Java Foundations

First Steps in Object Oriented Programming

Copyright

Introduction to Java Programming by Eric Matthews is licensed under a Creative Commons Attribution 3.0 Unsupported License. This license allows you to

- Copy, distribute and transmit the work
- Adapt the work
- Make commercial use of the work

Under the following conditions:

You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work

With the understanding that:

Waiver — Any of the above conditions can be waived if you get permission from the copyright holder.

Public Domain — Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

Other Rights — In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to one or more of the following web pages.

www.DeveloperGeekResources.com

www.EducationAnytime.com

Table of Contents

First Steps Into Object Oriented Programming 4

Before we even discuss OOP 4

Before we even discuss OOP 4

OOP's Three Main Ideas Defined 6

Encapsulation...6

Inheritance 9

Polymorphism...9

Java and OOP has a lot of Class 10

What is a class?...10

What is an object? 10

Building our first class 11

Using our first class 12

Getting Input from the Terminal when you run your program...15

constructors 16

Constructor overloading 17

THIS Keyword 19

Method overloading...19

Variables and variable scope 22

Quick inventory 26

FIRST STEPS INTO OBJECT ORIENTED PROGRAMMING

BEFORE WE EVEN DISCUSS OOP

This subject, unlike the previous one, I have a great deal of narrative and offer up a great deal of the my perspective. It is important that you read all of the text, and even more important that you understand the ideas that I am presenting. If you are like me your instinct will be to blow past the text and to get to the examples. I ask that you avoid this instinct and read and understand the text. To do Object Oriented Programming (from here on out referred to as OOP) you must learn to think in OOP. This type of programming is very different than the structured programming most of you are accustomed to.

These ideas and concepts are not easy to understand and they are even harder to master and apply. Do not get discouraged if this subject does not come to you on the first go around. Be persistent and stay focused on learning and internalizing the three tenants of OOP. I learned them and so can you.

BEFORE WE EVEN DISCUSS OOP

This subject is the most critical. Now that you have a good understanding of basic Java syntax it is my intent to walk you through the doors of OOP. This is my slim window of opportunity to draw you in and get you to see the value of this way of programming, and to begin thinking of programming in terms of objects.

For you, the stakes may be even higher. Look at the main languages in vogue today; Java, C++, Php, Java Script. These languages are all object oriented or object based. Granted there is still some Cobol and C around. But folks, the times have been a changing since the turn of the century. Keeping up with technology today takes more than learning the mechanics of a language. You must understand more than technical details today. You must understand ideas and concepts.

I am not going to bash Cobol or C or any of the more mature languages here. In fact, C is still one of my favorite languages and is the esteemed grandfather of C++, Java, Perl, and JavaScript. Cobol was a revolutionary idea in computer science in its heyday. It is unfair and unwise to laugh at the older technologies as it was through these technologies that we were able to evolve.

A main goal of computer science and technology is and has always been to evolve and build a better widget. SQL was created to better deal will entity relationships, something Cobol did not do well. C++, as I said before is an iteration of C, an attempt to build a better C. Java was created to better handle programming tasks that C++ was giving James Gosling (the inventor of Java) fits with. Whether these languages actually achieved their objective is certainly a subject to debate.

I do not want to make this a history lesson about computer science. I do want to stress this one point. Those of us passionate about software engineering have one thing we can generally agree on. Good software engineering at it foundation is about achieving (or knowing when to and when not to achieve) two central principles…

REUSE and COMMUNICATION

One of the best books on this subject is a book titled “The Mythical Man-Month” by Fredrick Brooks. I feel this should be required reading by anyone in our profession. I remember the first time I read this in the mid-nineties thinking he had used my company to do his research. I was floored to find that the book had been written in 1975. It is sad that even today, with all the advancements in computer science, that the issues and problems Brooks spoke of in 1975 are still present today. Why is this?

I am not wise enough to offer a full answer here. Nor would addressing an answer in the full context of that book be appropriate for the subject at hand. I would encourage you to pick-up this book and read it though. It is quite enlightening!

I do want to offer my perspective on reuse and communication. These are concepts I believe to be very important prior to discussing the basics of OOP. One of the failures in software engineering is that we are so focused on writing code that we sometimes forget the plans and organization required to write and maintain good code. You can have a pile of lumber and materials on your building site but without a set of blueprints it is unlikely that you will build anything that comes close to approximating a house. How many times have you brought something home from the hardware store, like a barbecue, looked at the picture on the box and figured, “instructions - I don’t need no stinking instructions”… only to find that you did need them and even then you ended up with a few leftover hardware items. Well software is not any different in this aspect. Because software is less visual under the hood, it is even more important that we define a reasonable blueprint or means of communicating what we have done and why. And if this is so obvious to all of us, why aren’t many organizations taking the time to do it?

