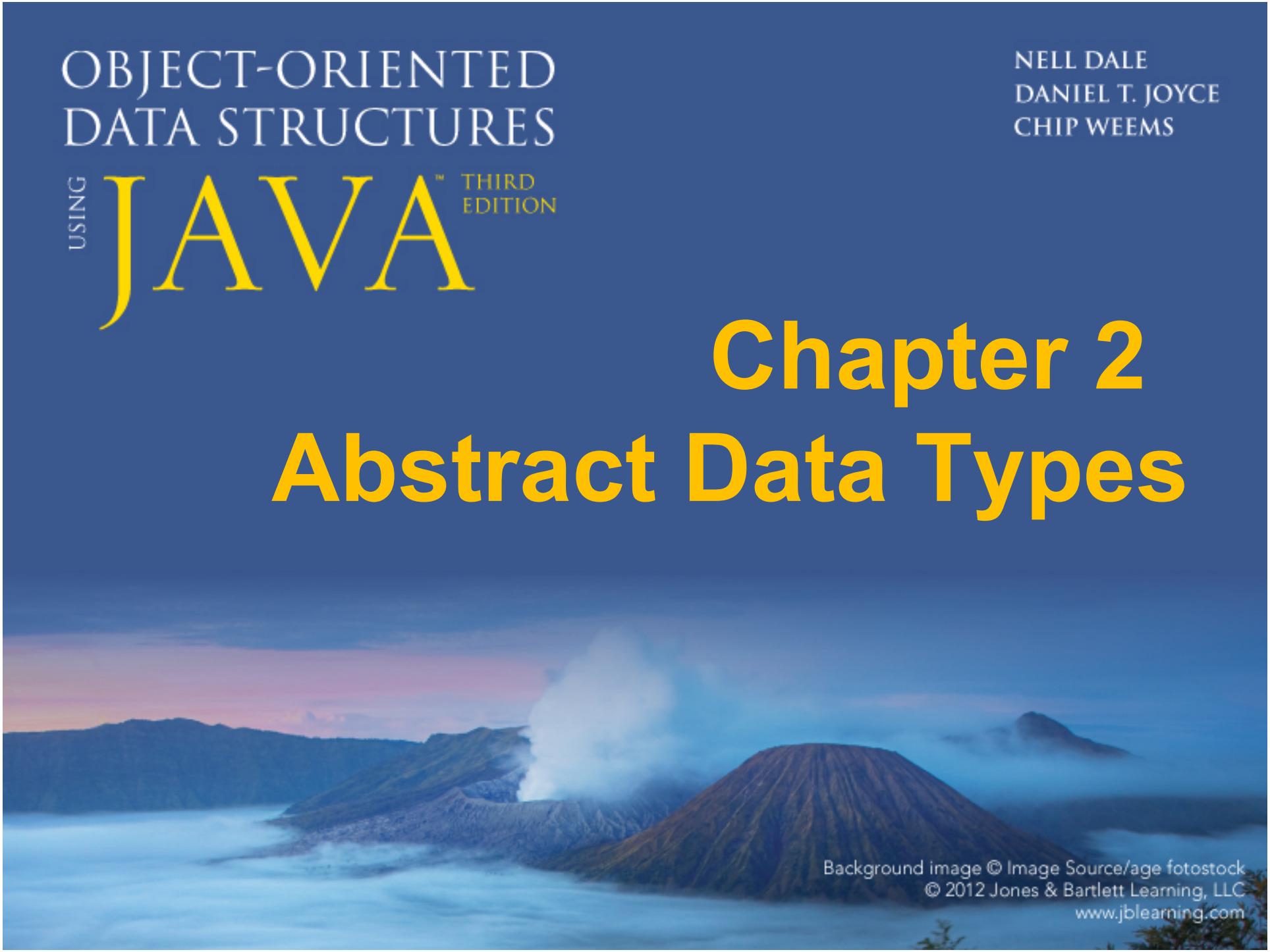

OBJECT-ORIENTED
DATA STRUCTURES
USING JAVA™ THIRD EDITION
NELL DALE
DANIEL T. JOYCE
CHIP WEEMS
Chapter 2
Abstract Data Types
Background image © Image Source/age fotostock
© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

Chapter 2: Abstract Data Types

2.1 – Abstraction
2.2 – The StringLog ADT Specification
2.3 – Array-Based StringLog ADT Implementation
2.4 – Software Testing
2.5 – Introduction to Linked Lists
2.6 – Linked List StringLog ADT Implementation
2.7 – Software Design: Identification of Classes
2.8 – Case Study: A Trivia Game

2.1 Abstraction

- **Abstraction** A model of a system that includes only the details essential to the perspective of the viewer of the system
- **Information hiding** The practice of hiding details within a module with the goal of controlling access to the details from the rest of the system
- **Data abstraction** The separation of a data type's logical properties from its implementation
- **Abstract data type (ADT)** A data type whose properties (domain and operations) are specified independently of any particular implementation

ADT Perspectives or Levels

- *Application (or user or client) level*: We use the ADT to solve a problem. When working at this level we only need to know how to create instances of the ADT and invoke its operations.
- *Logical (or abstract) level*: Provides an abstract view of the data values (the domain) and the set of operations to manipulate them. At this level, we deal with the "*what*" questions. What is the ADT? What does it model? What are its responsibilities? What is its interface?
- *Implementation (or concrete) level*: Provides a specific representation of the structure to hold the data and the implementation of the operations. Here we deal with the "*how*" questions.

Preconditions and Postconditions

- **Preconditions** Assumptions that must be true on entry into a method for it to work correctly
- **Postconditions or Effects** The results expected at the exit of a method, assuming that the preconditions are true
- We specify pre- and postconditions for a method in a comment at the beginning of the method

Java: Abstract Method

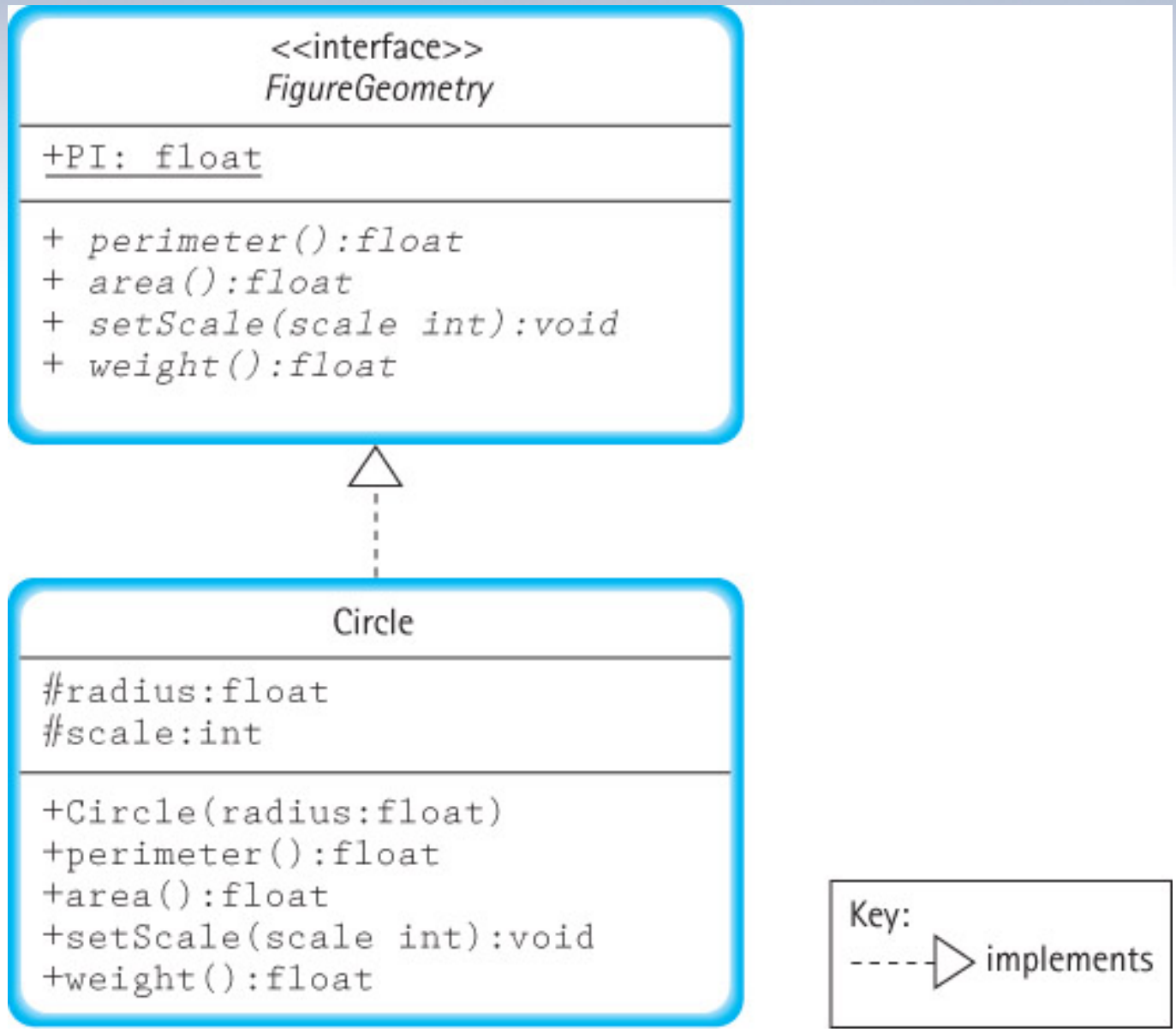
- Only includes a description of its parameters
- No method bodies or implementations are allowed.
- In other words, only the *interface* of the method is included.

