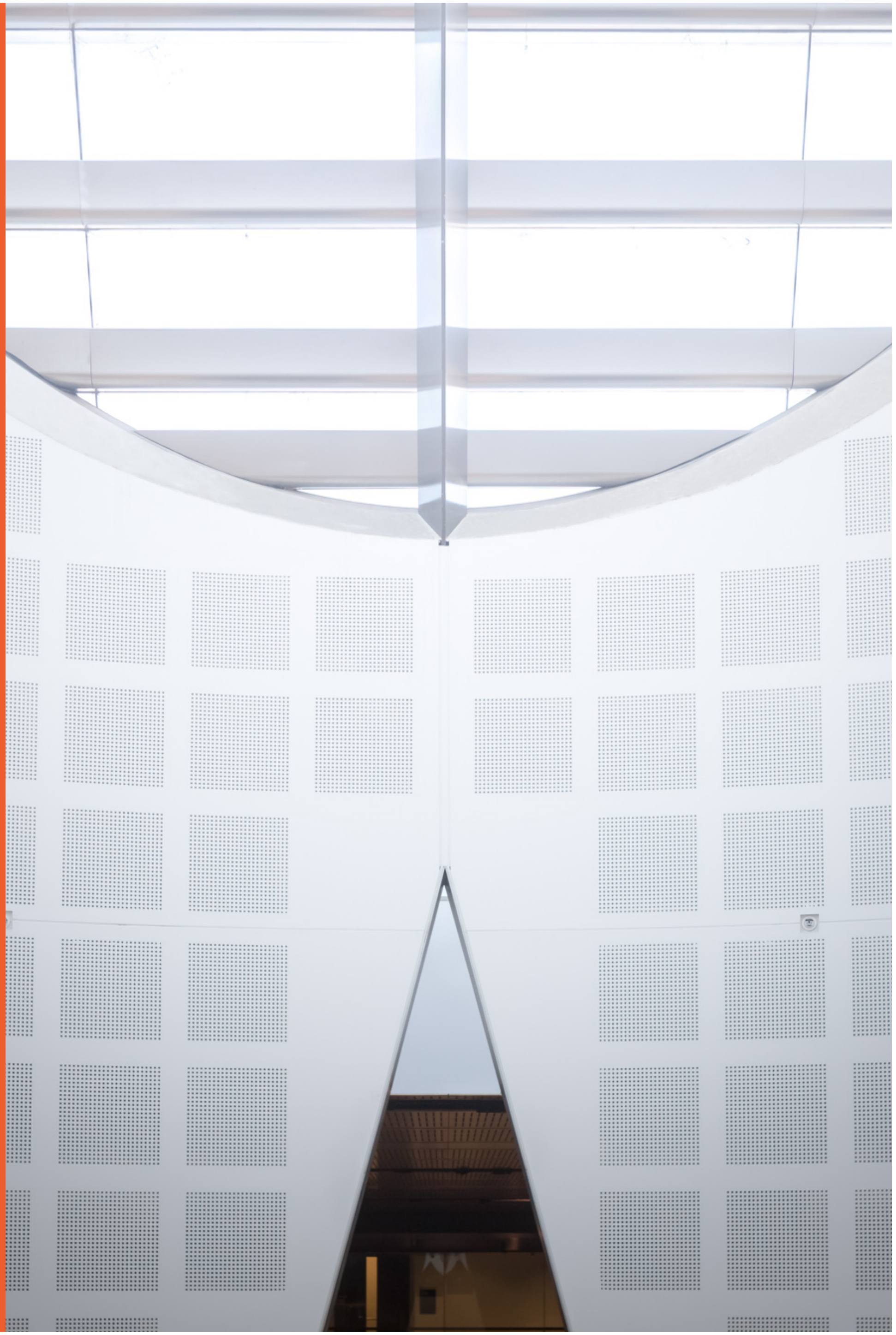
INFO1105/1905/9105
Data Structures
Week 1b: Introduction

Professor Alan Fekete
Professor Seokhee Hong
School of Information Technologies

Some content is due to A/Prof Yacef or A/Prof Charleston (now at UTas), and some is taken from material provided by the textbook publisher J. Wiley

THE UNIVERSITY OF SYDNEY



Copyright warning

Agenda

- Data Structures: introduction and big ideas
- Review of Java programming foundations
- Review of recursion
- Week 1 lab

