



## Java Foundations

# Misc Data/Object Issues

## Copyright

Introduction to Java Programming by Eric Matthews is licensed under a Creative Commons Attribution 3.0 Unported 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](http://www.DeveloperGeekResources.com)

[www.EducationAnytime.com](http://www.EducationAnytime.com)

---

## Table of Contents

<i>About this document</i> .....	4
<i>Java Access specifiers</i> .....	5
friendly .....	5
public .....	5
private.....	5
<i>Understanding Storage of Objects and Primitives</i> .....	7
The Stack and the Heap.....	7
<b>Stack</b> .....	7
<b>Heap</b> .....	7
<b>Application of these concepts</b> .....	7
<i>Initialization Issues</i> .....	16
How are variables initialized? .....	16
When does a constructor get called? .....	16
<i>Passing data in and out of Methods</i> .....	18
Using return.....	18
Passing arrays as an argument.....	20
Passing Objects as an argument .....	25

## ABOUT THIS DOCUMENT

This is a collection of various topics that do not fit naturally into the others. You know that you really should cover a topic at a certain place, but you also know that it could lead you down a long and winding trail, or serve to confuse and bewilder your audience.

I have taught structured programming and object oriented programming classes on and off for twenty years and have found this to be the case.

I mention these things in case you try to use these documents outside of my class. I apologize that the materials are not necessarily in a format that is all that linear. I am not sure that I will ever be able to get them into a linear format, as OOP does not exactly progress along a linear path.

As a final thought that might help you to use this document, I found that these materials are best covered after the module, "Second Steps in Object Oriented Programming".

## JAVA ACCESS SPECIFIERS

First of all what is a Java access specifier? An access specifier determines what the consumer of your class can and cannot access. Sometimes you will here this access referred to as “member access”. Make note that a member refers to a class, a method, or a variable

### FRIENDLY

This is the default access. The default access has no keyword. This type of access has package scope. This means that all classes that are part of the package have the ability to access the members.

### PUBLIC

Public means that anyone and their brother can access the member

### PRIVATE

Private means that the member is only available within the scope of that member. For instance consider the following program.

**Source** Access1.java

```
class Access1
{
    // constructor
    void access1()
    {
        Hello();
        Goodbye();
    }

    public void My_interface()
    {
        Hello();
        Goodbye();
    }

    private void Hello()
    {
        System.out.println("hello");
    }

    private void Goodbye()
    {
        System.out.println("goodbye");
    }
}
```

```
public static void main(String args[])
{
    Access1 myaccess = new Access1();
    myaccess.Hello();
    myaccess.Goodbye();
}
}
```

In this program we have two methods that are both declared as private. We are still able to access these methods from within the class. However, no one outside of the class has access to these. So how would others be able to access these methods?

Take a look at the following program.

**Source** Useaccess.java

```
public class Useaccess
{

    public static void main(String args[])
    {
        Access1 myaccess = new Access1();
        myaccess.My_interface();
        //myaccess.Hello();
    }
}
```

We are able to access the methods by creating an instance of access1. If you refer back to the access1 code you will see that we have a constructor that is public that calls these two private methods. Now if you comment out the line “//myaccess.Hello();” in Useaccess and recompile this you will get the following compiler error.

```
Useaccess.java:7: Hello() has private access in Access1
    myaccess.Hello();
                ^
1 error
```

It can't find the method Hello() because it is private. Private gives you the ability to hide code from other consumers. This is what encapsulation is all about. These specifiers allow you the ability to hide code from the consumer, but to create interfaces that allow the consumer to use your code. Since Java is all about objects, the consumer is using your classes (which get compiled into bytecode). Since our reuse effort is not with respect to the source code we really have insulated our code from others (well of course there are always disassemblers if someone is really intent on hacking your stuff. It is important to reiterate that reuse efforts take a great deal of thought and planning. Further, you need to be able to guarantee your methods when you encapsulate your code. Meaning... What is a programmer to do if they are using your stuff and suddenly it breaks their stuff because you have made changes?

## UNDERSTANDING STORAGE OF OBJECTS AND PRIMITIVES

### THE STACK AND THE HEAP

I find that it is very important to understand how storage and instantiation occurs in Java. Without this understanding you will end up either very confused by certain behaviors that result in the code you write. You will ultimately end up living in the realm of cause and effect and relishing some of what Java does to magic. This would be unnecessary and unfortunate. I promise to limit my discussion of the heap and stack in ways that can be directly applied to the code we write.

#### Stack

The stack lives in RAM (random access memory). The stack provides direct support for the processor via what is referred to as a stack pointer. The stack pointer is used to allocate (push) and deallocate (pop) memory. Stored in this memory are static data types, handles (references) to objects, and other assembly level instructions. The stack is incredibly fast and efficient. However, the size, scope and life of data and instructions on the stack must be known by the compiler at compilation in order to be able to generate the code needed to move the stack pointer. Such a requirement serves to limit ones flexibility in programming. Of course where sheer brute force and speed are required this is where languages like Assembler, C, TAL, and C++ come in real handy.

