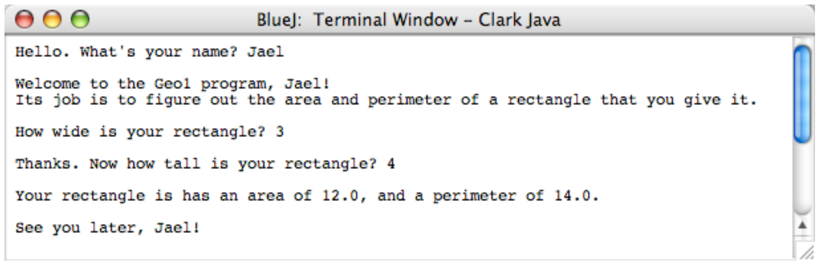
Mr G's Java Jive

#11: Formatting Numbers

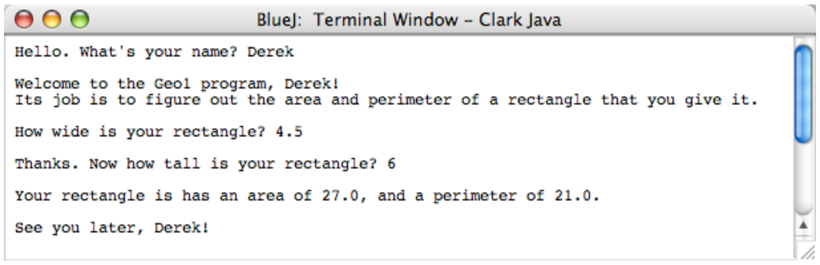
Now that we've started using **double** values, we're bound to run into the question of just how many decimal places we want to show. This where we get to deal with that issue.

Math Isn't Pretty

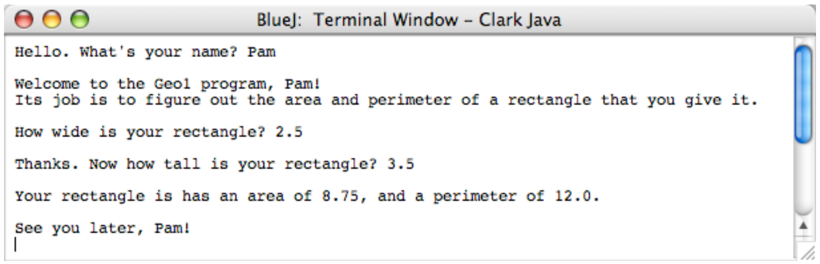
Back in the 70s (yes, way before any of you were born), comedian Steve Martin used to say that comedy isn't pretty. Well, neither is math. In fact, sometimes math is just plain old ugly – uglier than it really needs to be. Do you need proof? Check out the following four runs of Geo1:



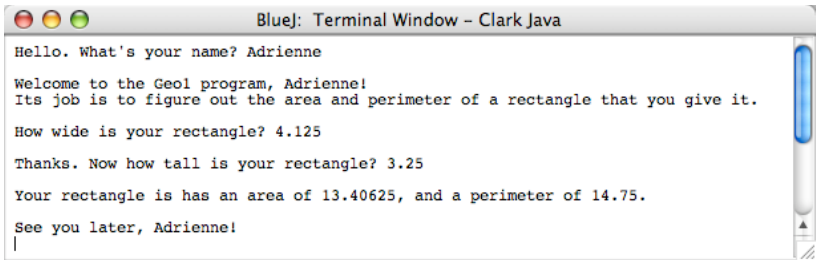
```
Hello. What's your name? Jael

Welcome to the Geo1 program, Jael!
Its job is to figure out the area and perimeter of a rectangle that you give it.

How wide is your rectangle? 3

Thanks. Now how tall is your rectangle? 4

Your rectangle is has an area of 12.0, and a perimeter of 14.0.

See you later, Jael!
```



```
Hello. What's your name? Derek

Welcome to the Geo1 program, Derek!
Its job is to figure out the area and perimeter of a rectangle that you give it.

How wide is your rectangle? 4.5

Thanks. Now how tall is your rectangle? 6

Your rectangle is has an area of 27.0, and a perimeter of 21.0.

See you later, Derek!
```



```
Hello. What's your name? Pam

Welcome to the Geo1 program, Pam!
Its job is to figure out the area and perimeter of a rectangle that you give it.

How wide is your rectangle? 2.5

Thanks. Now how tall is your rectangle? 3.5

Your rectangle is has an area of 8.75, and a perimeter of 12.0.

See you later, Pam!
```



```
Hello. What's your name? Adrienne

Welcome to the Geo1 program, Adrienne!
Its job is to figure out the area and perimeter of a rectangle that you give it.

How wide is your rectangle? 4.125

Thanks. Now how tall is your rectangle? 3.25

Your rectangle is has an area of 13.40625, and a perimeter of 14.75.

See you later, Adrienne!
```

In the first run, Jael entered two **whole number** values (remember that integers are a subset of reals), and her results came out looking quite nice. There was an unnecessary **decimal place** and a **0** with her **whole number** results, but what she got looked pretty good.

