What should a list be able to do?

Write down as many useful list operations as you can think of.

(Your response)
Look at all the things we’ve done!

Concepts / skills:

- functional programming
- imperative programming
- code analysis
  - summations • recurrence relations • Big O
- sequential data structures
  - linked lists • arrays
- higher-order functions
  - map • fold • filter • lambda • functions as data
- object-oriented programming
  - classes • object • methods • fields • this • static
- memory model
  - “box and arrow” diagrams

Professional practices:

- write tests (early)!
- write documentation (early)!
- inline comments • high-level comments • JavaDoc
- reuse as much code as possible
- use a style guide
- always use this • call existing methods if possible
Today: data structures in Java

We’ll revisit:

- functional programming
- imperative programming
- code analysis
- summations • recurrence relations • Big O
- sequential data structures
  - linked lists • arrays
- higher-order functions
  - map • fold • filter • lambda • functions as data
- object-oriented programming
  - classes • object • methods • fields • this • static
- memory model
  - “box and arrow” diagrams

We’ll introduce:

- how to make new data structures
- Java interface
- type parameters
- private, inner classes
- and more!
Designing and implementing a new data structure

- **Interface**
  Describes: *what* this data structure can do

- **Implementation: encoding**
  Describes: *how* the structure is stored, using existing data structures

- **Implementation: operations**
  Defines: *how* the structure provides its interface via algorithms over the encoding

It should be possible to replace the implementation without modifying the interface.
Lists
Designing and implementing a new data structure

• Interface
  Describes: **what** this data structure can do

• Implementation: encoding
  Describes: **how** the structure is stored, using existing data structures

• Implementation: operations
  Defines: **how** the structure provides its interface via algorithms over the encoding

It should be possible to replace the implementation without modifying the interface.
Designing and implementing a new data structure

- **Interface**
  Describes: what this data structure can do

- **Implementation: encoding**
  Describes: how the structure is stored, using existing data structures

- **Implementation: operations**
  Defines: how the structure provides its interface via algorithms over the encoding

It should be possible to replace the implementation without modifying the interface.
Exercise: two list “implementations”

Implement two, constant-time versions of
- length
- addToFront

One version for each of the following encodings:

**Linked list**
with the following, constant-time operations
- cons
- first
- rest

**Array**
with the following, constant-time operations
- length
- read a value at any index
- write a value at any index
Big idea

We can trade space for time.
Designing and implementing a new data structure in Java

• **Interface**
  • The interface is the public methods & fields.
  • These operations can be specified in a Java interface, if there might be multiple ways to implement the same interface.
  • An operation’s cost is sometimes an unofficial part of the interface.

• **Implementation: encoding**
  • Typically: private fields (whose types are smaller data structures).

• **Implementation: operations**
  • The bodies of the public methods.
  • Also, often private, helper methods.
**Interface List&lt;E&gt;**

Type Parameters:

E - the type of elements in this list

All Superinterfaces:

Collection&lt;E&gt;, Iterable&lt;E&gt;

All Known Implementing Classes:

AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

public interface List&lt;E&gt;
extends Collection&lt;E&gt;

An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

The List interface places additional stipulations, beyond those specified in the Collection interface, on the contracts of the iterator, add, remove, equals, and hashCode methods. Declarations for other inherited methods are also included here for convenience.

The List interface provides four methods for positional (indexed) access to list elements. Lists (like Java arrays) are zero based. Note that these operations may execute in time proportional to the index value for some implementations (the LinkedList class, for example). Thus, iterating over the elements in a list is typically preferable to indexing through it if the caller does not know the implementation.

The List interface provides a special iterator, called a ListIterator, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the Iterator interface provides. A method is provided to obtain a list iterator that starts at a specified position in the list.

The List interface provides two methods to search for a specified object. From a performance standpoint, these methods should be used with caution. In many implementations they will perform costly linear searches.

The List interface provides two methods to efficiently insert and remove multiple elements at an arbitrary point in the list.

Note: While it is permissible for lists to contain themselves as elements, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a list.

