Chapter 2
Abstract Data Types

Homework
Abstract Data Types

display some answers

2. Which of these statements is always true?

- All of the program requirements must be completely defined before design begins.
- All of the program design must be complete before any coding begins.
- All of the coding must be complete before any testing can begin.
- **Different development activities often take place concurrently, overlapping in the software life cycle.**

11. Research question: List and briefly describe the UML's 12 main diagramming types.

http://en.wikipedia.org/wiki/Unified_Modeling_Language

17. Modify the *Date* class so that

- Its *toString* method returns a string of the form “month number, number” – for example, “May 13, 1919.”
- It provides a public class *mjd* (standing for Modified Julian Day) that returns the number of days since November 17, 1858.
- It provides a public class *djd* (standing for Dublin Julian Day) that returns the number of days since January 1, 1900.

See code

20. Use the *DaysBetween* application to answer the following using Feb 6, 2012:

- On Feb 6, 2012, how old are you in days? **(it depends on your B-day)**
- On Feb 6, 2012, how many days has it been since the United States adopted the Declaration of Independence on July 4, 1776? **(86048)**
- How many days are between November 21, 1783 and July 20, 1969? **(67811)**

28. Given the definitions of the *Date* and *IncDate* classes in this chapter and the following declarations

int temp;

Date date1 = new Date(10,2,1989);

Date date2 = new Date(4.2.1992);

IncDate date3 = new IncDate(12,25,2001);

- Indicate which of the following statements are illegal and which are legal. Explain your answers.
- temp = date1.getDay();		Legal - *getDay* is a public method that returns an *int*
- temp = date1.getYear();		Legal- *getYear* is a public method that returns an *int*
- date1.increment();		Illegal - *increment* is not defined for *Date* objects
- date3.increment();		Legal - *increment* is defined for IncDate objects
- date2=date1;		Legal - object variables can be assigned to objects of the same class
- date2=date3;		Legal - subclasses are assignment compatible with the superclasses above them in the class hierarchy
- date3=date2;		Illegal - superclasses are not assignment compatible with the subclasses below them in the class hierarchy

39. What is an alias? Show an example of how it is created by a Java program. Explain the dangers of aliases.

- See page 35

46. Describe the order of magnitude of each of the following functions using Big-O notation?

• $N^2 + 3N$	**$O(N^2)$**
• $3N^2 + N$	**O(N2)**
• $N^5 + 100N^3 + 245$	**O(N5)**
• $3N\log_2 N + N^2$	**O(N2)**
• $1 + N + N^2 + N^3 + N^4$	**O(N4)**
• $(N*(N-1))/2$	**O(N2)**

48. Describe the order of magnitude of each of the following code sections, using the big-O notation.

```
count = 0;                                  // O(N)
for (i = 1; i <= N; i++)
    count ++;
 ------
count = 0;                                  // O(N^2)
for (i = 1; i <= N; i++)
    for (j = 1; j <= N; j++)
          count ++;
 ------
value = N;                                  // O(Log_2 N)
count = 0;
while (value > 1)
{
    value = value / 2;
    count++;
}
 ------
count = 0;                                  // O(1)
value = N;
value = N * (N - 1);
count = count + value;
```

```
count = 0;                                  // O(N)
for (i = 1; i <= N; i++)
    count ++;
for (i = N; i >= 0; i++)
    count ++;
```

51. Algorithm 1 does a particular task in a "time" of N^3, where N is the number of elements processed. Algorithm 2 does the same task in a "time" of 3N + 1000.

- What is the Big-O efficiency of each algorithm? **Algorithm 1 is $O(N^3)$; Algorithm 2 is O(N).**
- Which algorithm is more efficient by Big-O standards? **Algorithm 2**
- Under what conditions, if any, would the "less efficient" algorithm execute more quickly than the "more efficient" algorithm?

  Solve: $N^3 < 3N + 1000$

  or

N	**N^3**	**3N + 1000**
1	**1**	**1003**
2	**8**	**1006**
5	**125**	**1015**
10	**1000**	**1030**
11	**1331**	**1033**

Therefore for N <= 10 the "less efficient" algorithm is faster.

Debugging

- Download Date.java and DaysBetween.java from myAVC
- I have created a few errors. It is your job to debug and test the code.

Quiz

- What are the order of magnitude of the following functions, using Big-O notation

1.	$x^5/5 + x^4/4 + x^3/3 + x^2/2 + x + 1{,}000{,}000$	**$O(N^5)$**
2.	$10N\log_2 N + 25N^2$	**$O(N^2)$**
3.	350	**$O(1)$**
4.	y^{789}	**$O(N^{789})$**
5.	π (as in AreaOfCircle = π * radius^2)	**$O(1)$**
6.	An algorithm with a nested set of loops (otherwise sequential)	**$O(N^2)$**
7.	The statement: System.out.println();	**$O(1)$**

8. Describe the order of magnitude of the following code section using Big(O) notation k = 0;

```
for (i = 0; i < N; i++)
 for (j = (2 * N); j > 0; j--)
   for (k = 3; k < N; k++)
     x = x + 1;
```

