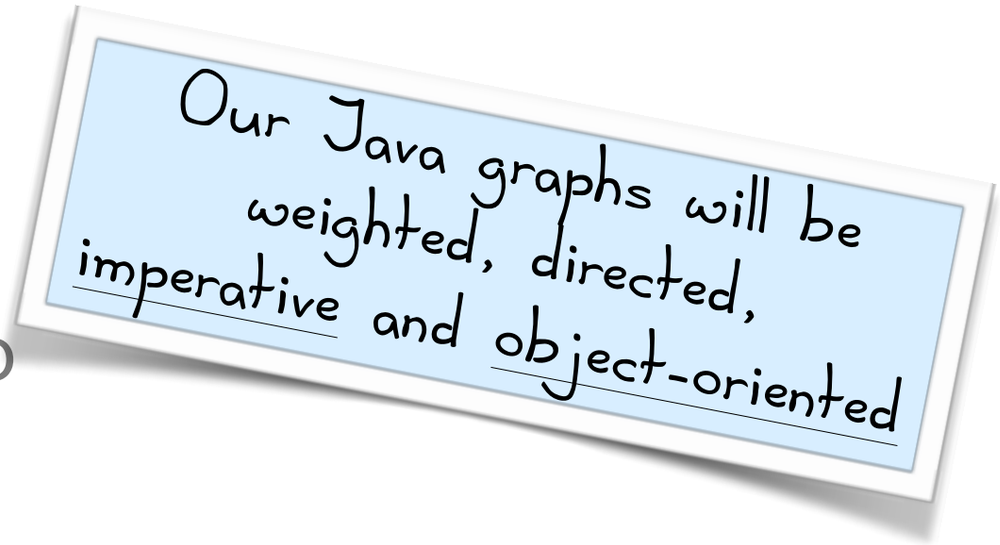
What are all the data structures we've seen so far, this semester?

Name Th. 12/6

(your response)

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.

What's the interface of a graph?

What operations should a graph support?

Also, consider the interfaces for a node and an edge.

Designing and implementing a new data structure

- Interface

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

Nodes

- construct
- set contents
- get contents

Edges

- construct
- set source
- set destination
- set weight
- get source
- get destination
- get weight

Graphs

- construct empty graph
- add / remove edge
- add / remove node
- test if empty
- get all edges / nodes
- search for edge / node
- is node b reachable from a?

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.

How might we encode a graph?

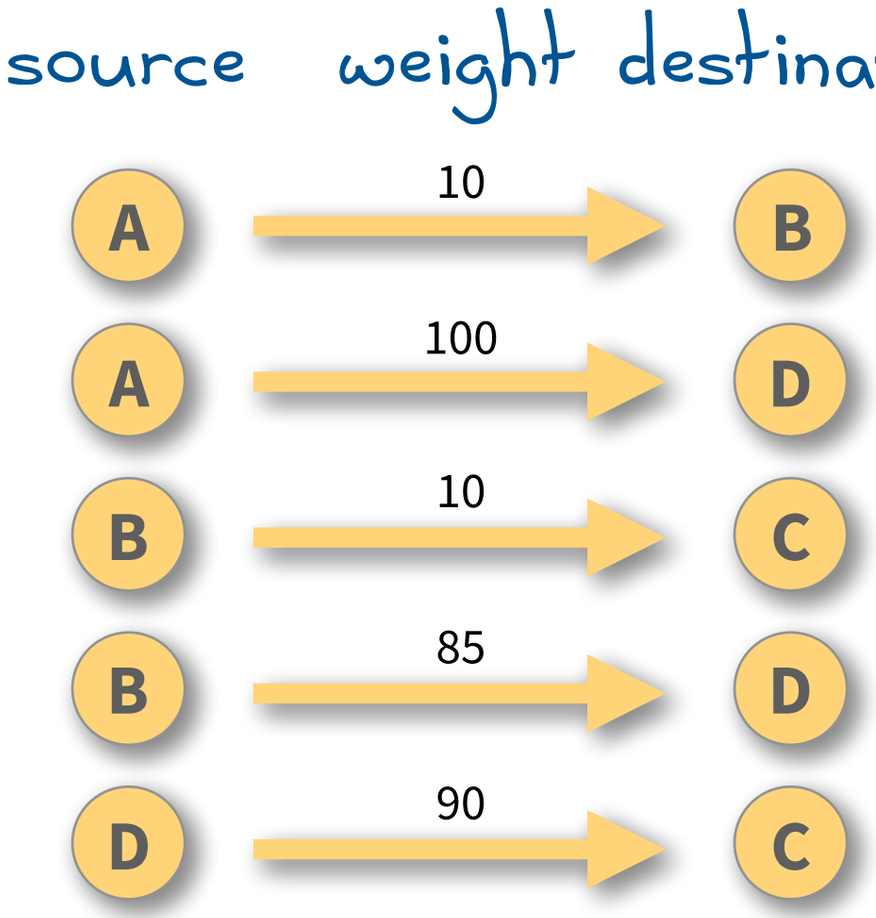
What information should we store, and what existing data structures should we use to store it?

