**Abstract Data Types**

Over the years, computer scientists have developed a variety of strategies for collecting and organizing data. These strategies are known as *abstract data types* (or *ADT*, for short). Each abstract data type describes a different model for storing and manipulating data.

When designing a program or program module, an experienced designer will frequently choose an ADT based on the type of data that will be manipulated and the way in which those data will be processed. The programmer who codes the design then decides how to implement the chosen ADT. An actual implementation of an ADT is called a *data structure*.

In the context of this course, the ADTs are interfaces; they include List<E>, Stack<E>, Queue<E>, PriorityQueue<E>, Set<E>, and Map<K,V>. Since they are interfaces, ADTs cannot actually store data. The objects that are used to actually store and retrieve data are instances of classes that implement these interfaces; that is, they are instances of data structures. For example, as we have [already](https://www.eimacs.com/eimacs/mainpage?epid=E2124394844&cid=162149) seen, the [ArrayList<E>](https://www.eimacs.com/eimacs/mainpage?epid=E1932277674&cid=162149#ArrayListIntro) class implements the List<E> interface and, as you are already aware, ArrayList<E> objects can be used to store and retrieve data.

Notice that all the interfaces we have just mentioned are *generic* in the same sense that ArrayList<E> is a [generic class](https://www.eimacs.com/eimacs/mainpage?epid=E2173579202&cid=162149#TypeParams). The usual convention — for all except Map<K,V> — is to use E as the *formal type parameter*, which in any particular case is replaced by the name of a reference data type. The Map<K,V> interface has two formal type parameters, separated by a comma. The letters K and V are used in this case for reasons that will become obvious once we officially introduce the interface.

The advantage of working with two layers — ADTs and data structures — is that program designers can choose to distance themselves from the actual implementation details of the program they are designing. For example, it is not unusual for a program specification to say "implement a list to store the names and addresses". It may then be left to the programmer to figure out what is the best implementation in this particular case. In addition to simplifying the task for the program designer, the use of ADTs also gives the programmer the freedom to change the implementation later without breaking the program design. This can be important in prototyping, for example, where a programmer may be required to build a program as quickly as possible in order to check that the design actually meets its specifications. In such a situation, programmers often choose a simple or familiar implementation of an ADT (for example, using ArrayList<E> to implement the List<E> ADT), and change to a more efficient (but potentially more complex) implementation only after the prototype has turned out to be successful.

This section of this online course considers the standard ADTs that you are likely to meet early on if you study Computer Science at college or university and the data structures that implement them.