Java Interfaces

- Similar to a Java class
  - can include variable declarations
  - can include methods
- However
  - Variables must be constants
  - Methods must be abstract.
  - A Java interface cannot be instantiated.
- We can use an interface to formally specify the logical level of an ADT:
  - It provides a template for classes to fill.
  - A separate class then "implements" it.
- For example, see the `FigureGeometry` interface (next slide) and the `Circle` class that implements it (following slide)

```
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();
  // Precondition: Scale of this figure has been set.
  //
  // Returns weight of this figure. Weight = area X scale.
}
```

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

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

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

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

  public void setScale(int scale)
    // Scale of this figure
    // is set to "scale".
    {
      this.scale = scale;
    }

    public float weight()
    // Precondition: Scale of this figure
    // has been set.
    //
    // Returns weight of this figure.
    // Weight = area X scale.
    {
      return(this.area() * scale);
    }
}
```



Benefits

- We can formally check the syntax of our specification. When we compile the interface, the compiler uncovers any syntactical errors in the method interface definitions.
- We can formally verify that the interface “contract” is met by the implementation. When we compile the implementation, the compiler ensures that the method names, parameters, and return types match what was defined in the interface.
- We can provide a consistent interface to applications from among alternate implementations of the ADT.

2.2 The StringLog ADT Specification

- The primary responsibility of the StringLog ADT is to remember all the strings that have been inserted into it and, when presented with any given string, indicate whether or not an identical string has already 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
  - A constructor creates a new instance of the ADT. It is up to the implementer of the StringLog to decide how many, and what kind, of constructors to provide.
- Transformers
  - insert(String element): assumes the StringLog is not full; adds element to the log of strings.
  - clear: resets the StringLog to the empty state; the StringLog retains its name.

StringLog Methods

- Observers
  - contains(String element): returns true if element is in the StringLog, false otherwise; We ignore case when comparing the strings.
  - size: returns the number of elements currently held in the StringLog.
  - isFull: returns whether or not the StringLog is full.
  - getName: returns the name attribute of the StringLog.
  - toString: returns a nicely formatted string that represents the entire contents of the StringLog.

The StringLogInterface

```
//--------------------------------------------------------------------
// StringLogInterface.java     by Dale/Joyce/Weems            Chapter 2
//
// Interface for a class that implements a log of Strings.
// A log "remembers" the elements placed into it.
//
// A log must have a "name".
//--------------------------------------------------------------------

package ch02.stringLogs;

public interface StringLogInterface
{
  void insert(String element);
  // Precondition:   This StringLog is not full.
  //
  // Places element into this StringLog.

  boolean isFull();
  // Returns true if this StringLog is full, otherwise returns false.
```

The StringLogInterface continued

```
  int size();
  // Returns the number of Strings in this StringLog.

  boolean contains(String element);
  // Returns true if element is in this StringLog,
  // otherwise returns false.
  // Ignores case differences when doing string comparison.

  void clear();
  // Makes this StringLog empty.

  String getName();
  // Returns the name of this StringLog.

  String toString();
  // Returns a nicely formatted string representing this StringLog.
}
```

Application Example

```
//-------------------------------------------------------------------------
// UseStringLog.java        by Dale/Joyce/Weems                 Chapter 2
//
// Simple example of the use of a StringLog.
//-------------------------------------------------------------------------
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"));
  }
}
```

Output from example

```
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
```

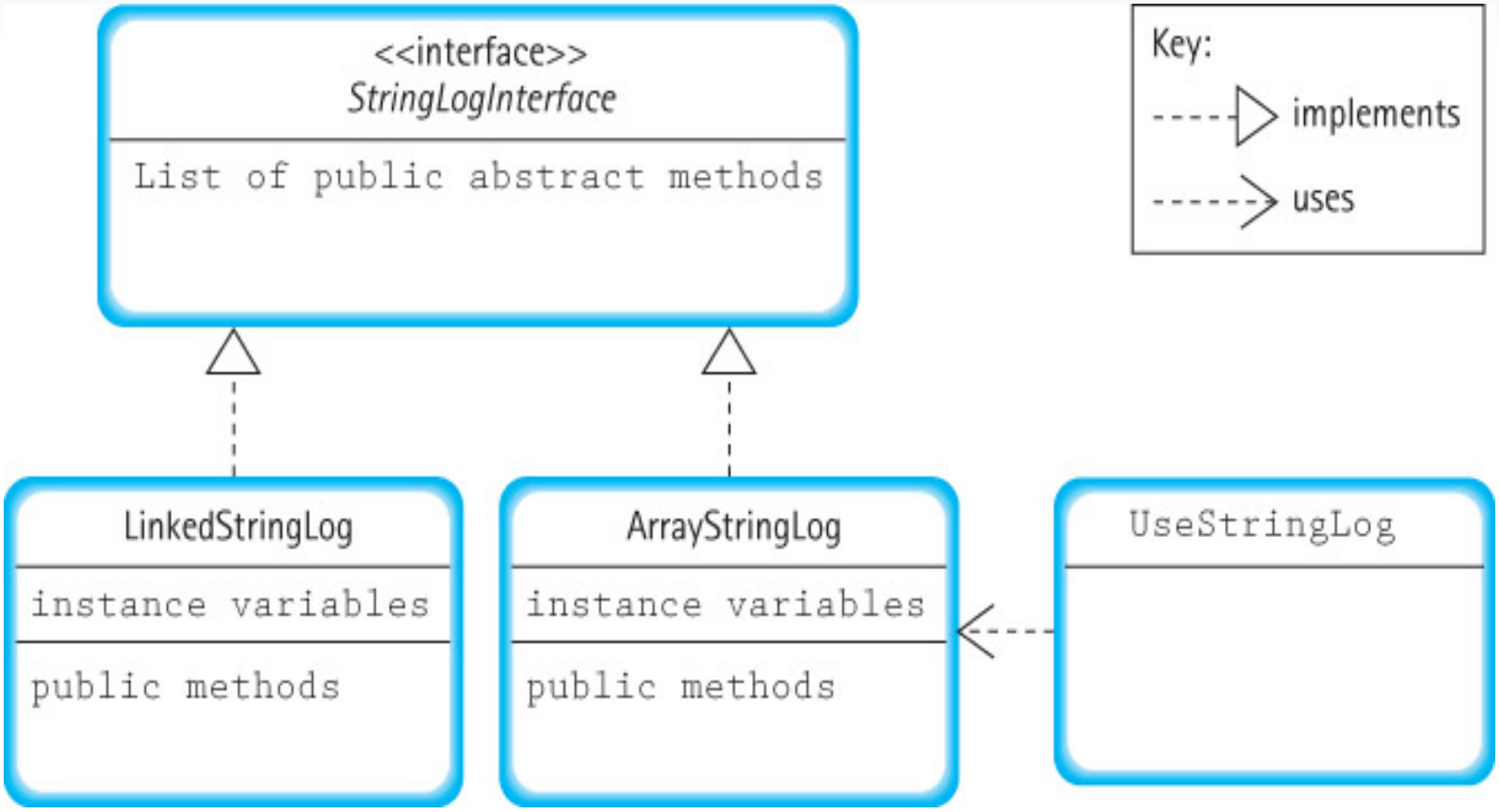
Review: the three levels

- *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.

- *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 developed in Section 2.3 provides a specific implementation of the StringLog ADT, fulfilling the contract presented by the StringLogInterface. It is used by applications such as UseStringLog. Likewise, the LinkedStringLog class (see Section 2.6) also provides an implementation.

Relationships among StringLog classes



2.3 Array-Based StringLog ADT Implementation

- Class name: ArrayStringLog
- Distinguishing feature: strings are stored sequentially, in adjacent slots in an array
- Package: ch02.stringLogs (same as StringLogInterface)

Instance Variables

- String[] log;
  - The elements of a StringLog are stored in an array of String objects named log.
- int lastIndex = -1;
  - Originally the array is empty. Each time the insert command is invoked another string is added to the array. We use this variable to track the index of the "last" string inserted into the array.
- String name;
  - Recall that every StringLog must have a *name.* We call the needed variable name.

Instance variables and constructors

```
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

public ArrayStringLog(String name, int maxSize)
// Precondition:   maxSize > 0
//
// Instantiates and returns a reference to an empty StringLog object
// with name "name" and room for maxSize strings.
{
  log = new String[maxSize];
  this.name = name;
}

public ArrayStringLog(String name)
// Instantiates and returns a reference to an empty StringLog object
// with name "name" and room for 100 strings.
{
  log = new String[100];
  this.name = name;
}
```

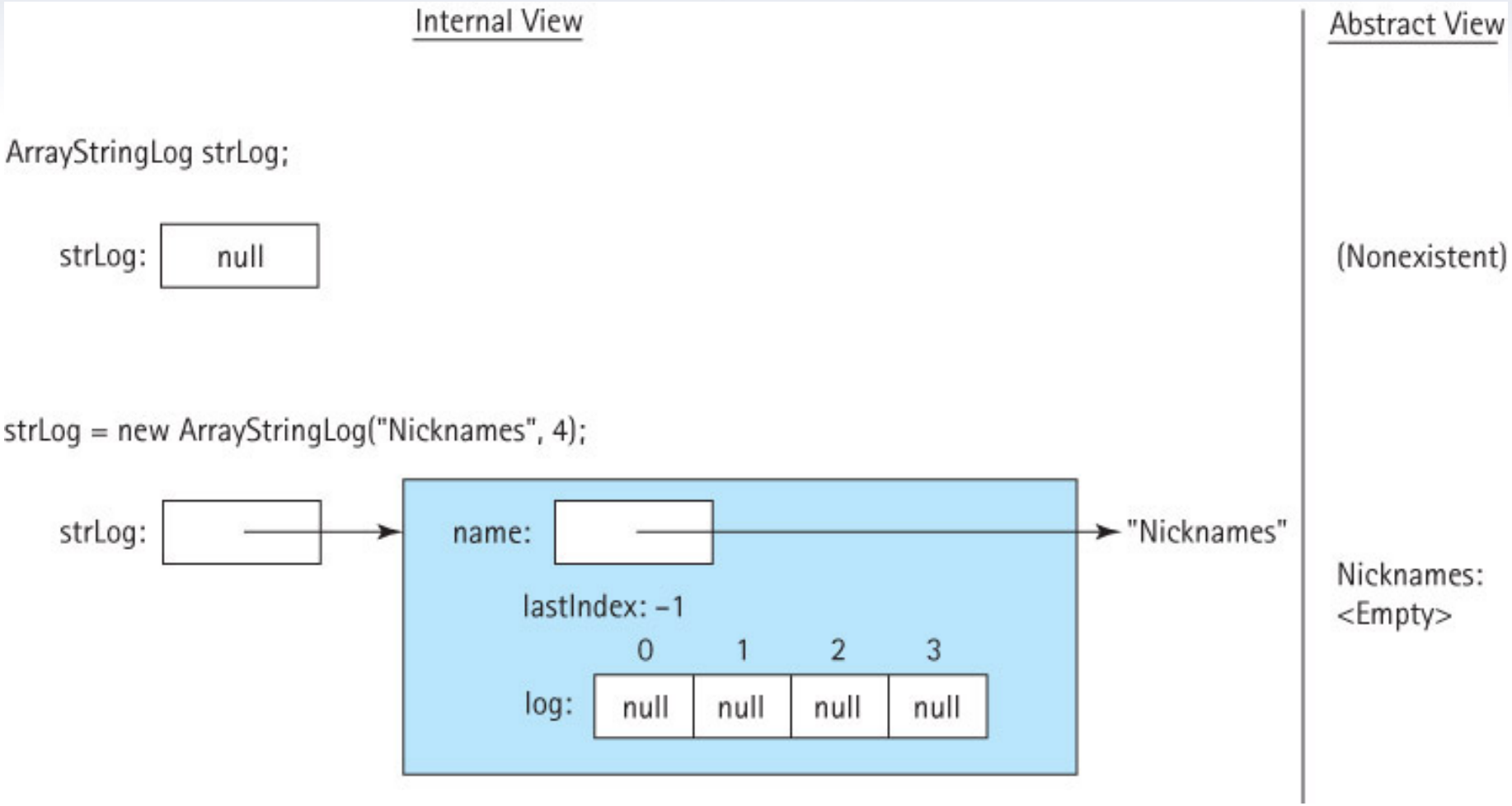
The insert operation

