Chapter 2.5 – 2.8
Abstract Data Types
Linked Lists

Review Homework 3

- This is set up for your final project
- Brute force forward and reverse sort
  - Given: ForwardSort.java
  - Create: ReverseSort.java
- Examine and correct errors in
  - Sorts.java
  - SortsTest.java

The following code is useful for benchmarking your code

```
import java.util.Date;
long startTime = new Date().getTime();
long endTime = new Date().getTime();
```

Group Lab Exercise

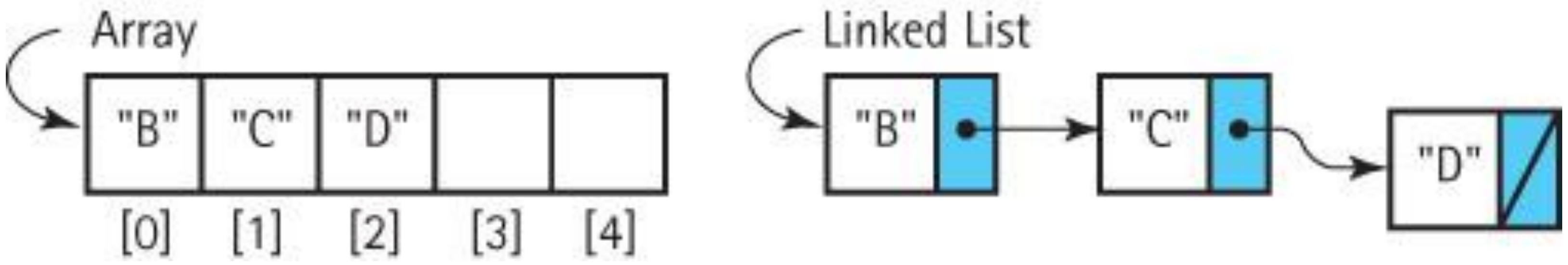
- Load SortTest.java
  - We will run several speed tests
  - This is a class project - divide and conquer
    - Testing the Big-O with speed testing
    - Perform three runs with each of the following parameters
      - SIZE = 10, 100, 1000, 10000, 100000
    - Row1: use RAND_RANGE = 100;
    - Row2: use RAND_RANGE = 1000;
    - Row3: use RAND_RANGE = 10000;
    - Row4: use RAND_RANGE = 100000;
  - Plot the results

Lab

- Implement a similar sorting sorting algorithm with the UseStringLog. Benchmark it.
  - Import Lab-StringLog
  - Follow instructions in the comments

Chapter 2: Abstract Data Types

2.5 Introduction to Linked Lists



- Arrays and Linked Lists are different in
  - use of memory
    - Array: fixed
    - Linked List: variable
  - manner of access
    - Array: by index
    - Linked List: sequential
  - Operational
    - E.g. inserting data

To create a linked list we need to:

1. dynamically allocate space for a node
2. allow a node to link to or reference another node

Nodes of a Linked-List

- A **node** in a linked list
  - is an object that holds information
    - for example “a string”
    - plus at least one link to an object of the same type
- **Self-referential class**
  - A class that includes an instance variable or variables that can hold a reference to an object of the same class
    - to support a linked implementation of the StringLog
      - we create the self-referential LLStringNode class (see next two slides)

