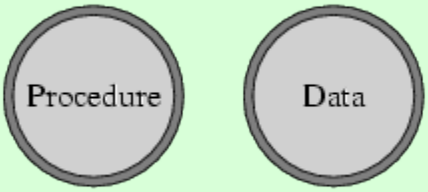
INTRODUCTION TO DATA AND PROCEDURE

In this book, our goal is to understand *computation*. In particular, we want to be able to take any computational problem and produce a technique for solving it that is both correct and efficient.

1.1. Procedure and data

Computation is built of two basic elements: *procedure* and *data*. (See Figure 1.1.) This fact is succinctly summarized by the title of a 1970s computer science textbook, *Data Structures + Algorithms = Programs* (by Niklaus Wirth, from Prentice Hall, 1976). This binary nature is reflected in object-oriented languages such as Java: Here, each *object*has two major categories of components, *instance methods* (i.e., procedures) and *instance variables* (i.e., data).

Figure 1.1: The two sides of the computation coin.



Like energy and mass are physical phenomena united by Einstein's formula $E = m\,c^2$, procedure and data are to some extent two different ways of viewing the same thing. Computer users understand this intuitively: While we often talk about a program being a procedure, in fact a program is text representing the procedure, and that text is just a long string of characters — that is, data. The compiled program, too, is simply a file on disk containing data. We understand this intuitively today, but understanding programs as just another form of data was a breakthrough in the early hisory of computing: In the ENIAC, introduced in 1946 as the first American computer, each task required rewiring the hardware, until in 1948, when the ENIAC was wired to be able to read programs as data from its own memory. This concept, called the von Neumann computer, quickly caught on. (In the reverse direction, Alonzo Church developed the theory of lambda calculus in the 1930s to study the fundamentals of computing. In his system, each integer and indeed every other piece of data is represented as a procedure.)

Any computational problem will involve both data and procedure in its solution. Sometimes the data portion or the procedure portion will be simple; but it is still present. Take, for example, the computational problem of determining whether a number is prime. Before we can begin a procedure for determining this, we must first have the number in question, which after all is data. The

representation of this data is simple but important. You might assume that the number would arrive us in its Arabic representation (e.g., 1003). But the problem is somewhat different if it comes in Roman representation (MIII) or Chinese representation (一千零三).

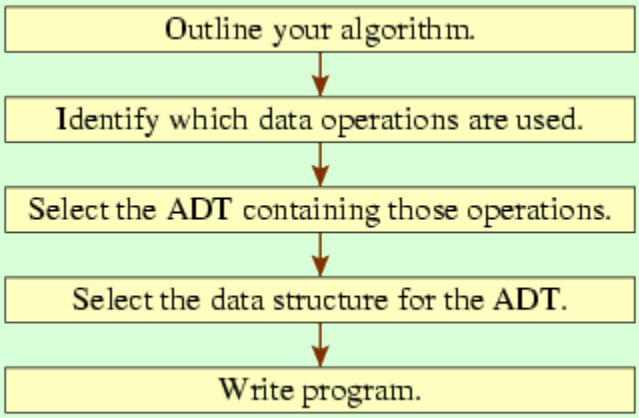
In fact, the representation of data can dramatically affect the efficiency of a procedure. Suppose, for example, if instead of an Arabic representation, the number is instead given us in terms of its prime factorization ($17 \cdot 59$). Then the procedure will be much more efficient: We simply to check whether the prime factorization contains more than one number.

This theme of how data should be represented will form a major portion of this book. We won't spend too much time on simple data like numbers, though: Our focus will be on representing a large collection of data so that it can be processed efficiently. Such techniques for storing data collections are called data structures, and we'll see several throughout the book.

1.1.1. Abstract data types

But before we get to data structures, we first need the concept of an abstract data type (often abbreviated ADT), which refers to a particular set of operations we want to be able to perform on a collection. Given an algorithm to solve a problem using a collection of data, we should be able to identify which operations are necessary on that collection, and then we find the ADT that matches it. Once we know which ADT to use, we can then select the appropriate data structure based on others' work concerning how best to implement the ADT. This program design process is summarized by the following diagram.

Figure 1.2: The steps to developing a data-intensive program.



