CS 304 Lecture 2

Abstract data types

Xiwei Wang, Ph.D.

Assistant Professor

Department of Computer Science

Northeastern Illinois University

Chicago, Illinois, 60625

30 August 2016

Java interfaces

- Abstract data type (ADT) - A data type whose properties (domain and operations) are specified independently of any particular implementation.
- Java abstract method
  - Only includes a description of its parameters.
  - No method bodies or implementations are allowed.
  - In other words, only the interface of the method is included.
- Java interface
  - Similar to a Java class: it can include variable declarations and methods.
  - However, variables must be constants; methods must be abstract.
  - A Java interface cannot be instantiated.
  - Java interfaces can be used to represent ADTs.

Data levels

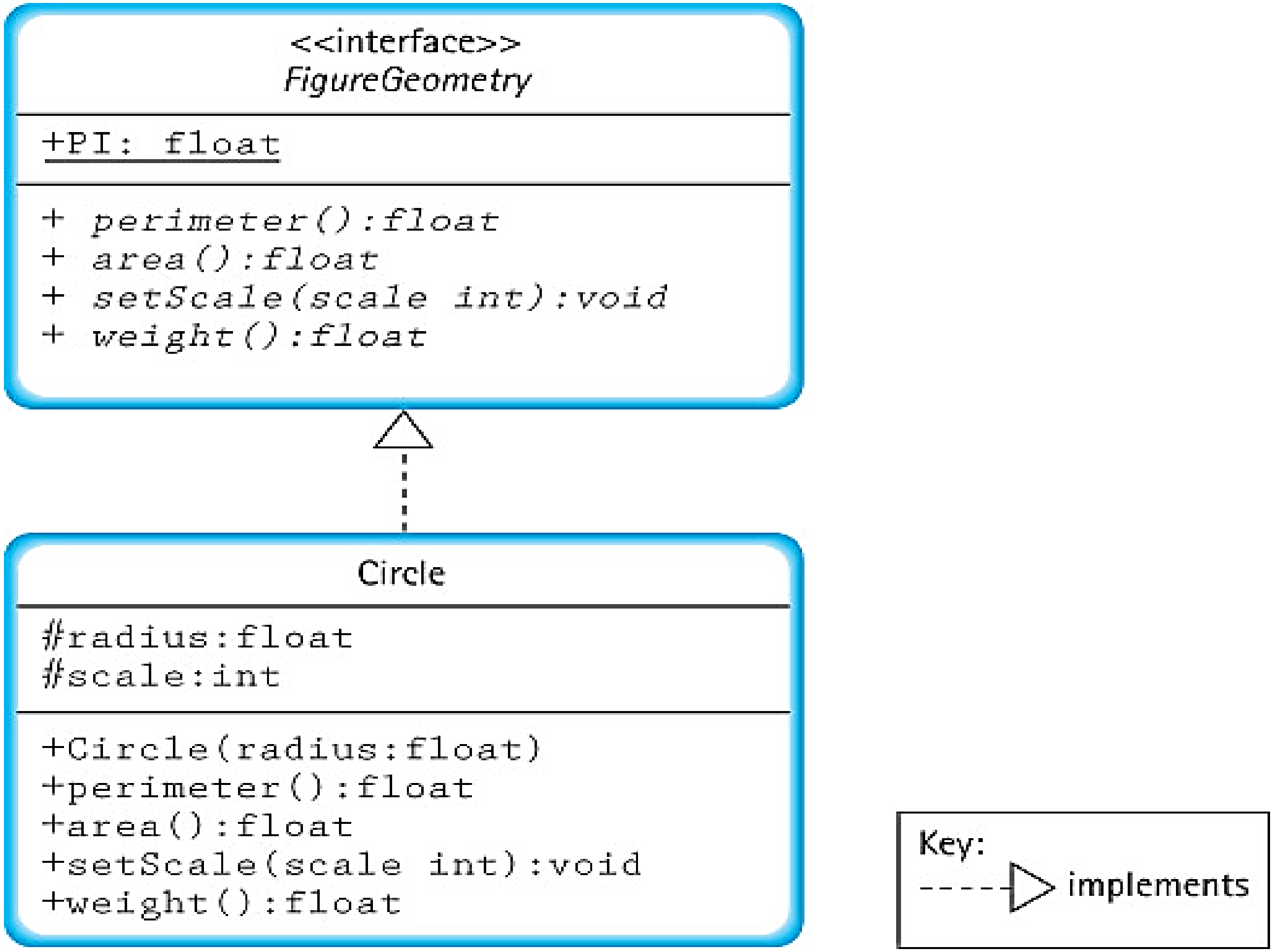
- We deal with ADTs from three perspectives:
  - Logical (or abstract) level: what is this ADT? What does it model? What are its responsibilities? What is its interface?
  - Implementation (or concrete) level: How do we represent and manipulate the data in memory? How do we fulfill the responsibilities of the ADT?
  - Application (or user, client) level: what program statements to use to create instances of the ADT and invoke its operations.

Preconditions and postconditions

- Access to the ADT is provided through its exported methods.
- To be able to invoke a method, an application programmer must know its exact interface: its name, the types of its expected parameters, and its return type.
- In addition, the programmer needs to know any assumptions that must be true for the method to work correctly (*preconditions*) and the effects (results) of invoking the method (*postconditions*).

Java interfaces

- The **`FigureGeometry`** interface and the **`Circle`** class that implements it:



<<interface>>
FigureGeometry

+PI: float

+ *perimeter():float*
+ *area():float*
+ *setScale(scale int):void*
+ *weight():float*

Circle

#radius:float
#scale:int

+Circle(radius:float)
+perimeter():float
+area():float
+setScale(scale int):void
+weight():float

Key:
implements

Java interfaces

```java
public interface FigureGeometry
{
  final float PI = 3.14f;
  float perimeter();  // Returns perimeter of this figure.
  float area();       // Returns area of this figure.
  void setScale(int scale);  // Scale of this figure is set
                             // to "scale".
  float weight(); // Returns weight of this figure.
                  // Weight = area X scale.
}
```

Java interfaces

```
public class Circle implements FigureGeometry
{
  protected float radius;
  protected int scale;

  public Circle(float radius)
  {
    this.radius = radius;
  }

  // Returns perimeter of this figure.
  public float perimeter()
  {
    return(2 * PI * radius);
  }
  ......
}
```

it is better to use a different name, e.g., `theRadius`. Why?

Benefits of using interfaces

- Interface provides a contract for all the implementation classes.
- We can formally verify that the interface "contract" is met by the implementation.
- We can provide a consistent interface to applications from among alternate implementations of the ADT.

The **`StringLog`** ADT specification

- The primary responsibility of **`StringLog`** ADT:
  - Remember all the strings that have been inserted into it;
  - Identify whether a given string has been inserted.
- A **`StringLog`** client uses a **`StringLog`** to record strings and later check to see if a particular string has been recorded. Every **`StringLog`** must have a "name".
- **`StringLog`** methods
  - constructors
  - transformers (mutators): **`insert(String element)`**, **`clear`**
  - observers (accessors): **`contains(String element)`**, **`size`**, **`isFull`**, **`getName`**, **`toString`**

An application example **StringLog**

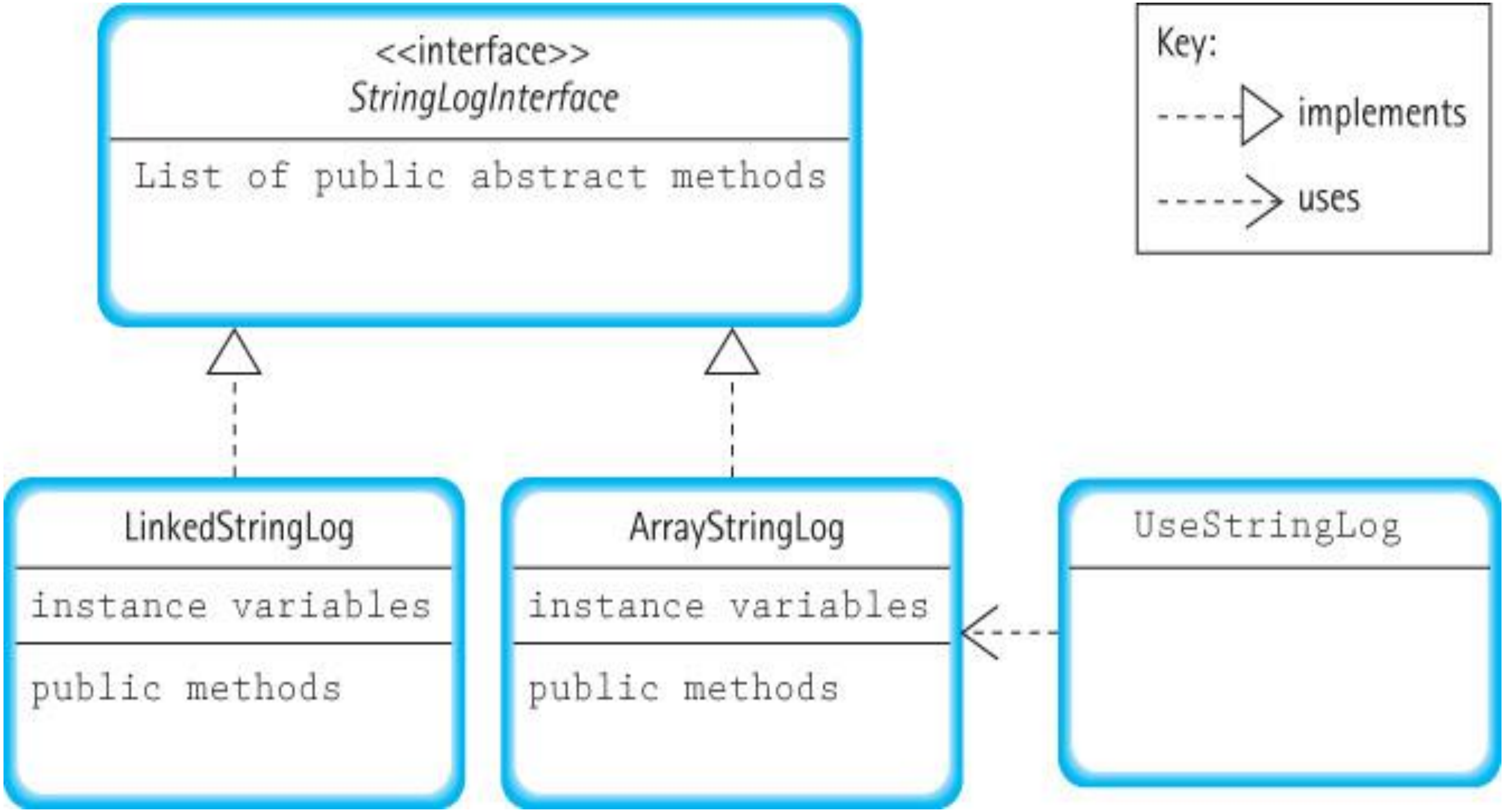
```java
import ch02.stringLogs.*;
public class UseStringLog
{
  public static void main(String[] args)
  {
    StringLogInterface log;
    log = new ArrayStringLog("Example Use");
    log.insert("Elvis");
    log.insert("King Louis XII");
    log.insert("Captain Kirk");
    System.out.println(log);
    System.out.println("The size of the log is " +
                        log.size());
    System.out.println("Elvis is in the log: " +
                        log.contains("Elvis"));
    System.out.println("Santa is in the log: " +
                        log.contains("Santa"));
  }
}
```

```
Log: Example Use
1. Elvis
2. King Louis XII
3. Captain Kirk
The size of the log is 3
Elvis is in the log: true
Santa is in the log: false
```

