Table of Contents

Preface

This module is one in a series of modules designed to teach you about the essence of Object-Oriented Programming (OOP) using Java with particular emphasis on accessibility for blind students.

Prerequisites

As mentioned in an earlier module, in addition to an Internet connection and a browser, you will need the following tools *(as a minimum)* to work through the exercises in this and the following modules:

- An audio screen reader that is compatible with your operating system, such as the NonVisual Desktop Access program (NVDA), which is freely available at http://www.nvda-project.org/.
- A refreshable Braille display capable of providing line by line tactile output of information displayed on the computer monitor is recommended. Such a display is described at http://www.userite.com/ecampus/lesson1/tools.php, is recommended.
- The Oracle Java Development Kit (JDK) *(See http://www.oracle.com/technetwork/java/javase/downloads/index.html)*
- Documentation for the Oracle Java Development Kit (JDK) *(See http://download.oracle.com/javase/8/docs/api/)*
- A simple IDE or text editor for use in writing Java code.

The minimum prerequisites for understanding the material in this and the following modules include:

- A cursory understanding of algebra.
- An understanding of the material covered in the *Programming Fundamentals* modules that you will find in two formats at the following URLs. These modules provide fundamental programming concepts using the Java programming language in a format that should be accessible.
    - http://cnx.org/content/m45179/latest/?collection=col11441/latest
    - http://cnx.org/contents/fb64661c-5b3f-4ea8-97c6-e48df112438a

The essence of OOP

My dictionary provides several definitions for the word essence. Among those definitions are the following:

- The property necessary to the nature of a thing
- The most significant property of a thing

Thus, this and several modules that follow describe and discuss the necessary and most significant aspects of OOP using Java -- the essence of OOP using Java. For the first few modules, I will provide that information in a high-level format, devoid of any requirement to understand detailed Java syntax. In those cases where an understanding of Java syntax is required, I will provide the necessary syntax information in the form of supplementary notes.

If you understand the fundamentals of computer programming described above, you should be able to read and understand the modules in this miniseries.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

Listings

- Listing 1. Instantiating a new Radio object.
- Listing 2. Calling the playStation method.

Preview

Three important concepts

In order to understand OOP, you need to understand the following three concepts:

- Encapsulation

- Inheritance
- Polymorphism

This module will concentrate on encapsulation. Encapsulation will be used as a springboard for a discussion of objects.

A description of an object-oriented program will be provided, along with a description of an object, and how it relates to encapsulation.

A car radio

I suspect that many of you grew up in a family where the family car contained a push-button radio and I also suspect that you learned how to operate that radio. In fact, as children, many of you may have quarreled with your siblings about which button to push and which radio station to listen to.

In order to relate object-oriented programming to the real world, a car radio will be used to illustrate and discuss several aspects of software objects.

What you will learn

For example, you will learn that car radios, as well as software objects, have the ability to store data, along with the ability to modify or manipulate that data.

You will learn that car radios, as well as software objects, have the ability to accept messages and to perform an action, modify their state, return a value, or some combination of the above.

You will learn some of the jargon used in OOP, including persistence, state, messages, methods, and behaviors.

You will learn where objects come from, and you will learn that a class is a set of plans that can be used to construct objects. You will learn that a Java object is an *instance* of a class.

You will see a little bit of Java code, used to create an object, and then to send a message to that object *(call a method on the object)*.

You will learn about Java references and reference variables. You will also learn a little about memory allocation for objects and variables in Java.

Discussion and sample code

Purpose of the miniseries

As mentioned earlier, I will describe and discuss the necessary and most significant aspects of OOP using Java.

The three pillars

Most books on OOP will tell you that in order to understand OOP, you need to understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

I agree with that assessment.

Begin with encapsulation

Generally, speaking, these three concepts increase in difficulty going down the list from top to bottom. Therefore, I will begin with Encapsulation and work my way down the list in successive modules.

What is an Object-Oriented Program?

Many authors would answer this question something like the following:

An Object-Oriented Program consists of a group of cooperating objects, exchanging messages, for the purpose of achieving a common objective.

What is an object?

An object in OOP is a software construct that encapsulates data, *(along with the ability to use or modify that data)*, into a software entity.

What is encapsulation?

