Recursive Data Structure

Dr. Youna Jung
Northeastern University
yo.jung@northeastern.edu

RECURSION

Computing Factorial

- Mathematic notation:

```
0! = 1
n! = n * (n-1)!,   n > 0
```

- Function:

```
factorial(0) = 1;
factorial(n) = n * factorial(n-1),  n > 0
```

Calculating Factorial

- Using **Loop**

```
import java.util.Scanner;
public class JavaExample {

    public static void main(String[] args) {

        //We will find the factorial of this number
        int number;
        System.out.println("Enter the number: ");
        Scanner scanner = new Scanner(System.in);
        number = scanner.nextInt();
        scanner.close();
        long fact = 1;
        int i = 1;
        while(i<=number)
        {
            fact = fact * i;
            i++;
        }
        System.out.println("Factorial of "+number+" is: "+fact);
    }
}
```

- Using **Recursion**

```
public static long factorial(int n) {
  if (n == 0) // Base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call
}
```

[ComputeFactorial]

Computing Factorial

factorial(4)

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

Computing Factorial

factorial(4) = 4 * factorial(3)

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

= 4 * 3 * (2 * (1 * factorial(0)))

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))
= 4 * 3 * (2 * 1)

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

= 4 * 3 * (2 * (1 * factorial(0)))

= 4 * 3 * (2 * (1 * 1)))

= 4 * 3 * (2 * 1)

= 4 * 3 * 2

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * (3 * factorial(2))
= 4 * (3 * (2 * factorial(1)))
= 4 * (3 * (2 * (1 * factorial(0))))
= 4 * (3 * (2 * (1 * 1))))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * (6)

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * (3 * factorial(2))
= 4 * (3 * (2 * factorial(1)))
= 4 * (3 * (2 * (1 * factorial(0))))
= 4 * (3 * (2 * (1 * 1))))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * (6)
= 24

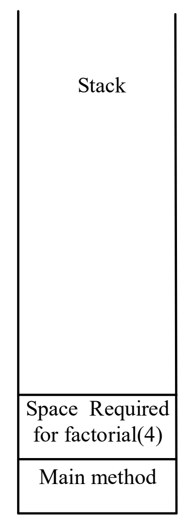
animation

Trace Recursive factorial

Executes factorial(4)

factorial(4)

```
public static long factorial(int n) {
  if (n == 0) // Base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call
}
```



animation

Trace Recursive factorial

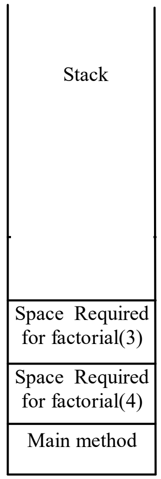
```
public static long factorial(int n) {
  if (n == 0) // Base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call
}
```

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Executes factorial(3)



Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

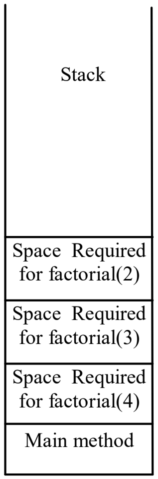
return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

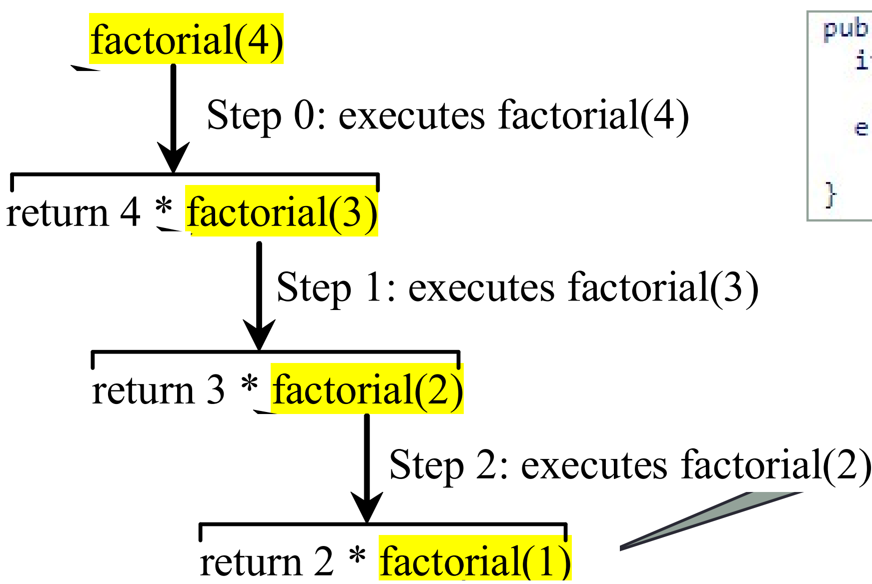
```
public static long factorial(int n) {
  if (n == 0) // Base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call
}
```

Executes factorial(2)



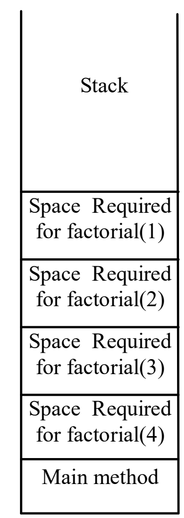
animation

Trace Recursive factorial

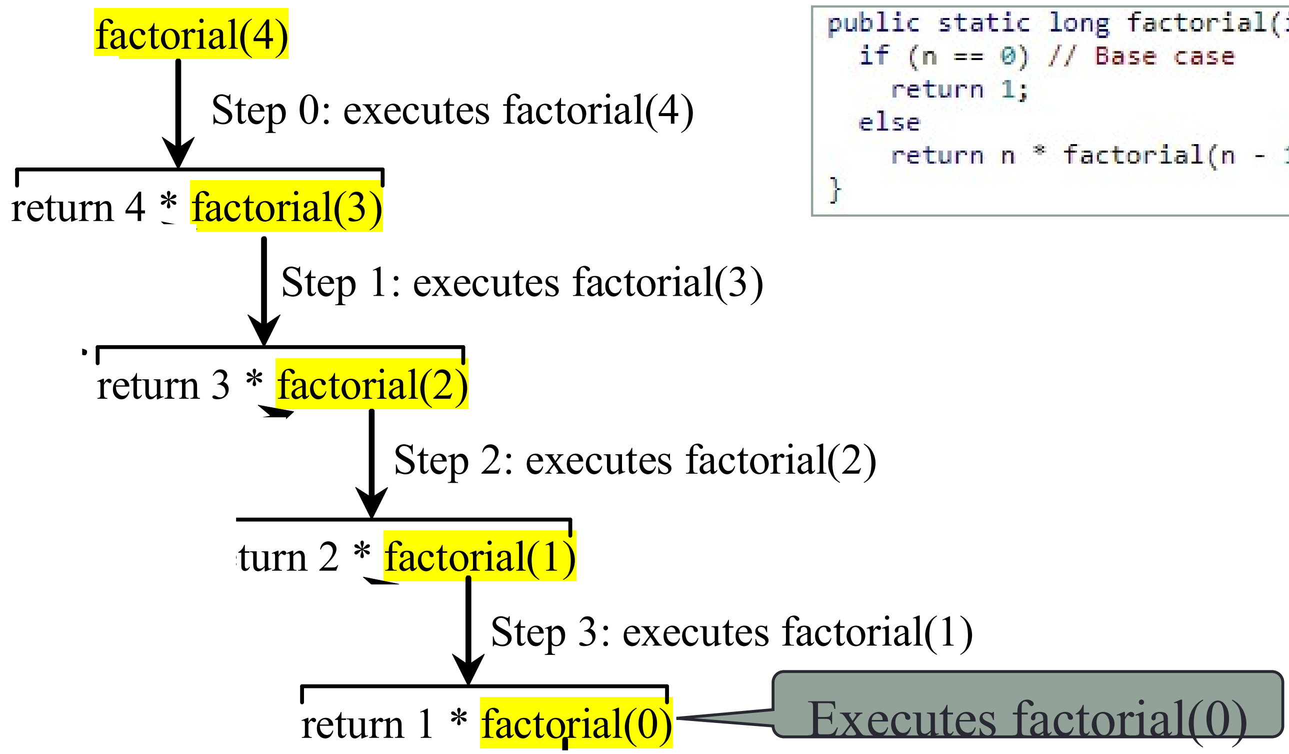



```
public static long factorial(int n) {
  if (n == 0) // Base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call
}
```

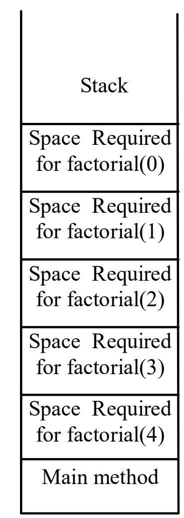
Executes factorial(1)



Trace Recursive factorial




```
public static long factorial(int n) {
  if (n == 0) // Base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call
}
```



animation

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

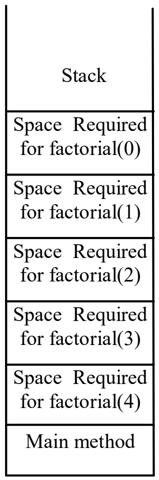
return 1 * factorial(0)

Step 4: executes factorial(0)

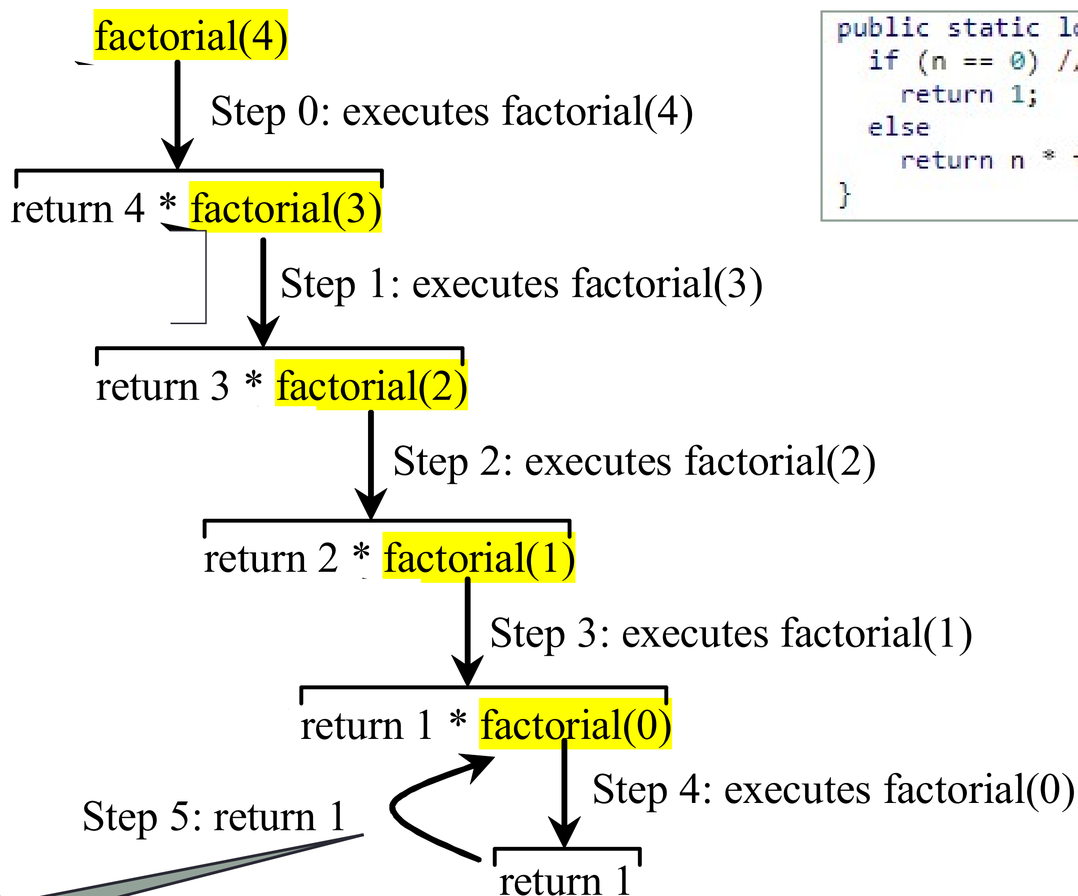
return 1

returns 1

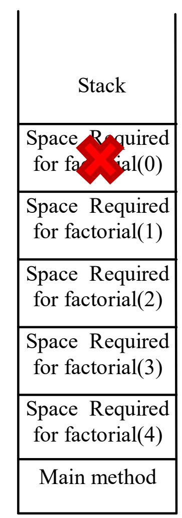
```
public static long factorial(int n) {
  if (n == 0) // Base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call
}
```