As for reuse, I will venture out on a limb and say that every programmer has at sometime in his or her career employed some aspect of reuse. You have no doubt written a piece of code that was very complex and took you a great deal of time and agony to accomplish. Down the road you were faced with solving a similar problem and instead of starting from scratch you used your prior solution as a starting point to solve the new problem. How many times have you cut-n-pasted a routine from one program into another? Cut-n-paste is practicing a form of reuse. Now this is a pretty crude form that is not likely to win you any accolades in the world of computer science. But hey, it is still reuse!

Reuse is not a concept that should be foreign to any of you reading this. You employ reuse because it saves you time and you do not have to reinvent the wheel.

OOP at its most foundational level exists to support code reuse. Reuse is something you are already familiar with so all we really need to delve into is how Java deals with reuse.

There are three important ideas to OOP. They are

- Encapsulation
- Inheritance
- Polymorphism

OOP'S THREE MAIN IDEAS DEFINED

At this point we are only going to discuss these ideas without respect to code. I want you to gain a very high level understanding of these three principles. We will take these ideas throughout this subject and in other parts of the book and expand upon them. It is important that you "buy-into" and embody these three ideas as they are the backbone of OOP. Most stuff I have read on OOP only talks about the benefits of OOP. This bothers me and I plan on addressing the tough questions as well.

Encapsulation

Encapsulation is a principle that allows a software engineer to do a couple of things. First, it allows data and code to live together. This by itself will not be a new concept for most of you. However, if you are a Cobol programmer this is indeed a very new concept. In Cobol, data is defined separate (Data Division) from the program code (Procedure Division). This makes reuse very difficult. By placing data and program code together we have the ability to more easily reuse code. Of course this same principle can be achieved in most other languages. So what is so special about encapsulation? (Keep this question in the back of your hat for the moment).

Second, by grouping data and code together we can potentially hide our code from other programmers, but allow them to reuse it. This sounds a bit unusual so let me further explain how this is done. Suppose I am building an application and want to globally define an address and all the functions one might want to do with the address. I might define the address data as follows:

Addressee
Street1
Street2
City
State
ZIP Code

I might then define the address functions as follows:

- Write an address to the address table
- Retrieve an address from the address table given partial address information
- Given the City and State determine and fill out the zip code
- Given the zip code determine and fill in the state and give the user a list of cities to choose from
- Provide a list of all addressees given a city or zip code
- Check for and prevent duplicate addresses in the address table
- Etc…

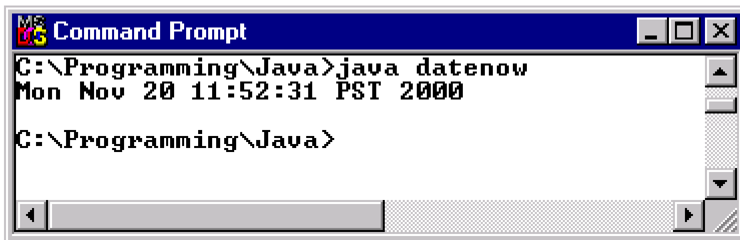
Now further suppose that I want to make this the standard anytime the application needs to deal with address. I also want to keep the code from being changed by just anyone in the shop. These two requirements seem to oppose one another. How can I write code that other developers can reuse but cannot access?

With Encapsulation I can write code whose access is restricted, but I can provide an interface that exposes the functionality of what I wrote so that it can be used by my fellow developers. This may still be a foreign idea so let's look at an example of this.

Source: Datenow.java

```
import java.util.*;

class Datenow
{
  public static void main (String arguments[])
  {
      System.out.println(new Date());
  }
}
```

The following code returns today's date.



By calling the Date() function we can get the current date. In Java (and other OOP languages) what I have actually been calling is referred to as a METHOD. Now that you know that a method is really just a function or subroutine, I will start referring to it from here on out as a method. The example I have used is calling a method that is part of the Java language. There are many such methods for dealing with and manipulating date and time.

Ponder for a moment the concept of encapsulation. It can be quite powerful. I can call some predefined programming service and get a predictable result without having a clue as to how the code is implemented.

I have read a number of books and dissertations on OOP and few of them address what I think are some of the most important things with respect to this subject.

1. **How do I know what methods there are to reuse?**

This is the number one question in my mind with respect to OOP or any language that practices reuse. If I am a busy developer, how do I know what is available for me to reuse so that I do not reinvent what has already been done? Java comes with thousands of pages of documentation that covers thousands of different methods that can be reused. How do I know what there is to reuse?

As a subject in this book, we will be examining these built-in methods. My goals will not be to teach you all of them, but to show you how to access them. Once you understand how to access one method you can pretty much access any method.