**In Java references to objects (also called handles) are stored on the stack.**

#### Heap

The heap is a pool of RAM where all Java objects live. The beauty of the heap is that the compiler does not need to know anything about storage requirements (persistence, size, etc...) at compile time. This gives one a great deal of flexibility in coding. This flexibility comes at the cost of performance when compared to stack storage.

#### Application of these concepts

You now have enough theory (hey, less than a page!) that we can look at a few toycode examples and apply all of this.

Let's first examine the code from the program below, and walk-through what is happening.

**source** Assign.java

```
public class Assign
{
    static int a=0;

    public static void main(String[] args)
    {
```

```
System.out.println("a= " + a);
a=8;
System.out.println("a= " + a);

Assign asgn1 = new Assign();
Assign asgn2 = new Assign();

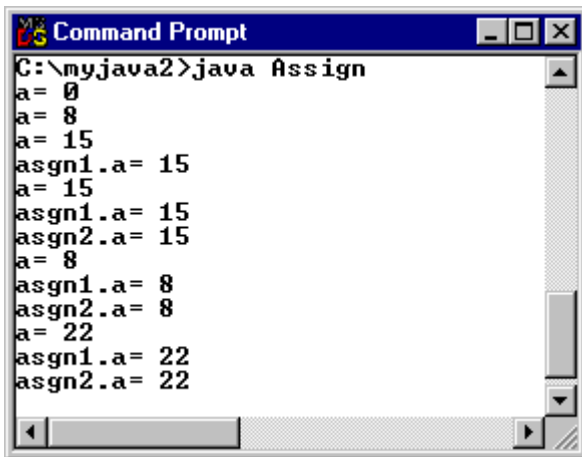
asgn1.a = 15;
System.out.println("a= " + a);
System.out.println("asgn1.a= " + asgn1.a);

asgn1 = asgn2;
System.out.println("a= " + a);
System.out.println("asgn1.a= " + asgn1.a);
System.out.println("asgn2.a= " + asgn2.a);

asgn2.a =8;
System.out.println("a= " + a);
System.out.println("asgn1.a= " + asgn1.a);
System.out.println("asgn2.a= " + asgn2.a);

asgn1.a =22;
System.out.println("a= " + a);
System.out.println("asgn1.a= " + asgn1.a);
System.out.println("asgn2.a= " + asgn2.a);
}
}
```

Please take a moment to look at the output of this program.



In this program we have created a class that only consists of a single variable. I want you to also note that this variable has been declared as “static”. I am going to define static for you as a “single instance”. You will see shortly why this is a pretty decent description. “static” is not a constant.

**Note:** *I would strongly urge you to slowly and methodically walk-through this discussion as it will unlock some of the subtleties of Java objects and primitives and give you a good solid foundation.*

The statement ...

```
static int a=0;
```

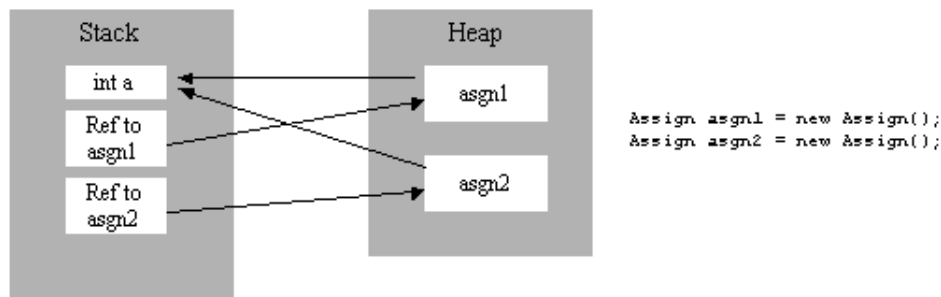
... will cause the compiler to allocate storage for an int on the stack.



Next, the statements ...

```
Assign asgn1 = new Assign();
Assign asgn2 = new Assign();
```

... will cause the compiler to create two instances of the object `Assign()` onto the heap.



It is important to note that since our class "Assign()" only contains a single static variable that the object created on the heap is similar to a pointer (reference, or handle) to the actual integer on the stack.