```
public void insert(String element)
// Precondition:   This StringLog is not full.
//
// Places element into this StringLog.
{
  lastIndex++;
  log[lastIndex] = element;
}
```

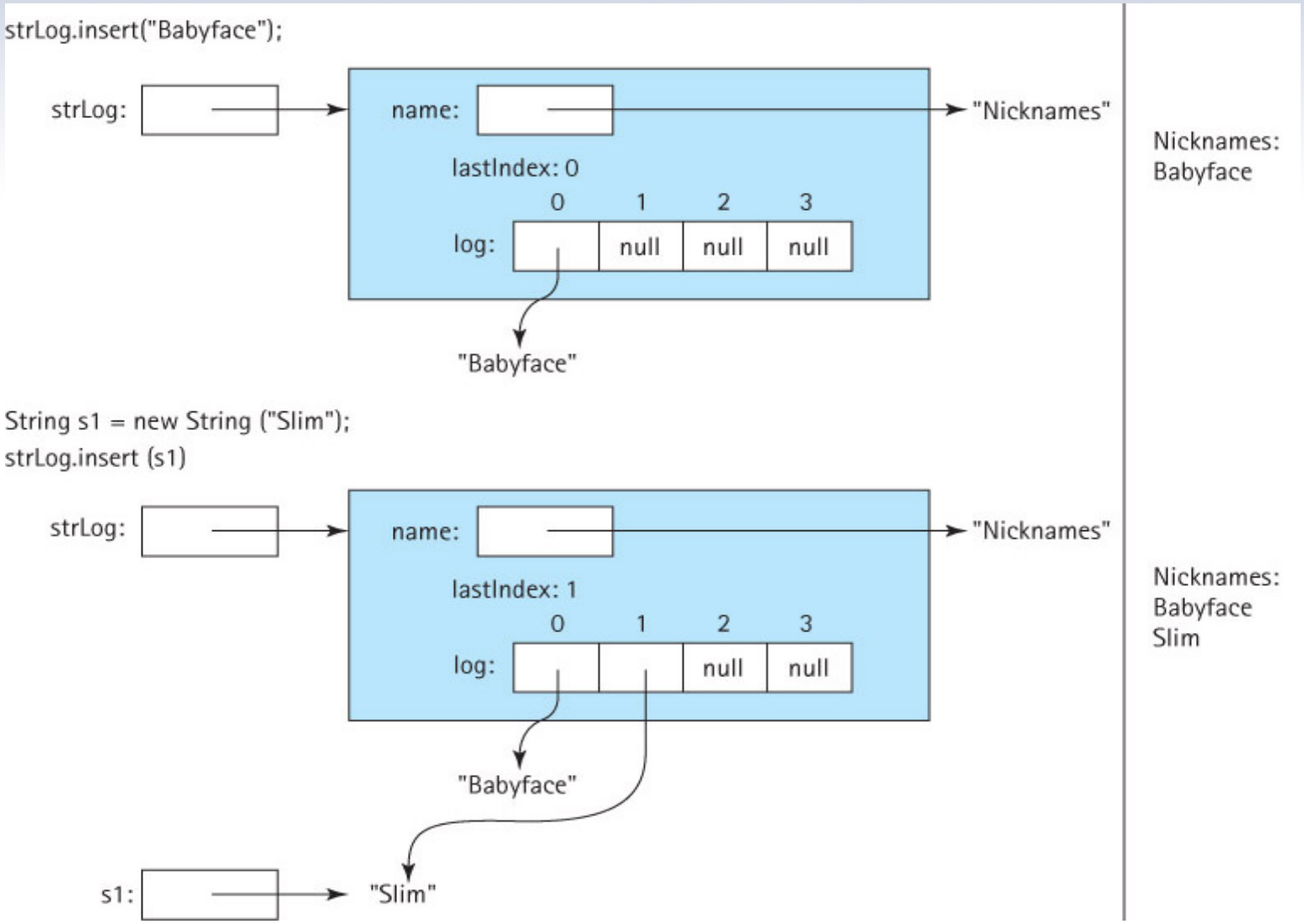
An example use:

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

Example use of insert



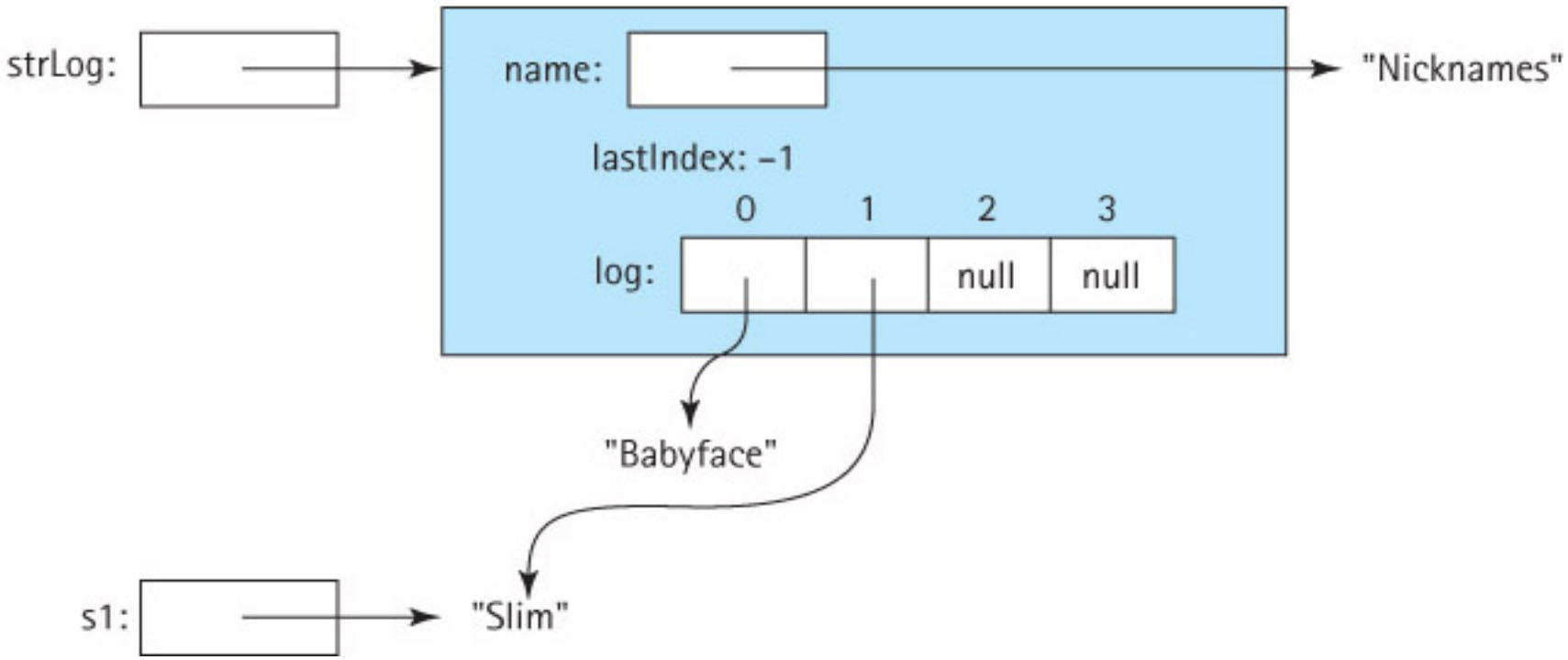
Example use of insert continued



The clear operation

The “lazy” approach:

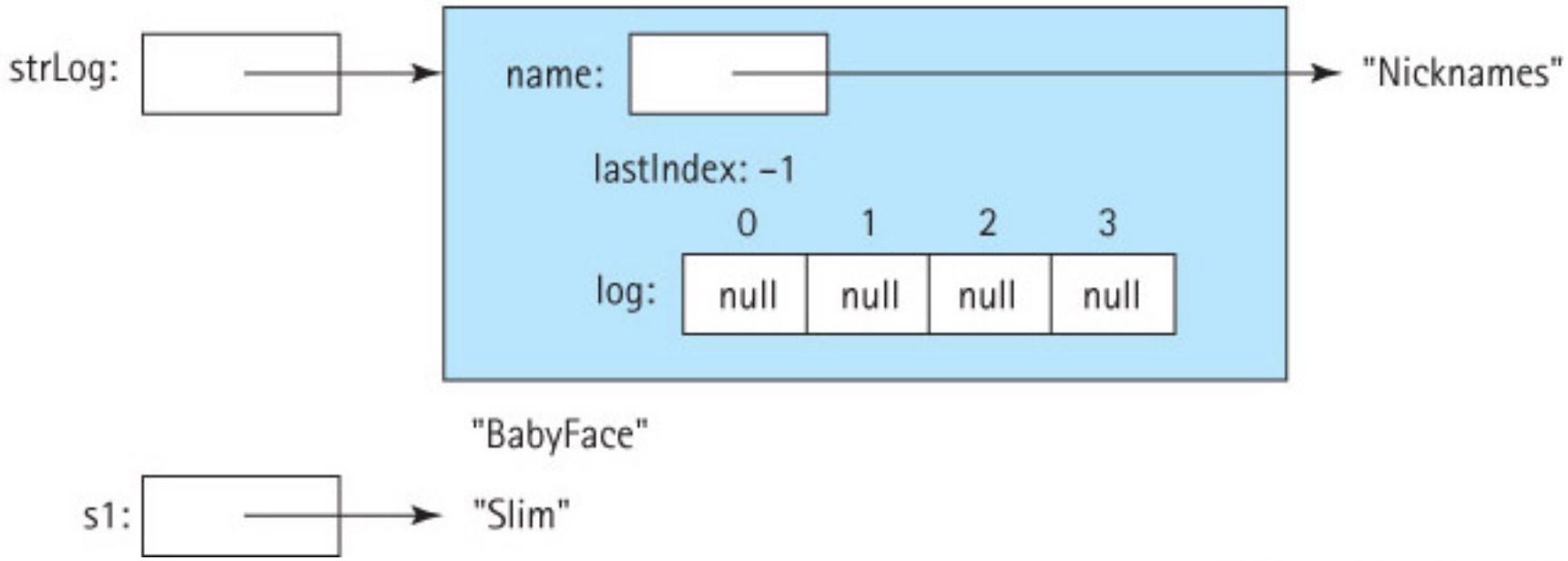
```
public void clear()
// Makes this StringLog empty.
{
 lastIndex = -1;
}
```



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 “Three Stooges” and contains the strings “Larry”, “Moe”, and “Curly Joe”, then the result of displaying the string returned by toString would be