Unfortunately the means of finding what is out there to reuse is pretty primitive. Some Integrated Development Environments sport object browsers and search capabilities to see what the libraries have to offer. The technology has a long way to go. At this point in time they are not that much better than a good index in a book. Of course if you are familiar with the terminology of the technology you are working with, a good index can take you a long way!

Java has a built-in system for documenting your code that I will be showing you.

2. **Can I rely on getting the same result if the method changes?**

You better be able to!

If you are writing code that others are going to reuse it is of vast importance that you maintain your code in a manner that any changes you make DO NOT have impact on the other developers' code!

This is not just a concept related to OOP. Imagine if you upgrade to Windows 2020 and none of the programs you run under Windows 2015 work. To say you would be "pissed off" is an understatement. It is your responsibility to ensure that the methods you write for others remain backwards compatible (at all costs and to the best of your ability). And if for whatever reason you cannot, you had better plan on some communication that allows your "customers" time to deal with your changes.

Ever wonder why each new release of Windows seems to double in size. It is because they are typically very good at practicing this principle.

3. **Should all code be written so that it can be reused?**

Of course not! Sometimes when studying the subject of OOP you get the idea that all code should be written to be reused. **This is an absolutely insane idea.** If for no other reason here is why. Assume I have a staff of 50 programmers. Given that each programmer not only needs to understand their own code and write for reuse, but they also need to be aware of what the other 49 programmers are doing and what they have to reuse. Let's get off the idealism for a moment and examine this. This would require a level of communication that is not only impractical to achieve, but would require so much of each developers time that any savings in the code they wrote for reuse would be lost in all of the time they were spending trying to figure out what the others were doing. Please, I love idealism, but it must be tempered with some sense of realism.

Then what should be reused? Common IO routines should be written with reuse in mind. How many times does one need to write a program that retrieves an order by invoice number? Other good candidates are general-purpose routines that upshift data, calculate dates or perform string manipulation. While there is no magic formula that can be applied with respect to this subject, code that is needed by more than one program should at least be given some degree of consideration for reuse. I would personally give greater consideration to reusing a routine that was going to be used in six programs more than one that was only needed in two. Knowing where you will need to use the same code is more of an art than a science, and past experience and prior development will generally give you a glimpse at the answer. Ideally it is better to be proactive than retrospective with regard to this subject. This is easier said than done.

4. **How does this idea affect how Development groups operate?**

Regardless of the programming paradigm, if one is going to employ a strategy that heavily includes reuse and wants to create code "object" or business services or whatever the heck you want call them, then some thought must be given to how you organize the development shop. Who is going to be focused on the reuse effort? Does your shop have well defined roles for Software Architects, Software Designers, Software Engineers, and for Programmers? Or, are these just titles that are thrown around like paper in a windstorm?

Generally the Senior Developers and Architects will be the ones that need to drive the analysis and efforts that result in reuse. One needs to have a very profound and in depth understanding of the project to be able to determine what can and cannot be reused. One also needs strong communication and leadership to document and get this information into the working hands of everyone else. Organizations that do not spend more time planning to code reuse than actually coding are usually doomed to fail. Personally, I would never let anyone lead a reuse project that does not have some prior experience, be it success or failure (assuming they learned valuable lessons from their failure).

5. **How do other developers know what there is to reuse?**

This is a really good question that you had better have a good answer for or no one will be reusing what has been built. Since it takes longer to write reusable code you will not have gained a whole lot except spending more time for little gain.

You may have noticed that I have been using the term reuse more than I have been using the term encapsulation. For me these terms are interchangeable in the context of this discussion. It is my contention that encapsulation is just a more narrowly defined implementation of reuse that falls under the reuse umbrella.

Inheritance

You can relax. I gave encapsulation more coverage than I plan on giving inheritance and polymorphism.

Inheritance is an idea that somewhat extends the idea of encapsulation by answering the following question. **What can I do if the code I am using does not fully meet my needs?** Inheritance allows the programmer to reuse and to extend or modify a piece of code that has already been written. I am not going to go into a lot of detail now on this subject but let me offer you this analogy. I might create a generic widget called a saw. All saws have sharp teeth. All saws cut wood. There are many kinds of saws out there (circular, jig, band, hand-crosscut, hand-rip, coping, table, etc…). The concept of Inheritance would work like this. I carefully analyze all of the attributes/properties (how many teeth per inch) and methods (cuts wood) that are common to all saws and create a generic saw object. Then I would inherit this object and tweak it further to satisfy the requirements of the particular saw I am dealing with. This in a nutshell is the concept of inheritance.

While inheritance is a very neat concept it is in reality one that requires even more attention than does encapsulation. At least with encapsulation you can call a method as a “black box” and expect a specific outcome. With inheritance you have to have a better understanding of all of the specific behaviors of the methods if you are to tweak them. It is this author’s opinion that inheritance requires a greater degree of documentation and education than does encapsulation. It most certainly creates a higher degree of code abstraction.

