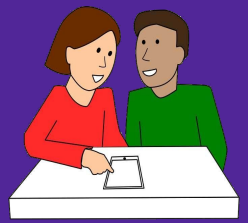
`List` and Friends

Activity: Understanding `List` and its Relationships in the `structure` Package

To properly understand and use the `structure` package, we need to be able to investigate the relationships between the various classes.

In this activity, you will focus on the relationships between the `List` class and the following:

`AbstractList`, `AbstractStructure`, `CircularList`, `DoublyLinkedList`, `DoublyLinkedNode`, `Node`, `SinglyLinkedList`, `Structure`, `Vector`



Look through code with a neighbor for 5 minutes.
Then we'll discuss the relationships as a group.

Related questions:

- How would you diagram this information?
- Are you surprised by `Vector`'s relationship to `List`?
- What is the benefit of `AbstractList`?
- How can you find which other classes use `List`?

```
-> pwd
/home/faculty/aaron/cs136/js/src/structure
-> grep Node *.java
BinarySearchTree.java:        BinaryTree newNode = new BinaryTree(value);
BinarySearchTree.java:            root = newNode;
BinarySearchTree.java:                insertLocation.setRight(newNode);
BinarySearchTree.java:                    predecessor(insertLocation).setRight(newNode);
BinarySearchTree.java:                    insertLocation.setLeft(newNode);
BinarySearchTree.java:    protected BinaryTree removeTop(BinaryTree topNode)
BinarySearchTree.java:        BinaryTree left  = topNode.left();
BinarySearchTree.java:        BinaryTree right = topNode.right();
BinarySearchTree.java:        topNode.setLeft(BinaryTree.EMPTY);
BinarySearchTree.java:        topNode.setRight(BinaryTree.EMPTY);
BinaryTree.java:     * Node must have a left child.  Relation between left child and node
BinaryTree.java:     * Node must have a right child.  Relation between right child and node
CircularListIterator.java:    protected Node tail;
CircularListIterator.java:    protected Node current;
CircularListIterator.java:    public CircularListIterator(Node t)
CircularList.java:    protected Node tail;
CircularList.java:        Node temp = new Node(value);
CircularList.java:        Node temp = tail.next(); // ie. head of list
CircularList.java:        Node finger = tail;
CircularList.java:        Node temp = tail;
CircularList.java:        Node finger;
CircularList.java:        Node finger = tail.next();
CircularList.java:        Node previous = tail;
CircularList.java:        Node finger = tail.next();
CircularList.java:        Node finger = tail.next();
CircularList.java:            Node previous = tail;
CircularList.java:            Node next = tail.next();
CircularList.java:            Node current = new Node(o,next);
CircularList.java:        Node previous = tail;
CircularList.java:        Node finger = tail.next(); // ie. head
CircularList.java:        Node finger = tail.next();
CircularList.java:        Node finger = tail.next();
DoublyLinkedListIterator.java:    protected DoublyLinkedNode head;
```

`grep` can be used on the command-line to help you search for patterns in files.
For example, running `grep Node *.java` from your `~/cs136/js/src/structure` folder reveals that `Node` is used in a dozen files in the `structure` package.

```java
public abstract class AbstractStructure implements Structure
{
    /**
     * The default constructor.  Initializes any internal variables.
     *
     * @post initializes internal variables
     */
    public AbstractStructure()
    {
    }

    /**
     * Determine if there are elements within the structure.
     *
     * @post return true iff the structure is empty
     * @return true if the structure is empty; false otherwise
     */
    public boolean isEmpty()
    {
        return size() == 0;
    }
```

```java
public abstract class AbstractList<E>
    extends AbstractStructure<E> implements List<E>
{
    /**
     * Default constructor for AbstractLists
     * @post does nothing
     */
    public AbstractList()
    {
    }

    /**
     * Determine if list is empty.
     *
     * @post returns true iff list has no elements
     *
     * @return True if list has no elements.
     */
    public boolean isEmpty()
    {
        return size() == 0;
    }
```

Note: During our discussion, it was pointed out that the implementation of `isEmpty()` in `AbstractList` is redundant, since the class extends `AbstractStructure`, which implements `isEmpty()` in the same way. Nice observation!

Extra Time?

```java
/**
 * Add a value to tail of list.
 *
 * @post adds value to end of list
 *
 * @param value The value to be added to tail of list.
 */
public void addLast(E value)
{
    // location for new value
    Node<E> temp = new Node<E>(value,null);
    if (head != null)
    {
        // pointer to possible tail
        Node<E> finger = head;
        while (finger.next() != null)
        {
            finger = finger.next();
        }
        finger.setNext(temp);
    } else head = temp;
    count++;
}
```

```java
/**
 * Remove last value from list.
 *
 * @pre list is not empty
 * @post removes last value from list
 *
 * @return The value actually removed.
 */
public E removeLast()
{
    Node<E> finger = head;
    Node<E> previous = null;
    Assert.pre(head != null,"List is not empty.");
    while (finger.next() != null) // find end of list
    {
        previous = finger;
        finger = finger.next();
    }
    // finger is null, or points to end of list
    if (previous == null)
    {
        // has exactly one element
        head = null;
    }
    else
    {
        // pointer to last element is reset
        previous.setNext(null);
    }
    count--;
    return finger.value();
}
```

Next class we'll return to singly linked lists.

- Try to understand how its `addLast` and `removeLast` methods are implemented.

```java
/**
 * Add a value to tail of list.
 *
 * @pre value is not null
 * @post adds new value to tail of list
 *
 * @param value value to be added.
 */
public void addLast(E value)
{
    // construct new element
    tail = new DoublyLinkedNode<E>(value, null, tail);
    // fix up head
    if (head == null) head = tail;
    count++;
}
```

```java
/**
 * Remove a value from tail of list.
 *
 * @pre list is not empty
 * @post removes value from tail of list
 *
 * @return value removed from list.
 */
public E removeLast()
{
    Assert.pre(!isEmpty(),"List is not empty.");
    DoublyLinkedNode<E> temp = tail;
    tail = tail.previous();
    if (tail == null) {
        head = null;
    } else {
        tail.setNext(null);
    }
    count--;
    return temp.value();
}
```

Above are the same methods implemented in the `DoubleLinkedList` class.
Why are the implementations in a doubly linked list simpler?