Lesson 01: Object-Oriented Concepts

Overview

In this lesson, 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.

Objectives

By the end of this lesson, you will be able to:

1. Differentiate between various types of computer programming languages.
2. Define the five things that make a language object-oriented.
3. Explain what a type is and how it is used in Java.
4. Explain how public interface and private implementation work together.
5. Describe loose coupling.
6. Explain the Single Responsibility Principle and Cohesion.
7. Explain the concept of delegation.
8. Relate delegation to encapsulation.

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 the approach of many languages.

Assembly language closely matches the underlying computer mechanics — 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. This means that you have to think about problems in terms of binary numbers, registers, addition, and subtraction. This requires you to understand how to do things like move data into registers, decrement the value in a register, and use various processor operations. Solving a problem with assembly language requires a thorough knowledge of the actual processor being used.

Functional languages model everything as mathematical functions and immutable data structures. Here, everything is a function that takes inputs and produces values or does work. This means that you have to think about the problem in terms of mathematical functions, inputs, outputs, and data structures.

Logic programming languages model all problems as relations, facts, and rules. These languages are based on formal logic and require the programmer to map the problem domain into facts and rules in these languages.

Object-oriented languages take a different approach in that they represent concepts in both the solution space and the problem space as objects. Since the real world is essentially full of objects, this is very convenient. Take a car for example — 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.). This is how we model objects in an object-oriented language: via properties and behaviors. Less translation is needed to map a car into one of these objects than is needed to map a car into some of the approaches taken by non-object-oriented languages.

Object Orientation

So what makes a language object-oriented? **Alan Kay (https://en.wikipedia.org/wiki/Alan_Kay)** summarized it like this:

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

Grady Booch (https://en.wikipedia.org/wiki/Grady_Booch) put it more simply: An object has state, behavior, and identity.

Let's break this down:

State: This means that the properties of an object have certain values at certain times. For example, a car might have a velocity of 50 miles per hour right now, but 0 miles per hour in 10 seconds. The combination of these values at a given point in time describes the object's state.

Behavior: This means that an object has some capacity to do something. In Java, these capabilities are represented as methods. Many times, these methods change the state of the object.

Identity: Here we are talking about the ability to distinguish one object from another even when the objects are the same type. For example, I can tell the shade tree in my front yard from the one in my back yard.

Types

Every object in Java has a type. A **type** is a classification that defines the structure and range of values for the type and the associated operations allowed on those values. On the one hand, there are native or intrinsic Java types (int, float, boolean, etc.), but we are also free to create our own types in Java. In fact, every time we define a new Java class, we define a new type! You may also have some experience using non-native types that are part of the JDK but not part of the Java language itself, such as Random and Scanner. In later lessons in this section, we'll look at the details of how to create our own data types and how to create programs that contain several data types cooperating to solve problems.

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 "what" an object does from "how" it does it. Calling code (the code from other parts of the program 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.

Let's look at an example from our everyday lives that illustrates why this concept is so important: the fast food drive-through. The public interface to a drive-through is familiar to all of us: first there is the menu displaying items and prices, next is the speaker where we place our order, next comes the window where we pay, and finally the window where we get our food. As a customer of the restaurant (and user of drive-through public interface), I have no idea about how my order is processed, how and when the ingredients are delivered, how the food is cooked, or how many cooks are in the kitchen — and frankly, I really don't care. I just want to order my food, pay for it, and enjoy my meal. The restaurant is free to upgrade its ordering system, get new stoves, hire more cooks, or make any other changes to their system and as long as the drive-through works as it did before and the food tastes the same, I'm a happy customer.

Encapsulation and Data Hiding

One way to help facilitate the notion of public interface and private implementation is through **encapsulation** and **data hiding**. Well-designed classes prevent direct access to their properties by calling code (remember, calling code is code from other parts of the program that use the object). Instead, they force this access through getter and setter methods. This prevents the calling code from being aware of the internal details of the object. This allows the internal representation of the properties to change without the knowledge of the calling code. This technique leads to **loose coupling** between the calling code and the object.

Single Responsibility Principle and Cohesion

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 means that a class is **cohesive** and follows the **single responsibility principle**. 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).

Let's return to our drive-through example. If the drive-through interface (i.e., the menu, order speaker, payment window, and pickup window) is to be cohesive, it must allow us to do everything involved in ordering, paying for, and picking up our meal. For example, the interface would not be cohesive if at the payment window, I had to get out of my car, walk over to the bank, transfer funds from my account to the restaurant, and then return to my car. On the other hand, the drive-through should be limited to just ordering and paying for food. For example, I shouldn't have the option of renewing my driver's license at the payment window — this is clearly outside the scope of what a fast food drive-through should do.

Although this is just an example, you can see how cohesive interfaces make sense. These principles apply to objects just as they apply to drive-through restaurants.

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.

For example, in the case of the drive-through, they don't specialize in baking bread, so they "delegate" this responsibility to a bakery that bakes the buns and delivers them to the restaurant. I, however, as the consumer of the restaurant interface, do not care whether the buns are baked on site or at a bakery and delivered. As long as the buns are fresh, taste good, and have the correct nutritional content, I am a happy customer. The restaurant specializes in putting together the finished hamburger. They delegate things like baking the buns and processing the meat to other companies.

Summary

This was a quick introduction to some of the big concepts of object-oriented programming. The remaining lessons in this section cover these topics in detail and will set you on the path to becoming an object-oriented programmer.

Video: Object Oriented Concepts

Time: 5:18

Transcript-Object-Oriented-Concepts.txt