In this book, we'll examine several ADTs and the data structures commonly used for them. One ADT is the Set ADT, intended for algorithms that use a data collection like a mathematical set. Operations in the Set ADT include:

- `size` to get the number of elements in the set.
- `contains(x)` to test whether a particular element `x` lies within the set.
- `add(x)` to insert an element `x` into the set, returning `true` if `x` is newly added. (As a set, we wouldn't add `x` if it were already present.)
- `remove(x)` to remove an element `x` from the set.

This ADT is quite useful; our study of it will be deferred, though, to Chapter 5.

Another ADT that we'll study is the List ADT, where elements are stored in a particular order. The List ADT contains the following operations.

- `size` to get the number of elements in the list.
- `add(x)` to add a value `x` to the end of a list.
- `get(i)` to fetch the element at index `i` of the list.
- `set(i, x)` to change the element at index `i` of the list to be `x` instead.
- `add(i, x)` to insert an element `x` into the list at index `i`.
- `remove(i)` to remove an element at index `i` from the list.

Although the List and Set ADTs are similar, they have some important differences. With the List ADT, each element is considered to be at a particular index, whereas the Set ADT has no notion of ordering; on the other hand, the Set ADT includes a `contains` operation that is absent from the List ADT.

In the course of this book, we'll study both of these ADTs along with others that people have found useful over the years. Our primary emphasis, though, will be on the best data structures to use for implementing them. This understanding enables writing efficient algorithms for many important problems. We'll study a sampling of these problems in the course of this book.

1.1.2. Interfaces

You might notice that the ADT concept resembles Java's concept of *interface*: Both are abstract sets of operations. The concepts are not entirely equivalent, though. One difference is that the ADT concept is intended to be language-neutral; the language doesn't need a construct similar to Java interfaces for the ADT concept to be useful. Another difference is that ADTs are really meant only for large masses of data, whereas Java programs use interfaces for other purposes, too.

If we're programming in Java, though, it's easy enough to define an ADT in terms of an interface.

```
public interface Set<E> {
    public int size();
    public boolean contains(E value);
```

```
    public boolean add(E value);
    public E remove(E value);
}

public interface List<E> {
    public int size();
    public boolean add(E value);
    public E get(int index);
    public E set(int index, E value);
    public void add(int index, E value);
    public E remove(int index);
}
```

Indeed, these interfaces are already built into Java's java.util package — but they include many more methods beyond those listed above, too. (In this book, we'll study many of the interfaces and classes defined in the java.util package. Figure 1.3 lists all of those we'll study.)

Figure 1.3: Parts of the java.util package appearing in this book.

Interfaces	Classes
List	ArrayList
Set	LinkedList
Iterator	TreeSet
ListIterator	TreeMap
Map	HashSet
	HashMap

Perhaps you have not before seen the `<E>` notation used above. These definitions use a feature of Java called *generics*, where we can designate an identifier — `E` in this case — to represent an arbitrary class; that is, the identifier acts something like a variable that stands for a type. When we use the interface, we'll specify what class the variable `E` should be taken to stand for. For List and Set, the individual elements of the collection will be instances of this class. (In fact, the *E* stands for *element*.) As an example of using these generics, if we want to declare a variable `names` to represent a list of strings, we could write the following.

```
List<String> names;
```

We can later create a List of Strings and assign `names` to refer to it. Then, `names`'s `get` method will return a String, since, after all, the `get` method's return type is `E`, and in `names`'s case, `E` refers to the String type.

```
String first = names.get(0);
```

If the notion of generics is unfamiliar to you, then you should read a Java book and become familiar with the concept. (Generics were introduced with Java 5; you won't find much information about the concepts in books preceding its release date of Summer 2004. Previous to Java 5, the java.util classes worked with Objects.)

Before continuing, note quickly that the Set and List interfaces contain several methods in common. The interface designers carefully named them the same so that programmers could remember them more easily. One peculiar thing about the commonality is the `add` method. In the Set interface, the method returns a **`boolean`**, so that it can indicate whether the value needed to be added to the set. (A set doesn't include any elements twice.) Just to be consistent, the designers made List's `add` method return a **`boolean`** also, but in fact`add`'s return value will always be **`true`** with a List.

To use the List interface, we need some way of getting at List objects. Of course, since List is an interface, we can't write new List to create List objects in a program. We can only create List objects through creating instances of an implementation of the List interface. We'll see one such implementation next: the ArrayList class. In the next chapter, we'll see another implementation, the LinkedList class.

Source : http://www.toves.org/books/data/ch01-intro/index.html