Some list implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically NullPointerException or ClassCastException. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the list may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

Collection, Set, ArrayList, LinkedList, Vector, Arrays.asList(Object[]), Collections.nCopies(int, Object), Collections.EMPTY_LIST, AbstractList, AbstractSequentialList

---

**Method Summary**

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>add(E e)</td>
</tr>
<tr>
<td></td>
<td>Appends the specified element to the end of this list (optional operation).</td>
</tr>
<tr>
<td>void</td>
<td>add(int index, E element)</td>
</tr>
<tr>
<td></td>
<td>Inserts the specified element at the specified position in this list (optional operation).</td>
</tr>
</tbody>
</table>
import java.util.List;
import java.util.LinkedList;
import java.util.ArrayList;

public class ListExamples {

    public static void addRange(List<Integer> values, int start, int end) {
        for (int i = start; i < end; i++) {
            values.add(i);
        }
    }

    public static <T> void printValues(List<T> values) {
        for (T value : values) {
            System.out.println(value);
        }
    }

    public static void main(String[] args) {
        List<Integer> linkedValues = new LinkedList<Integer>();
        List<Integer> arrayValues = new ArrayList<Integer>();
        addRange(linkedValues, 0, 10);
        addRange(arrayValues, 100, 105);
        System.out.println("Printing linked values");
        printValues(linkedValues);
        System.out.println("Printing array values");
        printValues(arrayValues);
    }
}
Maps
(aka dictionaries, in Java)
Designing and implementing a new data structure

• Interface
  Describes: what this data structure can do

• Implementation: encoding
  Describes: how the structure is stored, using existing data structures

• Implementation: operations
  Defines: how the structure provides its interface via algorithms over the encoding

It should be possible to replace the implementation without modifying the interface.
public interface Map<K, V>

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.

The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no argument) constructor which creates an empty map, and a constructor with a single argument of type Map, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw UnsupportedOperationException if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an UnsupportedOperationException if the invocation would have no effect on the map. For example, invoking the putAll(Map) method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.

Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically NullPointerException or ClassCastException. Attempting to query the presence of an ineligible key or value may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible key or value whose completion would not result in the insertion of an ineligible element into the map may throw an exception or it may succeed, at "optional" in the specification for this interface.

Many methods in Collections Framework interfaces are defined in terms of the equals method. For example, the specification for the containsKey(Object key) method says: "returns true if and only if this map contains a mapping for a key k such that (key.equals(k))." This specification should not be construed to imply that invoking Map.containsKey with a non-null argument key will cause key.equals(k) to be invoked for any key k. Implementations are free to implement optimizations whereby the equals invocation is avoided, for example, by first comparing the hash codes of the two keys. (The Object.equals() specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying Object methods wherever the implementor deems it appropriate.

Some map operations which perform recursive traversal of the map may fall with an exception for self-referential instances where the map directly or indirectly contains itself. This includes the clone(), equals(), hashCode() and toString() methods. Implementations may optionally handle the self-referential scenario, however most current implementations do not do so.

This interface is a member of the Java Collections Framework.

Since: 1.2

See Also: HashMap, TreeMap, Hashtable, SortedMap, Collection, Set
import java.util.Map;
import java.util.HashMap;
import java.util.TreeMap;

public class MapExamples {

    /**
     * Prints all the keys in a map
     * @param values
     */
    public static <Key, Value> void printKeys(Map<Key, Value> values) {
        for (Key key : values.keySet()) {
            System.out.println(key);
        }
    }

    public static void main(String[] args) {
        Map<Integer, String> hashValues = new HashMap<Integer, String>();
        hashValues.put(500, "five hundred");
        hashValues.put(1, "one");
        hashValues.put(2, "two");
        hashValues.put(42, "forty two");

        System.out.println("Printing hashValues' keys");
        printKeys(hashValues);

        Map<Integer, String> treeValues = new TreeMap<Integer, String>();
        treeValues.putAll(hashValues);

        System.out.println("Printing treeValues' keys");
        printKeys(treeValues);
    }
}
Good programming practices

If two objects are .equals, their hashCode methods should return the same value.