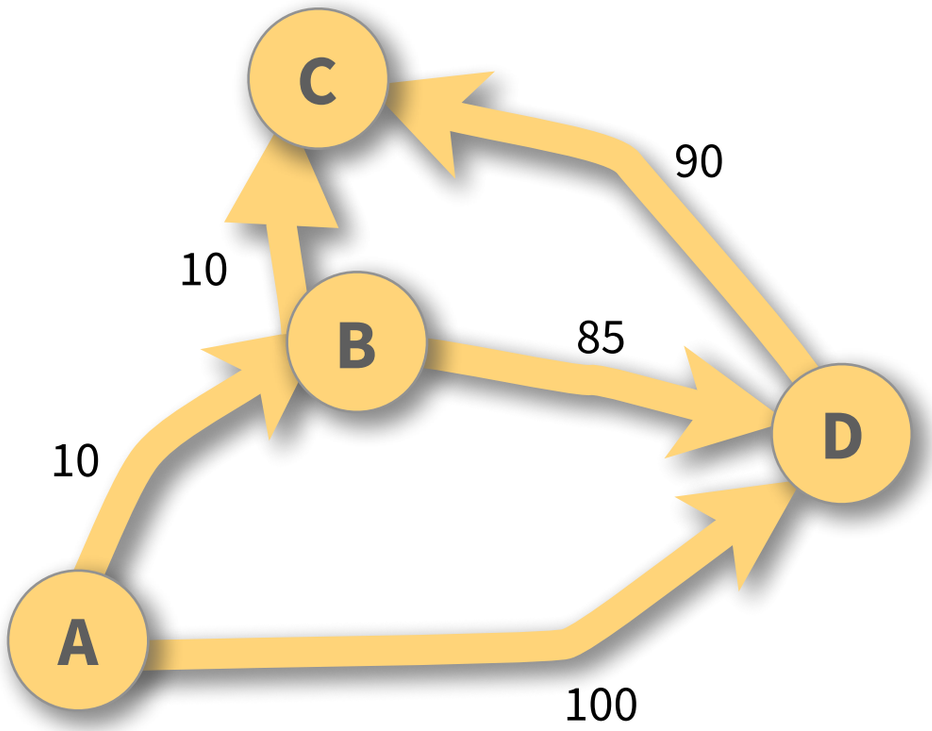
Edge list

← important vocabulary!

An edge stores source, weight, and destination.

A graph stores a list of edges.



