Object-Oriented Programming

Lecturer: Prof Ronan Reilly
email: Ronan.Reilly@nuim.ie

Thanks to previous lecturers on this course

Dr Rosemary Monahan, NUIM

Prof Brian A. Malloy, Clemson University

Dr Paul Gibson, formerly NUIM

Containers in Java

Based on a hierarchy of data types:

Collection, List, Set, Sorted Set

Map, Sorted Map

Arrays

- Array: Sequence of values of the same type
- Construct array:

```
new double[10]
```

- Store in variable of type `double[]`

```
double[] data = new double[10];
```

- When array is created, all values are initialized depending on array type:
  - Numbers: `0`
  - Boolean: *`false`*
  - Object References: *`null`*

Arrays

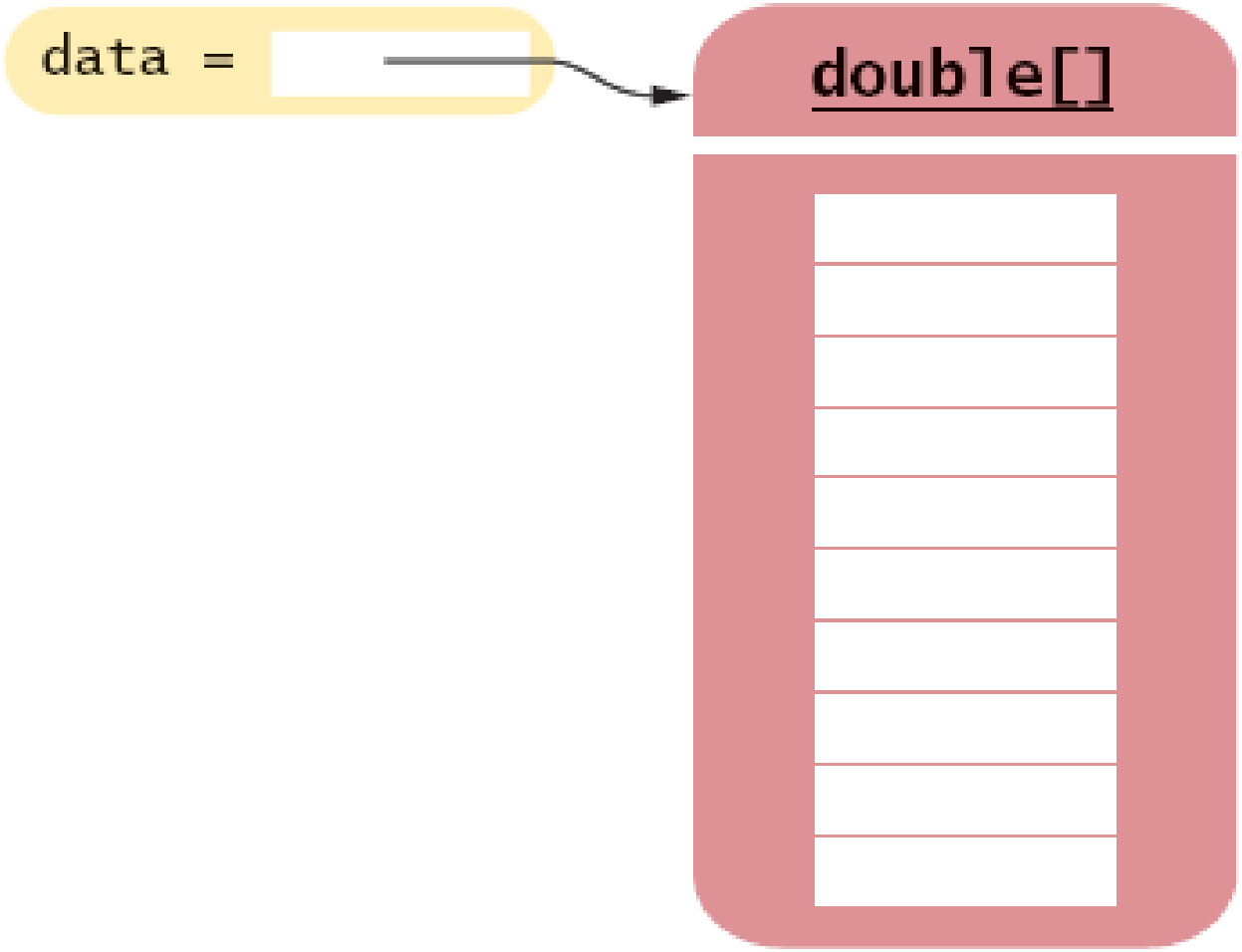

Figure 1 An Array Reference and an Array

Arrays

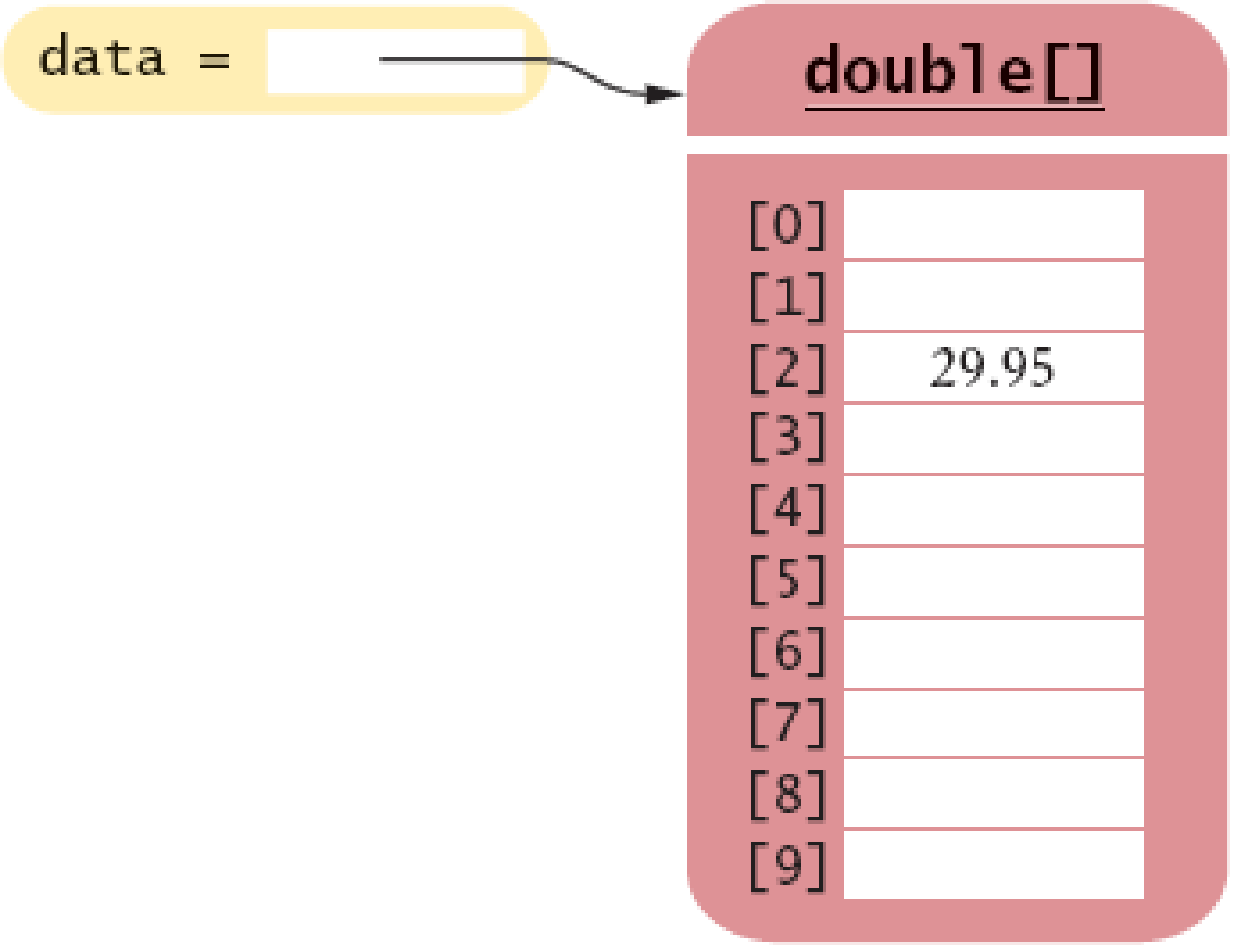

Figure 2 Storing a Value in an Array

Arrays

- Using the value stored:

```
System.out.println("The value of this data item is "
      + data[4]);
```

- Get array length as `data.length` (Not a method!)
- Index values range from `0` to `length - 1`
- Accessing a nonexistent element results in a bounds error

```
double[] data = new double[10];
data[10] = 29.95; // ERROR
```

- Limitation: Arrays have fixed length

Array Lists

- The `ArrayList` class manages a sequence of objects
- Can grow and shrink as needed
- `ArrayList` class supplies methods for many common tasks, such as inserting and removing elements
- The `ArrayList` class is a generic class: `ArrayList<T>` collects objects of type `T`:

```
ArrayList<BankAccount> accounts = new
        ArrayList<BankAccount>();
accounts.add(new BankAccount(1001));
accounts.add(new BankAccount(1015));
accounts.add(new BankAccount(1022));
```

- `size` method yields number of elements

Retrieving Array List Elements

- Use `get` method
- Index starts at 0
-
```
BankAccount anAccount = accounts.get(2); // gets the third element of the array list
```
- Bounds error if index is out of range
- Most common bounds error:

```
int i = accounts.size();
anAccount = accounts.get(i); // Error
//legal index values are 0. . .i-1
```

Adding Elements

- `set` overwrites an existing value

```
BankAccount anAccount = new BankAccount(1729);
accounts.set(2, anAccount);
```

- `add` adds a new value before the index

```
accounts.add(i, a)
```

Continued

The Generalized `for` Loop

- Traverses all elements of a collection:

```
double[] data = . . .;
double sum = 0;
for (double e : data) // You should read this loop as
          "for each e in data"
{
   sum = sum + e;
}
```