An interesting description of encapsulation was provided in an article by Rocky Lhotka regarding VB.NET. That description reads as follows:

"Encapsulation is the concept that an object should totally separate its interface from its implementation. All the data and implementation code for an object should be entirely hidden behind its interface.

The idea is that we can create an interface (Public methods in a class) and, as long as that interface remains consistent, the application can interact with our objects. This remains true even if we entirely rewrite the code within a given method thus the interface is independent of the implementation."

I like this description, so I won't try to improve on it. However, I will try to illustrate it in the paragraphs that follow.

A real-world analogy

Abstract concepts, *(such as the concept of an object or encapsulation)*, can often be best understood by comparing them to real-world analogies. One imperfect, but fairly good analogy to a software object is the push-button radio in a car.

The ability to store data

Most car radios have the ability to store data, and to allow you to use and modify that data at will. *(However, you can only use and modify that data through use of the human interface that is provided by the manufacturer of the radio.)*

The data that can be stored in a car radio includes a list of five or more frequencies that correspond to your favorite radio stations.

Using the stored data

The radio provides a mechanism *(human interface)* that allows you to use the data stored therein.

When you press one of the frequency-selector buttons on the front of the radio, the radio automatically tunes itself to the frequency corresponding to that button. *(In this case, you, the user, are sending a message to the radio object asking it to perform a particular action.)*

If you have previously stored a favorite frequency in the storage location corresponding to that button, pressing the button *(sending the message)* will cause the radio station transmitting at that frequency to be heard through the radio's speakers.

If you have not previously stored a favorite frequency in the storage location corresponding to that button, you will probably only hear static. *(That doesn't mean that the radio object failed to respond correctly to the message. It simply means that its response was based on bad data.)*

Modifying the stored data

The human interface also makes it possible for you to store or modify those five or more frequency values. This is done in different ways for different radios. On my car radio, the procedure is:

- Manually tune the radio to the desired frequency.
- Press one of the buttons and hold it down for several seconds.

When the radio beeps, I know that the new frequency value has been stored in a storage location that corresponds to that particular button.

Please change your state

By following this procedure, I "send a message" to the radio object asking it to "change its state". The beep that I hear could be interpreted as the radio object returning a value back to me indicating that the mission has been accomplished. *(Alternately, we might say that the radio object sent a message back to me.)*

We say that an object has changed its state when one or more data values stored in the object have been modified.

We also say that when an object responds to a message, it will usually perform an action, change its state, return a value, or some combination of the above.

Please perform an action

Following this, when I press that button *(send a message)*, the radio object will be automatically tuned to that frequency.

Historical note:

While the ability to cause your car radio to remember your list of favorite stations may seem like a miracle of modern digital electronics, the truth is that radios had this capability long before they contained digital electronics. My first car had a radio that accomplished this feat using strings, pulleys, and levers. In fact, the radio in the first car that I owned many years ago is the first programmable device of any complexity that I can remember.

As I recall, in order to set the frequency for a button, I had to manually tune the radio to a station by turning a knob, pull one of the buttons out about a quarter of an inch, and then push it in again. From that point until I did the same thing again, whenever I pressed that button, some kind of a mechanical contraption caused a big rotary capacitor to turn just the right amount to tune for a particular radio station.

Also, I remember my grandfather having a table-model radio in the early 1940's that had radio buttons. He used them to select his favorite stations, as he surfed the airwaves.

(Interestingly, the term radio button has now become a part of programming jargon, signifying certain visual components used in graphical user interfaces.)

Enough of that, now back to my modern car radio

If I drive to Dallas and press a button that I have associated with a particular radio station in Austin, I will probably hear static. In that case, I may want to change the frequency value associated with that button. I can follow the same procedure described earlier to *set* the frequency value associated with that button to correspond to one of the radio stations in Dallas. *(Again, I would be sending a message to the radio object asking it to change its state.)*

Jargon

As you can see from the above discussion, the world of OOP is awash with jargon, and the ability to translate the jargon is essential to an understanding of the published material on OOP. Therefore, as we progress through this series of modules, I will introduce you to some of that jargon and try to help you understand the meaning of the jargon.

Persistence

The ability of your car radio to remember your list of favorite stations is often referred to as persistence. An object that has the ability to store and remember values is often said to have persistence.

