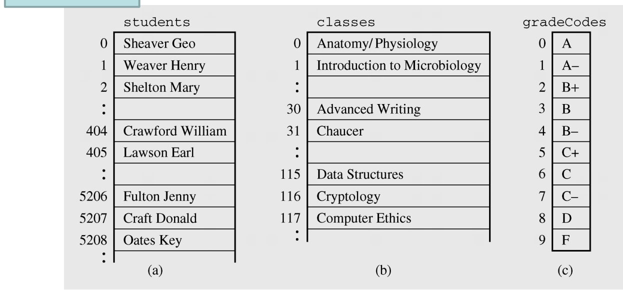
Sparse Tables

- A **sparse table** refers to a table that is populated sparsely by data and most of its cells are empty
- With a sparse table, the table can be replaced by a system of linked lists

Sparse Tables (continued)

Data source



students

0	Sheaver Geo
1	Weaver Henry
2	Shelton Mary
:	
404	Crawford William
405	Lawson Earl
:	
5206	Fulton Jenny
5207	Craft Donald
5208	Oates Key
:	

(a)

classes

0	Anatomy/Physiology
1	Introduction to Microbiology
:	
30	Advanced Writing
31	Chaucer
:	
115	Data Structures
116	Cryptology
117	Computer Ethics
:	

(b)

gradeCodes

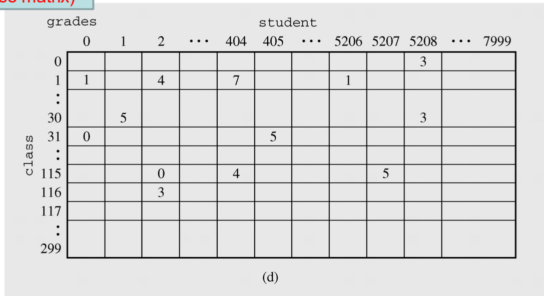
0	A
1	A–
2	B+
3	B
4	B–
5	C+
6	C
7	C–
8	D
9	F

(c)

Figure 3-22 Arrays and sparse table used for storing student grades

Sparse Tables (continued)

Table of grades (sparse matrix)



grades

student

class	0	1	2	⋯	404	405	⋯	5206	5207	5208	⋯ 7999
0										3	
1	1		4		7			1			
: 30		5								3	
31	0					5					
:											
115			0		4				5		
116			3								
117											
: 299											

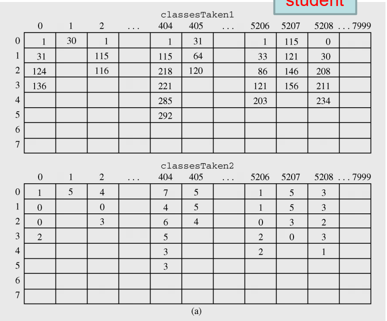
(d)

Figure 3-22 Arrays and sparse table used for storing student grades (continued)

Sparse Tables (continued)

student

class



classesTaken1

	0	1	2	. . .	404	405	. . .	5206	5207	5208	. . . 7999
0	1	30	1		1	31		1	115	0	
1	31		115		115	64		33	121	30	
2	124		116		218	120		86	146	208	
3	136				221			121	156	211	
4					285			203		234	
5					292						
6											
7											

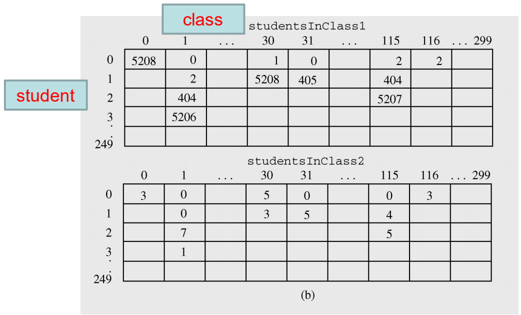
classesTaken2

	0	1	2	. . .	404	405	. . .	5206	5207	5208	. . . 7999
0	1	5	4		7	5		1	5	3	
1	0		0		4	5		1	5	3	
2	0		3		6	4		0	3	2	
3	2				5			2	0	3	
4					3			2		1	
5					3						
6											
7											

(a)

Figure 3-23 Two-dimensional arrays for storing student grades

Sparse Tables (continued)



class

student

studentsInClass1

	0	1	. . .	30	31	. . .	115	116	. . . 299
0	5208	0		1	0		2	2	
1		2		5208	405		404		
2		404					5207		
3		5206							
⋮ 249									

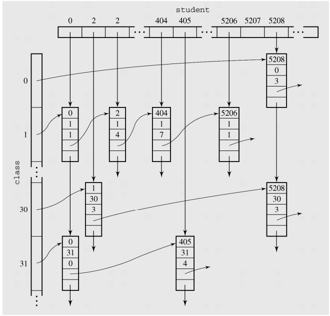
studentsInClass2

	0	1	. . .	30	31	. . .	115	116	. . . 299
0	3	0		5	0		0	3	
1		0		3	5		4		
2		7					5		
3		1							
⋮ 249									

(b)

Figure 3-23 Two-dimensional arrays for storing student grades (continued)

Lists in `java.util`



Horiziontal list: Students study the same class (subject)

Grade is a field of a node of the list

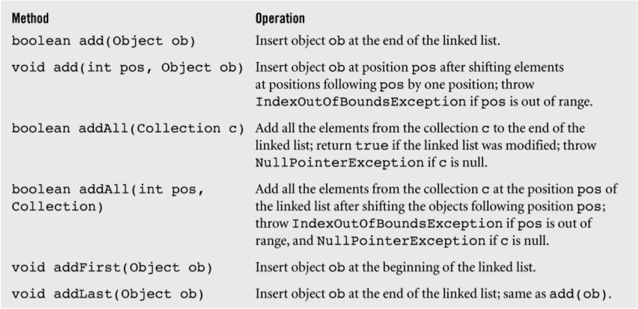
Vertical list: List of subjects that this student enrolled

