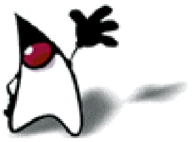
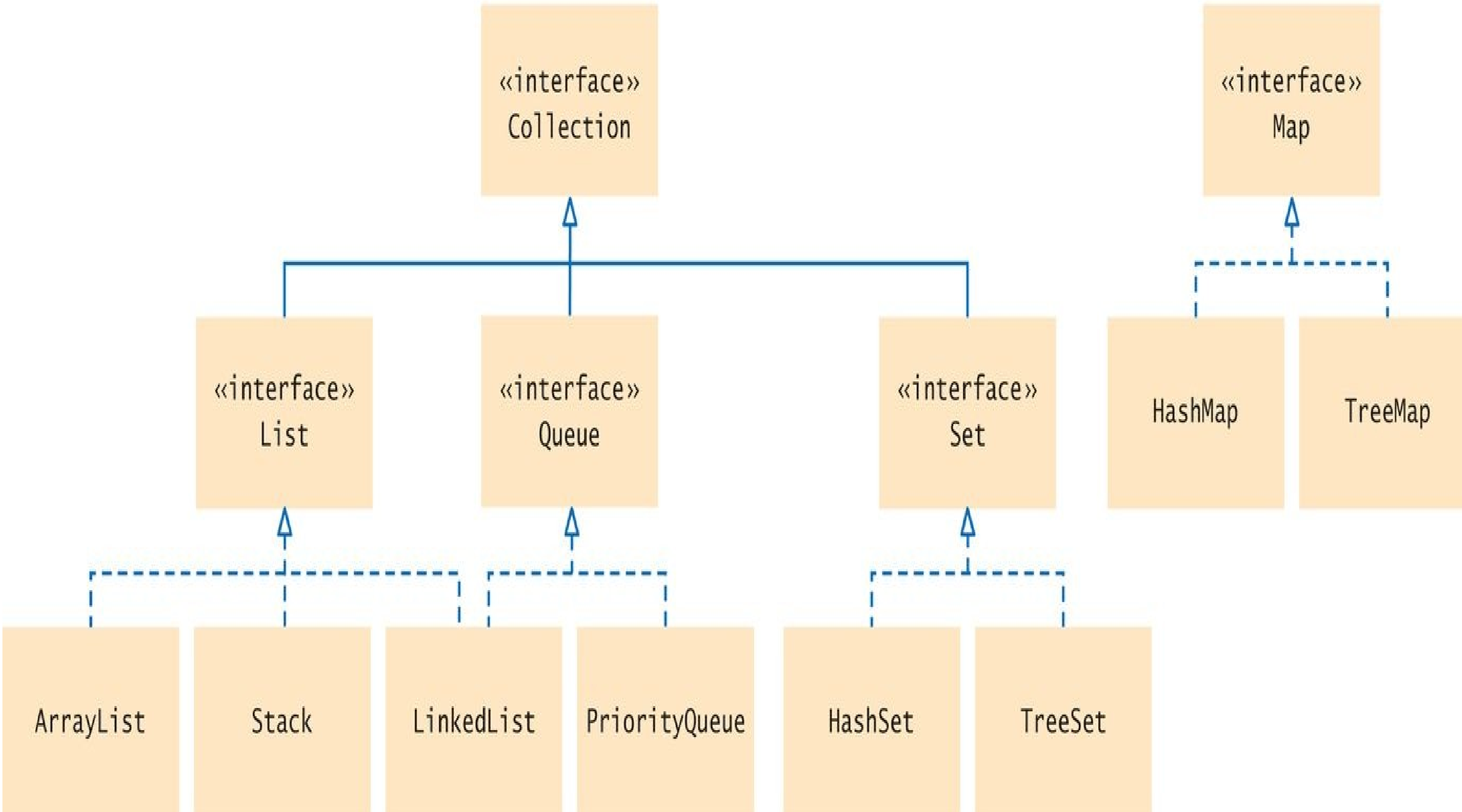
The Java Collections Framework



