Chapter 1

Big Ideas

This course is an introduction to programming and problem solving. We shall focus on some "big ideas" of computer science – key themes that are seen again and again throughout the discipline. While mastering these ideas is clearly important if you are planning to major or minor in computer science, it is worthwhile to take this course even if you do not plan to take any other computer science courses. The ideas you will learn here crop up in various forms in many other disciplines, such as mathematics, natural sciences, engineering, and art. And the problem solving skills you learn here will help you formulate and solve problems in any field.

We will use the Java programming language as a tool for exploring the big ideas of computer science. However, the focus of the course is on the big ideas, and not Java per se. Indeed, you should be able to transfer what you learn here to programming in most any programming language. While you will learn a significant amount about programming in Java, there are many important aspects of practical Java programming that you will not learn in this course; you will need to take further courses if you wish to learn these.

In the rest of this chapter, we briefly introduce the big ideas covered in this course.

1.1 Computational Recipes

Computer science is not really a science.[1] Scientists form hypotheses about how the world works and perform experiments to test these hypotheses. Although there are exceptions, most computer scientists do not do this. Instead, they design and build computational artifacts, In this sense, computer scientists are more like engineers or artists.

Computer science does not have much to do with computers. Although computers are an important tool used by computer scientists to explore computation, they are just a tool. Computer science is no more about computers than chemistry is about test tubes and Bunsen burners.

If computer science is not a science and has little to do with computers, then what is it? The essence of computer science is *imperative ("how to") knowledge*, which stands in contrast to the *declarative ("what is") knowledge* on which mathematicians tend to focus.

[1]The view of computer science espoused here was advanced by Hal Abelson and Jerry Sussman in their introductory MIT course (6.001). To them we owe the insight that computer science is not a science and has little to do with computers but is really about imperative knowledge and controlling the complexity of large systems.

For example, to a mathematician, the fraction $\frac{22}{7}$ is a number that, when multiplied by 7, yields 22. To a computer scientest, $\frac{22}{7}$ implies a *process* (such as long division) for determining the digits in the floating point representation $3.\overline{142857}$.

Imperative knowledge is expressed via *algorithms*, which are *computational recipes* that specify how a computational process evolves over time. Algorithms can be encoded as *programs* written in formal languages known as *programming languages*. In this course, we shall write our algorithms in Java, but they could also be expressed (albeit with many differences) in almost every other programming language.

A wonderful thing about programs is that they can be executed on a computer, which automatically manages all the details of carrying out the computational process specified by the program. But programs also have meaning independent of their ability to be executed on a computer; they can also be used to communicate "how to" knowledge between people. As noted by Abelson and Sussman:

> A computer language, from this perspective, is a novel formal medium for expressing ideas about methodology, not just a way to get a computer to perform operations. Programs are written for people to read, and only incidentally for machines to execute.

1.2 Problem Solving

Writing a program is a classic problem solving activity. You are given a goal (the program should have a specified behavior) and a set of materials/tools (the programming language and its associated libraries) and you try to achieve the goal with the given materials. There are a number of problem solving techniques that we will study for bridging the gap between the given materials and the desired goal. While these techniques are particularly applicable to computer science, they are helpful in any problem solving activity.

We shall study the following techniques:

- *Divide/Conquer/Glue*

  An important strategy for deriving a solution S for a problem P is to *divide* P into n subproblems $P_1, P_2, \ldots P_n$, *conquer* (i.e. solve) the subproblems to yield n subsolutions $S_1, S_2, \ldots S_n$, and *glue* these subsolutions together to form the whole solution S.

  For example, if you want to bake a chocolate turtle cake you need to solve the following subproblems:

  - Find the recipe;
  - Make sure you have all the ingredients;
  - Follow the recipe to make the cake.

  Each of the above subproblems can be decomposed further into subsubproblems. For instance, to make the cake you need to (1) make two cake layers (2) make a caramel/pecan filling and (3) make a chocolate frosting. These components are physically combined to yield the cake.

There are often many different ways to solve a particular subproblem. For instance, you could buy the cake layers from a store, or bake them from a cake mix, or bake them from scratch.

Eventually, the problem division process "bottoms out" at problems that are so small and simple that they can be solved directly. These "leaves" at the ends of the problem-solving "tree" are called *primitives.* What seem like primitives to us often hide problem-solving complexity for others. For instance, if we buy cake layers at a store, then the problem of baking the cake layers has been pushed off to someone else.