Trace Recursive factorial

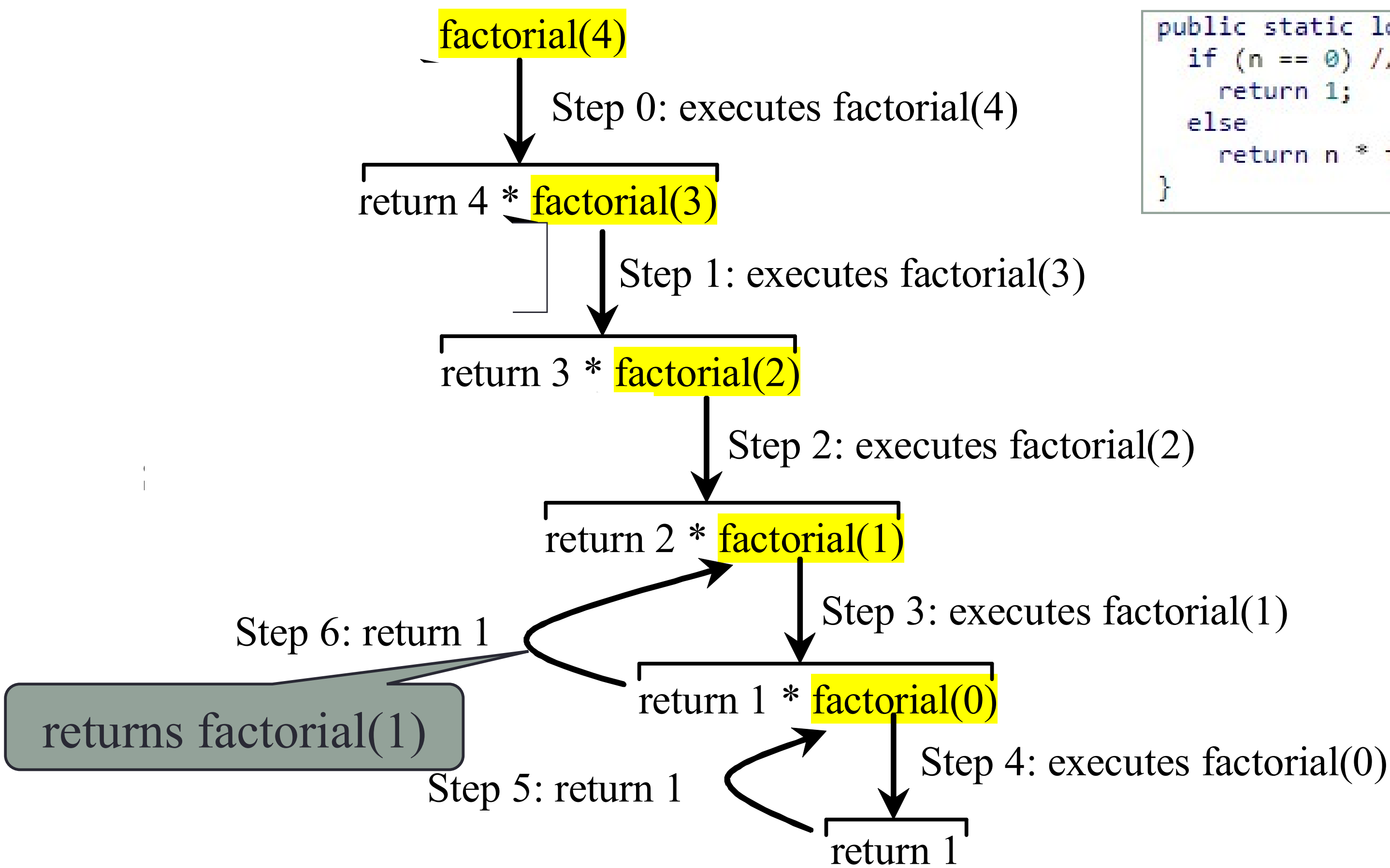



```
public static long factorial(int n) {
  if (n == 0) // Base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call
}
```

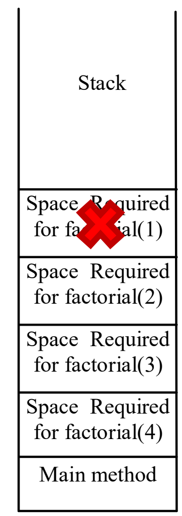


returns factorial(0)

Trace Recursive factorial

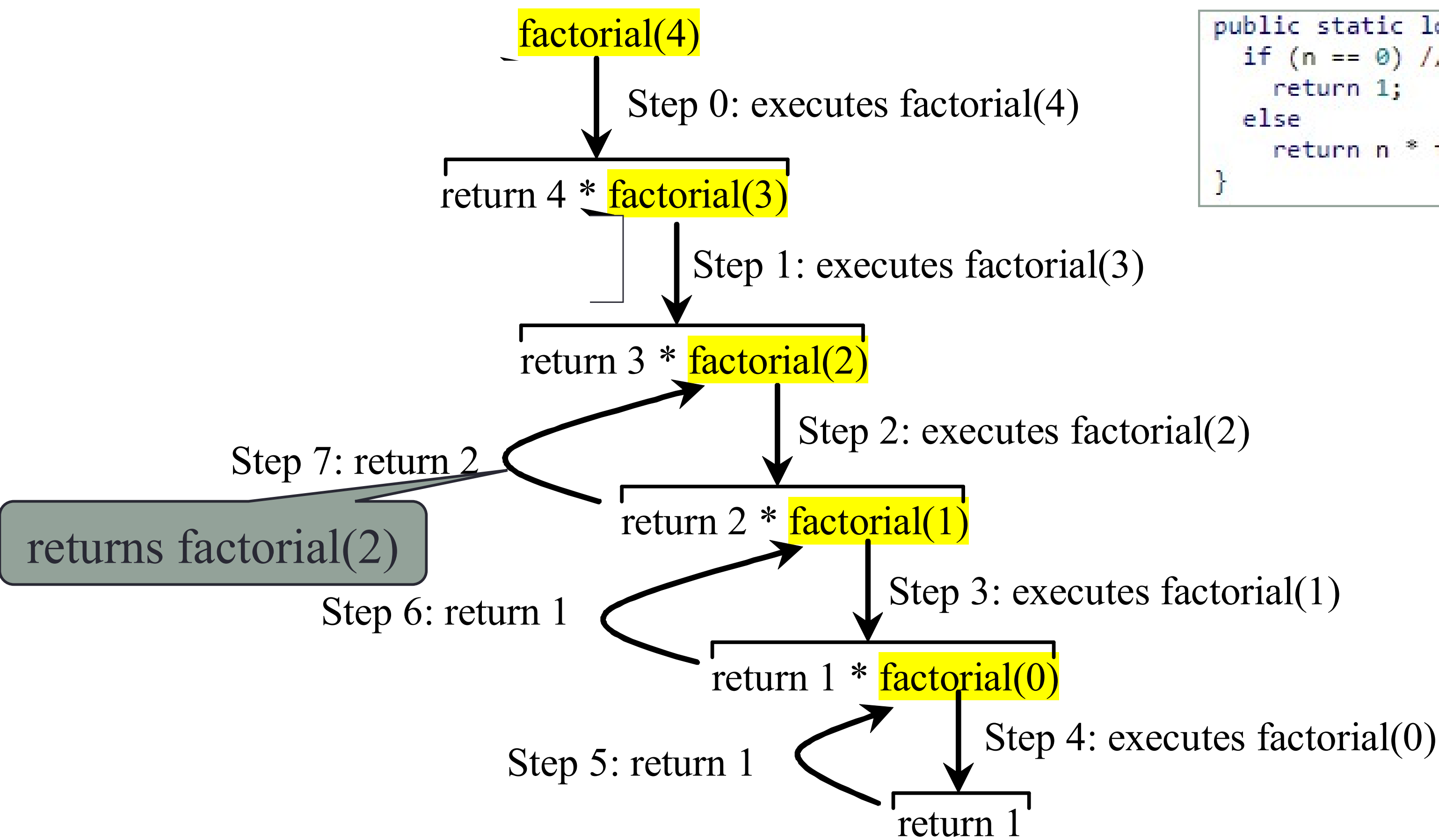



```
public static long factorial(int n) {
  if (n == 0) // Base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call
}
```

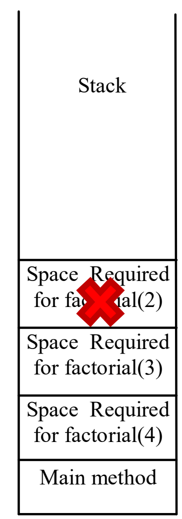


animation

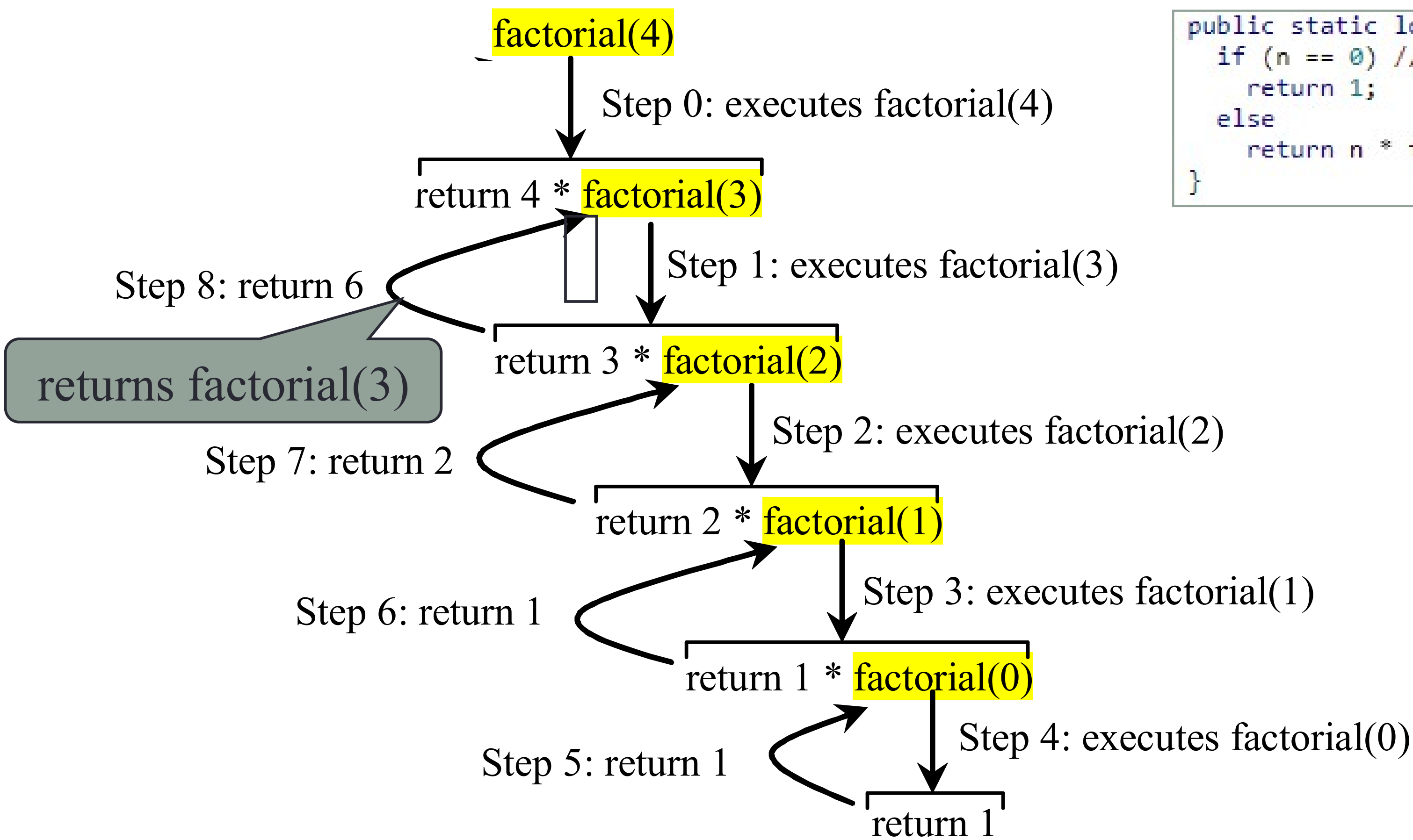
Trace Recursive factorial



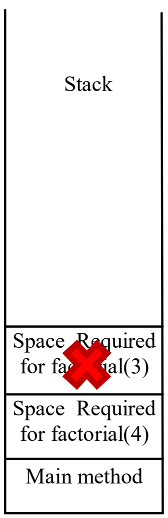

```
public static long factorial(int n) {
  if (n == 0) // Base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call
}
```



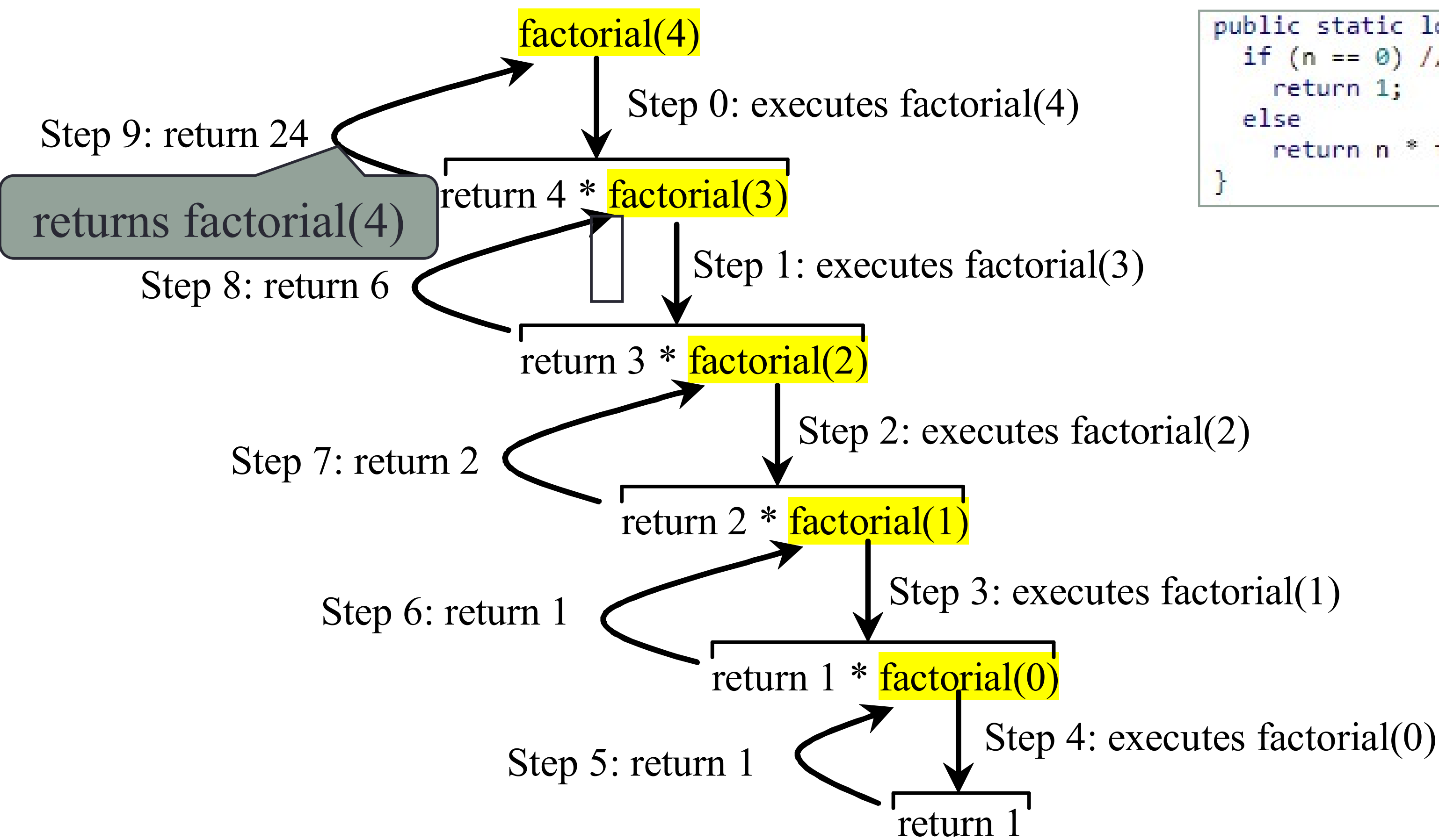
Trace Recursive factorial



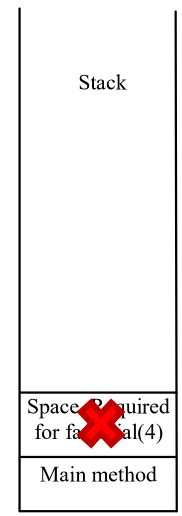

```
public static long factorial(int n) {
  if (n == 0) // Base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call
}
```