The three levels

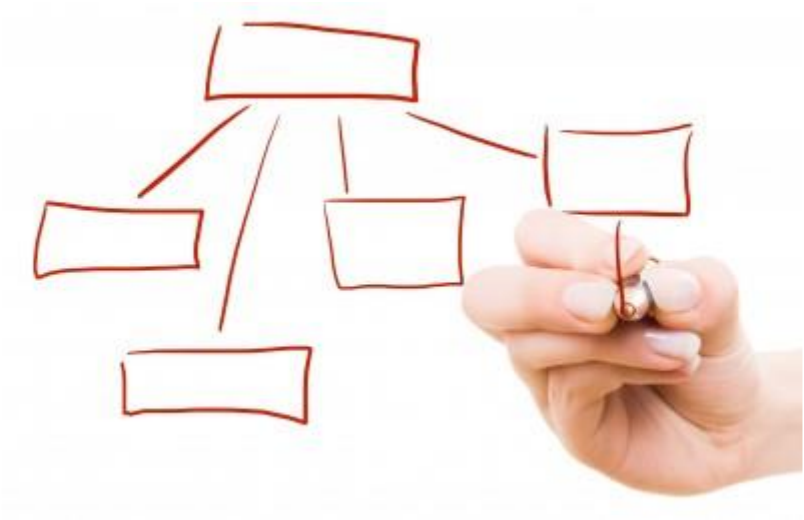
- Logical (or abstract) level
  - `StringLogInterface` provides an *abstract view* of the `StringLog` ADT.
  - It is used by the `UseStringLog` application and implemented by the `ArrayStringLog` class.
- Implementation (or concrete) level
  - The `ArrayStringLog` class provides a specific implementation of the `StringLog` ADT, fulfilling the contract presented by the `StringLogInterface`.
- Application (or user or client) level
  - The `UseStringLog` program is the application.
  - It declares a variable `log` of type `StringLogInterface`.
  - It uses the `ArrayStringLog` implementation of the `StringLogInterface` to perform some simple tasks.

The `StringLog` classes



Array-based `StringLog` ADT implementation

- Recall the following **`StringLog`** methods
  - constructors
  - transformers (mutators): **`insert(String element)`**, **`clear`**
  - observers (accessors): **`contains(String element)`**, **`size`**, **`isFull`**, **`getName`**, **`toString`**



Array-based **StringLog** ADT implementation

```
package ch02.stringLogs;

public class ArrayStringLog implements StringLogInterface
{
  protected String name;     // name of this log
  protected String[] log;    // array that holds log strings
  protected int lastIndex = -1; // index of last string in array

  // constructors
  public ArrayStringLog(String name, int maxSize)
  {
    log = new String[maxSize];
    this.name = name;
  }

  public ArrayStringLog(String name)
  {
    log = new String[100];
    this.name = name;
  }
```

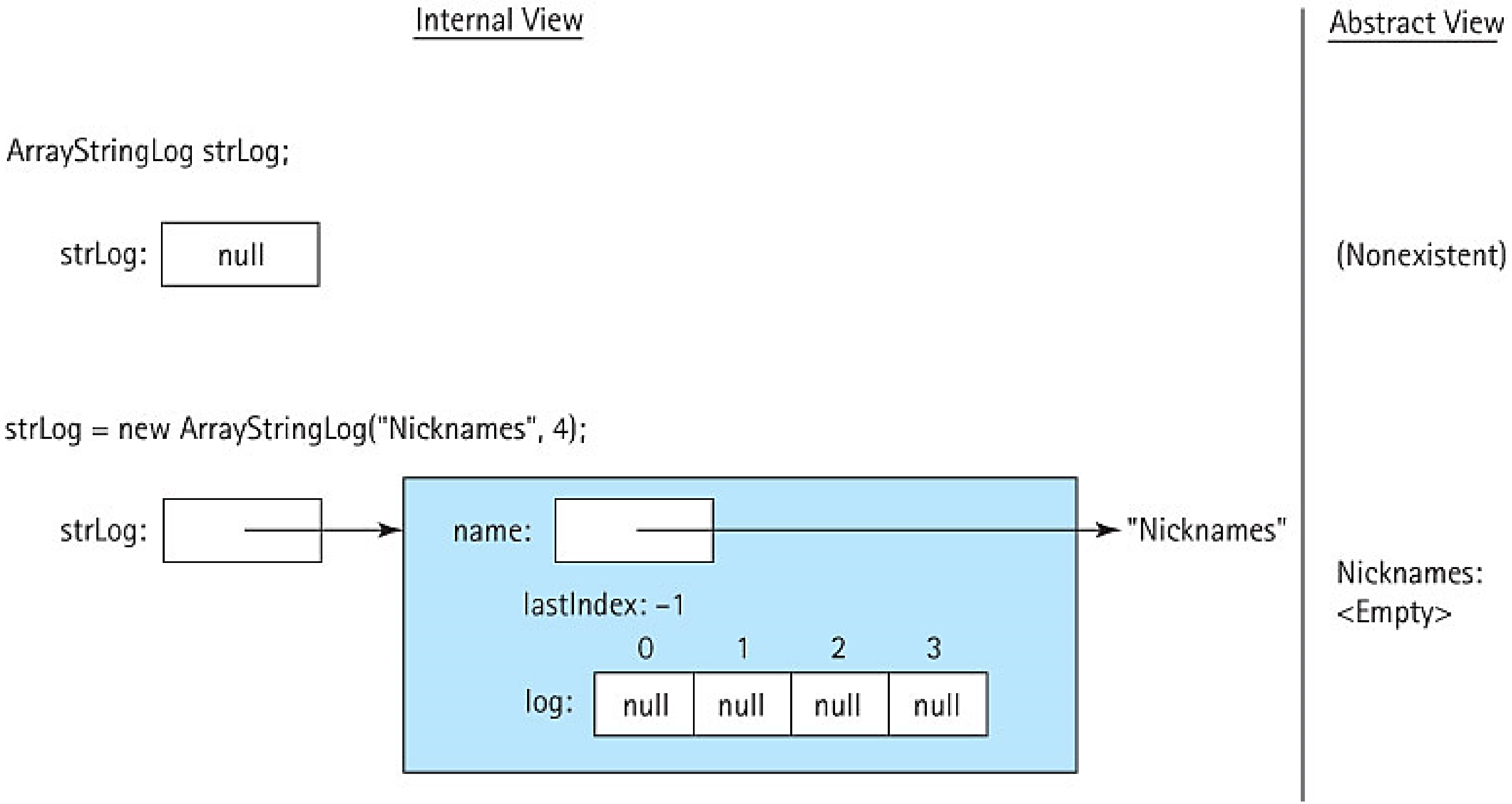
The insert operation

```
public void insert(String element)
// Places element into this StringLog.
{
  lastIndex++;
  log[lastIndex] = element;
}
```

An example use:

```
ArrayStringLog strLog;
strLog = new ArrayStringLog("aliases", 4);
strLog.insert("Babyface");
String s1 = new String("Slim");
strLog.insert(s1);
```

The **`insert`** operation



Internal View

Abstract View

ArrayStringLog strLog;

strLog: null

(Nonexistent)

strLog = new ArrayStringLog("Nicknames", 4);

strLog:

name:

"Nicknames"

