COMP 1020 – Introductory Computer Science 2

Course Description

Calendar entry

(Lab Required) More features of a procedural language, elements of programming. May not be held with COMP 1021. Prerequisite: [One of COMP 1010, COMP 1011, COMP 1012, or COMP 1013] or [Computer Science 40S (75%) and (one of 40S Mathematics (50%), MATH 1018, or MSKL 0100)].

General Course Description

By this point students have learnt to design basic algorithms and write instructions in a procedural language to be executed by a computer. This includes defining and using variables, methods / functions, conditional expressions, iteration via loops, and simple data structures such as arrays.

In this class, we introduce a new programming paradigm (object-oriented programming), reading/writing data to/from files stored on a computer, data structures that use memory references, and using recursion to solve simple problems. We also discuss different algorithms for searching and sorting.

Detailed Prerequisites

Before entering this course, a student should be able to:

- Represent ideas and information in a way that computers can understand and act on.
- Read, write, and run moderately complex programs using a procedural programming language.
- Describe basic programming concepts and structures in plain English.
- Analyze and implement basic algorithms such as searching.

Course Goals

By the end of this course students will:

- Use classes and objects effectively to design and implement structured representation of information.
- Use and implement data structures to solve a problem, with emphasis on arrays and linked lists.
- Use an interface to define and make use of the stack and queue abstract data types.

- Write software that performs operations on textual data.
- Experience working with a collection of data using a built-in data type (e.g., Java ArrayLists).
- Write software that deals with large sets of data using files stored on disk.
- Formulate recursive solutions to simple problems and write simple recursive methods.
- Implement simple searching and sorting algorithms.
- Discover how algorithm complexity is analyzed via big-Oh notation using simple searching and sorting algorithms as examples.
- Practice skills needed to write code that handles failures gracefully, including through basic error checking and the use of language constructs like Exceptions.

Learning outcomes

Introduction to procedural elements of Java

Students should be able to:

1. Write and run a procedural program in a *new* programming language (Java), transferring skills learned in previous courses (Python or Processing).
2. Download and install the JDK and a simple text editor / IDE (e.g., Dr. Java).
3. Write, compile, and run a basic Java program in a single file, with a main method.

OOP Basics

Students should be able to:

1. Write a simple-to-moderately complex class which includes constructors, instance variables and methods, and class variables and methods.
2. Compile and run a Java program with multiple files located in the same directory.
3. Use instances of user-defined classes in other user-defined class and within main methods.
4. Explain how and why the concept of encapsulation is useful, and how encapsulation is achieved via access modifiers and accessors / mutators.
5. Understand object references and use them appropriately in code, including the this keyword in Java and deep versus shallow object copies.

File I/O and Exceptions

Students should be able to:

1. Write code that creates, uses, throws, and catches built-in and user-defined exceptions.
2. Be aware that code can also use a finally block when handling exceptions.
3. List the order of operations in a try/catch/finally block when given a piece of code.

4. Write code that can read and write text files.
5. Write code that uses data from a file to instantiate objects.

Strings, ArrayLists, and Multi-dimensional arrays

Students should be able to:

1. Write code that performs a range of manipulations on textual data, including splitting a String according to a very simple expression (e.g., blank space), accessing individual characters and accessing substrings.
2. Create and use instances of a built-in Java data type such as an ArrayList.
3. Use Java-specific wrapper classes to manipulate primitive types as objects.
4. Compare and contrast arrays and a Java-defined data type.
5. Write code that declares, initializes, and uses multi-dimensional arrays, including ragged arrays.
6. Given a piece of code, draw a diagram representing the state of references in a multi-dimensional array.

Interfaces

Students should be able to:

1. Differentiate between an interface and its implementation.
2. Force a class to implement abstract methods by having it implement an interface.
3. Use interfaces as variable types, parameter types, and return value types.

Linked Lists

Students should be able to:

1. Compare and contrast lists and arrays based on operation running times and storage differences.
2. Write code that creates, traverses, and manipulates a linked list data structure.
3. Differentiate between an abstract data type and a data structure.
4. Implement Stack and Queue abstract data types using a linked list data structure.

Recursion

Students should be able to:

1. Create and implement recursive solutions to simple problems such as simple mathematical calculations and linked list traversals.
2. Write a recursive solution to a problem with a helper function.
3. Identify and explain the base case and recursive step components of a recursive algorithm.

Searching and Sorting

Students should be able to:

1. Describe why Computer Scientists care about the major "steps" in an algorithm over raw measurements like CPU time.
2. Express the complexity of a basic algorithm using big-O notation.
3. Compare and contrast the data management requirements of linear versus binary search.
4. Compare and contrast the running times of linear versus binary search using big-O notation.
5. Write code that implements linear and binary search on an array.
6. Describe sorting algorithms such as insertion, selection, merge, and quick sort in plain English.
7. Compare and contrast iterative and recursive sorting algorithms, including insertion, selection, merge, and quick sort.
8. Write code that implements simple sorting algorithms such as insertion, selection, and merge sort.