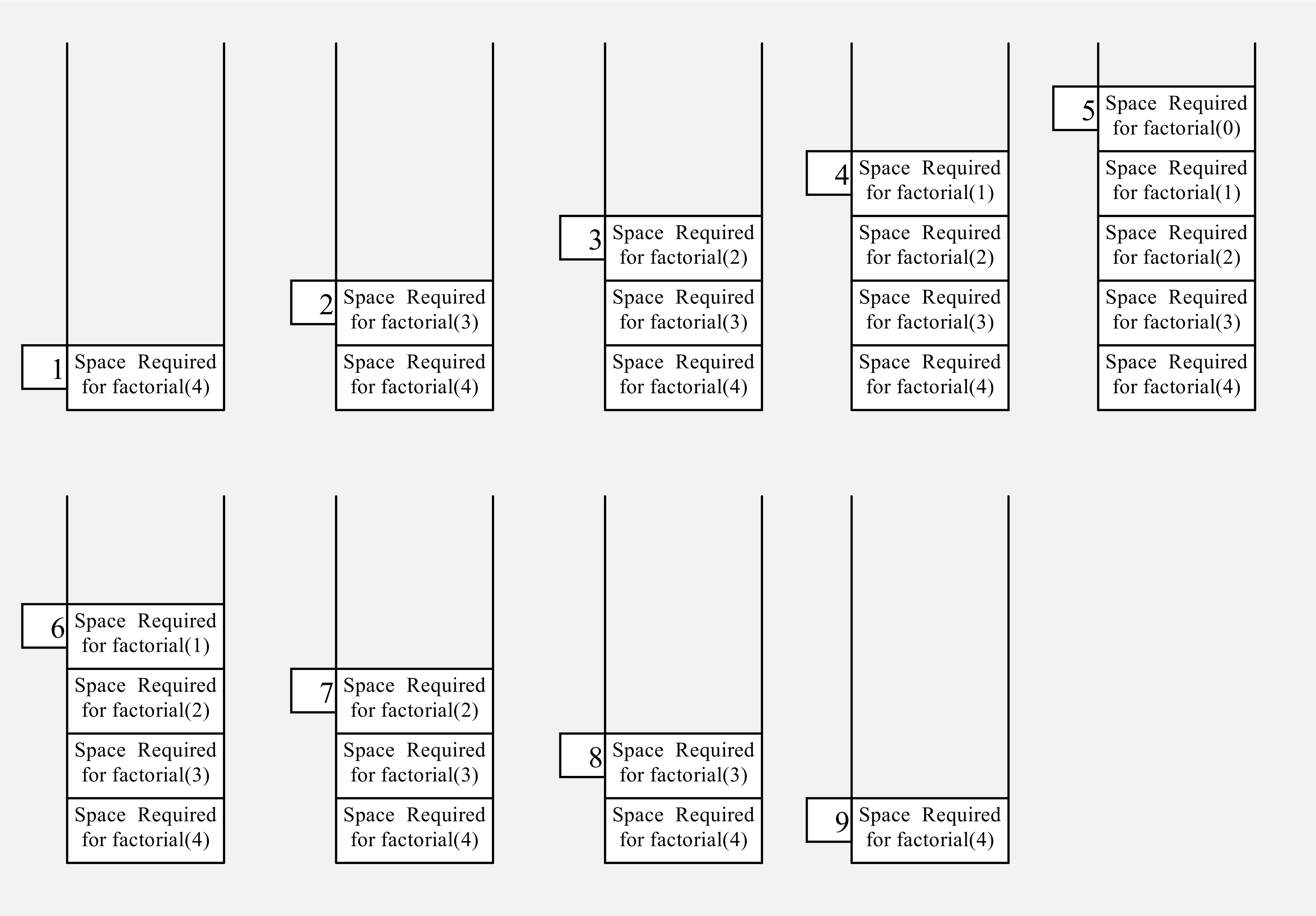
Trace Recursive factorial




```
public static long factorial(int n) {
  if (n == 0) // Base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call
}
```



`factorial(4)` Stack Trace



Practice

- Write two java programs calculating Factorial
  - ✓ One using a loop
  - ✓ Another using recursion.

Fibonacci Numbers

- Math: $fib(0) = 0$; $fib(1) = 1$;

  $fib(index) = fib(index - 1) + fib(index - 2)$, index >=2

```
Fibonacci series: 0 1 1 2 3 5 8 13 21 34 55 89…
         indices: 0 1 2 3 4 5 6 7  8  9  10 11
```

- Function:

  $f(0) = 0$, $f(1) = 1$

  $f(n) = f(n-1) + f(n-2)$;

  - Ex) $fib(3) = fib(2) + fib(1)$

    $= (fib(1) + fib(0)) + fib(1)$

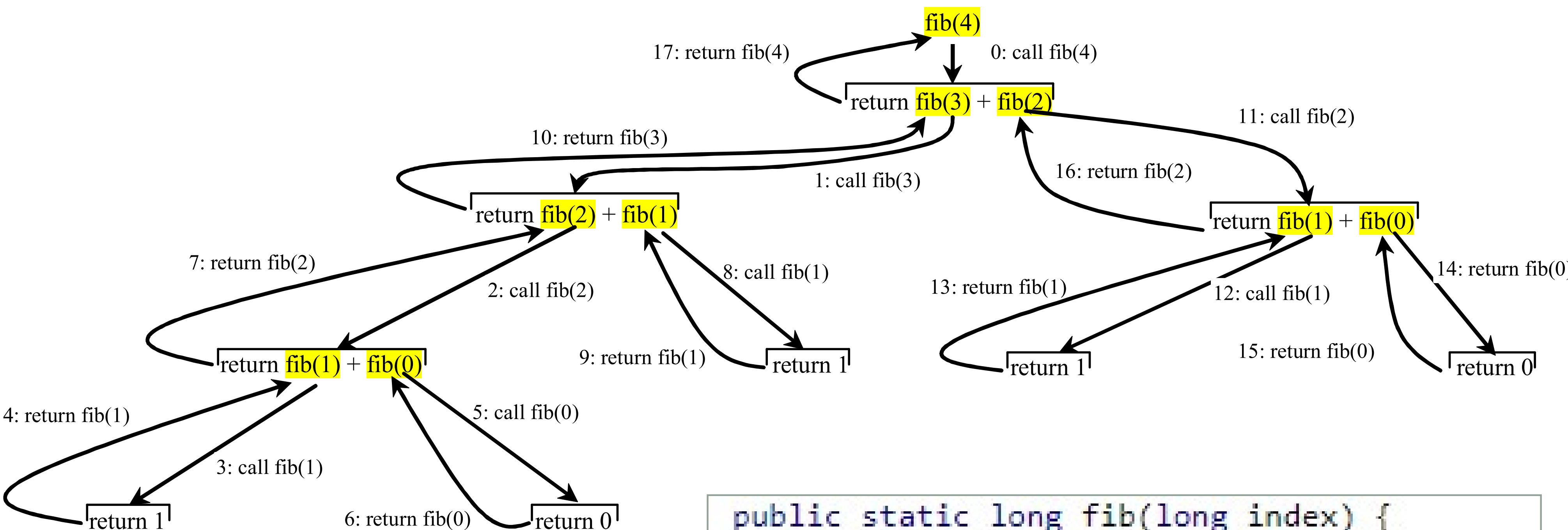
    $= (1 + 0) + fib(1)$

    $= 1 + fib(1) = 1 + 1 = 2$

```java
public static long fib(long index) {
  if (index == 0) // Base case
    return 0;
  else if (index == 1) // Base case
    return 1;
  else  // Reduction and recursive calls
    return fib(index - 1) + fib(index - 2);
}
```

ComputeFibonacci

Fibonnaci Numbers




```java
public static long fib(long index) {
  if (index == 0) // Base case
    return 0;
  else if (index == 1) // Base case
    return 1;
  else  // Reduction and recursive calls
    return fib(index - 1) + fib(index - 2);
}
```

Characteristics of Recursion

- **One** or **more base cases** (the simplest case) are used to **stop** recursion.
  - ✓ The method is implemented using a **conditional** statement that **leads** to **base** cases.

- **Every** recursive **call reduces** the original **problem**, bringing it increasingly **closer** to a **base** case until it becomes that case.

Infinite Recursion

- If 1) recursion **does not reduce the problem** in a manner that allows it to eventually converge into the base case OR 2) a **base** case is **not specified**, infinite recursion can occur

```
Public static long factorial (int n) {

      Return n * factorial(n-1);
}
```

StackOverflowError

Problem Solving Using Recursion

- Let us consider a simple problem of printing a message for *n* times.
  - ✓ The **base case** for the problem is ***n==0***.
  - ✓ You can break the problem into 2 sub-problems
    1) **one** is to **print** the message **one** time
    2) the **other** is to **print** the message for **n-1** times.

- Recursive solution: `nPrintln("Welcome", 5);`

```
public static void nPrintln(String message, int times) {
  if (times >= 1) {
      System.out.println(message);
      nPrintln(message, times - 1);
  } // The base case is times == 0
}
```

Recursive Palindrome Solution

- A string is a **palindrome** if it is reads the **same** from the **left** and from the **right** (“mom” and “dad”)
- **Recursive** solution
  1) Check whether the **first** and the **last** character of the string are **equal**
  2) **Ignore** the **two end** characters & **check** whether the **rest** of the substring is a palindrome

```
public static boolean isPalindrome(String s) {
  if (s.length() <= 1) // Base #1
      return true;
  else if (s.charAt(0) != s.charAt(s.length() - 1)) // Base #2
      return false;
  else
      return isPalindrome(s.substring(1, s.length()-1));   }
```

Recursive Binary Search

- **Base case**
  1. If there is a **match** **Base case #1**
  2. When the search is **exhausted** **Base case #2**
- **Potential cases**
  1) If the key is **less than** the **middle** element
     - ➔ **recursively** search the key in the **first half**
  2) If the key is equal to the middle element, the search ends with a match. **Base case #1**
  3) If the key is **greater than** the **middle** element
     - ➔ **recursively** search the key in the **second half** of the array.

Practice

- Write two java programs performs **Binary Search**
  1) One using a **loop**
  2) Another using **recursion**.