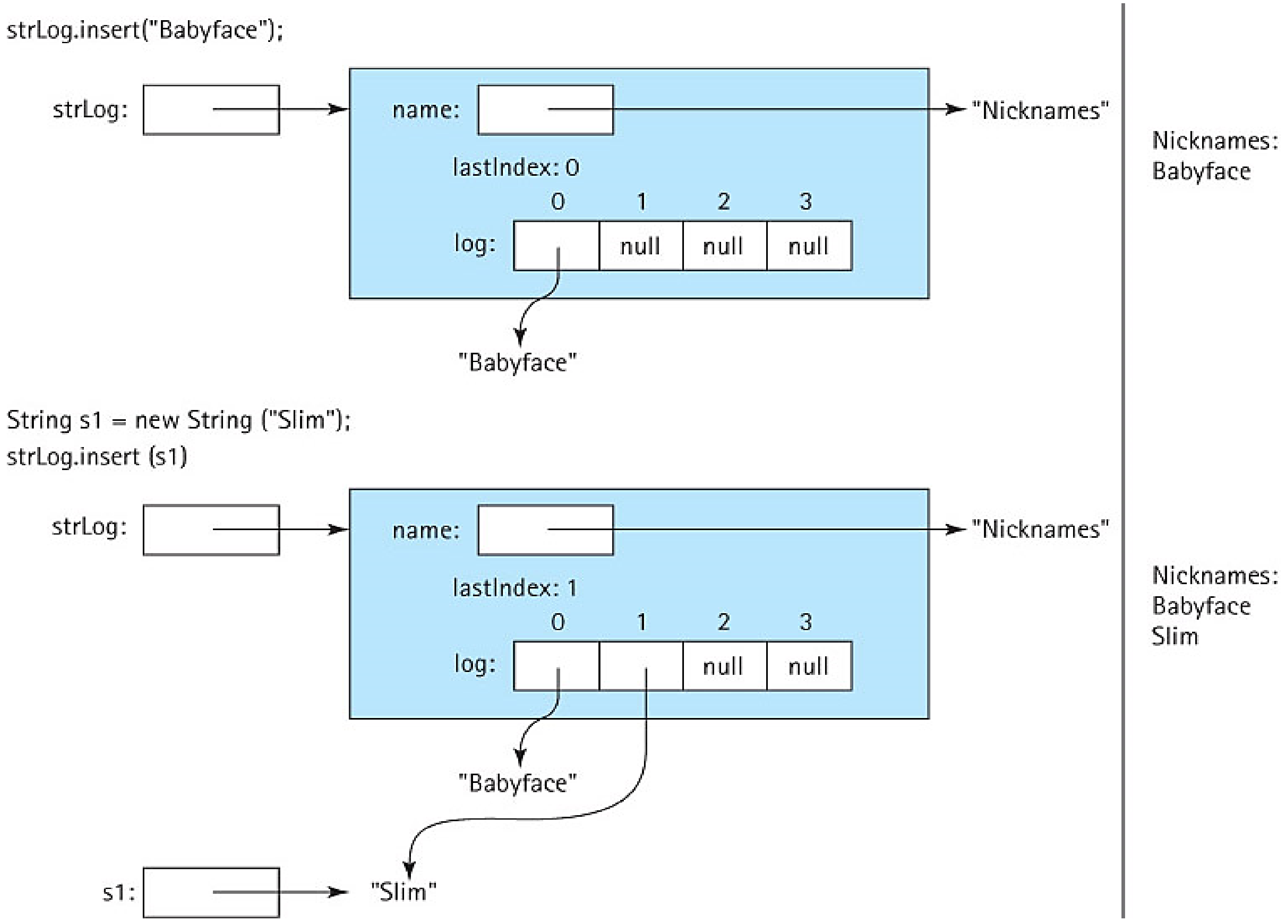
lastIndex: –1

0	1	2	3
null	null	null	null

log:

Nicknames:
<Empty>

The **`insert`** operation



```
strLog.insert("Babyface");
```

```
Nicknames:
Babyface
```

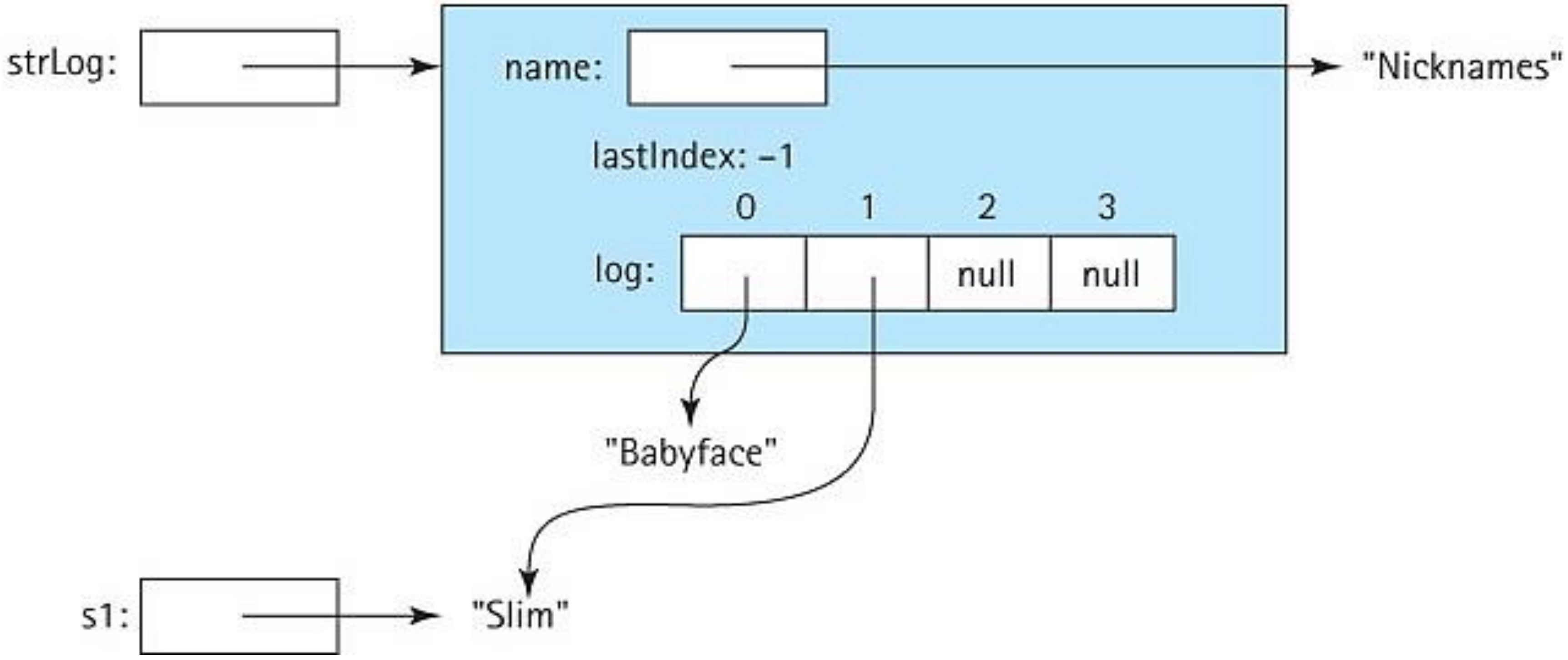
```
String s1 = new String ("Slim");
strLog.insert (s1)
```

```
Nicknames:
Babyface
Slim
```

The `clear` operation

The "lazy" approach:

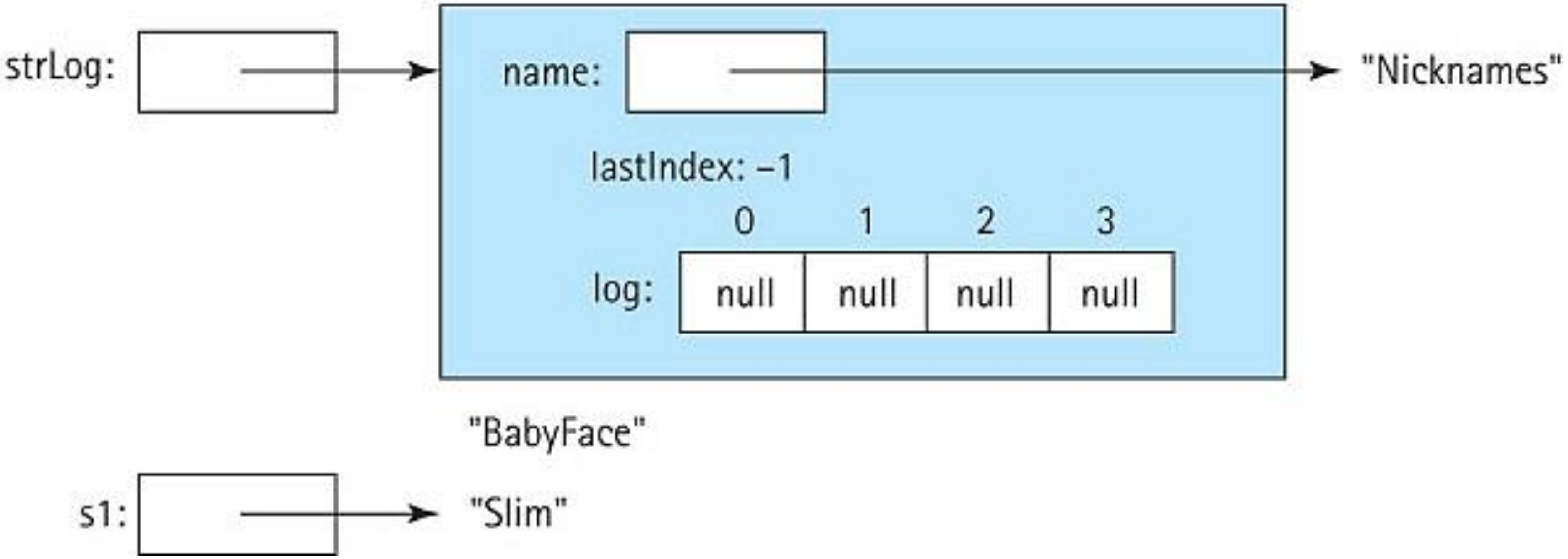
```
public void clear()
// Makes this StringLog empty.
{
  lastIndex = -1;
}
```