- *Recursion*

  It is often the case that the suproblems for a problem are just smaller versions of the whole problem. In this case, the general divide/conquer/glue strategy is called *recursion.* Recursion is an incredibly powerful problem solving technique of which we shall see many examples in this course.

- *Iteration*

  A common special case of recursion is when the problem decomposes into a single smaller subproblem and there is no glue step. In this case, the general divide/conquer/glue strategy is called *iteration.* Because this is such a common case, Programming languages (including Java) have special looping constructs for expressing iterations.

- *Debugging*

  Programming is an iterative process, and things almost never go right the first time. An important part of problem solving is figuring out why things don't work and fixing them. Program errors are usually referred to as *bugs*, and the process of finding and fixing bugs is called *debugging.* Programmers often spend more time debugging their programs than they do writing them in the first place, so it is important to acquire good debugging skills. Throughout these notes, we will introduce numerous techniques for finding bugs and for preventing them in the first place.

1.3 Controlling Complexity: Abstraction and Modularity

Computer programs can be very large. For instance, the Windows XP operating system contains over 40 million lines of code. This is a lot of code! Printing out all the code with 50 lines per page, double-side, on standard printer paper would produce a pile of paper over 130 feet high. If you were to spend 8 hours per day, seven days a week, reading the code at one line per second, it would take nearly 4 years to read through 40 million lines – and this doesn't include any time for actually *understanding* the code!

Clearly, for programs the size of Windows XP, it is impossible for a single human being to understand every detail of the program. To create such a large and complex software artifact, it must be the case that individual programmers can contribute to the project without understanding every detail of the project. In software engineering (indeed, in any engineering discipline) this is made possible by techniques for controlling the complexity of large systems. The two most important techniques are abstraction and modularity:

- *Abstraction* is the principle that idioms should be captured and generalized into "black box" entities with simple interfaces. Here, *black box* serves as a metaphor for an *abstraction barrier* that separates the users of the device (who should be able to use the device without understanding how it works) from the implementers of the device (who must understand the details of how the device works).

  Abstraction is ubiquitous in the modern world and we depend on it for functioning in our day-to-day lives. We are able to use a wide array of machines and devices (e.g. cars, telephones, stereos, computers) without having to understand the details of how they work. Supermarkets, department stores, and utilities are purveyors of abstractions; for the most part, we do not need or want to know how a loaf of bread is baked, how a piece of clothing is made, or how our electricity, water, and gas are produced.

- *Modularity* is the principle that systems should be composed out of reusable mix-and-match parts. Having a large collection of parts that can be glued together in standard ways simplifies designing, building, debugging, and extending complex systems. Good examples of modularity include:

  – standard connectors for electronic devices (e.g., power cords, telephone cords, network cables, bus connectors for computer peripherals);
  – construction toys (e.g., LEGO, K'NEX, TinkerToys);
  – mix-and-match color-coordinated clothing.

In this course, we shall see that abstraction and modularity are indispensible for simplifying the process of programming. In particular:

- Capturing computational recipes via Java methods and data abstractions via Java objects will help us to solve some difficult problems.

- We can leverage the work of other programmers by using *libraries* of code that they have written, tested, and debugged. As long as they provide *contracts* that explain how to use their libraries, we can incorporate their libraries into our programs without understaning how they are implemented.

- Java objects and standard data structures like lists and arrays facilitate the construction of generic solutions to common problems that can be used in mix-and-match ways.

1.4 Models

Unlike natural languages, programming languages have statements and expressions with extremely precise interpretations. In order to write Java programs effectively, it is essential to have a detailed understanding of the structure and meaning of Java programs. Throughout these notes, we will introduce and emphasize the use of several *models* that explain important aspects of the execution of Java programs. These visually-oriented models are high-level and self-contained; they do not require any understanding of computational hardware.

In particular, we shall study the following models:

- *Syntax trees* represent the abstract tree structure of Java declarations, statements, and expressions.
- *Object diagrams* show the state of Java objects at a given instant in time.
- The *Java Execution Model (JEM)* explains the meaning of statements and expressions, particularly the meaning of method invocations.
- *Invocation trees* summarize the dynamic structure of method invocations in the Java Execution Model.