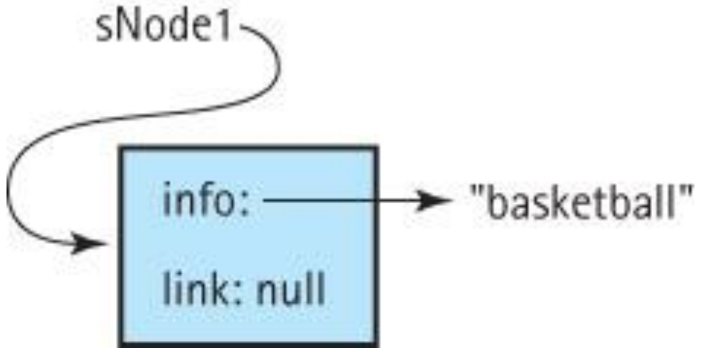
LLStringNode Class

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

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

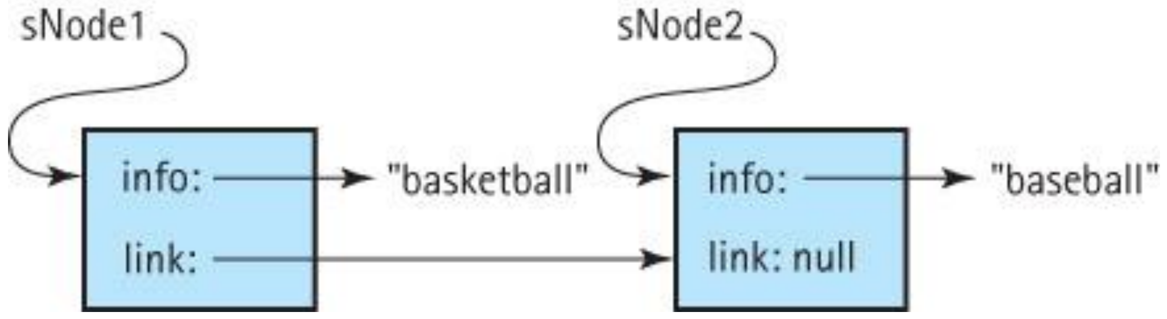
Using the LLStringNode class

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

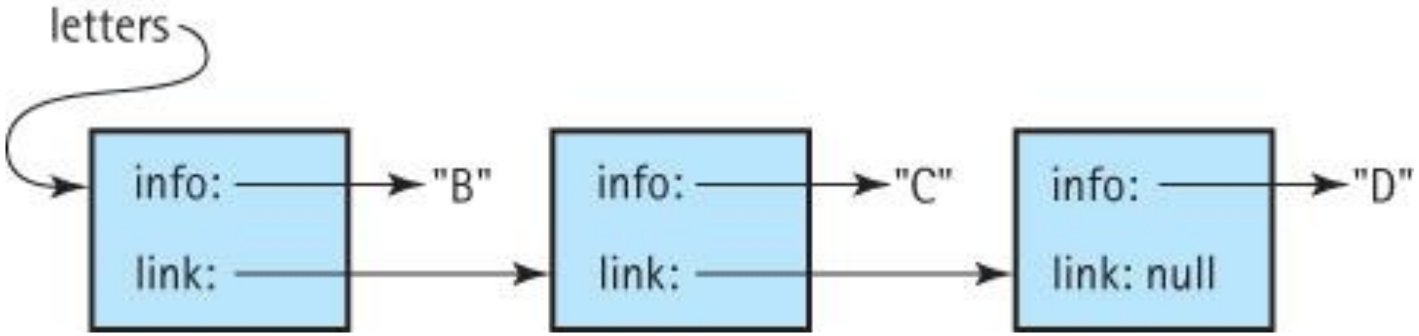


2: suppose that in addition to sNode1 we have SNode2 with info “baseball” and perform

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



Traversal of a Linked List



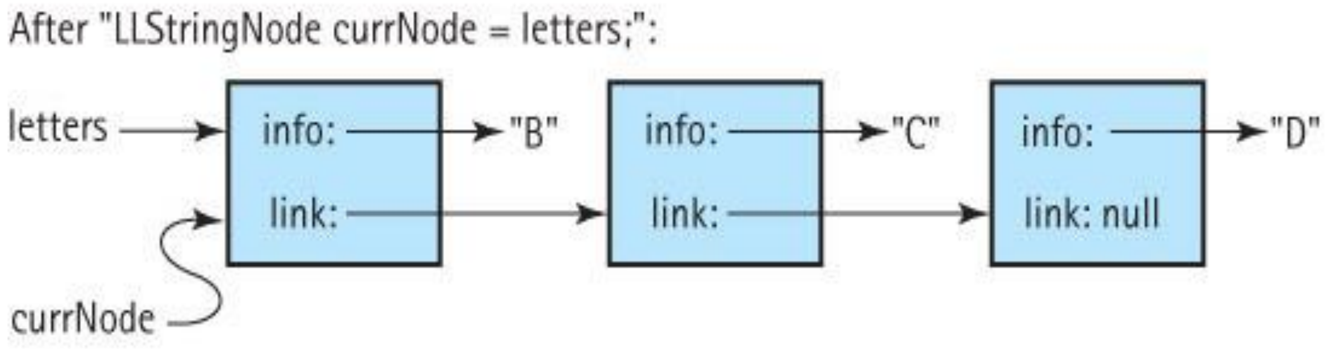
```
LLStringNode currNode = letters;
while (currNode != null)
{
  System.out.println(currNode.getInfo());
  currNode = currNode.getLink();
}
```

Tracing a Traversal (part 1)

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

Internal View

Output



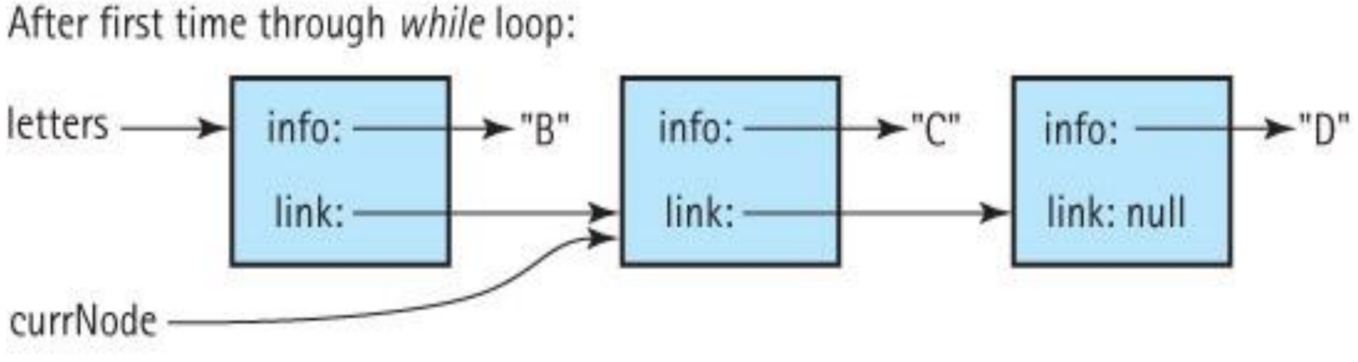
Tracing a Traversal (part 2)

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

Internal View

Output

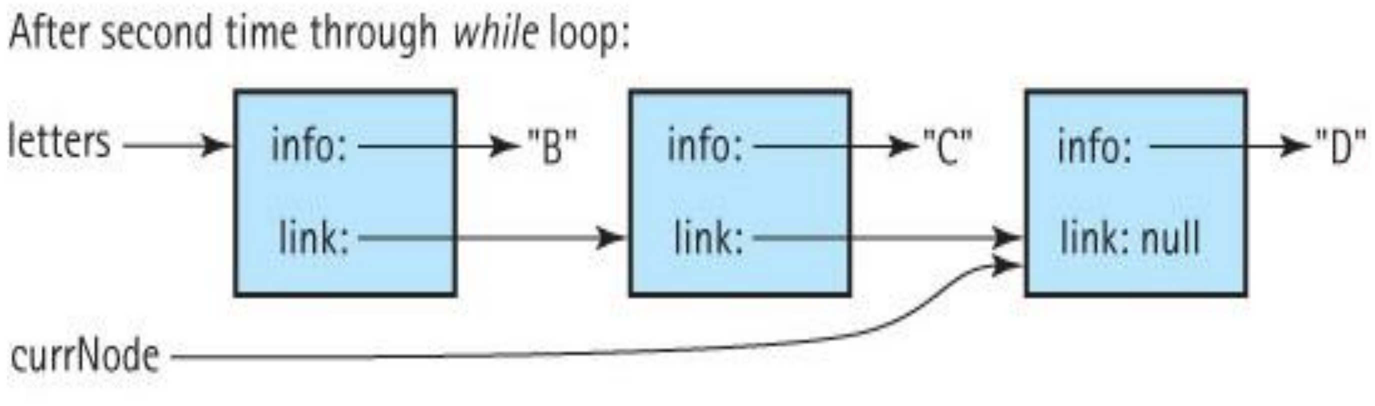
B



Tracing a Traversal (part 3)