```
Log: Three Stooges
1. Larry
2. Moe
3. Curly Joe
```

Stepwise Refinement

- Approach a problem in stages.
- Similar steps are followed during each stage, with the only difference being the level of detail involved.
- The completion of each stage brings us closer to solving our problem.
- There are two standard variations of stepwise refinement:
  - Top-down: The problem is broken into several large parts. Each part is in turn divided into sections, then the sections are subdivided, and so on. *Details are deferred as long as possible.* The top-down approach is often used for the design of non-trivial methods.
  - Bottom-up: Details come first. They are brought together into increasingly higher-level components. A useful approach if you can identify previously created program components to reuse in creating your system.

contains method:
top-down stepwise refinement phase 1

```
public boolean contains(String element)
{
  Set variables
  while (we still need to search)
  {
    Check the next value
  }
  return (whether or not we found the element)
}
```

A combination of a programming language with a natural language, such as we use here, is called *pseudocode* and is a convenient means for expressing algorithms.

contains method:
top-down stepwise refinement phase 2

```
public boolean contains(String element)
{
  Set variables;
  while (we still need to search)
  {
    if (the next value equals element)
      return true;
  }
  return false;
}
```

contains method:
top-down stepwise refinement phase 3

```
public boolean contains(String element)
{
  int location = 0;
  while (we still need search)
  {
    if (element.equalsIgnoreCase(log[location]))
      return true;
    else
      location++;
  }
  return false;
}
```

contains method:
top-down stepwise refinement phase 4

```
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 (element.equalsIgnoreCase(log[location]))  // if they match
      return true;
    else
      location++;
  }
  return false;
}
```

2.4 Software Testing

- The process of executing a program with data sets designed to discover errors
- Software testing is one facet of software verification.

Verification and Validation

- **Software validation** The process of determining the degree to which software fulfills its intended purpose
- **Software verification** The process of determining the degree to which a software product fulfills its specifications
- **Deskchecking** Tracing an execution of a design or program on paper
- **Walk-through** A verification method in which a team performs a manual simulation of the program or design
- **Inspection** A verification method in which one member of a team reads the program or design line by line and the others point out errors

Test cases

- The software testing process requires us to devise a set of test cases that, taken together, allow us to assert that a program works correctly.
- For each test case, we must:
  - determine inputs that represent the test case
  - determine the expected behavior of the program for the given input
  - run the program and observe the resulting behavior
  - compare the expected behavior and the actual behavior of the program

Identifying test cases

- **Functional domain** The set of valid input data for a program or method
- In those limited cases where the functional domain, is extremely small, one can verify a program unit by testing it against every possible input element.
- This *exhaustive testing*, can prove conclusively that the software meets its specifications.
- In most cases, however, the functional domain is very large, so exhaustive testing is almost always impractical or impossible.

Identifying test cases

- Cover general dimensions of data.
- Within each dimension identify categories of inputs and expected results.
- Test at least one instance of each combination of categories across dimensions.
- Testing like this is called **black-box testing**. The tester must know the external interface to the module—its inputs and expected outputs—but does not need to consider what is being done inside the module (the inside of the black box).

Identifying test cases - example

- Identified dimensions and categories for the contains method of the StringLog ADT could be:
  - Expected result: true, false
  - Size of StringLog: empty, small, large, full
  - Properties of element: small, large, contains blanks
  - Properties of match: perfect match, imperfect match where character cases differ
  - Position of match: first string placed in StringLog, last string placed in StringLog, "middle" string placed in StringLog
- From this list we can identify dozens of test cases, for example a test where the expected result is true, the StringLog is full, the element contains blanks, it's an imperfect match, and the string being matched was the "middle" string placed into the StringLog.

More on Testing

- **Test plan** A document showing the test cases planned for a program or module, their purposes, inputs, expected outputs, and criteria for success
- **Test driver** A program that calls operations exported from a class, allowing us to test the results of the operations

Pseudocode for an Interactive Test Driver for an ADT Implementation

```
Prompt for, read, and display test name
Determine which constructor to use, obtain any needed
  parameters, and instantiate a new instance of the ADT
while (testing continues)
{
  Display a menu of operation choices, one choice for each
    method exported by the ADT implementation, plus a "show
    contents" choice, plus a "stop Testing" choice
  Get the user's choice and obtain any needed parameters
  Perform the chosen operation
}
```

The ITDArrayStringLog program ("ITD" stands for "Interactive Test Driver") is a test driver based on the above pseudocode for our ArrayStringLog class.

Try it out!

Professional Testing

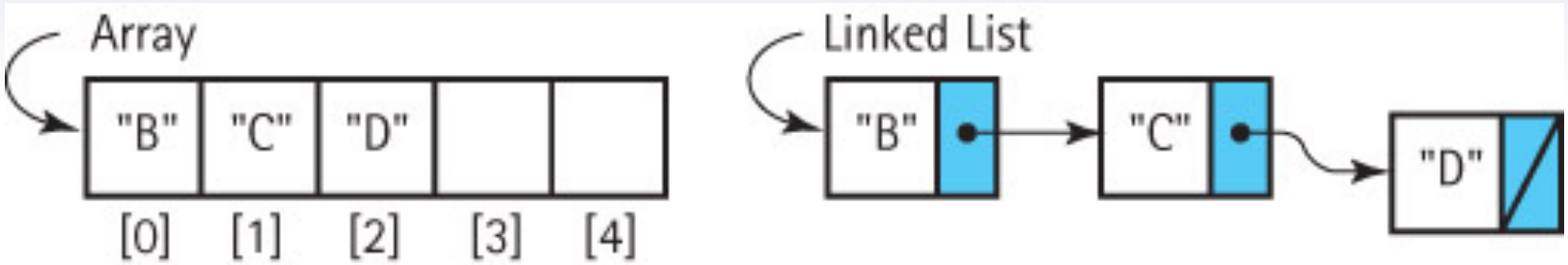
- In a production environment where hundreds or even thousands of test cases need to be performed, an interactive approach can be unwieldy to use. Instead, automated test drivers are created to run in batch mode.
- For example, here is a test case for the contains method:

```
import ch02.stringLogs.*;

public class Test034
{
  public static void main(String[] args)
  {
    ArrayStringLog log = new ArrayStringLog("Test 34");
    log.insert("trouble in the fields");
    log.insert("love at the five and dime");
    log.insert("once in a very blue moon");
    if (log.contains("Love at the Five and Dime"))
      System.out.println("Test 34 passed");
    else
      System.out.println("Test 34 failed");
  }
}
```

Professional Testing

- Test034 can run without user intervention and will report whether or not the test case has been passed.
- By developing an entire suite of such programs, software engineers can automate the testing process.
- The same set of test programs can be used over and over again, throughout the development and maintenance stages of the software process.
- Frameworks exist that simplify the creation, management and use of such batch test suites.

2.5 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** 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 (see next two slides)

LLStringNode Class

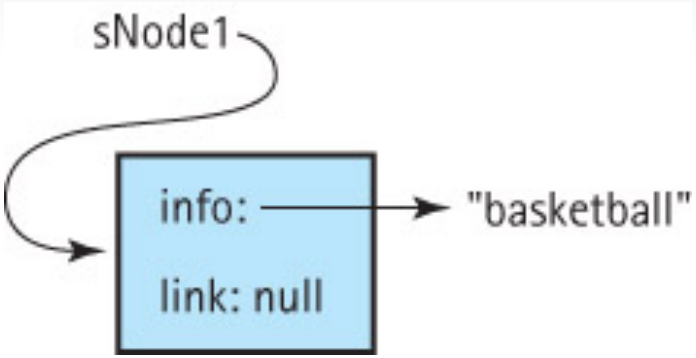
```
package ch02.stringLogs;

