CS 228: Introduction to Data Structures
Lecture 2

Abstract Data Types and their Implementations

An ***abstract data type*** (ADT) is a specification of a data type from the user's point of view. Formally, an ADT consists of

- a ***class of objects*** and
- a set of ***operations*** on the objects.

Last time, we saw one example of an ADT: a collection of integers. We also saw one way to specify an ADT in Java: via an ***interface***.

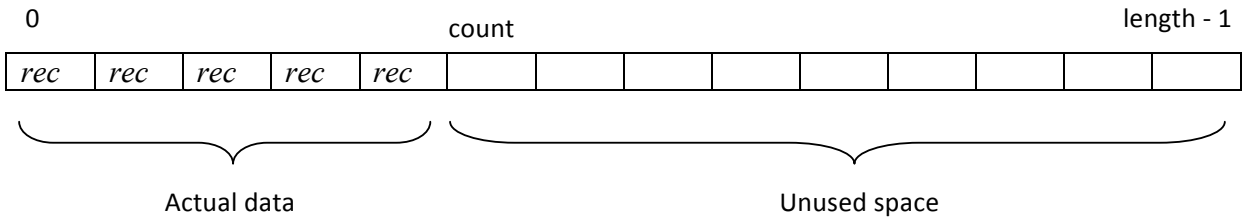
```
public interface IntCollection
{
  void add(int k);
    // adds a new integer to the set

 boolean contains(int k);
    // returns true if k is in collection;
    // returns false otherwise.
```

```
  int size();
   // returns the number of integers in
   // the collection.
}
```

To be able to use an ADT or an Interface we need to ***implement*** it. In Java, we would do something like this:

```
public class UnsortedIntCollection
  implements IntCollection
{
  private int[] data;
  // Other instance variables

  public UnsortedIntCollection()
  { // Constructor: TODO }

  void add(int k) { // TODO }

  boolean contains(int k) { // TODO }

  int size() { // TODO }
}
```

There are often multiple ways to implement an ADT. For example we saw two ways to implement `IntCollection`, both based using an array "`data`" as a backing store:



Approach 1: Unsorted array + linear search
Approach 2: Sorted array + binary search

In both cases, `size()` requires only one operation — just return `count` — irrespective of the number of items in the collection. For the other two methods, we have trade-offs.

Operation	Approach 1	Approach 2
`add(k)`	2 operations: (a) put k in data[count] (b) increment count	(a) Sequential scan to find position to add k. (b) Shift elements right. (c) Add k in vacated position. Worst case: Linear in number of elements.
`contains(k)`	Worst case time: Linear in number of elements. (May examine all elements and fail to find k.)	Worst case: logarithmic in number of elements. (Logarithmic is a lot faster than linear — we'll see why in a few weeks.)

Note that arrays are not the only way to implement collections of integers. ***Binary search trees*** and ***hashing*** are two alternatives. We will study these and other data structures in the second half of this course

Summing up

The three ideas we just saw,

1. defining an ***abstract data type*** using an ***interface***,
2. coming up with a ***data structure*** to implement it, and
3. ***analyzing*** the implementation

will be the recurring themes in this course.

Modularity and Abstraction

ADTs embody two important ideas in software design:

- ***Modularity***, which means building systems out of ***components*** that can be developed independently of each other.

- ***Abstraction***: View components in terms of their essential features, ignoring irrelevant details. Each component provides a well-defined ***interface*** specifying exactly how we can interact with it. Here we are using the word interface in its conventional sense, not specifically as a Java keyword.

Together, modularity and abstraction allow us to reduce ***coupling*** among components. Thus, changes in one component should not force other components to change. ***Object-oriented programming*** (OOP) is one approach to achieving modularity and abstraction.

Objects and classes

In OOP, systems are structured as collections of interacting ***objects*** that communicate through well-defined interfaces. An ***object*** is a software component that has

- ***state*** – instance variables that hold their values between method calls.

