CS 2110: Object-Oriented Programming and Data Structures

Lectures

Here we have space to list detailed reading expectations for each lecture, along with instructor commentary that may be a little tangential.

Lecture 1: Course overview

Post-lecture tasks

There are a lot of these after the first lecture, but don't worry—the pace will even out quickly.

- Follow the setup guide to install and configure IntelliJ on your computer
- Register your iClicker remote
- Read the syllabus (yes, all of it) and take the **syllabus quiz** on Canvas
- If you are new to Java, read the transition to Java guide
- Access the textbook's online supplement 1: Java Basics (this will be assigned reading for the next few lectures, but it is only available online)
- Start working on **Assignment 1**

Required reading

If you are already familiar with Java, you can skim these resources to review any details you may have forgotten. If you are new to Java, these are the resources you should use to learn its procedural syntax (these foundations will not be discussed further in lecture).

- Transition to Java
- Supplement 1—Java Basics: S1.2–S1.31, S1.38–S1.50, S1.57–S1.63, S1.66–S1.80, S1.86–S1.96

Supplemental reading

Supplemental readings are optional, but they should be the first place you go if a topic is still unclear after reviewing lecture notes and the textbook. They are also valuable if you want to broaden and deepen your knowledge of the material. These curated links are of much higher quality than the top search results on Google.

- The Java Tutorials: Language Basics

Lecture 2: Objects

Required reading

Unless linked or otherwise specified, readings refer to chapters and sections from our textbook, *Data Structures and Abstractions with Java, Fifth Edition*. If you do not have a paper copy, you should be able to access an electronic version via the "Course Materials" link on Canvas.

- Appendix B—Java Classes: B.1–B.16, B.22–B.28, B.34–B.35
- Java Interlude 6—Mutable and Immutable Objects
- Object Diagram Rules

Post-lecture tasks

- Take **Quiz 1** on Canvas

Supplemental reading

- The Java Tutorials: Classes and Objects
- JavaHyperText: *types* 0–2, *classes* 1A–1C
- Myers: Objects and Values

Lecture 3: Abstraction, encapsulation

Required reading

- Prelude—Designing Classes: P.1–P.11
- Appendix B—Java Classes: B.17–B.26

Supplemental reading

- JavaHyperText: *access modifier*, *constructor*, *new-expression*…
- Myers: Encapsulation and Information Hiding

Lecture 4: Specifications, testing

Required reading

- Appendix A—Documentation and Programming Style: A.1–A.12
- Code style

Post-lecture tasks

- Take **Quiz 2** on Canvas

Supplemental reading

- JavaHyperText: *class invariant*, *javadoc comment*, *specification*, *precondition*, *postcondition*
- Myers: Designing and documenting interfaces, Modular Design and Testing

Lecture 5: Interfaces, subtyping

Required reading

- Prelude—Designing Classes: P.12–P.20

Post-lecture tasks

- Start working on **Assignment 2**

Supplemental reading

- The Java Tutorials: Interfaces
- JavaHyperText: *interface*, *compile-time reference rule*
- Myers: Interfaces and subtyping

Lecture 6: Inheritance

Required reading

- Appendix C—Creating Classes from Other Classes: C.1–C.24
- Prelude—Designing Classes: P.21–P.22, P.28–P.31

Post-lecture tasks

- Take **Quiz 3** on Canvas
- Review Assignment 1 feedback

Supplemental reading

- The Java Tutorials: Inheritance
- JavaHyperText: *inheritance*, *extends*, *subclass*, *superclass*, *bottom-up rule*, *override*, *super*, *this*
- Myers: Inheritance and the specialization interface

Lecture 7: Exceptions, I/O

Required reading

- Java Interlude 2—Exceptions
- Java Interlude 3—More About Exceptions: J3.1–J3.8
- Supplement 1—Java Basics: S1.32–S1.37, S1.81–S1.85
- Supplement 2—File Input and Output: S2.1–S2.19
- Input/output in Java

Supplemental reading

- The Java Tutorials: Exceptions, Basic I/O
- JavaHyperText: *exceptions*
- Myers: Exceptions

Lecture 8: Generics

Required reading

- Introduction—Organizing Data
- Chapter 1—Bags
- Java Interlude 1—Generics
- Chapter 2—Bag Implementations That Use Arrays