Figure 3-24 Student grades implemented using linked lists

Project No2

- Using List in Java to implement this application

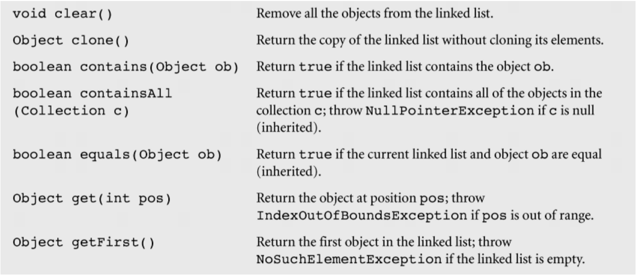
Lists in `java.util` (continued)



Method	Operation
`boolean add(Object ob)`	Insert object ob at the end of the linked list.
`void add(int pos, Object ob)`	Insert object ob at position pos after shifting elements at positions following pos by one position; throw `IndexOutOfBoundsException` if pos is out of range.
`boolean addAll(Collection c)`	Add all the elements from the collection c to the end of the linked list; return `true` if the linked list was modified; throw `NullPointerException` if c is null.
`boolean addAll(int pos, Collection)`	Add all the elements from the collection c at the position pos of the linked list after shifting the objects following position pos; throw `IndexOutOfBoundsException` if pos is out of range, and `NullPointerException` if c is null.
`void addFirst(Object ob)`	Insert object ob at the beginning of the linked list.
`void addLast(Object ob)`	Insert object ob at the end of the linked list; same as `add(ob)`.

Figure 3-25 An alphabetical list of methods in the class `LinkedList` including some inherited methods

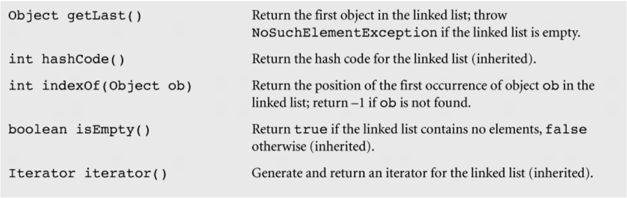
Lists in `java.util` (continued)



`void clear()`	Remove all the objects from the linked list.
`Object clone()`	Return the copy of the linked list without cloning its elements.
`boolean contains(Object ob)`	Return `true` if the linked list contains the object `ob`.
`boolean containsAll (Collection c)`	Return `true` if the linked list contains all of the objects in the collection `c`; throw `NullPointerException` if `c` is null (inherited).
`boolean equals(Object ob)`	Return `true` if the current linked list and object `ob` are equal (inherited).
`Object get(int pos)`	Return the object at position `pos`; throw `IndexOutOfBoundsException` if `pos` is out of range.
`Object getFirst()`	Return the first object in the linked list; throw `NoSuchElementException` if the linked list is empty.

Figure 3-25 An alphabetical list of methods in the class `LinkedList` including some inherited methods (continued)

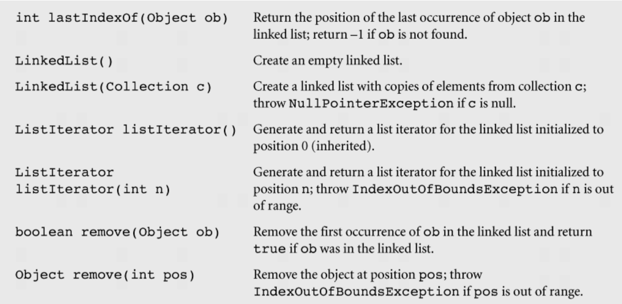
Lists in `java.util` (continued)



`Object getLast()`	Return the first object in the linked list; throw `NoSuchElementException` if the linked list is empty.
`int hashCode()`	Return the hash code for the linked list (inherited).
`int indexOf(Object ob)`	Return the position of the first occurrence of object `ob` in the linked list; return –1 if `ob` is not found.
`boolean isEmpty()`	Return `true` if the linked list contains no elements, `false` otherwise (inherited).
`Iterator iterator()`	Generate and return an iterator for the linked list (inherited).

Figure 3-25 An alphabetical list of methods in the class `LinkedList` including some inherited methods (continued)

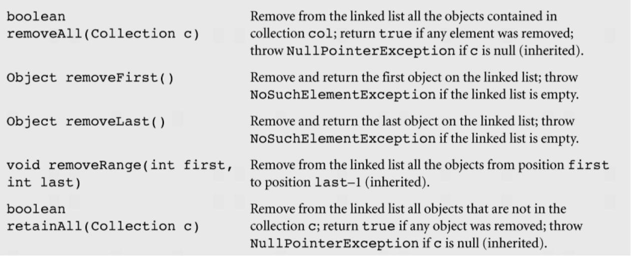
Lists in `java.util` (continued)



`int lastIndexOf(Object ob)`	Return the position of the last occurrence of object `ob` in the linked list; return –1 if `ob` is not found.
`LinkedList()`	Create an empty linked list.
`LinkedList(Collection c)`	Create a linked list with copies of elements from collection `c`; throw `NullPointerException` if `c` is null.
`ListIterator listIterator()`	Generate and return a list iterator for the linked list initialized to position 0 (inherited).
`ListIterator listIterator(int n)`	Generate and return a list iterator for the linked list initialized to position `n`; throw `IndexOutOfBoundsException` if `n` is out of range.
`boolean remove(Object ob)`	Remove the first occurrence of `ob` in the linked list and return `true` if `ob` was in the linked list.
`Object remove(int pos)`	Remove the object at position `pos`; throw `IndexOutOfBoundsException` if `pos` is out of range.

