COMP2160/2860 - Data Structures

The University of Sydney
School of Information Technologies

Week 1:
- Introduction
- Abstract data types

Welcome to COMP2160

- Dr Michael Charleston (Unit coordinator)
  Rm 412
  mcharleston@it.usyd.edu.au
- Dr Tasos Viglas
  Rm 411
  tasos@it.usyd.edu.au

Lectures

- Tuesdays 15:00-17:00,
  Carslaw Lecture Theatre 373
- Labs and tutorials on Wednesdays and Thursdays (see unit website for details)
- COMP2860:
  Advanced tutorials

Announcements

- Website
  http://www.it.usyd.edu.au/~comp2160
- Also accessible from webCT
- All tutorials start next week
  - School of IT Lab 115
    - 1 hour
- Tutors
  - Enoch Lau (Wednesdays)
  - Timothy De Vries (Thursdays)

Textbook:

Data Abstraction & Problem Solving with Java
2e

COMP2860 (Advanced)

- Same textbook
- Extra material for the advanced section (advanced tutorials)
Assessment

<table>
<thead>
<tr>
<th>Assessment</th>
<th>% of final mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 quizzes</td>
<td>2 x 10 = 20</td>
</tr>
<tr>
<td>Assignment 1</td>
<td>10</td>
</tr>
<tr>
<td>Assignment 2</td>
<td>10</td>
</tr>
<tr>
<td>Final Exam</td>
<td>60</td>
</tr>
</tbody>
</table>

Marks

- Quizzes are during lecture times
- Quizzes will focus on theoretical aspects
- Assignments will be more practical
- Final exam mark must be above 40% to pass this course
- Final mark may undergo scaling

COMP2160 ...

- is not about Java
- is not only a programming course
- is not “Problem Based Learning”
- includes both theoretical and practical work
- involves programming in Java

COMP2160 is good for you

- Toolbox for solving complex problems
- Prevent re-inventing the wheel
- Efficient implementation of algorithms
- Building blocks of programs
- Better understanding of programs
- Modular solutions

What is a data structure?

- A table of data including structural relationships
  - Donald Knuth (Turing award ’74)
- Algorithms + data structures = programs
  - Niklaus Wirth (Turing award ’84)

Data structures

- Programs use data
- Data needs to be stored and accessed
- Efficiently
- Different applications have different requirements
Course contents (short)
- List structures, stacks, queues
- Trees
- Sorting
- Heaps, priority queues
- Hashing
- Graphs
- Introduction to algorithms

Course contents (long)
- List structures
- Stacks
- Queues
- Implementation issues
  - Arrays, linked lists
  - Doubly linked lists
  - Dynamic or static
- Recursion

Course contents (long)
- Trees
- Binary trees
- Balanced trees
  - 2-3 trees, red-black
- Binary search trees
- Recursion

Course contents (long)
- Tables
  - Items with a search key
- Priority queues
- Heaps and heapsort
- Hashing
- Hash functions and hash tables

Course contents (long)
- Sorting
  - Selection and insertion sort
  - Bubblesort
  - Mergesort
  - Quicksort
  - Heapsort
  - Comparison of sorting algorithms

Course contents (long)
- Graphs
- Representations
- Graph traversals
- Topological sorting
- Spanning trees
- Shortest paths
Course contents (long)

- Advanced Data structures
- Binomial queues
- Treaps
- Disjoint sets/union find
- Fibonacci heaps
- Analysis of data structures

What is Problem Solving?

- Problem solving
  - The process of taking the statement of a problem and developing a computer program that solves that problem
- A solution consists of:
  - Algorithms
    - Algorithm: a step-by-step specification of a method to solve a problem within a finite amount of time
  - Ways to store data

What is a Good Solution?

- A solution is good if:
  - The total cost it incurs over all phases is minimal
    - Specification, Design, Verification
    - Coding, testing, refining
    - Maintenance
- The cost of a solution includes:
  - Computer resources that the program consumes
  - Difficulties encountered by those who use the program
  - Consequences of a program that does not behave correctly
- Programs must be well structured and documented
- Efficiency is only one aspect of a solution’s cost

Achieving a Modular Design: Abstraction and Information Hiding

- A modular solution to a problem should specify what to do, not how to do it
- Abstraction
  - Separates the purpose of a module from its implementation
- Procedural abstraction
  - Separates the purpose of a method from its implementation