The `clear` operation

The "thorough" approach:

```
public void clear()
// Makes this StringLog empty.
{
  for (int i = 0; i <= lastIndex; i++)
    log[i] = null;
  lastIndex = -1;
}
```



Three observers

```
public boolean isFull()
// Returns true if this StringLog is full, otherwise returns
false.
{
  if (lastIndex == (log.length - 1))
    return true;
  else
    return false;
}

public int size()
// Returns the number of Strings in this StringLog.
{
  return (lastIndex + 1);
}

public String getName() // Returns the name of this StringLog.
{
  return name;
}
```

The **toString** observer

```
public String toString()
// Returns a nicely formatted string representing this
StringLog.
{
  String logString = "Log: " + name + "\n\n";
  for (int i = 0; i <= lastIndex; i++)
    logString = logString + (i + 1) + ". " + log[i] + "\n";
  return logString;
}
```

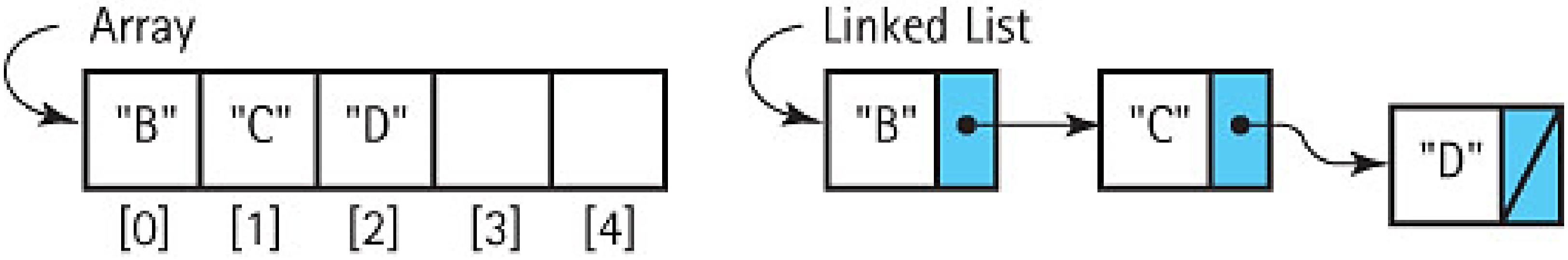
For example, if the **StringLog** is named "Top coders" and contains the strings "Jeffrey", "Sherry", and "Larry", then the result of displaying the string returned by **toString** would be

```
Log: Top coders
1. Jeffrey
2. Sherry
3. Larry
```

The `contains` method

```
public boolean contains(String element)
// Returns true if element is in this StringLog
// otherwise returns false.
// Ignores case differences when doing string comparison.
{
  int location = 0;
  while (location <= lastIndex)
  {
    // if they match
    if (element.equalsIgnoreCase(log[location]))
       return true;
    else
       location++;
  }
  return false;
}
```

Introduction to linked lists



- Arrays and linked lists are different in
    - use of memory
    - manner of access
    - language support

Nodes of a linked-list

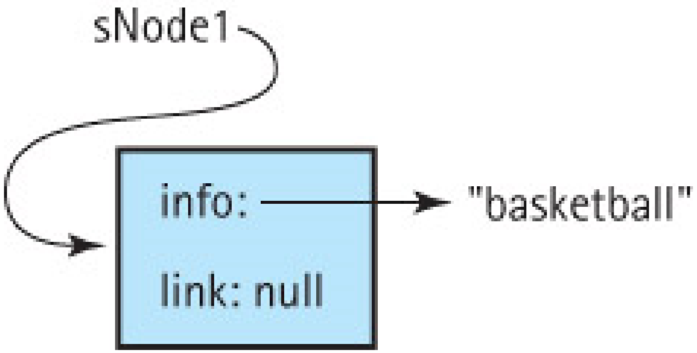
- A node in a linked list is an object that holds some important information, such as a string, plus *a link* to the exact same type of object, i.e. to an object of the same class.
- Self-referential class (recursive definition) - A class that includes an instance variable or variables that can hold a reference to an object of the same class.
- For example, to support a linked implementation of the **`StringLog`** we create the self-referential **`LLStringNode`** class.

LLStringNode Class

```
package ch02.stringLogs;

public class LLStringNode
{
  private String info;
  private LLStringNode link;

  public LLStringNode(String
                      info)
  {
    this.info = info;
    link = null;
  }

  public void setInfo(String
                      info)
  // Sets info string of this
  // LLStringNode.
  {
    this.info = info;
  }

  public String getInfo()
  // Returns info string of this
  // LLStringNode.
  {
    return info;
  }

  public void setLink(LLStringNode
                      link)
  // Sets link of this LLStringNode.
  {
    this.link = link;
  }

  public LLStringNode getLink()
  // Returns link of this
  // LLStringNode.
  {
    return link;
  }
}
```

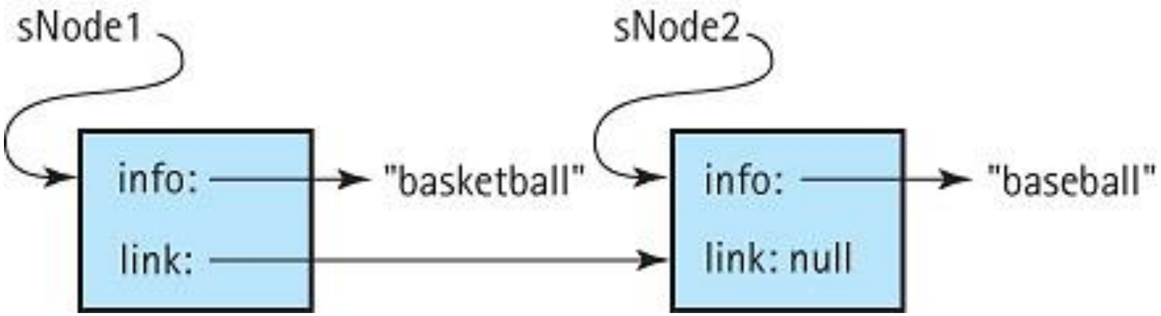
Using the **LLStringNode** Class

1: `LLStringNode sNode1 = new LLStringNode("basketball");`

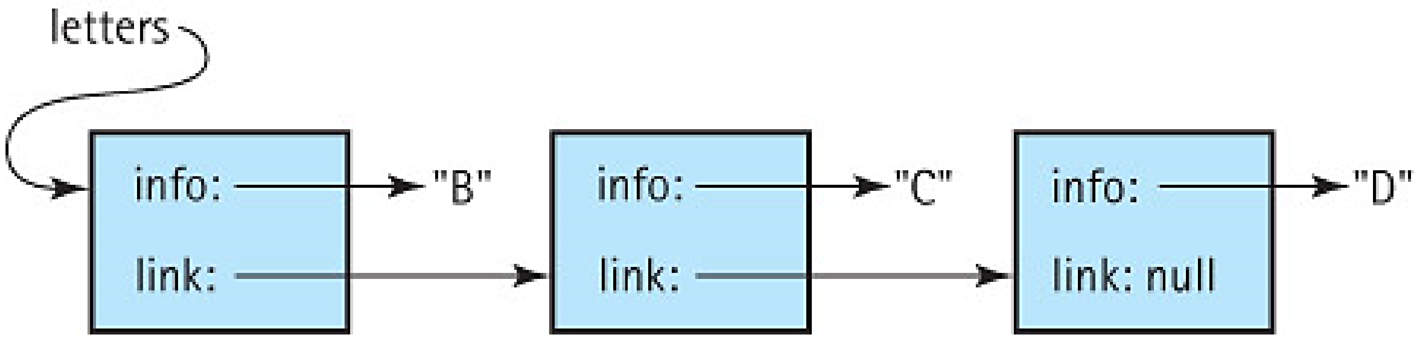


2: Suppose that in addition to **sNode1** we have **sNode2** with info "baseball" and perform

```
sNode1.setLink(sNode2);
```



Traversal of a linked list



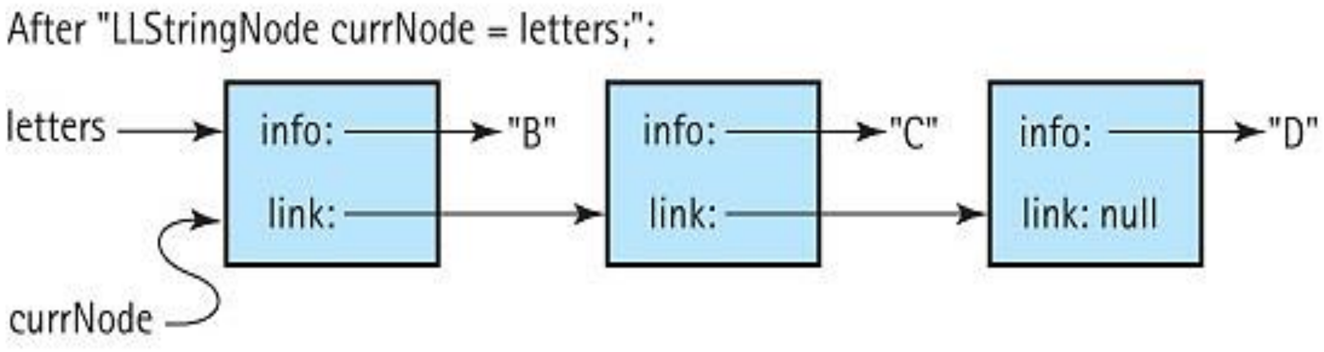
```
LLStringNode currNode = letters;
while (currNode != null)
{
  System.out.println(currNode.getInfo());
  currNode = currNode.getLink();
}
```

