CSCI 136:
Data Structures
and
Advanced Programming

Lecture 12

Abstract data types

Instructor: Kelly Shaw

Williams

Topics

- ADTs
- More linked lists

Your to-dos

1. Read **before Fri**: Bailey, Ch 9.4–9.5.
2. Quiz 4, **due Saturday by noon**.
3. Lab 4, **due Tuesday 10/11 by 10pm**.

Announcements

- Colloquium: **What I Did Last Summer (Industry)**, 2:35pm in Wege Auditorium with **cookies**.

Linked List

A **linked list** is a recursive data structure. A linked list is composed of simple pieces called **list nodes**. A list node contains **data** (of generic type **T**) and a **reference** (a "link") to either **another list node** or **null**.

Linked List

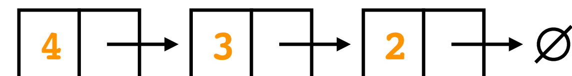
∅

The empty list is defined as **null**.

Linked List

Every other list has at least one list node.

Singly Linked List



There's only **one link** in each node, to the **rest** of the list.

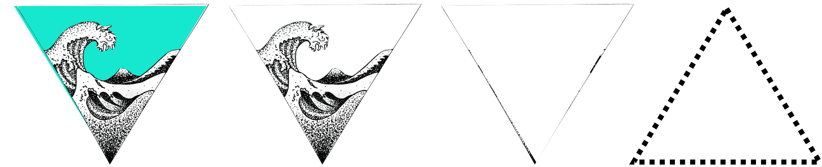
How would I represent the above idea **in Java**?

The purpose of a class:

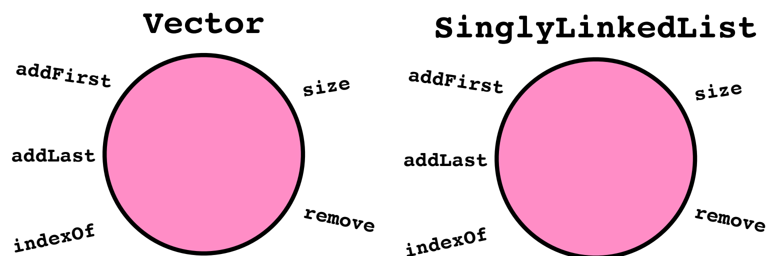
To "abstract away" implementation details.

Abstraction

Abstraction is the process of **removing irrelevant information** so that a program is easier to understand.



Do you see any similarities?



The two classes share the same **interface**.

Interface

An **interface** defines a boundary between two systems across which they share information. An interface is a **contract**: calling a method defined in an interface returns the data as promised.

Because an interface **contains no implementation**, programmers who use them **cannot rely on implementation details**.

E.g., the `List` interface states that there must be an `add` method but does not say how it should be implemented.

List

A **list** is an **ordered collection** of items of an element of type `E`. It supports **prepending** an element to the front, **appending** (**adding**) an element to the end, **finding** an element, and element **removal**.

A `Vector` is a **list**.

A `SinglyLinkedList` is a **list**.

A `DoublyLinkedList` is a **list**.

Observe that this similarity is "deeper" than just what an **interface** provides….

Abstract Data Type

An **abstract data type** is a mathematical formulation of a data type. ADTs abstract away **accidental** properties of data structures (e.g., implementation details, programming language). Instead, ADTs contain only **essential** properties and are **concisely defined by their logical behavior** over a **set of values** and a **set of operations**.

In an ADT, **precisely how data is represented** on a computer **does not matter**.

By contrast: data structure

A **data structure** is the physical form of a data type, i.e., it is an implementation of an ADT. Generally, data structures are designed to efficiently support the logical operations described by the ADT.

For data structures, precisely **how data is represented on a computer matters a lot**. Simple data structures are often composed of simple representations, like primitives, while more complex data structures are composed of other data structures.

`Vector`, `SinglyLinkList`, etc. are **data structures**.

A Vector is a List

structure5

Class Vector<E>

```
java.lang.Object
  └ structure5.AbstractStructure<E>
      └ structure5.AbstractList<E>
          └ structure5.Vector<E>
```

All Implemented Interfaces:

java.lang.Cloneable, java.lang.Iterable<E>, List<E>, Structure<E>

```
public class Vector<E>
extends AbstractList<E>
implements java.lang.Cloneable
```

A Linked List is a List

structure5
Class SinglyLinkedList<E>

```
java.lang.Object
  └ structure5.AbstractStructure<E>
      └ structure5.AbstractList<E>
          └ structure5.SinglyLinkedList<E>
```

All Implemented Interfaces:
java.lang.Iterable<E>, List<E>, Structure<E>

```
public class SinglyLinkedList<E>
extends AbstractList<E>
```

Vector Big-O

operation	worst	best
addFirst(E e)	O(n)	O(1)
get(int i)	O(1)	O(1)
indexOf(E e)	O(n)	O(1)
remove(E e)	O(n)	O(1)
size()	O(1)	O(1)

Singly-Linked List Big-O

operation	worst	best
addFirst(E e)	O(1)	O(1)
get(int i)	O(n)	O(1)
indexOf(E e)	O(n)	O(1)
remove(E e)	O(n)	O(1)
size()	O(n) [O(1) w/mod.]	O(n) [O(1) w/mod.]

Missing from Java: ADT behavior

Java provides no way of specifying behavior independently of implementation.

E.g., a **`List`** interface might require

```
public void prepend(T elem)
```

But there's no way to **require** that an implementation actually *place the element at the beginning of the list*.

The best we can do in Java: static types

Java uses **types** to stand in for ADTs.

However, Java provides some control over **abstractness**, and we can use this control to approximate what we want.

interface → **fully** abstract

abstract class → **partially** abstract

class → **not** abstract

Interface

An **interface** defines boundary between two systems across which they share information. An interface is a **contract**: calling a method defined in an interface returns the data as promised.

An interface **contains no implementation**!

You cannot specify **behavior** at all!

`Honkable`