Note: You must submit 1) **2 java files** and 2) **screenshot of test results for both programs**.

```java
public class RecursiveBinarySearch {
  public static int recursiveBinarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;
    return recursiveBinarySearch(list, key, low, high);
  }

  private static int recursiveBinarySearch(int[] list, int key,
      int low, int high) {

    if (low > high) //Base case 2
      return -low - 1;

    int mid = (low + high) / 2;
    if (key < list[mid])
      return recursiveBinarySearch(list, key, low, mid - 1);
    else if (key == list[mid])
      return mid; //Base case 1
    else
      return recursiveBinarySearch(list, key, mid + 1, high);
  }
}
```

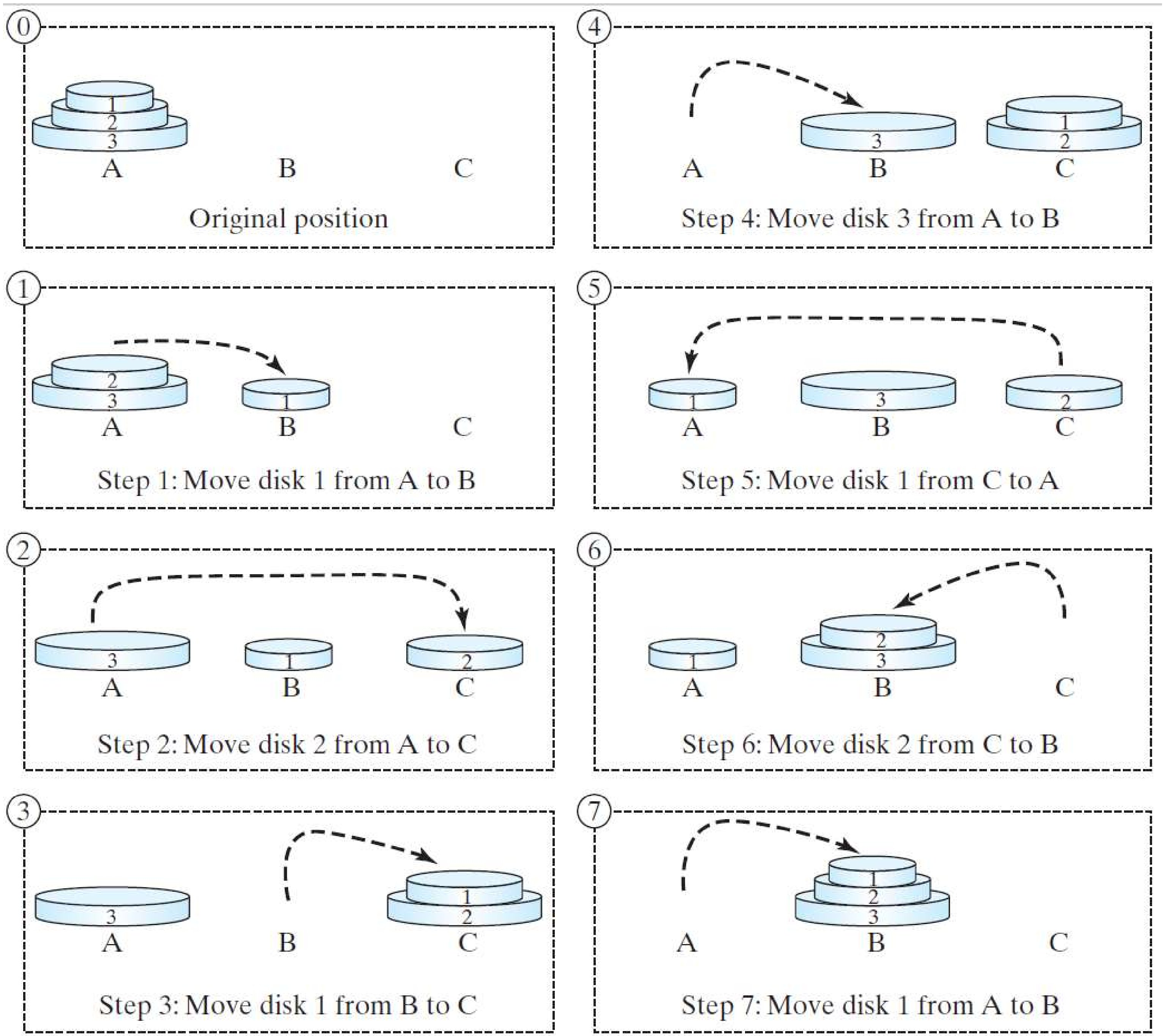
Tower of Hanoi

- There are ***n*** disks labeled 1, 2, 3, . . ., *n*, and three towers labeled **A** (from), **B**(to), and **C**(aux).

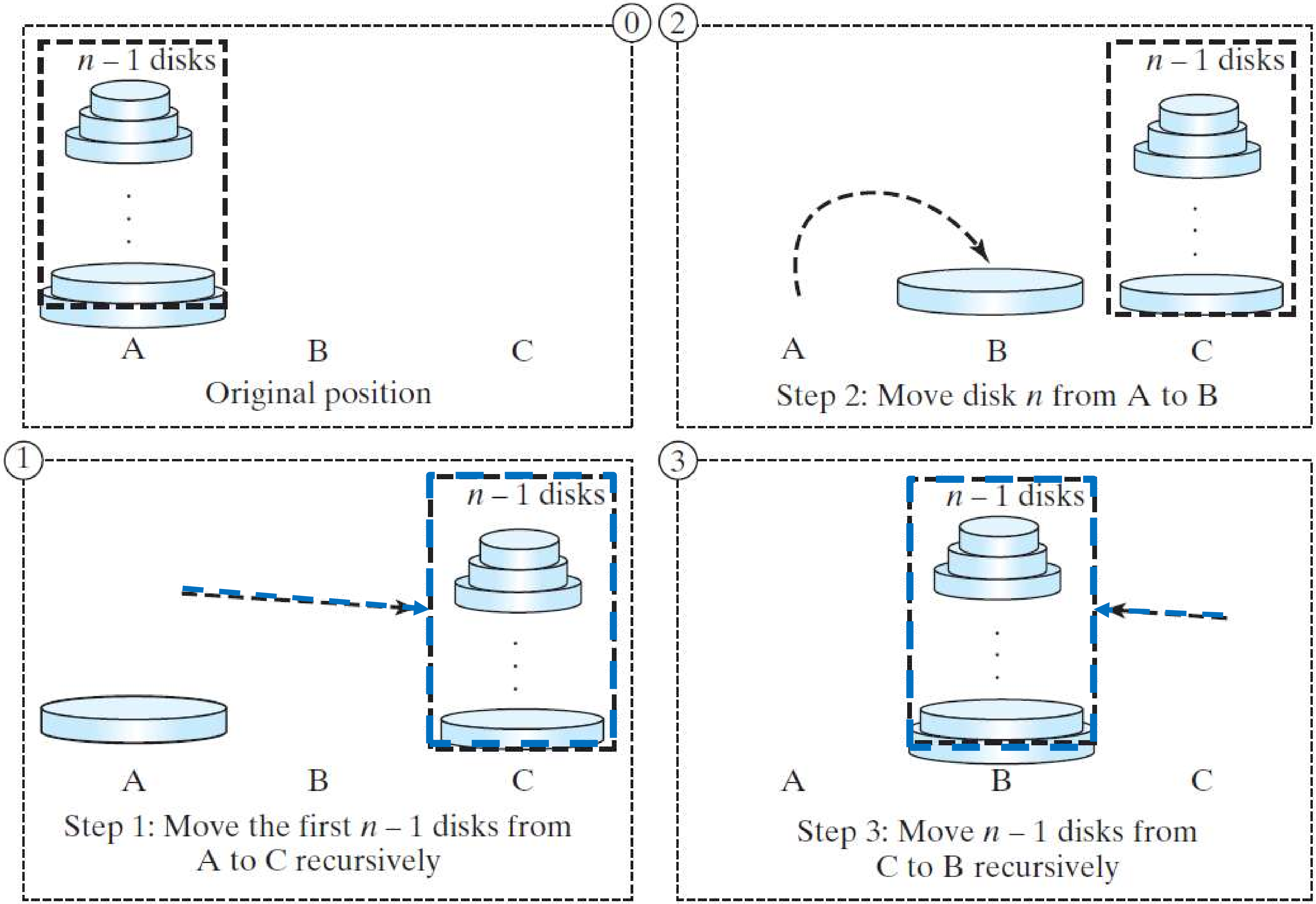
- **Goal** is to **move all** the disks from **A** to **B**

- 3 Rules
  1) No disk can be on **top** of a **smaller** disk at any time.
  2) All the disks are **initially** placed on **tower A**.
  3) **Only one** disk can be **moved at a time**, and it must be the **top** disk on the tower.

Simulation


0
Original position
1
Step 1: Move disk 1 from A to B
2
Step 2: Move disk 2 from A to C
3
Step 3: Move disk 1 from B to C
4
Step 4: Move disk 3 from A to B
5
Step 5: Move disk 1 from C to A
6
Step 6: Move disk 2 from C to B
7
Step 7: Move disk 1 from A to B

Recursive Solution to Tower of Hanoi



Solution to Tower of Hanoi

- Base case
  1) **n ==1**
     - You could simply move the disk from A to B

- Subproblems
  1) Move the **first n - 1** disks from **A** to **C** with the assistance of tower B.
  2) Move **disk n** from **A** to **B** (*Base Case*).
  3) Move **n - 1 disks** from **C** to **B** with the assistance of tower A.

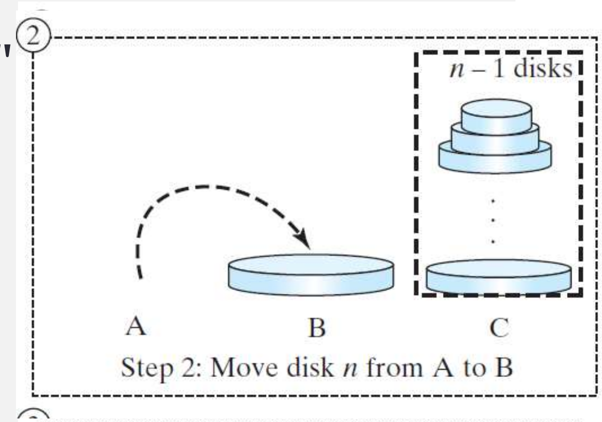
TowerOfHanoi

```java
import java.util.Scanner;

public class TowerOfHanoi {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter number of disks: "
    int n = input.nextInt();

    // Find the solution recursively
    System.out.println("The moves are:");
    moveDisks(n, 'A', 'B', 'C');
  }

  /** The method for finding the solution to move n disks
      from fromTower to toTower with auxTower */
  public static void moveDisks(int n, char fromTower,
      char toTower, char auxTower) {
    if (n == 1) // Stopping condition
      System.out.println("Move disk " + n + " from " +
        fromTower + " to " + toTower);
    else {
      moveDisks(n - 1, fromTower, auxTower, toTower);
      System.out.println("Move disk " + n + " from " +
        fromTower + " to " + toTower);
      moveDisks(n - 1, auxTower, toTower, fromTower); } } }
```



Step 2: Move disk n from A to B

Recursion VS Iteration

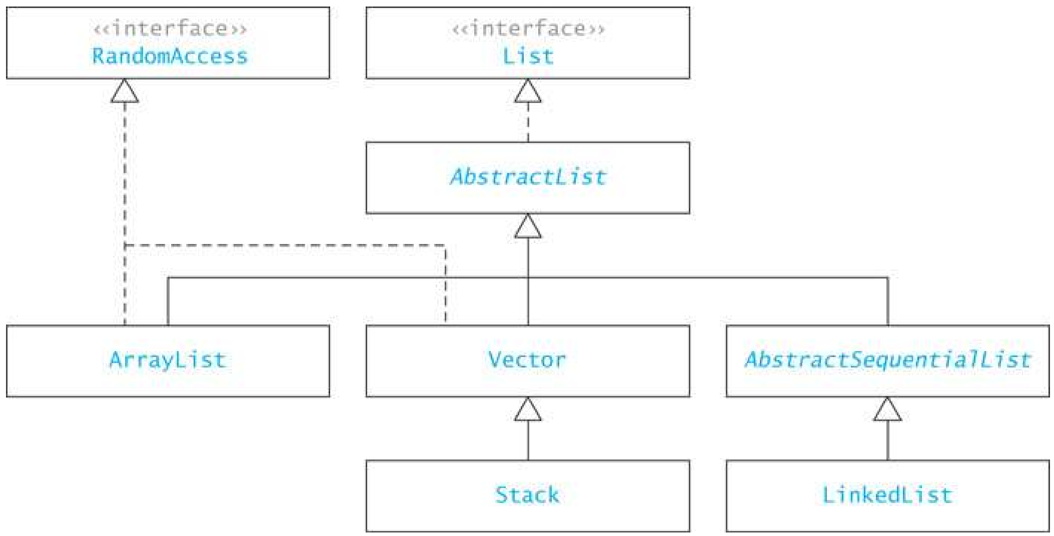
- **Recursion**
  - ✓ An alternative form of program control. It is essentially repetition **without** a loop.
  - ✓ Advantage
    - — enables you to specify a **clear** and **simple** solution for an inherently recursive problem
  - ✓ Disadvantages
    - —Uses up too **much time** and too **much memory** (***overhead***).

- If the **speed** and **resource** efficiency is **critical**, use **iteration**
- However, a problem is **inherently recursive** and then **recursion** is good for solving the problem.

