# Mr G's Java Jive

## #7: if Statements and the Mine Class

With this handout you'll learn about **if** statements and how to create your own **helper** class

### More Ways to Be an Idiot

In the last handout you learned how to do some basic idiot-proofing, thanks to the **gatling** class. But using the different methods found there won't solve all of your idiot problems. That's because idiots are remarkably resourceful and creative. Just when you've taken care of one stupid thing they can do, they come up with a totally new way to be stupid. You're going to try one of those other ways right now.

Run your **Converse** program again, and enter **your name** and the **year you were born** at the **first two prompts**. No problems so far.

Now, at the third prompt, enter the year **1914**. Check out the on-screen result.

### Being Negative About Idiots

OK, you didn't get a **type mismatch** this time, but you did get a result that was totally unrealistic. How can anyone be a negative number of years old? Clearly we need some way to handle the occasional dipstick who enters a year before they were born for the current year. Yes, we could just find a way to get the current year from the system, but that's a project for another program later on.

There are two ways we can take care of this problem: **locally** and **globally** (that is for just **this** program and for **all** programs). We'll start off with the **local** solution.

### if Statements

An **if** statement is a statement that looks at a **condition**, and makes a **decision** based on whether or not that condition is true. It looks sort of like the examples shown below:

```
if (condition) //a single-line if, indented under the condition
  {do stuff;}

if (condition) //a multi-line if, indented under the curly brackets
{//start if
   do stuff;
   do more stuff
   do still more stuff
   do stuff ad nauseum
}//end if
```

As you can see in the first example, since the result was only **one line**, I indented the whole thing below the condition without labeling the curly brackets. On the other hand, since the second one was **four lines**, I indented it the standard way.

### One Result

Technically speaking, you're only allowed **one result** for an if statement. How is it, then, that in the second example, the result has **four lines**? The answer has to do with a mall gift card.

Imagine that you've been given a $1000 gift card to a local mall. Cool, right? But this gift certificate comes with a few conditions. First of all, you can only use it for **one trip** to the mall. Second, once you leave the mall (even to go to the parking lot), the gift card **expires**. Third, you can only carry **one package** out of the mall.

Sounds tough, right? There's a ton of stuff you'd like to buy, but you can only bring out **one package**, and the gift card expires the moment you leave the mall. How do you get the most out of that gift card?

By getting the **mother of all shopping bags**. That shopping bag is **one package** that you're carrying out of the mall, even though it may have lots of other things inside. So now you can go to **Macy's**, **The Discovery Channel Store**, **The Apple Store**, **Borders**, and who knows what all else, shopping till you drop – as long as you can fit everything inside of that bag. Of course, I suppose you could also buy gift cards for each of those stores and come back later on.

With our **if** statement, the shopping bag is the **curly brackets**. Anything inside of them, no matter how many lines long, counts as part of the **one result**. Now that you know that, let's take a look at how to set up those conditions.

**Boolean Operators**

There's that word again – **boolean**. Boolean operators are based on the boolean math created by the mathematician **George Boole** (check him out on Wikipedia). You've seen most of these operators or their variants before. They are:

| | |
|---|---|
| == | is equal to |
| != | is not equal to |
| < | is less than |
| > | is greater than |
| <= | is less than or equal to |
| >= | is greater than or equal to |

Right now I know what you're thinking. You're wondering why == means **is equal to** when we already know that = means that. The difference is that the standard **single equal sign** is an **assignment operator**. In other words, it **assigns** the value on the right hand side to the variable on the left.

The **double equal sign**, on the other hand, is a **comparison operator**. It looks at what's on both sides and tries to figure out if they're equal to each other.

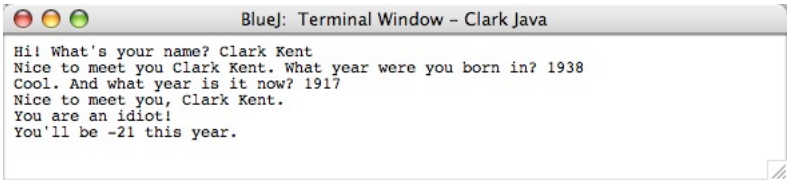
**Solving the Problem**

Now that you know a little about **if** statements and **boolean** operators, we can get to work on solving the problem of our latest batch of idiots. So open up **Converse** and modify your code so that it looks like the example below:

```
//give response
  System.out.println("Nice to meet you, "+name+".");
  if(age<0)
    {System.out.println("You are an idiot!");}
  System.out.println("You'll be "+age+" this year.");
```

Be very careful <u>not</u> to put a semicolon right after the **condition**. If you do that, Java will think that that's the end of the **if** statement and will do what's in the curly brackets regardless of whether the condition is true or false.

Now when you run the program, it should look like the example shown below:



**On the Other Hand…**

We've only solved half of the problem with this **if** statement. Actually, we haven't really solved it, we've merely commented on it. We've told the user that they're an idiot, but we still are seeing the invalid age. How can we solve that? With an **else**.

An **else** statement is the optional partner to an **if** that means "otherwise." It gives the program something to do if the **condition** of the **if** statement is **false**. Take a look at the modification I've made to **Converse** to see how that works.

```
//give response
  System.out.println("Nice to meet you, "+name+".");
  if(age<0)
    {System.out.println("You are an idiot!");}
  else
    {System.out.println("You'll be "+age+" this year.");}
```

What I did there was to make the sentence telling how old the user was part of the **else** statement. This means that it won't print at all unless the age is greater than 0. Try it out with good data and bad to see what happens.