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

Internal View



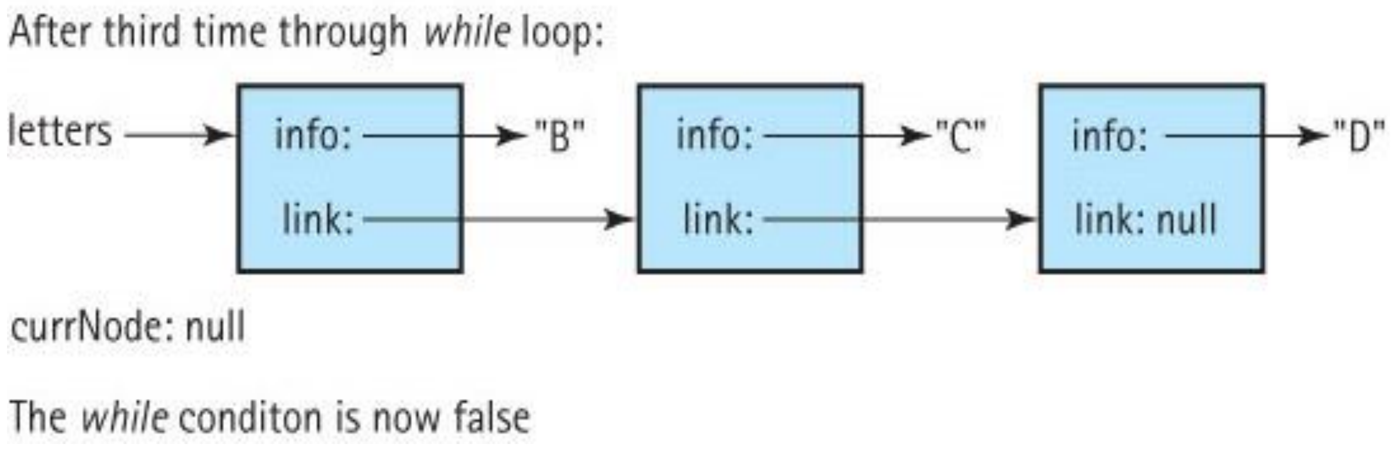
Output

B
C

Tracing a Traversal (part 4)

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

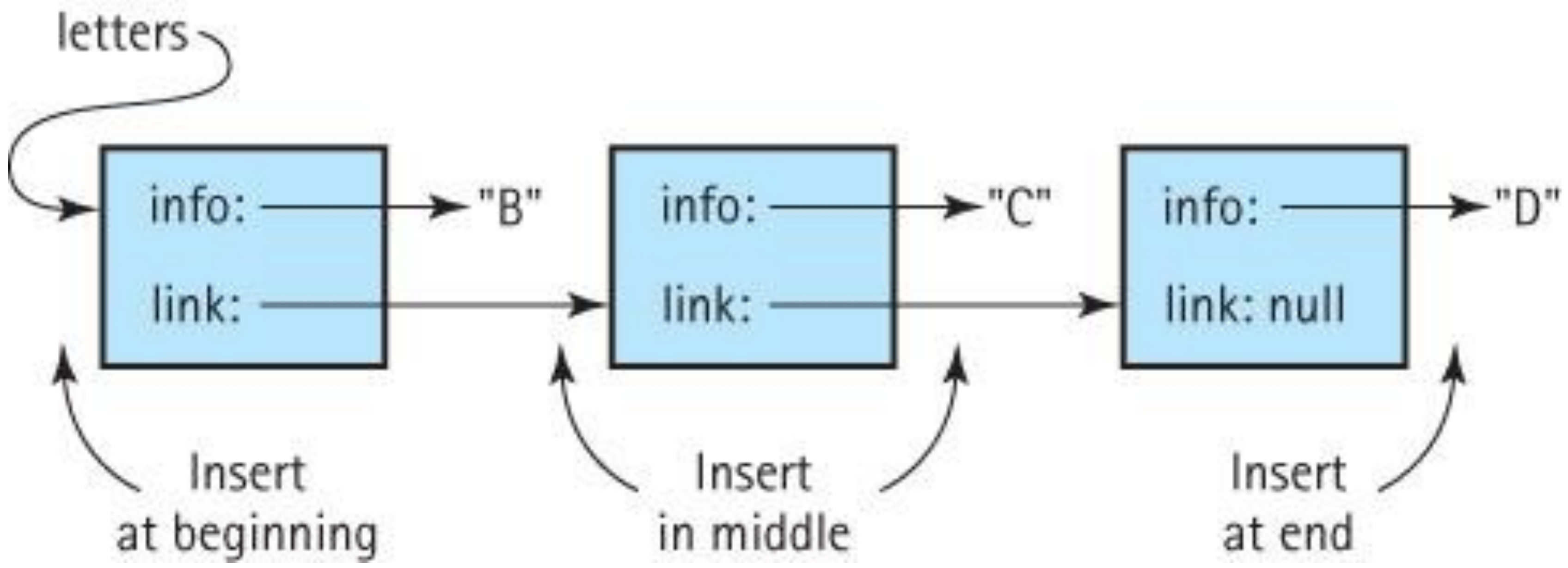
Internal View


After third time through *while* loop:
letters
info:
"B"
link:
info:
"C"
link:
info:
"D"
link: null
currNode: null
The *while* conditon is now false