Abstraction and Information Hiding

- Data abstraction
  - Focuses on the operations of data, not on the implementation of the operations
  - Abstract data type (ADT)
    - A collection of data and a set of operations on the data
    - An ADT’s operations can be used without knowing how the operations are implemented, if:
      - the operations’ specifications are known
  - Data structure
    - A construct that can be defined within a programming language to store a collection of data
Abstraction and Information Hiding

- Public view of a module
  - Described by its specifications
- Private view of a module
  - Consists of details which should not be described by the specifications
- Principle of information hiding
  - Hide details within a module
  - Ensure that no other module can tamper with these hidden details

A simple example

keyboard → key buffer → Operating System

Press a key: key code is written in the buffer
Operating system reads keys First In First Out (FIFO)

How do we implement the buffer?
What does the buffer need to do?
Two things:
- add keys to the queue or en-q
- read keys from the queue or de-q

Buffer example

key buffer

Need a data structure that supports two operations: enq and deq.
The actual implementation is not part of this abstract description, or Abstract Data Type (ADT)

Any implementation can be used
- Array with two pointers
- Linked list

Abstract Data Types

- Abstract description of a data structure
- Describe what the data structure does
  - not how it does it
- How to use the data structure
  - interface
- Properties of the data structure
  - example: operation en-queue must be O(log n)
  - buffer example: both operations O(1)

Object-Oriented Design

- Principles of object-oriented programming (OOP)
  - Encapsulation
  - Objects combine data and operations
  - Inheritance
  - Classes can inherit properties from other classes
  - Polymorphism
    - Objects can determine appropriate operations at execution time

Top-Down Design

- Object-oriented design (OOD)
  - Produces modular solutions for problems that primarily involve data
- Top-down design (TDD)
  - Produces modular solutions for problems in which the emphasis is on the algorithms
  - A task is addressed at successively lower levels of detail
Top-Down Design

General Design Guidelines

- Use Object Oriented and Top Down Design
  - Use Object Oriented Design for problems that primarily involve data
  - Use Top Down Design to design algorithms for an object's operations
- Focus on what, not how, when designing both ADTs and algorithms
- Consider incorporating previously written software components into your design

A Summary of Key Issues in Programming

- Modularity
  - Has a favorable impact on:
    - Constructing the program
    - Debugging the program
    - Reading the program
    - Modifying the program
    - Eliminating redundant code
- Modifiability
  - Ways to make a program easy to modify:
    - Use of methods
    - Named constants

A Summary of Key Issues in Programming

- Ease of Use
  - Considerations for the user interface
    - In an interactive environment, the program should prompt the user for input in a clear manner
    - A program should always echo its input
    - The output should be well labeled and easy to read

A Summary of Key Issues in Programming

- Fail-Safe Programming
  - Fail-safe program
    - A program that will perform reasonably no matter how anyone uses it
  - Types of errors:
    - Errors in input data
    - Errors in the program logic

A Summary of Key Issues in Programming

- Style
  - Five issues of style
    - Extensive use of methods
    - Use of private data fields
    - Error handling
    - Readability
    - Documentation
A Summary of Key Issues in Programming

- **Debugging**
  - Programmer must systematically check a program’s logic to determine where an error occurs
  - Tools to use while debugging:
    - Watches
    - Breakpoints
    - `System.out.println` statements
    - Dump methods

**Summary**

- Software engineering studies ways to facilitate the development of computer programs
- Software life cycle consists of:
  - Specifying the problem
  - Designing the algorithm
  - Analyzing the risks
  - Verifying the algorithm
  - Coding the programs
  - Testing the programs
  - Refining the solution
  - Using the solution
  - Maintaining the software

**Summary**

- An evaluation of the quality of a solution must take into consideration
  - The solution’s correctness
  - The solution’s efficiency
  - The time that went into the development of the solution
  - The solution’s ease of use
  - The cost of modifying and expanding the solution

**Summary**

- A combination of object-oriented and top-down design techniques will lead to a modular solution
- The final solution should be as easy to modify as possible
- A method should be as independent as possible and perform one well-defined task
- A method should always include an initial comment that states its purpose, its precondition, and its postcondition

**Summary**

- A program should be as fail-safe as possible
- Effective use of available diagnostic aids is one of the keys to debugging
- To make it easier to examine the contents of complex data structures while debugging, dump methods that display the contents of the data structures should be used

**Abstract Data Types**

- **Modularity**
  - Keeps the complexity of a large program manageable by systematically controlling the interaction of its components
  - Isolates errors
  - Eliminates redundancies
  - A modular program is
    - Easier to write
    - Easier to read
    - Easier to modify
Abstract Data Types

