# CMSC 132 Overview

This document describes a minimal set of expectations for organization and teaching CMSC 132, Object-Oriented Programming II. We have included enough material to take up about 13 full weeks of lecture time, leaving some leeway for individual instructors to customize the course by either adding topics of their choosing, or spending more time emphasizing topics that they feel should be given more attention.

## Table of Contents

[High-Level Outcomes and Approach](#_soif9kg9clch)

[High-level Outcomes](#_mwtet5woei6c)

[Approach](#_mlvt73xgqox9)

[Course Topics](#_4rthoo92dmhq)

[Topics Outline](#_ilt1tz5a44m4)

[Note on Systematic Program Design](#_1q3kccsttynu)

[Optional Topics](#_2q293i2mxwi1)

[Sample Schedule](#_nfbi6k4e7llt)

# 

# High-Level Outcomes and Approach

This section presents outcomes of the course, organized at a high-level. Individual topics are organized below. This section also presents a teaching philosophy that aims to ensure these outcomes are met.

## High-level Outcomes

After this course, students should be able to:

1. Design effective and efficient solutions to computational problems
   1. Employ Systematic Program Design to go from a problem/concept to a well-designed, correct implementation
   2. Learn basic algorithm strategies, recognize problems in which they can be employed, and implement those algorithms in a standard language
   3. Know and apply standard data abstractions, algorithms and program principles
   4. Explain the abstract behavior and advantages/disadvantages of standard data abstractions including
      1. Linear data abstractions, including Lists, Stacks and Queues
      2. Search trees
      3. Graphs, weighted and unweighted, directed and undirected
      4. Sets and Maps
   5. Apply common algorithms associated with particular data abstractions, both recursive and iterative versions
   6. Explain numerous sorting algorithms and when and how to apply them.
   7. Explain and apply concepts of concurrent programming, including data races, deadlock, and synchronization.
2. Implement computational solutions effectively in a general O.O. programming language (e.g., Java)
   1. Explain how inheritance is used in Object-Oriented programming, and master the underlying intricacies.
   2. Create efficient, concrete implementations of standard data structures.
   3. Create implementations employing parameterized types.
   4. Use the Java Collections Framework, including basic use of generics
   5. Choose an appropriate Collection for a specific problem.
   6. Effectively apply advanced (Java) features, OO including:
      1. Enumerations
      2. Inner and nested classes
      3. Initialization blocks
      4. Comparators, Comparable
      5. Cloneable, clone
      6. Catching and throwing checked vs. unchecked exceptions
3. Analyze and test programs of moderate size and complexity
   1. Test and debug small (2000 lines or less) programs.
   2. Master the basics of algorithmic complexity, including big-O notation, and analyze complexity of (non-recursive) code fragments.

## Approach

To ensure students master these materials we place a **strong emphasis on solving problems through programming**. In particular, students will be expected to solve problems posed to them by writing code from scratch, both for take-home projects and on exams.

On quizzes and exams, the scope of the programming task will necessarily be smaller than a take-home project. Nevertheless, it should be challenging enough that students can demonstrate programming and problem solving facilities. It **extremely important to avoid leaning *only* on multiple-choice and short-answer questions**.Such questions are very useful for concepts and testing basic understanding, but they cannot take the place of the task for which we are preparing students directly: problem solving through programming.

For take-home projects, we emphasize realistic programming tasks, giving a sense of real-world programming. Here are some features we emphasize good programming projects should have:

* To help students achieve success, larger, more realistic **problems can be broken down into parts**, guiding the student from one part to the next until the entire task is achieved.
* Each project (part) should have **some aspect of “design.”** Good problems admit multiple solutions, some better than others.
* Projects should be **graded on coding style and non-correctness-related concerns** (e.g., algorithmic efficiency, a key theme of the class) in addition to correctness related one.
* Students should be provided with *some* correctness-related tests against which to judge their solution. Tests are useful, executable evidence that the student is on the right track, and clarify the project description. But s**ome tests used for grading should be “held back,”** so that students do not rely on instructor-provided tests too much. They need to think about ***how to test and debug their own programs***.