Output

B
C
D

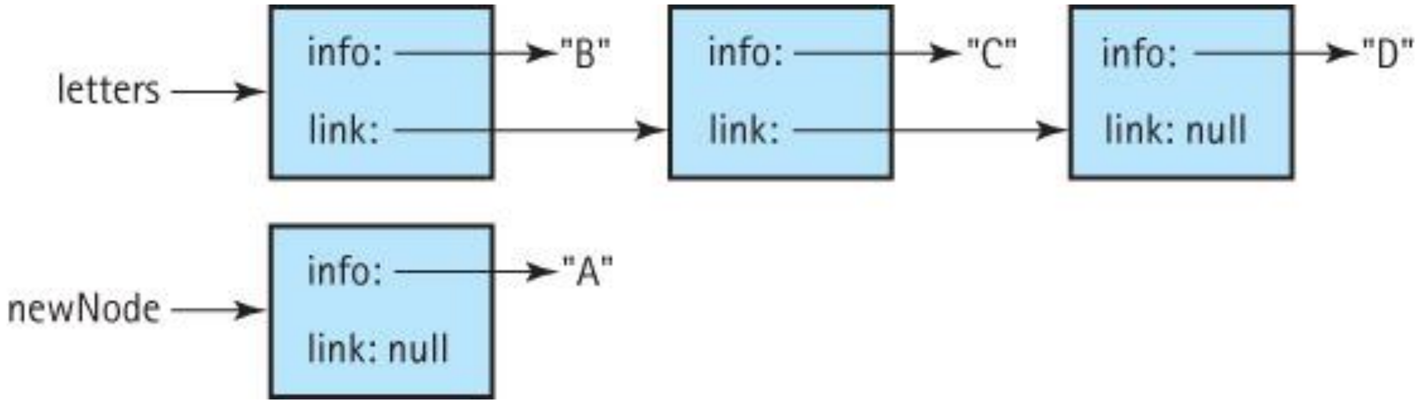
Three general cases of insertion



See page 113 of the text
StringLog vs. Stack vs. Queue vs. List

Insertion at the front (part 1)

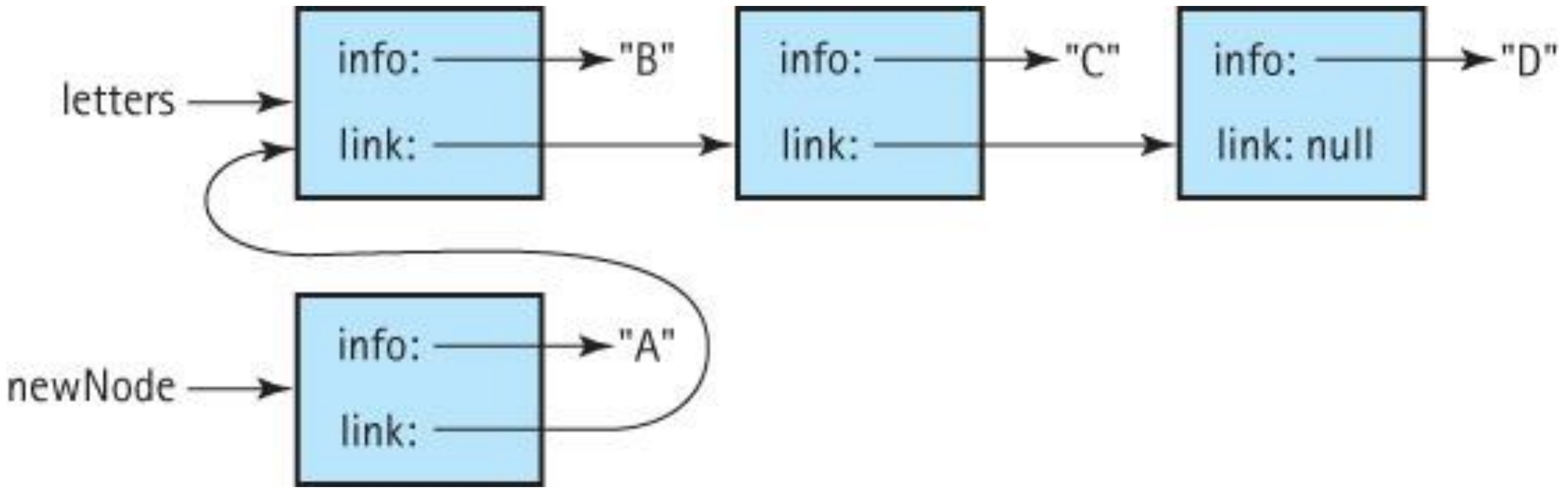
Suppose we have the node newNode to insert into the beginning of the letters linked list:



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 :