- The isolation of modules is not total
  - Methods’ specifications, or contracts, govern how they interact with each other

- Typical operations on data
  - Add data to a data collection
  - Remove data from a data collection
  - Ask questions about the data in a data collection

- Data abstraction
  - Asks you to think what you can do to a collection of data independently of how you do it
  - Allows you to develop each data structure in relative isolation from the rest of the solution
  - A natural extension of procedural abstraction

- Abstract data type (ADT)
  - An ADT is composed of
    - A collection of data
    - A set of operations on that data
  - Specifications of an ADT indicate
    - What the ADT operations do, not how to implement them
  - Implementation of an ADT
    - Includes choosing a particular data structure

- Data structure
  - A construct that is defined within a programming language to store a collection of data
  - Example: arrays

- ADTs and data structures are not the same

- Data abstraction
  - Results in a wall of ADT operations between data structures and the program that accesses the data within these data structures

- A wall of ADT operations isolates a data structure from the program that uses it
Specifying ADTs

- In a list
  - Except for the first and last items, each item has
    - A unique predecessor
    - A unique successor
  - Head or front
    - Does not have a predecessor
  - Tail or end
    - Does not have a successor

Figure 3.5
list A grocery

The ADT List

- ADT List operations
  - Create an empty list
  - Determine whether a list is empty
  - Determine the number of items in a list
  - Add an item at a given position in the list
  - Remove the item at a given position in the list
  - Remove all the items from the list
  - Retrieve (get) the item at a given position in the list
  - Items are referenced by their position within the list

The ADT List

- ADT List operations – needed?
  - Remove all the items from the list

The ADT List

- ADT List operations – all needed?
  - Remove all the items from the list
  - Determine number of items $n$ in the list
  - for $i=1$ to $n$ remove item at position $i$

The ADT List

- Specifications of the ADT operations
  - Define the functionality of the ADT list
  - Do not specify how to store the list or how to perform the operations
  - ADT operations can be used in an application without the knowledge of how the operations will be implemented

Figure 3.6
The wall between displayList and the implementation of the ADT list
The ADT Sorted List

- The ADT sorted list
  - Maintains items in sorted order
  - Inserts and deletes items by their values, not their positions

Designing an ADT

- The design of an ADT should evolve naturally during the problem-solving process
- Questions to ask when designing an ADT
  - What data does a problem require?
  - What operations does a problem require?

Axioms

- For complex abstract data types, the behavior of the operations must be specified using axioms
  - Axiom: A mathematical rule

Axioms

- Axioms for the ADT List
  - (aList.createList()).size() = 0
  - (aList.add(i, x)).size() = aList.size() + 1
  - (aList.remove(i)).size() = aList.size() - 1
  - (aList.createList()).isEmpty() = true
  - (aList.add(i, x)).isEmpty() = false
  - (aList.createList()).remove(i) = error
  - (aList.add(i, x)).remove(i) = aList
  - (aList.createList()).get(i) = error
  - (aList.add(i, x)).get(i) = x
  - aList.get(i) = (aList.add(i, x)).get(i+1)
  - aList.get(i+1) = (aList.remove(i)).get(i)

Implementing ADTs

- Choosing the data structure to represent the ADT's data is a part of implementation
  - Choice of a data structure depends on
    - Details of the ADT's operations
    - Context in which the operations will be used
  - Implementation details should be hidden behind a wall of ADT operations
    - A program would only be able to access the data structure using the ADT operations

Implementing ADTs

- Figure 3.7
  - ADT operations provide access to a data structure
Implementing ADTs

Java Classes
• Object-oriented programming (OOP) views a program as a collection of objects
• Encapsulation
  – A principle of OOP
  – Can be used to enforce the walls of an ADT
  – Combines an ADT’s data with its method to form an object
  – Hides the implementation details of the ADT from the programmer who uses it

Java Classes
• A Java class
  – A new data type whose instances are objects
  – Class members
    • Data fields
      – Should almost always be private
    • Methods
      – All members in a class are private, unless the programmer designates them as public

Java Classes
• A Java class (Continued)
  – Constructor
    • A method that creates and initializes new instances of a class
    • Has the same name as the class
    • Has no return type
  – Java’s garbage collection mechanism
    • Destroys objects that a program no longer references

Java Classes
• Constructors
  – Allocate memory for an object and can initialize the object’s data
  – A class can have more than one constructor
  – Default constructor
    • Has no parameters
    • Typically, initializes data fields to values the class implementation chooses
Java Classes