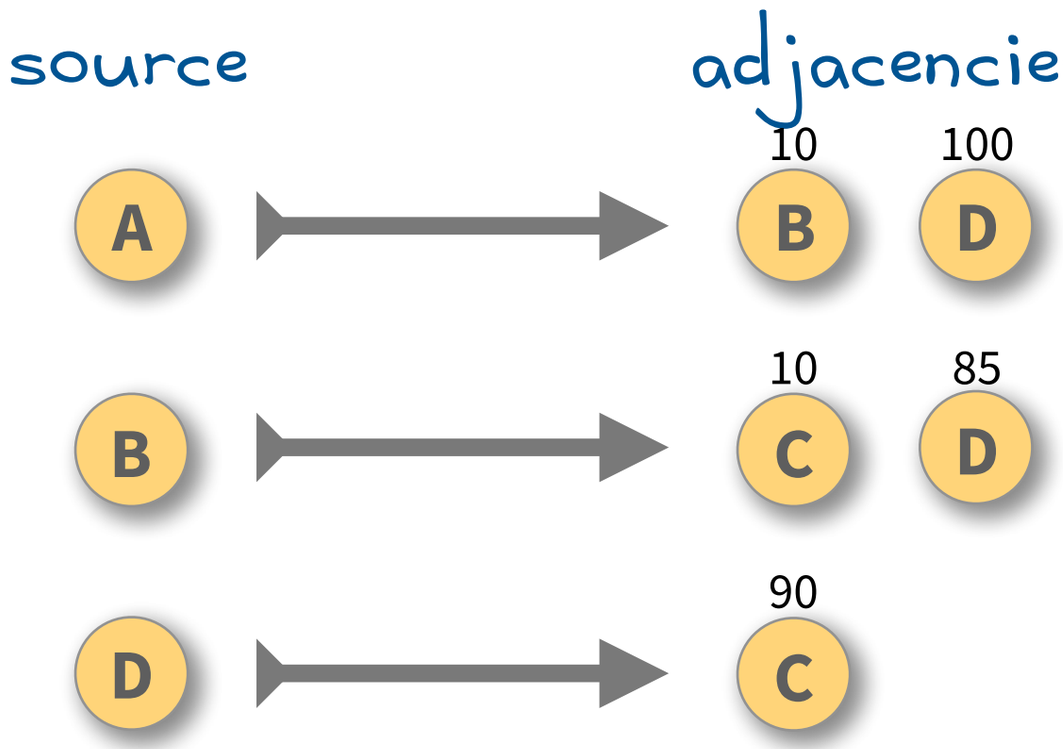


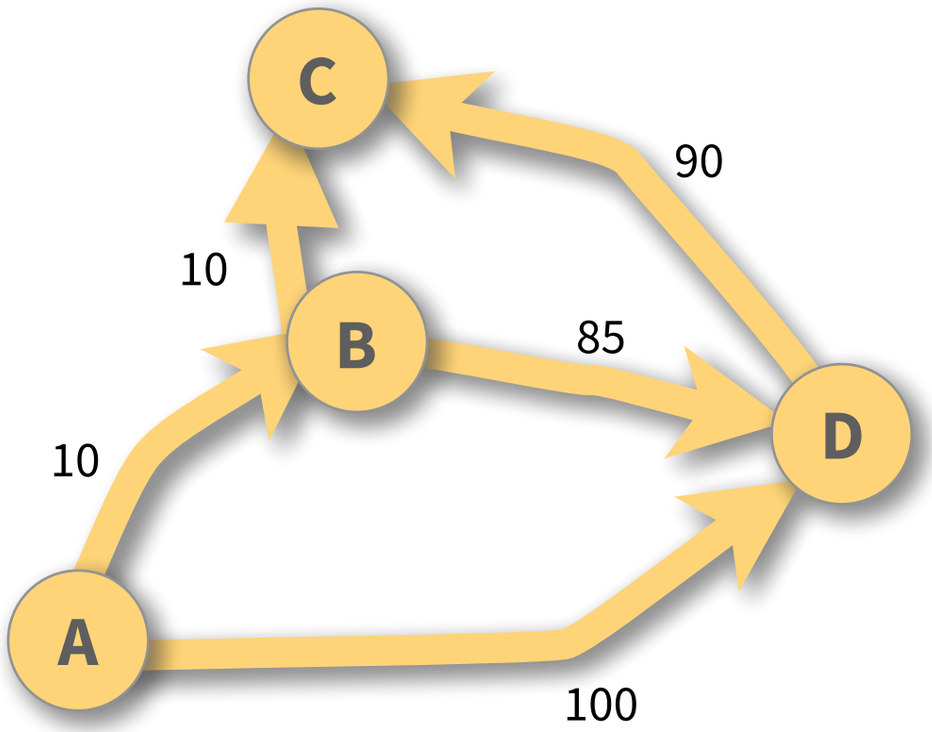
Adjacency list

← important vocabulary!