**Authors Note:** *Without arguing the semantics of the terms reference, pointer, or handle...albeit to say that these terms mean essentially mean the same thing. Essentially is a key word here. The truth be told a handle/reference in Java is really not the equivalent of a pointer in C or C++. Java does not give the programmer real access to pointers (one of its claims to fame). I will be using the terms handle and reference interchangeably. I will not be using the term pointer again.*

In the next statement we assign `asgn1` to `asgn2`. This changes the reference of that object from pointing to the stack variable to the object `asgn2`.

```
asgn1 = asgn2;
```

This of course is a moot point, as our reference is still to the static variable on the stack. This is evidenced by the line of code ...

```
asgn1.a =22;
```

We have assigned the entire object `asgn1` to `asgn2`. But when we change the reference to the variable within `asgn1` we see that the value referred to in `asgn2` also changes. Again, this is because these objects only have a reference to the variable on the stack, because the variable was declared as static!

**"single instance"**

You can also view a static variable as a global variable whose scope is to the class.

What would happen if we tried to break out the program into two separate programs.

**source** Assign1.java

```
public class Assign1
{
    static int a=0;
}
```

**source** UseAssign1.java

```
public class UseAssign1
{
    public static void main(String[] args)
    {

        System.out.println("a= " + a);
        a=8;
        System.out.println("a= " + a);

        Assign asgn1 = new Assign();
        Assign asgn2 = new Assign();

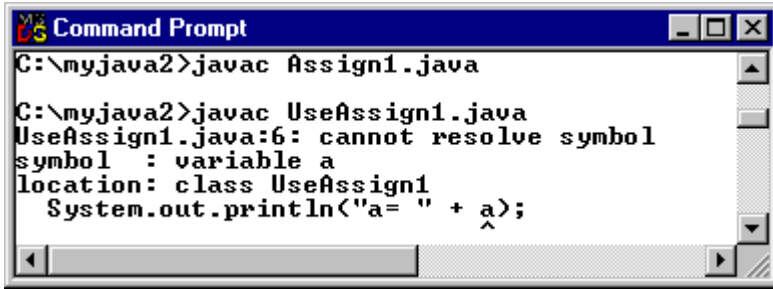
        asgn1.a = 15;
        System.out.println("a= " + a);
        System.out.println("asgn1.a= " + asgn1.a);

        asgn1 = asgn2;
        System.out.println("a= " + a);
        System.out.println("asgn1.a= " + asgn1.a);
        System.out.println("asgn2.a= " + asgn2.a);

        asgn2.a =8;
        System.out.println("a= " + a);
        System.out.println("asgn1.a= " + asgn1.a);
        System.out.println("asgn2.a= " + asgn2.a);

        asgn1.a =22;
        System.out.println("a= " + a);
        System.out.println("asgn1.a= " + asgn1.a);
        System.out.println("asgn2.a= " + asgn2.a);
    }
}
```

Assign1.java compiles fine. But when we try to compile UseAssign1.java we receive a slew of errors.



```
Command Prompt
C:\myjava2>javac Assign1.java
C:\myjava2>javac UseAssign1.java
UseAssign1.java:6: cannot resolve symbol
symbol : variable a
location: class UseAssign1
    System.out.println("a= " + a);
                        ^
```

The errors we receive tell us that we do not have access to variable “a” as it is out of our scope since our main method is no longer a part of the class Assign1.

I want to make another point here related to Assign.java. Consider if we rewrite the following code from

```
asgn1 = asgn2;
```

To

```
asgn1.a = asgn2.a;
```

Our program would still run and produce the same results but let’s think about what the above statement means. This statement is telling the compiler to assign the value of variable a in object asgn2 to the value of variable a in object asgn1. This statement is a meaningless one as both objects, when instantiated, did not create a copy of the int, rather a reference to the int on the stack as the int was declared as static.

What we have just covered does not represent a programming technique you would ever want to practically employ. My purpose was to demonstrate some of the subtleties of objects and variables. There are those in the OOP ranks that consider using static to be a violation of object oriented programming. I have just presented you an extreme example that certainly demonstrates this point. While I agree with this premise I do believe that there are times when the use of static variables or methods is warranted.

Another reason I have covered this to such a degree is that you may be coming from a language where the use of global variables is commonplace. For example, in a Cobol program all variables are global and their scope is for the life of the program. Coming into Java you will need to rethink how you deal with issues of storage as well as start thinking of code in terms of objects. This is easier said than done.

Now, consider the next program and the results produced.