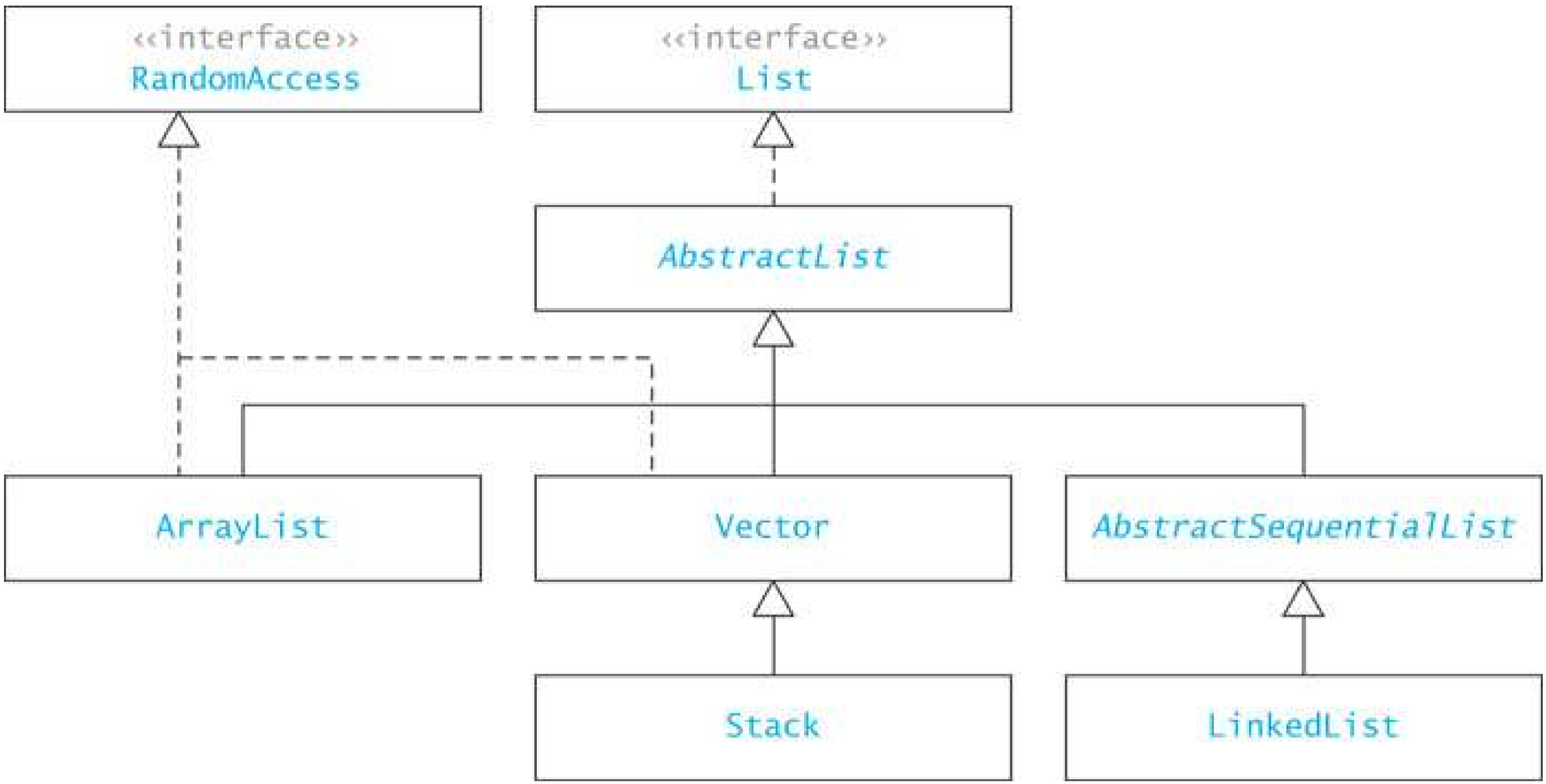
LIST

List

- A **list** is **a collection of elements**, **each** with a position or **index**
- **Iterators** facilitate sequential access to lists
- Classes **`ArrayList`**, **`Vector`**, and **`LinkedList`** are *subclasses* of abstract class **`AbstractList`** and *implement* the **`List`** interface



`java.util.List` Interface: Its Implementers



`List` Interface and `ArrayList` Class

- An **array** is an indexed structure
  - elements may be accessed in any order using subscript values
  - elements can be accessed in sequence using a loop that increments the subscript

- With a Java `Array`, you cannot
  - increase or decrease its length (length is fixed)
  - add an element at a specified position without shifting elements to make room
  - remove an element at a specified position and keep the elements contiguous without shifting elements to fill in the gap

`List` Interface and `ArrayList` Class

- Java provides a **`List` interface** as part of its API `java.util`
  - Methods in the `List` Interface - **E** is a type parameter

Method	Behavior
`E get(int index)`	**Returns the data** in the element **at position `index`**
`E set(int index, E anEntry)`	**Stores a reference to `anEntry`** in the element **at position `index`. Returns the data formerly at position `index`**
`int size()`	Gets the **current size** of the `List`
`boolean add(E anEntry)`	**Adds a reference to `anEntry` at the end** of the List. Always returns **`true`**
`void add(int index, E anEntry)`	**Adds a reference to `anEntry`**, inserting it **before the item at position `index`**
`int indexOf(E target)`	**Searches for `target`** and **returns the position of the first occurrence**, or **-1** if it is not in the `List`
`E remove (int index)`	**Removes the entry** formerly **at position `index`** and returns it
`static of(E... elements)`	**Creates an unmodifiable list** of elements. (Useful for testing)

`List` Interface and `ArrayList` Class

- Unlike the `Array` data structure, **`ArrayList`** classes that implement the `List` interface cannot store ~~primitive types~~
  - ✓ Classes must store values as **objects**

- This requires you to wrap primitive types, such as **`int`** and **`double`** in object wrappers, in this case, **`Integer`** and **`Double`**

`ArrayList` Class

- The simplest class that implements the `List` interface
- An improvement over an array object
- Use when:
  - ✓ you will be adding new elements to the end of a list
  - ✓ you need to access elements quickly in any order

`ArrayList` Class

- To declare a **`List`** object whose elements will reference **`String`** objects:

```
List<String> myList = new ArrayList<String>();
```

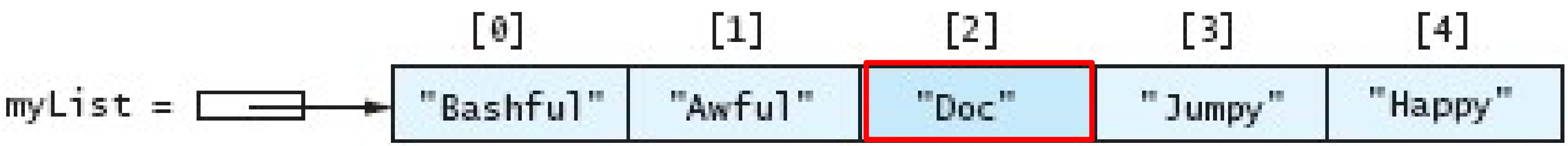
- The initial ArrayList is empty and has a default initial capacity of 10 elements
- To add strings to the list,

```
myList.add("Bashful");
myList.add("Awful");
myList.add("Jumpy");
myList.add("Happy");
```

`ArrayList` Class

- Adding an element with subscript 2:

```
myList.add(2, "Doc");
```



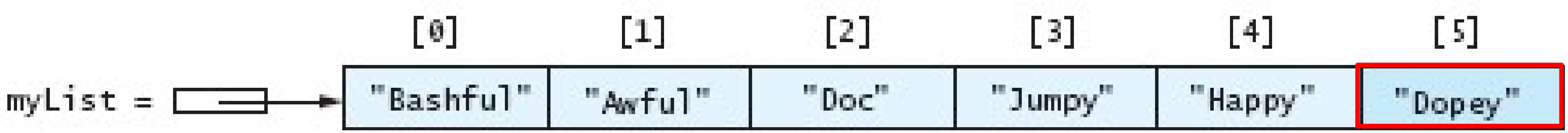
After insertion of "Doc" before the third element

- Notice that the subscripts of `"Jumpy"` and `"Happy"` have changed from [2],[3] to [3],[4]

`ArrayList` Class

- When no subscript is specified, an element is added at the end of the list:

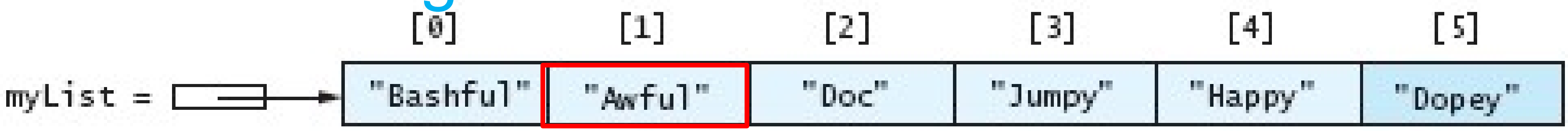
```
myList.add("Dopey");
```



After insertion of "Dopey" at the end

`ArrayList` Class

- Removing an element:




```
myList.remove(1);
```

After removal of "Awful"

- The subscripts strings referenced by [2] to [5] have changed to [1] to [4]

`ArrayList` Class

- You may also replace an element:


```
myList.set(2, "Sneezy");
```

After replacing "Jumpy" with "Sneezy"

ArrayList Class

- You cannot access an element using a bracket index as you can with arrays (~~myList[1]~~)
- Instead, you must use the **get()** method:

```
String dwarf = myList.get(2);
```

- The value of **dwarf** becomes "**Sneezy**"

`ArrayList` Class

- You can also search an `ArrayList`:

```
myList.indexOf("Sneezy");
```

- This returns `2` while

```
myList.indexOf("Jumpy");
```

- returns `-1` which indicates an unsuccessful search

Generic Collections

- The statements

```
List<String> myList = new ArrayList<String>();
List<Integer> myInts = new ArrayList<>();
var myFamily = new ArrayList<People>();
```

use a language feature called *generic collections* or *generics*

- The second statement uses the *diamond operator* **<>** to reduce redundancy
- The third statement uses the keyword **`var`** (introduced in Java 10) to simplify declarations when data type can be implied
- All 3 statements creates a `List` of objects of a specified type (`String`, `Integer`, or `People`);
  - only **references** of the specified type can be stored in the list
- The type parameter sets the data type of all objects stored in a collection

Generic Collections (cont.)

- The general **declaration** for generic collection is

```
CollectionClassName<E> variable = new CollectionClassName<>();
```

  - The **<E>** indicates a type parameter
  - Adding a noncompatible type to a generic collection will generate an error during compile time
  - However, primitive types will be autoboxed:

```
ArrayList<Integer> myList = new ArrayList<>();
myList.add(Integer.valueOf(3)); // ok
myList.add(3); // also ok! 3 is automatically wrapped
                          in an Integer object
myList.add(new String("Hello")); // generates a type
                                               incompatability error
```

Why Use Generics?

- Better type-checking: catch more errors, catch them earlier
- Documents intent
- Avoids the need to downcast from `Object`
- Can use methods of a generic class to process objects of different types by changing the type parameter for the class

LINKED LIST

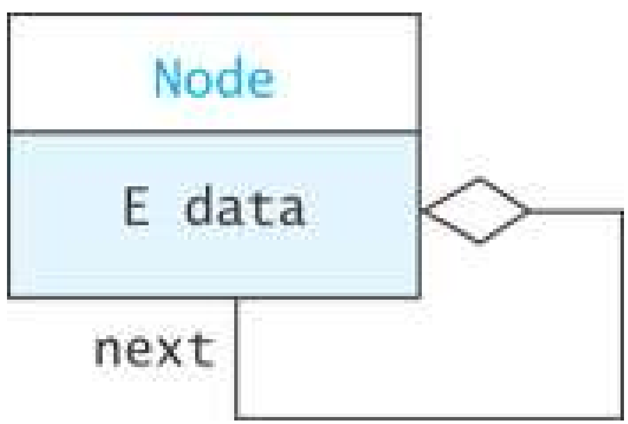
SINGLE-LINKED LISTS

Single-Linked Lists

- A **linked list** is useful for **inserting** and **removing** at **arbitrary locations**
- The `ArrayList` is limited because its `add` and `remove` methods operate in linear (`O(n)`) time—requiring a loop to shift elements
- A linked list can add and remove elements at a known location in **O(1)** time
- In a linked list, instead of an index, each element is linked to the following element

List Node

- A **node** can contain:
  - a **data** item
    - the node contains a data field named **`data`** of type **`E`**
  - one or more **links**
    - A link is a reference to a list node, the next node, named **`next`**



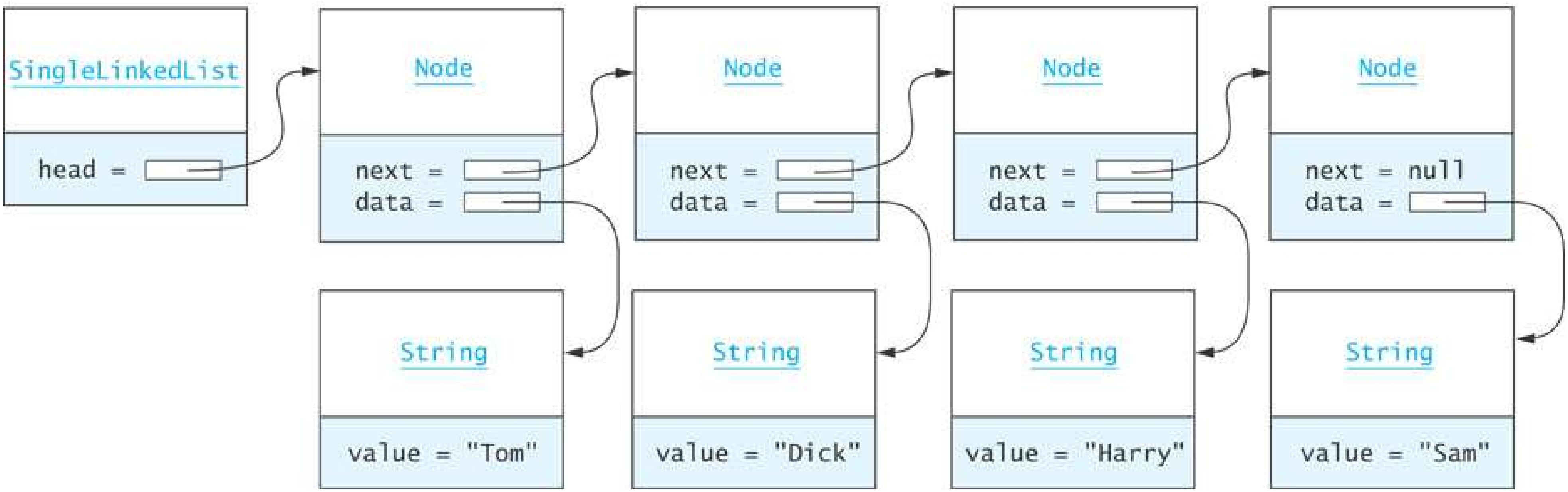
```
private static class Node<E> {
  private E data;
  private Node<E> next;

  /** Creates a new node with a null next field
      @param dataItem  The data stored
  */
  private Node(E data) {
    data = dataItem;
    next = null;
  }

 /** Creates a new node that references another node
      @param dataItem  The data stored
      @param nodeRef  The node referenced by new node
  */
  private Node(E dataItem, Node<E> nodeRef) {
    data = dataItem;
    next = nodeRef;
  }
}
```

Generally, all details of the `Node` class should be private. This applies also to the data fields and constructors.