• Constructors (Continued)
  – Compiler-generated default constructor
    • Generated by the compiler if no constructor is included in a class
  – Client of a class
    • A program or module that uses the class

Java Classes

• Inheritance
  – Base class or superclass
  – Derived class or subclass
    • Inherits the contents of the superclass
    • Includes an extends clause that indicates the superclass
    • super keyword
      – Used in a constructor of a subclass to call the constructor of the superclass

Java Classes

• Object Equality
  – equals method of the Object class
    • Default implementation
      – Compares two objects and returns true if they are actually the same object
    • Customized implementation for a class
      – Can be used to check the values contained in two objects for equality

Java Interfaces

• An interface
  – Specifies methods and constants, but supplies no implementation details
  – Can be used to specify some desired common behavior that may be useful over many different types of objects
  – The Java API has many predefined interfaces
    • Example: java.util.Collection

Java Interfaces

• A class that implements an interface must
  – Include an implements clause
    – Provide implementations of the methods of the interface
  – To define an interface
    – Use the keyword interface instead of class in the header
    – Provide only method specifications and constants in the interface definition

Java Exceptions

• Exception
  – A mechanism for handling an error during execution
  – A method indicates that an error has occurred by throwing an exception
Java Classes

• A Java class
  – A new data type whose instances are objects
  – Class members
    • Data fields
      – Should almost always be private
    • Methods
  – All members in a class are private, unless the programmer designates them as public

Java Classes

• A Java class (Continued)
  – Constructor
    • A method that creates and initializes new instances of a class
    • Has the same name as the class
    • Has no return type
  – Java’s garbage collection mechanism
    • Destroys objects that a program no longer references

Java Classes

• Constructors
  – Allocate memory for an object and can initialize the object’s data
  – A class can have more than one constructor
  – Default constructor
    • Has no parameters
    • Typically, initializes data fields to values the class implementation chooses

Java Classes

• Constructors (Continued)
  – Compiler-generated default constructor
    • Generated by the compiler if no constructor is included in a class
  – Client of a class
    • A program or module that uses the class

Java Classes

• Inheritance
  – Base class or superclass
  – Derived class or subclass
    • Inherits the contents of the superclass
    • Includes an `extends` clause that indicates the superclass
    • `super` keyword
      – Used in a constructor of a subclass to call the constructor of the superclass

Java Classes

• Object Equality
  – `equals` method of the Object class
    • Default implementation
      – Compares two objects and returns true if they are actually the same object
    • Customized implementation for a class
      – Can be used to check the values contained in two objects for equality
Java Interfaces

- An interface
  - Specifies methods and constants, but supplies no implementation details
  - Can be used to specify some desired common behavior that may be useful over many different types of objects
  - The Java API has many predefined interfaces
    - Example: `java.util.Collection`

Java Interfaces

- A class that implements an interface must
  - Include an `implements` clause
  - Provide implementations of the methods of the interface
- To define an interface
  - Use the keyword `interface` instead of `class` in the header
  - Provide only method specifications and constants in the interface definition

Java Exceptions

- Exception
  - A mechanism for handling an error during execution
  - A method indicates that an error has occurred by throwing an exception

Java Exceptions

- Catching exceptions
  - `try` block
  - A statement that might throw an exception is placed within a `try` block
  - Syntax
    ```java
    try {
    statement(s);
    } // end try
    ```

Java Exceptions

- Types of exceptions
  - Checked exceptions
    - Instances of classes that are subclasses of the `java.lang.Exception` class
    - Must be handled locally or explicitly thrown from the method
    - Used in situations where the method has encountered a serious problem
Java Exceptions

• Types of exceptions (Continued)
  – Runtime exceptions
    • Used in situations where the error is not considered as serious
    • Can often be prevented by fail-safe programming
    • Instances of classes that are subclasses of the RuntimeException class
    • Are not required to be caught locally or explicitly thrown again by the method

Java Exceptions

• Throwing exceptions
  – A throw statement is used to throw an exception
    throw new exceptionClass (stringArgument);

• Defining a new exception class
  – A programmer can define a new exception class

An Array-Based Implementation of the ADT List

• An array-based implementation
  – A list’s items are stored in an array items
  – A natural choice
    • Both an array and a list identify their items by number
    • A list’s kth item will be stored in items[k-1]