Polymorphism

This is probably the most abstract and ill explained idea of all the OOP concepts. Many years ago I interviewed a very experienced C++ programmer for a job and while he eloquently defined the previous two terms and provided practical examples of implementing each, he passed on defining this term.

While in a sense abstract, I think that this idea is made to be harder to understand than it really is. First let's start by dissecting the word. Poly means *more than one*; and morphic *pertains to form*. Now as the definition of morphic may still be nebulous for some, let's look to an example.

Consider a thermostat. A thermostat displays temperature and is used to regulate temperature. Imagine if you could buy a thermostat that could be used with any type of heating or cooling system. This would be polymorphic thermostat.

A polymorphic thermostat would have the innate ability to recognize what it is interfaced to! Polymorphism is a pretty hard thing to achieve in terms of applied engineering. A trip to your local hardware store will verify this. You can find thermostats that can handle electric and gas, but most thermostats are specific to the system they are going to be attached to.

In software development there are many times where polymorphism could provide an elegant solution to our problems. Consider something as simple as an address. The address is not such a simple data structure in software engineering. Does a U.S. address look like a Canadian or a European Address? Is the address for a person or a business?

Not so simple after all. This is a problem where a polymorphic solution would be quite elegant. Build an interface that can understand what type of address it is.

Now having offered the explanation and example, you will many times hear this term defined as ***"one interface, multiple implementations"***.

In terms of writing code it is my opinion that polymorphism is an advanced programming technique and not well suited for introductory material. As such you will not be getting any instructions or hands-on pertaining to what Java refers to as interfaces or abstract classes.

JAVA AND OOP HAS A LOT OF CLASS

If you have looked closely at the code you have been writing and the examples that I have been presenting you, you will have noticed that all of our code starts with this…

```
class <name of class>
{

}
```

…syntax. This is because in Java everything except the handful of variables I covered in the first lesson is a class.

What is a class?

A class is a blueprint used to create objects. Objects are created through instantiation of a class. Instantiation means create an instance of. A class is **<u>not</u>** the source code! The Java source is just the source code. It is not a class. It only becomes a class once we compile it.

What is an object?

An object is an instantiation of a class. In Java, one does not run a class. Remember a class is just a blueprint or template for an object. One runs and executes objects (or instances of a class).

Building our first class

Before jumping straight into writing the code for our first class, it would be a good idea to determine, at a high level, a set of objectives we need to meet in order to build a class.

- Define and create the data elements that the class will need
- Define the methods that the class will use

We can (and will) expand on this later. But for now these two bullets set enough direction for us to build our own class.

Let's build a class called phone. The phone class will have three data elements (area code, exchange, extension) and will have two methods. Display phone in U.S format and display phone in European format.

Source Phone.java

```
class Phone
{
      String area_cde  = " ";
      String exchange  = " ";
      String extension = " ";

      void phone_nbr_us() {
            String phn = "";
      phn = (area_cde + "-" + exchange + "-" + extension);
      System.out.println(phn);
      }

      void phone_nbr_europe() {
            String phn = "";
      phn = (area_cde + "." + exchange + "." + extension);
      System.out.println(phn);
      }

}
```

First I define a class named "Phone". I then create the three pieces of data that represent a phone number. While there is no requirement to initialize the data, I do anyway. I prefer to initialize all my data. Here I set each field to a single space.

Next I create the methods. The first method I name "phone_nbr_us". Since this method is not going to return anything I must make it type "void". In the method I create a string named "phn". I then assign phn the result of my three phone fields concatenated by the "-". Finally I print "phn".

My second method is identical to the first except my method name has changed and I am using the "." to concatenate the number. All said the code is pretty straightforward.

Next I need to compile the class.

Using our first class

The class I have created is not a standalone program like the ones we have created so far. We now need to create a Java program (class) or two that will use the “Phone” class.

Accessing the data in our class

Source Phn1.java

```
class Phn1
{
 public static void main(String args[])
 {
  Phone p = new Phone();
  p.area_cde = "422";
  p.exchange = "255";
  p.extension = "1903";
  System.out.println(p.area_cde);
  System.out.println(p.exchange);
  System.out.println(p.extension);
 }
}
```

Not a great deal of code here to review, but there are a few key nuggets of gold to cover. First let’s examine the line…

Instantiating a class

```
  Phone p = new Phone();
```

One the left side of the expression we are creating a handle (or alias) named “p” for the class “Phone” that we want to instantiate. On the right side of the expression we are instantiating the “Phone” class by using the key word **new**. If the phone class accepted parameters we would have to pass these parameters to the class through the parentheses “Phone()”. But our Phone class does not accept any parameters. However, we still require the parentheses. At this point, the general syntax for instantiating a class looks like.