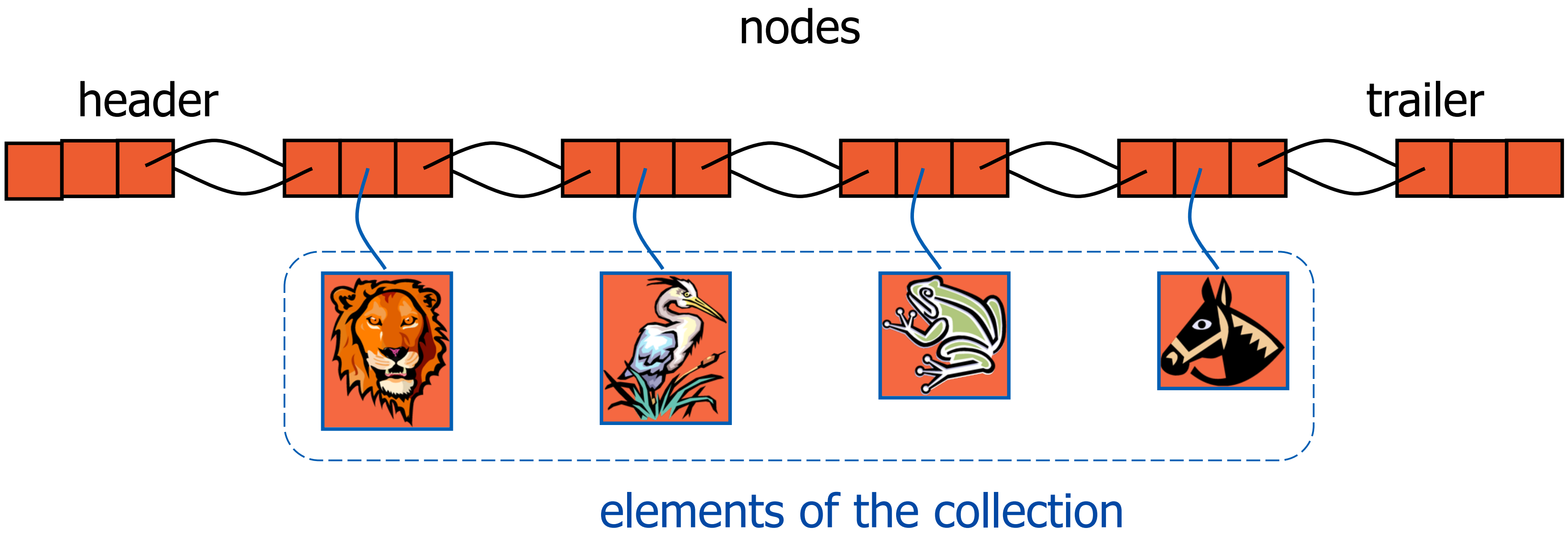
What is a data structure?

- "a **data structure** is a particular way of organizing data in a computer so that it can be used efficiently" [Wikipedia]
- "An organization of information, usually in memory, for better algorithm efficiency" [NIST, Dictionary of Algorithms and Data Structures]

- A data structure is typically used to keep several items that one wants to treat as a collection
  - eg all the students in info1105
  - eg the map of road routes in Sydney
- The vital issue is how data elements are arranged and connected with one another
  - usually done with inter-object references (that is, instance variable in one object whose value is another object)

Example data structure: Doubly Linked List (week 2)

A sequence of Nodes, each with reference to prev and to next; special nodes are used for the "header" and "trailer"; Each Node, except header and trailer, contains a reference to one item that is a member of the collection



Big ideas: operations of a data structure

- For a given data structure, we can write code that performs methods that apply to the collection as a whole
  - these usually traverse the structure (move from one object to another, following references), and/or manipulate the structure (change the value of some references, to have different values)
- The details of the data structure require careful choice of the algorithm for each operation
  - Keep the data structure together, and properly formed
  - eg how would we insert another element somewhere in the middle of the collection?

Uses of data structures in industry practice

- Facebook Platform
  - Graph (textbook chapter 14) represents relationships between nodes
- Google Maps
  - Map locations are nodes of a weighted graph (textbook chapter 14)
- Bitcoin
  - Bloom filter uses hashing (textbook section 10.2) to verify payments
- TeraSort with Hadoop
  - Trie (textbook section 13.3) for speedy sort

Big ideas: analysis of runtime

- The particular choice of data structure and algorithms for the operations has a huge influence on how quickly the program runs
  - especially, on how the running time grows, as we have more and more data items in the collection
- There is a mathematical language that we can use to describe how runtime (or other measurements) grow as a feature increases
  - “big-Oh” notation
  - allows to distinguish constant time, logarithmic time, linear time, quadratic time, exponential time algorithms
- There is mathematical theory that allows us to reason, to work out what growth a particular algorithm has, based on knowledge of how the algorithm works

Big ideas: Abstract Data Types (ADTs)

- A model for what behaviour a data structure should have
  - This is what a client or user needs to know
  - Based on the abstract content of the collection, not its arrangement as a data structure
    - eg what elements are there, and perhaps in what order, but not how they are connected by references
- What operations are provided (including what arguments each takes)?
- How is the result of each operation described (in terms of the abstract content, and its changes)?