$O(N^3)$

9. Which algorithm is more efficient and why?
   a. The first has a Big-O of $O(N^3)$ and contains 10 lines of code
   b. The second has a Big-O of $O(1)$ and contains 4000 lines of code

Chapter 2: Abstract Data Types

2.1 Abstraction

1. **Abstraction**
   - A model of a system that includes only the details essential to the viewer's perspective that system
   - All about WYSWIG
2. **Information hiding**
   - Hiding details within a module in order to control access to those details from the rest of the system
   - Today… Who wants to do this with what?
3. **Data abstraction**
   - Separating a data type's logical properties from its implementation
   - Creating new views
4. **Data encapsulation**
   - Separate how data is represented from the applications that use the data at a logical level
   - "The physical representation of the data remains hidden"
5. **Abstract data type (ADT)**
   - Properties (domain and operations) are specified independently of any particular implementation

Coding sounds kinda' CIA! Truth be told, the underlying theme is robustness.

How many of you use ADTs?

- What data types do / have you used?

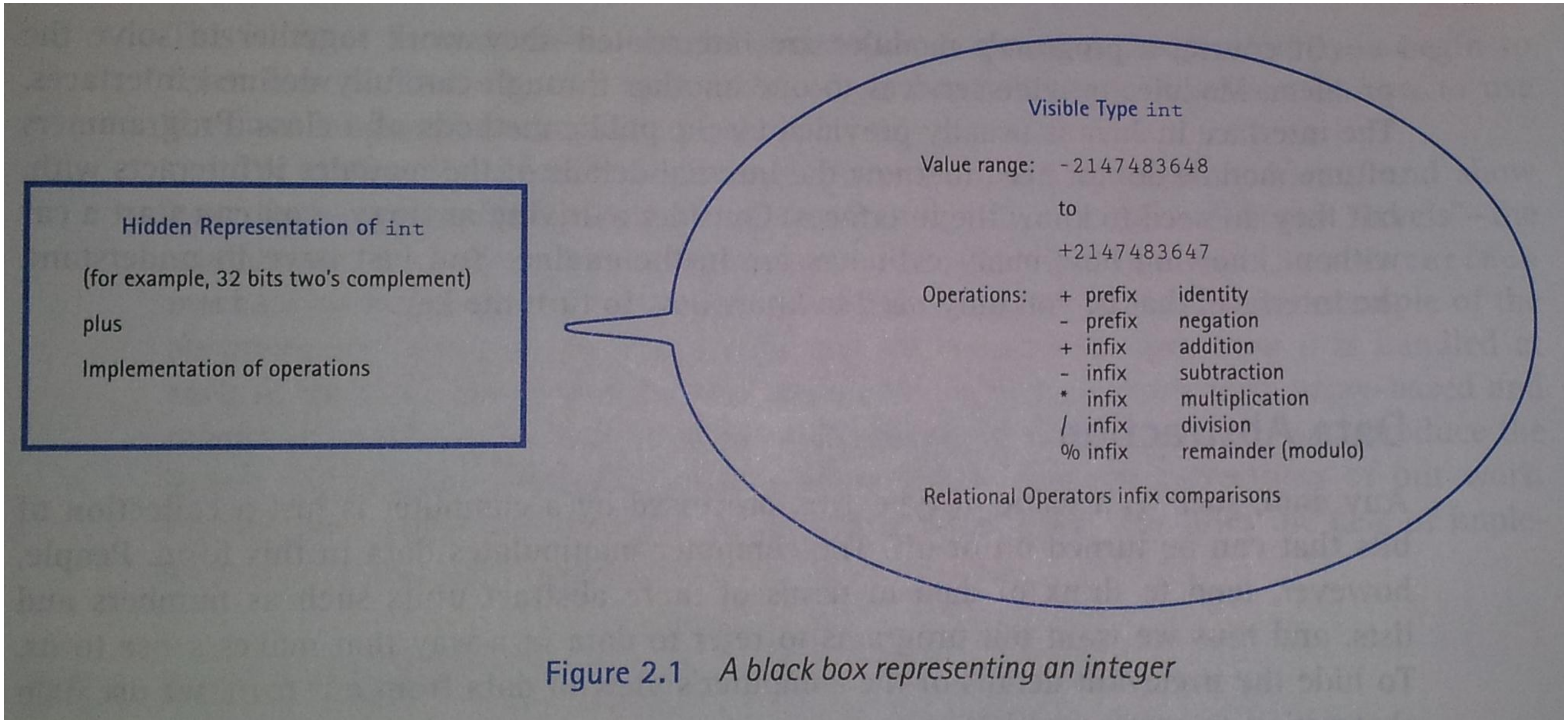

Hidden Representation of int
(for example, 32 bits two's complement)
plus
Implementation of operations
Visible Type int
Value range: -2147483648 to +2147483647
Operations:
+ prefix identity
- prefix negation
+ infix addition
- infix subtraction
* infix multiplication
/ infix division
% infix remainder (modulo)
Relational Operators infix comparisons