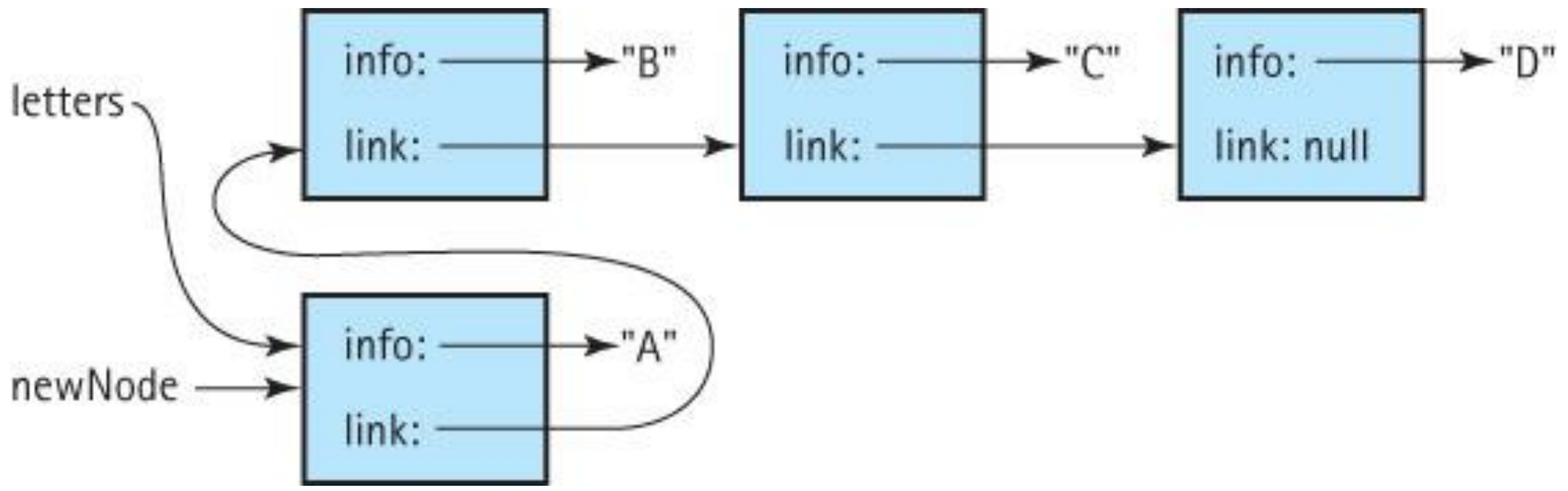
```
newNode.setLink(letters);
```



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;
```



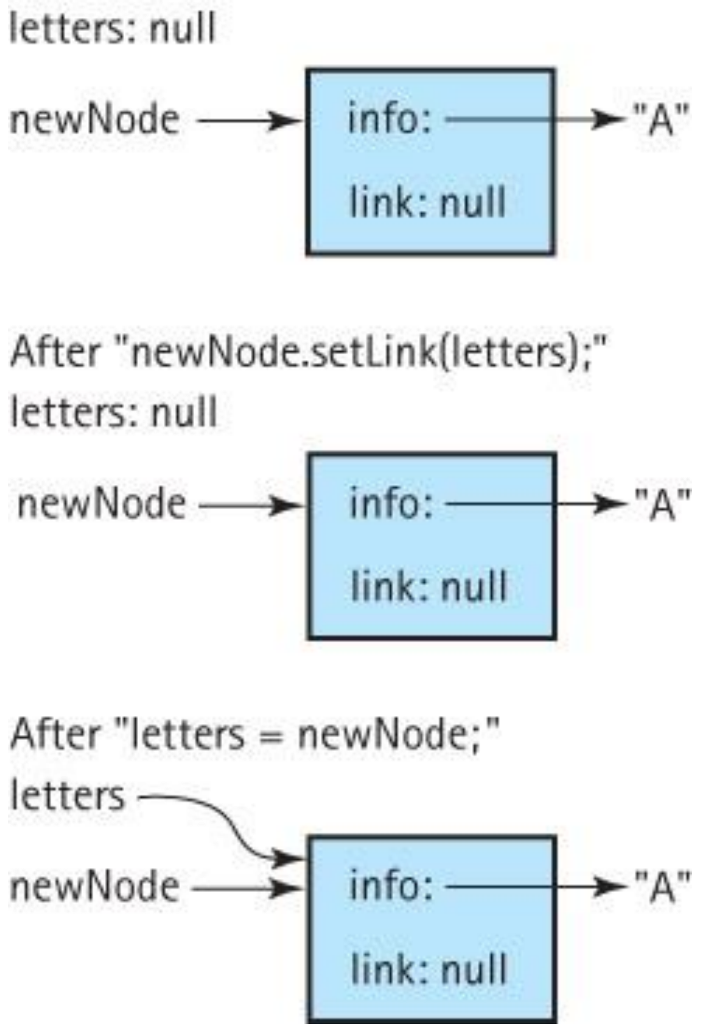
Insertion at front of an empty list

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

- LinkedStringLog class
  - fulfills the StringLog specification
  - implements the StringLogInterface interface.

- Approach is reference-based

- LinkedStringLog class is part of the ch02.stringLogs package
  - Similar to the ArrayStringLog class (… reusability???)
  - Different from the ArrayStringLog
    - the LinkedStringLog will implement an unbounded StringLog

Instance Variables

- LLStringNode log;
  - The elements of a StringLog are stored in a linked list of LLStringNode objects.
  - 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.

Transformer: insert operation

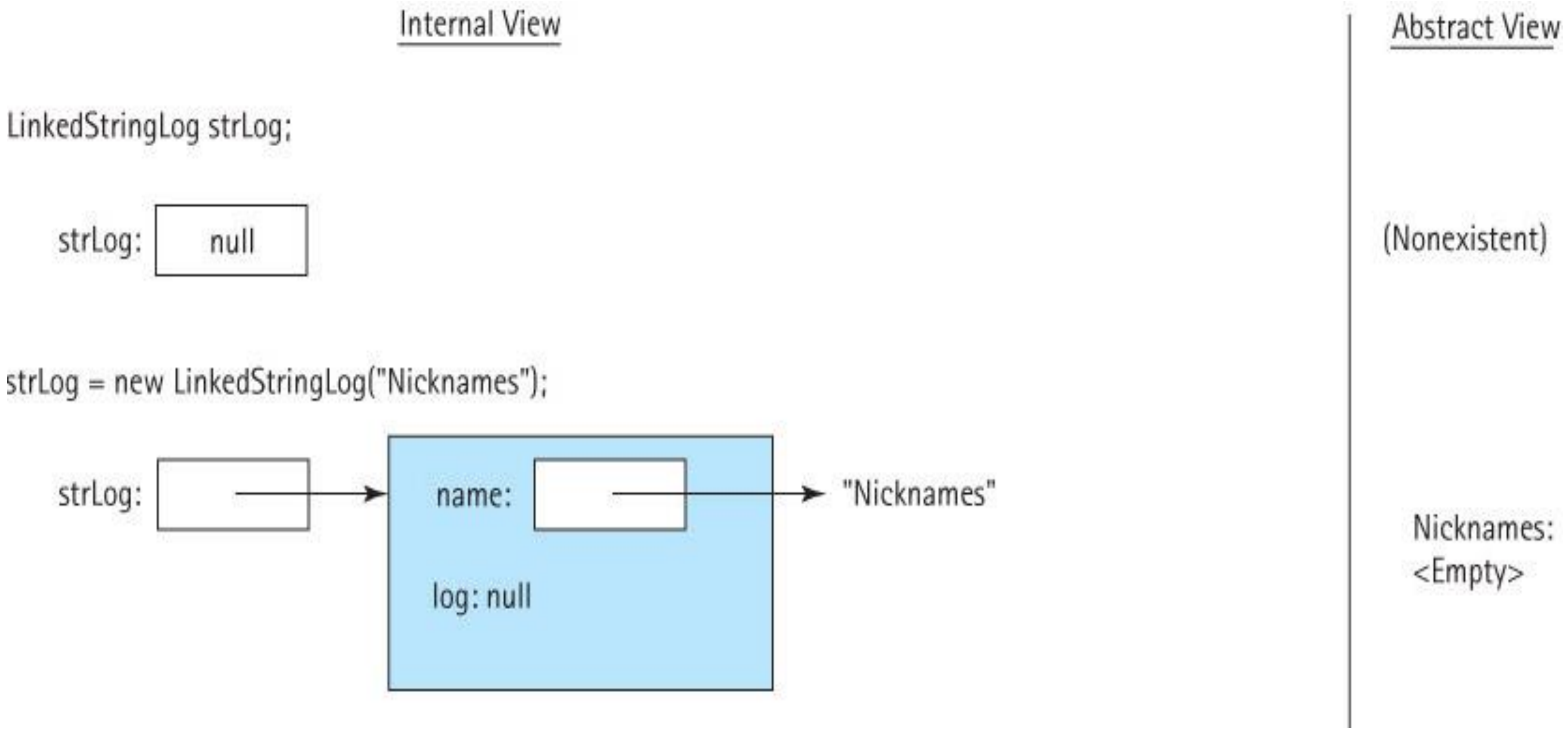
Insert the new string in the front:

```
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:

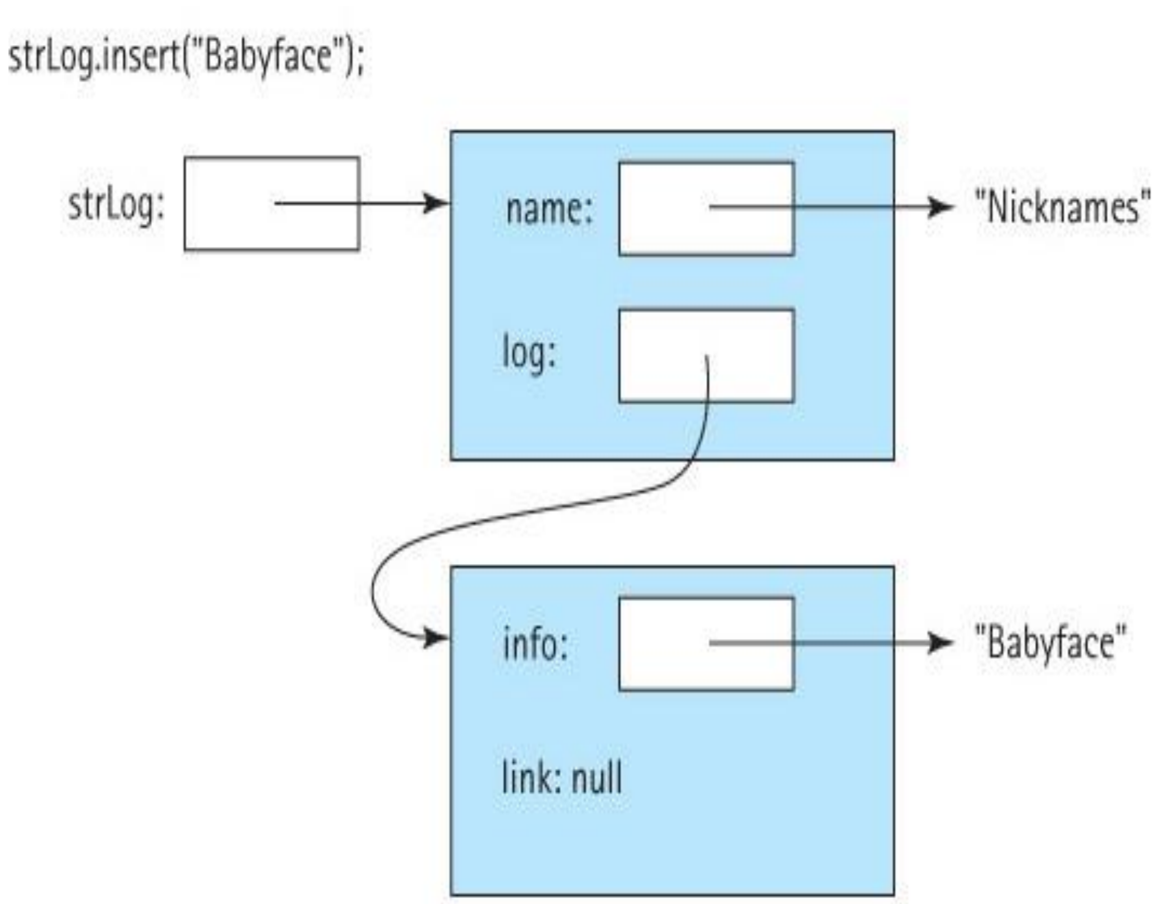
```
LinkedStringLog strLog;
strLog = new ArrayStringLog("aliases");
strLog.insert("Babyface");
String s1 = new String("Slim");
strLog.insert(s1);
```

Example use of insert (part 1)



Example use of insert (part 2)