Syntax

```
<class_name> <alias_name> = <class_name> (<optional_arguments>) ;
```

Note: This type of method is referred to as an “**Instance Method**”.

Referring to data in a class

Since we have instantiated, or made a copy of the phone class, we now have the ability to access both the data and methods that are exposed to us in the class. In the code below it is our intent to set the values for our phone number,

```
  p.area_cde = "422";
```

```
  p.exchange = "255";
  p.extension = "1903";
```

Notice that we refer to the data by using the alias (or handle) we made for the class.

Syntax
```
<alias_name>.<data_name_from_class>
```

```
  System.out.println(p.area_cde);
  System.out.println(p.exchange);
  System.out.println(p.extension);
```

Finally, we print the values we set. Now that you have seen how to access data members of a class, we can move on and examine how you access the methods of a class.

Source phn2.java

```
class Phn2
{
 public static void main(String args[])
 {
  Phone p = new Phone();
  p.area_cde = "422";
  p.exchange = "255";
  p.extension = "1903";

  p.phone_nbr_us();

 }
}
```

Above, see how we refer to a method in a class. The method prints…

422-255-1903

Syntax
```
<alias_name>.<method_from_class>(<optional_arguments>);
```

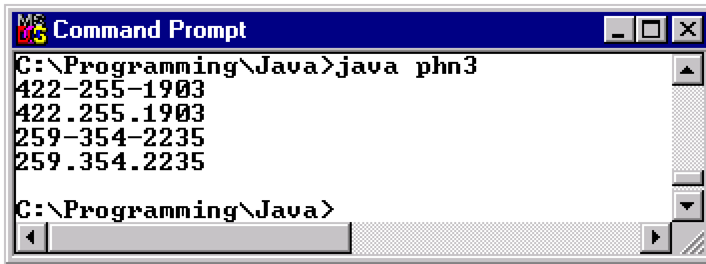
In the next example you see we can call both methods, and we can (since our class is still in scope, I will discuss scope a little later) change the phone number.

Source Phn3.java

```
class Phn3
{
 public static void main(String args[])
 {
  Phone p = new Phone();
  p.area_cde = "422";
  p.exchange = "255";
  p.extension = "1903";

  p.phone_nbr_us();
  p.phone_nbr_europe();
```

```
  p.area_cde = "259";
  p.exchange = "354";
  p.extension = "2235";

  p.phone_nbr_us();
  p.phone_nbr_europe();
 }
}
```

…yields…



So what if we want to create two phone numbers? In the last example we merely replaced the phone number. The answer is very easy. All we need to do in instantiate and use a second class.

Source Phn4.java

```
class Phn4
{
 public static void main(String args[])
 {
  Phone p = new Phone();
  Phone p2 = new Phone();

  p.area_cde = "422";
  p.exchange = "255";
  p.extension = "1903";

  p.phone_nbr_us();
  p.phone_nbr_europe();

  p2.area_cde = "259";
  p2.exchange = "354";
  p2.extension = "2235";

  p2.phone_nbr_us();
  p2.phone_nbr_europe();
 }
}
```

This is pretty slick if you think about it. We have created a “phone” class and have placed no restrictions as to how many phone numbers one can have in their program. We have also not only defined the schema (data), but we have also associated code with the data.

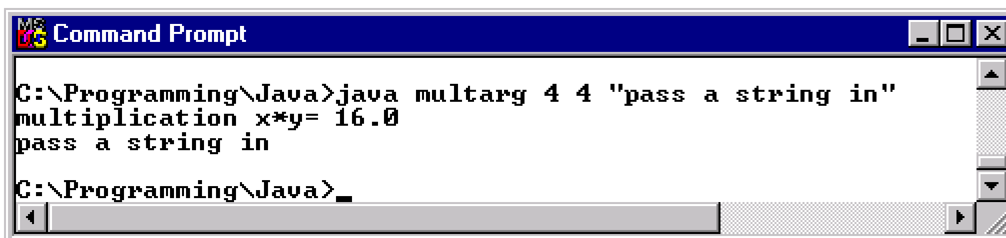
GETTING INPUT FROM THE TERMINAL WHEN YOU RUN YOUR PROGRAM

This may seem like an odd spot to cover this subject. The reason I have placed this topic here is that as I expand the discussion of classes I am going to increase the complexity of the examples and exercises such that we are going to want to test our classes without having to hardcode the values in our test program (the program/class that instantiates the class we create). Note, you saw this topic covered in the module "Writing Hello World Programs". Here we go again. This time I am showing you how to work with more than one argument.