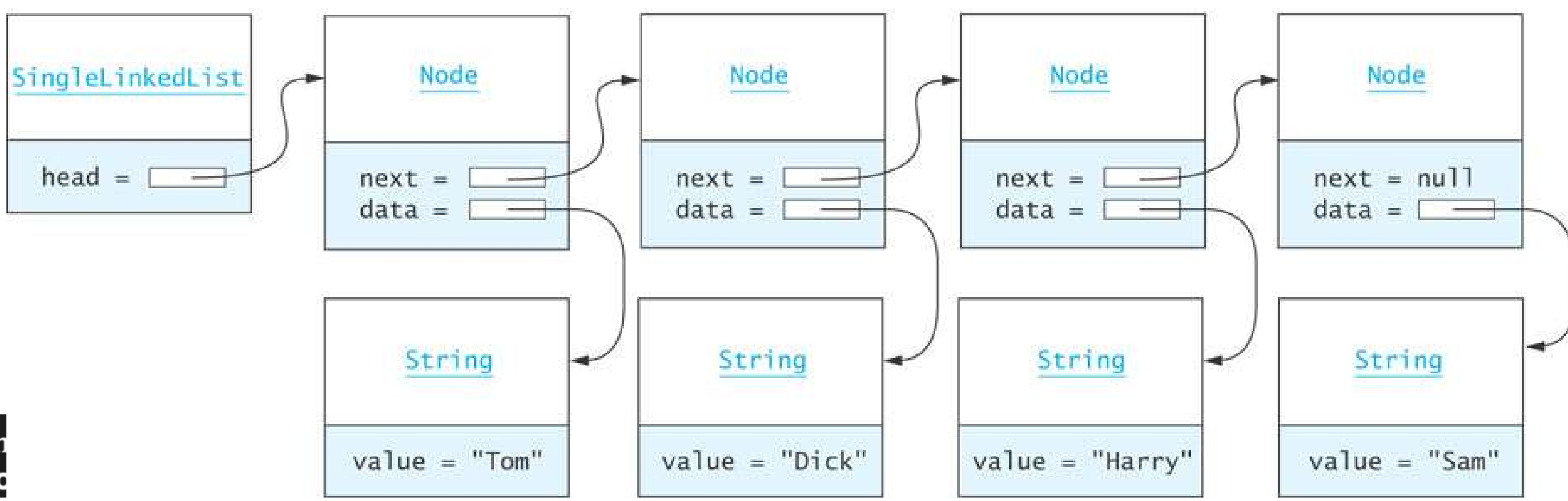
Connecting Nodes in a Single Linked List



Connecting Nodes (cont.)

```
var sLL = new KWSingleLinkedList<String>();
var tom = new Node<String>("Tom");
var dick = new Node<String>("Dick");
var harry = new Node<String>("Harry");
var sam = new Node<String>("Sam");

sLL.head = tom;
tom.next = dick;
dick.next = harry;
harry.next = sam;
```

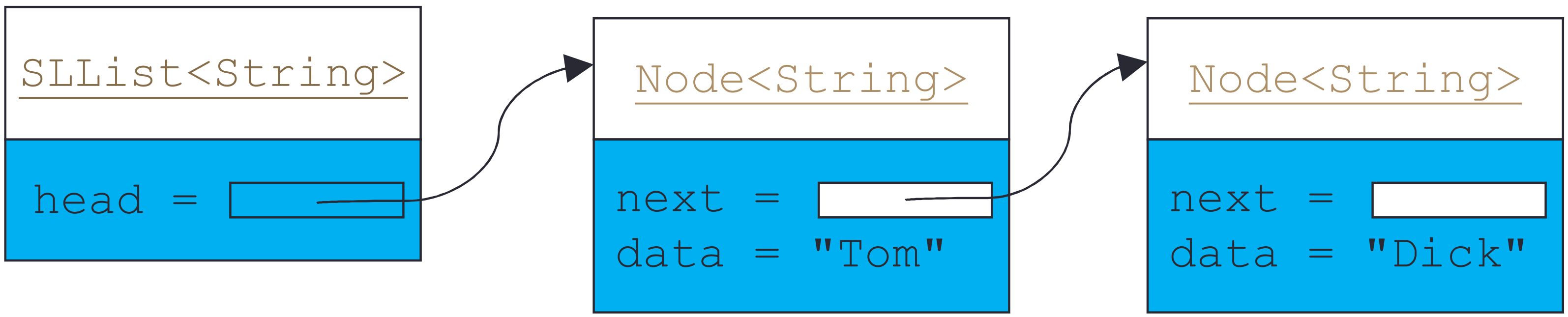


A Single-Linked List Class

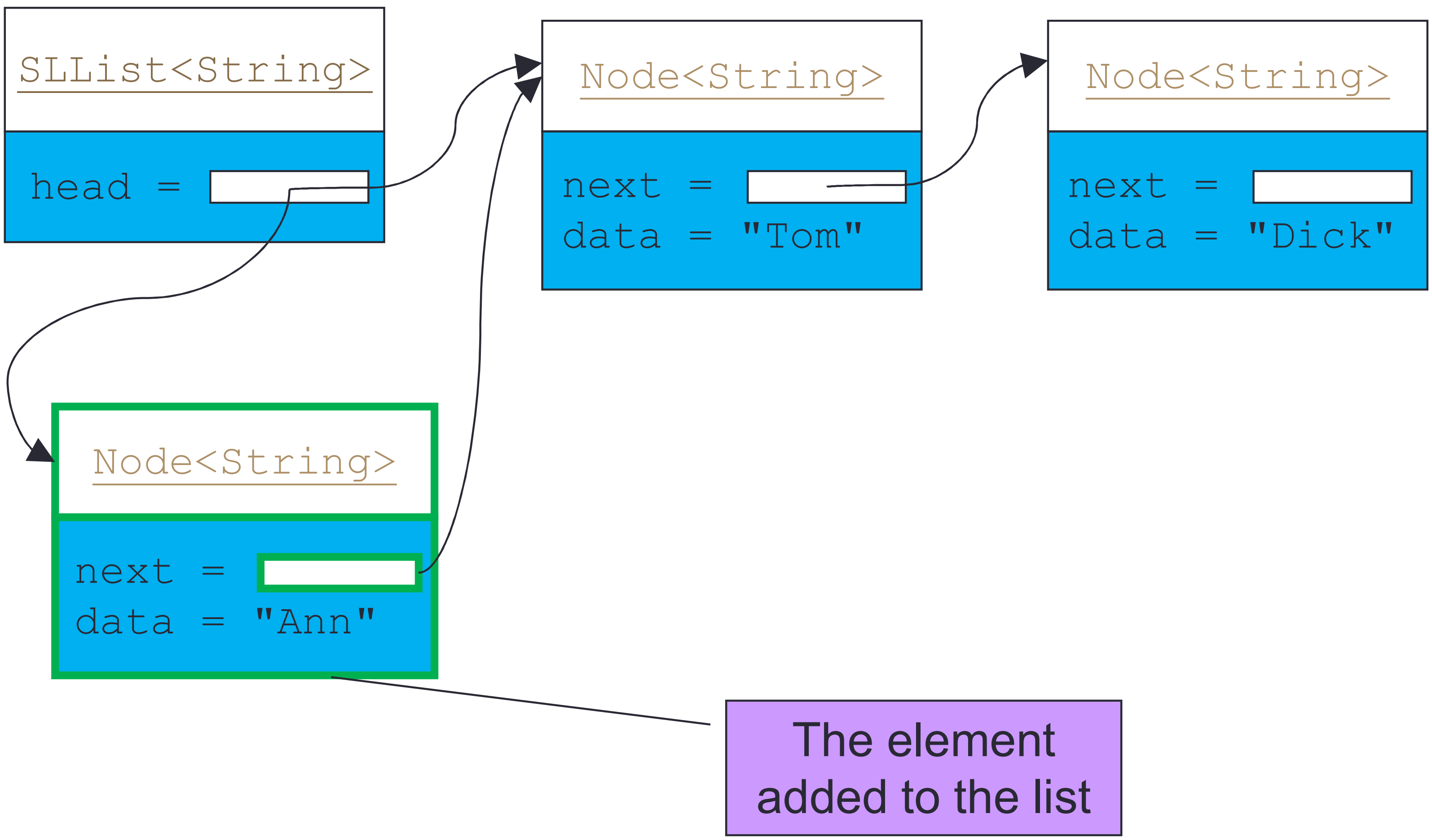
- Generally, we do not have individual references to each node.
- A `KWSingleLinkedList` object has a data field **`head`**, the *list head*, which references the first list node

```
public class KWSingleLinkedList<E> {
  private Node<E> head = null;
  private int size = 0;
  ...
}
```

KWSingleLinkedList (= SLList): Example



Implementing `addFirst(E)`



Implementing `addFirst(E)` (cont.)

```
private void addFirst (E item) {
  Node<E> temp = new Node<E>(item, head);
  head = temp;
  size++;
}
```

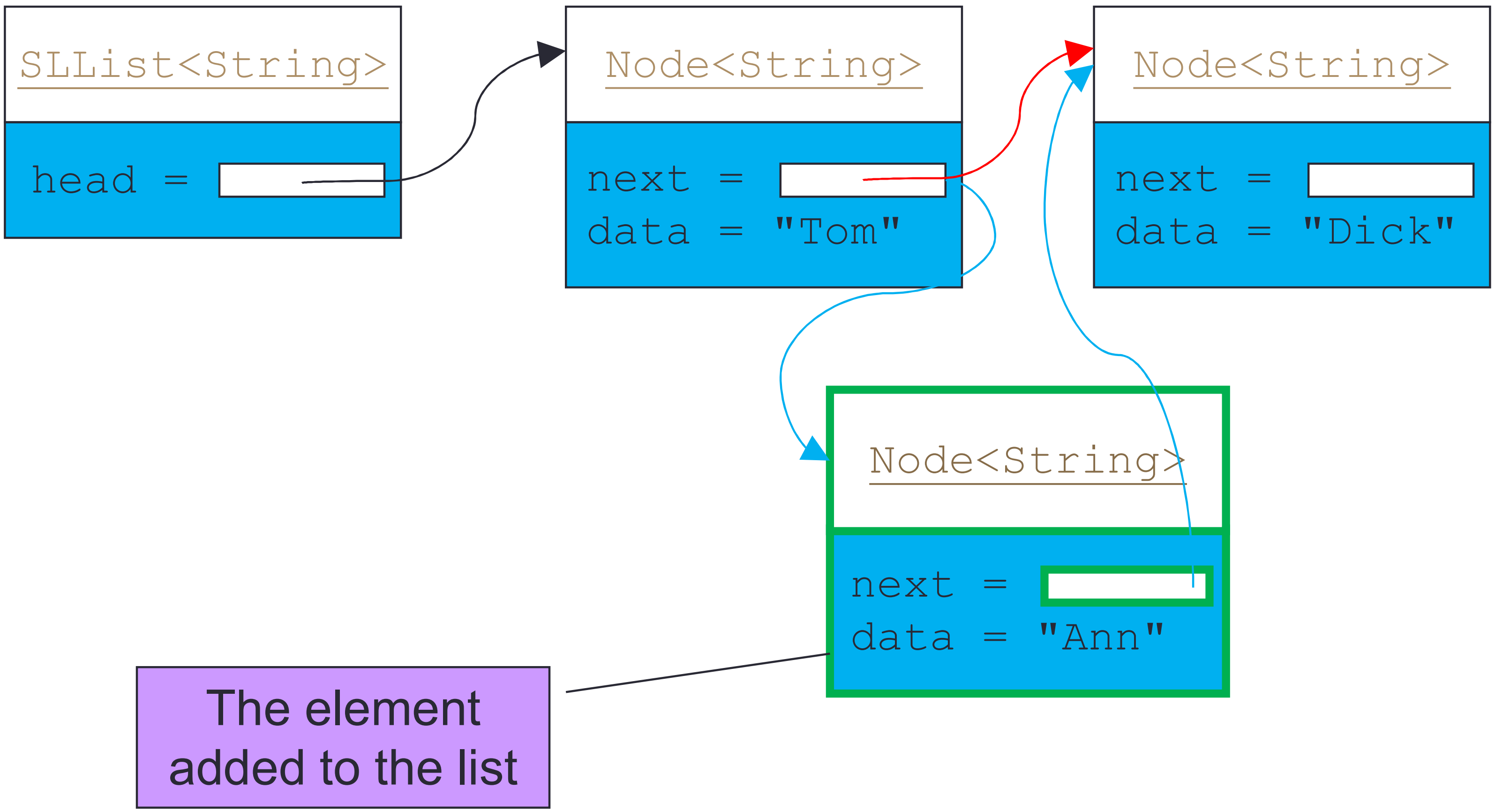
or, more simply ...

```
private void addFirst (E item) {
  head = new Node<E>(item, head);
  size++;
}
```

O(1)

This works even if head is null.

addAfter(Node<E>, E)



addAfter(Node<E>,E) (cont.)

