# CMSC 131 Overview

This repository contains materials used for teaching CMSC 131 (Object-Oriented Programming I) at UMCP. There is enough material to take up about 14 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.

The content has been carefully divided into two halves to accommodate the possibility of having two smaller courses. The second half of the material corresponds, in our experience, to the mini-course needed by students with some experience, but not sufficient experience to bypass 131 entirely. The first half contains the basic fundamentals (variables, loops, conditional statements, etc.). The second half contains more advanced material (Java interfaces, privacy leaks, recursion, etc.).

## Table of Contents

[High-level Outcomes and Approach](#_9hpmr9nnjktz)

[High-level Learning Outcomes](#_lpfv1ziqqv6m)

[Approach](#_mlvt73xgqox9)

[Course Topics](#_fl5v1c7gg8g3)

[First half of course](#_uiorrggwhk5)

[Second Half](#_8uayoow76xpn)

[Optional Topics](#_ick4tq6e14ub)

[Sample Lecture Schedule](#_myjesgy9zmo8)

[First Half](#_hr7dyqw8s9p5)

[Second Half](#_pn7hzku31lrr)

# 

# 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 Learning Outcomes

1. Define and apply definitions of key terminology related to programming and computing.
2. Read, write, and apply elements of a general O.O. programming language (e.g., Java)
   1. Write declarative statements and use operators, conditional statements, and loops to solve specific tasks
   2. Write (Java) methods and classes
   3. Use (Java) primitive types and use their encodings
   4. Learn about collections, including arrays and Java's ArrayList class
   5. Master basic exception handling (both throwing and handling them)
   6. Use appropriate tools (e.g., Eclipse) to enter and run programs
3. Solve and document problems of complexity illustrated by example projects
   1. Design algorithms to accomplish tasks described in natural language.
   2. Learn to apply *systematic program design* to go from a problem or concept to an actual implementation.
      1. Object-Oriented approach
      2. Identifying design recipes from problem statements
      3. Design test case scenarios that can be used to verify the accuracy of executable code
   3. Write code to implement an algorithm described in natural language or pseudo-code, implementing appropriate methods and classes, as necessary.
   4. Regularly write informative comments that tie code back to description of a problem or algorithm
   5. Identify and solve problems where a recursive approach works well.
4. Trace, test, and debug a program
   1. Recognize syntactic errors in a block of text that is supposed to represent a valid Java program
   2. Find and fix semantic software bugs using tools available in their programming environment
   3. Trace given sections of code by hand to predict the output
   4. Trace given sections of code and draw detailed memory diagrams including the call stack and heap
   5. Explain the importance of unit testing, and incorporate it as an essential component of software development
      1. Develop working familiarity with JUnit
      2. Learn to write concise tests for general cases
      3. Learn to identify and test corner cases
      4. Explain the limitations of unit testing

## 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 **some 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, other than those grouped into the first half of the course, and the second. Highlighted topics define the critical "core" of the course -- they *must* be included and should be emphasized. Non-highlighted topics are ones we typically cover, and are recommended. This material should take about 14 weeks total, so instructors should have room to add some optional content; some topics are listed at the bottom of the page.

## First half of course

1. Define and apply definitions of key terminology related to programming and computing.
   1. Brief history of programming languages
   2. RAM and how numbers/text are stored
   3. Number base conversions
   4. Garbage collection
2. Read, write and apply elements of a general OO programming language (eg, Java)
   1. Overview of Eclipse IDE
   2. Variables and basic types
      1. Local variables
      2. Primitive types
      3. Strings
      4. Memory requirements for primitives
      5. Reference variables (What is stored?)
      6. Variable initialization and default values
   3. Basic operators and expressions
      1. Arithmetic (+, -, \*, /, %)
         1. Integer division
      2. Comparison (, <=, >=)
      3. Equality (==, !=)
      4. Logical (&&, ||, !)
         1. Short circuiting
      5. Increment/decrement (++, --)
      6. Assignment operator
      7. Compound assignment operators (+=, -=, /=, \*=, %=)
      8. Precedence rules
   4. Text input/output
      1. System.out.println
      2. Scanner
   5. Conditional statements
      1. if
      2. if-else
      3. "else-if" style
      4. switch
   6. Loops
      1. while
      2. do-while
      3. for
      4. break
      5. continue
   7. Static methods
      1. Writing methods
      2. How method calls work
      3. Parameters
      4. Return types/values
   8. Standard Java methods
      1. Calling equals method
   9. Classes
      1. Static variables
      2. Visibility and encapsulation
      3. Variable scope
      4. Packages and import
   10. Objects
       1. Instance variables
       2. Instance methods
       3. "this"
       4. Constructors (including copy constructor)
       5. toString
3. Solve and document problems of complexity illustrated by example projects
   1. Proper style
      1. Choosing identifiers
      2. Placement of braces
      3. Indentation
      4. Named constants
      5. Proper commenting (without Javadoc)
4. Trace, test and debug a program
   1. Classifying programming errors (syntax, semantic, logical)
   2. Memory Diagrams
      1. Call stack (with frames)
      2. Heap