State

It is often said that the *state* of an object at a particular point in time is determined by the values stored in the object. In our analogy, even if we own identical radios, unless the two of us have the same list of favorite radio stations, associated with the same combination of buttons, the state of your radio object at any particular point in time will be different from the state of my radio object.

Identical objects with identical states:

It is perfectly OK for the two of us to own identical radios and to cause the two radio objects to contain the same list of frequencies. Even if two objects have the same state at the same time, they are still separate and distinct objects. While this is obvious in the real world of car radios, it may not be quite as obvious in the virtual world of computer programming.

Sending a message

A person who speaks in OOP-speak might say that pressing one of the frequency-selector buttons on the front of the radio sends a message to the radio object, asking it to perform an action *(tune to a particular station)*. That person might also say that storing a new frequency that corresponds to a particular button entails sending a message to the radio object asking it to change its state.

Calling a method

Java-speak is a little more specific than general OOP-speak. In Java-speak, we might say that pressing one of the selector buttons on the front of the radio *calls a method* on the radio object. The behavior of the method is to cause the object to perform an action.

As a practical matter, the physical manifestation of sending a message to an object in Java is to cause that object to execute one of its methods.

Similarly, we might say that storing a new frequency that corresponds to a particular button calls a *setter* method on the radio object.

(In an earlier paragraph, I said that I could follow a specific procedure to set the frequency value associated with a button to correspond to one of the radio stations in Dallas. Note the use of the words set and setter in this jargon.)

Behavior

In addition to state, objects are often also said to have *behavior*. The overall behavior of an object is determined by the combined behaviors of its individual methods.

For example, one of the behaviors exhibited by our radio object is the ability to play the radio station at a particular frequency. When a frequency is selected by pressing a selector button, the radio knows how to translate the radio waves at that frequency into audio waves compatible with our range of hearing, and to send those audio waves out through the speakers.

Thus, the radio object behaves in a specific way in response to a message asking it to tune to a particular frequency.

Where do objects come from?

In order to mass-produce car radios, someone must first create a set of plans, *(drawings, or blueprints)* for the radio. Once the plans are available, the manufacturing people can produce millions of nearly identical radios.

A class definition is a set of plans

The same is true for software objects. In order to create a software object in Java, it is necessary for someone to first create a plan.

In Java, we refer to that plan as a *class*.

The class is defined by a Java programmer. Once the class definition is available, that programmer, *(or other programmers)*, can use it to produce millions of nearly identical objects.

(While millions may sound like a lot of objects, I'm confident that since Java was released into the programming world around 1997, Java programmers around the world have created millions of objects using the standard Java class named ***Button****.)*

An instance of a class

If we were standing at the output end of the factory that produces car radios, we might pick up a brand new radio and say that it is an *instance* of the plans used to produce the radio. *(Unless they were object-oriented programmers, the people around us might think we were a little odd when they hear us say that.)*

However, it is common jargon to refer to a software object as an *instance of a class*.

To instantiate an object

Furthermore, somewhere along the way, someone turned the word instance into a verb, and it is also common jargon to say that when creating a new object, we are *instantiating* an object.

A little bit of code

It is time to view a little bit of Java code.

Assuming that you have access to a class definition, there are several different ways that you can create an object in Java. The most common way is using syntax similar to that shown in Listing 1 below.

Listing 1. Instantiating a new Radio object.

```
Radio myObjRef = new Radio();
```

What does this mean?

Technically, the expression on the right-hand side of the equal sign in Listing 1 applies the ***new*** operator to a *constructor* for the class named **Radio** in order to cause the new object to come into being and to occupy memory.

(Suffice it at this point to say that a constructor is code that assists in the creation of an object according to the plans contained in a class definition. The primary purpose of a constructor is to provide initial values for the new object, but the constructor is not restricted to that behavior alone.)

A reference to the object

The right-hand expression in Listing 1 returns a *reference* to the new object.

What can you do with a reference?

The reference can later be used to send messages to the new object *(call methods belonging to the new object)*.

Saving the reference

In order to use the reference later, it is necessary to save it for later use.

The expression on the left-hand side of the equal sign in Listing 1 declares a variable of the type **Radio** named **myObjRef**.

(Because this type of variable will ultimately be used to store a reference to an object, we often refer to it by the term ***reference variable****.)*

What does this mean?