Figure 2.1 *A black box representing an integer*

Remember the Chapter 1 Date Class

- Having programmed it
  - we know about the underlying
- The Date class
  - Composed of three *int* instance variables
    - The programmer of Date needs to know the names
    - Other application programmers do NOT need to know
  - Application programmers need to know how to use the Date object, i.e.
    - How to create the Date object
    - How to invoke the Date objects methods

ADT Perspectives (3 Levels)

(Reordered from the book)

1. ***Logical (or abstract) level***:
   - Provides an abstract view of the data values (the domain) and the set of operations to manipulate them.
     - What is the ADT?
     - What does it model?
     - What are its responsibilities?
     - What is its interface?
2. ***Implementation (or concrete) level***:
   - Provide a specific representation of the structure to hold the data and the implementation (coding) of the operations.
     - How to represent and manipulate data in memory?
     - How to fulfill the responsibilities of the ADT?
3. ***Application (or user or client) level***:
   - Use the ADTs to solve a problems.
   - Create instances of an ADT
   - Invoke the ADT's operations

When might a programmer deal with each?

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
- **Remember to**…
  - Specify pre- and postconditions for a method in a comment at the beginning of the method
    - Comment on commenting… (page 68 makes suggestions)
    - Commenting should be uniform and …

Interfaces

- **Interface**
  - boundary shared by two interacting systems
- **User interface**
  - is the part of the program interacting with the user
- **Interface of an objects method**
  - is its set of parameters and the return value it provides
- **Java interface**
  - interface is a keyword
  - represents a specific program unit
  - looks similar to a Java class
    - all variables declared in an interface must be constants
    - the methods must be abstract (**abstract method**)

Java: Abstract Method

- Only includes a description of its parameters
- No method bodies
- No implementations

Only the *interface* of the method is included.

Java Interfaces

- **Similarties** to a Java class
  - can include variable declarations
  - can include methods
- **Differences** from a Java class
  - Variables must be constants
  - Methods must be abstract
  - A Java interface cannot be instantiated.
- Can be used to formally specify the logical level of an ADT
  - Provides a template for classes to fill
  - A separate class must "implements" it.
- Java interfaces are used to describe class requirements

Example Interface

template class

```
public interface FigureGeometry
{
  final float PI = 3.14f;                    // constant

  float perimeter();                         // abstract method
  // Returns perimeter of this figure.

  float area();                              // abstract method
  // Returns area of this figure.

  void setScale(int scale);                  // abstract method
  // Scale of this figure is set to "scale".

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

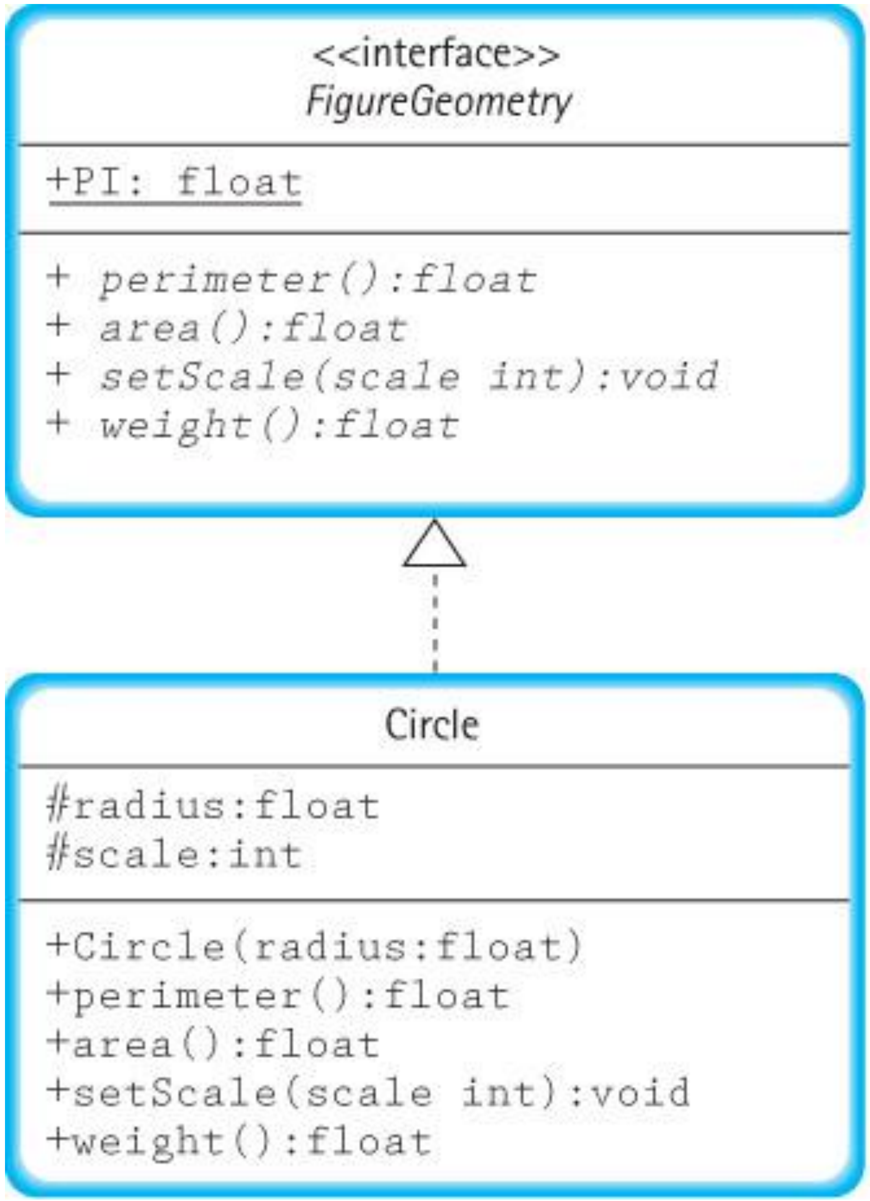
Kinda' looks like a .h file in c++…

Example of a class implementing the interface

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

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

  public float perimeter()
  // Returns perimeter
  {
    return(2 * PI * radius);
  }

  public float area()
  // Returns area
  {
    return(PI * radius * radius);
  }
```

```
public void setScale(int scale)
  // Set the "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);
  }
}
```