#### 

## Second Half

1. Define and apply definitions of key terminology related to programming and computing.
   1. Roundoff error
2. Read, write and apply elements of a general OO programming language (eg, Java.)
   1. Control structures
      1. For-each loops
      2. Recursion
   2. Exception handling
      1. When/how exceptions are thrown
      2. try
      3. catch
      4. finally
   3. Classes
      1. Java interfaces
      2. Polymorphism
      3. Wrapper classes and auto boxing/unboxing
      4. Writing equals method (with Object parameter)
      5. Idea of inheritance (via extension)
      6. Object class
   4. Memory issues
      1. Mutability
      2. Privacy leaks
      3. Defensive copies
      4. Deep vs. shallow copy
   5. Collections and data types
      1. Arrays
         1. One dimensional
         2. Two dimensional (rectangular only)
      2. Overview of collections
      3. Brief description of common Abstract Data Types (ADTs)
      4. Brief overview of Java collections framework
      5. Basic generic notation
      6. ArrayList
3. Solve and document problems of complexity illustrated by example projects
   1. Javadoc
4. Trace, test and debug a program
   1. Unit testing and JUnit
      1. Limitations of testing (and notion of "formal verification")
      2. Writing effective tests for general cases
      3. Identifying and testing corner cases
   2. Using Eclipse debugger

### 

### Note about on Systematic Program Design

We have allocated a small amount of lecture time in the beginning of the second half of the course to *Systematic Program Design*. Our intention is for this material to be not only presented during the lecture, but also woven into lab sessions, exercises, and assignments throughout the second half of this course, and throughout CMSC 132. 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.

* More detailed intro to inheritance
* More advanced details about interfaces
* Compilers vs. interpreters
* File I/O
* Intuitive idea of complexity (Big-O)
* Ternary operator
* Additional data types (Sets, Maps, etc.)

# 

# Sample Lecture Schedule

This is a potential weekly schedule for the course. We have allocated 14 weeks of total lecture time, divided into two (7 week) halves to facilitate the possibility of offering two separate half-semester courses. We anticipate that the remaining time during the semester will be used to accommodate a selection of "optional" topics, at the instructor's discretion. See the lecture notes for details about each of these topics.

Note that individual instructors may choose to order topics differently. In particular, instructors who are teaching the full-semester 131 (not divided into halves) are free to move topics from one half to the other. For example, some instructors might like to present JUnit earlier in the semester, even though it is included in the second half, below.

#### 

## First Half

|  |  |
| --- | --- |
| Week 01 | Course intro, Eclipse IDE, brief history of programming languages, RAM and storage of numbers/text, number base conversions |
| Week 02 | Local variables, primitive types, assignment, Strings, concatenation, creating a project in Eclipse, System.out.println, Scanner, arithmetic operators, integer division, comparison and equality operators, calling equals method |
| Week 03 | Conditional statements (if, if-else, switch), logical operators, short-circuiting, ternary operator (?:), choosing identifiers, named constants, programming errors, variable scope and initialization |
| Week 04 | Loops (while, do-while, for), nested loops, break, continue, increment/decrement operators, alternate assignment operators, rules of precedence, casting with primitives |
| Week 05 | Writing static methods, parameters, return values, method overloading, static variables, intro to objects, instance variables and instance methods, assignment with references, memory diagrams, == vs. equals, garbage collection, this |
| Week 06 | Constructors, copy constructor, this in constructors, toString, commenting, memory diagrams for method calls (frames) |
| Week 07 | Public vs. private visibilities, encapsulation, "API", Java packages, import |

## Second Half

|  |  |
| --- | --- |
| Week 01 | Notion of code correctness, testing, JUnit, Eclipse debugger, pseudocode |
| Week 02 | Systematic program design, roundoff error, begin exception handling (throw, try/catch) |
| Week 03 | Continue exception handling (multiple catches, finally), overview of Abstract Data Types (ADTs) and Java collections framework, basic generic notation, ArrayList |
| Week 04 | Arrays of primitives, arrays of references, copying arrays, "resizing" arrays, two dimensional rectangular arrays, for-each loops |
| Week 05 | Mutability, StringBuffer, deep vs. shallow copy, privacy leaks, defensive copies, basic recursion, wrapper classes, autoboxing/unboxing |
| Week 06 | Java interfaces, polymorphism, brief intro to inheritance, overriding methods, Object class, writing equals method |
| Week 07 | More advanced recursion, Javadoc comments and Javadoc utility |