- Traditional alternative:

```
double[] data = . . .;
double sum = 0;
for (int i = 0; i < data.length; i++)
{
   double e = data[i];
   sum = sum + e;
}
```

The Generalized `for` Loop

- Works for `ArrayLists` too:

```
ArrayList<BankAccount> accounts = . . . ;
double sum = 0;
for (BankAccount a : accounts)
{
   sum = sum + a.getBalance();
}
```

- Equivalent to the following ordinary `for` loop:

```
double sum = 0;
for (int i = 0; i < accounts.size(); i++)
{
   BankAccount a = accounts.get(i);
   sum = sum + a.getBalance();
}
```

Syntax 7.3 The "for each" Loop

```
for (Type variable : collection)
   statement
```

Example:

```
for (double e : data)
   sum = sum + e;
```

Purpose:

To execute a loop for each element in the collection. In each iteration, the variable is assigned the next element of the collection. Then the statement is executed.

ch07/bank/Bank.java

```
01: import java.util.ArrayList;
02:
03: /**
04:    This bank contains a collection of bank accounts.
05: */
06: public class Bank
07: {
08:    /**
09:       Constructs a bank with no bank accounts.
10:    */
11:    public Bank()
12:    {
13:       accounts = new ArrayList<BankAccount>();
14:    }
15:
16:    /**
17:       Adds an account to this bank.
18:       @param a the account to add
19:    */
20:    public void addAccount(BankAccount a)
21:    {
22:       accounts.add(a);
23:    }
```

Continued

ch07/bank/Bank.java (cont.)

```
24:
25:    /**
26:       Gets the sum of the balances of all accounts in this bank.
27:       @return the sum of the balances
28:    */
29:    public double getTotalBalance()
30:    {
31:       double total = 0;
32:       for (BankAccount a : accounts)
33:       {
34:          total = total + a.getBalance();
35:       }
36:       return total;
37:    }
38:
39:    /**
40:       Counts the number of bank accounts whose balance is at
41:       least a given value.
42:       @param atLeast the balance required to count an account
43:       @return the number of accounts having least the given balance
44:    */
45:    public int count(double atLeast)
46:    {
```

Continued

ch07/bank/Bank.java (cont.)

```
47:       int matches = 0;
48:       for (BankAccount a : accounts)
49:       {
50:          if (a.getBalance() >= atLeast) matches++; // Found a match
51:       }
52:       return matches;
53:    }
54:
55:    /**
56:       Finds a bank account with a given number.
57:       @param accountNumber the number to find
58:       @return the account with the given number, or null if there
59:       is no such account
60:    */
61:    public BankAccount find(int accountNumber)
62:    {
63:       for (BankAccount a : accounts)
64:       {
65:          if (a.getAccountNumber() == accountNumber) // Found a match
66:             return a;
67:       }
68:       return null; // No match in the entire array list
69:    }
70:
```

Continued

ch07/bank/Bank.java (cont.)

```
71:    /**
72:       Gets the bank account with the largest balance.
73:       @return the account with the largest balance, or null if the
74:       bank has no accounts
75:    */
76:    public BankAccount getMaximum()
77:    {
78:       if (accounts.size() == 0) return null;
79:       BankAccount largestYet = accounts.get(0);
80:       for (int i = 1; i < accounts.size(); i++)
81:       {
82:          BankAccount a = accounts.get(i);
83:          if (a.getBalance() > largestYet.getBalance())
84:             largestYet = a;
85:       }
86:       return largestYet;
87:    }
88:
89:    private ArrayList<BankAccount> accounts;
90: }
```

ch07/bankBankTester.java

```
01: /**
02:    This program tests the Bank class.
03: */
04: public class BankTester
05: {
06:    public static void main(String[] args)
07:    {
08:       Bank firstBankOfJava = new Bank();
09:       firstBankOfJava.addAccount(new BankAccount(1001, 20000));
10:       firstBankOfJava.addAccount(new BankAccount(1015, 10000));
11:       firstBankOfJava.addAccount(new BankAccount(1729, 15000));
12:
13:       double threshold = 15000;
14:       int c = firstBankOfJava.count(threshold);
15:       System.out.println("Count: " + c);
16:       System.out.println("Expected: 2");
17:
18:       int accountNumber = 1015;
19:       BankAccount a = firstBankOfJava.find(accountNumber);
20:       if (a == null)
```

Continued

ch07/bankBankTester.java (cont.)

```
21:             System.out.println("No matching account");
22:          else
23:             System.out.println("Balance of matching account: " +
                        a.getBalance());
24:          System.out.println("Expected: 10000");
25:
26:          BankAccount max = firstBankOfJava.getMaximum();
27:          System.out.println("Account with largest balance: "
28:                + max.getAccountNumber());
29:          System.out.println("Expected: 1001");
30:       }
31: }
```

Output:

```
Count: 2
Expected: 2
Balance of matching account: 10000.0
Expected: 10000
Account with largest balance: 1001
Expected: 1001
```

Java Containers

- Map is a base class which stores <key, value> pairs with no duplicate keys allowed.
- Sorted Map Inherits from Map. SortedMap objects are map objects that store the pairs in key-sorted order
- Collection considers a container to be a group of objects, without any assumptions about the uniqueness of objects in the container. It is a base class declaring methods common to types List and Set. These include add, clear, IsEmpty, iterator, remove, removeAll, toArray (which constructs an array version of any container).
- Set inherits form Collection – all Set elements must be unique.
- SortedSet inherits from Set and stores elements in sorted order.