**source** UseAssign2.java

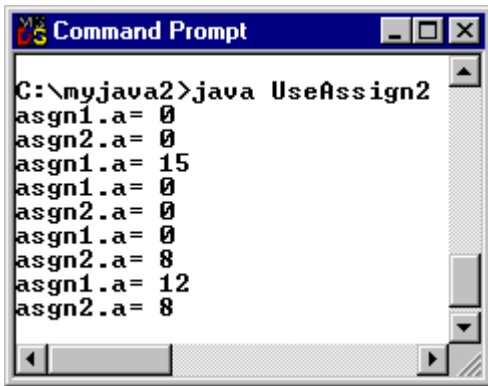
```
class UseAssign2
{
    static void main(String[] args)
    {
        Assign2 asgn1 = new Assign2();
        System.out.println("asgn1.a= " + asgn1.a);
        Assign2 asgn2 = new Assign2();
        System.out.println("asgn2.a= " + asgn2.a);
    }
}
```

```
    asgn1.a = 15;
    System.out.println("asgn1.a= " + asgn1.a);

    asgn1.a = asgn2.a;
    System.out.println("asgn1.a= " + asgn1.a);
    System.out.println("asgn2.a= " + asgn2.a);

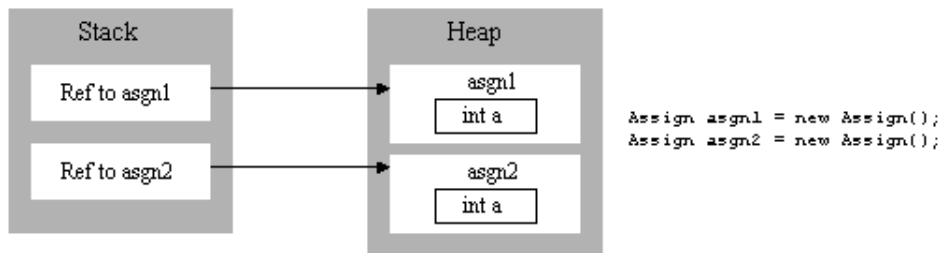
    asgn2.a =8;
    System.out.println("asgn1.a= " + asgn1.a);
    System.out.println("asgn2.a= " + asgn2.a);

    asgn1.a =12;
    System.out.println("asgn1.a= " + asgn1.a);
    System.out.println("asgn2.a= " + asgn2.a);
}
}
```



In this program we still have a class that only contains a single variable. This time our variable is not static. If you notice, we do not at any time make a direct reference to “int a”. If we did, we would receive a compiler error when we tried to compile UseAssign2.java.

For each instantiation of the Assign2.class we get a heap object. This is nothing cosmic here. In fact it is just good old-fashioned object code reuse. In this case each instance of Assign2.class will create a



You can see from the results that these variables are different and the only time they contain the same value is when we write an assignment statement.

```
asgn1.a = asgn2.a;
```

Notice that the assignment statement is granular to the variables for each object instance. Would we get the same results if we made the assignment statement at the object level? After all, the class we are creating an object from only contains one variable. The results may surprise you.

**source** UseAssign3.java

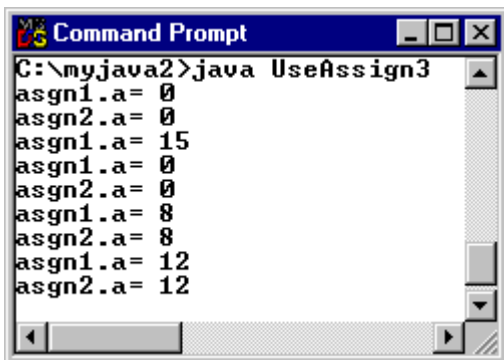
```
class UseAssign3
{
    static void main(String[] args)
    {
        Assign3 asgn1 = new Assign3();
        System.out.println("asgn1.a= " + asgn1.a);
        Assign3 asgn2 = new Assign3();
        System.out.println("asgn2.a= " + asgn2.a);

        asgn1.a = 15;
        System.out.println("asgn1.a= " + asgn1.a);

        asgn1 = asgn2;
        System.out.println("asgn1.a= " + asgn1.a);
        System.out.println("asgn2.a= " + asgn2.a);

        asgn2.a =8;
        System.out.println("asgn1.a= " + asgn1.a);
        System.out.println("asgn2.a= " + asgn2.a);

        asgn1.a =12;
        System.out.println("asgn1.a= " + asgn1.a);
        System.out.println("asgn2.a= " + asgn2.a);
    }
}
```



```
Command Prompt
C:\myjava2>java UseAssign3
asgn1.a= 0
asgn2.a= 0
asgn1.a= 15
asgn1.a= 0
asgn2.a= 0
asgn1.a= 8
asgn2.a= 8
asgn1.a= 12
asgn2.a= 12
```

```
asgn1 = asgn2;
System.out.println("asgn1.a= " + asgn1.a);
System.out.println("asgn2.a= " + asgn2.a);

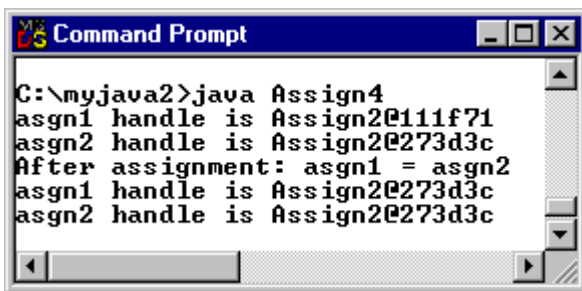
asgn2.a =8;
System.out.println("asgn1.a= " + asgn1.a);
System.out.println("asgn2.a= " + asgn2.a);
```

```
asgn1.a =12;  
System.out.println("asgn1.a= " + asgn1.a);  
System.out.println("asgn2.a= " + asgn2.a);
```