Source Multarg.java

```
public class Multarg
{

      public static void main (String[] args){
            int x,y,z;
            x=0;
            y=0;
            z=0;
                String str = "";

            if (args.length == 3)
                {
                  x = Integer.parseInt(args[0]);
                  y = Integer.parseInt(args[1]);
                  z=x*y;
                        str = args[2];
            System.out.println("multiplication " + "x*y= " + z );
            System.out.println(str);
                }
                else
                {
                 System.out.println("example: java multarg 9 8 \"hello world\"");
                }
      }
}
```

To help explain the code, look at the following example.



Arguments can be passed in when running your program. These arguments automatically get assigned to an array named "args". The array, as you can see from the code, is zero based. You should have also noticed the method "Integer.parseInt()". This method allows us to convert the string we get from "args[ ]" and convert it to an int.

CONSTRUCTORS

A constructor is a special method that allows a class to be automatically initialized when it is instantiated. The constructor must have the same name as the class. Remember back to the first class we wrote. We had to set the values for each data item in our class by explicitly referencing it. We can also use the constructor to implicitly set the data values. Let's look at the class below.

Source Fund_stock.java

```
class Fund_stock
{
   double shares;
   double price_per_share;
   double commission;
   double total_amount;


// this is the constructor
   Fund_stock(double shrs, double pps, double comm)
   {
    shares = shrs;
    price_per_share = pps;
    commission = comm;
    total_amount = ((shrs * pps) - comm);
   }

}
```

This class accepts three values: number of shares, price_per_share, and commission. Given these three values, it calculates total amount. Now we will look at the code that calls this class.

Source Run_fund_stock.java

```
class Run_fund_stock
{

  public static void main(String args[])
  {
   Fund_stock fs = new Fund_stock(34, 24.75, 7.95);
   System.out.println("         Shares: " + fs.shares);
   System.out.println("Price Per Share: " + fs.price_per_share);
   System.out.println("   Total Amount: " + fs.total_amount);
   System.out.println("     Commission: " + fs.commission);
  }

}
```

As you can see, we create an instance of the class and we must also pass it the three values it requires. While you do not have to explicitly reference the data members, you still need to know about them and what they represent.

CONSTRUCTOR OVERLOADING

Overloading is a totally novel concept related to OOP. In my assessment it is one of the cornerstone concepts that distinguish OOP from non-OOP languages. Overloading a constructor allows you to have more than one constructor for a class. Why would you need more than one constructor for a class?

Consider an account number. An account number can be defined as a number (for example 003425) or it can also be defined as a number with alphanumerics (for example AR34820). As interfacing between disparate systems is the holy grail to achieve in any age, wouldn't it be nice if there was an elegant solution that knew what data type was what and acted upon each accordingly?

Source Account.java

```
class Account
{

	int Iaccount;
	String Saccount;

	Account(int acct) {
	Iaccount = acct;
	System.out.println("Account is a number");
	}

	Account(String acct) {
	Saccount = acct;
	System.out.println("Account is a string");
	}

}
```

Source Run_account.java

```
class Run_account
{

    public static void main (String[] args)
    {
     Account a1 = new Account("EB346");
     Account a2 = new Account(24533);
    }
}
```

You really need to spend some time thinking about this example. As simple as it is, it demonstrates a totally cool, mind blowing concept that should get you jazzed! The concept of overloading is totally cool. Here is another more generic example for you to examine.

Source Cntrct.java

```
class Cntrct
{
```

```
/*
Demonstrates constructor overloading. cntrct_run is uses this class. Notice
we have 3 data types represented for the same class. This is a very simple
example of polymorphism, a very powerful technique in object oriented
programming.
*/

      int x;
      double y;
      String z;

      Cntrct(int a) {
      x = a;
      System.out.println("int: " + a);
      }

      Cntrct(double b) {
      y = b;
      System.out.println("double: " + b);
      }

      Cntrct(String c) {
      z = c;
      System.out.println("String: " + z);
      }

}
```

Source Run_cntrct.java

```
public class Run_cntrct
{
/*
This is an example of calling on the same object using different
data type arguments. While this is a pretty trivial example, it
is pretty cosmic if you think about it. This example demonstrates
the polymorphic aspect of object oriented programming. Imagine a
more practical example, like passing a system a medical record number
and not having to be concerned with data type issues.
*/

      public static void main (String[] args){

// the class Cntrct will accept String, int, or double data types
      Cntrct q = new Cntrct("goodbye");
      Cntrct r = new Cntrct(15);
      Cntrct s = new Cntrct(34.76);
      }
}
```

Overloading is one of the coding techniques you use to achieve polymorphism (one interface, multiple implementations).

THIS KEYWORD

The “this” keyword is used mostly (though not totally) as a stylistic convention. This refers to the object that is currently being executed. The following example works the same as the one offered above. The only code added is the key word reference. If you want to see this ‘in-action’, run “run_cntrct1.java”.