Figure 3-25 An alphabetical list of methods in the class `LinkedList` including some inherited methods (continued)

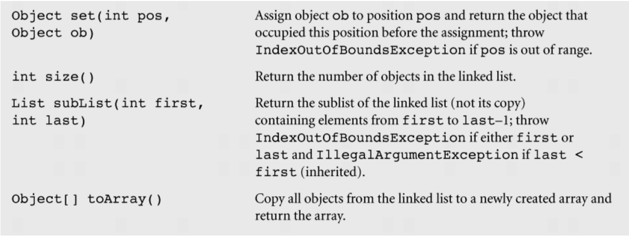
Lists in `java.util` (continued)



`boolean removeAll(Collection c)`	Remove from the linked list all the objects contained in collection `col`; return `true` if any element was removed; throw `NullPointerException` if `c` is null (inherited).
`Object removeFirst()`	Remove and return the first object on the linked list; throw `NoSuchElementException` if the linked list is empty.
`Object removeLast()`	Remove and return the last object on the linked list; throw `NoSuchElementException` if the linked list is empty.
`void removeRange(int first, int last)`	Remove from the linked list all the objects from position `first` to position `last`–1 (inherited).
`boolean retainAll(Collection c)`	Remove from the linked list all objects that are not in the collection `c`; return `true` if any object was removed; throw `NullPointerException` if `c` is null (inherited).

Figure 3-25 An alphabetical list of methods in the class `LinkedList` including some inherited methods (continued)

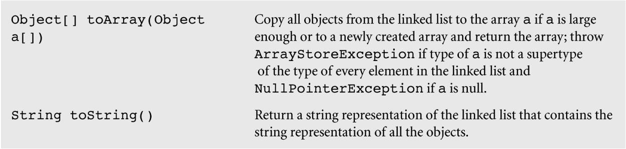
Lists in `java.util` (continued)



`Object set(int pos, Object ob)`	Assign object `ob` to position `pos` and return the object that occupied this position before the assignment; throw `IndexOutOfBoundsException` if `pos` is out of range.
`int size()`	Return the number of objects in the linked list.
`List subList(int first, int last)`	Return the sublist of the linked list (not its copy) containing elements from `first` to `last`−1; throw `IndexOutOfBoundsException` if either `first` or `last` and `IllegalArgumentException` if `last < first` (inherited).
`Object[] toArray()`	Copy all objects from the linked list to a newly created array and return the array.

Figure 3-25 An alphabetical list of methods in the class `LinkedList` including some inherited methods (continued)

Lists in `java.util` (continued)



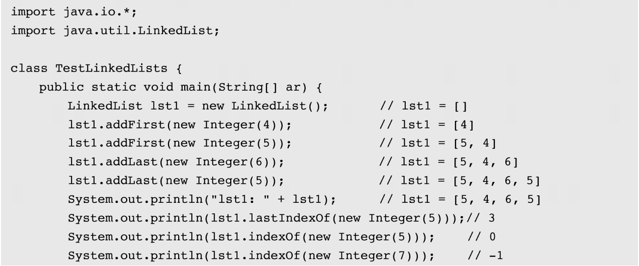
`Object[] toArray(Object a[])`	Copy all objects from the linked list to the array `a` if `a` is large enough or to a newly created array and return the array; throw `ArrayStoreException` if type of `a` is not a supertype of the type of every element in the linked list and `NullPointerException` if `a` is null.
`String toString()`	Return a string representation of the linked list that contains the string representation of all the objects.

Figure 3-25 An alphabetical list of methods in the class `LinkedList` including some inherited methods (continued)

LAB

- Tạo một LinkList
- Yêu cầu người dùng nhập vào số lượng "n" phần tử vào danh sách
- Thêm "n" phần tử là số nguyên từ [-1000,1000]
- Hãy in ra các giá trị trong danh sách
- Hãy tính tổng các giá trị trong danh sách
- Hãy tìm số lớn nhất trong danh sách
- Hãy loại ra khỏi danh sách các số là số nguyên không dương.
- Hãy nghịch đảo mảng lại và xuất ra danh sách các phần tử
- Hãy sắp xếp mảng theo thứ tự tăng dần và xuất ra màn hình
- Với mảng đã sắp xếp, hãy tìm các phần tử bằng tổng của 2 phần tử liền trước.
- Hãy xuất ra màn hình những phần tử bằng tổng của các phần tử khác trong danh sách (nếu có) và xuất ra (không có) trường hợp ngược lại

Lists in `java.util` (continued)

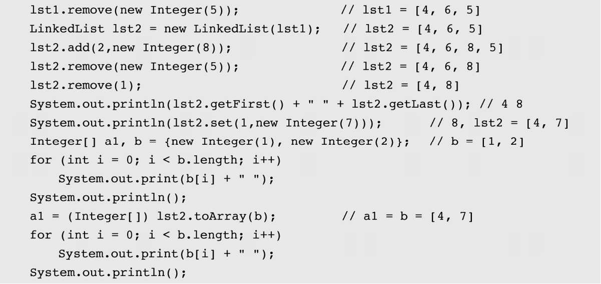


```
import java.io.*;
import java.util.LinkedList;

class TestLinkedLists {
    public static void main(String[] ar) {
        LinkedList lst1 = new LinkedList();       // lst1 = []
        lst1.addFirst(new Integer(4));            // lst1 = [4]
        lst1.addFirst(new Integer(5));            // lst1 = [5, 4]
        lst1.addLast(new Integer(6));             // lst1 = [5, 4, 6]
        lst1.addLast(new Integer(5));             // lst1 = [5, 4, 6, 5]
        System.out.println("lst1: " + lst1);      // lst1 = [5, 4, 6, 5]
        System.out.println(lst1.lastIndexOf(new Integer(5)));// 3
        System.out.println(lst1.indexOf(new Integer(5)));    // 0
        System.out.println(lst1.indexOf(new Integer(7)));    // -1
```