Supplemental reading

- The Java Tutorials: Generics, Introduction to Collections
- JavaHyperText: *bag*, *data structure*, *abstract data type*, *generics*
- Myers: Parametric Polymorphism (generics)

Post-lecture tasks

- Take **Quiz 4** on Canvas

Lecture 9: Linked structures

Required reading

- Chapter 3—A Bag Implementation That Links Data

Post-lecture tasks

- Start working on **Assignment 3**

Lecture 10: Ordered collections

Required reading

- Chapter 5—Stacks: 5.1–5.4
- Chapter 6—Stack Implementations: 6.1–6.12
- Chapter 7: 7.1–7.13 (Queue ADT)
- Chapter 8: 8.1–8.16 (Queue implementations: linked and array-based)
- Chapter 10—Lists
- Chapter 11—A List Implementation That Uses an Array: 11.1–11.13
- Chapter 12—A List Implementation That Links Data: 12.1–12.24

Supplemental reading

- JavaHyperText: *list*, *linked list*, *doubly-linked list*, *stack*, *queue*
- Myers: Linked lists

Lecture 11: Efficiency

Required reading

- Chapter 4—The Efficiency of Algorithms
- Chapter 19—Searching: 19.1–19.18, 19.21–19.24 (skips recursive searches, for now; will revisit in Lecture 15)

Post-lecture tasks

- Take **Quiz 5** on Canvas

Supplemental reading

- JavaHyperText: *algorithmic complexity*
- Myers: Asymptotic complexity

Lecture 12: Recursion

Required reading

- Chapter 5—Stacks: 5.22
- Chapter 9—Recursion

Supplemental reading

- JavaHyperText: *recursion* (do you get the joke?)
- Myers: Recursion
- Chapter 14—Problem Solving with Recursion

Lecture 13: Trees I

Required reading

- Chapter 24: 24.1–24.17, 24.22–22.24, 24.28–24.31
- Chapter 25: 25.1–25.2, 25.9–25.11, 25.18
- Chapter 26: 26.1–26.18, 26.40–26.43

Instructor’s notes

The textbook implements tree traversals using `Iterator`s so that the client is responsible for whatever operation should be done when each node is “visited.” This is a good design, but they are not quite as intuitive as a recursive implementation, since they need to explicitly maintain state between calls.

I find the textbook’s definition of a `TreeInterface` to be a little superfluous. Seen as a recursive data structure, a `Node` class is sufficient to represent a tree data structure, which could then be used to implement ADTs like Set, Dictionary, and Priority Queue, as well as more application-specific data types.

Post-lecture tasks

- Take **Quiz 6** on Canvas

Supplemental reading

- JavaHyperText: *trees* (1, 2, 3, 5)
- Myers: Trees

Lecture 14: Trees II

Post-lecture tasks

- Start working on **Assignment 4**

Lecture 15: Loop invariants

Required reading (JavaHyperText)

- Program correctness
- Loop invariants
- Linear search loop development
- Binary search loop development

Supplemental reading

- Myers: Loop invariants

Post-lecture tasks

- Take **Quiz 7** on Canvas

Lecture 16: Sorting

Required reading

- Chapter 15—An Introduction to Sorting: 15.1–15.14
- Chapter 16—Faster Sorting Methods: 16.1–16.7, 16.9–16.19, 16.23

Instructor’s notes

The textbook’s implementation of `partition()` is more complicated than what we present in class and requires a non-trivial precondition (a minimum array size of 4). I think our version, derived from a simple loop invariant, is much easier to reason about and implement correctly. That being said, the textbook’s version can produce a more balanced partition when lots of duplicate elements equal to the pivot value are present.

Note that the textbook’s functions operate on closed array ranges (e.g. `a[first..last]`), whereas lecture examples preferred half-open ranges (e.g. `a[begin..end)`). Using half-open ranges avoids a lot of `+1`s and `-1`s and is the convention adopted by many library functions in both Java and C++.