An adjacency stores weight and destination.

A graph stores a map from nodes to adjacencies.





Adjacency matrix

← important vocabulary!

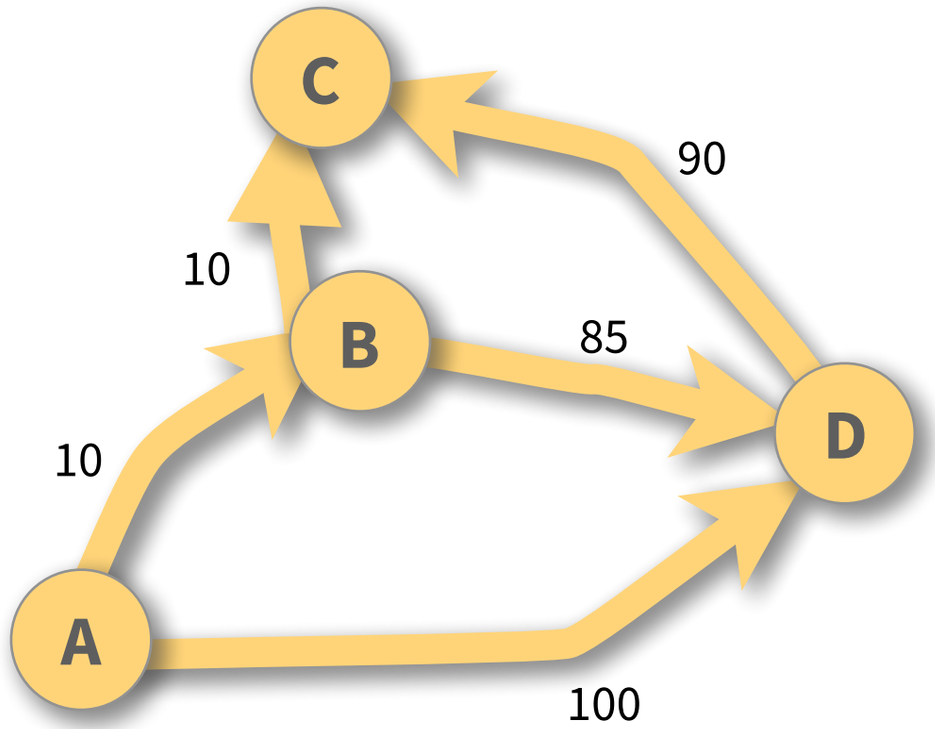
A graph stores an N x N table.

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 ∞

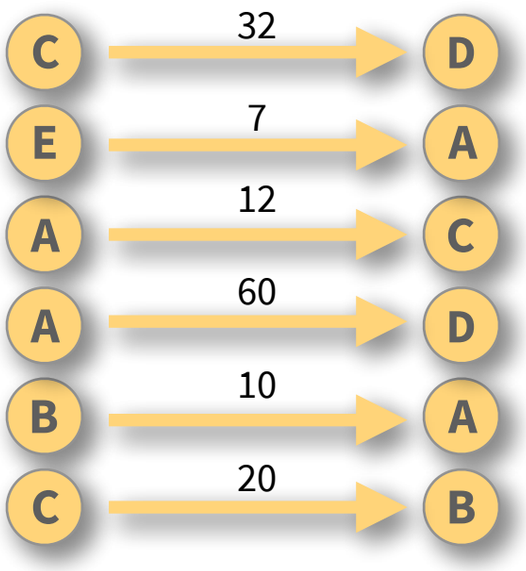
	A	B	C	D
A	0	10	∞	100
B	∞	0	10	85
C	∞	∞	0	∞
D	∞	∞	90	0