Therefore, if we define an equals method, we should define a hashCode method.

Often, we let Eclipse define both these methods for us.
Graphs
Designing and implementing a new data structure

• **Interface**
  Describes: *what* this data structure can do

• **Implementation: encoding**
  Describes: *how* the structure is stored, using existing data structures

• **Implementation: operations**
  Defines: *how* the structure provides its interface via algorithms over the encoding

It should be possible to replace the implementation without modifying the interface.
Designing and implementing a new data structure

- **Interface**
  Describes: **what** this data structure can do

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Edges</th>
<th>Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• construct</td>
<td>• construct</td>
<td>• construct empty graph</td>
</tr>
<tr>
<td>• set contents</td>
<td>• set source</td>
<td>• add / remove edge</td>
</tr>
<tr>
<td>• get contents</td>
<td>• set destination</td>
<td>• add / remove node</td>
</tr>
<tr>
<td></td>
<td>• set weight</td>
<td>• test if empty</td>
</tr>
<tr>
<td></td>
<td>• get source</td>
<td>• get all edges / nodes</td>
</tr>
<tr>
<td></td>
<td>• get destination</td>
<td>• search for edge / node</td>
</tr>
<tr>
<td></td>
<td>• get weight</td>
<td>• is node b reachable from a?</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>and more!</strong></td>
</tr>
</tbody>
</table>
Designing and implementing a new data structure

• **Interface**
  Describes: *what* this data structure can do

• **Implementation: encoding**
  Describes: *how* the structure is stored, using existing data structures

• **Implementation: operations**
  Defines: *how* the structure provides its interface via algorithms over the encoding

It should be possible to replace the implementation without modifying the interface.
How might we encode a graph?

What information should we store, and what existing data structures should we use to store it?
An edge stores source, weight, and destination.

A graph stores a list of edges.
Adjacency list

An adjacency stores weight and destination.
A graph stores a map from nodes to adjacencies.
A graph stores an N x N table.
where N is the number of nodes, rows are sources, columns are destinations

A cell in the table stores the weight of row → column edge.
if there is no edge from row to column, the cell’s weight is ∞
Draw the graph(s)
i.e., the circles and arrows

**edge list**

- C -> D (32)
- E -> A (7)
- A -> C (12)
- D -> B (60)
- B -> A (10)
- C -> B (20)
- D -> C (32)
- E -> D (7)

**adjacency list**

- A -> C
- B -> A
- C -> B, D
- D -> C, E
- E -> A

**adjacency matrix**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>∞</td>
<td>12</td>
<td>60</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td>20</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>∞</td>
<td>32</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>E</td>
<td>7</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>
Represent the graph in three different ways

element list

adjacency list

adjacency matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>∞</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td>0</td>
<td>2</td>
<td>∞</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>
Graphs in Java
Designing and implementing a new data structure

• Interface
  Describes: what this data structure can do

• Implementation: encoding
  Describes: how the structure is stored, using existing data structures

• Implementation: operations
  Defines: how the structure provides its interface via algorithms over the encoding

It should be possible to replace the implementation without modifying the interface.

Our Java graphs will be weighted, directed, imperative and object-oriented.
How does language affect implementation?

A functional data structure can’t be modified
Instead, any modification results in a new data structure

An imperative data structure can be modified
We can change the values and the structure of one piece (via references)

An object-oriented implementation helps us be abstract
Interface vs implementation • public vs private • type vs class
Java interface
What can graphs do?

```java
import java.util.Set;

public interface DirectedWeightedGraph<NodeDataType, EdgeDataType> {

    // add, remove, and access EDGES
    public EdgeDataType getEdge(NodeDataType srcNode, NodeDataType dstNode);
    public boolean adjacent(NodeDataType srcNode, NodeDataType dstNode);
    public boolean addEdge(NodeDataType srcNode, NodeDataType dstNode, EdgeDataType edge);
    public EdgeDataType removeEdge(NodeDataType srcNode, NodeDataType dstNode);

    // access, add, and remove NODES
    public boolean containsNode(NodeDataType nodeData);
    public Set<NodeDataType> getNodes();
    public Set<NodeDataType> neighbors(NodeDataType srcNode);  // adjacent nodes
    public boolean addNode(NodeDataType nodeData);
    public boolean removeNode(NodeDataType nodeData);

}
```
Java implementations (classes)