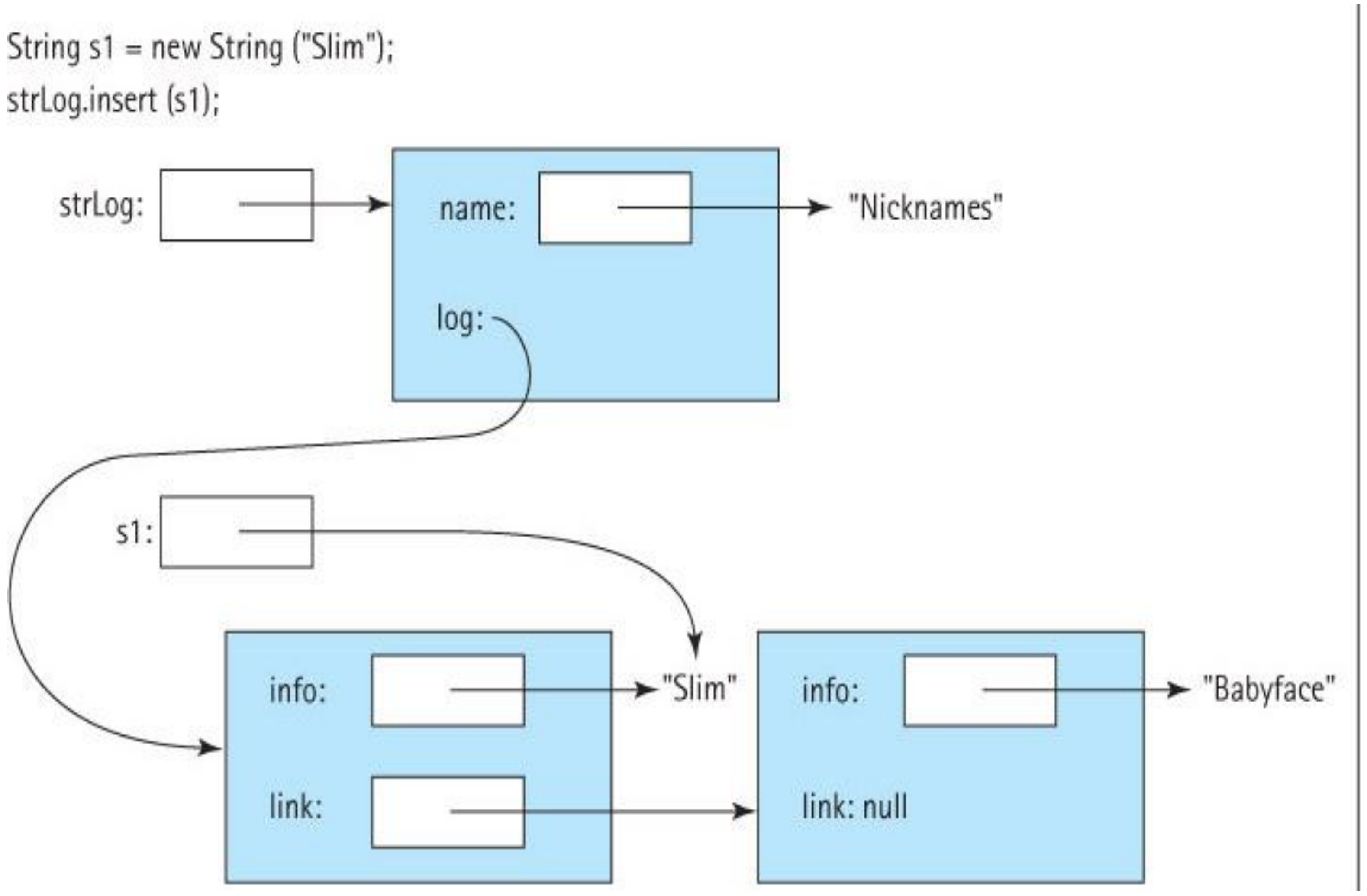
```
strLog.insert("Babyface");
```



```
Nicknames:
Babyface
```

Example use of insert (part 3)

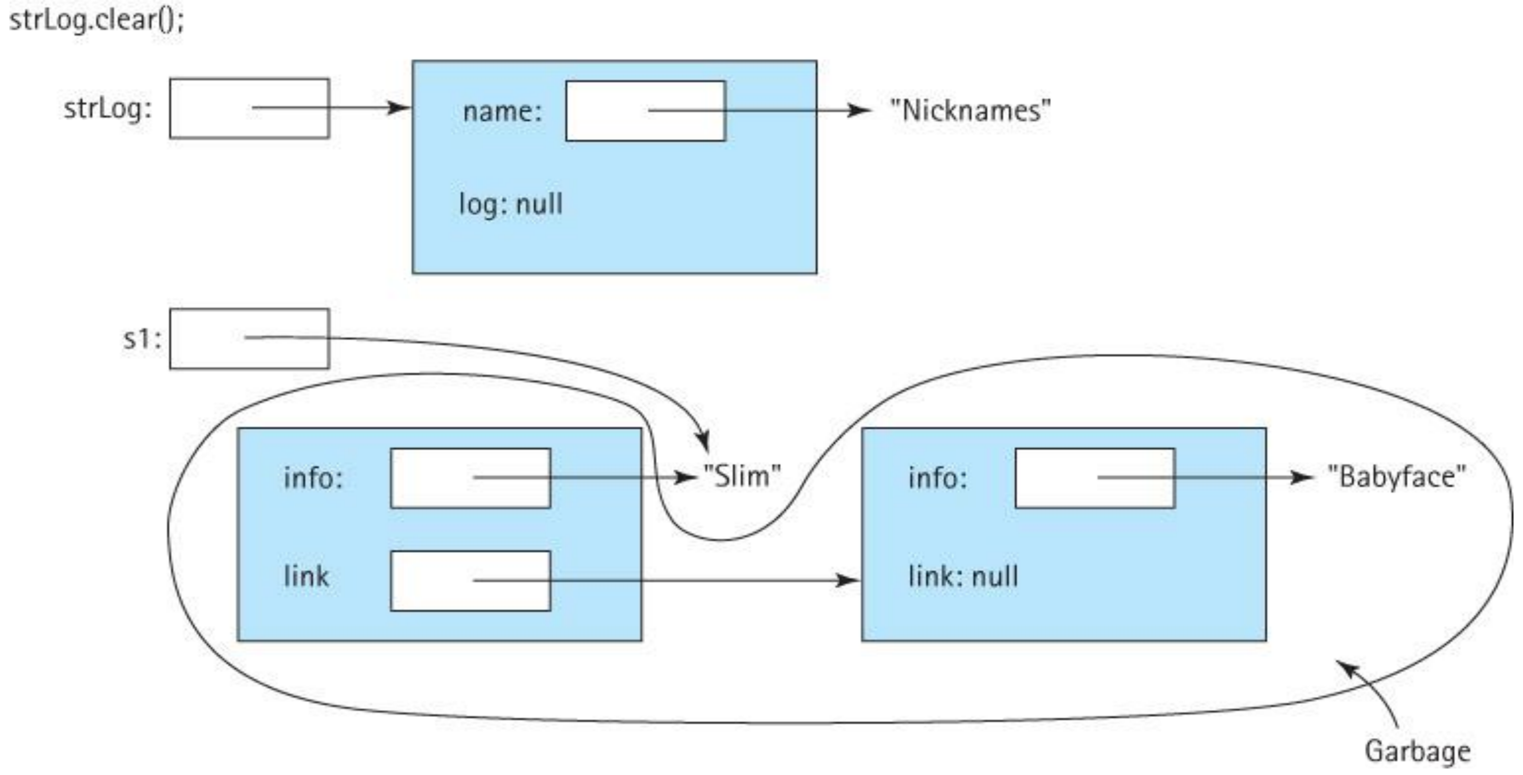
```
String s1 = new String ("Slim");
strLog.insert (s1);
```