Note: These are all Java Interfaces, so they only contain method declarations

Example: Lists

```
import java.util.*;

class ListOps {
    public static void main( String[] args )
    {
        List animals = new ArrayList();
        animals.add( "cheetah" );
        animals.add( "lion" );
        animals.add( "cat" );
        animals.add( "fox" );
        animals.add( "cat" );          //duplicate cat
        System.out.println( animals );  //cheetah, lion, cat, fox, cat

        animals.remove( "lion" );
        System.out.println( animals );  //cheetah, cat, fox, cat

        animals.add( 0, "lion" );
        System.out.println( animals );  //lion, cheetah, cat, fox, cat
```

```
animals.add( 3, "racoon" );
System.out.println( animals );  //lion, cheetah, cat, racoon, fox, cat

animals.remove(3);
System.out.println( animals );  //lion, cheetah, cat, fox, cat

Collections.sort( animals );
System.out.println( animals );  //cat, cat, cheetah,fox, lion

List pets = new LinkedList();
pets.add( "cat" ); pets.add( "dog" ); pets.add( "bird" );
System.out.println( pets );     //cat, dog, bird

animals.addAll( 3, pets );
System.out.println( animals );  //cat, cat, cheetah, cat, dog, bird, fox, lion

ListIterator iter = animals.listIterator(); /* ListIterators can move in 2 directions
    whereas iterators only move in one direction*/
while ( iter.hasNext() ) {
        System.out.println( iter.next()  );
    }
  }
}
```

Using Linked Lists

- A linked list consists of a number of nodes, each of which has a reference to the next node
- 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

Inserting an Element into a Linked List

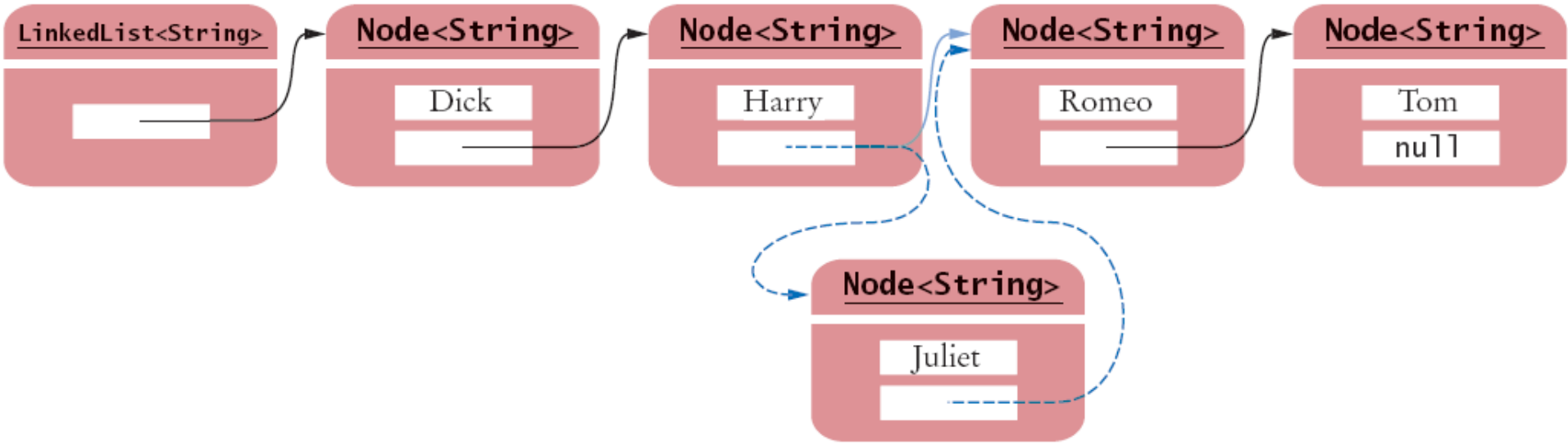

Figure 1 Inserting an Element into a Linked List

Java's `LinkedList` class

- Generic class
  - *Specify type of elements in angle brackets:* `LinkedList<Product>`
- Package: `java.util`
- Easy access to first and last elements with methods

  void addFirst(E obj)
  void addLast(E obj)
  E getFirst()
  E getLast()
  E removeFirst()
  E removeLast()

List Iterator

- `ListIterator` type
- Gives access to elements inside a linked list
- Encapsulates a position anywhere inside the linked list
- Protects the linked list while giving access

