Programming to an Interface

- We develop some data structure classes
- We don't need to...
  - Java 1.1 standard library already has some (`Vector`, `Hashtable`, `Stack`)
  - Java 1.2 standard library has even more (`LinkedList`)
  - There are other, better data structure libraries (e.g. the Java Generic Library, JGL)
- ...but doing so will illustrate some new bits of Java (e.g. packages, inner classes, Java interfaces)
- and, more importantly, the ideas of *programming to an interface* and *design patterns*

Packages

- Related classes can be grouped into *packages* (corresponding to your directory structure). For package `datastructs`, each file starts with:

```
package datastructs;
```

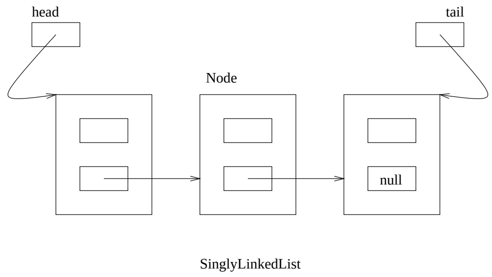
- *Classes* can be visible to other packages (`public`) or visible only within the package (leave off the word `public`)
- Methods & variables *within a class* then have their own visibility within the class' visibility
- If class `C` is `public` in package `p1`, a client outside `p1` can use `C` if we set the `CLASSPATH` and include an `import` statement.

The interface of `SinglyLinkedList`

```
public boolean contains(final Object anOb-
ject)
public Object getFirst()
public Object getLast()
public boolean isEmpty()
public int size()
public Enumeration elements()
public void add(final Object anObject)
public void addFirst(final Object anObject)
public void addLast(final Object anObject)
public void remove(final Object anObject)
public void removeFirst()
public void removeLast()
public String toString()
```

And `equals(obj)`, `clear()`, `add(idx, obj)`, `get(idx` `indexOf(obj)`, `remove(idx)`, etc., etc.

Implementing `SinglyLinkedList`

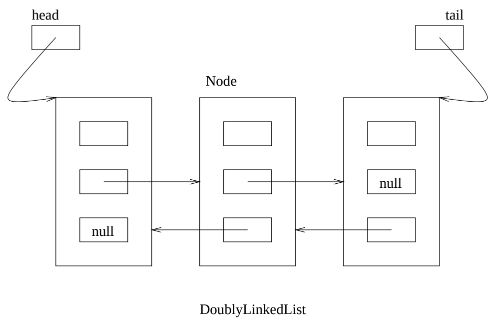


- Why point to head and tail?
- Two classes: `Node` and `SinglyLinkedList`
- `Node` is useful only within `SinglyLinkedList`, so can we 'hide' it from other classes?

‘Hiding’ a class

- Give it default visibility — visible in same package only
- Put it into same file — generally not attractive
- Make it an *inner class*
  - Define it *within* another class definition
  - 4 possibilities: nested classes, member classes, local classes and anonymous classes
  - Very complicated visibility rules, imperfectly implemented
  - We’ll use a *member class*, and we’ll make it `private` (not allowed for classes in general, but allowed for member classes)

Implementing `DoublyLinkedList`



- What is advantage/disadvantage compared with singly-linked list?
- Two classes: `Node` and `DoublyLinkedList`
- `Node` as a member class

Changing to a new data structure

- Revise client(s)
  - Change `SinglyLinkedList` to `DoublyLinkedLi` throughout
  - Recompile all changed clients
  - Error-prone and especially difficult if many client classes
- Revise data structure class definition
  - Change the class name, constructor name, filename, etc. for `DoublyLinkedList` to `SinglyLinkedList`
  - Means changing and recompiling only one file
  - But puts the other class definition out of service and can give badly named class definitions

Java interfaces

- **interface** (1) The services an object or class provides to its clients. (2) The Java language construction for defining a type by supplying only the set of services that can be invoked on values of that type.
- i.e. a set of method headers, with no method bodies
- Classes (headers and bodies) may then be declared to *implement the interface*: they offer implementations of all the methods in the interface
- (The classes might have other methods as well.)

