**Introduction to Objects**

This chapter will introduce you to the basic concepts of OOP, including an overview of development methods. This chapter, and this book, assumes that you have some programming experience, although not necessarily in C. If you think you need more preparation in programming before tackling this book, you should work through the Thinking in C multimedia seminar, downloadable from.

**The progress of abstraction**

All programming languages provide abstractions. It can be argued that the complexity of the problems you’re able to solve is directly related to the kind and quality of abstraction. By “kind” I mean, “What is it that you are abstracting?” Assembly language is a small abstraction of the underlying machine. Many so-called “imperative” languages that followed (such as FORTRAN, BASIC, and C) were abstractions of assembly language

**Everything is an object.**

Think of an object as a fancy variable; it stores data, but you can “make requests” to that object, asking it to perform operations on itself. In theory, you can take any conceptual component in the problem you’re trying to solve (dogs, buildings, services, etc.) and represent it as an object in your program.

**A program is a bunch of objects telling each other what to do by sending messages**.

To make a request of an object, you “send a message” to that object. More concretely, you can think of a message as a request to call a method that belongs to a particular object.

**Each object has its own memory made up of other objects.**

Put another way, you create a new kind of object by making a package containing existing objects. Thus, you can build complexity into a program while hiding it behind the simplicity of objects.

**Every object has a type.**

Using the parlance, each object is an instance of a class, in which “class” is synonymous with “type.” The most important distinguishing characteristic of a class is “What messages can you send to it?”

**All objects of a particular type can receive the same messages**.

This is actually a loaded statement, as you will see later. Because an object of type “circle” is also an object of type “shape,” a circle is guaranteed to accept shape messages

**An object has an interface**

Aristotle was probably the first to begin a careful study of the concept of type; he spoke of “the class of fishes and the class of birds.” The idea that all objects, while being unique, are also part of a class of objects that have characteristics and behaviors in common was used directly in the first object-oriented language, Simula-67, with its fundamental keyword class that introduces a new type into a program.

**An object provides services**

While you’re trying to develop or understand a program design, one of the best ways to think about objects is as “service providers.” Your program itself will provide services to the user, and it will accomplish this by using the services offered by other objects

**The hidden implementation**

It is helpful to break up the playing field into class creators (those who create new data types) and client programmers4 (the class consumers who use the data types in their applications). The goal of the client programmer is to collect a toolbox full of classes to use for rapid application development.

**Reusing the implementation**

Once a class has been created and tested, it should (ideally) represent a useful unit of code. It turns out that this reusability is not nearly so easy to achieve as many would hope; it takes experience and insight to produce a reusable object design. But once you have such a design, it begs to be reused. Code reuse is one of the greatest advantages that object-oriented programming languages provide.

**Inheritance**

By itself, the idea of an object is a convenient tool. It allows you to package data and functionality together by concept, so you can represent an appropriate problem-space idea rather than being forced to use the idioms of the underlying machine. These concepts are expressed as fundamental units in the programming language by using the class keyword.