Figure 3-26 A program demonstrating the operation of `LinkedList` methods

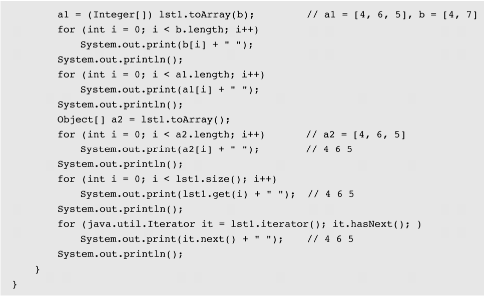
Lists in `java.util` (continued)



```
    lst1.remove(new Integer(5));             // lst1 = [4, 6, 5]
    LinkedList lst2 = new LinkedList(lst1);   // lst2 = [4, 6, 5]
    lst2.add(2,new Integer(8));              // lst2 = [4, 6, 8, 5]
    lst2.remove(new Integer(5));             // lst2 = [4, 6, 8]
    lst2.remove(1);                          // lst2 = [4, 8]
    System.out.println(lst2.getFirst() + " " + lst2.getLast()); // 4 8
    System.out.println(lst2.set(1,new Integer(7)));      // 8, lst2 = [4, 7]
    Integer[] a1, b = {new Integer(1), new Integer(2)};   // b = [1, 2]
    for (int i = 0; i < b.length; i++)
        System.out.print(b[i] + " ");
    System.out.println();
    a1 = (Integer[]) lst2.toArray(b);        // a1 = b = [4, 7]
    for (int i = 0; i < b.length; i++)
        System.out.print(b[i] + " ");
    System.out.println();
```

Figure 3-26 A program demonstrating the operation of `LinkedList` methods (continued)

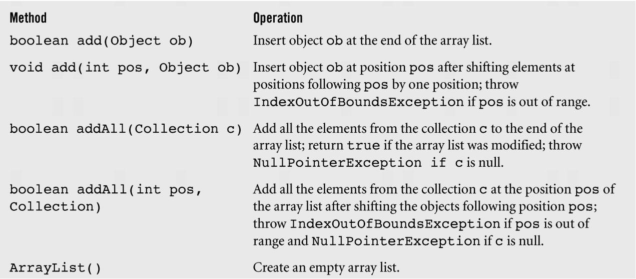
Lists in `java.util` (continued)



```
        a1 = (Integer[]) lst1.toArray(b);        // a1 = [4, 6, 5], b = [4, 7]
        for (int i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
        System.out.println();
        for (int i = 0; i < a1.length; i++)
            System.out.print(a1[i] + " ");
        System.out.println();
        Object[] a2 = lst1.toArray();
        for (int i = 0; i < a2.length; i++)      // a2 = [4, 6, 5]
            System.out.print(a2[i] + " ");       // 4 6 5
        System.out.println();
        for (int i = 0; i < lst1.size(); i++)
            System.out.print(lst1.get(i) + " ");  // 4 6 5
        System.out.println();
        for (java.util.Iterator it = lst1.iterator(); it.hasNext(); )
            System.out.print(it.next() + " ");    // 4 6 5
        System.out.println();
    }
}
```

Figure 3-26 A program demonstrating the operation of `LinkedList` methods (continued)

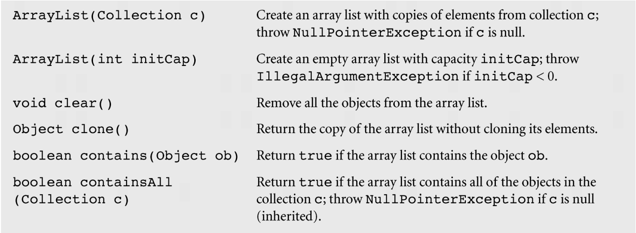
Lists in `java.util` (continued)



Method	Operation
`boolean add(Object ob)`	Insert object `ob` at the end of the array list.
`void add(int pos, Object ob)`	Insert object `ob` at position `pos` after shifting elements at positions following `pos` by one position; throw `IndexOutOfBoundsException` if `pos` is out of range.
`boolean addAll(Collection c)`	Add all the elements from the collection `c` to the end of the array list; return `true` if the array list was modified; throw `NullPointerException if c` is null.
`boolean addAll(int pos, Collection)`	Add all the elements from the collection `c` at the position `pos` of the array list after shifting the objects following position `pos`; throw `IndexOutOfBoundsException` if `pos` is out of range and `NullPointerException` if `c` is null.
`ArrayList()`	Create an empty array list.

Figure 3-27 An alphabetical list of methods in the class `ArrayList`

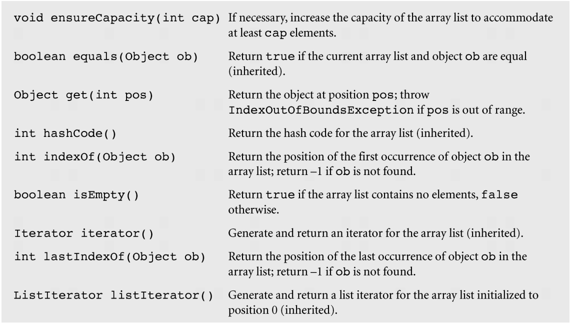
Lists in `java.util` (continued)



`ArrayList(Collection c)`	Create an array list with copies of elements from collection `c`; throw `NullPointerException` if `c` is null.
`ArrayList(int initCap)`	Create an empty array list with capacity `initCap`; throw `IllegalArgumentException` if `initCap` < 0.
`void clear()`	Remove all the objects from the array list.
`Object clone()`	Return the copy of the array list without cloning its elements.
`boolean contains(Object ob)`	Return `true` if the array list contains the object `ob`.
`boolean containsAll (Collection c)`	Return `true` if the array list contains all of the objects in the collection `c`; throw `NullPointerException` if `c` is null (inherited).