Tracing a traversal

```
LLStringNode currNode = letters;
while (currNode != null)
{
  System.out.println(currNode.getInfo());
  currNode = currNode.getLink();
}
```

Internal View

Output



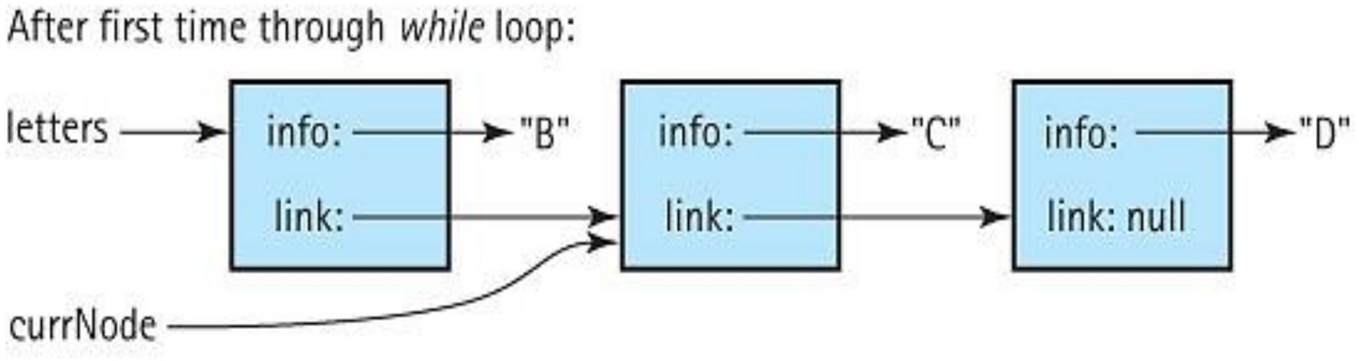
Tracing a traversal

```
LLStringNode currNode = letters;
while (currNode != null)
{
  System.out.println(currNode.getInfo());
  currNode = currNode.getLink();
}
```

Internal View

Output

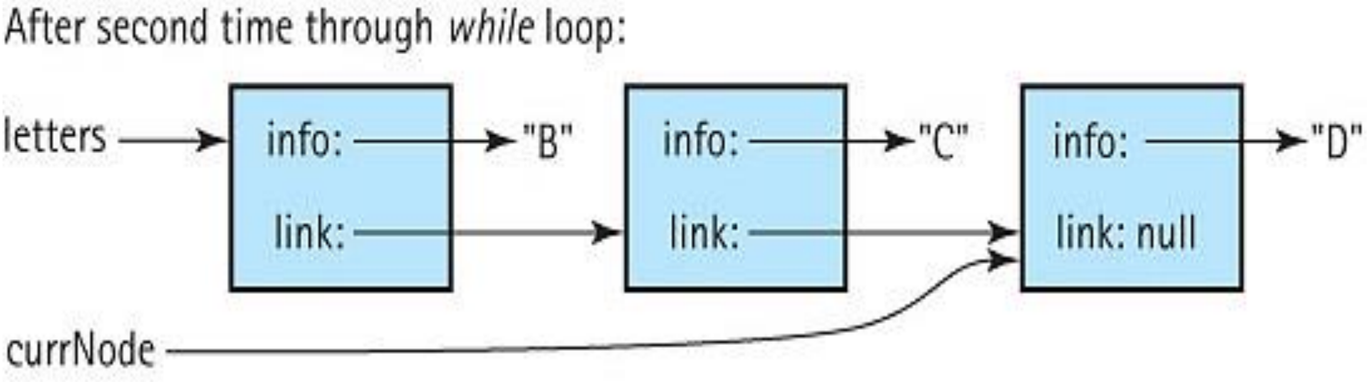
B



Tracing a traversal

```
LLStringNode currNode = letters;
while (currNode != null)
{
  System.out.println(currNode.getInfo());
  currNode = currNode.getLink();
}
```

Internal View



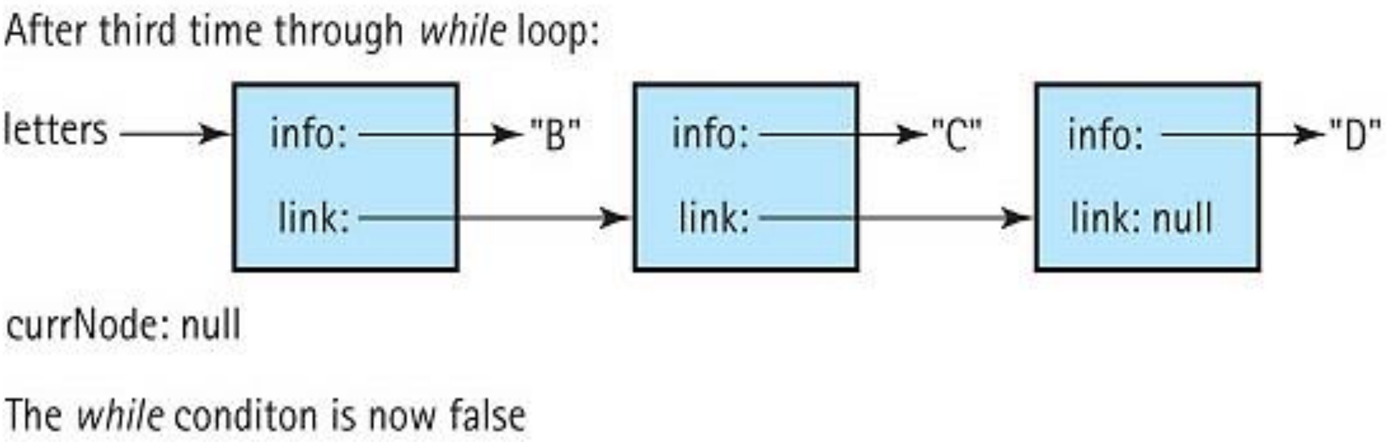
Output

```
B
C
```

Tracing a traversal

```
LLStringNode currNode = letters;
while (currNode != null)
{
  System.out.println(currNode.getInfo());
  currNode = currNode.getLink();
}
```

Internal View



After third time through *while* loop:

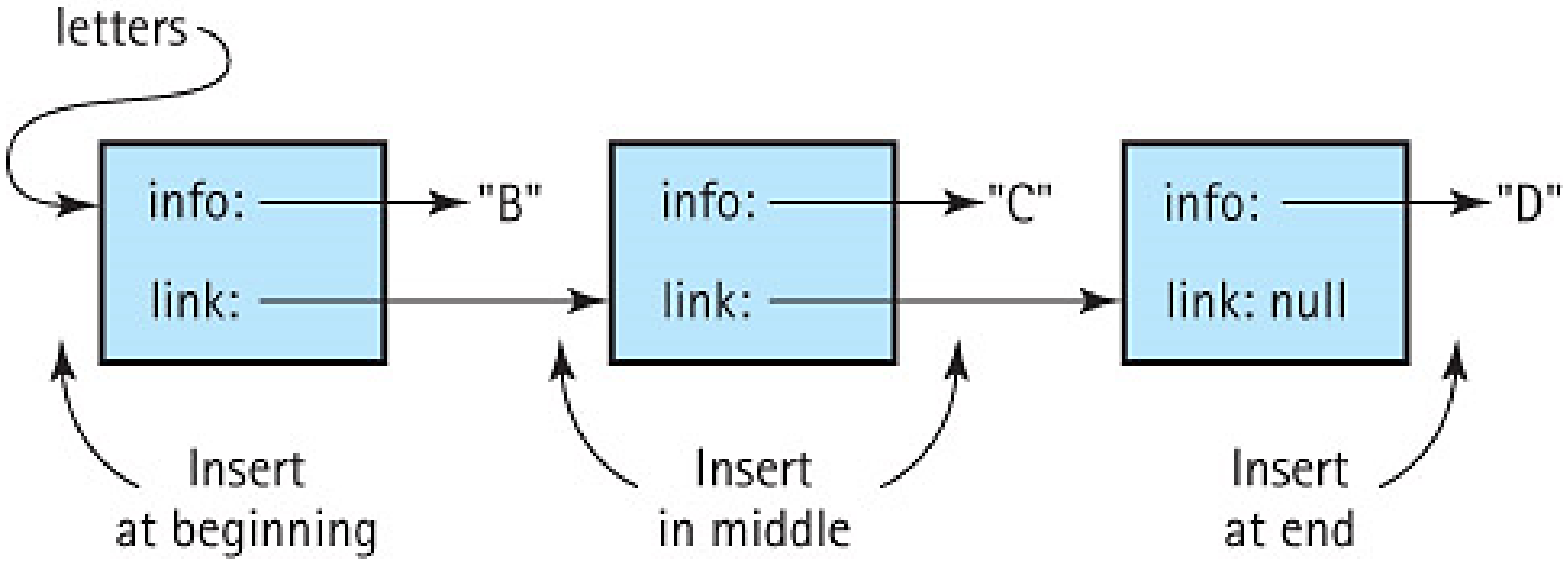
currNode: null

The *while* conditon is now false

Output

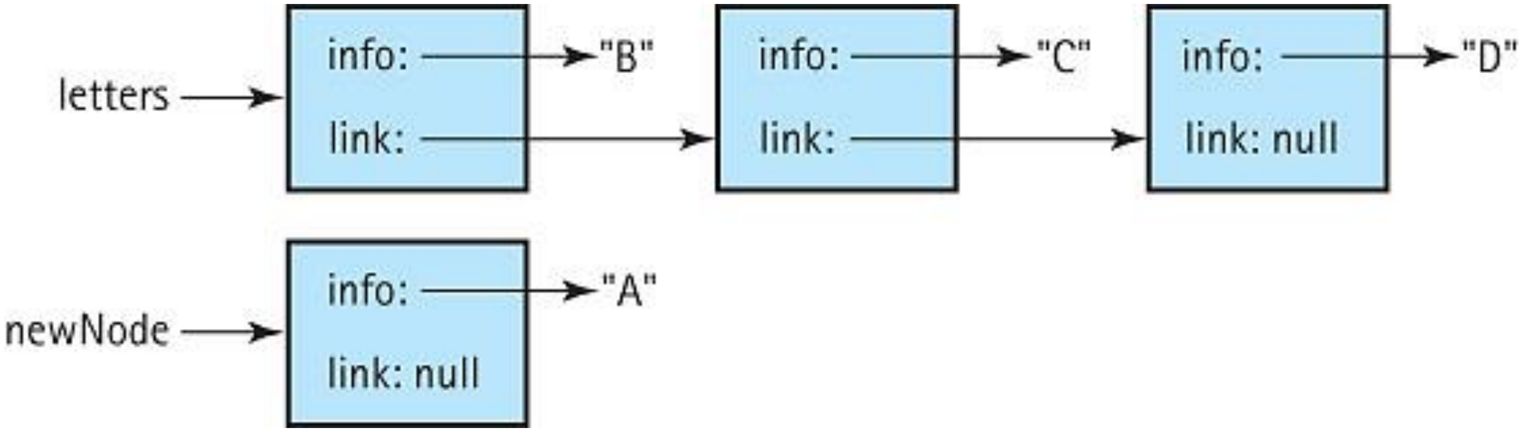
B
C
D

Three general cases of insertion



Insertion at the front

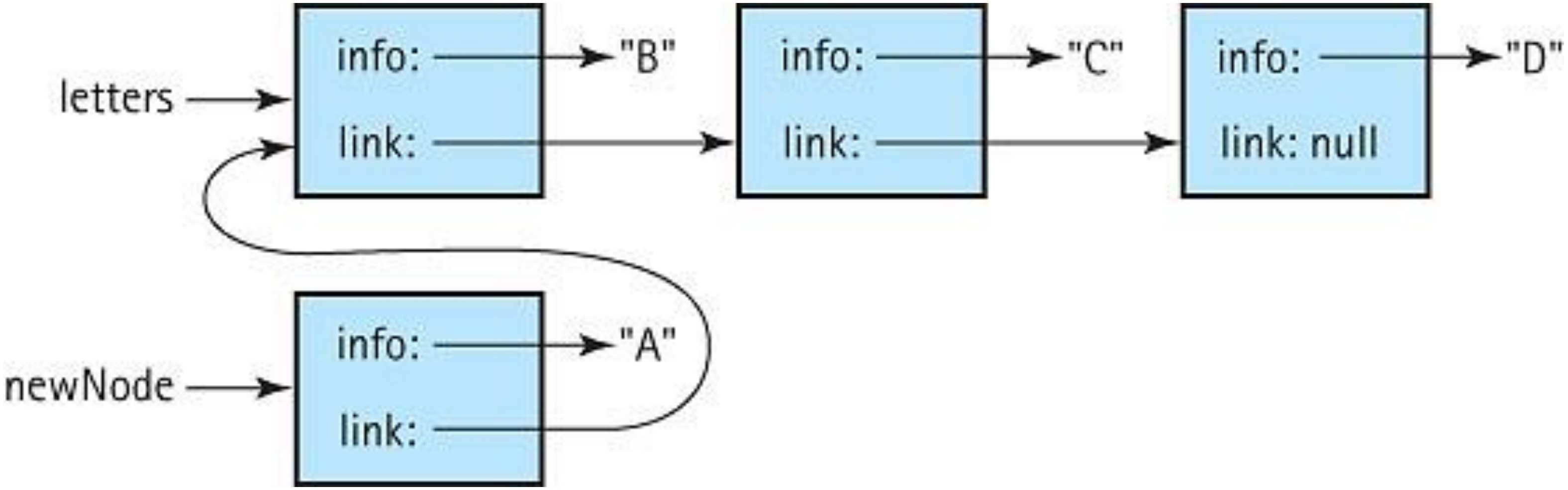
- Suppose we have the node **`newNode`** to insert into the beginning of the letters linked list:



Insertion at the front

- Our first step is to set the **`link`** variable of the **`newNode`** node to point to the beginning of the list :

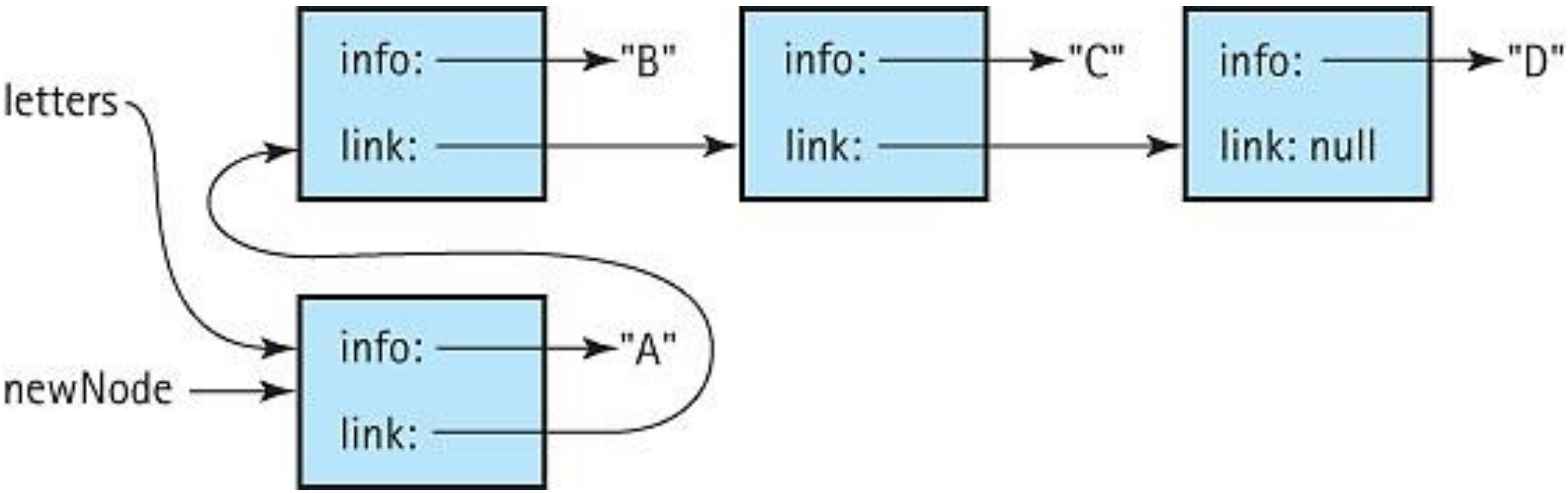
```
newNode.setLink(letters);
```



Insertion at the front

- To finish the insertion we set the **`letters`** variable to point to the **`newNode`**, making it the new beginning of the list:

```
letters = newNode;
```



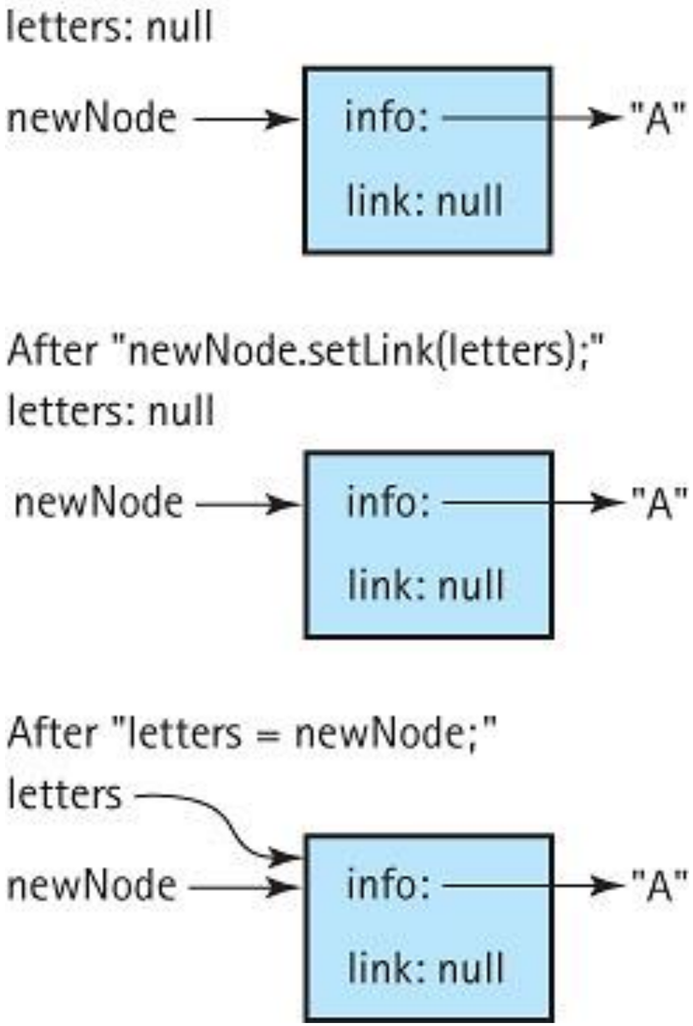
Insertion at the front of an empty list

The insertion at the front code is

```
newNode.setLink(letters);
letters = newNode;
```

What happens if our insertion code is called when the linked list is empty?