Please take notice that the behavior is markedly different between this program and the previous one. In the previous program the assignment statement just changed the value that we were storing in asgn1.a to the value that is stored in asgn2.a. However, when we assign the object asgn1 to asgn2 we are actually changing the handle (reference) of asgn1 to point to asgn2. Review the following program that is a rewrite to demonstrate this to you.

**source** UseAssign4.java

```
class UseAssign4  
{  
  
    public static void main(String[] args)  
    {  
  
        Assign3 asgn1 = new Assign3();  
        System.out.println("asgn1 handle is " + asgn1);  
        Assign3 asgn2 = new Assign3();  
        System.out.println("asgn2 handle is " + asgn2);  
  
        asgn1 = asgn2;  
        System.out.println("After assignment: asgn1 = asgn2");  
        System.out.println("asgn1 handle is " + asgn1);  
        System.out.println("asgn2 handle is " + asgn2);  
  
    }  
}
```



Let's look back to UseAssign3 for a moment.

```
asgn2.a =8;  
System.out.println("asgn1.a= " + asgn1.a);  
System.out.println("asgn2.a= " + asgn2.a);  
  
asgn1.a =12;  
System.out.println("asgn1.a= " + asgn1.a);  
System.out.println("asgn2.a= " + asgn2.a);
```

It might appear after the assignment of `asgn1` to `asgn2` that these two objects were somehow now synchronized with one another. After all it does not matter which variable we change, they both change. But, as you can see from `UseAssign4` that once we assign one object to the other what we really end up with is one object. Our original object `asgn1` has been marked for garbage collection. We still seem to be able to refer to `asgn1` though. How peculiar. Actually what has happened is by doing the assignment we now have an alias to `asgn2` named `asgn1`.

If you are coming from the world of structured programming you probably find much of this a bit peculiar. I think it is important to understand the concepts that I have just presented through actual code. This was not presented as a suggestion on how you should program in Java. But, in order to show these quirky things one must write some funky code. I think there are many of us that have come into Java from the world of structured programming and made similar mistakes and assumptions. I suppose we could re-title this, "Before you put your hand on the hot stove, look at this".

## INITIALIZATION ISSUES

### HOW ARE VARIABLES INITIALIZED?

The answer to this question depends on where they are declared. Consider the following program.

**source** Vinit1.java

```
class Vinit1
{
    int a;

    void meth1()
    {
        System.out.println(a);
        a=5;
        System.out.println(a);
        int b;
        System.out.println(b);
    }
}
```

This program will not compile as it is written. We receive a compiler error.

```
Vinit.java:11: variable b might not have been initialized
    System.out.println(b);
                    ^
```

The real question is not why did we receive a compiler error for not initializing “b”, rather why did we not receive an error for not initializing “a” as well? “a” is a primitive data type and a direct part of the class itself. “a” gets initialized automatically by the default constructor. The same holds true for any handles (as you see later on). What this means is that if you say “a=0;” that “a” gets initialized to zero twice; once by your statement, and once by the default constructor.

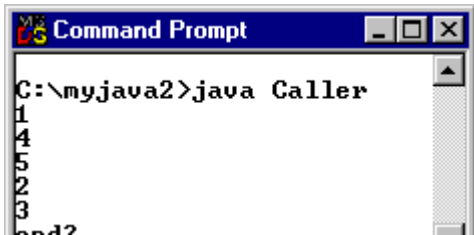
### WHEN DOES A CONSTRUCTOR GET CALLED?

A constructor is called only after all the variables exposed to the class are initialized. By variables I am referring to both primitives and handles. The next example should clarify what I just said.

**source** Caller.java

```
class Init1
{
    Init1(int i)
    {
        System.out.println(i);
    }
}
```

```
}  
  
class CallInit1  
{  
    Init1 I1 = new Init1(1);  
  
    CallInit1() // constructor  
    {  
        Init1 I2 = new Init1(2);  
        Init1 I3 = new Init1(3);  
    }  
  
    Init1 I4 = new Init1(4);  
    Init1 I5 = new Init1(5);  
}  
  
public class Caller  
{  
    public static void main(String[] args)  
    {  
        CallInit1 I = new CallInit1();  
    }  
}
```



This makes good sense. We do not want to initialize our class until we have created all the objects and storage that it requires.