Declaring a variable causes memory to be set aside for use by the variable. Values can then be stored in that memory space and accessed later by calling up the name given to the variable when it was declared.

Assignment of values

The equal sign in Listing 1 causes the object's reference returned by the right-hand expression to be assigned to, or saved as a value in, the reference variable named **myObjRef** *(created by the left-hand expression)*.

Memory allocation

Once the code in Listing 1 has finished execution, two distinct and different chunks of memory have been allocated and populated.

One *(potentially large)* chunk of memory has been allocated *(by the right-hand expression)* to contain the object itself. This chunk of memory has been populated according to the plans contained in the definition of the class named **Radio**.

The other chunk of memory is a relatively small chunk allocated *(by the left-hand expression)* for the reference variable containing the reference to the object.

Calling a method on the object

Assume that the definition of the **Radio** class defines a method with the following format *(also assume that this method is intended to simulate pressing a frequency-selector button on the front of the radio)*:

public void playStation(int stationNumber)

What does this mean?

Generally, in our radio-object context, this format implies that the behavior of the method named **playStation** will cause the specific station identified by an integer value passed as **stationNumber** to be selected for play.

Public and void

The *void* return type means that the method doesn't return a value.

The *public* modifier means that the button can be pressed by anyone in the car who can reach it.

(Car radios don't have frequency-selector buttons corresponding to the private modifier in Java.)

The method signature

Continuing with our exposure of jargon, some authors would say that the following constitutes the *method signature* for the method identified above:

playStation(int stationNumber)

A little more Java code

Listing 2 shows the code from the earlier listing, expanded to cause the method named **playStation** to be called.

Listing 2. Calling the playStation method.

```
Radio myObjRef = new Radio();
```

Listing 2. Calling the playStation method.

```
myObjRef.playStation(3);
```

The first statement in Listing 2 is a repeat of the statement from the earlier listing. It is repeated here simply to maintain continuity.

Method invocation syntax

The second statement in Listing 2 is new.

This statement shows the syntax used to send a message to a Java object, or to call a method on that object *(depending on whether you prefer OOP-speak or Java-speak)*.

Join the method name to the reference

The syntax required to call a method on a Java object joins the name of the method to the object's reference, using a period as the joining operator.

*(In this case, the object's reference is stored in the reference variable named **myObjRef**. However, there are cases where an object's reference may be created and used in the same expression without storing it in a reference variable. We often refer to such an object as an anonymous object.)*

Pressing a radio button

Given the previous discussion, the numeric value 3, passed to the method when it is called, simulates the pressing of the third button on the front of the radio *(or the fourth button if you elect to number your buttons 0, 1, 2, 3, 4, 5)*.

Summary

This is the first in a miniseries of modules that describe and discuss the necessary and most significant *(essential)* aspects of OOP using Java.

In order to understand OOP, you need to understand the following three concepts:

- Encapsulation
- Inheritance
- Polymorphism

This module has concentrated on encapsulation. Encapsulation was used as a springboard for a discussion of objects.

A description of an object-oriented program was provided, along with a description of an object, and how it relates to encapsulation.

In order to relate object-oriented programming to the real world, a car radio was used to illustrate and discuss several aspects of software objects.

You learned that car radios, as well as software objects, have the ability to store data, along with the ability to modify or manipulate that data.

You learned that car radios, as well as software objects, have the ability to accept messages and to perform an action, modify their state, return a value, or some combination of the above.

You learned some of the jargon used in OOP, including persistence, state, messages, methods, and behaviors.

You learned where objects come from, and you learned that a class is a set of plans that can be used to construct objects. You learned that a Java object is an instance of a class.

You saw a little bit of Java code, used to create an object, and then to send a message to that object (call a method on the object).

You learned about Java references and reference variables. You learned a little about memory allocation for objects and variables in Java.

What's next?

The next module in the miniseries will introduce you to the java class.

Continuing with the real-world example introduced in this module, the next module will provide a complete Java program that simulates the manufacture and use of a car radio.

Along the way, you will see examples of *(or read about)* class definitions, constructing objects, saving references to objects, setter methods, sending messages to objects, instance variables and methods, class variables, array objects, persistence, and objects performing actions.

Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Objects and Encapsulation
- File: Jbs1010.htm

- Published: 08/12/14

Affiliation: Prof. Baldwin is a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-