As with previous chapters, we are going to lay out the learning objectives for this chapter, and split them into what you should get on the first reading vs the second reading.

1 First Reading: Understand Key Ideas

On your first reading, the goal is to get the big ideas, and build the mental scaffolding in which to hold the details on your second reading—the learning objectives for your first reading are generally **Remember** and **Understand** objectives. If you are missing a few details (but not completely lost), it is ok to move ahead, and come back to them the next time through—you might ask questions on the course forums too! Note you should **sleep at least one night between the first and second reading** to give your brain time to process the information.

- **Programming in the Large: Intro**
  - **Define** *programming in the small.*
  - **Explain** three differences between programs you have been working on and real-world programs.
  - **Define** *programming in the large.*
  - **Explain** how programming the small interacts with programming in the large.
  - **Recognize** that we only introduce programming in the large here, and that you need a Software Engineering class to go deeper into it.
- **Abstraction**
  - **Recall** what *abstraction* is (from Chapter 3).
  - **Recall** what the *interface* is (from Chapter 3).
  - **Recall** what the *implementation* is (from Chapter 3).
  - **Explain** how abstraction lets you use functions from a library more easily.
  - **State** the number of items the human brain can work with at any given time.
  - **Explain** why it is hard to remember a sequence of gibberish letters, but easy to remember a meaningful word of the same length.
  - **Explain** how the seven item limit of human brain capacity relates to programming.
  - **Define** *hierarchical abstraction.*
  - **Explain** how hierarchical abstraction relates to the seven item limit of human brain capacity.
  - **Define** *bottom-up design.*
  - **Define** *top-down design.*
  - **Explain** the tradeoffs between bottom-up and top-down design.
  - **Explain** some ways to mitigate the downsides of top-down design.
  - **Define** *test stub*
    - Note: there are a variety of more complex approaches to test stubs—you will learn about them in a Software Engineering course.

  - Note: for either type of design, when you are building a program of significant size, you will want to make a minimal runnable system, test, add features, and repeat. This process will be discussed much more in a Software Engineering course.

- **Readability**
  - **Define** *readability.*
  - **Explain** why readability is important in real programs, but easily overlooked in introductory programming contexts.
  - **State** the rule for the maximum size of a single function.
  - **Explain** why that rule is not the only factor in determining if a function is appropriately sized.
  - **Explain** why names are important, and how to name things well.
  - **Explain** why formatting is important.
  - **Explain** how bracing styles can help prevent (or not) certain types of errors.
  - **Explain** how good comments can enhance readability.
  - **Explain** the guidelines for good comments.

- **Working In Teams**
  - **Explain** why/how working in teams magnifies the other considerations previously discussed in this chapter.
  - **Explain** why revision control is even more important when working in a team.
  - **Define** *integration*

- **A Modestly Sized Example**
  - **State** how to apply The Seven Steps to much larger problems.
  - **Explain** why very high-level steps are appropriate at the top-level of a large problem.
  - **Explain** how to think about what types you need from your work with Step 1–3 (note: this LO has appeared previously, but we revisit it now, especially as you have the knowledge to build more complex data structures now).
  - **Explain** why it is better to have many small programming tasks rather than one large programming task.
  - **Explain** why you might want to write functions to print output that is not going to be needed in the final program (*e.g.*, printClassRoster).
  - Note: The `main` that just reads the input and prints the roster is a great example of starting with a minimal runnable program, and then adding features. This idea is discussed briefly in the next section, but in much more detail in a Software Engineering course.
  - **Explain** how abstracting code into small functions makes it easier to make changes later.

- **Even Larger Programs**

- **Define** *Big Bang Development*.
- **Explain** why big bang development may be tempting, but is a bad idea.
- **Explain** "haste makes waste".

2 Second Reading

As always, we strongly recommend you sleep between your first and second reading. The second reading is where you want to focus on the higher-level learning objectives, which build on the lower-level learning objectives you worked on in the first reading. Sleeping gives your brain a chance to process the information from the first reading.

We are going to note that this chapter is a bit different than the others. First, we are trying to help you move towards larger programs. While you will work on learning objectives related to this topic—and move to slightly larger programs than you have been—really getting into these learning objectives is left to a Software Engineering course. For *all* the learning objectives here, you should be at an "I can start to work on that" point when you finish your second reading—mastery will come over a long time with a lot of work.

- **Abstraction**
  - **Separate** code into interface and implementation.
  - **Assess** when and why functions are too complex for programmers to easily think about.
  - **Break down** a large programming task into smaller tasks.
  - **Perform** incremental development.
  - Note: we leave entirely to a Software Engineering course learning objectives like **write** test scaffolds.
- **Readability**
  - **Assess** the readability of code.
  - **Select** good names for functions and variables.
  - **Use** consistent and appropriate formatting.
  - **Write** appropriate documentation for your code.
- **Working In Teams**
  - Note: you are working on the mechanics of programming individually in this course, so all the higher-level learning objectives (which need practice) for working in teams are left to Software Engineering.
- **A Modestly Sized Example**
  - Note: This section works through one example in detail to help with a lot of the learning objectives we listed above. We won't repeat those LOs here, but you should work through this in detail to help with all those LOs. In particular, you should see how the `readInput` here is much better (in all the ways we discussed) than the equivalent `readInput` at the start of the chapter. You should also be able to break down something of this size/complexity yourself.

  - **Simplify** a programming task to the minimal runnable program (like the `main` which only reads and prints the roster).
  - **Assess** when additional helper functions are useful to show intermediate state (which are not required in the final output—like printing the class roster).

- **Even Larger Programs**
  - **Select** the smallest runnable iteration of your program.
  - **Recommend** an order to add features/capabilities to your program.
  - **Determine** when you need to switch between writing code and testing code (using incremental development).
  - **Write** much larger programs!

3 Key Learning Objectives

As we mentioned in our discussion of the second reading, this chapter touches the boundary of our learning—with an introduction to programming in the large, but much of the practice and deeper discussion left to Software Engineering. However, the following learning objectives are the most important to work on over the rest of this semester.

- **Perform** incremental development—test frequently.
- **Break down** a programming problem into smaller tasks.
- **Assess** what additional code might help with testing and debugging.
- **Assess** when and why functions are too large and complex to think about easily.
- **Design** and **implement** larger programs.