## PASSING DATA IN AND OUT OF METHODS

The following program will give a good perspective on passing values into a method, and returning a result. Review the code then read the walkthrough that follows.

Additionally, be aware that arguments are passed to methods by value. This means that a copy is made for the argument we are passing. And while I am thinking about it I realize that we always use the term passing to describe this operation and that this is really a poor choice of words. If I pass you a football, this means that you now have the football, and I do not. In computer science when I pass a parameter by value, I am essentially making a clone of the football and sending you the clone. If I pass you the parameter by reference I am not really giving you the football, I am sending you a pointer to where I have the football.

Anyhow, it is important that you understand that we pass by value. The ensuing code and discussion will demonstrate passing various primitives and objects into and out of methods.

### USING RETURN

**Source** Return1.java

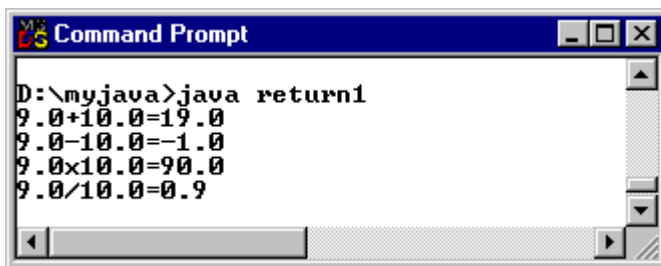
```
class Return1
{
    public float mult(float i,float y)
    {
        float x=0;
        x=(i*y);
        return x;
    }

    public float div(float i,float y)
    {
        float x=0;
        if (y != 0)
        {
            x=(i/y);
        } else
        {
            System.out.println("Divide by 0");
        }
        return x;
    }

    public float add(float i,float y)
    {
        float x=0;
        x=(i+y);
        return x;
    }
}
```

```
public float subtr(float i,float y)
{
    float x=0;
    x=(i-y);
    return x;
}

public static void main(String args[])
{
    float r=0;
    float g=9;
    float h=10;
    Return1 c = new Return1();
    r=c.add(g,h);
    System.out.println(g + "+" + h + "=" + r);
    r=c.subtr(g,h);
    System.out.println(g + "-" + h + "=" + r);
    r=c.mult(g,h);
    System.out.println(g + "x" + h + "=" + r);
    r=c.div(g,h);
    System.out.println(g + "/" + h + "=" + r);
}
}
```



```
Command Prompt
D:\myjava>java return1
9.0+10.0=19.0
9.0-10.0=-1.0
9.0x10.0=90.0
9.0/10.0=0.9
```

Lets first look at the methods. Since they are all pretty much the same we will examine the code for div().

```
public float div(float i,float y)
{
    float x=0;
    if (y != 0)
    {
        x=(i/y);
    } else
    {
        System.out.println("Divide by 0");
    }
    return x;
}
```

The div() method accepts two arguments. Both are primitive data of type float. It is important to remember that we are actually passing by value, not by reference. Variables "i" and "y" are local to the

method. While changing the values of either of these will affect the outcome of what “x” returns they have not impact whatsoever on the variables that are used by the caller. Our method div() will return float “x” to the caller.

Back to our main program...

```
float r=0;
float g=9;
float h=10;
Return1 c = new Return1();
```

... we declare and set the variables we need in order to do the method call. Then, we create a Return1 object. Once we have created this object we have access to the methods (it allows us exposure to, which in this case is all of them).

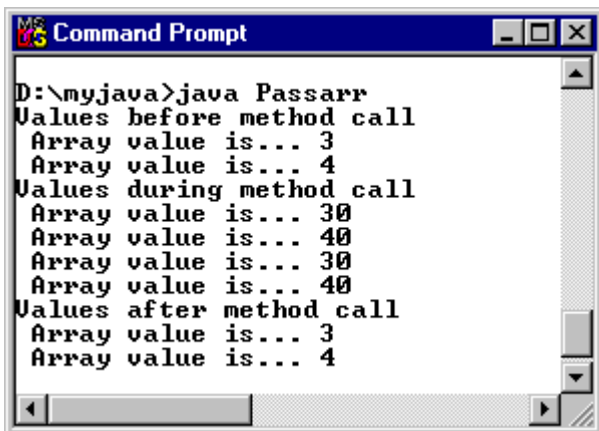
```
r=c.div(g,h);
System.out.println(g + "/" + h + "=" + r);
```

We call the div() method as part of an expression. We are looking through the assignment operator for the method to return a value to our variable “r”. We call the method on the other side of the assignment operator. Notice we separate what we are passing with the call using the comma (,) operator. Position is important. The method will return “x” to “r”. I purposely used different variable names so that you could see that these variables too are indeed different. Because of how namespaces are used in Java we could have used the same names for all of our variables.

It is also possible to pass arrays and for that matter, any objects of any type into a method. It is also possible to return such items as part of a method call.