- There can be several different data structures that provide the exact same ADT!

Benefits of ADT approach

- Code is easier to understand if different issues are separated into different places
- Client can be considered at a higher, more abstract, level
- Many different systems can use the same library
  - only code (and test!) tricky manipulations once, rather than in every client system
- There can be choices of implementations with different performance tradeoffs, and the client doesn’t need to be rewritten extensively to change which implementation it uses
- This provides good *modularity* and low *coupling* in the overall software design

Data structures in Java

- The Java collections library (in java.util.*) has many examples of data structures that are useful in programming
  - LinkedList
  - ArrayList
  - HashMap
  - TreeMap
- The ADTs are provided as interfaces
  - List
  - Map
  - Set
  - etc

- In this unit, we will focus more on simpler, textbook-specific collection classes and interfaces
  - similar ideas to the standard library, and the same underlying principles, but usually fewer methods supported (and often fewer exceptions too)
  - Understanding how these classes are written can help you understand when to what is happening in the real libraries, and especially how to choose between alternatives

- The skills can be useful if in future you want to produce your own collection, for a special purpose
  - Even offer it to others as a library

Java Topics to review (from info1103)

- Java syntax
  - variables and types
  - assignment and expressions
  - control flow
- Objects and classes
- Inheritance
- Interfaces
- Exceptions
- Recursion

See ch 1 and 2 of textbook!!

Java syntax revisited

- Variables must be declared before use
- Declaration says what type the variable has
  - an initial value can also be assigned
- Types include builtin types (boolean, int, float etc) and also object ones defined by class definitions in libraries or elsewhere (eg String, LinkedList)

```
int size;
LinkedList<String> words;
```

- types also on arguments of methods, and for return

```
boolean hasNoVowels (String word) {
    // code here
}
```

Java syntax revisited

- one can assign to a variable, the value of an expression
  - end each assignment statement with semicolon
- the expression can be built by applying operations to constants, the value of other variables, the result of calling methods

```
size = (3 * base) + obj.getMax();
```

Java syntax revisited

- Control flow: execute statements in turn, through the code, except where flow is altered
- if, while, for statements are controled by the value of boolean expressions
- be careful to put control conditions in parentheses
- be careful to use curly braces correctly to determine what happens in each branch or iteration
  - there can be nested control structures

```
if (x<0 || y <0) {
     System.out.println("Some negative inputs");
} else {
     System.out.println("Non-negative inputs");
}
```

Example task

Triangles

This is a classic problem: given a single line of input containing a number, print a triangle of asterisks of that size.

For example, given this input:

```
5
```

Your program should print:

```
*
**
***
****
*****
```

solution in Java

```java
package intro;

import java.util.Scanner;

public class Triangles {

    public static void main(String[] args) {
        int size; // you have to declare variables before you use them!
        System.out.print("How big do you want the triangle? ");
        // reading in from standard input can be fiddly:
        Scanner in = new Scanner(System.in);
        size = in.nextInt();
        for (int i = 1; i <= size; ++i) {
            for (int j = 1; j <= i; j++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

Java Objects and Classes Revisited

- Object-oriented programming (OOP) views a program as a collection of object instances
- Each object instance has a structure and behavior defined by a class

Java Classes Revisited

- A Java class
  - A new data type whose instances are objects
  - Class members
    - Data fields
    - Methods

- Client of a class
  - A program or module that uses the class
    - Has a data field or variable whose type is the class
    - Calls a method of the class

Java Classes Revisited

- Constructor
  - A method that creates and initializes new instances of a class
- Has the same name as the class
- Has no return type (not even `void`)
- Object instance is constructed by a statement such as
  - `Book b = new Book();`
- A class can have more than one constructor by "overloading" – the pattern of argument types decides which is used

- Java's garbage collection mechanism
- Destroys objects that a program no longer references

Java Inheritance Revisited

- Inheritance
  - Base class (or superclass)
  - Derived class (or subclass)
    - Inherits the contents of the superclass
    - Includes an `extends` clause that indicates the superclass
    - The `super` keyword is used in a constructor of a subclass to call the constructor of the superclass

Java Inheritance Revisited

- 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 revisited

– 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 revisited

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

Java Interfaces revisited