A List Iterator

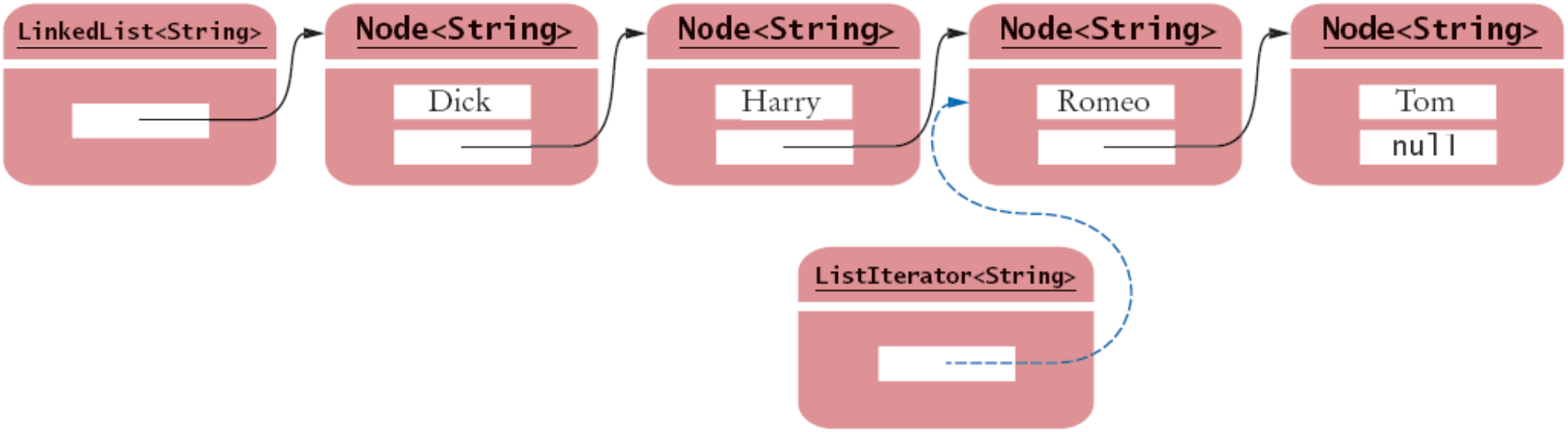

Figure 2 A List Iterator

A Conceptual View of the List Iterator

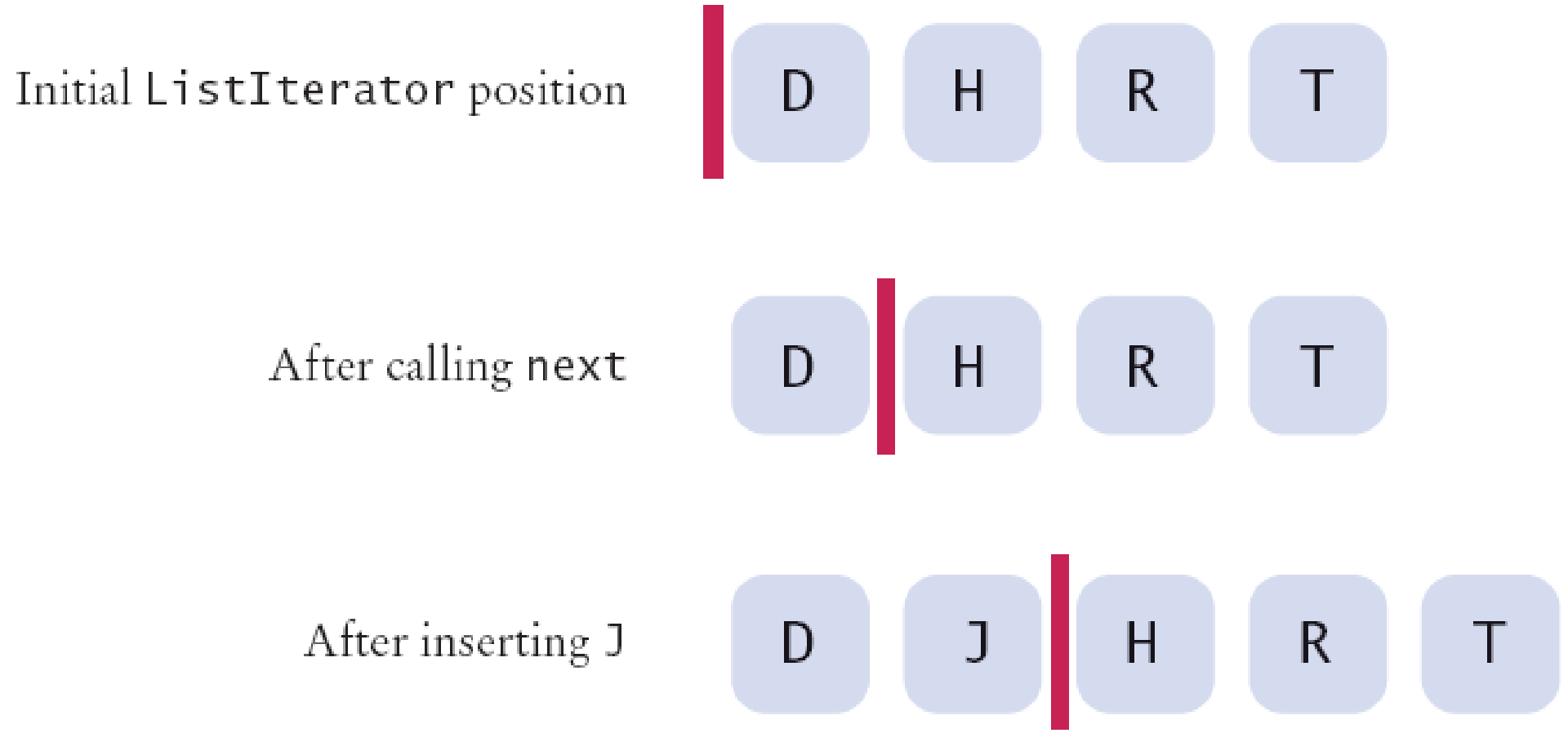

Figure 3 A Conceptual View of the List Iterator

List Iterator

- Think of an iterator as pointing between two elements
  - *Analogy: like the cursor in a word processor points between two characters*
- The `listIterator` method of the `LinkedList` class gets a list iterator

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator =
      employeeNames.listIterator();
```

List Iterator

- Initially, the iterator points before the first element
- The `next` method moves the iterator

```
iterator.next();
```

- `next` throws a `NoSuchElementException` if you are already past the end of the list
- `hasNext` returns true if there is a next element

```
if (iterator.hasNext())
iterator.next();
```

List Iterator

- The `next` method returns the element that the iterator is passing

```
while iterator.hasNext()
{
   String name = iterator.next();
   Do something with name
}
```

- Shorthand:

```
for (String name : employeeNames)
{
   Do something with name
}
```

Behind the scenes, the for loop uses an iterator to visit all list elements

List Iterator

- `LinkedList` is a *doubly linked list*
  - *Class stores two links:*
    - One to the next element, and
    - One to the previous element
- To move the list position backwards, use:
  - *`hasPrevious`*
  - *`previous`*

Adding and Removing from a LinkedList

- The `add` method:
  - *Adds an object after the iterator*
  - *Moves the iterator position past the new element*

```
iterator.add("Juliet");
```

Adding and Removing from a LinkedList

- The `remove` method
  - *Removes and*
  - *Returns the object that was returned by the last call to `next` or `previous`*

```
 //Remove all names that fulfill a certain condition
