CSCI 136:
Data Structures
and
Advanced Programming

Lecture 10

Abstract Data Types

Instructor: Dan Barowy

Williams

Announcements

- Lab 3: how's it going?

Outline

1. Practice Quiz
2. ADTs
3. Interfaces

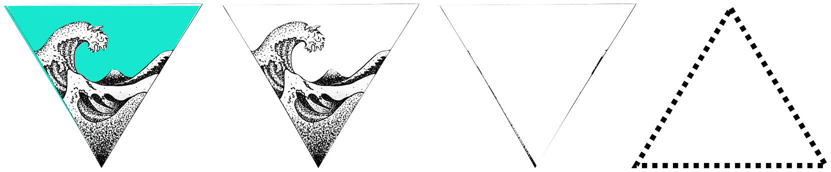
Practice Quiz

The purpose of a class:

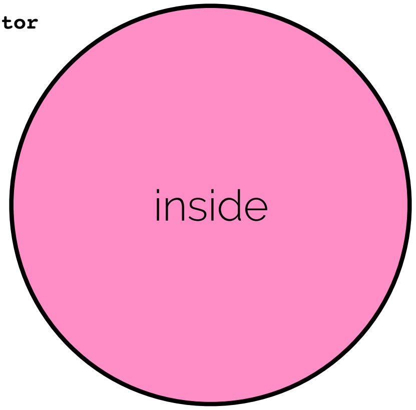
To "abstract away" problems.

Abstraction

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



Think of a class as having two sides.



`Vector`

inside

outside

Design so user never needs to "look inside".

Think of a class as having two sides.

The outside: A class should represent **one idea**, and the class's methods should support working with that one idea.

E.g., `Vector:` Represents an arbitrarily long sequence of elements. Ideally, it also has the same asymptotic properties as an array.

You can:

- `add` to it
- `remove` from it
- ask it for its `size`…
- convert it `toString`
- etc.

Think of a class as having two sides.

The inside: A class should contain whatever is necessary to achieve that **one idea** and nothing else.

E.g., `Vector:` Represents an arbitrarily long sequence of words.

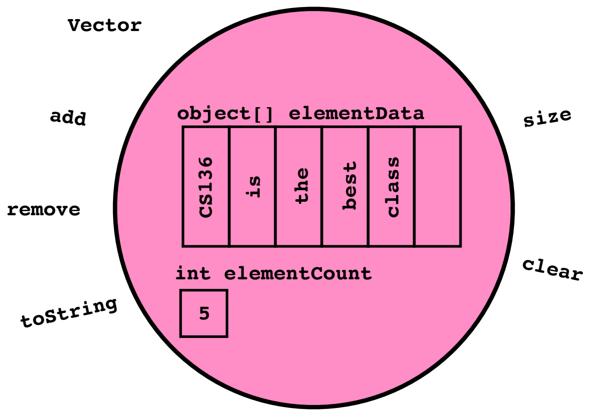
Stores:

- `E[]` of elements.
- `elementCount`.

Ensures:

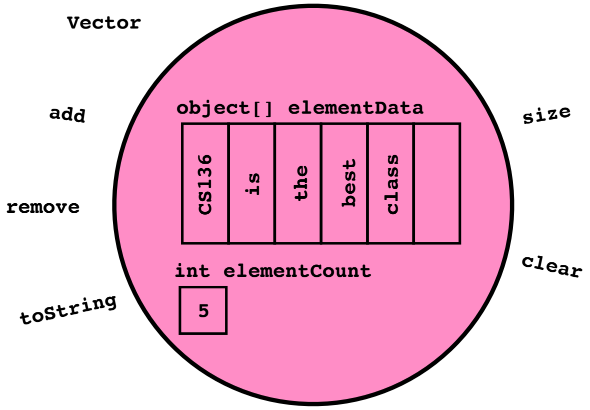
- `String[]` is always big enough (via `ensureCapacity`)

Think of a class as having two sides.

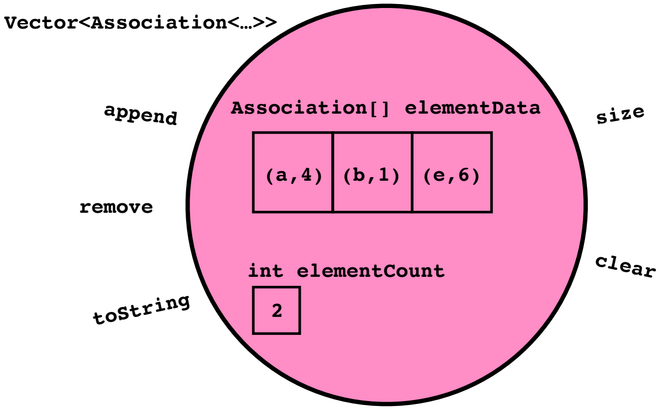


Design so user never needs to "look inside".