Interface: UML

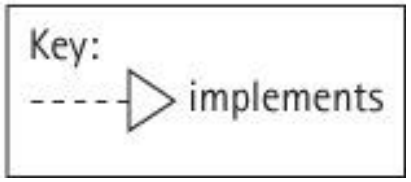


OOP Concepts

1. Interfaces can be used to specify the logical views of ADTs

2. A class implementing an interface…
   - can access interface constants
   - provide code for the body for all abstract methods declared in the interface

3. Many classes can implement the same interface
   - rectangle
   - parallelogram
   - ellipse



Benefits of interfaces

- Formal syntax checking
  - When the interface is compiled
  - Compiler identifies syntax errors in the method interface definitions.
- Formal verification that the interface “contract” is met by the implementation.
  - When the implementation is compiled
  - Compiler ensures that the method names, parameters, and return types match those defined in the interface.
- Consistency
  - Implementation1 of interface can optimize memory
  - Implementation2 of interface can optimize speed
  - Implementation3, 4, … can all broaden the scope & function of the interface
  - All implementations of the ADT share a common interface

2.2 The StringLog ADT Specification

- StringLog ADT
  - Remembers all the strings that have been inserted into it
  - When presented with any given string, it indicates whether or not an identical string has already been inserted.
- A StringLog client uses a StringLog to …
  - Record strings
  - Check to see if a particular string has been recorded
- Every StringLog must have a “name”.

StringLog Methods: Constructors

- Create a new instance of the ADT
- The implementer of the StringLog decides how many and what kind of constructors to provide.
  - Unbounded
  - Bounded
    - Default size
    - Constructor parameter may indicate the maximum size

StringLog Methods: 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
  - Note: the StringLog retains its name

StringLog Methods: Observers

- contains(String element)
  - returns true if element is in the StringLog, false otherwise
  - String comparison is not case sensitive, i.e. (“Jim” == “jim”) is True
- 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 formatted string that represents the entire contents of the StringLog.

The StringLogInterface

```
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.

  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.
}
```

Novel commenting is not advisable!

The StringLogInterface

```
package ch02.stringLogs;

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

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

  // .size: returns the number of Strings in this StringLog.
  int size();

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

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

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

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

1. **Tradition: comment before code!**
2. **Comments: KISS rule – clear & short**
3. **Leave out all adjectival opinions**

StringLogInterface Example

```
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:

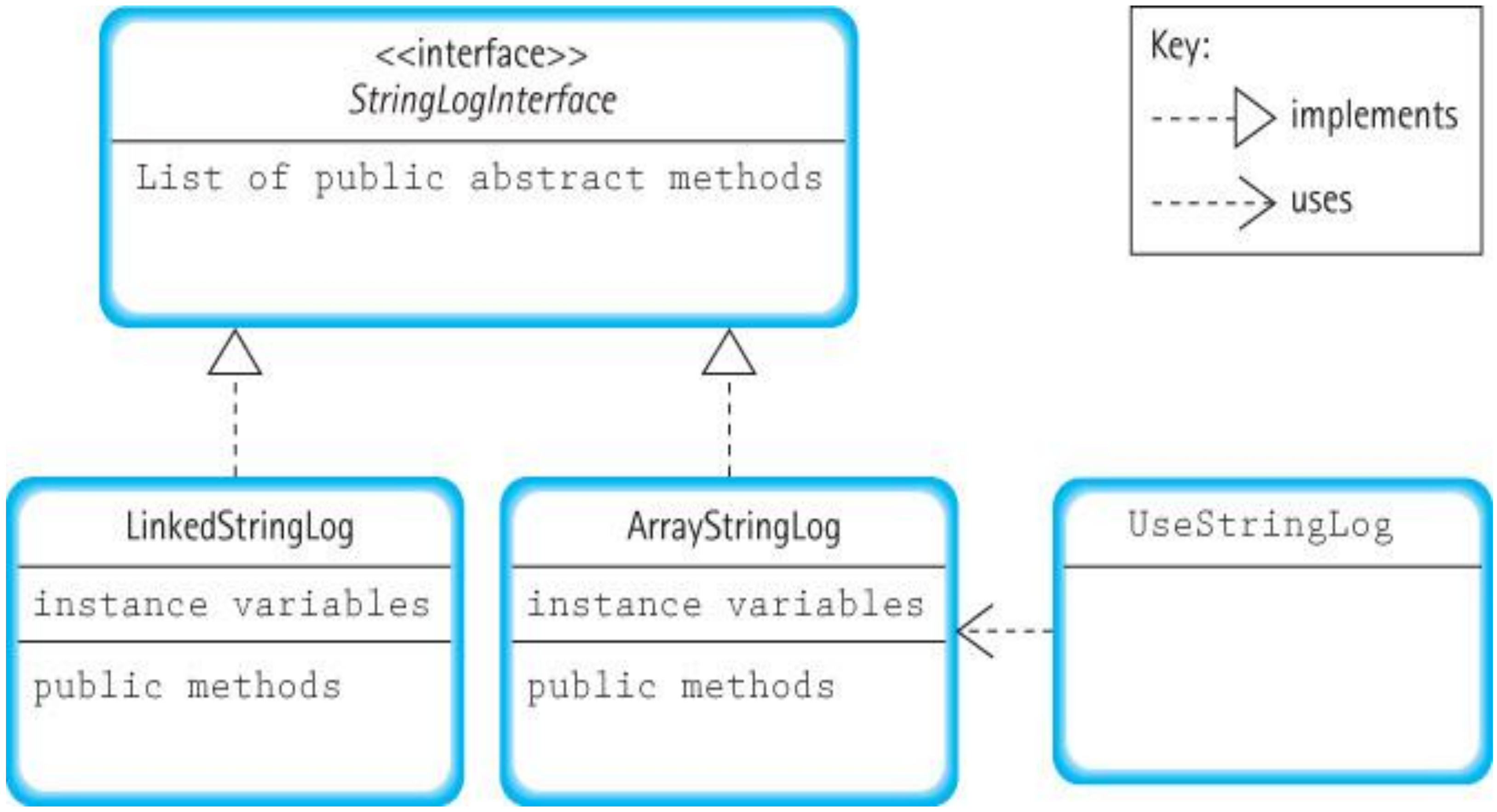
```
Log: Example Use
1. Elvis
2. King Louis XII
3. Captain Kirk
The size of the log is 3
Elvis is in the log: true
Santa is in the log: false
```

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.
  - It is 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.
  - The LinkedStringLog class (see Section 2.6) also provides an implementation.

UML:

Relationships among StringLog classes



2.3 Array-Based StringLog ADT Implementation

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

Instance Variables

- String[] log;
  - StringLog elements are stored in an array of String objects named log.
  - log.length tells the length of the ‘log’ array
- int lastIndex = -1;
  - Originally the array is empty.
  - When the insert command is invoked a string is added to the array.
  - lastIndex is used 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;
}                 // Constructor efficiency is O(N) where N is maxSize