Supplemental reading

- JavaHyperText: *sorting* (1, 2)
- Myers: Sorting

Lecture 17: Dictionaries and hashing

Required reading

- Chapter 20—Dictionaries
- Chapter 22—Introducing Hashing: 22.1–22.8, 22.10–22.15, 22.24–22.26
- Chapter 23—Hashing as a Dictionary Implementation: 23.1–23.2, 23.6–23.8, 23.16–23.17

Instructor's notes

The textbook's treatment of hashing is very "classical" (I have a textbook from 1984 that treats the topic in the same way). In response to the fact that real-world hash codes are not uniformly distributed (which is still true of Java's hash codes for most built-in types), it advocates for limiting table sizes to prime numbers and dedicates several pages to quadratic probing and double hashing. But there are ways to improve the diffusion of the index derivation process (for example, by computing a quality hash of the original hash code), and with uniformly-distributed indices, these approaches are less relevant in practice. As an example, Java's `HashMap` uses tables whose sizes are a power of 2.

Chaining remains a popular and performant approach to collision resolution, but linear probing has seen a resurgence in popularity. Techniques like "Robin Hood hashing" reduce the number of probes necessary to find keys contained within the table, and keeping the probe sequence close to the original index provides good memory locality, allowing clusters to be searched very quickly despite their length.

In section 23.2, the textbook claims that, for probing, elements in the "available" state do not affect the load factor. Unless I'm missing something, this is false—the presence of "tombstones" in a table causes all searches to take longer, so they must be included in the numerator of the load factor definition.

Post-lecture tasks

- Take **Quiz 8** on Canvas

Supplemental reading

- JavaHyperText: *hash table*
- Myers: Hash tables

Lecture 18: Graphical User Interfaces

Instructor's notes

The textbook does not cover graphical user interfaces, and there are too many classes and methods provided by the Swing framework to ask you to read about them all. But you should be familiar with the following general concepts:

- Windows and `JFrame`
- Containers and layout managers
- Components and common subclasses (buttons, labels, text fields)
- Separating concerns into Model, View, and Controller (MVC)

The Java Tutorial trail Creating a GUI With Swing is a good reference, and sections of it are required reading for A5.

Supplemental reading

- The Swing tutorial: Using Top-Level Containers
- The Swing tutorial: A Visual Guide to Layout Managers
- Myers: Graphical User Interfaces: Display and Layout

Lecture 19: Event-driven programming

Instructor's notes

The textbook does not cover graphical user interfaces, and there are too many classes and methods provided by the Swing framework to ask you to read about them all. But you should be familiar with the following general concepts:

- Inversion of control
- Events, event sources, and event handlers
- The Event Dispatch Thread (EDT)
- Painting

The Java Tutorial lesson Writing Event Listeners is a good reference, and I recommend reading at least the first two sections to supplement our slides. The lesson Concurrency in Swing is also important and relates to our upcoming lecture on concurrency.

Lambda expressions have many applications outside of GUIs. You have already seen `assertThrows()` in JUnit tests for assignments.

Post-lecture tasks

- Start working on **Assignment 5**

Supplemental reading

- The Swing Tutorial: Writing Event Listeners
- The Swing Tutorial: Concurrency in Swing
- Myers: Graphical User Interfaces: Events
- The Java Tutorials: Lambda Expressions

Lecture 20: Concurrency

Required reading

- Processes and Threads
- Thread Objects (including subsections)
- Thread Interference, Memory Consistency Errors

Instructor's notes

The textbook does not cover concurrency. The Java Tutorial lesson Concurrency is a good reference for this lecture and the next. You should be familiar with the following general concepts:

- Threads of execution
- Java's `Thread` class
- `Runnable`
- Concurrent execution via time slicing or parallelism
- Race conditions

> One inconvenience when experimenting with threads is that *blocking* operations like `sleep()`, `join()`, and `wait()` could throw `InterruptedException`, which is a "checked" exception that you must handle before it propagates outside of your `Runnable`'s `run()` method. Java does this to support universal thread *interruption*—if an application needs all of its threads to stop working and clean up so it can exit, it can call `interrupt()` on all of its `Thread` objects. Thread runnables are supposed to periodically poll their thread's `interrupted()` status to know when to clean up, but if the interrupt occurs while they are blocked, then the blocking method will complete early and throw `InterruptedException` to let its caller know.
>
> The general rule for handling checked exceptions applies: if the caller knows how to respond to the exception (i.e., how to halt the task and clean up), then it should do so; otherwise, it should propagate the exception by declaring that it, too, `throws InterruptedException`. It is almost never appropriate to "swallow" the exception and ignore it. In demos, where interrupts will not occur and where no cleanup would be necessary anyway, our preferred approach is to propagate the exceptions up to `run()`, which will then catch and rethrow them as unchecked `RuntimeException`s. This expresses the intent that they are unexpected while still responding to them in the desired way (stopping the thread).

