Notes – Linked List Example
August 2020
Let’s look at another example with linked lists.

**Problem scenario:**

We want to create a linked list of books.

For this example, let’s just have the ability to add a book to the list and to print the list.

**Some design specifics:**

Use “Book” nodes that keep track of the book’s title (only).

Use “BookNodes” that are the actual nodes in the linked list. These have book fields and “pointer” fields.
More design specifics:

Use a “BookList” class to represent the collection of books. This, in essence, is the interface to the linked list that “main” will access.

Design summary:

In main an object of type “BookList” will be created and used to add books to the list.

“BookList” will manage a list of “BookNodes”.

A “BookNode” has instance variables of “Books” and references to other “BookNodes”.
class Book
{
    private String title; // instance variable

    public Book (String newTitle) // Constructor
    {
        title = newTitle;
    }

    public String toString ()
    {
        return title;
    }
}
Start of public class Library

```java

BookList books = new BookList();

An object of type “BookList” is created.

Recall: BookList manages a list of BookNodes. So let’s work on the BookNode class next!
```
class BookNode
{
    private Book book;
    private BookNode next;

    // Constructor to set up the node
    public BookNode (Book theBook)
    {
        book = theBook;
        next = null;
    }
}

Normally there would be some accessor and mutator methods in this class. There are none given, why?  (No, the Instructor did not forget!)
```java
class BookList {
  private BookNode list;

  // Constructor sets up an initially empty list of books.
  public BookList () {
    list = null;
  }

  public void add (Book newBook) { // add to end of list
    BookNode node = new BookNode (newBook);
    BookNode current;

    if (list == null) {
      list = node;
    } else {
      current = list;
      while (current.next != null) { // no accessor method
        current = current.next;
      } // no mutator method
      current.next = node;
    }
  }
}
```
public String toString() {
    String result = "";
    BookNode current = list;

    while (current != null) {
        result = result + current.book.toString() + "\n";
        current = current.next;
    }
    return result;
}

Regarding .....  current.book.toString()

Current is a reference to a BookNode (list). The book field is from the BookNode object, toString is from the Book class.
Let’s answer why no accessor and mutator methods were used in the BookList class.

Take a look at the actual program code and see where the BookNode class is located.
This is called an “inner class”. We will spend more time talking about such classes later but for now let’s consider the following:

A nonstatic, nested class is called an *inner class*. Because it is not static, an inner class is associated with each instance of the enclosing class.
No member inside an inner class can be declared static.

An instance of an inner class can exist only within an instance of the enclosing class.

The inner class feature is valuable because it allows you to group classes that logically belong together and to control the visibility of one within the other.

In the example, a useful feature is that the outer class can access the inner class members directly.
Inner classes - continued

Special .class filename for the inner class. Consider the Library example and note the $.

If you want to make an object of the inner class anywhere except from within a non-static method of the outer class, you must specify the type of that object as:

OuterClassName.InnerClassName
Part of the first programming assignment:

Write a program that utilizes (and implements) a stack construct. Use a linked list data structure and use an inner class for the nodes. The main should have a loop that asks the user what operation he/she desires.

Choices are: Push an element, pop an element, peek at the top element, and determine if the stack is empty.

To impose a degree of “conformity” – let’s get introduced to a new Java construct.
Java Interfaces

This topic is fairly complex when its full use is discussed. However, at its basic level, it is fairly simple. For now, we will look at it in this way.

A Java interface is a program component that contains public constants, signatures for public methods, and, ideally, comments that describe them.

The methods are abstract in that they do not contain an implementation. The header of the method, including its parameters, is simply followed by a semicolon.

An interface cannot be instantiated, but we will see that variables can be declared with its type.
Java Interfaces - continued

The methods in an interface are public by default. So you can omit “public” from their signatures.

The interface does not declare any constructors.

An interface can contain fields, but these are implicitly static and final.

In essence, an interface provides only a form, but no implementation.

A separate class will “implement” an interface by providing method implementations for each of the abstract methods defined in the interface.

For those of you familiar with abstract classes, an interface is different in that it is not a “base class”.
Using an interface is a two part process. First, the interface is written using the keyword “interface”.

This interface is stored in a file with the same name and with a .java extension.

Next, some other class needs to implement this interface. That is done using the keyword “implements”.

**A class can implement more than one interface. For now, let’s stick with some very basic examples.**

**Example 1:**

*Imagine that we want to declare an interface for something called a “Speaker”. A speaker can speak (give his/her philosophy) and can also announce (make an announcement).*
Writing the interface

public interface Speaker
{
    public void speak();
    public void announce(String str);
}

Store this in a file called "Speaker.java". Compile it in the usual way.

That’s all there is to writing a simple interface!
Writing one class to implement the interface

public class Philosopher implements Speaker
{
    private String philosophy;

    public Philosopher (String thoughts)    // constructor
    {  philosophy = thoughts;  }

    public void speak ( )
    {  System.out.println (philosophy);  }

    public void announce (String announcement)
    {  System.out.println (announcement);  }
}
Writing a second class to implement the interface

```java
public class Dog implements Speaker {

    // no instance variables and no constructor

    public void speak () {
        System.out.println("woof");
    }

    public void announce (String announcement) {
        System.out.println("woof: " + announcement);
    }
}
```
Note: The Philosopher and Dog class each implement the two methods in the interface class, albeit in different ways.

A class that implements an interface may have additional data members and methods.

Now let’s look at how to use these classes!
public class Talking {
    public static void main (String [ ] args) {
        Dog fido = new Dog();
        fido.speak();

        Philosopher Yogi_berra = new Philosopher("It ain't over till it's over!");
        Yogi_berra.speak();
        Yogi_berra.announce("This is like deja vu all over again.");
    }
}
Before we get to the next example, let’s look at the programming assignment and define an interface.

```java
public interface MyStack
{
    public void push(int item);
    public int pop();
    public int peek();
    public boolean isEmpty();
}
```
Program Assignment – *MyStack*

Let’s design the solution with three classes:

1) A “driver” class that makes calls to push, pop, peek and isEmpty

2) A class that manages the “stack of nodes”. This will be similar to the “BookNodes” class in the example from last time.

3) A class that represents nodes. This is similar to the “Book” class in the example from last time. This class will be an inner class.

Question: Which of these three “implements” the MyStack interface?