Polymorphism, again

- *Polymorphism:* the ability to take several forms
- A variable of a superclass type may refer to objects of that class or its subclasses
- A variable of a Java interface type may refer to objects from any class that implements the Java interface

```
Sequence seq;
seq = new SinglyLinkedList();
seq = new DoublyLinkedList();
```


Dynamic method binding, again

- *Dynamic method binding:* deciding, *at run time*, which of several methods to run
- The decision is made on the basis of the class of object being referenced

  ```
  Sequence seq;
  seq = new SinglyLinkedList();
  seq.addFirst("abcd");
  ```

  will invoke the definition of `addFirst` to be found in `SinglyLinkedList`
- Use polymorphism and dynamic method binding to write clients that are much more changeable: to change the data structure, change one line and recompile the client

Programming to an Interface (Near quotation from Gamma et al, p.18)

There are two benefits to manipulating objects solely in terms of the services defined by a Java interface:

1. Clients remain unaware of the specific class of objects they use, as long as the objects adhere to the interface that clients expect.
2. Clients remain unaware of the class definitions that implement these objects. Clients only know about the Java interfaces.

This so greatly reduces the implementation dependencies (coupling) between subsystems that it leads to the following principle of re-usable object-oriented design:

> *Program to an interface, not an implementation.*

When should I program to an interface?

- Always.
  - Avoid hard-wiring associations between objects
  - Avoid hard-wiring the sending of messages between objects
- Well... not quite always. But especially wherever you anticipate change:
  - data structures (changed for performance)
  - end-user interfaces (changed for usability)
  - platform-specific subsystems (changed for portability)
  - ill-understood subsystems (changed when understanding grows)
  - etc.

Two more implementations of the `Sequence` interface

- Java provides a `Vector` class: expandable arrays
- But they don't exactly implement our interface (missing methods, methods with different names)
- One re-use strategy: create a subclass of `Vector`
  - problems if `final` or if if can't successfully overload/override
- Another re-use strategy: write an *adapter class*
  - the adapter contains a `Vector`; messages sent to the adapter are translated into operations on the `Vector`

Design Patterns

- A proven way of putting together a set of communicating objects and classes to solve a general (recurring) design problem in a particular context.
- These days, we catalogue design experience for re–use of successful design experience
- In these catalogues, a pattern consists of
  - a name (makes communicating about designs easier)
  - a problem description (when to apply the pattern)
  - a solution description (in general terms, the objects, classes, messages, etc. — as a kind of template)
  - the consequences (trade-offs, benefits, costs)

Adapter Pattern (very very much simplified)

- Name: adapter (Also known as wrapper; Strictly this is object adapter)
- Problem: You want to re-use an existing class definition, C, but its interface does not match the one you need.
- Solution: Your client sends messages to C', the Adapter, which then sends messages to its associated C.
- Consequences: Introduces a level of indirection which worsens performance.

Reuse

- Reuse of code
  - Code salvaging (cut-and-paste)
  - Direct use of existing class definitions
  - Subclassing an existing class definition
  - Adapting an existing class definition
- Reuse of designs
  - e.g. design patterns
- Reuse of documentation, tests, . . .

Iterator: Another Design Pattern

- Clients need to visit each item in a data structure and do some processing to each item
- Poor solution: add more & more methods to the data structure class (unfeasible)
- Poor solution: pass them the list (exposes the implementation)
- Good solution: another class, the iterator class, which can see the implementation, offers a standard set of methods for traversing the data structure

Iterator Pattern (very very much simplified)

- Name: iterator (Also known as cursor)
- Problem: You want to access the elements of a data structure sequentially without exposing its underlying implementation
- Solution:
  - Your client sends the `elements()` message to the data structure to create the iterator object;
  - this object has sufficient knowledge of the implementation of the data structure to offer `hasMoreElements()` and `nextElement` methods to the client
  - (The iterator class will satisfy a Java interface (`Enumeration`) so that traversing different data structures has a uniform interface.)

Iterator pattern, cont'd

- Consequences:
  - Separating the traversal mechanism from the data structure allows definition of different traversal policies without bloating the data structure class or making it application-specific;
  - we can have multiple traversals going on at once;
  - there are dangers in modifying the data structure (e.g. deleting an object from it) while a traversal is going on (depending on whether the iterator object works on the original data structure or a copy of it)