Source Cntrct1.java

```
class Cntrct1
{

// Demonstrates use of this keyword

      int x;
      double y;
      String z;

      Cntrct1(int a) {
      int x = 0;
      this.x = a;
      System.out.println("int: " + a);
      }

      Cntrct1(double b) {
      this.y = b;
      System.out.println("double: " + b);
      }

      Cntrct1(String c) {
      this.z = c;
      System.out.println("String: " + z);
      }

}
```

METHOD OVERLOADING

An overloaded method works the same as an overloaded constructor. An overloaded method is achieved when two or more methods share the same name. Overloaded methods are most useful as they help you to code variations of a similar concept using the same naming convention. This makes documentation and reuse easier and can be used to abstract the interface between consumers of your code.

The “println()” method we have used in every program is an example of an overloaded method.

Source Method_called.java

```
class Method_called
{

  static void Method_called (String str)
  {
   System.out.println("method called is... " + str);
```

```
  }

  void datatype(char c)
  {
   Method_called("char");
  }

  void datatype(int b)
  {
   Method_called("int");
  }

  void datatype(String s)
  {
   Method_called("String");
  }

  void datatype(double d)
  {
   Method_called("double");
  }

  public static void main(String[] args)
  {
   Method_called p1 = new Method_called();
   p1.datatype('2');
   p1.datatype(2);
   p1.datatype("2");
   p1.datatype(2.01);
  }

}
```

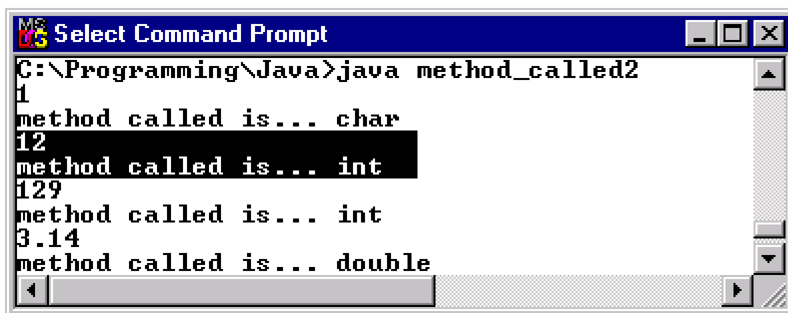
You can see from the program above that the method called is determinant on the value that is passed in the argument of the method call. Interesting enough, sometimes a data type gets “promoted to another data type”. Consider the code below (in particular the parts that are bolded):

Source Method_called2.java

```
class Method_called2
{

  static void Method_called (String str)
  {
   System.out.println("method called is... " + str);
  }

  void datatype(char c)
  {
   char v='0';
   v=c;
   System.out.println(v);
   Method_called("char");
  }

  void datatype(float f)
```

```
  {
   float v=0;
   v=f;
   System.out.println(f);
   Method_called("float");
  }

  void datatype(int i)
  {
   int v=0;
   v=i;
   System.out.println(v);
   Method_called("int");
  }

  void datatype(double d)
  {
   double v=0;
   v=d;
   System.out.println(v);
   Method_called("double");
  }

  void datatype(byte b)
  {
   byte v=0;
   v=b;
   System.out.println(v);
   Method_called("byte");
  }



  public static void main(String[] args)
  {
   Method_called p1 = new Method_called();
   double x = 12.243;
   p1.datatype('1');
   p1.datatype(12);
   p1.datatype(x);
  }
}
```

We might expect that when we pass the value 12 to the method we would get the byte method as this number best fits this data type. However, the value is promoted to type “int”.



VARIABLES AND VARIABLE SCOPE

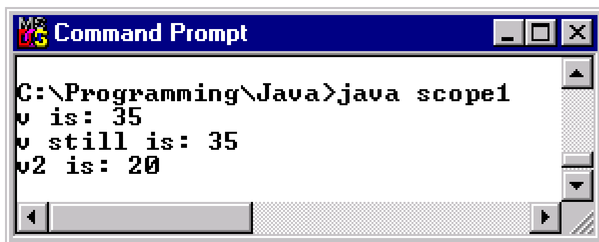
We actually have covered this subject, though we have done so indirectly. I did not want to get into a specific discussion on this while we were elbow deep into classes for the first time. We will quickly review what you have already seen and coded yourself.

In Java there are three classifications of variables. They are…

- Instance variables
- Static variables
- Local variables

You have seen and used two types thus far (Instance and Local). First and foremost, both static and instance variables are global in their scope. The difference between a static variable and an instance variable is best explained with some toy code - as some toycode is better than a thousand words.