The code still works, with the new node becoming the first and only node on the linked list.



Linked list **`StringLog`** ADT implementation

- We call our new **`StringLog`** class the **`LinkedStringLog`** class, to differentiate it from the array-based class.
- We also refer to this approach as a reference-based approach.
- The class fulfills the **`StringLog`** specification and implements the **`StringLogInterface`** interface.
- Unlike the **`ArrayStringLog`**, the **`LinkedStringLog`** will implement an unbounded **`StringLog`**.

Linked list `StringLog` ADT implementation

```
package ch02.stringLogs;

public class LinkedStringLog implements StringLogInterface
{
  protected LLStringNode log; // reference to first node of
                              // linked list that holds the
                              // StringLog strings
  protected String name;      // name of this StringLog

  public LinkedStringLog(String name)
  // Instantiates and returns a reference to an empty StringLog
  // object with name "name".
  {
    log = null;
    this.name = name;
  }
```

Note that we do not need a constructor with a size parameter since this implementation is unbounded.

The `insert` operation

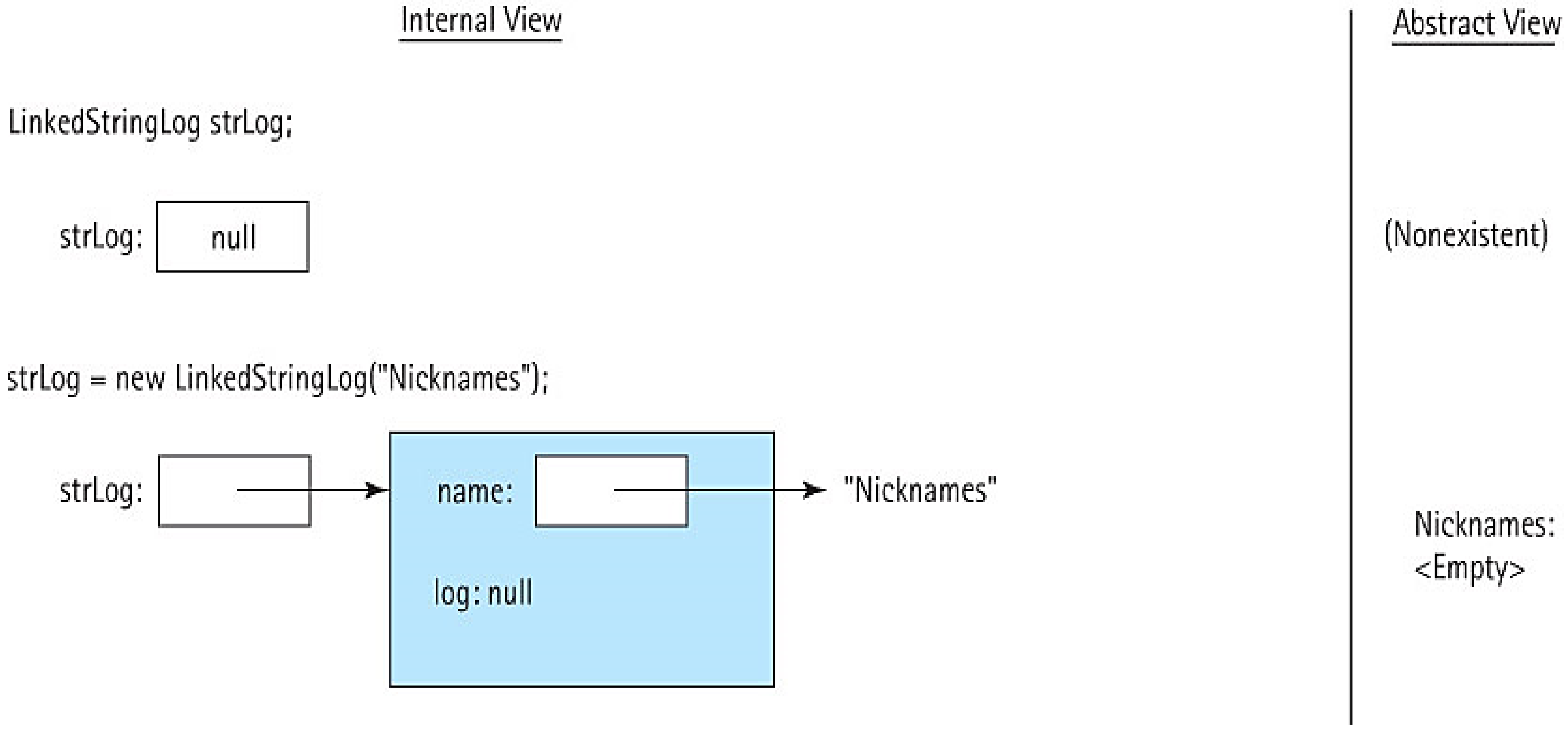
Insert the new string in the front:

```
public void insert(String element)
// Places element into this StringLog.
{
  LLStringNode newNode = new LLStringNode(element);
  newNode.setLink(log);
  log = newNode;
}
```

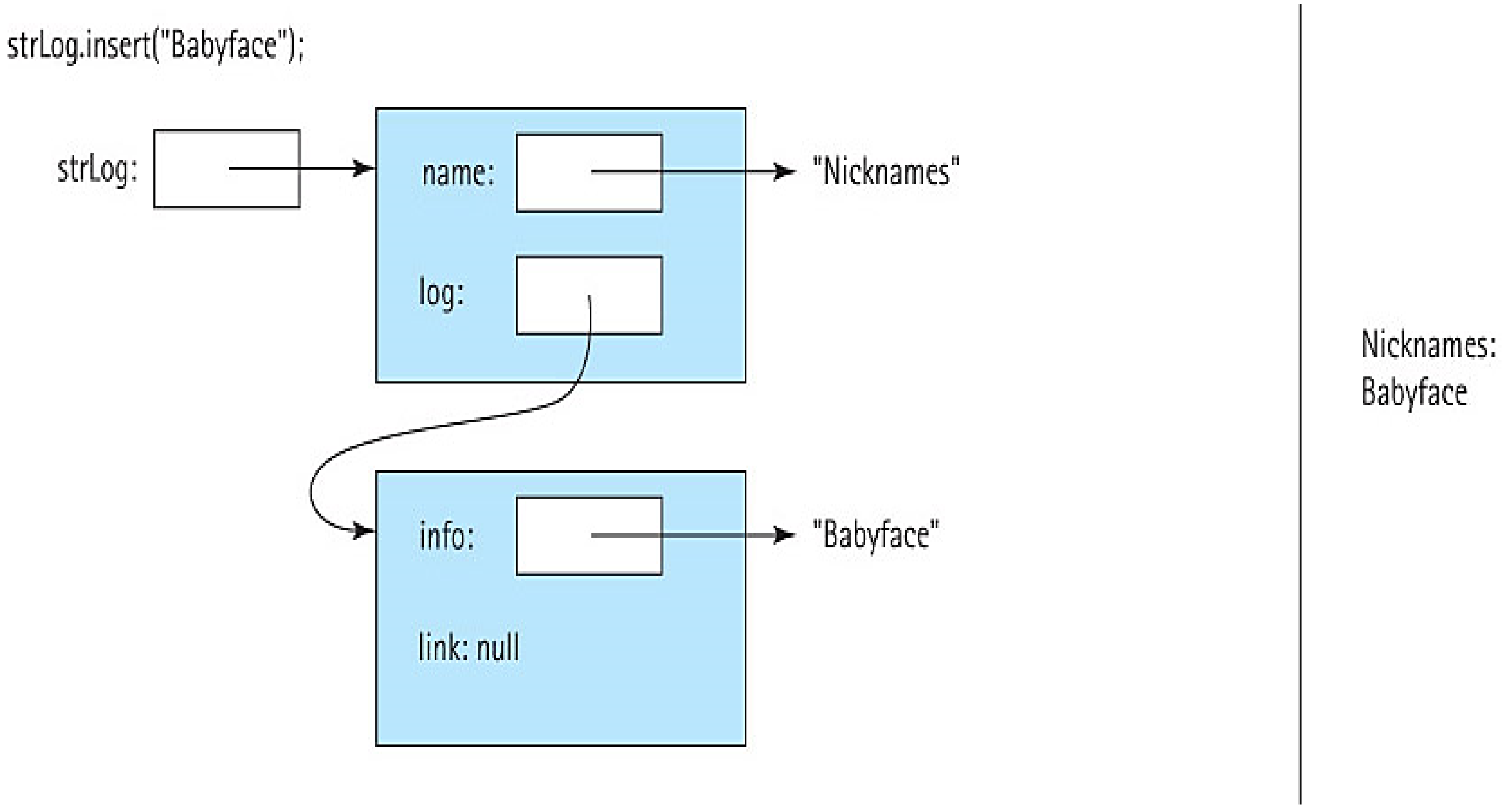
An example use:

```
LinkedStringLog strLog;
strLog = new LinkedStringLog("Nicknames");
strLog.insert("Babyface");
String s1 = new String("Slim");
strLog.insert(s1);
```