List implementations

• Options for implementing an ADT
  – Array
    • Has a fixed size
    • Data must be shifted during insertions and deletions
  – Linked list
    • Is able to grow in size as needed
    • Does not require the shifting of items during insertions and deletions
Preliminaries

Object References

- A reference variable
  - Contains the location of an object
  - Example
    ```java
    Integer intRef;
    intRef = new Integer(5);
    ```
  - As a data field of a class
    - Has the default value `null`
  - A local reference variable to a method
    - Does not have a default value

Object References

- When one reference variable is assigned to another reference variable, both references then refer to the same object
  ```java
  Integer p, q;
  p = new Integer(6);
  q = p;
  ```

- A reference variable that no longer references any object is marked for garbage collection

Object References

- When one reference variable is assigned to another reference variable, both references then refer to the same object
  ```java
  Integer p, q;
  p = new Integer(6);
  q = p;
  ```

- A reference variable that no longer references any object is marked for garbage collection
Object References

- An array of objects
  - Is actually an array of references to the objects
  - Example
    ```java
    Integer[] scores = new Integer[30];
    ```
  - Instantiating Integer objects for each array reference
    ```java
    scores[0] = new Integer(7);
    scores[1] = new Integer(9); // and so on ...
    ```

Object References

- Equality operators (== and !=)
  - Compare the values of the reference variables, not the objects that they reference
- `equals` method
  - Compares objects field by field
- When an object is passed to a method as an argument, the reference to the object is copied to the method's formal parameter
- Reference-based ADT implementations and data structures use Java references

Resizable Arrays

- The number of references in a Java array is of fixed size
- Resizable array
  - An array that grows and shrinks as the program executes
  - An illusion that is created by using an allocate and copy strategy with fixed-size arrays
- `java.util.Vector` class
  - Uses a similar technique to implement a growable array of objects

Recursive Solutions

- Recursion
  - An extremely powerful problem-solving technique
  - Breaks a problem in smaller sub-problems
  - An alternative to iteration
    - An iterative solution involves loops

Recursive Solutions

- Facts about a recursive solution
  - A recursive method calls itself
  - Each recursive call solves an identical, but smaller, problem
  - A test for the base case enables the recursive calls to stop
    - Base case: a known case in a recursive definition
  - Eventually, one of the smaller problems must be the base case

Recursive Solutions

- Four questions for constructing recursive solutions
  - How can you define the problem in terms of a smaller problem of the same type?
  - How does each recursive call diminish the size of the problem?
  - What instance of the problem can serve as the base case?
  - As the problem size diminishes, will you reach this base case?
A Recursive Valued Method: The Factorial of n

- Problem
  - Compute the factorial of an integer $n$
- An iterative definition of factorial(n)
  \[
  \text{factorial}(n) = n \times (n-1) \times (n-2) \times \ldots \times 1 \quad \text{for any integer } n > 0
  \]
  \[
  \text{factorial}(0) = 1
  \]

A Recursive Valued Method: The Factorial of n

- A recursive definition of factorial(n)
  \[
  \text{factorial}(n) = \begin{cases} 
  1 & \text{if } n = 0 \\
  n \times \text{factorial}(n-1) & \text{if } n > 0
  \end{cases}
  \]
- A recurrence relation
  - A mathematical formula that generates the terms in a sequence from previous terms
  - Example
  \[
  \text{factorial}(n) = n \times [(n-1) \times (n-2) \times \ldots \times 1] \\
  = n \times \text{factorial}(n-1)
  \]

A Recursive Valued Method: The Factorial of n

- Box trace
  - A systematic way to trace the actions of a recursive method
  - Each box roughly corresponds to an activation record
  - An activation record
    - Contains a method’s local environment at the time of and as a result of the call to the method

Function calls

- Function call
  - save the “environment” -- PUSH operation
  - execute the function
  - restore the environment -- POP operation
  - continue execution
- We need a stack for nested function calls

Factorial example

\[
\text{fact}(n) : \\
\text{if } n=1 \text{ return } 1 \\
\text{return } \text{fact}(n-1) \times n
\]

- The ‘environment’ here is just ‘n’
- Execute fact(3)
Factorial example

```
fact(3):
  if (n=1) return 1
  return fact(2) * 3
```

Function叫fact(n=3):
- Save environment on the stack

```
fact(n=3):
  if (n=1) return 1
  return fact(n=2) * n
```