# Course Topics

The following topics define the typical content of this course. Topics are listed here conceptually; the order shown here is not intended to suggest order of presentation (see the [sample schedule](#_nfbi6k4e7llt) below). Highlighted topics define the critical "core" of the course -- they *must* be included and should be emphasized. Non-highlighted topics in the list below are ones we typically cover, and are recommended. Topics highlighted in green are expected to be review from a previous course.

Other potential topics are listed at the bottom of the page. It is likely that if all topics below are covered, there will be no time for these optional ones. If desired they might substitute for non-core topics in the main list.

## Topics Outline

After this course, students should be able to:

1. Design effective and efficient solutions to computational problems
   1. Systematic program design
   2. Identify and apply the general algorithmic concepts of
      1. Data and procedural abstraction
      2. Encapsulation
      3. Recursion
   3. List and explain the abstract behavior, conceptual implementation, and advantages/disadvantages of Data structures including
      1. Linear
         1. Array (review)
         2. Linked list
         3. Doubly-linked list
         4. Stack
         5. Queue
      2. Trees
         1. Binary tree
         2. Binary search tree
         3. Tree traversal algorithms
         4. Heap
      3. Graphs
         1. Weighted vs. unweighted
         2. Directed vs. undirected
         3. Underlying graph representations
      4. Set
      5. Hash tables
         1. Hashing with linear probing
         2. Hashing with buckets
         3. Load factor
         4. Finding a good hash function
   4. Tree and graph search algorithms
      1. Depth-first search
      2. Breadth-first search
      3. Djikstra's algorithm
      4. Kruskal's algorithm
   5. Sorting
      1. Heap sort
      2. Tree sort
      3. Bubble sort
      4. Selection sort
      5. Insertion sort
      6. Quicksort
      7. Mergesort
      8. Counting sort
      9. Radix sort
   6. Overview of algorithm strategies
      1. Divide and conquer
      2. Dynamic programming
      3. Greedy
      4. Brute force
      5. Branch and bound
      6. Backtracking
      7. Using heuristics
2. Implement computational solutions effectively in a general O.O. programming language (e.g., Java)
   1. Explain inheritance is used in Object-Oriented programming, and master the underlying intricacies.
      1. Inheritance and related mechanisms
         1. extends
         2. Superclasses and subclasses
         3. Java interfaces
         4. Static vs. dynamic binding
         5. Object class
         6. Inheritance vs. composition
         7. Abstract classes
         8. Singleton design pattern
      2. Working with inheritance
         1. Method overriding
         2. super
         3. Constructors (relying on superclass constructors)
         4. Protected visibility
         5. Casting
         6. instanceof
         7. Wrapper classes and autoboxing/unboxing
         8. Polymorphism
         9. Correct equals method
         10. final (methods, classes)
      3. Object equality
         1. Copy constructor
         2. Shallow vs. deep copy
      4. Use Java collections framework in implementing data structures
         1. Collection interface
         2. Iterator interface
         3. Iterable interface
         4. Comparable interface
         5. Comparator
         6. Java enumerations
         7. For-each loops
   2. Use specific core interfaces from the Collections
      1. List
         1. ArrayList
         2. LinkedList
      2. Set
         1. HashSet
         2. LinkedHashSet
         3. TreeSet
      3. Map
         1. HashMap
         2. LinkedHashMap
         3. TreeMap
      4. Hash details
         1. Java hashcode method
         2. Java hashcode contract
   3. Apply Java Generics
      1. Invoking a generic type
      2. Parameterizing over types
      3. Simple wildcard (? extends)
   4. Effectively apply more advanced OOP (Java) features including:
      1. Enumerations
      2. Initialization blocks
      3. Static initialization blocks
      4. Cloneable, clone
      5. Inner classes
      6. Nested classes
      7. Exceptions
         1. Using checked vs. unchecked exceptions, throwing exceptions
         2. Exception handling (throw, try/catch/finally)
   5. Garbage collection
   6. Concurrent programming
      1. Runnable interface
      2. Thread class
      3. Join
      4. Data races
      5. Synchronization
      6. Deadlock
3. Analyze and test algorithms and programs of moderate size and complexity
   1. Test and debug small (2000 lines or less) programs.
      1. Use JUnit
      2. Program errors (syntax, semantic, logical)
      3. Eclipse and the Eclipse Debugger
   2. Asymptotic complexity
      1. Intuition of growth rates
      2. Definition of big-O
      3. Easy proofs that f is O(g)
      4. Best/worst/average/expected cases
      5. Looking at (non-recursive) code and finding big-O