In the second run, Derek entered a **real number** and a **whole number**, and his results came out looking fairly decent too. In fact, they came out to two **whole numbers** (do you know why the math just happened to work out that way?). No problem here.

In the third run, Pam entered two **real numbers**, and her results still didn't look too bad. But take a look at Adrienne's. Her **area** results came out with five decimal places. That's not really bad as things go, but maybe she didn't really want that many. Maybe all anyone really needs here is three places (especially when you get into results with **repeating decimals**). To set decimal places the way we want them, we need to learn how to use the **DecimalFormat** class.

Much More Than You Need To Know

If you want to find out all there is to know about the **DecimalFormat** class, you could check out the documentation page at <**http://java.sun.com/j2se/1.5.0/docs/api/java/text/DecimalFormat.html**>. But that's some pretty dense reading for a beginner – in fact, it's pretty dense even for me, so I'll give you the condensed, **Reader's Digest** version.

Importing the Tools

The **DecimalFormat** class contains tools that let you format a number based on a certain **String pattern**. But you can't just use this "straight out of the box." If you tried creating a decimal format of your own right now, it wouldn't compile. That's because you need to **import** the tools you need to do it. So make the change shown below to Geo1:

```
//Geo1
//a program to figure out the area and perimeter of a rectangle
//8.10.06 Clark Kent

import java.text.*;  //needed to do number formats

public class Geo1
{//start class
```

The line **import java.text.*** tells the compiler to import everything from the class **java.text**. The ***** character is a **wildcard** that means "everything." It's a lot easier that way. From now on, you'll put this line at the beginning of all of your programs.

Creating A Decimal Format

Now that we've told the compile to import the necessary external code, it's time to create a simple number format that sets everything to **three fixed decimal places**. We'll call it **d3f** (for **decimals**, **three**, and **fixed**). Check out the changes below:

```
//variables
  String name;
  double height, width, area, perimeter;

//formats
  DecimalFormat d3f = new DecimalFormat("0.000");

//get input
```

The first thing you should notice here is that we've added a new section, just below the one for **variables**. It's called **formats**. Most Java texts put the formats in with the variables, but I figured it would be a good idea to keep them separate, for the sake of clarity.

The second thing you should notice is how we create a new number format. Did you catch that we're creating a new **DecimalFormat** and that **DecimalFormat** is capitalized? Did you remember that that means it's an **object**?

So we've said that we want to create a new **DecimalFormat** object called **d3f**, and that it'll be a new **DecimalFormat** object based on the string **"0.000"**. What does this mean?

It means that our numbers will always show **three** decimal places. It also means that there will always be **at least one whole number place** – even if it's **0**.

Using Our Number Format

Just creating the number format doesn't solve the problem for us. We have to use it, and the easiest way to use it right now is to create two new **String** objects that the formatted numbers will go into. Check out the changes shown below:

```
//variables
  String name, astring, pstring;
  double height, width, area, perimeter;
```

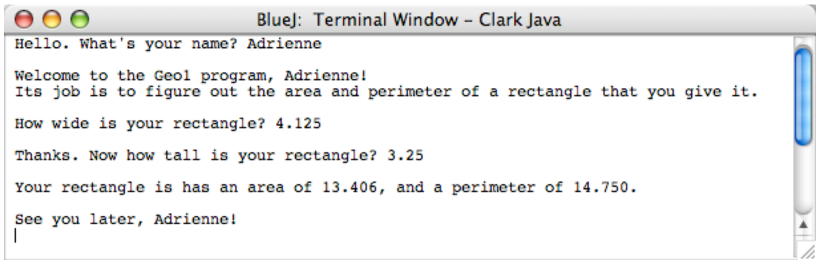
Now that we have these Strings, we need to add two lines to the **calculations** section:

```
//do calculations
  area=height*width;
  perimeter=2*(height+width);
  astring=d3f.format(area);
  pstring=d3f.format(perimeter);
```