Source Scope1.java



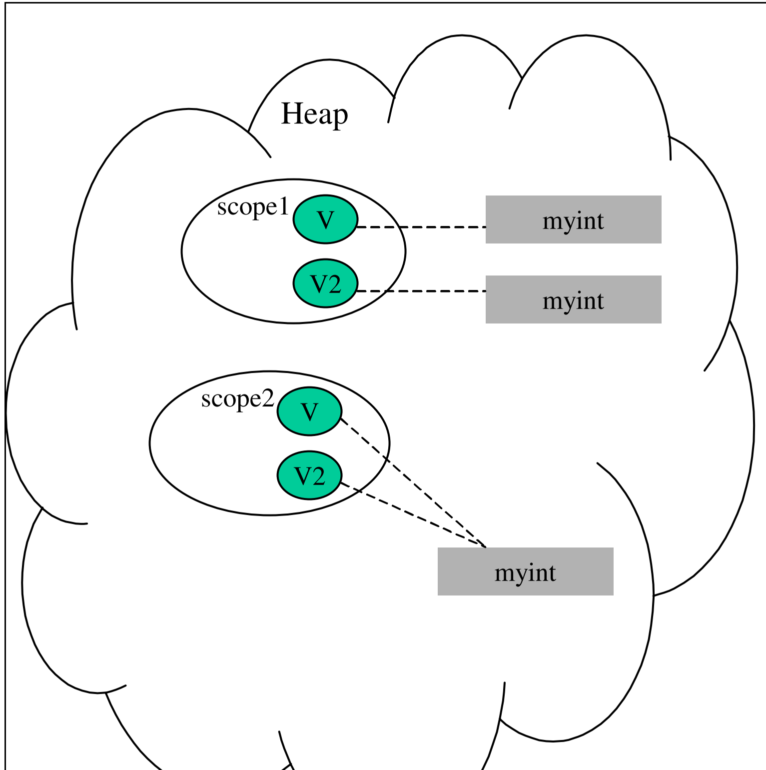

```
class Scope1
{
 public static void main(String args[])
 {
  Var v = new Var();
  v.myint = 35;
  System.out.println("v is: " + v.myint);

  Var v2 = new Var();
  v2.myint = 20;
  System.out.println("v still is: " + v.myint);
  System.out.println("v2 is: " + v2.myint);
 }
}

class Var
{
  int myint;
}
```

In this example each time we instantiate the class, a copy of the variable “v” is created on the heap. These types of variables are referred to as **instance** variables.

Source Scope2.java

```
class Scope2
{
 public static void main(String args[])
 {
  Var v = new Var();
  v.myint = 35;
  System.out.println("v is: " + v.myint);

  Var v2 = new Var();
  v2.myint = 20;
  System.out.println("v is now: " + v.myint + "  hmmmmm...");
  System.out.println("v2 is: " + v2.myint);
 }
}

class Var
{
  static int myint;
}
```

In this example, because “v” was declared as static, it is only defined once regardless of how many times its class is instantiated. This is obvious by examining the code and the results of running the program. These types of variables are referred to as **static** variables. Following is a picture that graphically depicts the concept I just showed you in code. By the way, the heap is an addressable portion of memory. The heap is where Java objects live. This should be enough detail on this subject.



Finally, there are local variables. These are variables that are declared within a method. The scope of these variables is local to the method. Actually scope of a variable is even more granular than this (in or outside of a method). Scope is defined within a block. Consider the code below:

Source Scope3.java

```
class Scope3
{
 public static void main(String args[])
 {
  Var v = new Var();
  v.mymethd();
 }
}

class Var
{
  static int myint=0;     //myint in scope

  void mymethd()
  {
     int foo=30;           //foo in scope

    {
     int a=5;              //a in scope
       {
        int b=3;           //b in scope
            {
             int c=10;     //c in scope
```

```
            myint += c;
            System.out.println("Sum: " + myint);
           }                //c no longer in scope
            myint += b;
            System.out.println("Sum: " + myint);
       }                    //b no longer in scope
            myint += a;
            System.out.println("Sum: " + myint);
    }                       //a no longer in scope
            myint += foo;
            System.out.println("Sum: " + myint);
            myint += myint;
            System.out.println("Sum: " + myint);
  }                         //foo no longer in scope
          //note: we cannot write code outside of a method so
          //      the last 2 lines of code are inside the method
}                           //myint no longer in scope
```

The comment codes in the program explain what is happening so no further narrative is offered. You should be able to grasp scope of a variable at the class, method and block level.

QUICK INVENTORY

We have covered a great deal of ground. I have broken the subject of OOP into multiple lessons so you have time to digest and absorb each item covered.

- Thus far you need to have a conceptual understanding of the three main principles of OOP. Since OOP is about reuse you should be getting onto the reuse bandwagon. You should have an idea of the benefits, and just as important, the drawback's of reuse as it is applied to OOP.
- Understand the relationship between a class and a method.
- Begin understanding the use of variables within a class and within methods.
- Understand and use constructors and overloaded constructors in classes.
- Understand and use overloaded methods

If you can check off this inventory in its entirety, you are ready for the next lesson.