CSE 12

Basic Data Structures
and
Object Oriented Design

http://ieng6.ucsd.edu/~cs12s
Introduction

• Review course structure and requirements on the course website

• Overview course content

• Introduction to Abstraction, Abstract Data Types, Java Interfaces and Classes…
CSE 12 Topics

- Abstraction and Abstract Data Types
- Application Programmer Interfaces
- Java Classes and Interfaces
- Collections, and the Java Collections Framework
- Java Generics
- Algorithm Time, Space, and Energy Cost Analysis
- Object-oriented Software Design Patterns, including Inheritance, Composition, Adapter, and Iterator patterns
- Software Testing and the JUnit framework
- Arrays, Stacks, Queues, Circular Arrays
- Searching and Sorting Algorithms
- Trees, Heaps, Binary Search Trees, Abstract Syntax Trees
- Hashing and Hash Tables
Abstraction

• Abstraction means:
  • *Hiding irrelevant details to focus on the essential features needed to understand and use a thing*

• Abstraction is an essential tool for managing complexity
  • Designing, implementing, and using complex systems would be impossible without abstraction

• Examples:
  – File deletion using icons
  – The brakes on a car
  – Television remote

• What are some other examples of abstraction?
Data Types and Abstraction

- Specifying a **data type** requires specifying two things:
  - The possible values for instances of the type
  - The operations that can be performed on those instances
- For example, an integer data type specification might be:
  - The possible values are whole numbers in the range -2,147,483,648 through 2,147,483,647
  - The operations are addition, subtraction, multiplication, addition
- But how are those values and operations implemented?
- An **Abstract Data Type** (ADT) specification does not specify implementation details… it abstracts away from them!
ADT’s and Data Structures

• CSE 12 is a course in data structures. Why care about ADT’s?

• A data structure is an object that contains data, and that permits certain operations, such as inserting, deleting, and finding data.

• When specifying a data structure, you should say what data values it can contain, and what its operations do, without specifying unnecessary implementation details.

• But in doing that, you are specifying an ADT!
ADT Users and Implementers

• An ADT specification should clearly state what the possible values are for instances of the type, and what operations can be performed on them.

• The specification is what any client (user) of the ADT must know, in order to use it successfully.

• The specification is also what any implementer (programmer) of the ADT must know, in order to implement it correctly.

• It specifies the interface between users of the ADT, and the internal implementation details of the ADT.
This kind of abstraction is very powerful.

The ADT can be implemented without having to know details of how it will be used (except that it will be used in accordance with its interface specification), and…

The ADT can be used without having to know the details of how it is implemented (except that it was implemented to satisfy its interface specification)

With any specification of an ADT interface:
  • The ADT can be used infinitely many ways, and…
  • The ADT can be implemented infinitely many ways!

This makes designing, implementing, using and maintaining software easier and more flexible than it would otherwise be
Example: a String ADT

- You might define a String abstract data type along these lines:

- Values: a String is a sequence of zero or more Unicode 1.0 characters

- Operations:
  - Create a String
  - Add a character to the end of a String
  - Say what character is at a particular position in a String
  - ...

- What other operations might be good to specify for a String ADT?
ADT’s are language-neutral

- An Abstract Data Type is... abstract
- An ADT does not specify how the data type will be implemented...
- ... and in fact does not even specify what programming language will be used to implement it!

- In CSE 12 we will concentrate on implementing ADT’s in Java
- And we will consider many features of Java that are useful for implementing ADT’s
- But always keep in mind that the basic principles of ADT design and analysis are language-neutral!
Application Programmer Interface (API)

- The *implementer* of an ADT in a particular language is a programmer.

- And, usually, the *client* or *user* of an ADT is also a programmer in that language, who wants to use the ADT as a component in a larger software application.

- Therefore the interface to an ADT in a particular language is said to be the **Application Programmer Interface**, or **API**, for the ADT in that language.
Checkpoint

• Abstraction is useful, but...

• Are there times when it would be useful to know the implementation details of an ADT's operations or values?
Abstraction by interface

The Application Programmer Interface (API) of List specifies everything an application programmer (client, user) needs to know in order to use a List.
Any client (user) can make use of a List through its public Application Programmer Interface (API) without knowledge of how the List is implemented.
There are many possible implementations of any ADT. The implementation details are hidden from the user of the ADT, who can rely just on the ADT’s API.
User-defined data types in Java

- Java has 8 **primitive types**: byte, short, int, long, char, float, double, boolean

- And those are all there are: Java does not allow additional user-defined primitive types

- If you want to define a type in Java, you must define either an **interface**, or a **class**

- Often we will use both interfaces and classes when implementing an ADT in Java
ADT’s and Java classes

• An ADT specifies the possible values that instances of the ADT can have, and the operations that can be performed on them

• A Java class defines instance variables, and instance methods

• So, there is a very close relationship between an ADT and a class!

  – Variables defined in the class correspond to ADT values
  – Methods defined in the class correspond to ADT operations
ADT’s and Java interfaces

• Usually, instance variables should be *private* in a class
  • They are considered part of the implementation, not to be accessed directly from outside the class

• Then, we write *public* instance methods to manipulate the instance variables
  • *Mutator* methods change the values of instance variables; *Accessor* methods just ‘read out’ the values
  • This permits precise control over how the instance variables can be accessed and changed

• So, the principle of abstraction suggests we can concentrate on the public instance methods

• Java interfaces do that: they define a Java type, but specify only public instance method signatures (no method bodies, no instance variables)
Next time

• Abstraction, continued
• The Inheritance and Composition Patterns
• Intro to UML
• Intro to the Java Collections Framework (JCF)
• Intro to Unit Testing with JUnit

Reading: Gray, Ch 1 and Ch 2