How do graphs do it?

```java
public class EdgeList<NodeDataType, EdgeDataType> implements DirectedWeightedGraph<NodeDataType, EdgeDataType> {
   ...
}

public class AdjacencyList<NodeDataType, EdgeDataType> implements DirectedWeightedGraph<NodeDataType, EdgeDataType> {
   ...
}

public class AdjacencyMatrix<NodeDataType, EdgeDataType> implements DirectedWeightedGraph<NodeDataType, EdgeDataType> {
   ...
}
```
Edge list in Java

An edge stores source, information, and destination.
A graph stores a list of edges.

```java
public class EdgeList<NodeDataType, EdgeDataType>
    implements DirectedWeightedGraph<NodeDataType, EdgeDataType> {

    /** private, inner class to represent an edge */
    private class Edge {
        NodeDataType source;
        NodeDataType destination;
        EdgeDataType data;
        ...
    }

    /** the edge list */
    private Collection<Edge> theGraph = new ArrayList<Edge>();

    /** keeps track of all nodes (including disconnected ones) */
    private Set<NodeDataType> allNodes = new HashSet<NodeDataType>();

    ...
}
```
Adjacency list in Java

An adjacency stores edge data and destination. A graph stores a map from nodes to adjacencies.

```java
public class AdjacencyList<NodeDataType, EdgeDataType>
    implements DirectedWeightedGraph<NodeDataType, EdgeDataType> {

    /** private, inner class to represent an edge */
    private class Edge {
        NodeDataType source;
        NodeDataType destination;
        EdgeDataType data;
    }

    /** the edge list */
    private Collection<Edge> theGraph = new ArrayList<Edge>();

    /** keeps track of all nodes (including disconnected ones) */
    private Set<NodeDataType> allNodes = new HashSet<NodeDataType>();

    /** private, inner class to represent an adjacency */
    private class Adjacency {
        NodeDataType destination;
        EdgeDataType data;
    }

    /** the adjacency list */
    Map<NodeDataType, Collection<Adjacency>> theGraph =
        new HashMap<NodeDataType, Collection<Adjacency>>();

    . . .

    . . .
}
```
Adjacency matrix in Java

A graph stores an N x N table. where N is the number of nodes, rows are sources, columns are destinations. A cell in the table stores the weight of row → column edge. if there is no edge from row to column, the cell’s weight is ∞.

```java
public class AdjacencyMatrix<NodeDataType, EdgeDataType>
    implements DirectedWeightedGraph<NodeDataType, EdgeDataType> {

    /** the adjacency matrix */
    private EdgeDataType[][] theGraph = new EdgeDataType[0][0];

    // ...
Adjacency matrix in Java

A graph stores an N x N table.
where N is the number of nodes, rows are sources, columns are destinations

A cell in the table stores the weight of row → column edge.
if there is no edge from row to column, the cell’s weight is ∞

```java
public class AdjacencyMatrix<NodeDataType, EdgeDataType>
    implements DirectedWeightedGraph<NodeDataType, EdgeDataType> {

    /** the adjacency matrix */
    private Object[][] theGraph = new Object[0][0];
```

✔
A graph stores an N x N table.
where N is the number of nodes, rows are sources, columns are destinations

A cell in the table stores the weight of row → column edge.
if there is no edge from row to column, the cell’s weight is ∞

```java
public class AdjacencyMatrix<NodeDataType, EdgeDataType>
        implements DirectedWeightedGraph<NodeDataType, EdgeDataType> {

    /** the adjacency matrix */
    private Object[][] theGraph = new Object[0][0];

    /** allows us to convert a node's data to an array index */
    private Map<NodeDataType, Integer> nodeIndexLookup =
            new HashMap<NodeDataType, Integer>();

    // ...