Figure 3-27 An alphabetical list of methods in the class `ArrayList` (continued)

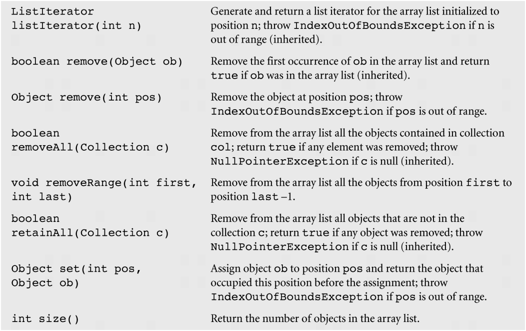
Lists in java.util (continued)



`void ensureCapacity(int cap)`	If necessary, increase the capacity of the array list to accommodate at least `cap` elements.
`boolean equals(Object ob)`	Return `true` if the current array list and object `ob` are equal (inherited).
`Object get(int pos)`	Return the object at position `pos`; throw `IndexOutOfBoundsException` if `pos` is out of range.
`int hashCode()`	Return the hash code for the array list (inherited).
`int indexOf(Object ob)`	Return the position of the first occurrence of object `ob` in the array list; return –1 if `ob` is not found.
`boolean isEmpty()`	Return `true` if the array list contains no elements, `false` otherwise.
`Iterator iterator()`	Generate and return an iterator for the array list (inherited).
`int lastIndexOf(Object ob)`	Return the position of the last occurrence of object `ob` in the array list; return –1 if `ob` is not found.
`ListIterator listIterator()`	Generate and return a list iterator for the array list initialized to position 0 (inherited).

Figure 3-27 An alphabetical list of methods in the class `ArrayList` (continued)

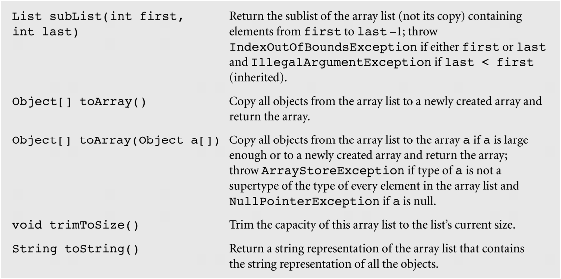
Lists in java.util (continued)



`ListIterator listIterator(int n)`	Generate and return a list iterator for the array list initialized to position `n`; throw `IndexOutOfBoundsException` if `n` is out of range (inherited).
`boolean remove(Object ob)`	Remove the first occurrence of `ob` in the array list and return `true` if `ob` was in the array list (inherited).
`Object remove(int pos)`	Remove the object at position `pos`; throw `IndexOutOfBoundsException` if `pos` is out of range.
`boolean removeAll(Collection c)`	Remove from the array list all the objects contained in collection `col`; return `true` if any element was removed; throw `NullPointerException` if `c` is null (inherited).
`void removeRange(int first, int last)`	Remove from the array list all the objects from position `first` to position `last` –1.
`boolean retainAll(Collection c)`	Remove from the array list all objects that are not in the collection `c`; return `true` if any object was removed; throw `NullPointerException` if `c` is null (inherited).
`Object set(int pos, Object ob)`	Assign object `ob` to position `pos` and return the object that occupied this position before the assignment; throw `IndexOutOfBoundsException` if `pos` is out of range.
`int size()`	Return the number of objects in the array list.

Figure 3-27 An alphabetical list of methods in the class `ArrayList` (continued)

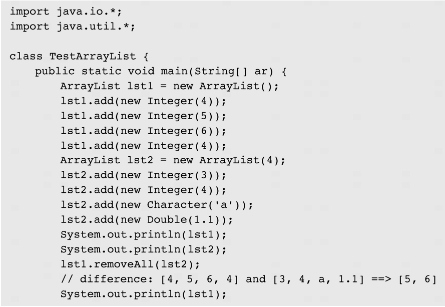
Lists in java.util (continued)



`List subList(int first, int last)`	Return the sublist of the array list (not its copy) containing elements from `first` to `last` –1; throw `IndexOutOfBoundsException` if either `first` or `last` and `IllegalArgumentException` if `last < first` (inherited).
`Object[] toArray()`	Copy all objects from the array list to a newly created array and return the array.
`Object[] toArray(Object a[])`	Copy all objects from the array list to the array `a` if `a` is large enough or to a newly created array and return the array; throw `ArrayStoreException` if type of `a` is not a supertype of the type of every element in the array list and `NullPointerException` if `a` is null.
`void trimToSize()`	Trim the capacity of this array list to the list's current size.
`String toString()`	Return a string representation of the array list that contains the string representation of all the objects.

Figure 3-27 An alphabetical list of methods in the class `ArrayList` (continued)

Lists in java.util (continued)

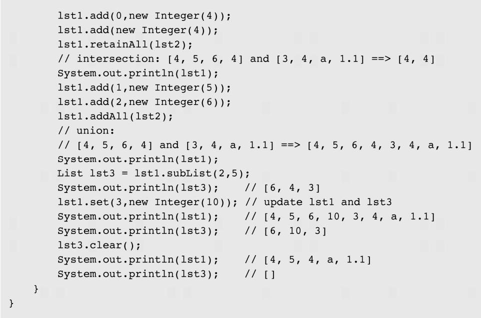


```
import java.io.*;
import java.util.*;

class TestArrayList {
    public static void main(String[] ar) {
        ArrayList lst1 = new ArrayList();
        lst1.add(new Integer(4));
        lst1.add(new Integer(5));
        lst1.add(new Integer(6));
        lst1.add(new Integer(4));
        ArrayList lst2 = new ArrayList(4);
        lst2.add(new Integer(3));
        lst2.add(new Integer(4));
        lst2.add(new Character('a'));
        lst2.add(new Double(1.1));
        System.out.println(lst1);
        System.out.println(lst2);
        lst1.removeAll(lst2);
        // difference: [4, 5, 6, 4] and [3, 4, a, 1.1] ==> [5, 6]
        System.out.println(lst1);
```

