CS 228: Introduction to Data Structures
Lecture 2
Wednesday, January 14, 2014

Modularity and Abstraction

Recall our example from last lecture.

```
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.
}
```

The interface defines an ***abstract data type*** (ADT); that is, it defines

- a ***collection of things*** (integers) and
- the ***behavior*** of the ***operations*** on the collection,

without specifying ***how*** the collection is going to be implemented.

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 details that aren't relevant to our particular concerns. 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.

Note. Of course, we can't program entirely in the abstract. To actually *use* the `IntCollection` interface, we need an implementation of the interface. That might look something like this:

```
public class UnsortedIntCollection
  implements IntCollection
{
  private ArrayList<Integer> data;
  // Other instance variables

  public UnsortedIntCollection()
  { // Constructor: TODO }

  void add(int k)
  { // TODO }

  boolean contains(int k)
  { // TODO }

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

But, we're getting ahead of ourselves . . .

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; for instance, `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:

1. The implementation is independent of the 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.

2. Encapsulation prevents us from writing applications that corrupt a module's internal data. In real-world programming, encapsulation reduces debugging time. A lot.

3. Teamwork. Once you've rigorously defined interfaces between modules, each programmer can independently implement a module without having access to the other modules. A large, complex programming project can be broken up into dozens of pieces.

4. Documentation and maintainability. By defining an unambiguous interface, you make it easier for other programmers to fix bugs that arise years after you've left the company. Many bugs are a result of unforeseen interactions between modules. If there's a clear specification of each interface and each module's behavior, bugs are 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.

Interface Implementation

As we saw, a Java ***interface*** — specified via the keyword `interface` — is a set of public methods without bodies. That is, an interface specifies the method names, return

types, and parameter types, and nothing else[1]. An interface is a ***contract*** between module writers, specifying exactly how they will communicate.

Implementing an interface is the simplest form of inheritance. To illustrate this, consider the following simple interface specifying just one method.

```
public interface ISpeaking
{
  void speak();
}
```

Note. All methods in a Java interface are public by default, so the `public` keyword is redundant.

An implementation of this interface could be:

[1] This is not strictly true for the latest release of Java, Java 8, which allows the definition of static and default methods in interfaces. For the time being, we will ignore this issue.

```
public class Bird implements ISpeaking
{
  @Override
  public void speak()
  {
    System.out.println("tweet");
  }
}
```

The `implements` keyword means:

> "I promise to provide an implementation (a method body containing actual code) for each method specified by the `ISpeaking` interface."

We say that `Bird` is a ***subtype*** of `ISpeaking`, and `ISpeaking` is a ***supertype*** of `Bird`.

Note. The "`@Override`" is an ***annotation***. It is not required for the code to compile and run, but it is very useful: it tells the compiler that your *intention* is to implement a method defined in an interface (or, we will see later, a superclass). That way, the compiler can check that you have the method signature correct.

There can be other implementations of the same interface, e.g.,

```
public class Person implements ISpeaking
{
  @Override
  public void speak()
  {
    System.out.println("Hi!");
  }
}
```

Now `Person` is a subtype of `ISpeaking`, and `ISpeaking` is a supertype of `Person`.

Note that you ***can*** declare an object whose type is an interface, so the following is ***legal***:

```
ISpeaking b;   // OK
```

After the above statement, b can reference an object of type `Bird` or `Person`. E.g., you could do:

```
b = new Bird();
```

You ***cannot***, however, instantiate an interface variable, so the following is ***illegal***:

```
ISpeaking b = new ISpeaking();  // NO!!
```

This makes sense, since an interface does not define any constructors.