- ***identity*** – once you create an object, you can refer to it again, and distinguish it from others

- ***operations*** – its public interface or API (Application Programming Interface). These are the public methods.

A ***class*** is a type of object, say a bank account. Many objects of the same class might exist; e.g., `myAccount` and `yourAccount`.

Textbook authors sometimes summarize the principles of object-oriented programming in the form of three ideas (the "pillars" of OOP):

- ***Encapsulation*** means that objects only interact through a well-defined set of operations (the public ***interface*** or API), while the representation of the state, and the implementation of the operations are kept hidden.

- ***Inheritance*** allows us to derive a class of objects from a more general class. A derived class inherits properties from the superclass from which it derives. For example, the `SavingsAccount` class might inherit from the `BankAccount` class the property of storing a balance.

- ***Polymorphism*** is the ability to have one method work on several different classes of objects, even if those classes need different implementations of the method.

For example, one line of code might be able to call the `add` method on *every* kind of `List`, even though adding an item to a list of `BankAccounts` might be completely different from adding an item to a list of integers.

OOP allows us to easily create small modules that closely model one coherent thing or concept in the problem domain. In the bank account example, what we really want is something that models a bank account. It should keep track of its own attributes and state (e.g., balance, transaction history, etc.) and provide the rest of the system with an interface to update and query its state.

Why encapsulation is your friend:

- *Encapsulation makes implementation independent of functionality.* A programmer who has the documentation of the interface can implement a new version of the module or ADT independently. The new implementation can replace the old one.

- *Encapsulation prevents us from writing applications that corrupt a module's internal data.* This reduces debugging time. A lot.

- *Encapsulation facilitates teamwork.* Rigorously defined module interfaces allow a complex programming project to be broken up into manageable pieces. The modules can then be independently implemented by different programmers.

- *Encapsulation facilitates documentation and maintainability.* Unambiguous interfaces make it easier for other programmers to fix bugs that arise years after you've left the company. Bugs are often a result of unforeseen interactions between modules. A clear specification of each interface and each module's behavior makes bugs easier to trace.

Java offers facilitates encapsulation through things such as packages and the `private`, `package`, and `protected` modifiers for field and method declarations. You've already seen several of these ideas in CS 227. We'll review the key points next.

Review: OOP in Java

The next example reviews some basics about objects in Java.

```
public class Point
{
  private int x;
  private int y;

  public Point()
  {
    // x and y get default value 0
  }

  public Point(int x, int second)
  {
    this.x = x;
    y = second;
  }

  public int getX()
  {
    return x;
  }

  public int getY()
  {
    return y;
  }

  public double distance(Point other)
  {
    double dx = x - other.x;
    double dy = y - other.y;
    return Math.sqrt(dx * dx + dy * dy);
  }
}
```

Instance variables are normally private. Instance variables differ from ***local variables*** because they continue to exist between method calls.

`this` keyword distinguishes between ***local variable*** (parameter) **`x`** and ***instance variable*** **`x`**.

Accessor ("***getter***") methods. Classes often have ***mutator*** ("***setter***") methods too, but neither is required.

Methods of **`Point`** class always have access to the private data and methods of **`Point`** instances (but no other objects do).

`other`, **`dx`**, and **`dy`** are ***local variables*** – they only exist for the duration of the method call.

Given the definition of a class, we can create objects that are ***instances*** of that type. In Java this is done with the new keyword, which returns a ***reference*** to the new object:

```
Point p = new Point(3, 7);
```

Creating a new instance is also called ***instantiation***. To instantiate an object we must invoke a ***constructor***, whose role is to establish the initial values of the instance variables.

Inheritance

Java offers two ways to implement inheritance:

- interface implementation, and
- class extension.

We will illustrate these concepts next time using the `Insect` class[1], which is posted on Blackboard.

[1] A version of the `Insect` class was used in a ComS 228 exam in a previous semester,