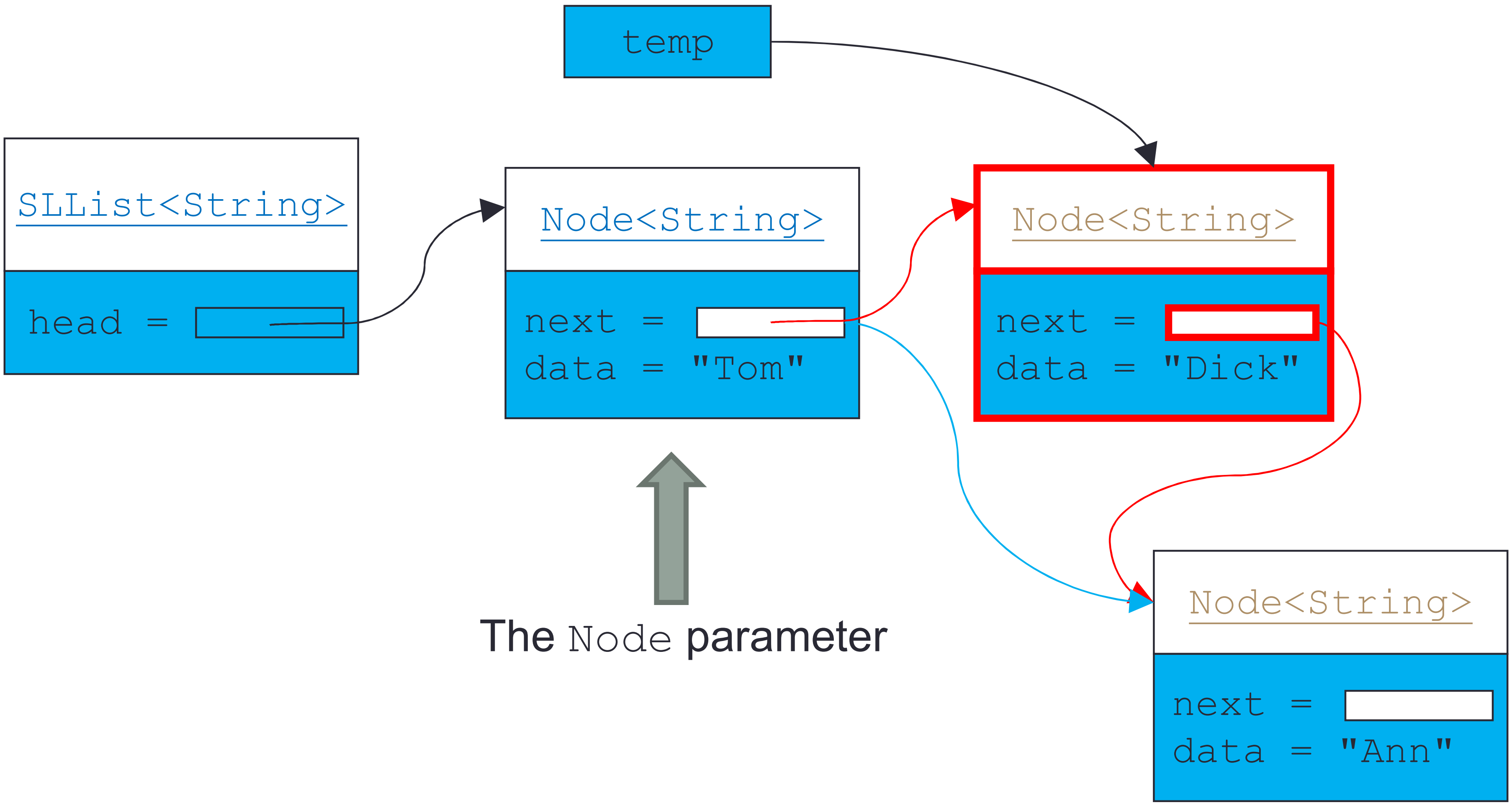
```
private void addAfter (Node<E> node, E item) {
  Node<E> temp = new Node<E>(item, node.next);
  node.next = temp;
  size++;
}
```

or, more simply ...

```
private void addAfter (Node<E> node, E item) {
  node.next = new Node<E>(item, node.next);
  size++;
}
```

We declare this method `private` since it should not be called from outside the class. Later we will see how this method is used to implement the public add methods

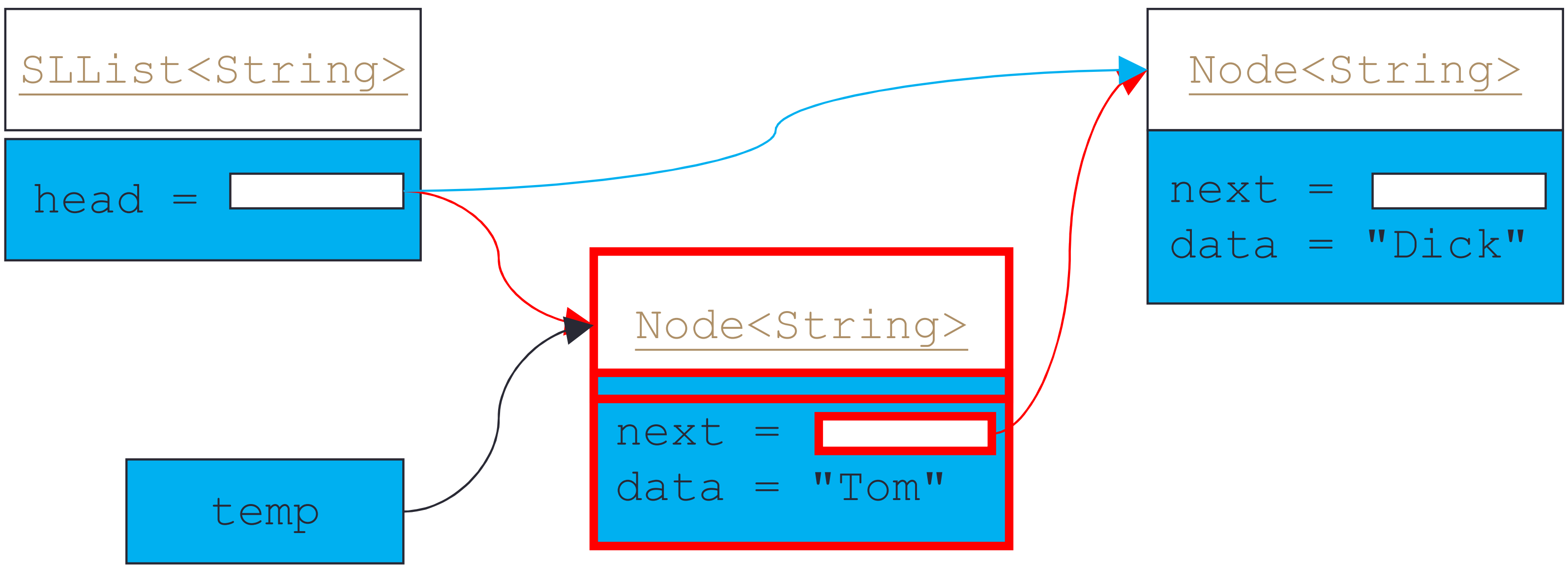
removeAfter(Node<E>)



removeAfter(Node<E>) (cont.)

```
private E removeAfter(Node<E> node) {
  Node<E> temp = node.next;
  if (temp != null) {
    node.next = temp.next;
    size--;
    return temp.data;
  } else {
    return null;
  }
}
```

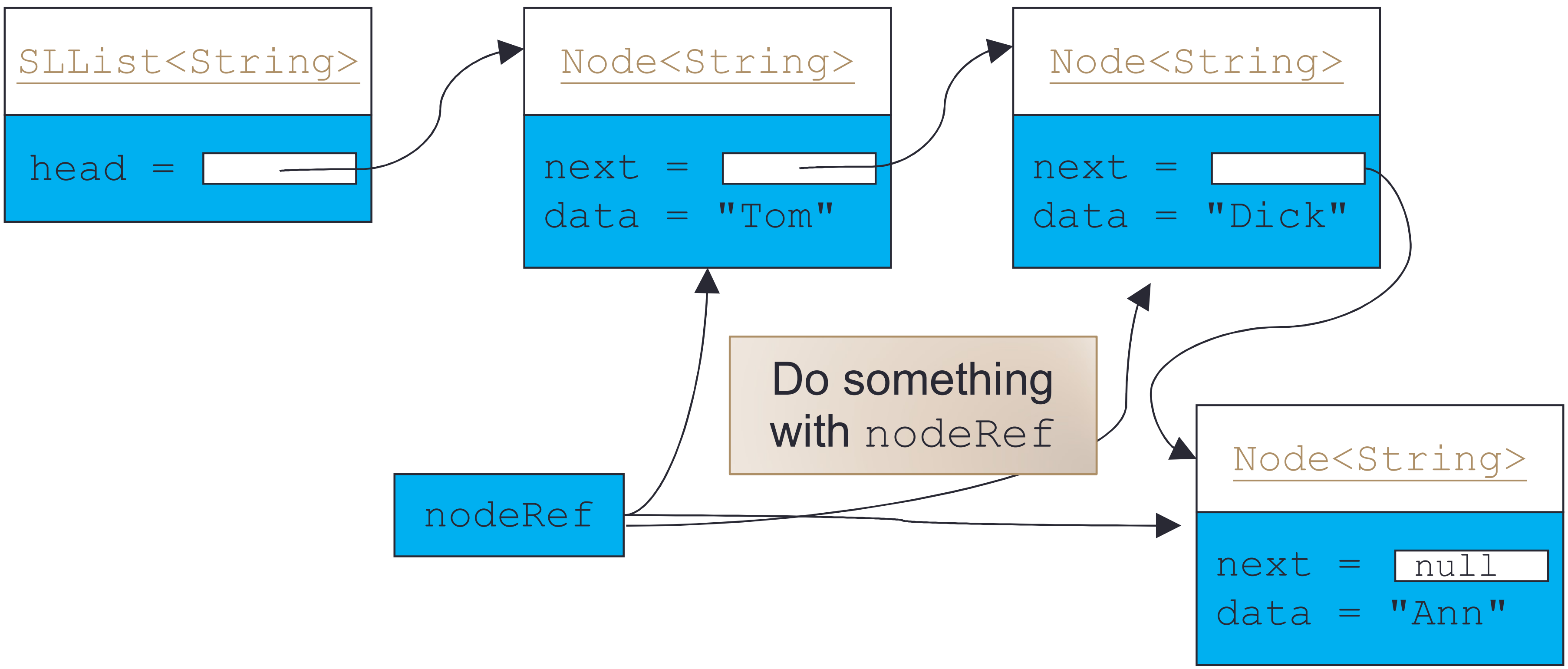
removeFirst()



removeFirst() (cont.)

```
private E removeFirst() {
  Node<E> temp = head;
  if (head != null) {
    head = head.next;
  }
  if (temp != null) {
    size--;
    return temp.data
  } else {
    return temp;
  }
}
```

Traversing a Single-Linked List



Traversing a Single-Linked List (cont.)

- `toString()` can be implemented with a traversal:

```
public String toString() {
  Node<String> nodeRef = head;
  StringBuilder result = new StringBuilder();
  while (nodeRef != null) {
    result.append(nodeRef.data);
    if (nodeRef.next != null) {
      result.append(" ==> ");
    }
    nodeRef = nodeRef.next;
  }
  return result.toString();
}
```

SLList.getNode(int)

- In order to implement methods required by the List interface, we need an additional **helper** method:

```
private Node<E> getNode(int index) {
  Node<E> node = head;
  for (int i=0; i<index && node != null; i++) {
    node = node.next;
  }
  return node;
}
```

O(N)

Table 2.5 More Methods of `List<E>` Interface in `KWSingleLinkedList<E>`

Method	Behavior
`public E get(int index)`	Returns a reference to the element at position `index`.
`public E set(int index, E anEntry)`	Sets the element at position `index` to reference `anEntry`. Returns the previous value.
`public int size()`	Gets the current size of the `List`.
`public boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the `List`. Always returns `true`.
`public void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`.
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or –1 if it is not in the `List`.

public E get(int index) O(N)

```
public E get (int index) {
 if (index < 0 || index >= size) {
     throw new IndexOutOfBoundsException(Integer.toString(index));
   }
   Node<E> node = getNode(index);
   return node.data;
}
```

public E set(int index, E newValue)

```
public E set (int index, E newValue) {
   if (index < 0 || index >= size) {
    throw new IndexOutOfBoundsException(Integer.toString(index));
   }
   Node<E> node = getNode(index);
   E result = node.data;
   node.data = newValue;
   return result;
}
```

O(N)

public void add(int index, E item)

```
public void add (int index, E item) {
  if (index < 0 || index > size) {
    throw new
      IndexOutOfBoundsException(Integer.toString(index));
  }
  if (index == 0) {
    addFirst(item);
  } else {
    Node<E> node = getNode(index-1);
    addAfter(node, item);
  }
}
```

O(N)

public boolean add(E item)

- To add an item to the end of the list

```
public boolean add(E item) {
  add(size, item);
  return true;
}
```

O(N)

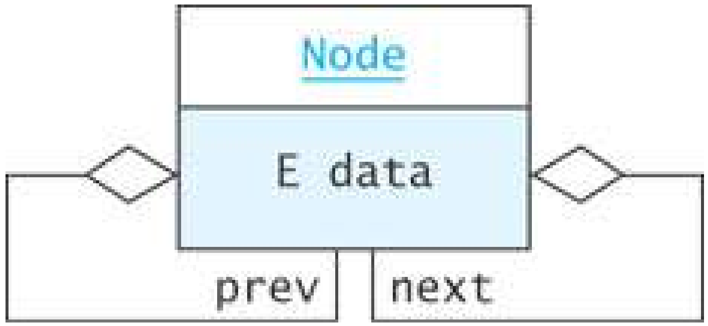
DOUBLE-LINKED LISTS AND **CIRCULAR LISTS**

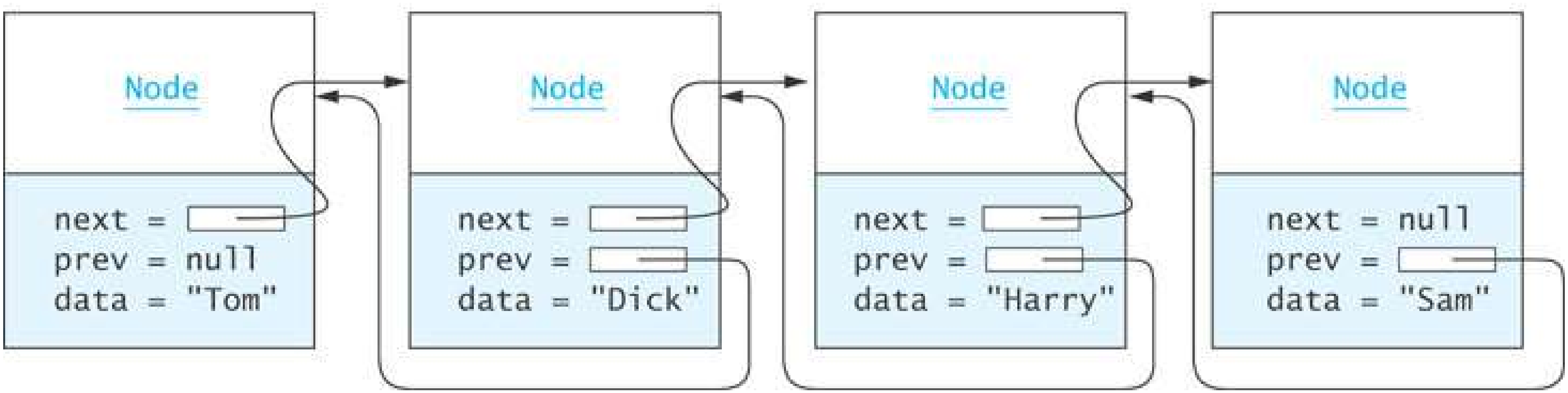
Section 2.6

Double-Linked Lists

- Limitations of a **singly-linked list** include:
  - Insertion at the front is O(1); insertion at other positions is **O(*n*)**
  - Insertion is convenient **only after** a referenced node
  - Removing a node requires a reference to previous node
  - We can traverse list only in the forward direction
- We can overcome these limitations:
  - Add a reference in each node to the previous node, creating a ***double-linked list***

Double-Linked Lists (cont.)