Nicknames:
Babyface
Slim

Transformer: clear operation

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



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;
}
```

Observer: toString

```
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

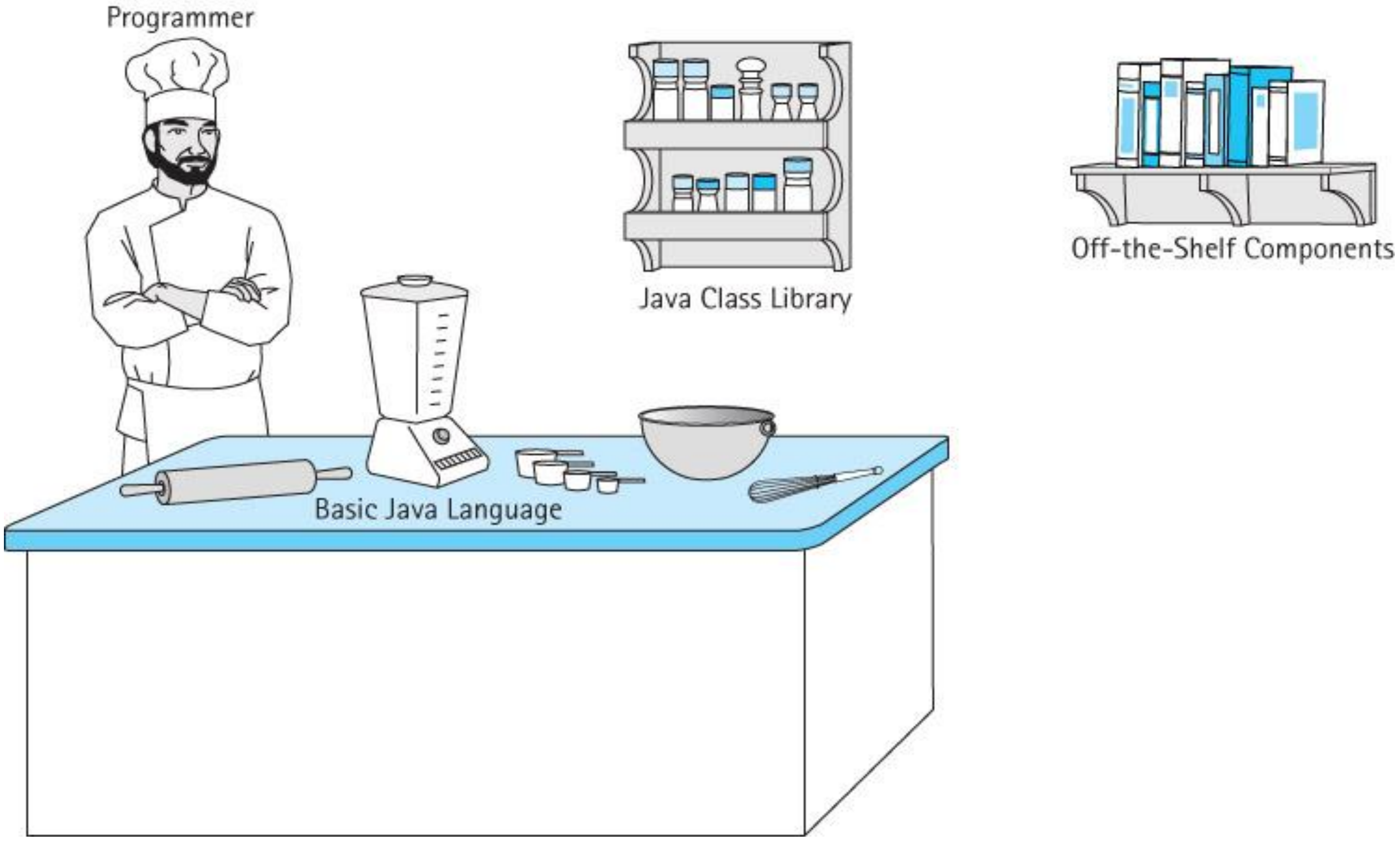
Brainstorm ideas, perhaps using the nouns in the problem statement to help identify potential object classes.

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



Lab 3 - Trivia

- Open the Trivia workspace
  - Information is in a text file game.txt
    - Examine the game.txt
    - This approach gives some freedom
  - Versatile: can support alternative data sources

Summary CH2: ADT

- Examined StringLog ADT
  - 2 forms
    - Arrays
    - References: linked lists
      - Create and manipulate
  - Data abstraction to work with data structures
    - Log of strings
  - Take home message in Ch2 is reusability
    - Single structure and its accessing operations can be used by other programs for completely different applications as long as the correct interfaces are maintained

Homework – Part 1

1. Implement the sorting test for the linked list implementation of *UseStringLog*

   – You will need to implement some of the methods for *LinkedStringLog.java* that you did in *ArrayStringLog.java* in exercises 21-26

Homework – Part 2

2. Arrays vs. Linked Lists: performance comparison
   - Perform three runs for each parameter pair for both sets of code (array based and linked list based)
     - String log size: 100
       - Random number size: 100, 1000, 10000, 100000
     - String log size: 1000
       - Random number size: 100, 1000, 10000, 100000
     - String log size: 10000
       - Random number size: 100, 1000, 10000, 100000
     - Report the data for each run
     - Compare the results for the average of each three runs using the same parameters
       - 12 averages for the linked list implementation
       - 12 averages for the array implementation
     - Analyze the data… Does this meet Big-O expectations? Why?