## PASSING ARRAYS AS AN ARGUMENT

When you pass an array by argument the array reference is passed by value. This is going to take some time to explain so let’s first look at the program from a data perspective.



**Source** Passarr.java

```

class Passarr
{
    public static void main(String args[])
    {
        int arr[] = {3,4};
        System.out.println("Values before method call");
        disp(arr);
        acceptarr(arr);
        System.out.println("Values after method call");
        disp(arr);
    }

    static void acceptarr(int p_arr[])
    {
        int d[] = {30,40};
        p_arr=d;
        System.out.println("Values during method call");
        disp(p_arr);
        disp(d);
    }

    static void disp(int a[])
    {
        System.out.println(" Array value is... " + a[0]);
        System.out.println(" Array value is... " + a[1]);
    }
}

```

Here is what is happening. First we create an array named “arr” and place two values in it (3,4). Next we pass this array into a method named “acceptarr”. We assign the array we passed to another array named “d” in our method. The values of this array are (30,40). After the method call we display the contents of the array we passed and see that the values are still (3,4). At this point one might conclude that p\_arr and arr are two separate arrays. This however is not the case.

Let’s look at a slightly modified version of this program where we can see the handles (references) of the arrays in our program.

**Source** Passarr\_a.java

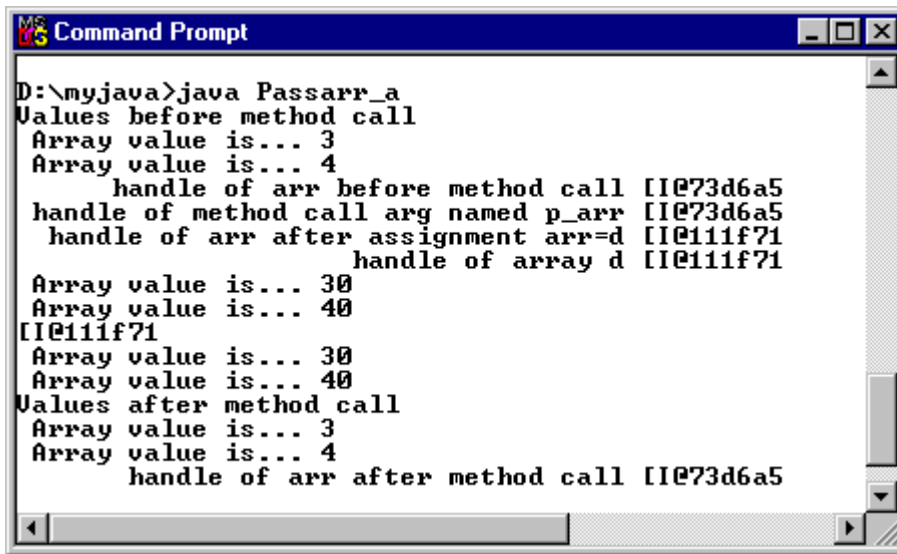
```

class Passarr_a
{
    public static void main(String args[])
    {
        int arr[] = {3,4};
        System.out.println("Values before method call");
        disp(arr);
        System.out.println("        handle of arr before method call " + arr);
        acceptarr(arr);
        System.out.println("Values after method call");
        disp(arr);
        System.out.println("        handle of arr after method call " + arr);
    }
}

```

```
static void acceptarr(int p_arr[])
{
    System.out.println(" handle of method call arg named p_arr " + p_arr);
    int d[] = {30,40};
    p_arr=d;
    System.out.println(" handle of arr after assignment arr=d " + p_arr);
    System.out.println("                handle of array d " + p_arr);
    disp(p_arr);
    System.out.println(p_arr);
    disp(d);
}

static void disp(int a[])
{
    System.out.println(" Array value is... " + a[0]);
    System.out.println(" Array value is... " + a[1]);
}
}
```



```
MS-DOS Command Prompt
D:\myjava>java Passarr_a
Values before method call
Array value is... 3
Array value is... 4
    handle of arr before method call [I@73d6a5
handle of method call arg named p_arr [I@73d6a5
    handle of arr after assignment arr=d [I@111f71
    handle of array d [I@111f71
Array value is... 30
Array value is... 40
[I@111f71
Array value is... 30
Array value is... 40
Values after method call
Array value is... 3
Array value is... 4
    handle of arr after method call [I@73d6a5
```

An array itself will always hold a value. This value will be the reference (or handle). This represents the address in memory where the array starts. An array is a data type that stores a value. That value is a handle to memory. Just like a variable an array is copied by value (you just need to keep in mind that the value is a reference). Since we did not return anything from the method you can see that the handle of “arr” from the caller remains the same.