while (iterator.hasNext())
{
String name = iterator.next();
if (name fulfills condition)
   iterator.remove(); }
```

- Be careful when calling `remove`:
  - *It can be called only once after calling `next` or `previous`*
  - *You cannot call it immediately after a call to `add`*
  - *If you call it improperly, it throws an `IllegalStateException`*

Sample Program

- `ListTester` is a sample program that
  - *Inserts strings into a list*
  - *Iterates through the list, adding and removing elements*
  - *Prints the list*

ch15/uselist/ListTester.java

```
01: import java.util.LinkedList;
02: import java.util.ListIterator;
03:
04: /**
05:    A program that tests the LinkedList class
06: */
07: public class ListTester
08: {
09:    public static void main(String[] args)
10:    {
11:       LinkedList<String> staff = new LinkedList<String>();
12:       staff.addLast("Dick");
13:       staff.addLast("Harry");
14:       staff.addLast("Romeo");
15:       staff.addLast("Tom");
16:
17:       // | in the comments indicates the iterator position
18:
19:       ListIterator<String> iterator
20:             = staff.listIterator(); // |DHRT
21:       iterator.next(); // D|HRT
22:       iterator.next(); // DH|RT
```

Continued

ch15/uselist/ListTester.java (cont.)

```
23:
24:       // Add more elements after second element
25:
26:       iterator.add("Juliet"); // DHJ|RT
27:       iterator.add("Nina"); // DHJN|RT
28:
29:       iterator.next(); // DHJNR|T
30:
31:       // Remove last traversed element
32:
33:       iterator.remove(); // DHJN|T
34:
35:       // Print all elements
36:
37:       for (String name : staff)
38:          System.out.print(iterator.next() + " ");
39:       System.out.println();
40:       System.out.println("Expected: Dick Harry Juliet Nina Tom");
41:    }
42: }
```

ch15/uselist/ListTester.java (cont.)

Output:

```
Dick Harry Juliet Nina Tom
Expected: Dick Harry Juliet Nina Tom
```

Sets

- Set: unordered collection of distinct elements
- Elements can be added, located, and removed
- Sets don't have duplicates

Sets

- We could use a linked list to implement a set
  - *Adding, removing, and containment testing would be relatively slow*
- There are data structures that can handle these operations much more quickly
  - *Hash tables*
  - *Trees*
- Standard Java library provides set implementations based on both data structures
  - `HashSet`
  - `TreeSet`
- Both of these data structures implement the `Set` interface

Iterator

- Use an iterator to visit all elements in a set
- A set iterator does not visit the elements in the order in which they were inserted
- An element can not be added to a set at an iterator position
- A set element can be removed at an iterator position

Code for Creating and Using a Hash Set

- `//Creating a hash set`
  `Set<String> names = new HashSet<String>();`
- `//Adding an element`
  `names.add("Romeo");`
- `//Removing an element`
  `names.remove("Juliet");`
- `//Is element in set`
  `if (names.contains("Juliet") { . . .}`

Listing All Elements with an Iterator

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
   String name = iter.next();
   Do something with name
}

// Or, using the "for each" loop
for (String name : names)
{
   Do something with name
}
```

ch16/set/SetDemo.java

```
01: import java.util.HashSet;
02: import java.util.Scanner;
03: import java.util.Set;
04:
05:
06: /**
07:    This program demonstrates a set of strings. The user
08:    can add and remove strings.
09: */
10: public class SetDemo
11: {
12:    public static void main(String[] args)
13:    {
14:       Set<String> names = new HashSet<String>();
15:       Scanner in = new Scanner(System.in);
16:
17:       boolean done = false;
18:       while (!done)
19:       {
20:          System.out.print("Add name, Q when done: ");
21:          String input = in.next();
```

Continued

ch16/set/SetDemo.java (cont.)

```
22:             if (input.equalsIgnoreCase("Q"))
23:                done = true;
24:             else
25:             {
26:                names.add(input);
27:                print(names);
28:             }
29:          }
30:
31:          done = false;
32:          while (!done)
33:          {
34:             System.out.print("Remove name, Q when done: ");
35:             String input = in.next();
36:             if (input.equalsIgnoreCase("Q"))
37:                done = true;
38:             else
39:             {
40:                names.remove(input);
41:                print(names);
42:             }
43:          }
44:       }
```

Continued

ch16/set/SetDemo.java (cont.)

```
45:
46:    /**
47:       Prints the contents of a set of strings.
48:       @param s a set of strings
49:    */
50:    private static void print(Set<String> s)
51:    {
52:       System.out.print("{ ");
53:       for (String element : s)
54:       {
55:          System.out.print(element);
56:          System.out.print(" ");
57:       }
58:       System.out.println("}");
59:    }
60: }
61:
62:
```

Continued

ch16/set/SetDemo.java (cont.)

Output:

```
Add name, Q when done: Dick
{ Dick }
Add name, Q when done: Tom
{ Tom Dick }
Add name, Q when done: Harry
{ Harry Tom Dick }
Add name, Q when done: Tom
{ Harry Tom Dick }
Add name, Q when done: Q
Remove name, Q when done: Tom
{ Harry Dick }
Remove name, Q when done: Jerry
{ Harry Dick }
Remove name, Q when done: Q
```

Self Check 16.1

Arrays and lists remember the order in which you added elements; sets do not. Why would you want to use a set instead of an array or list?

Answer: Efficient set implementations can quickly test whether a given element is a member of the set.

Self Check 16.2

Why are set iterators different from list iterators?

Answer: Sets do not have an ordering, so it doesn't make sense to add an element at a particular iterator position, or to traverse a set backwards.

Maps

- A map keeps associations between key and value objects
- Mathematically speaking, a map is a function from one set, the *key set*, to another set, the *value set*
- Every key in a map has a unique value
- A value may be associated with several keys
- Classes that implement the `Map` interface
  - *`HashMap`*
  - *`TreeMap`*