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;
}
```

Overloading: Repeating a method name with a unique signatures
Signature: distinguishing features of a method heading
  Method name
  Number and types of its parameters in their given order

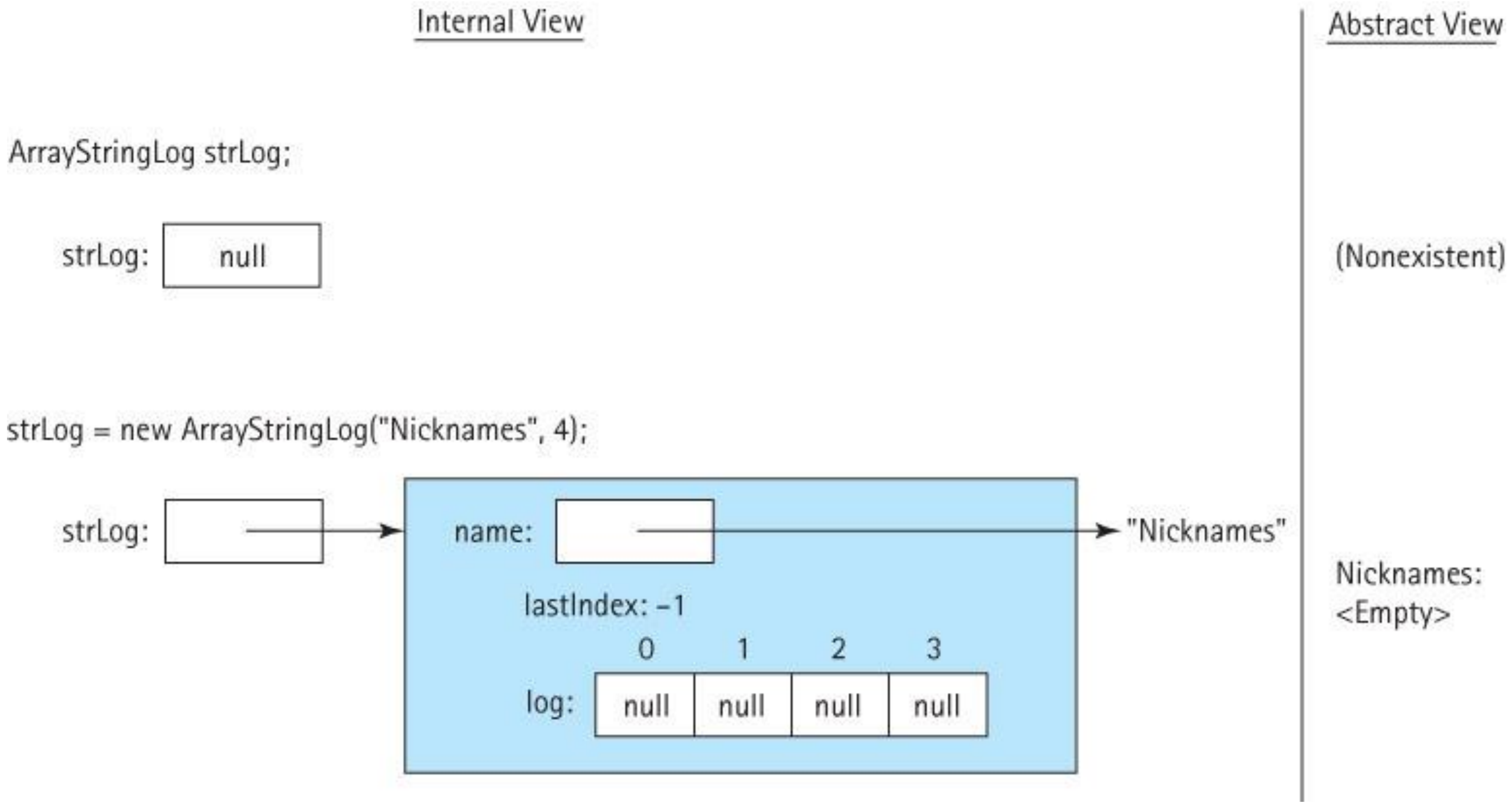
Transformers: The insert operation

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

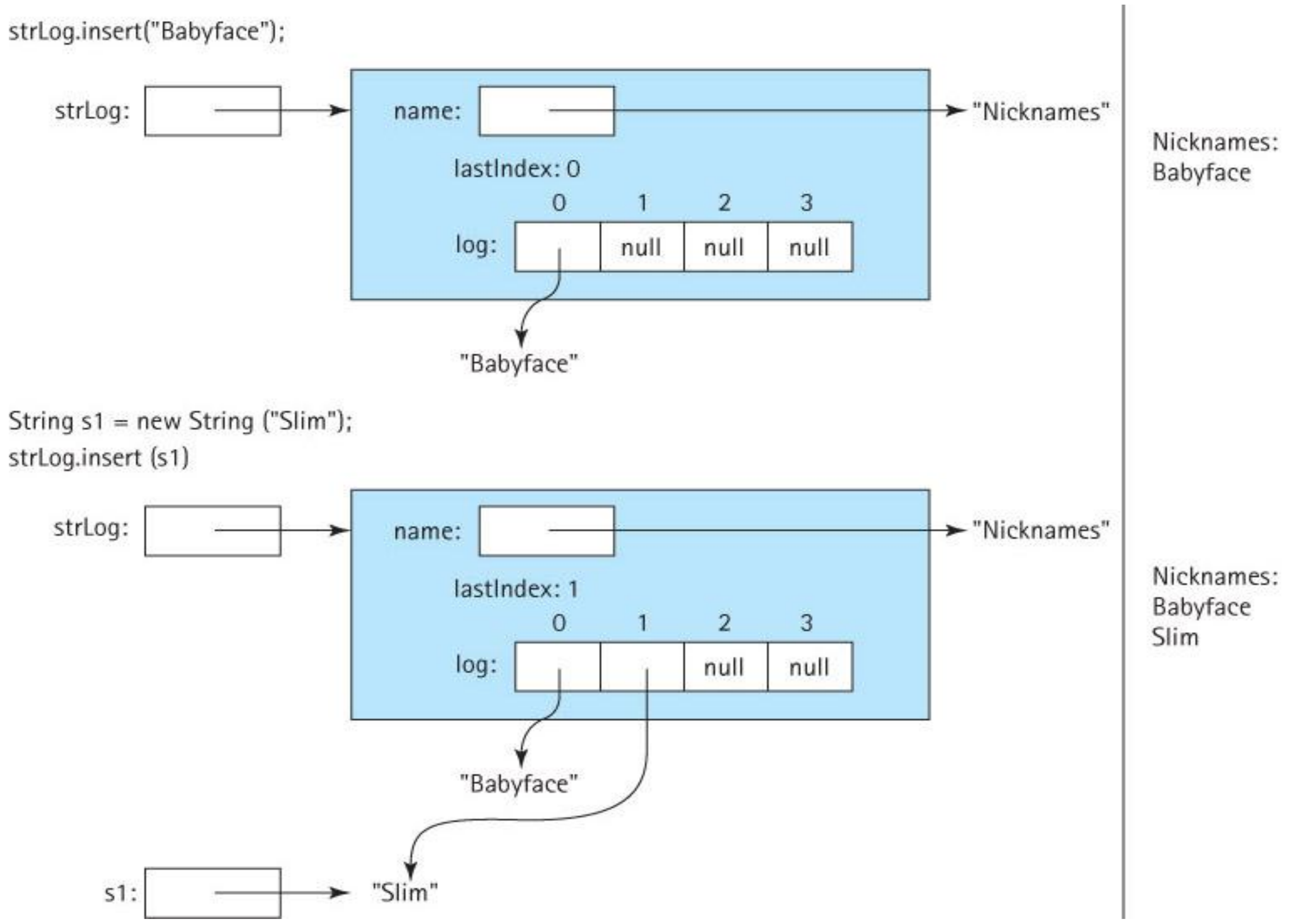
