Programming with Data Structures

CMPSCI 187

Spring 2016

- **Please find a seat**
  - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

Reminders

- Read course webpage.
- Make sure you've received a Piazza invitation by email and that you've logged in.
- Get iClicker **2** and register it in Moodle.
- Assignment 2 is due next week (this one is harder).

Lecture 3: StringLog with Arrays

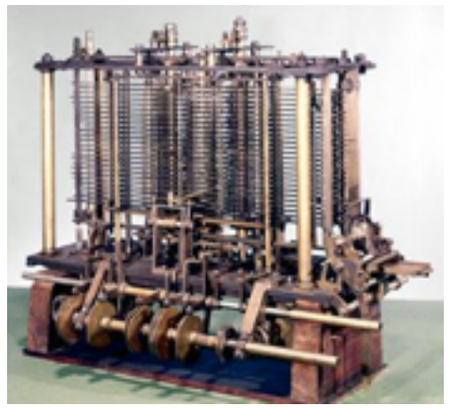
- Abstraction
- Interfaces
- StringLog
- Code for the StringLog interface
- Array-based implementation for StringLog

Abstraction

- **Abstraction**: a simplifying model of an object or process that includes only the essential details (irrelevant details are ignored).
- **Information hiding**: hiding certain details within a module such that they are not accessible to or modifiable by other modules.
  - This helps manage a complex software system.
  - One module does not need to know the internal details of other modules. Modules provide services to each other through carefully defined **interfaces**.

Abstract Data Type (ADT)

- Data type specified by a set of possible values and associated operations, independent of the particular implementation.
- For example, Java's **`int`** is an ADT.
  - Java specifies the value range of **`int`**, associated operators and expected behavior of operators.
- How the data is represented in hardware and operations implemented in hardware do not matter, and do not need to be specified to programmers.



Abstract Data Type (ADT)

- You can also define **custom ADTs** using Java's classes.
  - Define the '**logical**' view of the data, including the properties of the data, the valid range of values.
  - Define the set of **operations** (intuitively, 'actions' you can apply on the data), including properties and expected behavior of these operations.

Multiple Implementations

- Since ADT is implementation-independent, you can implement the same ADT in multiple ways.
- The first data structure that DJW studies in detail is a **`StringLog`** — a collection of **`String`** objects. We can
  - insert strings into a **`StringLog`**
  - count how many strings it contains
  - test whether a particular string exists in the log
  - dump the entire **`StringLog`** into a single string.

Multiple Implementations

- One way to implement StringLog is with an **array** of strings: **`String[]`**. Of course an array has a fixed length, where a real log may expand over time.
- We can also implement StringLog using a **linked list** structure, where each node contains a String object and a pointer to the next node.
- If we write code that only uses the specified operations of a StringLog, it should work equally well whichever implementation is chosen.

Java's Interfaces

- An **interface** is similar to a class, but it can only contain **constants** and **abstract methods** (method signatures) and not the actual implementation of the methods (i.e. method body).
  - in Java 8, an interface may include **`default`** methods with method bodies.
- An interface cannot be instantiated.
- However, you can define a variable of the type of an interface. You will see an example soon.

Java's Interfaces

- A class can **implement** (similar to inherit) an interface, and if so it must implement all the methods specified by the interface.
- A class can implement multiple interfaces, and if so it must implement methods specified by all interfaces.
- An interface can be extended (similar to a class).

FigureGeometry Interface

```
public interface FigureGeometry {
   final float PI = 3.14f;  // constant
   float perimeter();  // by default, methods are
   float area();       // all public and abstract
   void setScale(int scale);
   float weight();
}
```

The **`FigureGeometry`** interface is used for geometric figures. It defines the set of operations that we want to carry out for any specific geometric figure, be it a circle, a square, or a triangle.

A Circle Class

```
public class Circle implements FigureGeometry {
  protected float r;
  protected int s;
  public Circle(float r) { this.r = r; }
  public float perimeter() {
    return (2 * PI * r);
  }
  … // all interface methods must be implemented
}
```

A **`Circle`** is a specific implementation of the **`FigureGeometry`** interface. Think of **`FigureGeometry`** as the parent / superclass of **`Circle`**.

A Circle Class

```
Circle c1 = new Circle(1.0f);  // ok

FigureGeometry f;  // ok

f = new Circle(2.0f);  // ok

f = new FigureGeometry(); // Error!!!
```

It's ok to define a variable of type **`FigureGeometry`** (even though it's an interface), and assign it to a Circle object (which implements the interface). However, recall that you cannot instantiate a **`FigureGeometry`** object!

Defining the StringLog ADT

- So at the abstract level, what is it?
- We want it to have a name (or title), and store a collection of strings (think of log records).
- We want to be able to insert new strings to the collection, check if a particular string is in the collection, count the number of strings, and print out the entire collection of strings. These are the methods we need.
- For now we will not delete any specific string from the collection, but we can clear the entire collection.

Three Types of Methods

In general, we can group methods into three categories:

- **Constructors** to create and initializes new objects
- **Transformers** to alter the data fields of an object
- **Observers** to read, or get information about, the data fields.