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;
  }
```

LLStringNode class continued

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

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

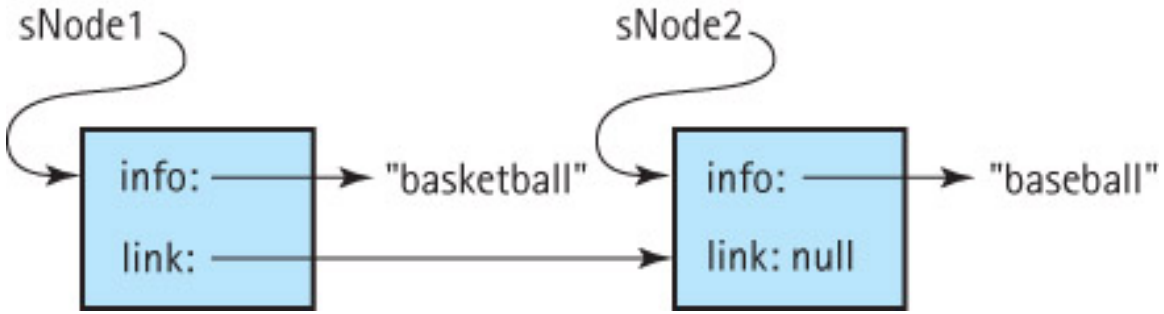
Using the LLStringNode class

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

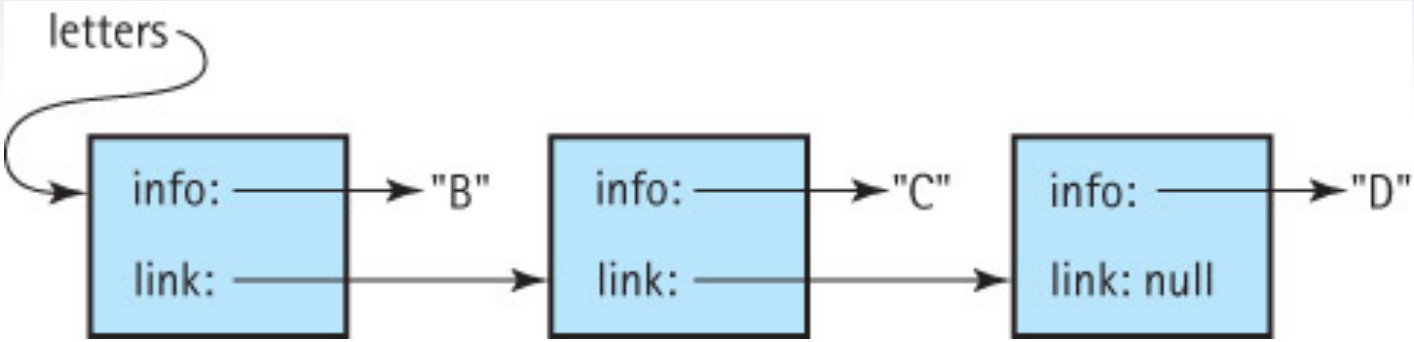


2: suppose that in addition to sNode1 we have SNode2 with info “baseball” and perform

`sNode1.setLink(sNode2);`



Traversal of a Linked List



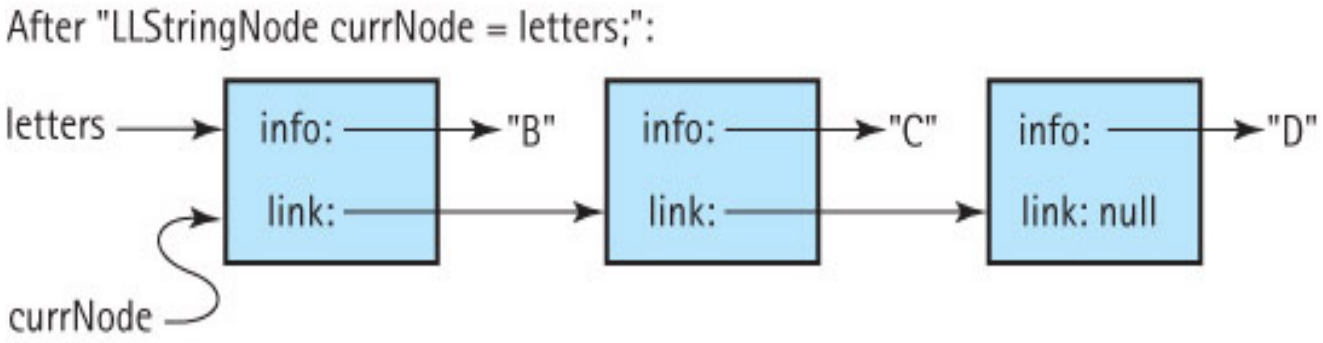