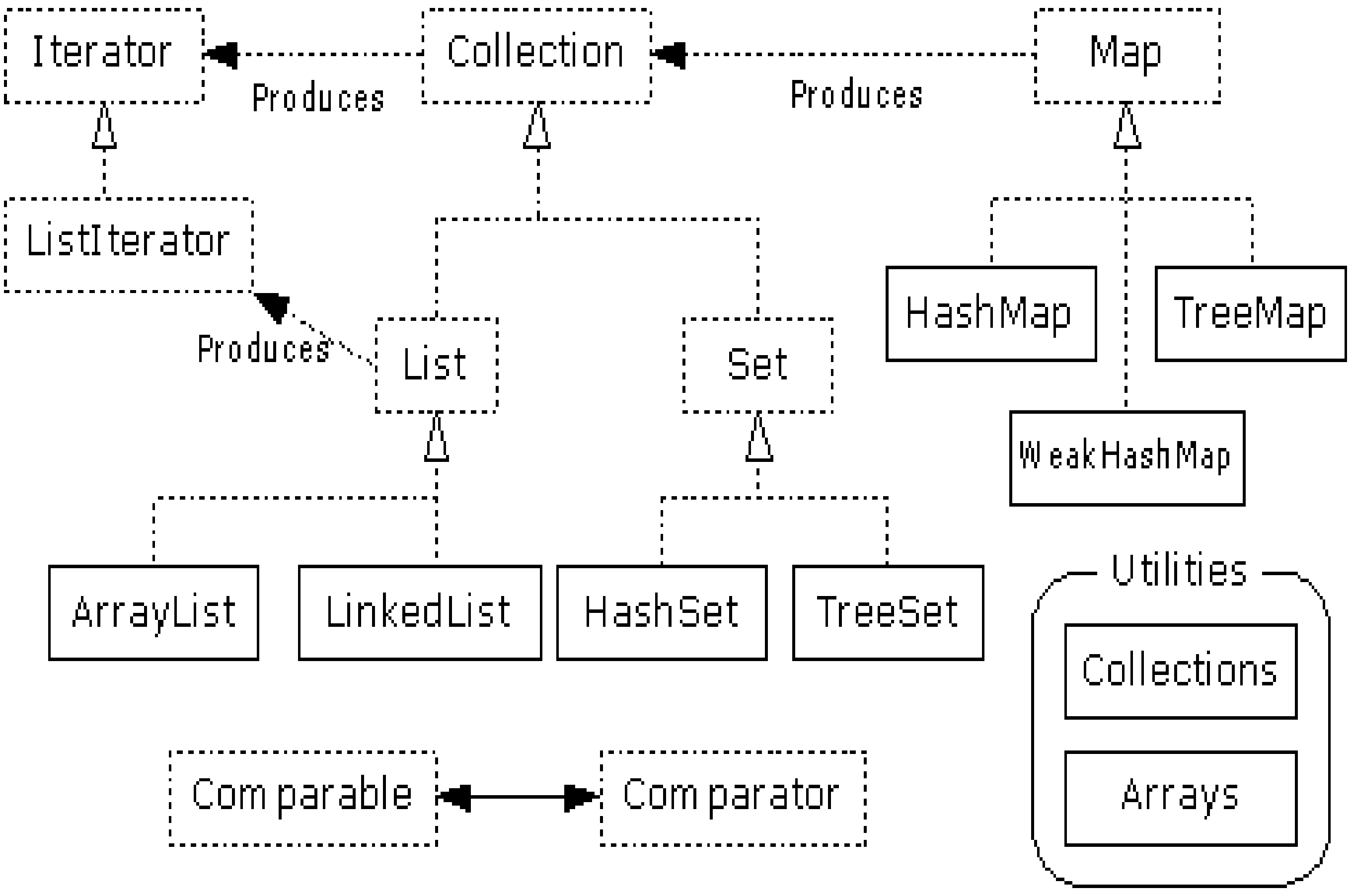
Goals

- To learn how to use the collection classes supplied in the Java library
- To use iterators to traverse collections
- To choose appropriate collections for solving programming problems
- To study applications of stacks and queues

An Overview of the Collections Framework



Collection Interface



Collections Framework

- A collection groups together elements and allows them to be retrieved later.
- Java collections framework: a hierarchy of interface types and classes for collecting objects.
  - Each interface type is implemented by one or more classes
- The Collection interface is at the root
  - All Collection class implement this interface
  - So all have a common set of methods