An Example of a Map

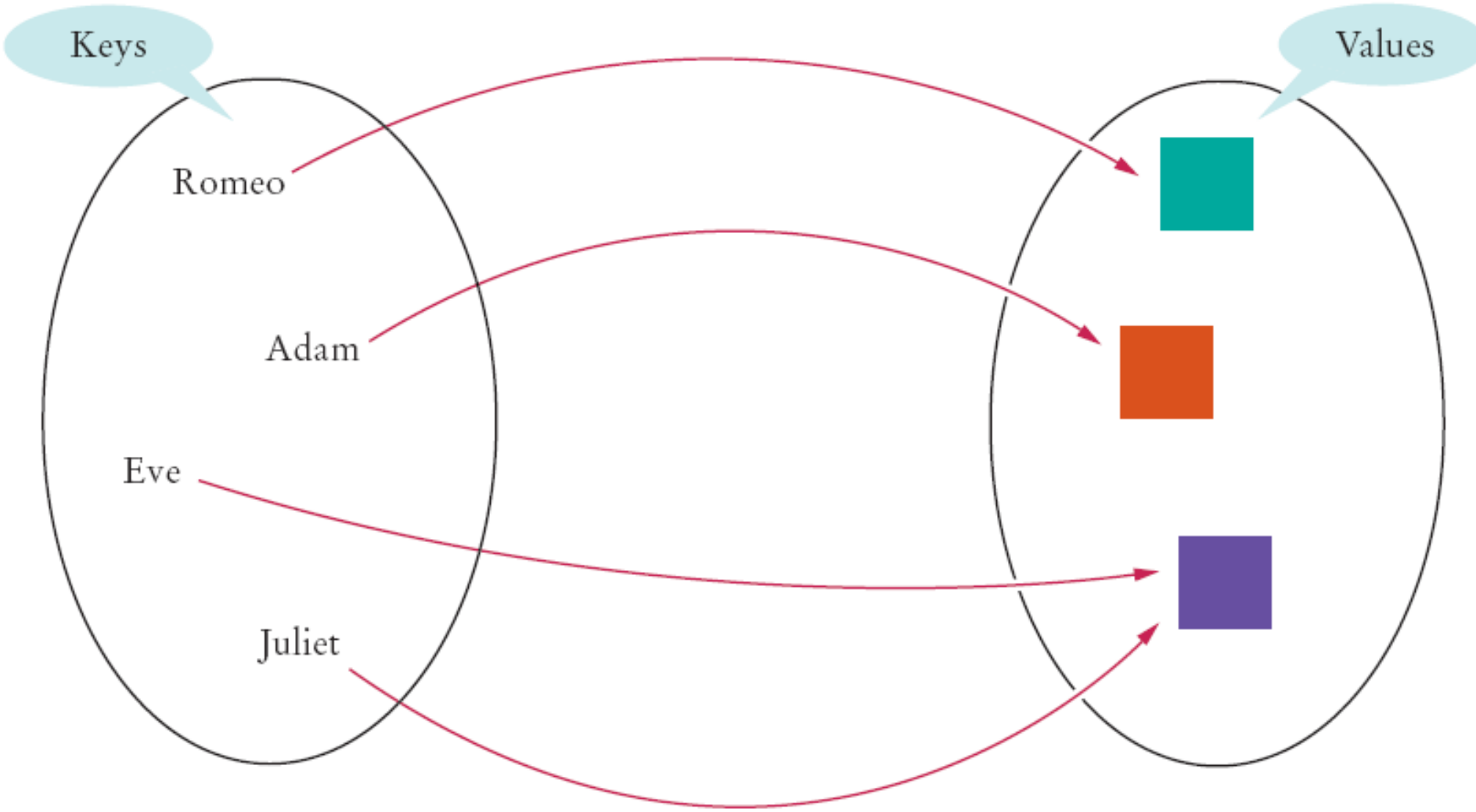

Figure 3 A Map

Code for Creating and Using a HashMap

- ```
  //Creating a HashMap
  Map<String, Color> favoriteColors = new HashMap<String,
        Color>();
  ```
- ```
  //Adding an association
  favoriteColors.put("Juliet", Color.PINK);
  ```
- ```
  //Changing an existing association
  favoriteColor.put("Juliet",Color.RED);
  ```
- ```
  //Getting the value associated with a key
  Color julietsFavoriteColor =
        favoriteColors.get("Juliet");
  ```
- ```
  //Removing a key and its associated value
  favoriteColors.remove("Juliet");
  ```
  ```

Printing Key/Value Pairs

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
   Color value = m.get(key);
   System.out.println(key + "->" + value);
}
```

ch16/map/MapDemo.java

```
01: import java.awt.Color;
02: import java.util.HashMap;
03: import java.util.Map;
04: import java.util.Set;
05:
06: /**
07:    This program demonstrates a map that maps names to colors.
08: */
09: public class MapDemo
10: {
11:    public static void main(String[] args)
12:    {
13:       Map<String, Color> favoriteColors
14:             = new HashMap<String, Color>();
15:       favoriteColors.put("Juliet", Color.PINK);
16:       favoriteColors.put("Romeo", Color.GREEN);
17:       favoriteColors.put("Adam", Color.BLUE);
18:       favoriteColors.put("Eve", Color.PINK);
19:
```

Continued

ch16/map/MapDemo.java (cont.)

```
20:         Set<String> keySet = favoriteColors.keySet();
21:         for (String key : keySet)
22:         {
23:            Color value = favoriteColors.get(key);
24:            System.out.println(key + "->" + value);
25:         }
26:     }
27: }
```

Continued

ch16/map/MapDemo.java (cont.)

Output:

```
Romeo->java.awt.Color[r=0,g=255,b=0]
Eve->java.awt.Color[r=255,g=175,b=175]
Adam->java.awt.Color[r=0,g=0,b=255]
Juliet->java.awt.Color[r=255,g=175,b=175]
```

Java Containers:Vectors

- The Vector class dates back to early Java releases.
- It has now been retro-fitted to implement the List interface to enable older Java code to run on newer platforms.
- Vectors have all of the nmethods listed in the List interface …
  - Plus a few more e.g. addElement, size, elementAt(i)

- Examples in the folder called Java Container Examples

Java Containers Algorithms

- The java.util.Collections class provides some predefined algorithms for activities such as
  - sorting, searching, copying, filling, finding maximum values in a container, finding minimum values in a container, reversing the contents of a container, shuffling the container contents etc…

Java API at
http://java.sun.com/j2se/1.5.0/docs/api/