Figure 3-28 A program demonstrating the operation of `ArrayList` methods

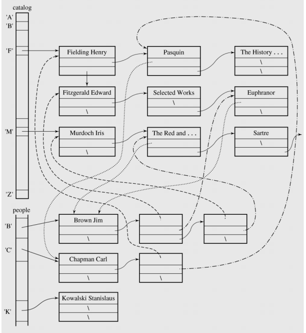
Lists in `java.util` (continued)



```
        lst1.add(0,new Integer(4));
        lst1.add(new Integer(4));
        lst1.retainAll(lst2);
        // intersection: [4, 5, 6, 4] and [3, 4, a, 1.1] ==> [4, 4]
        System.out.println(lst1);
        lst1.add(1,new Integer(5));
        lst1.add(2,new Integer(6));
        lst1.addAll(lst2);
        // union:
        // [4, 5, 6, 4] and [3, 4, a, 1.1] ==> [4, 5, 6, 4, 3, 4, a, 1.1]
        System.out.println(lst1);
        List lst3 = lst1.subList(2,5);
        System.out.println(lst3);    // [6, 4, 3]
        lst1.set(3,new Integer(10)); // update lst1 and lst3
        System.out.println(lst1);    // [4, 5, 6, 10, 3, 4, a, 1.1]
        System.out.println(lst3);    // [6, 10, 3]
        lst3.clear();
        System.out.println(lst1);    // [4, 5, 4, a, 1.1]
        System.out.println(lst3);    // []
    }
}
```

Figure 3-28 A program demonstrating the operation of `ArrayList` methods (continued)

Case Study: A Library



Read yourself
Go to the summary)

Figure 3-29 Linked lists indicating library status

Case Study: A Library (continued)

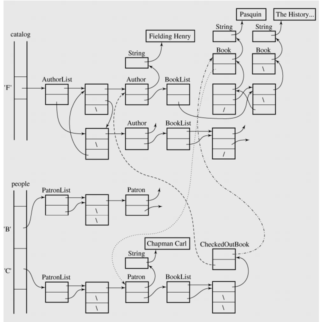
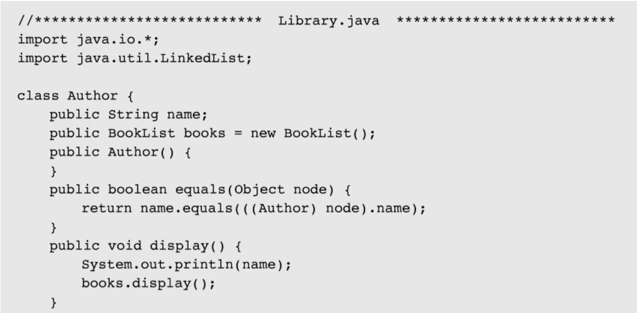


Figure 3-30 Fragment of structure from Figure 3-29 with all the objects used in the implementation

Case Study: A Library (continued)

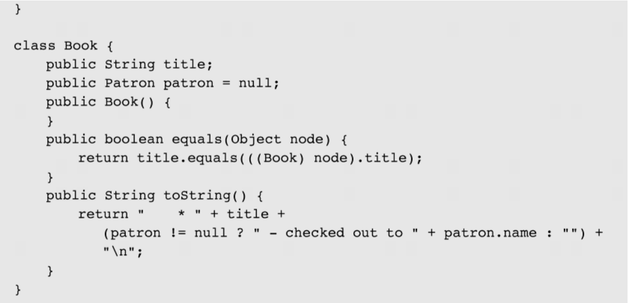


```
//**************************  Library.java  **************************
import java.io.*;
import java.util.LinkedList;

class Author {
    public String name;
    public BookList books = new BookList();
    public Author() {
    }
    public boolean equals(Object node) {
        return name.equals(((Author) node).name);
    }
    public void display() {
        System.out.println(name);
        books.display();
    }
```

Figure 3-31 The library program (continued)

Case Study: A Library (continued)

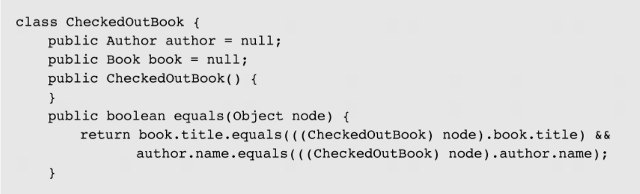


```
}

class Book {
    public String title;
    public Patron patron = null;
    public Book() {
    }
    public boolean equals(Object node) {
        return title.equals(((Book) node).title);
    }
    public String toString() {
        return "    * " + title +
          (patron != null ? " - checked out to " + patron.name : "") +
          "\n";
    }
}
```

Figure 3-31 The library program (continued)

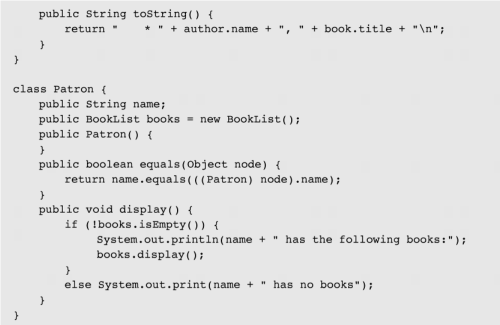
Case Study: A Library (continued)



```
class CheckedOutBook {
    public Author author = null;
    public Book book = null;
    public CheckedOutBook() {
    }
    public boolean equals(Object node) {
        return book.title.equals(((CheckedOutBook) node).book.title) &&
               author.name.equals(((CheckedOutBook) node).author.name);
    }
```

Figure 3-31 The library program (continued)

Case Study: A Library (continued)