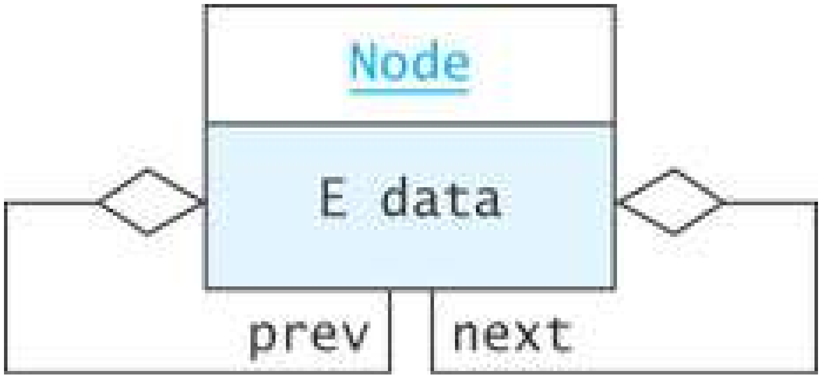




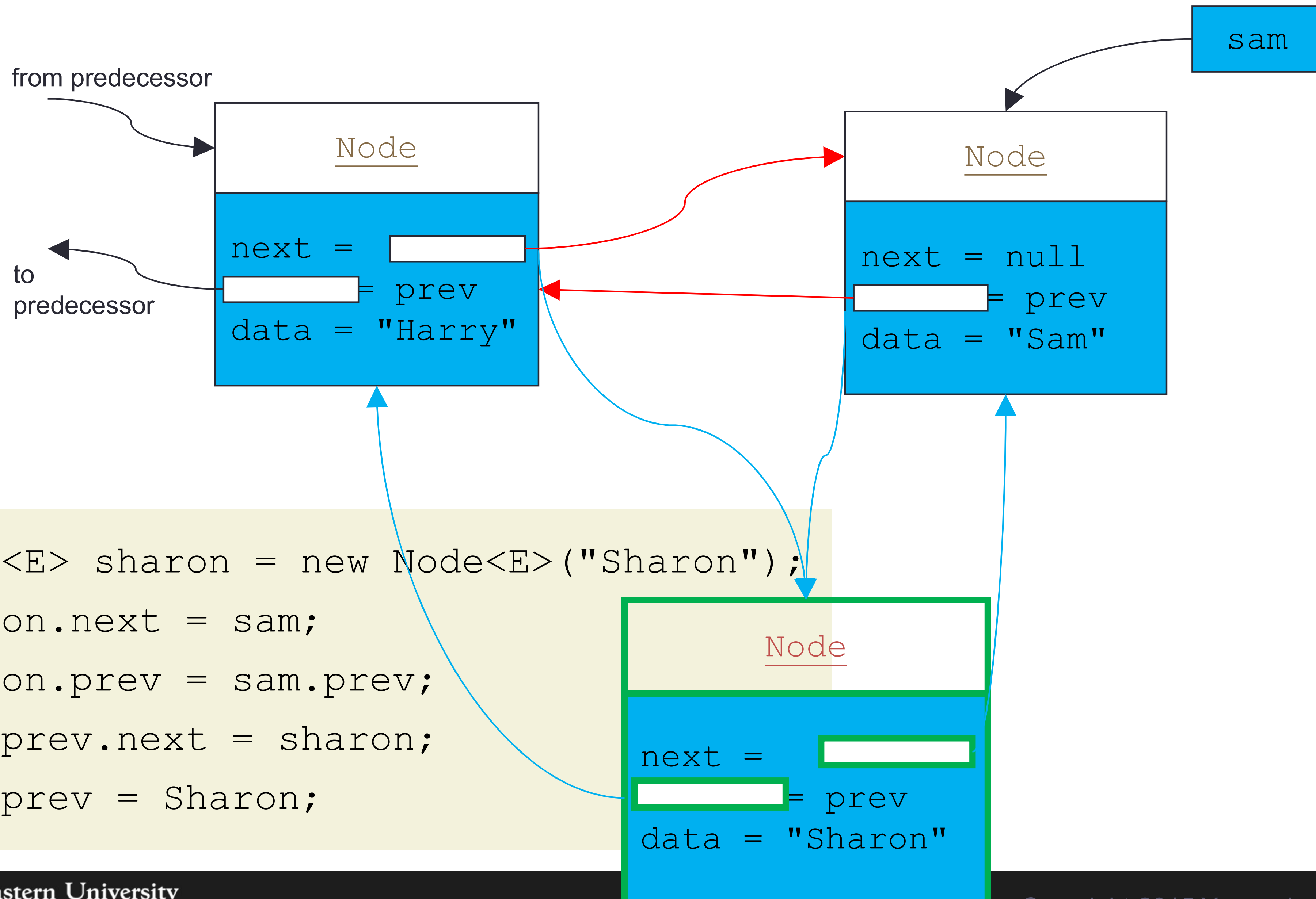
Node<E> Class

```
private static class Node<E> {
  private E data;
  private Node<E> next = null;
  private Node<E> prev = null;

  private Node(E dataItem) {
    data = dataItem;
  }
}
```

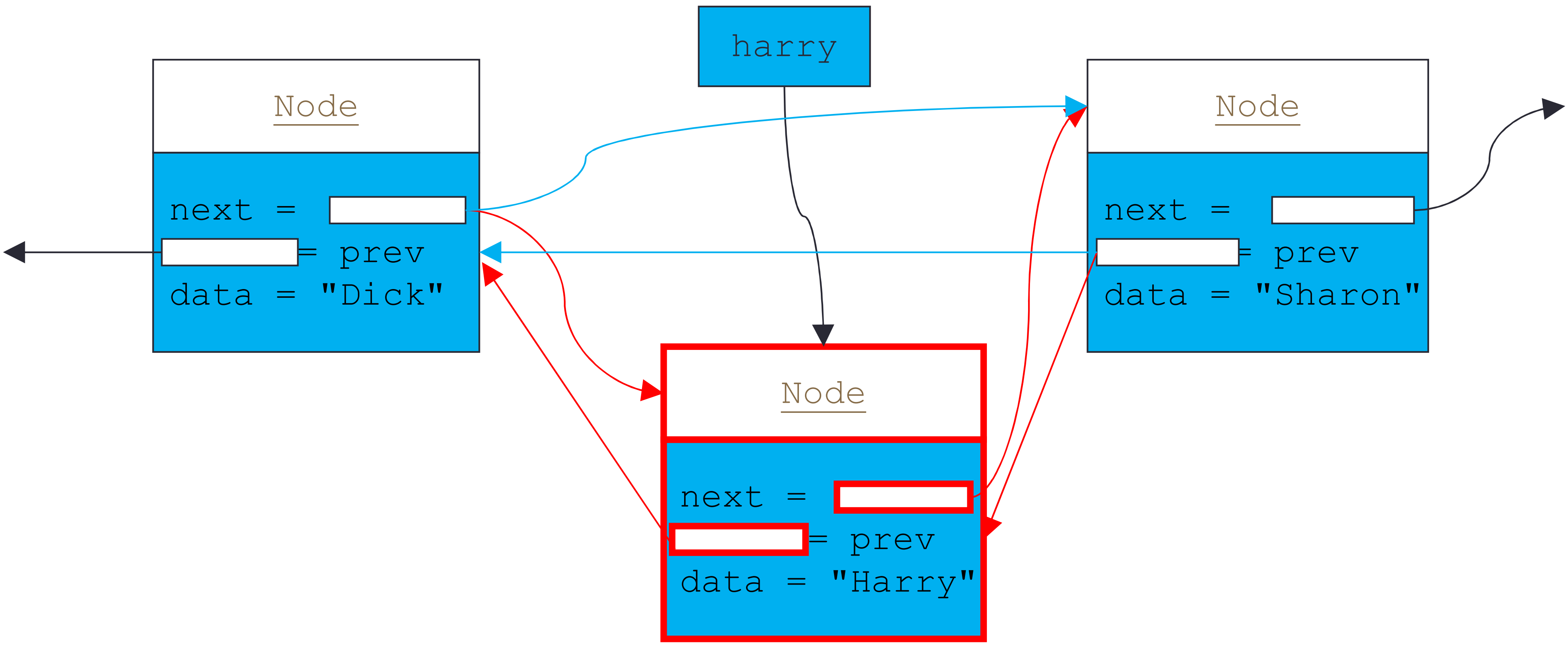


Inserting into a Double-Linked List




```
Node<E> sharon = new Node<E>("Sharon");
sharon.next = sam;
sharon.prev = sam.prev;
sam.prev.next = sharon;
sam.prev = Sharon;
```

Removing from a Double-Linked List




```
harry.prev.next = harry.next
harry.next.prev = harry.prev
```

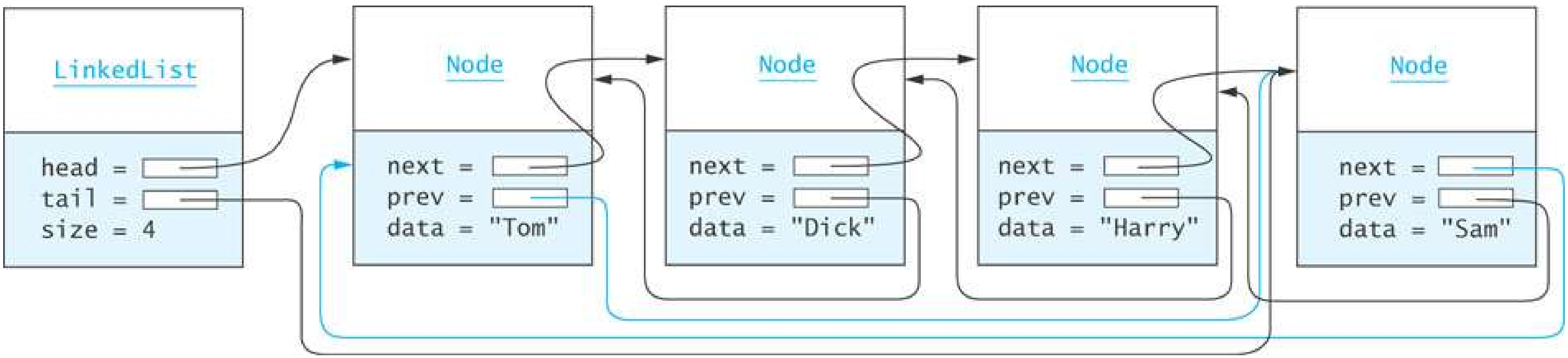
A Double-Linked Class

- A double-linked list object has data fields:
  - **`head`** (a reference to the first list `Node`)
  - **`tail`** (a reference to the last list `Node`)
  - **`size`**
- Insertion at either end is **O(1)**; insertion elsewhere is still **O(n)**

Circular Lists

- **Circular double-linked** list:
  - Link **last** node to the **first** node, and
  - Link **first** node to the **last** node
- We can also build **singly-linked circular lists**:
  - Traverse in forward direction only
- **Advantages**:
  - **Continue** to **traverse** even after passing the first or last node
  - **Visit all** elements from **any starting point**
  - Never fall off the end of a list
- **Disadvantage**: Code must avoid an **infinite loop**!

Circular Lists (cont.)



Methods of Class `LinkedList<E>`

Method	Behavior
`public void add(int index, E obj)`	Inserts object `obj` into the list at position `index`.
`public void addFirst(E obj)`	Inserts object `obj` as the first element of the list.
`public void addLast(E obj)`	Adds object `obj` to the end of the list.
`public E get(int index)`	Returns the item at position `index`.
`public E getFirst()`	Gets the first element in the list. Throws `NoSuchElementException` if the list is empty.
`public E getLast()`	Gets the last element in the list. Throws `NoSuchElementException` if the list is empty.
`public boolean remove(E obj)`	Removes the first occurrence of object `obj` from the list. Returns `true` if the list contained object `obj`; otherwise, returns `false`.
`public int size()`	Returns the number of objects contained in the list.