Collections Framework

- Unified architecture for representing and manipulating collections.
- A collections framework contains three things
  - Interfaces
  - Implementations
  - Algorithms

Collection Interface

- Defines fundamental methods
  - `int size();`
  - `boolean isEmpty();`
  - `boolean contains(Object element);`
  - `boolean add(Object element);`
  - `boolean remove(Object element);`
  - `Iterator iterator();`
- These methods are enough to define the basic behavior of a collection
- Provides an Iterator to step through the elements in the Collection

Iterator Interface

- Defines three fundamental methods
  - `Object next()` (returns the next item in the iterator)
  - `boolean hasNext()` (returns true if the iterator has any more items)
  - `void remove()`
- These three methods provide access to the contents of the collection
- An Iterator knows position within collection
- Each call to next() “reads” an element from the collection
  - Then you can use it or remove it

Iterator Interface

- Note that the next pointer is out-of-bounds after the last call to *next()*
  - another call to *next()* will result in exception
- Should always call *hasNext()* before calling *next()*

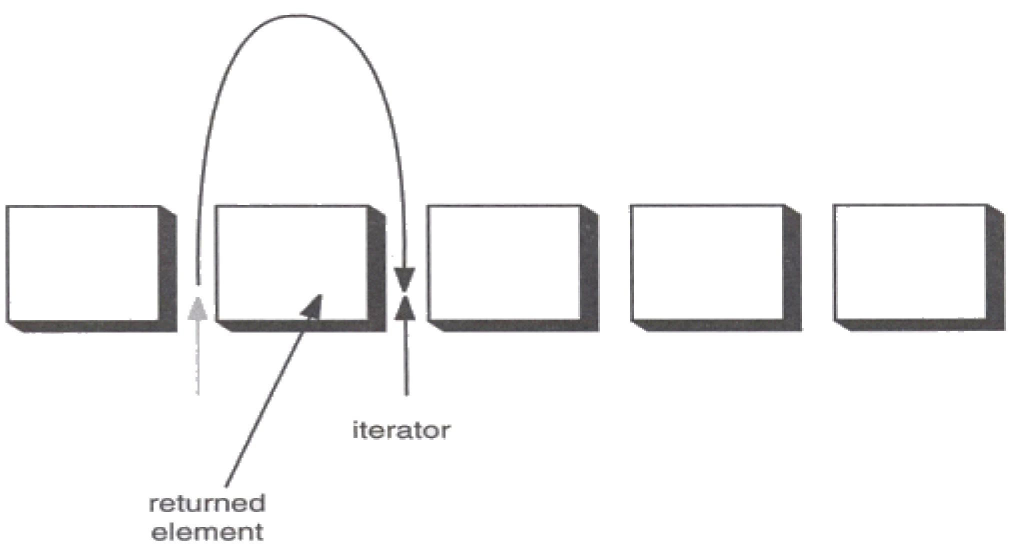


Figure 2–3: Advancing an iterator

Example - SimpleCollection

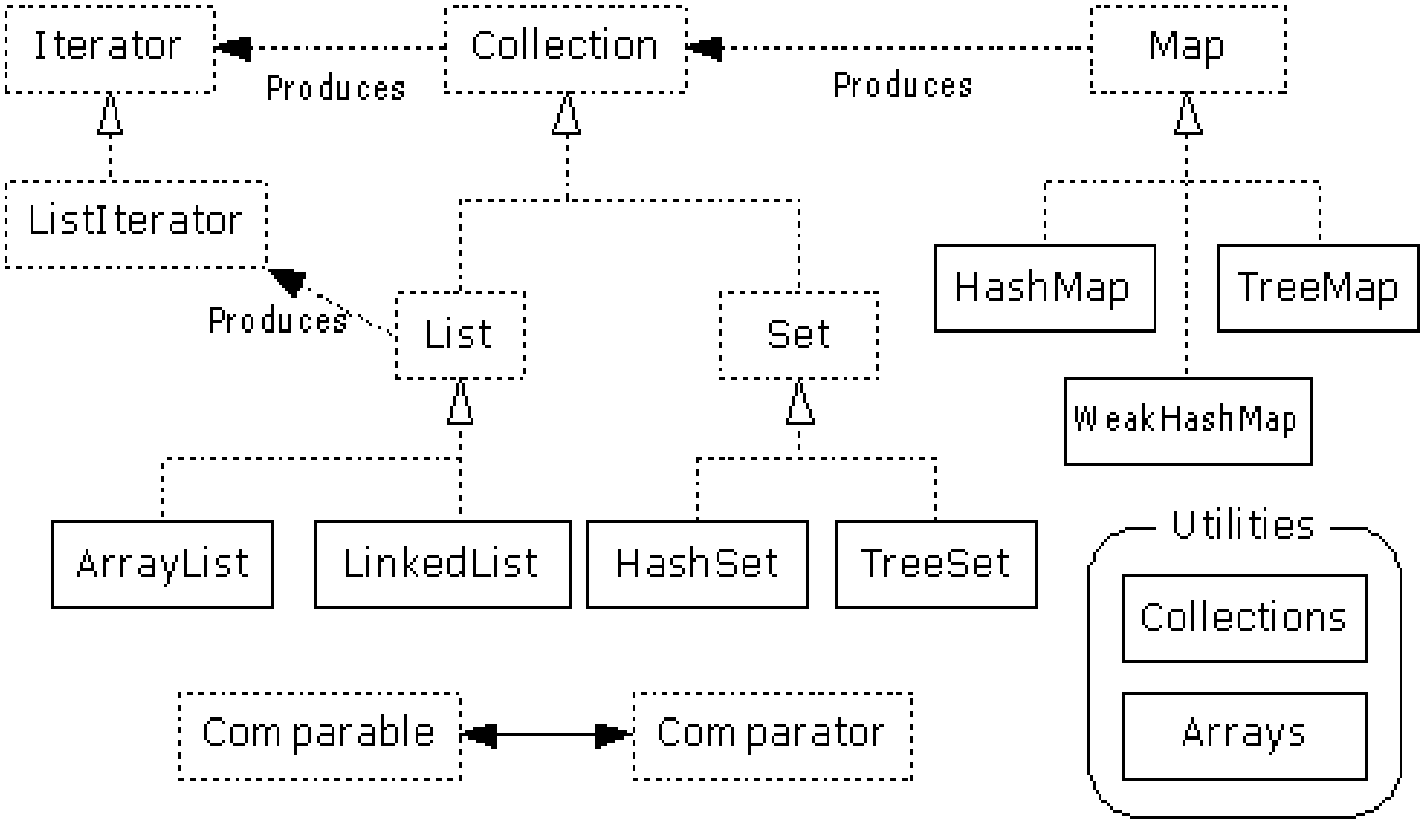
```
public class SimpleCollection  {
  public static void main(String[] args) {
    Collection c;
    c = new ArrayList();
    System.out.println(c.getClass().getName());
    for (int i=1; i <= 10; i++) {
            c.add(i + " * " + i + " = "+i*i);
    }
    Iterator iter = c.iterator();
    while (iter.hasNext())
            System.out.println(iter.next());
  }
}
```

Limitations of Iterator

- You can only move towards forward direction.
  - No previous

- Iterator can only perform read and remove operations.
  - No replacement

Collections Framework Diagram



ListIterator Interface

- Extends the Iterator interface
- Defines three fundamental methods
  - **`void add(Object o)`** - before current position
  - **`boolean hasPrevious()`**
  - **`Object previous()`**
- The addition of these three methods defines the basic behavior of an ordered list
- A ListIterator knows position within list

