Java OO Concepts Unit

Lesson 01.1: Object-Oriented Concepts

SOFTWARE GUILD

Lesson 01.1: Object-Oriented Concepts

Overview

In this portion of Lesson 01, we are going to look at some general object-oriented concepts. We will explore different ways of abstracting problems, define object orientation (one way to abstract problems), describe the characteristics of an object-oriented language, and discuss the concept of public interface vs. private implementation.

Abstraction

There are many ways to approach a particular problem that you are trying to solve. One approach is to start with the model of the solution space and then attempt to map the problem into it. This is essentially the approach of many "low-level" languages such as assembler language and C. These languages closely match the underlying computer — if you want to solve a particular problem, you must think in terms of how the computer works first and then figure out how to map the problem into that paradigm. Other approaches to abstraction include viewing all problems as mathematical functions (as in functional programming languages) or viewing all problems as relations, facts, and rules (as in logic programming languages).

Object-oriented languages take a different approach in that they represent concepts in both the solution space and the problem space as objects. This works well for modeling things in the real world because our world is filled with objects and is essentially object-oriented.

Think about a car — it can be described as a collection of properties (weight, color, number of doors, etc.) and behaviors (drive, turn, roll up window, turn on radio, etc.). That's how objects are modeled in an object-oriented language: via properties and behaviors.

Object Orientation

So what makes a language object-oriented? Alan Kay summarized it like this:

Everything is an object.

1. A program is a collection of objects telling each other what to do by sending messages (in Java's case, these messages are methods calls).
2. Each object can be made up of other objects (this is called composition in Java).
3. Every object has a type.
4. All objects of a particular type can receive the same messages (in Java this means that they all have the same methods).

Grady Booch put it this way: An object has state, behavior, and identity.

Types

Every object in Java has a type. So far we have mostly looked at the native or intrinsic Java type (int, float, boolean, etc.) but we are also free to create our own types in Java. In fact, we have been creating our own types all along — every time we define a new Java class, we define a new type.

Public Interface / Private Implementation

Every class should have a public interface that defines how the outside world can interact with it. Behind this public contract should be a private implementation. This allows us to separate the "what" an object does from the "how" it does it. A client (the code that uses the object)

should not be concerned with how an object fulfills the contract and should in no way ever rely on the specifics of the implementation (or side effects of a particular implementation) when using the object. The implementer of the object reserves the right to change the implementation details at his/her discretion.

Encapsulation

A well designed class has a well-defined area of responsibility. Generally speaking, this means that the class does one thing, does it completely, and does it well. This is known as **encapsulation**. The class should fully contain all aspects of its area of responsibility. The public interface of the class must be defined so that its function is crystal clear (even though how it is implemented is hidden).

Data Hiding

Data hiding is a concept that goes right along with encapsulation. Data hiding simply means that the internal state of the object is hidden from the users of that object.

Delegation

The concept of delegation is complementary to encapsulation. If our class is well-encapsulated, it will only handle tasks that are within its well-defined area of responsibility. If one or more of the tasks within the class's area of responsibility require a subtask that is outside of the class's main area of responsibility, the class must **delegate** that task to another class. We have already seen examples of this in our code — we delegate to System.out for writing to the Console and to the Scanner object for reading from the Console.