The **`insert`** operation

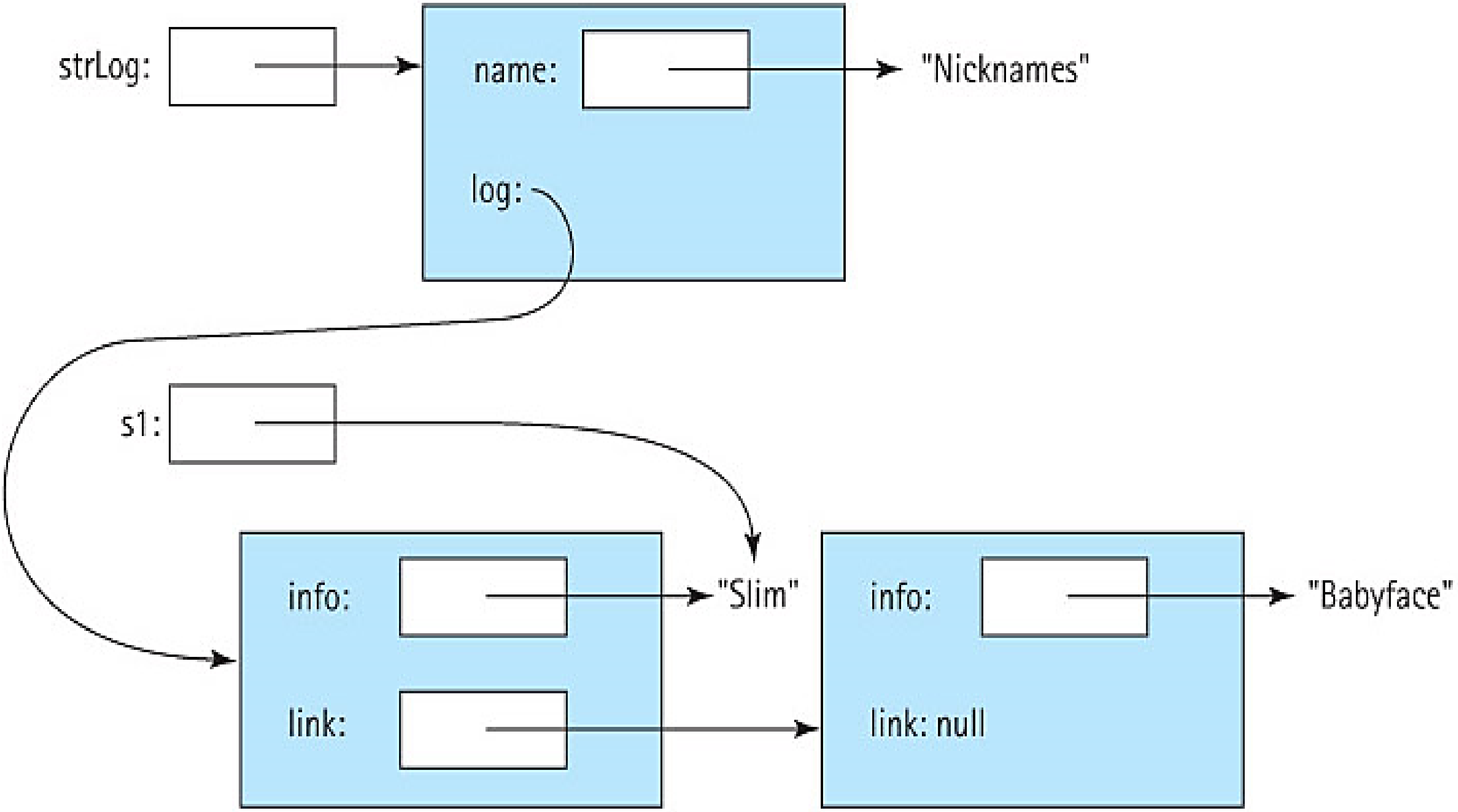


The **`insert`** operation



The `insert` operation

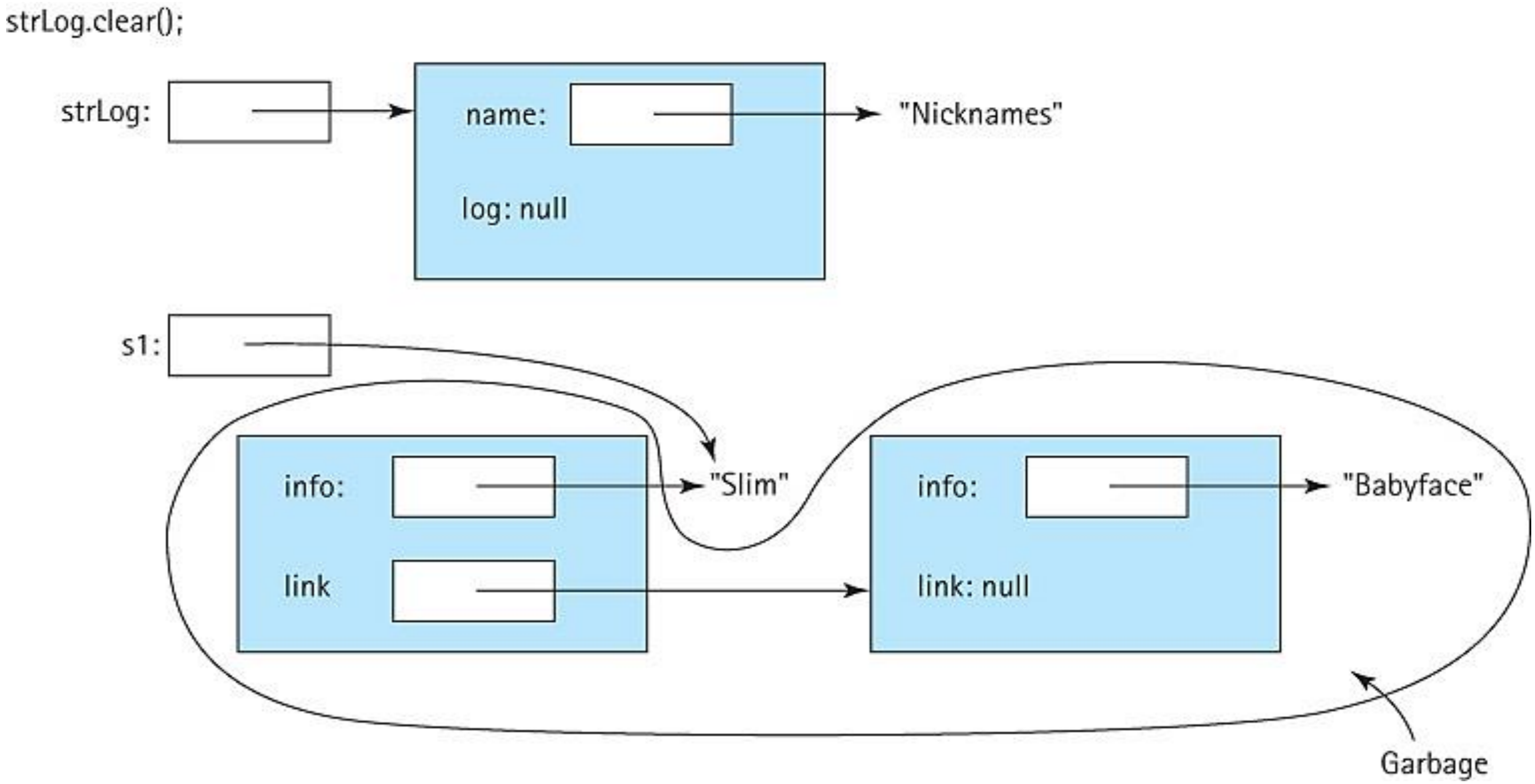
```
String s1 = new String ("Slim");
strLog.insert (s1);
```



```
Nicknames:
Babyface
Slim
```

The `clear` operation

```
public void clear()
// Makes this StringLog empty.
{
  log = null;
}
```



Three observers

```
public boolean isFull()
// Returns true if this StringLog is full, false otherwise.
{
  return false;
}

public String getName()
// Returns the name of this StringLog.
{
  return name;
}
```

Three observers

```
public int size()
// Returns the number of Strings in this StringLog.
{
  int count = 0;
  LLStringNode node;
  node = log;
  while (node != null)
  {
    count = count + 1;
    node = node.getLink();
  }
  return count;
}
```

The `toString` observer

```
public String toString()
// Returns a nicely formatted string representing this
// StringLog.
{
  String logString = "Log: " + name + "\n\n";
  LLStringNode node;
  node = log;
  int count = 0;

  while (node != null)
  {
    count = count + 1;
    logString = logString + count + ". "
                + node.getInfo() + "\n";
    node = node.getLink();
  }

  return logString;
}
```

The `contains` observer

- We reuse our design from the array-based approach, but use the linked list counterparts of each operation:

```
public boolean contains(String element)
{
  LLStringNode node;
  node = log;
  while (node != null)
  {
    // if they match
    if (element.equalsIgnoreCase(node.getInfo()))
      return true;
    else
      node = node.getLink();
  }
  return false;
}
```

Action items

- Read book chapter 2.