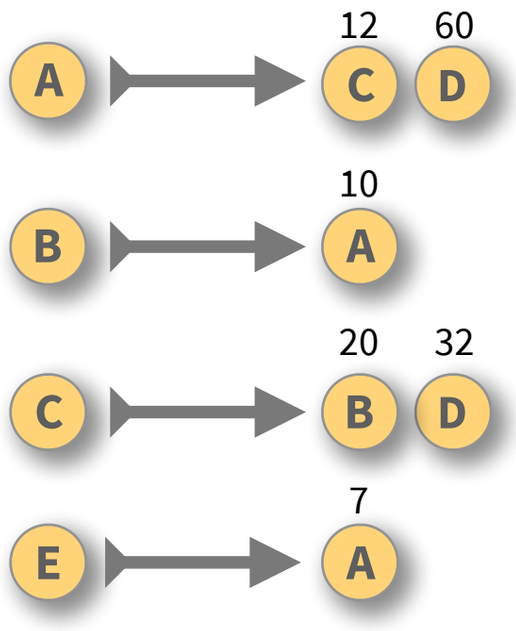
Draw the graph(s)

i.e., the circles and arrows

edge list

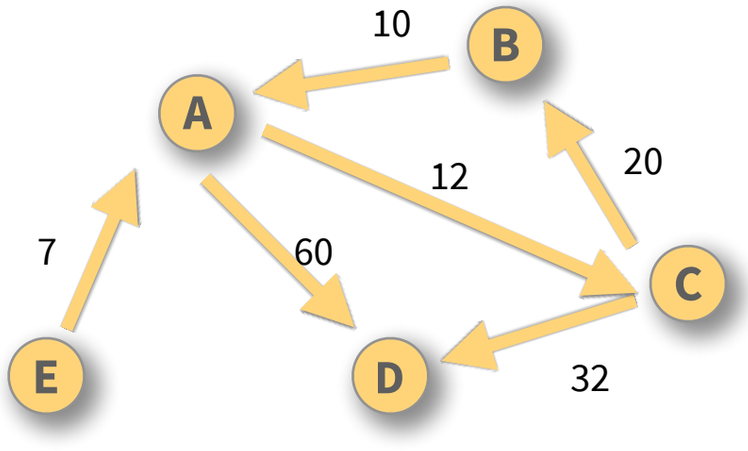


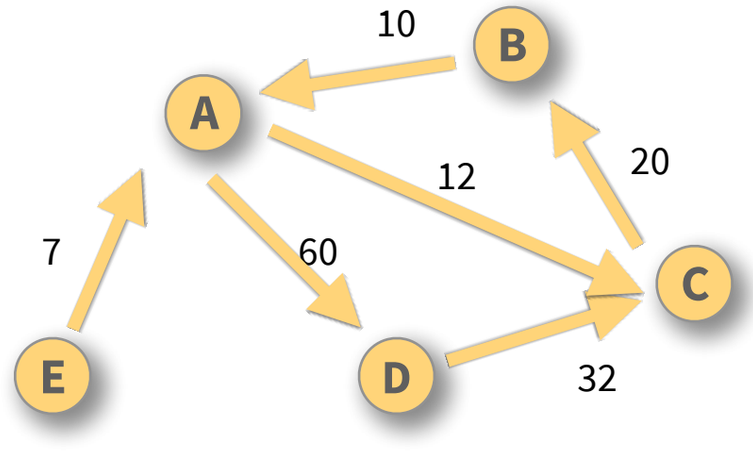
adjacency list



adjacency matrix

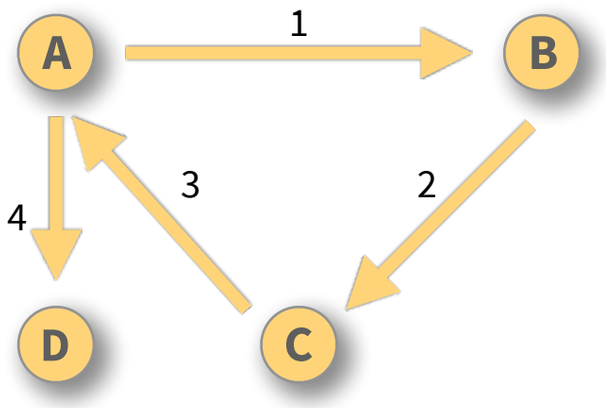
	A	B	C	D	E
A	0	∞	12	60	∞
B	10	0	∞	∞	∞
C	∞	20	0	∞	∞
D	∞	∞	32	0	∞
E	7	∞	∞	∞	0