**Think Locally, Act Globally**

OK, so now you know how to write an **if** statement, and you've successfully used one, along with an **else** to comment on the user's idiocy and suppress bogus results. The problem is that this is only a **local** solution. If this situation ever comes up again in other programs, you'll have to write similar lines of code there too. Besides, as I said earlier, you haven't really solved the problem, you've merely commented on it. The time has come now to finally solve it, and to do this we're going to create a new class. A **helper** class called **Mine**.

**Mine, Mine, Mine**

To create your new class, go to the **project window**, click in the **New Class** button, and tell BlueJ that you want to call the class **Mine**. Pay careful attention to the fact that it starts with an **uppercase M**. Now start off by entering the following lines of code:

```
//Mine
//personal helper class of generally useful stuff
//7.13.16 Clark Kent

public class Mine
{//start Mine
}//end Mine
```

**Parking Spaces**

The thing that makes **Mine** a **helper** class is that it won't have a **main** method. Instead, it'll have many methods that are **generally useful**. By **generally useful**, I mean that they might be useful in many programs that you'll write in the future, and not just a specific few. Because you're going to put many methods into this class, it would be a good idea to organize the class so you can find them easily. With that in mind, the next step is to create what I'll call **parking spaces**.

What to parking spaces have to do with Java programming? A lot, actually. If you've paid close attention over the years, you've probably noticed that there are two types of parking lots: those with painted lines and those without painted lines. The lots without painted lines are a little harder to park in, since people tend to make up their own spaces, and often don't leave the right amount of room on

either side for other cars. On the other hand, the lots that have lines make it perfectly clear to everyone where the spots are. This makes things a lot more organized and easy to understand.

We're going to start out by making a few parking spaces in **Mine** so that you know where your helper methods will go. And the thing that makes **Mine** better than a parking lot is that if you run out of spaces, you can just add more.

Now that you know that, modify your code so it looks like the example below:

```
//Mine
//personal helper class of generally useful stuff
//7.13.16 Clark Kent

public class Mine
{//start Mine
   //=====

   //=====

   //=====

   //=====

   //=====

   //=====

}//end Mine
```

Now you're ready to write your first helper method.

**The getHiInt Method**

The problem we had in **Converse** was that users could enter a number for the current year that was less than the year they were born in. The **getHiInt** method will solve that problem by checking the year they enter for **now** against the one they entered for **born**, and insisting that they enter a higher number. This will be **generally useful** because we're not going to write it specifically for years, but for any situation where you want to make sure that the user enters a number higher than one that was entered previously.

Start off by entering the following code:

```
public class Mine
{//start Mine
   //=====
      public static int getHiInt(int lo)
      {//start getHiInt
      }//end getHiInt
   //=====
```

The line **public static int getHiInt(int lo)** is the **method header**. It tells the **name** of the method (**getHiInt**), what kind of value it **returns** (an **int**), and what **types of values it needs to bring in** so it can do its work (an **int** that it will call **lo**). As far as that **public static** stuff goes, you can ignore that for now. All you need to know about that is that you need it.

Now look at the additions I've made in the example below:

```
public static int getHiInt(int lo)
     {//start getHiInt
        //variables
          int hi = lo;
     }//end getHiInt
```

We've created one new variable called **hi**, and we've made it equal to **lo**. Why? So that right from the start hi is too low, and that will force the next part of the program to run.

```
{//start getHiInt
        //variables
          int hi = lo;

        //while loop
          while(hi<=lo)
          {//start while
          }//end while
     }//end getHiInt
```

Here's something new – a **while loop**. A **while loop** is sort of like an **if** statement. An **if** does something **once if** the condition is true. A **while loop**, on the other hand, does something **many times while** the condition is true. In order to force users to enter a higher number we need to use a **while loop** just in case they're not only stupid, but obstinate. The condition for running the while loop is that the value of **hi** must be **less than or equal to** the value of **lo**.

Now let's take a look at the result of the **while loop**.

```
while(hi<=lo)
  {//start while
     System.out.print("Please enter a number higher than "+lo+": ");
     hi=gatling.getInt();
  }//end while
```

So as long as the value of **hi** is less than or equal to the value of **lo**, the user will be prompted to enter a higher number. Now there's just one thing we lack. Once we have a valid value for **hi** we need to **return** it to the calling method. We'll do that below:

```
//while loop
          while(hi<=lo)
          {//start while
             System.out.print("Please enter a number higher than "+lo+": ");
             hi=gatling.getInt();
          }//end while

          return hi;
     }//end getHiInt
```

With all this done, it's time to use what we've written. Let's go back to **Converse**.

**Using getHiInt**

In order for this to work, we need to change two lines of code in **Converse**. Take a look at the example below. Notice that we're only changing part of the line.

```
//get input
        System.out.print("Hi! What's your name? ");
        name=gatling.getLine();
        System.out.print("Nice to meet you "+name+". What year were you born in? ");
        born = gatling.getInt();
        System.out.print("Cool. And what year is it now? ");
        now = gatling.getInt() Mine.getHiInt(born);
```

Compile it, run it, and see what happens when you put in a year that's lower than the year you were born. If everything went well, you now have a method you can use any time you want to make sure that the integer input the user gives you is higher than some predefined lower value.

**Next?**

In the next handout we'll open up a ceramics shop and sell some pots.