### 

### Note on Systematic Program Design

The small amount of lecture time allocated at the beginning of the course to present/review *Systematic Program Design* is not meant only to be presented during that lecture, but should also woven into lab sessions, exercises, and assignments throughout the course. Instructors who have taught previous offerings of CMSC 131 and/or CMSC 132 are encouraged to spend time considering this approach, and to find ways to incorporate it into existing materials that they may wish to continue using.

## Optional Topics

The following are suggestions for optional topics that instructors may choose to incorporate into the course. This list is not intended to be exhaustive.

* GUIs, MVC model, Swing library, JavaFX, etc.
* Design patterns (visitor, observer, etc.)
* More advanced techniques for multi-threaded applications
* Software lifecycle, process models
* More about asymptotics (theta, little-o, big-omega, little-omega, P vs. NP, etc.)
* Self-balancing trees (red/black trees, 2-3-4 trees, etc.)

# 

# Sample Schedule

Below is a potential weekly schedule for the course. We have allocated 13 weeks of total lecture time. The remaining time during the semester can be used to accommodate a selection of optional topics, at the instructor's discretion. More detailed lecture notes are available separately.

|  |  |
| --- | --- |
| Week 01 | Course intro, Eclipse intro, review Java interfaces, polymorphism, data and procedural abstraction, encapsulation, Java collections overview, Collections interface, basic generic notation (ArrayList), for-each loops, Iterator interface, Iterable, enumerated types, autoboxing/unboxing |
| Week 02 | Inheritance, super, constructors of subclasses, casting, overriding, early vs. late binding, protected visibility, Object class, abstract classes, instanceof, "final", equals method, Comparable |
| Week 03 | Comparators, testing, code coverage, programming errors, exception handling, checked vs. unchecked exceptions, throws, inner classes, nested classes, deep vs. shallow copy |
| Week 04 | clone, Cloneable, initialization blocks, static initialization blocks, garbage collection, systematic program design |
| Week 05 | Asymptotic complexity, big-O notation, best/average/expected/worst cases, analyzing code fragments, classes with parameterized types, ? extends, overview of linear data structures and data types |
| Week 06 | Implementing linked lists, stacks and queues, doubly-linked list, Java ArrayList, LinkedList |
| Week 07 | Set (ADT), hash tables, linear probing, hashing with buckets, Java hashCode, HashSet, LinkedHashSet, TreeSet, maps, HashMap, LinkedHashMap, TreeMap, begin recursion |
| Week 08 | More recursion, singleton design pattern, trees, binary search tree |
| Week 09 | BST traversals, tree sort, heaps, heapsort, priority queue, begin graphs |
| Week 10 | More graphs, graph implementations, depth-first search, depth-first search, Djikstra's algorithm, begin sorting (bubble sort, selection sort) |
| Week 11 | More sorting (mergesort, quicksort, counting sort, radix sort), concurrent programming, threads, Runnable, Thread class, join, begin data races |
| Week 12 | More data races, synchronization, deadlock, Java I/O streams |
| Week 13 | Intro to networking, threaded web server example, overview of algorithm strategies |