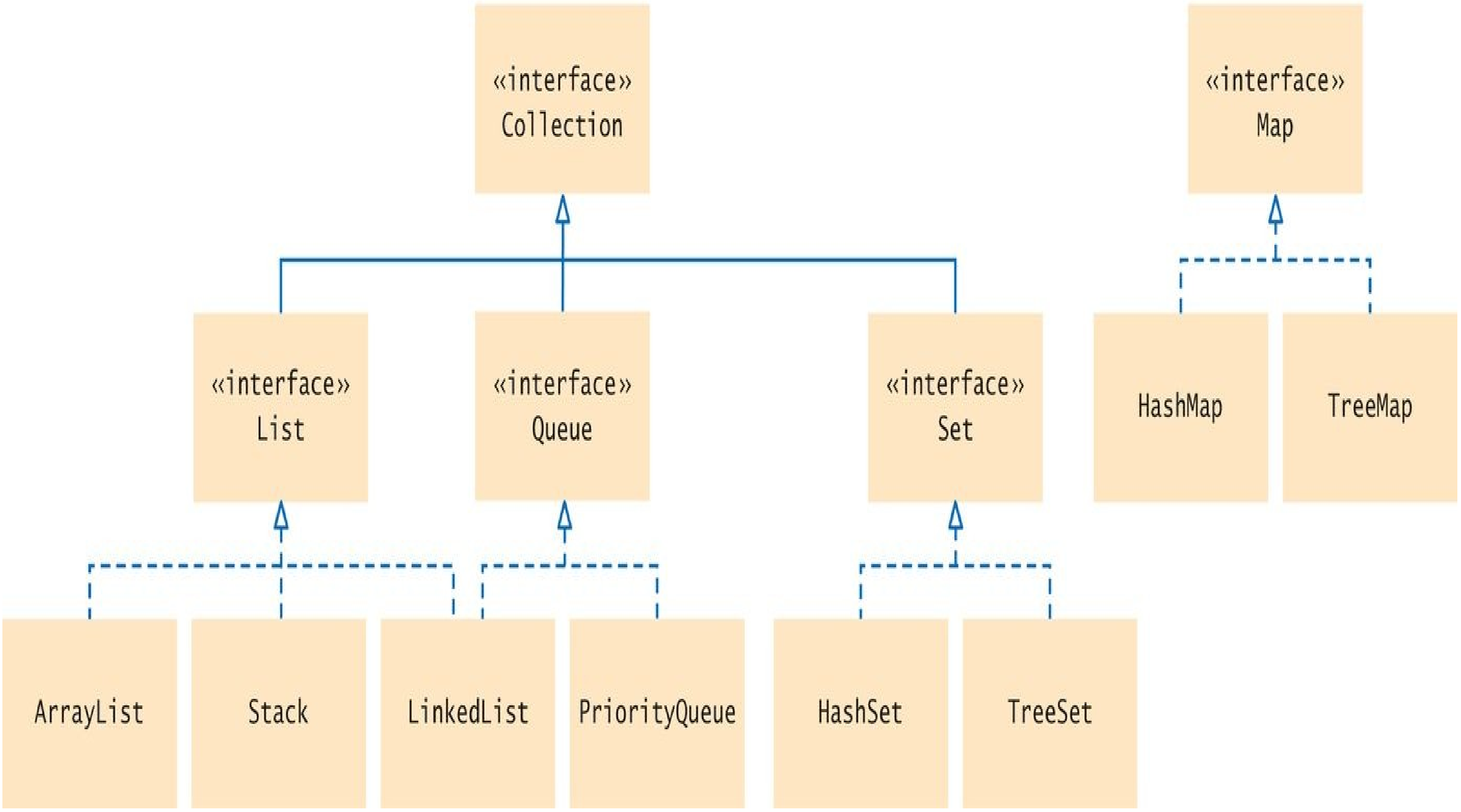
ListIterator Interface

- **Forward direction**
  - `Object next()`
  - `boolean hasNext()`
  - `int nextindex()`
- Backward direction
  - `Object previous`
  - `Boolen hasPrevious()`
  - `int previousindex()`
- Other capabilitie
  - `void remove()`
  - `void set(Object new)`
  - `void add(Object new)`

ListIterator Interface

```
public class IteratorInterface {

	public static void main(String argsp[]){
		LinkedList myList = new LinkedList();
		myList.add("Mike");
		myList.add("John");
		myList.add("Scott");
		myList.add("Patrick");
		System.out.println(myList);

		ListIterator iter = myList.listIterator();

		while (iter.hasNext()){
			String name = (String) iter.next();
			if (name == "John"){
				iter.remove();
			}
			else if (name == "Scott"){
				iter.add("Nancy");
			}
			else if (name == "Patrick")
			{
				iter.set("Michell");
			}
		}
		System.out.println(myList);
	}

}
```

List Implementations



List Interface

- The List interface adds the notion of *order* to a collection
- Lists typically allow *duplicate* elements
- The user of a list has control over where an element is added in the collection
- Provides a ListIterator to step through the elements in the list.