Supplemental reading

- JavaHyperText: *concurrency*
- Myers: Concurrency

Lecture 21: Synchronization

Required reading

- Synchronized Methods, Intrinsic Locks and Synchronization
- Deadlock
- Guarded Blocks

Instructor's notes

The textbook does not cover concurrency. The Java Tutorial lesson Concurrency is a good reference for this lecture and the previous. For more detail, including lots of examples, read *Java Concurrency in Practice*. You should be familiar with the following general concepts:

- Producer–consumer problems
- Ring buffers
- Locks (mutexes) and the `synchronized` keyword
- Deadlock
- Condition variables: `wait()` and `notifyAll()`

Post-lecture tasks

- Take **Quiz 9** on Canvas

Supplemental reading

- Myers: Synchronization

Lecture 22: Graphs

Required reading

- Chapter 29: 29.1–29.10, 29.25–29.26
- Chapter 30: 30.1–30.22

Instructor's notes

We will cover chapters 29 and 30 over the course of two or three lectures. The graph *algorithms* parts of the chapters are the main sections we skip in this introductory lecture. So for today's reading you can skip over anything about traversals or shortest paths; we'll come back to those later. Do concentrate on terminology and on representations.

Supplemental reading

- JavaHyperText: graphs, topics 1–3: definitions, terminology, representations
- Myers: Graphs

Lecture 23: Graph traversals

Required reading

- Chapter 29: 29.11–29.15
- Chapter 30: 30.23

Instructor’s notes

DFS is most cleanly implemented recursively (as in lecture), which is not shown in the textbook. The textbook’s iterative implementation does a better job of matching a recursive version than others I’ve seen (e.g. JavaHyperText), but the pseudocode leaves out some details, and there is no concrete implementation in chapter 30.

The textbook’s treatment of topological sorting describes Kahn’s algorithm, but in reverse, starting with the last vertex in the order and accumulating vertices in a Stack to effectively reverse the order that they were added in. It also only provides pseudocode, leaving out details for how to efficiently find nodes with no successors after vertices have been removed. The lecture slides and demo code show the “forward” Kahn’s algorithm, including an efficient way of identifying nodes with no predecessors. Discussion activity 12 will introduce an alternative algorithm that uses depth-first search instead (which I find more elegant). It turns out that reversing the DFS “settled” order gives a topological order.

Post-lecture tasks

- Take **Quiz 10** on Canvas

Supplemental reading

- JavaHyperText: Depth-first and breadth-first search
- Myers: Graph traversals

Lecture 24: Shortest paths

Required reading

- The shortest-path algorithm
- Chapter 29: 29.16–29.24
- Chapter 30: 30.24

Instructor’s notes

The online reading (including videos) from JavaHyperText is much more thorough than the textbook on this topic, highlighting the loop invariant and proof of correctness.

Post-lecture tasks

- Start working on **Assignment 6**

Supplemental reading

- Myers: Dijkstra’s single-source shortest path algorithm

Lecture 25: Priority queues and heaps

Required reading

- Chapter 7: 7.19–7.21
- Chapter 8: 8.34
- Chapter 24: 24.32–24.34
- Chapter 27

Instructor’s notes

The textbook places the root of the tree at array index 1 instead of 0, which simplifies the parent/children formulas a bit. But it later reimplements `reheap()` (analogous to our `bubbleDown()`) using a 0-based convention for use in heapsort. We’ll stick to the 0-based convention.

The textbook makes a point of avoiding swaps when bubbling up and down by only moving existing values and not writing the new value until its final location is known. This is a legitimate optimization, but we’ll stick with swaps because they make it easier to visualize the comparisons that are taking place.

At this point in the textbook, its method specifications range from poor to missing altogether. We’ll try to do better in our demo code.

Post-lecture tasks

- Take **Quiz 11** on Canvas

Supplemental reading

- JavaHyperText: *Heaps*, *HeapSort*

- Myers: Priority Queues and Heaps