Hiding data inside a class is called: encapsulation



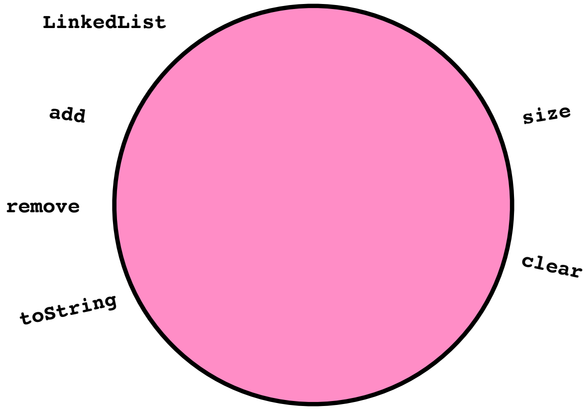
Classes can encapsulate other classes!



This is how we design complex software.

An object stores data and has operations.

Remember LinkedList from last week?



`LinkedList`

`add`

`remove`

`toString`

`size`

`clear`

“Outside” is very similar to Vector!

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.

ADT example: List

A **list** is a linear collection of data elements, whose order is not necessarily given by their placement in memory. Elements may store **any type of value**. A list supports **inserting**, **searching** for, and **deleting** any value in a list, although not necessarily efficiently.

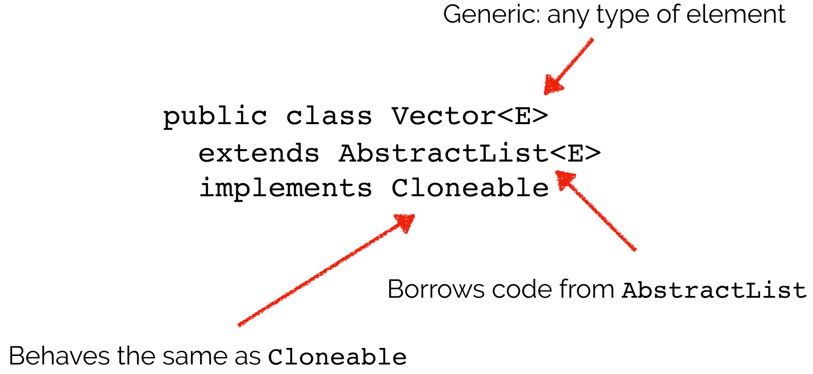
ADTs cannot be expressed in Java

At least **not directly**.

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

Because types in Java are often bound to an implementation, Java provides two mechanisms for programmers to use a type without depending on a mechanism: **interfaces** and **abstract classes**.

Example



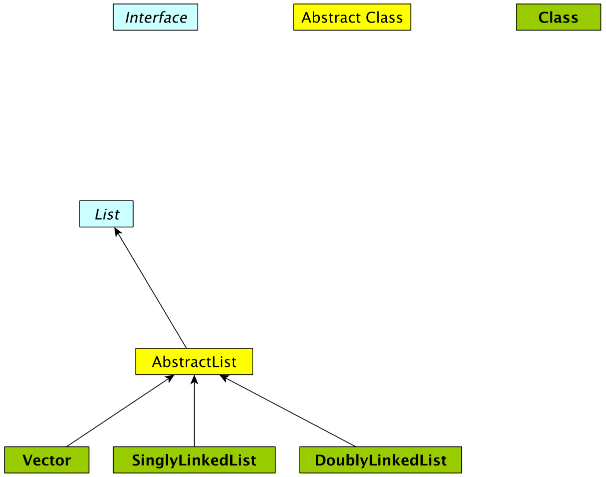
Generic: any type of element

```
public class Vector<E>
  extends AbstractList<E>
  implements Cloneable
```

Borrows code from `AbstractList`

Behaves the same as `Cloneable`

structure5 List implementations



Interface

Abstract Class

Class

List

AbstractList

Vector

SinglyLinkedList

DoublyLinkedList

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.

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

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

Abstract class

An **abstract class** is a partial implementation, mainly used as a **labor-saving device**.

E.g., many `List` implementations will implement methods the same way. Why duplicate all that work?

`isEmpty()` can always be implemented by checking that `size() == 0`.

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 the implementation actually place the element at the beginning of the list.

Next best thing: `assert` statements

This is why we encourage you to write pre- and post-conditions.

E.g.,

```
public void prepend(T elem) {
  T oldHead = head();
  …
  Assert.post(head().next() == oldHead)
}
```

Recap & Next Class

Today we learned:

ADTs

Interfaces

Next class:

The many varieties of List

Mathematical Induction