```
LLStringNode currNode = letters;
while (currNode != null)
{
  System.out.println(currNode.getInfo());
  currNode = currNode.getLink();
}
```

Tracing a Traversal (part 1)

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

Internal View

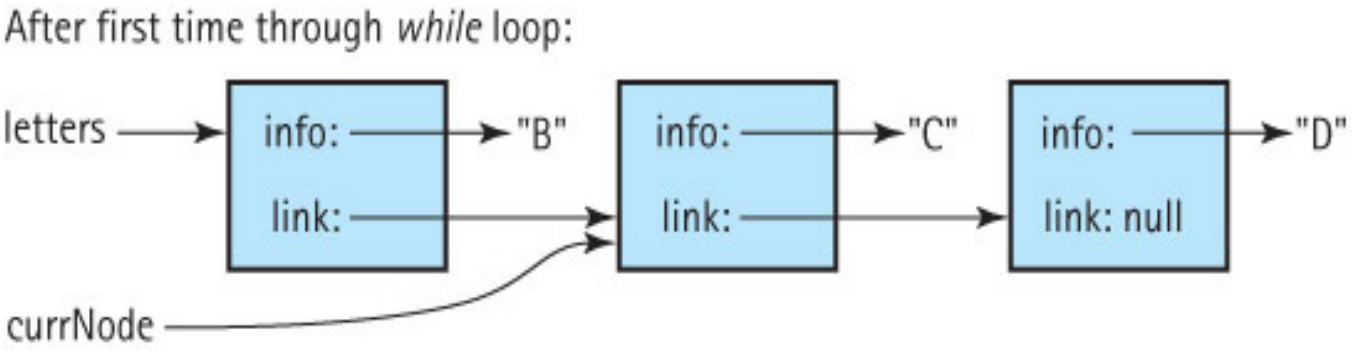
Output



Tracing a Traversal (part 2)

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

Internal View



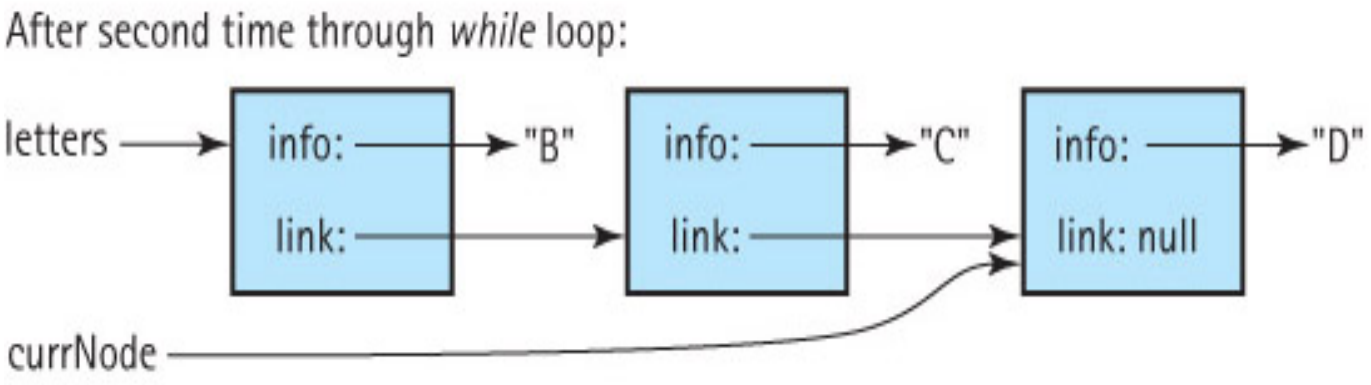
Output

B

Tracing a Traversal (part 3)

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

Internal View



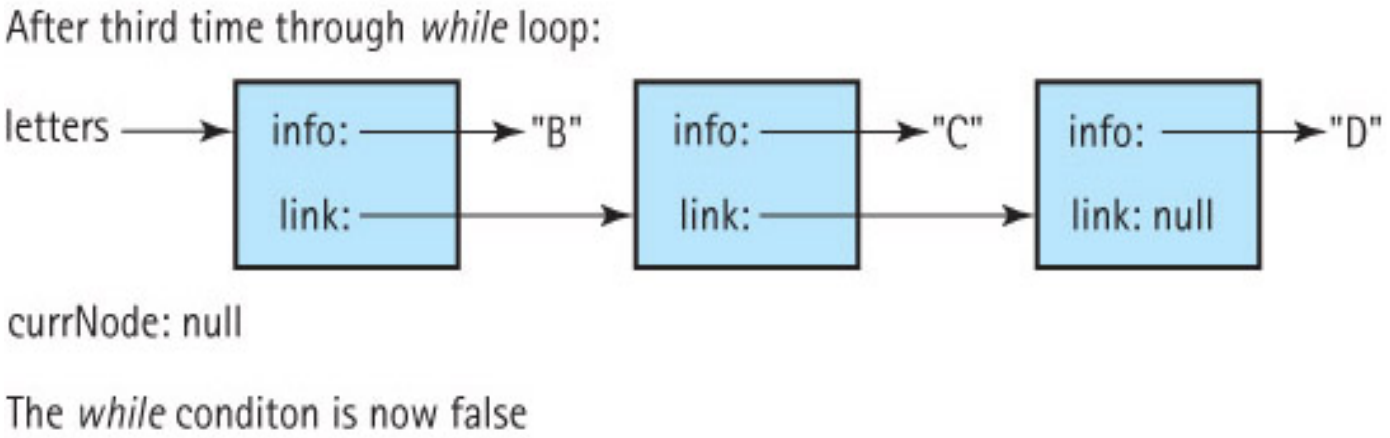
Output

B
C

Tracing a Traversal (part 4)

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

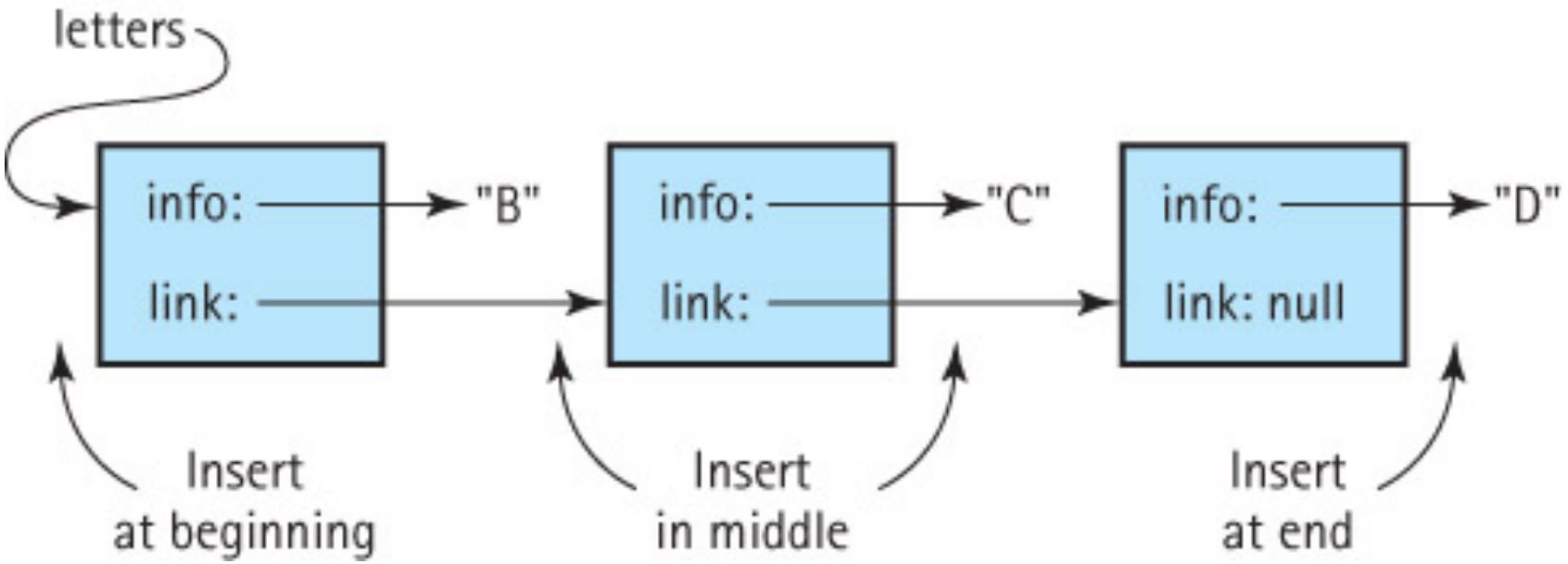
Internal View



Output

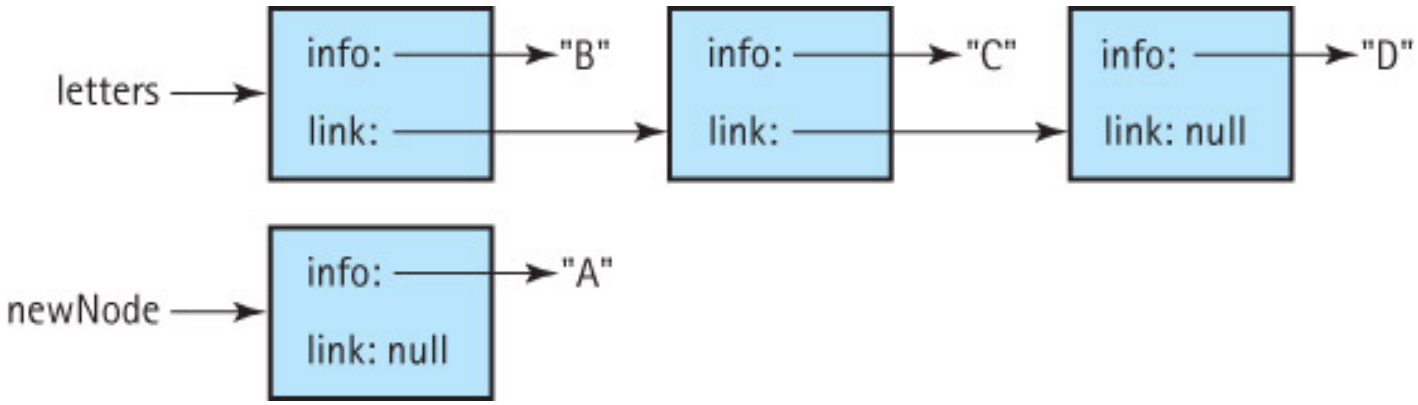
B
C
D

Three general cases of insertion



Insertion at the front (part 1)

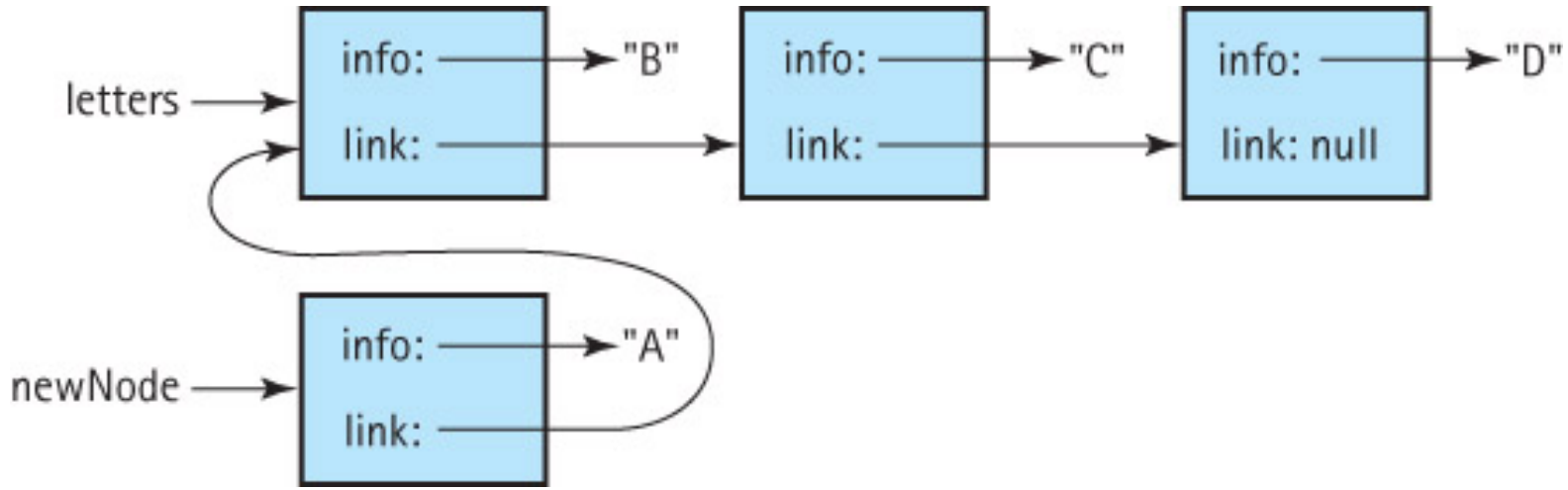
Suppose we have the node newNode to insert into the beginning of the letters linked list:



Insertion at the front (part 2)

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

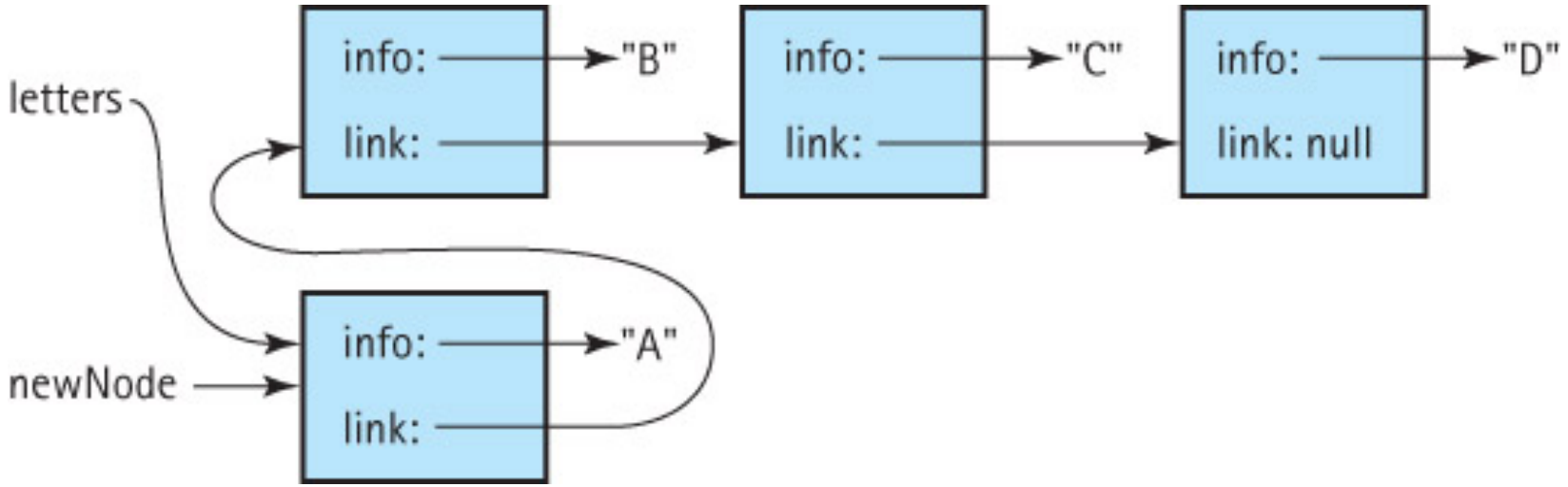
```
newNode.setLink(letters);
```



Insertion at the front (part 3)

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