Function call

```
fact(n)
n=3
```

```
if (n=1) return 1
return fact(n=0) * n
```

```
fact(n)
n=2
```

```
if (n=1) return 1
return fact(n=0) * n
```

```
fact(n)
n=3
```

```
if (n=1) return 1
return fact(n=0) * n
```
A Recursive void Method:
Writing a String Backward

• Problem
  – Given a string of characters, write it in reverse order

• Recursive solution
  – Each recursive step of the solution diminishes by 1 the length of the string to be written backward
  – Base case
    • Write the empty string backward

writeBackward(s)

writeBackward(s minus last character)
A Recursive void Method: Writing a String Backward

- Execution of `writeBackward` can be traced using the box trace
- Temporary `System.out.println` statements can be used to debug a recursive method

```
void writeBackward(String s)
{
    if (s is empty)
        do nothing
    else
        write the last character of s
        writeBackward(s minus its last character)
}
```

Multiplying Rabbits (The Fibonacci Sequence)

- “Facts” about rabbits
  - Rabbits never die
  - A rabbit reaches sexual maturity exactly two months after birth, that is, at the beginning of its third month of life
  - Rabbits are always born in male-female pairs
    - At the beginning of every month, each sexually mature male-female pair gives birth to exactly one male-female pair

```
rabbit(n) =
  1 if n is 1 or 2
  rabbit(n-1) + rabbit(n-2) if n > 2
```

Multiplying Rabbits (The Fibonacci Sequence)

- Problem
  - How many pairs of rabbits are alive in month n?
- Recurrence relation
  - `rabbit(n) = rabbit(n-1) + rabbit(n-2)`
Searching an Array:
Finding the Largest Item in an Array

- A recursive solution
  ```java
  if (anArray has only one item) {
    maxArray(anArray) is the item in anArray
  } else if (anArray has more than one item) {
    maxArray(anArray) is the maximum of
    maxArray(left half of anArray) and
    maxArray(right half of anArray)
  } // end if
  ```

Finding the Largest Item in an Array

Figure 2.13
Recursive solution to the largest-item problem

Binary Search

- A high-level binary search
- Looking for a value 'x' in a sorted array

1 2 3 4 5 6 7 8 9 10

- x < a ? Continue on left half else right

Binary Search

- Implementation issues:
  - How will you pass "half of anArray" to the
    recursive calls to binarySearch?
  - How do you determine which half of the
    array contains the value?
  - What should the base case(s) be?
  - How will binarySearch indicate the
    result of the search?

Binary Search

• Searching a number in a list of length n
• How many steps? (worst case)
  - "Step" is "list item access"
  - \( T(n) = ? \)
  - \( T(1) = 1 \)
Binary Search

- Searching a number in a list of length n
- How many steps? (worst case)
- “Step” is “list item access”
- \( T(n) = 1 + T(n/2) \)
  \( = 1 + 1 + T((n/2)/2) \)
  \( = 2 + T(n/4) \)
  \( = \ldots \) (repeat k times)
  \( = k + T(n/2^k) \)

- \( T(1) = 1 \)
- \( T(n) = k + T(n/2^k) \)
- Stop when: \( n/2^k = 1 \) or \( k = \log n \)
- \( T(n) = O(\log n) \)

The Towers of Hanoi

- Pseudocode solution

  ```java
  solveTowers(count, source, destination, spare)
  if (count is 1) {
    Move a disk directly from source to destination
  } else {
    solveTowers(count-1, source, spare, destination)
    solveTowers(1, source, destination, spare)
    solveTowers(count-1, spare, destination, source)
  } //end if
  ```

Recursion and Efficiency

- Some recursive solutions are so inefficient that they should not be used
- Factors that contribute to the inefficiency of some recursive solutions
  - Overhead associated with method calls
  - Inherent inefficiency of some recursive algorithms
Summary

- Data abstraction: a technique for controlling the interaction between a program and its data structures
- An ADT: the specifications of a set of data management operations and the data values upon which they operate
- The formal mathematical study of ADTs uses systems of axioms to specify the behavior of ADT operations
- Only after you have fully defined an ADT should you think about how to implement it

A client should only be able to access the data structure by using the ADT operations
- An object encapsulates both data and operations on that data
  - In Java, objects are instances of a class, which is a programmer-defined data type
- A Java class contains at least one constructor, which is an initialization method
- Typically, you should make the data fields of a class private and provide public methods to access some or all of the data fields

Done

- This lecture: lists, recursion
- Next week: stacks and queues
- Next week tutorial: implementation issues for stacks and queues
- Textbook: p113-170 (recursion) p170-250 (lists)