Conceptual building blocks

In future chapters you will learn about various ideas and the particular Swift programming language words that you use to express those ideas in a program. Other programming languages will express those ideas differently than Swift, just like different spoken languages would have different words for the same objects or concepts. However, just as the idea of "tree" is the same whether the word you use to represent the idea is the English word "tree", or the Spanish word for that concept, or the Japanese word for that concept, likewise, the computational ideas we will learn in this course will be the same, regardless of what language we happen to be using to express those ideas. In this course, we will use Swift to express our ideas, and if this is the first time you've done any programming, it might be easy to think we are learning Swift ideas instead of general programming ideas. That is not the case. We are learning general programming ideas which you will find in almost all modern programming languages, and the particular symbols which represent those ideas in Swift, won't necessarily be the same as the particular symbols that represent those ideas in other languages.

All computer programs basically consist of two components: (1) obtaining information and (2) running instructions that work with, process, and/or display that information. In computer science, the information we gather, manipulate and display is referred to as **data**.

With this broad characterization in mind, there are three conceptual ideas we can introduce early on that you will use as you design programs throughout your programming career.

1. primitives: Primitives are the lowest-level data values and procedures provided by a programming language. Primitives are not things you need to define, nor are they customizable, since they are built right into the particular programming language. As 'primitives' they can not be broken down into smaller parts without changing their meaning. (The concept of a primitive is similar to that of an "atom" in chemistry, that is, "a basic building block".)

- ***Information primitives***: Here, the primitives are built-in data values that your program can use without any further definition. For example, you can type numbers such as 5, -2 or 38.6 directly into your Swift program. The Swift language is designed to understand numerical values like these automatically. Similarly, characters such as a lowercase 'h' or the '@' symbol, are understood by the language.

- ***Procedural primitives***: When talking about procedures, primitives are small, basic operations such as simple arithmetic. Adding two numbers, multiplying two numbers, subtraction and division, would all be considered primitive operations – they are built into the language and you can use them without any further definition.

2. composition: taking smaller entities, and putting them together to build a larger entity.

- ***Data composition***: building larger pieces of information by collecting together smaller pieces of information. A "user record" in Facebook would include a name, email address, birthdate, a list of friends (which are just other user records), a list of likes (which might be "user records", "event records" or "product records"). The chunk of data known as a "user record", is in reality composed of many smaller pieces of data such as your name, email address, and so on. The user record is a composition of other, smaller data values, which together form a larger, more complex piece of data. These smaller components may themselves be simpler composites of information or they may be primitive data entities.

- ***Procedural composition***: building larger procedures out of smaller ones. For example, it is possible to add two numbers together, and it is possible to divide a number by four. Let's apply the first procedure three times adding four numbers together and then apply the second procedure (division by four) to get an answer. Given four numbers:
  - (a) add the first two numbers together
  - (b) add the sum from the previous step to the third number
  - (c) add the sum from the preyious step to the fourth number
  - (d) divide the sum from the previous step by four

  Show what this might look like in a mathematical expression: {textbox}

We just applied some small, primitive procedures – addition and division - to produce an average of the four numbers we were given. By doing so, we have effectively constructed a procedure for finding the average of four numbers. We were able to produce a new, useful procedure that is slightly more complex, by combining primitive mathematical operations in a new way. This more complex procedure is a composition of other, smaller procedures that together form a larger, more complex procedure.

3. abstraction: the concept of focussing on the "big picture" of an idea, and ignoring the details. The advantage of abstraction is that worrying about those details of a procedure often bogs us down with many little issues that we don't need to worry about. When you talk to a friend on your cell phone, you might know that the phone is converting sound waves coming from your voice into electrical signals, but you don't really worry too much about that when you speak to your friend. Imagine that every time you wanted to use your phone you had to do the conversion on your own! As an abstraction, you can trust that your phone will perform the conversion properly, forget about it, and just pick up the phone and speak. In abstract terms, we are concerned with the overall purpose of the phone, rather than the details of how that purpose is achieved or implemented. Certainly, someone or something has to handle the details, but that someone or something doesn't have to be you.

We make use of this idea when writing computer programs, as well:

- ***Data abstraction***: The idea of viewing a data composition in terms of what the overall collection of data is trying to represent, rather than focusing on the details of what pieces of data make up the composition. For example, our program might "display a Facebook user record" or "copy a Facebook event record". In abstracting these ideas, we don't need to worry about sending a lot of little pieces of data that compose each user record to the *display* procedure; we simply send one piece of data, a Facebook user record for a certain person, and don't need to worry that a user record is "really" lots of smaller pieces of data.

- ***Procedural abstraction***: A composition of smaller procedures combined to achieve a task. The resulting composition should be considered as a single unit and be referenced in terms of what the overall goal is, rather than viewing it in terms of the smaller procedures that make up the larger one. For example, once we have written a procedure to calculate the average of four values, we can make use of it whenever we have four values and we need to average them. We can send the four values to our "averageOfFour()" procedure, and get the average back. We no longer need to worry about the details of how the average is calculated.

  For example, we can use the abstract composite procedure like this:

  ```
  averageOfFour(72, 68, 67, 71)
  ```

  where Rory McIlroy's golf scores for a tournament were 72, 68, 67, and 71. His average for the four-day tournament will be simply: averageOfFour(72, 68, 67, 71). We do not need to think about what is going on behind-the-scenes, that 72 is added to 68 and 67 and 71 and the sum of the four numbers is divided by four.

What is a data primitive?

Give some other examples of a data primitives below:

Give some other examples of a procedural primitives:

What is mean by composition?

Give some other examples of a data composition:

Give some other examples of a procedural composition below:

What is mean by abstraction?

Give some other examples of a data abstraction:

Give some other examples of a procedural abstraction:

All these ideas fit together to help us in program design. Quite often, we will compose a larger piece of data out of smaller pieces of data, and then focus at that point on the larger data concept rather than worrying about the smaller pieces of data that form the larger one. Sometimes, those smaller pieces of data that form the larger piece of data, will be primitives, and sometimes they will instead be other data compositions we created previously. Ultimately, if you break any piece of data down into small enough pieces, you will find that it is composed of the same limited set of data primitives built into the language. We just choose to focus on the "big picture" when we can, instead of always peeking into a data composition to see the smaller pieces of data from which it is made.

Similarly, we will often compose a larger procedure out of smaller ones and subsequently focus on the larger procedure and what the overall task accomplishes, rather than worrying about all the smaller procedures from which it was constructed. Sometimes, the smaller procedures we use to form the larger procedure will be primitives and sometimes they will instead be other procedural compositions we have previously created. If you break any procedure down into small enough detail however, you will find that it is composed of the same limited set of procedural primitives that the language (and processor) supports. We choose to abstract away the more detailed procedures from which the composite procedure is made.

In the first portion of our course we will learn about many of the built-in data types and procedures available to us in the Swift programming language, and we will learn about the language used in Swift for implementing data composition and abstraction. We will also learn the language and common patterns used to create procedural compositions and abstractions. Finally, we will see that the tool of abstraction is used often in computer science as a way to disregard detail in favor of a big-picture conceptual approach for problem solving.