```
letters = newNode;
```



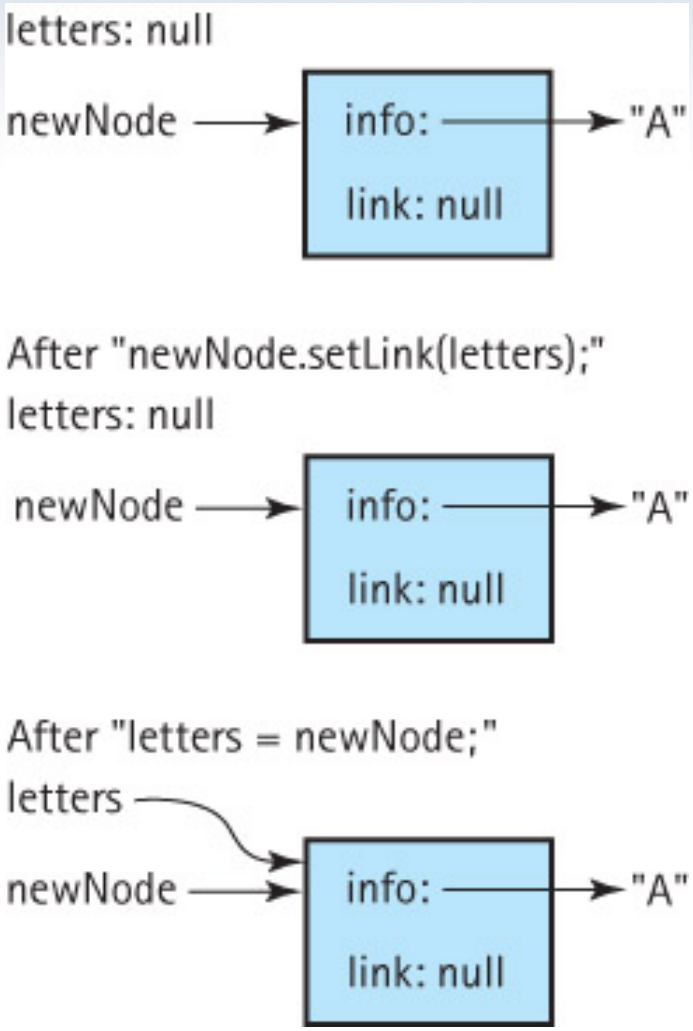
Insertion at front of an empty list

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?

As can be seen at the right the code still works, with the new node becoming the first and only node on the linked list.



2.6 Linked List StringLog ADT Implementation

- We call our new StringLog class the LinkedStringLog class, to differentiate it from the array-based class of Section 2.3.
- We also refer to this approach as a reference-based approach.
- Like the ArrayStringLog class, our LinkedStringLog class is part of the ch02.stringLogs package.
- The class fulfills the StringLog specification and implements the StringLogInterface interface.
- Unlike the ArrayStringLog, the LinkedStringLog will implement an unbounded StringLog.

Instance Variables

- LLStringNode log;
  - In this implementation, the elements of a StringLog are stored in a linked list of LLStringNode objects. We call the instance variable that we use to access the strings `log`. It will reference the first node on the linked list, so it is a reference to an object of the class LLStringNode.

- String name;
  - Recall that every StringLog must have a *name.* We call the needed variable `name`.

Instance variables and constructor

```
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

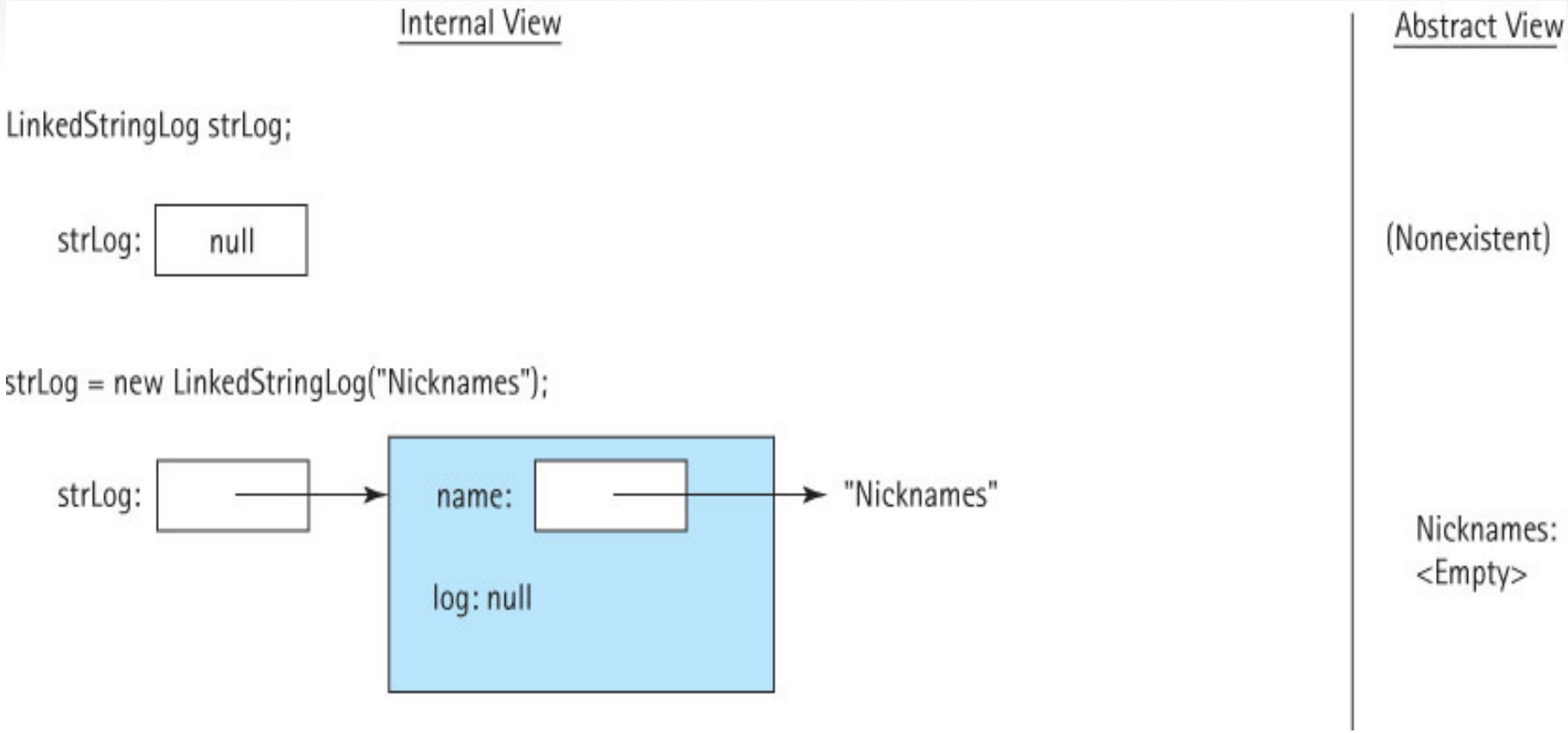
Insert the new string in the front:

```
public void insert(String element)
// Precondition:   This StringLog is not full.
//
// Places element into this StringLog.
{
  LLStringNode newNode = new LLStringNode(element);
  newNode.setLink(log);
  log = newNode;
}
```

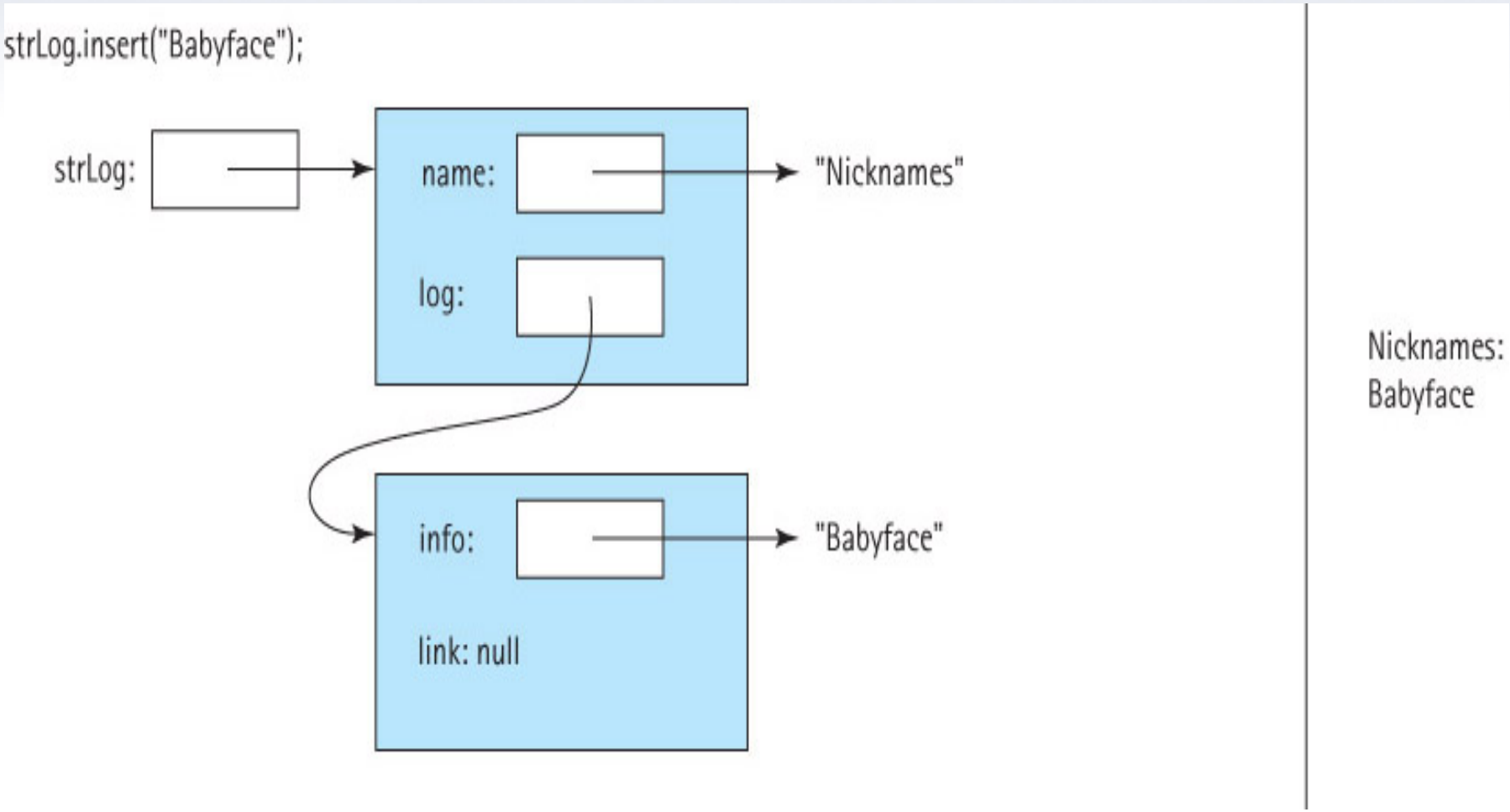
An example use:

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

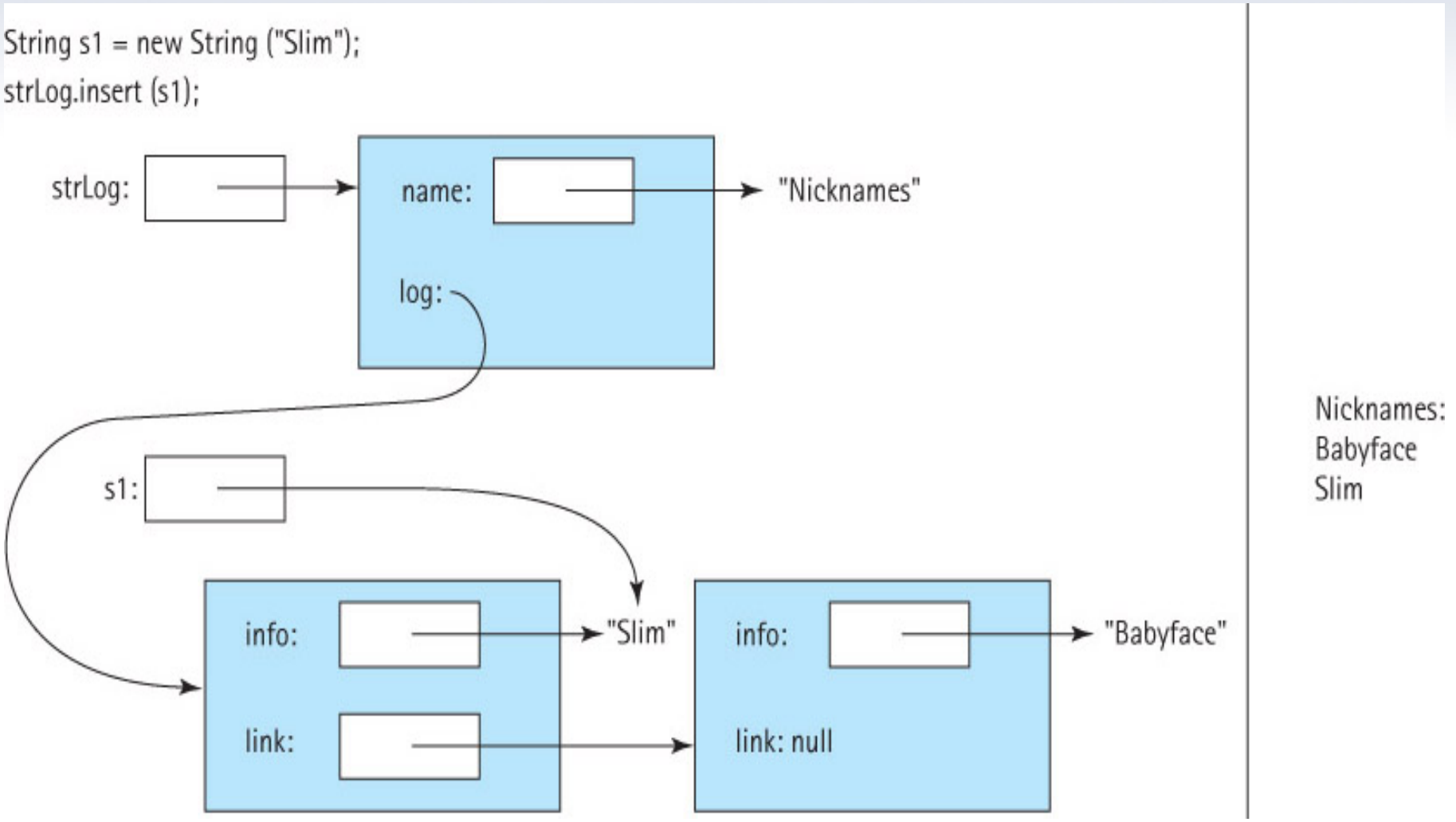
Example use of insert (part 1)



Example use of insert (part 2)



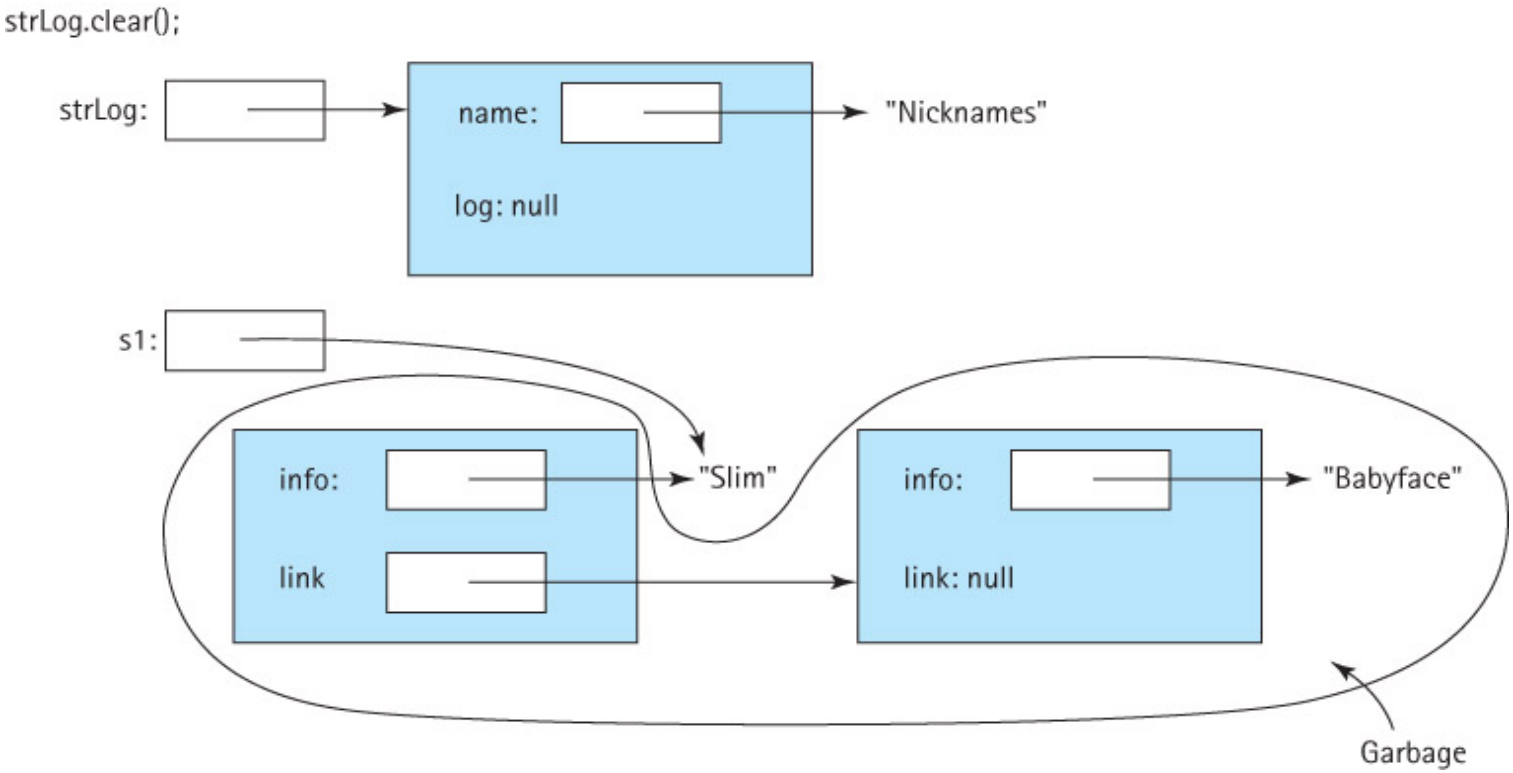
Example use of insert (part 3)



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;
}

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;
}
```

Note that size, toString, and contains (next slide) all use a form of a linked list traversal.

The contains method

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 (element.equalsIgnoreCase(node.getInfo()))  // if they match
      return true;
    else
      node = node.getLink();
  }
  return false;
}
```

2.7 Software Design: Identification of Classes

Repeat

Brainstorm ideas, perhaps using the nouns in the problem statement to help identify potential object classes.

Filter the classes into a set that appears to help solve the problem.

Consider problem scenarios where the classes carry out the activities of the scenario.

Until the set of classes provides an elegant design that successfully supports the collection of scenarios.

Sources for Classes