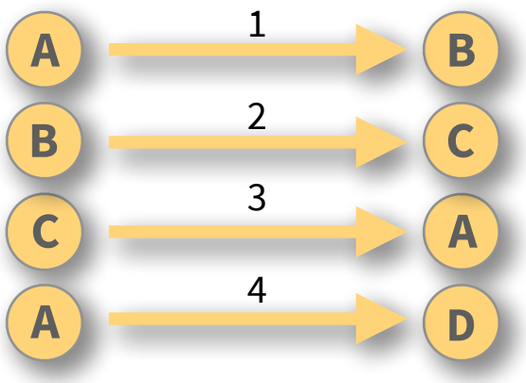


Represent the graph

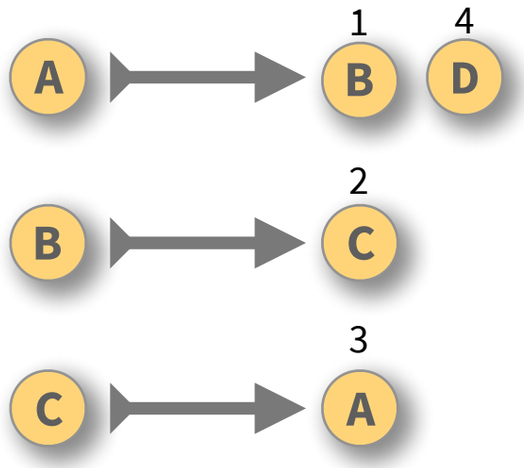
in three different ways



edge list



adjacency list



adjacency matrix

	A	B	C	D
A	0	1	∞	4
B	∞	0	2	∞
C	3	∞	0	∞
D	∞	∞	∞	0

Java Lists & Generics

```java
import java.util.List;
import java.util.LinkedList;
import java.util.ArrayList;

public class ListExamples {

    public static void main(String[] args) {
        List linkedValues = new LinkedList();
        List arrayValues = new ArrayList();

    }
}
```

```java
import java.util.List;
import java.util.LinkedList;
import java.util.ArrayList;

public class ListExamples {
```

type parameter

```java
    public static void main(String[] args) {
        List<int> linkedValues = new LinkedList<int>();
        List<int> arrayValues = new ArrayList<int>();
```

type parameters must inherit from Object

```java
    }
}
```

```java
import java.util.List;
import java.util.LinkedList;
import java.util.ArrayList;

public class ListExamples {

    public static void main(String[] args) {
        List<Integer> linkedValues = new LinkedList<Integer>();
        List<Integer> arrayValues = new ArrayList<Integer>();









    }
}
```

compact1, compact2, compact3
java.util

Interface List<E>

Type Parameters:
`E - the type of elements in this list`

All Superinterfaces:
`Collection<E>, Iterable<E>`

All Known Implementing Classes:
`AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector`

```
public interface List<E>
extends Collection<E>
```

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

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**add**(int index, E element) Inserts the specified element at the specified position in this list (optional operation).

Maps

(aka dictionaries, in Java)

```java
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);
    }
}
```

compact1, compact2, compact3
java.util

Interface Map<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

All Known Subinterfaces:

`Bindings`, `ConcurrentMap<K,V>`, `ConcurrentNavigableMap<K,V>`, `LogicalMessageContext`, `MessageContext`, `NavigableMap<K,V>`, `SOAPMessageContext`, `SortedMap<K,V>`

All Known Implementing Classes:

`AbstractMap`, `Attributes`, `AuthProvider`, `ConcurrentHashMap`, `ConcurrentSkipListMap`, `EnumMap`, `HashMap`, `Hashtable`, `IdentityHashMap`, `LinkedHashMap`, `PrinterStateReasons`, `Properties`, `Provider`, `RenderingHints`, `SimpleBindings`, `TabularDataSupport`, `TreeMap`, `UIDefaults`, `WeakHashMap`

`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 arguments) 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 the option of the implementation. Such exceptions are marked as "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==null ? k==null : 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.hashCode()` 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 fail 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`

Nested Class Summary

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 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?

```
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?

```
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.

```
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.

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


    /** 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 ∞

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

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

    . . .
}
```

Cannot create a generic array of EdgeDataType

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 ∞

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

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

```
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>();


    . . .
}
```