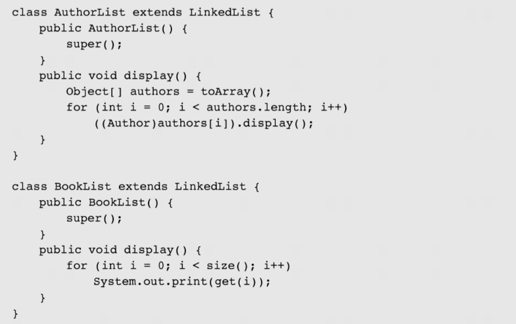


```
    public String toString() {
        return "    * " + author.name + ", " + book.title + "\n";
    }
}

class Patron {
    public String name;
    public BookList books = new BookList();
    public Patron() {
    }
    public boolean equals(Object node) {
        return name.equals(((Patron) node).name);
    }
    public void display() {
        if (!books.isEmpty()) {
            System.out.println(name + " has the following books:");
            books.display();
        }
        else System.out.print(name + " has no books");
    }
}
```

Figure 3-31 The library program (continued)

Case Study: A Library (continued)

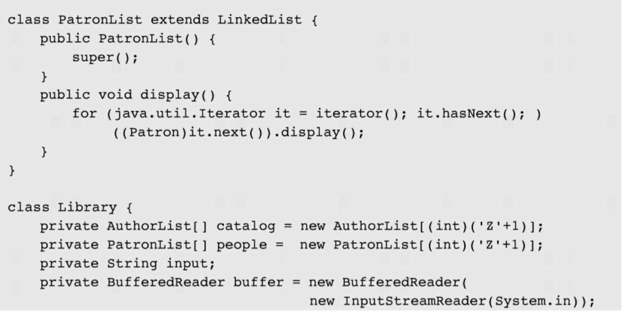


```
class AuthorList extends LinkedList {
    public AuthorList() {
        super();
    }
    public void display() {
        Object[] authors = toArray();
        for (int i = 0; i < authors.length; i++)
            ((Author)authors[i]).display();
    }
}

class BookList extends LinkedList {
    public BookList() {
        super();
    }
    public void display() {
        for (int i = 0; i < size(); i++)
            System.out.print(get(i));
    }
}
```

Figure 3-31 The library program (continued)

Case Study: A Library (continued)

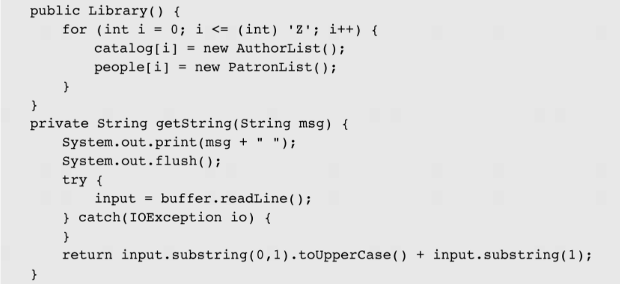


```
class PatronList extends LinkedList {
    public PatronList() {
        super();
    }
    public void display() {
        for (java.util.Iterator it = iterator(); it.hasNext(); )
             ((Patron)it.next()).display();
    }
}

class Library {
    private AuthorList[] catalog = new AuthorList[(int)('Z'+1)];
    private PatronList[] people =  new PatronList[(int)('Z'+1)];
    private String input;
    private BufferedReader buffer = new BufferedReader(
                                     new InputStreamReader(System.in));
```

Figure 3-31 The library program (continued)

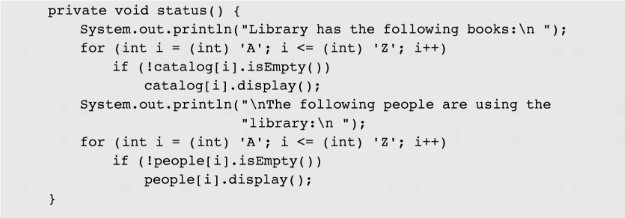
Case Study: A Library (continued)



```
    public Library() {
        for (int i = 0; i <= (int) 'Z'; i++) {
            catalog[i] = new AuthorList();
            people[i] = new PatronList();
        }
    }
    private String getString(String msg) {
        System.out.print(msg + " ");
        System.out.flush();
        try {
            input = buffer.readLine();
        } catch(IOException io) {
        }
        return input.substring(0,1).toUpperCase() + input.substring(1);
    }
```

Figure 3-31 The library program (continued)

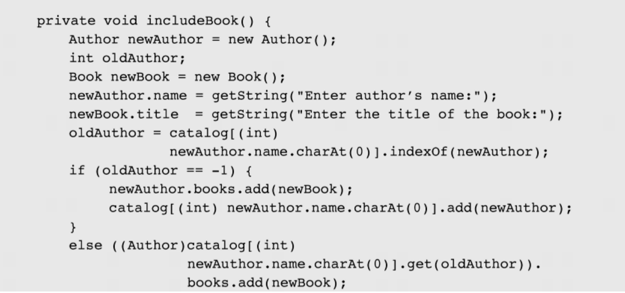
Case Study: A Library (continued)



```
    private void status() {
        System.out.println("Library has the following books:\n ");
        for (int i = (int) 'A'; i <= (int) 'Z'; i++)
            if (!catalog[i].isEmpty())
                catalog[i].display();
        System.out.println("\nThe following people are using the
                           "library:\n ");
        for (int i = (int) 'A'; i <= (int) 'Z'; i++)
            if (!people[i].isEmpty())
                people[i].display();
    }
```

Figure 3-31 The library program (continued)

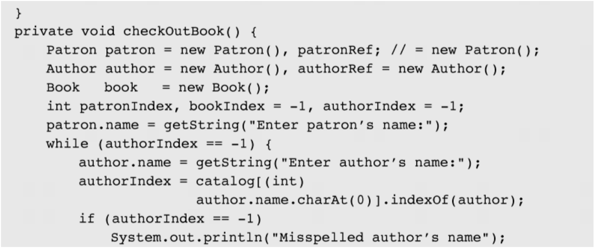
Case Study: A Library (continued)



```
    private void includeBook() {
        Author newAuthor = new Author();
        int oldAuthor;
        Book newBook = new Book();
        newAuthor.name = getString("Enter author’s name:");
        newBook.title  = getString("Enter the title of the book:");
        oldAuthor = catalog[(int)
                    newAuthor.name.charAt(0)].indexOf(newAuthor);
        if (oldAuthor == -1) {
             newAuthor.books.add(newBook);
             catalog[(int) newAuthor.name.charAt(0)].add(newAuthor);
        }
        else ((Author)catalog[(int)
                      newAuthor.name.charAt(0)].get(oldAuthor)).
                      books.add(newBook);
```