```
interface Repairable {
  boolean isWorking();
   // note: no method body
  void repair();
}
```

The interface is defined in a file Repairable.java, just like a class.

Note: interface names often end in “able” since they describe what the type can do

Java Interfaces revisited

```
class WashingMachine implements Repairable {
      public boolean isWorking() {
       // code for this method
      }
      public void repair() {
       // code for this method
      }
      // other methods
      // instance variables

      // constructor
      public WashineMachine() {
        // code for the constructor
      }
}
```

Java Interfaces revisited

An interface can be used as a type; that is, a variable can be declared this way

- but the interface does not have any constructor
- to construct an instance, you need a class that implements the interface
  - `Repairable appliance = new WashingMachine();`
  - `// alternative approach`
    ```
    WashingMachine w = new WashingMachine();
    Repairable appliance = w;
    ```
  - `List<Cow> herd = new LinkedList<Cow>();`

Java Exceptions revisited

– Exception

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

Java Exceptions revisited

- Catching exceptions
  - `try` block
    - A statement that might throw an exception is placed within a `try` block
    - Syntax

```
try {
  statement(s);
} // end try
```

Java Exceptions revisited

– Catching exceptions (Continued)
  – `catch` block
    - Used to catch an exception and deal with the error condition
    - Syntax

```
catch (exceptionClass identifier) {
  statement(s);
} // end catch
```

Java Exceptions revisited

- 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 revisited

- 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 revisited

- 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

Recursion concept revisited

- Recursion: when a method calls itself (with different arguments, or target, etc)
- Classic example: the factorial function
  - n! = 1· 2· 3· ··· · (n-1)· n
  - Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

Java recursion revisited

Base case(s)

- Values of the input variables for which we perform no recursive calls are called base cases
  - there should be at least one base case.
  - Every possible chain of recursive calls must eventually reach a base case.

Recursive calls

- Calls to the same method
  - Each recursive call should be defined so that it makes progress towards a base case.

```java
// recursive factorial function
public static int  factorial(int n) {
     if  (n  ==  0) {
          return  1;  //base case
     } else {
          return  n * factorial(n- 1); //recursive case
     }
}
```

Recursion Example: computing powers

- Consider $p(x,n)=x^n$

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- Class activity: Write this as a (static) Java method
- Extra activity at home: code for a different (much more efficient, for large n) recursive definition of the same mathematical function

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, (n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x, n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

List and Map

- Java Collections Framework is in java.util.*
- it contains very useful interfaces and implementations
- List (implemented by ArrayList, LinkedList and others) keeps an ordered collection that can be traversed
  - for-each loop
- Map (implemented by HashMap) keeps a table or dictionary, that associates pairs of (key,value), with lookup based on just knowing the key
- We will study these ideas (both the ADTs and the data structures to implement them) in considerable depth

Example task

Write a program that reads multiple lines of plain text from standard input (not a file), then prints out each different word in the input with a count of how many times that word occurs. Don't worry about punctuation or case – the input will just be words, all in lower case.

For example, given this input:

```
which witch
is which
```

Your program should print this:

```
is 1
which 2
witch 1
```

```java
package intro;

import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class WordCounter {
    public static void main(String[] args) {
        Map<String, Integer> occurrences = new HashMap<String, Integer>();
        Scanner scanIn = new Scanner(System.in);
        String[] line = (scanIn.nextLine().trim()).split("\\s");
        while (!line[0].equals("")) {
            for (String word : line) {
                if (occurrences.get(word) == null) {
                    occurrences.put(word, 1);
                } else {
                    occurrences.put(word, occurrences.get(word)+1);
                }
            }
            line = (scanIn.nextLine().trim()).split("\\s");
        }
        for (String str : occurrences.keySet()) {
            System.out.println(str + ", " + occurrences.get(str));
        }
    }
}
```

References

- From Goodrich, Tamassia, Goldwasser (6th ed)
  - Chapter 1 and Chapter 2: Review of Java
  - Chapter 5.1 Examples of Recursion

- If you want more material to revise Java, there are many different Java textbooks, including many online! Find the one whose style suits you.

Summary

- Make sure you remember and/or revise Java knowledge
- Labs start this week
  - Week 1 lab is intended as review of Java, introduction to use of Eclipse (for those without previous experience), review of recursion, and review of use of Map and List interfaces
  - Weekly task from week 1 must be submitted through edstem by noon Wednesday August 9
- Check regularly:
  - eLearning site
  - edstem site
  - unit tutorial site