The simplest transformers are **setters** and the simplest observers are **getters**.

Constructors

A log that has an **unbounded** number of entries.

```
l1 = new StringLogImplementation(“Todo List");

l2 = new StringLogImplementation("Class", 100);
```

A log that has a **bounded** number of entries.

Transformers

Transformers **alter the content** of the StringLog in some way. We will look at two such operations:

- **insert** - this operation will insert a new string into the log. It does not dictate any relationship between the strings in the log (e.g., ordering).
- **clear** - this operation resets the StringLog to the empty state. The name of the StringLog remains the same.

Observers

Observers return **observed information** about the state of the StringLog. We will look at five observers:

- **contains** - case **insensitive** search for a string in the StringLog
- **size** - returns the number of strings
- **isFull** - returns true if the StringLog is full. If full, the client should no longer call insert.
- **getName** - returns the name of the StringLog
- **toString** - returns a nicely formatted string representing the entire contents of the StringLog

Code for the Interface

```
public interface StringLogInterface {
   // insert assumes log is not full
   void insert(String element);
   boolean isFull();
   int size();
   // case insensitive search
   boolean contains(String element);
   void clear();
   String getName();
   String toString();
}
```

Choosing Data Fields

- We a StringLog object must store a name and a collection of strings.
- To store the string collection, today we will look at the first implementation using an **array.**
- In the next lecture, we will use a **linked list**.
- How big should the array be?

ArrayStringLog

- We'll set the size of the array (i.e. capacity) when creating the object (i.e. in the constructors).
- Since data is stored consecutively in the array, we keep track of the **index of the last string** inserted to the array.
- The index is initialized to -1, indicating no string has been inserted yet.

```
protected String[] log;
protected int lastIndex = -1;
```

- At any time, the valid data is in the index range **`[0, lastIndex]`**, inclusive on both ends.

Coding Constructors

```
public ArrayStringLog(String name, int maxSize) {
    log = new String [maxSize];
    this.name = name;
}
public ArrayStringLog(String name) {
    this (name, 100);  // default capacity 100
}
```

- Unlike DJW, we use the **`this`** constructor call here to avoid repeating the code of the first constructor when we do the same job with the second.

Coding Transformers

- To insert a new string, we make the next location (in the array) active by incrementing **`lastIndex`**, then fill that location with the given string.
- Insertion assumes there is an available spot (this is called a *pre-condition*). The user should call **`isFull`** beforehand to check.
- To clear the log (leaving the name and capacity the same), we just need to change the **`lastIndex`** to **`-1`** again to indicate the log is now empty (hence the number of available spots is the capacity again).

Coding Transformers

```
public void insert (String element) {
    lastIndex++;
    log[lastIndex] = element;
}
public void clear ( ) {
    for (int i = 0; i <= lastIndex; i++)
        log[i] = null;
    lastIndex = -1;
}
```

- Wiping out the strings (i.e. setting references to null) is technically not necessary, but is a good practice for Java's garbage collector release the memory occupied by the strings.

Coding Observers

- The easy code first (three of the five observers):

```
public boolean isFull( ) {
     ???
}
public int size( ) {
     ???
}
public String getName( ) {
     return name;
}
```

Coding Observers

- The easy code first (three of the five observers):

```
public boolean isFull( ) {
     return (lastIndex == (log.length - 1));
}
public int size( ) {
     return lastIndex + 1;
}
public String getName( ) {
     return name;
}
```

Coding Observers

- **contains**: to test whether a given string is in the log, we have to go through the entire collection of strings as it might be anywhere in the list.
- **toString**: to assemble a single string with the content of the entire collection, we similarly have to go through each string in the array again.
- Complexity: while the methods on the last slide take O(1) time each, the contains method and toString method each takes O(n) time.

Code for **`contains`**

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

- We have a **`while`** loop where we could have used a **`for`** loop. Note that we take advantage of Java String's built-in method to do case insensitive comparison.

Code for `toString`

```
public String toString( ) {
    String logString = "Log: " + name + "\n\n";
    for (int i = 0; i <= lastIndex; i++)
        logString += ((i+1) + ". " + log[i] + "\n");

    return logString;
}
```

- For toString we get a title line, two blank lines, then each string on its own line with a number.

Code for `toString`

- Example Output:

```
Log: CMPSCI 187
1. Assignment 1 due
2. Discussion section 1
3. Lecture 3
… … …
```

Using the StringLogInterface

```
public static void main(String[] args)
{
  StringLogInterface l;
  l = new ArrayStringLog("Example Use");
  l.insert("Elvis");
  l.insert("King Louis XII");
  l.insert("Captain Kirk");
  System.out.println(l);  // internally calls toString
  System.out.println("Log size is " + l.size());
  System.out.println("Elvis is in the log:”+l.contains("Elvis"));
  System.out.println("Santa is in the log:”+l.contains("Santa"));
}
```

Note that `l` has type `StringLogInterface`

but references a `ArrayStringLog` object