We could indeed change the handle of “arr” in the caller by returning “p\_arr” as demonstrated in the next program.

```

D:\myjava>java Passarr_b
Values before method call
Array value is... 3
Array value is... 4
    handle of arr before method call [I@70a1b4a8
handle of method call arg named p_arr [I@70a1b4a8
    handle of arr after assignment arr=d [I@723db4a8
        handle of array d [I@723db4a8
Array value is... 30
Array value is... 40
[I@723db4a8
Array value is... 30
Array value is... 40
Values after method call
Array value is... 30
Array value is... 40
    handle of arr after method call [I@723db4a8

```

Source Passarr\_b.java

```

class Passarr_b
{
    public static void main(String args[])
    {
        int arr[] = {3,4};
        System.out.println("Values before method call");
        disp(arr);
        System.out.println("    handle of arr before method call " + arr);
        arr = acceptarr(arr);
        System.out.println("Values after method call");
        disp(arr);
        System.out.println("    handle of arr after method call " + arr);
    }

    static int[] acceptarr(int p_arr[])
    {
        System.out.println(" handle of method call arg named p_arr " + p_arr);
        int d[] = {30,40};
        p_arr=d;
        System.out.println("  handle of arr after assignment arr=d " + p_arr);
        System.out.println("                    handle of array d " + p_arr);
        disp(p_arr);
        System.out.println(p_arr);
        disp(d);
        return p_arr;
    }

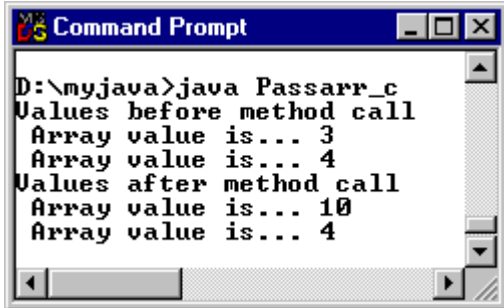
    static void disp(int a[])
    {
        System.out.println(" Array value is... " + a[0]);
        System.out.println(" Array value is... " + a[1]);
    }
}

```

In recapping, here is the bottom line that you need to understand at this point regarding passing arrays to a method. **An array is passed to a method by value. The array itself, not the contents of the array! The value represents the handle of the array that was passed.**

If you do not understand the concept above I would encourage you to again read the previous explanation before continuing on.

While the array is passed by value, we can still change the contents of the array we are passed from within our method. Please carefully review the code below.



```
D:\myjava>java Passarr_c
Values before method call
Array value is... 3
Array value is... 4
Values after method call
Array value is... 10
Array value is... 4
```

**Source** Passarr\_c.java

```
class Passarr_c
{
    public static void main(String args[])
    {
        int arr[] = {3,4};
        System.out.println("Values before method call");
        disp(arr);
        acceptarr(arr);
        System.out.println("Values after method call");
        disp(arr);
    }

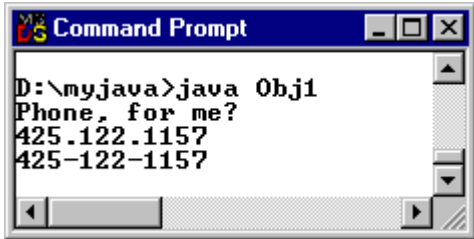
    static void acceptarr(int arr[])
    {
        arr[0] = 10;
    }

    static void disp(int a[])
    {
        System.out.println(" Array value is... " + a[0]);
        System.out.println(" Array value is... " + a[1]);
    }
}
```

If you closely examine this program you will see that we are not returning anything from the method. Yet, we have managed to change the first item in the array. Notice that in this program instead of assigning the contents of one array to the caller's array we have instead changed the value of an array element itself. We can see from this program that the method has access to the data elements within an array passed by a caller. This certainly will take some getting used to on your part.

## PASSING OBJECTS AS AN ARGUMENT

Guess what? Arrays are objects in Java. So we really have already covered this subject. But to expand it a bit we will create our own object and pass it in and out of a method.



```
Command Prompt
D:\myjava>java Obj1
Phone, for me?
425.122.1157
425-122-1157
```

Source Obj1.java

```
public class Obj1
{
    public static void main(String args[])
    {
        Phone p = new Phone();
        p = acceptobj(p);
        p.phone_nbr_us();
    }

    static phone acceptobj(phone p)
    {
        System.out.println("Phone, for me?");
        p.area_cde = "425";
        p.exchange = "122";
        p.extension = "1157";
        p.phone_nbr_europe();
        return p;
    }
}
```

Here we are creating an instance of the phone object. Then we are passing the object to our method named “acceptobj”. As you can see, this method can act on the phone object, and when it is done can return it back to the caller. Being able to pass an object into a method for processing opens up a whole new realm of possibilities that one just not have with non-OOP languages. One can create very complex data types and pass them around using this technique. Pretty cool stuff.