An example use:

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

Example use of insert



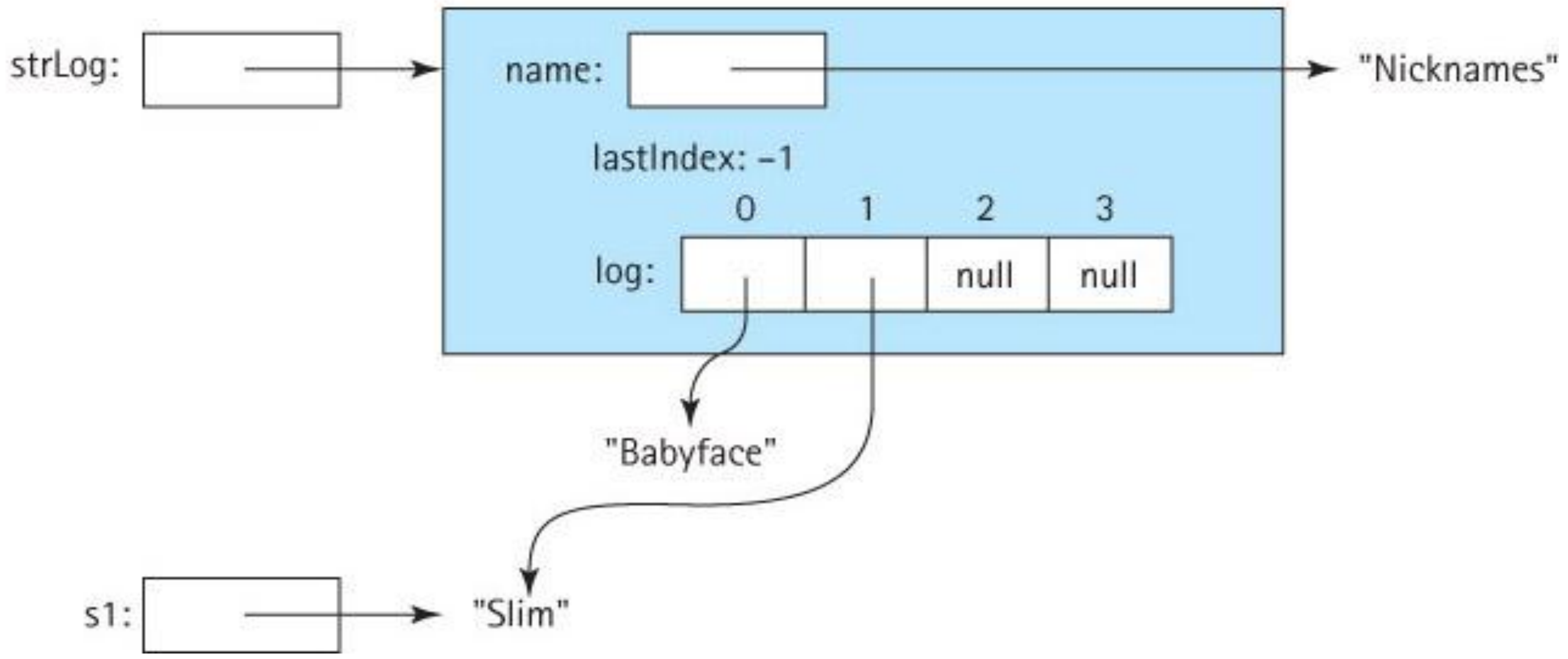
Example use of insert continued



The clear operation

The "lazy" approach:

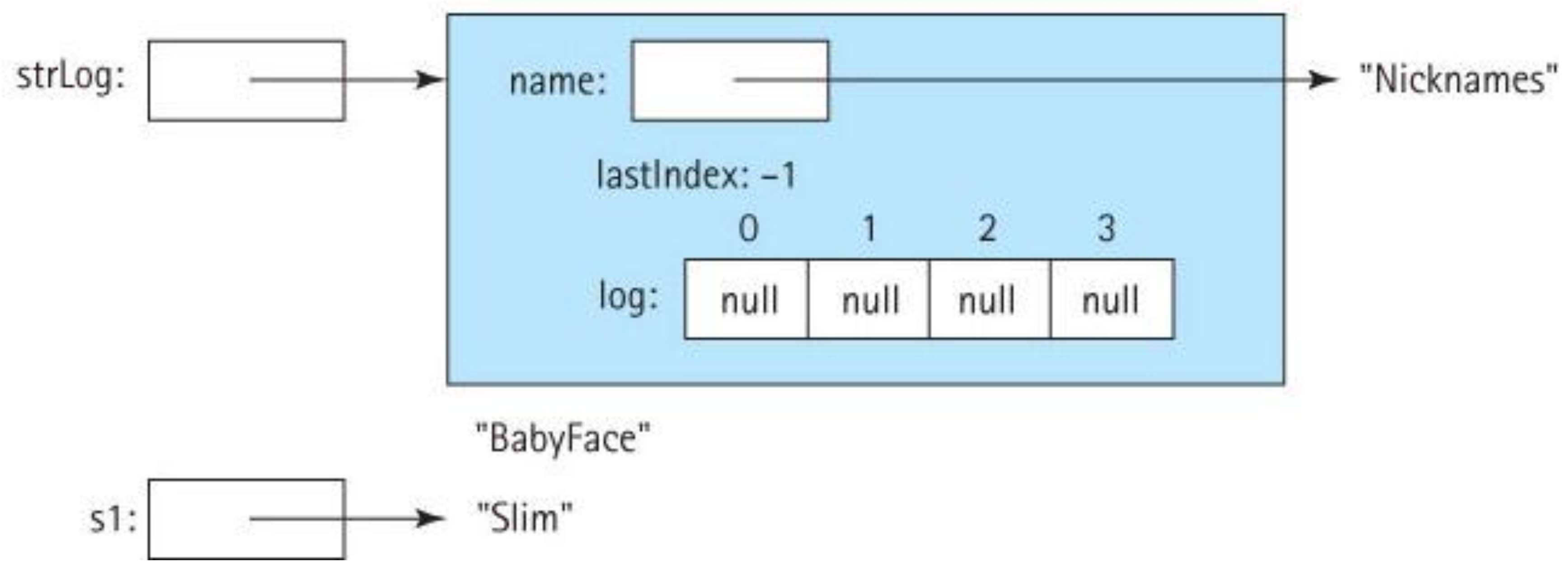
```
public void clear()
// Makes this StringLog empty.
{
 lastIndex = -1;
}
```



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;
}
```