Figure 3-31 The library program (continued)

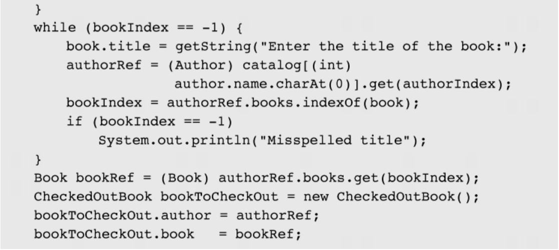
Case Study: A Library (continued)



```
}
private void checkOutBook() {
    Patron patron = new Patron(), patronRef; // = new Patron();
    Author author = new Author(), authorRef = new Author();
    Book   book   = new Book();
    int patronIndex, bookIndex = -1, authorIndex = -1;
    patron.name = getString("Enter patron’s name:");
    while (authorIndex == -1) {
        author.name = getString("Enter author’s name:");
        authorIndex = catalog[(int)
                      author.name.charAt(0)].indexOf(author);
        if (authorIndex == -1)
            System.out.println("Misspelled author’s name");
```

Figure 3-31 The library program (continued)

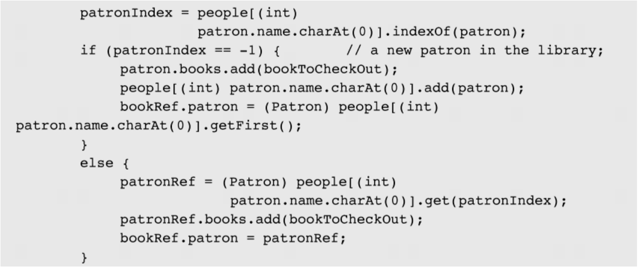
Case Study: A Library (continued)



```
}
while (bookIndex == -1) {
    book.title = getString("Enter the title of the book:");
    authorRef = (Author) catalog[(int)
                 author.name.charAt(0)].get(authorIndex);
    bookIndex = authorRef.books.indexOf(book);
    if (bookIndex == -1)
        System.out.println("Misspelled title");
}
Book bookRef = (Book) authorRef.books.get(bookIndex);
CheckedOutBook bookToCheckOut = new CheckedOutBook();
bookToCheckOut.author = authorRef;
bookToCheckOut.book   = bookRef;
```

Figure 3-31 The library program (continued)

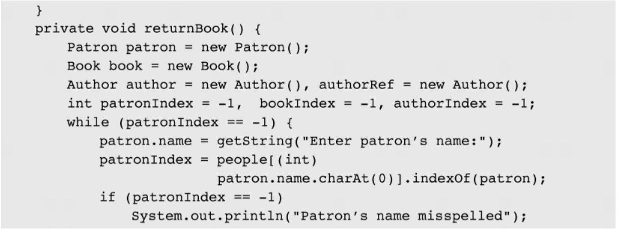
Case Study: A Library (continued)



```
        patronIndex = people[(int)
                      patron.name.charAt(0)].indexOf(patron);
        if (patronIndex == -1) {       // a new patron in the library;
            patron.books.add(bookToCheckOut);
            people[(int) patron.name.charAt(0)].add(patron);
            bookRef.patron = (Patron) people[(int)
patron.name.charAt(0)].getFirst();
        }
        else {
            patronRef = (Patron) people[(int)
                         patron.name.charAt(0)].get(patronIndex);
            patronRef.books.add(bookToCheckOut);
            bookRef.patron = patronRef;
        }
```

Figure 3-31 The library program (continued)

Case Study: A Library (continued)



```
}
private void returnBook() {
    Patron patron = new Patron();
    Book book = new Book();
    Author author = new Author(), authorRef = new Author();
    int patronIndex = -1,  bookIndex = -1, authorIndex = -1;
    while (patronIndex == -1) {
        patron.name = getString("Enter patron’s name:");
        patronIndex = people[(int)
                      patron.name.charAt(0)].indexOf(patron);
        if (patronIndex == -1)
            System.out.println("Patron’s name misspelled");
```

Figure 3-31 The library program (continued)

Project 3

- Implement the case study

Summary

- A linked structure is a collection of nodes storing data and links to other nodes.
- A linked list is a data structure composed of nodes, each node holding some information and a reference to another node in the list.
- A singly linked list is a node that has a link only to its successor in this sequence.
- A circular list is when nodes form a ring: The list is finite and each node has a successor.
- LL Advantages: Insert, remove operations are performed out efficiently.
- LL Disadvantages: Search operation is not performed effectively because of sequential scanning.

Summary (continued)

- How to speed up the searching process in LL?
  - Use a skip list is a variant of the ordered linked list that makes a nonsequential search possible: Nodes in the list contains ordered values. Each node contains an arrays of references to following nodes. When a key is searched, based on this array, a part of the list will be scanned only and other parts are omitted.
  - Self-organizing lists: Re-order the list based on some criteria. Four methods for organizing lists:
    - move-to-front method: The node it has been accessed will be bring to the beginning of the list.
    - transpose (chuyển vị) method: The node it has been accessed will be swap with the previous node.
    - count method: Each node has a field to count the number of accessing and the list will be ordered based on this count descending.
    - ordering method*: The list has a fixed order.*
  - Optimal static ordering - all the data are already ordered by the frequency of their occurrence in the body of data so that the list is used only for searching, not for inserting new items.

Summary (continued)

- A sparse table refers to a table that is populated sparsely by data and most of its cells are empty.
- Linked lists allow easy insertion and deletion of information because such operations have a local impact on the list.
- The advantage of arrays over linked lists is that they allow random accessing.