ArrayList overview

- Constant time positional access (it's an array)
- One tuning parameter, the initial capacity
  - // create an empty array list with an initial capacity
  - `ArrayList<String> color_list = new ArrayList<String>(5);`
- The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array
  - **`Object get(int index)`**
  - **`Object set(int index, Object element)`**
- Indexed add and remove are provided, but can be costly if used frequently
  - **`void add(int index, Object element)`**
  - **`Object remove(int index)`**

The Diamond Syntax

- Convenient syntax enhancement for array lists and other generic classes.
  - Mentioned in Chapter 7 Special Topic 7.5.
  - You can write:
    - `ArrayList<String> names = new ArrayList<>();`
  - `Instead of:`
    - `ArrayList<String> names = new ArrayList<String>();`
  - This shortcut is called the "diamond syntax" because the empty brackets <> look like a diamond shape.

LinkedList overview

- A data structure used for collecting a sequence of objects:
- Allows efficient addition and removal of elements in the middle of the sequence.
- A linked list consists of a number of nodes;
- Each node has a reference to the next node.
- A node is an object that stores an element and references to the neighboring nodes.
- Each node in a linked list is connected to the neighboring nodes

LinkedList overview

- Adding and removing elements in the middle of a linked list is efficient.
- Visiting the elements of a linked list in sequential order is efficient.
- Random access is not efficient.
  - Start from beginning or end and traverse each node while counting

LinkedList overview

- When inserting or removing a node:
  - Only the neighboring node references need to be updated

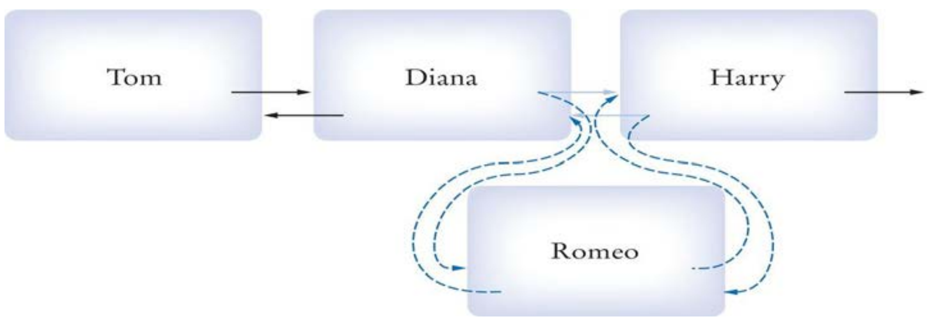


Figure 7 Inserting a Node into a Linked List

Figure 8 Removing a Node From A Linked List

- Visiting the elements of a linked list in sequential order is efficient.
- Random access is not efficient.

LinkedList overview

- When to use a LinkedList:
  - You are concerned about the efficiency of inserting or removing elements
  - You rarely need element access in random order

LinkedList overview

- Generic class
- Specify type of elements in angle brackets: `LinkedList<Product>`
- Package: `java.util`
- `LinkedList` has the methods of the `Collection` interface.
- Some additional `LinkedList` methods:

Table 2 Working with Linked Lists

`LinkedList<String> list = new LinkedList<>();`	An empty list.
`list.addLast("Harry");`	Adds an element to the end of the list. Same as `add`.
`list.addFirst("Sally");`	Adds an element to the beginning of the list. `list` is now `[Sally, Harry]`.
`list.getFirst();`	Gets the element stored at the beginning of the list; here `"Sally"`.
`list.getLast();`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = list.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `list` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = list.listIterator()`	Provides an iterator for visiting all list elements (see Table 3 on page 684).

ArrayList Vs LinkedList

- ArrayList
  - low cost random access.
  - high cost insert and delete.

    (It means move some elements back and then put the element in the middle empty spot hence it's slower)
  - array that resizes if need be.

    (copy contents to new array if array gets full which makes inserting an element into ArrayList of O(n) in worst case)
- LinkedList
  - sequential access.
  - low cost insert and delete.
  - high cost random access.