Why???

What is the Big-O for each?

O(1)

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
```

Lab

- Create a new Java Project
  - Name it something meaningful
  - Import ArrayStringLog.java
  - Import StringLogInterface.java
  - Import UseStringLog.java
  - All imports are in bookfiles\ch02
- Do exercises 21-26
  - These add function to the ArrayStringLog class
  - Note: this is also part of your HW assignment, so… you might want to save it.

Stepwise Refinement

- Approach a problem in stages.
- Similar steps are followed during each stage, with the only difference being the level of detail involved.
- Completion 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.
    - Pseudo code evolves into real code
  - 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.
    - Author of cookbook

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)
}
```

Implemented through…
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;
}
```

Eclipse for total beginners lessons 3-5

2.4 Software Testing
a facet of software verification

- The process of executing a program with data sets designed to discover errors
  - Hypothesis: the code is good and robust
  - Method: Verify that the logic is sound
    - Test all boundaries
    - Provide garbage input
    - It only takes one failed example to disprove a hypothesis

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
- **Unit Testing**
  - Debug modules one at a time rather than the entire program

Identifying test cases

- **Functional domain**
  - The set of valid input data for a program or method
- When the functional domain, is extremely small
  - verify a program unit by testing it against every possible input element
  - ***Exhaustive testing***
    - can prove conclusively that the software meets its specifications
    - Impractical, when the functional domain is large or very large
      - Impractical, but not necessarily impossible
      - What would happen to Micosoft if exhaustive testing was instituted on all new OS and software releases?

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 dimension/category combination
- This is basically **Black-box testing**
  - The tester must know the external interface to the module
    - its inputs and expected outputs
    - does not need to consider what is being done inside the module (the inside of the black box).

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

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.

Lab: 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!

Books take: 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:

```
public class Test034

import ch02.stringLogs.*;
{
  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");
  }
}
```

Book Take: 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.

Chapter 2: Homework

- Book Questions
  - 7, 8, 9, 15, 18, 21-26 (finish and submit lab), 33
- Extra Credit: 14, 28