Those two new lines mean to apply the **d3f** format to the **value** and put them into the **String**. Now all we have to do is change our final output code, and we can see some impressive results. Try this:

```
//give output
  System.out.print("\nYour rectangle is has an area of "+~~area~~astring+", ");
  System.out.println("and a perimeter of "+~~perimieter~~pstring+".");
  System.out.println("\nSee you later, "+name+"!");
```

Compile it and run it with Adrienne's previous values of **4.125** and **3.25**, and see what a difference it makes!



```
Hello. What's your name? Adrienne

Welcome to the Geo1 program, Adrienne!
Its job is to figure out the area and perimeter of a rectangle that you give it.

How wide is your rectangle? 4.125

Thanks. Now how tall is your rectangle? 3.25

Your rectangle is has an area of 13.406, and a perimeter of 14.750.

See you later, Adrienne!
```

That's a whole lot better, isn't it? Now let's take a closer look at our formatting options.

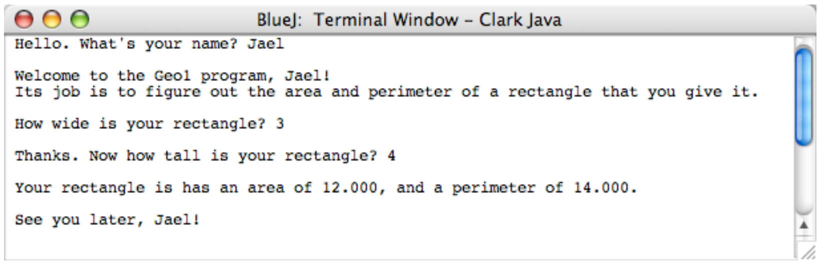
Three Simple Rules For Formatting Numbers

As I said before, I'm not going to make you read the entire document on the **DecimalFormat** class. You don't need that level of detail. All you need to know are some simple rules about creating a format of your own. Here they are.

1. All decimal formats are defined as **String patterns**. This means that you can define the format **directly** or **indirectly**. More on that later.
2. A **0** in the format pattern means a **required place** that, if necessary, will show up as **0**.
3. A **#** in the format pattern means an **optional place**, that won't show up at all if it's not a **significant digit**.

A New Number Format: d3o

Now that you've created and used a number format that sets up a **fixed** number of decimal places, it's time to try one that has **optional** ones. We'll also take this opportunity to learn what I meant when I said that the patterns could be defined either **directly** or **indirectly**. But first let's run the program again using the two **whole number** values that Jael started off with. The output is shown below:



BlueJ: Terminal Window – Clark Java

```
Hello. What's your name? Jael

Welcome to the Geol program, Jael!
Its job is to figure out the area and perimeter of a rectangle that you give it.

How wide is your rectangle? 3

Thanks. Now how tall is your rectangle? 4

Your rectangle is has an area of 12.000, and a perimeter of 14.000.

See you later, Jael!
```

Even though they're both **whole numbers**, the **area** and **perimeter** values are shown with **three decimal places**. Seems like a bit of a waste to me. We'll fix this by creating a new format: **d3o** (for **decimals**, **three places**, and **optional**. Check out the following code changes:

```
//variables
  String name, astring, pstring;
  String fmt = "#.###";
  double height, width, area, perimeter;

//formats
  DecimalFormat d3f = new DecimalFormat("0.000");
  DecimalFormat d3o = new DecimalFormat(fmt);
```

The first change we made was to create a **String** object called **fmt**. This object contains the **pattern** we want to use to define our new format.

The second change we made was to define the new format, **d3o**, as being a new **DecimalFormat** based on **fmt**. This is different from the way we defined **d3f** because here we're defining the format **indirectly**, by calling a previously created string with the pattern in it. The format **d3f** was defined **directly**, in one step.

Pay careful attention to the fact that when you define a format **directly** the **argument** in the parentheses has to be in **quotation marks** since you're trying to pass a string into it. On the other hand, when you define a format **indirectly**, you don't put quotation marks around the argument. That's because you're passing a String object into it that's already been defined once using the quotation marks. Putting quotation marks around the string name will give you some rather "interesting" results.

Advantages of Defining Formats Indirectly

Why would anyone want to define their formats indirectly, since it involves an extra step? There actually are some advantages, although I'll admit that most of the time it actually is easier just to define them directly. The one advantage that defining your formats indirectly gives you is that it allows you to redefine formats "on the fly." If you're really dying to find out what I'm talking about right now, take a look in the **gatling.java** class at the second **rAlign** method. That's all I'll say about that for now.

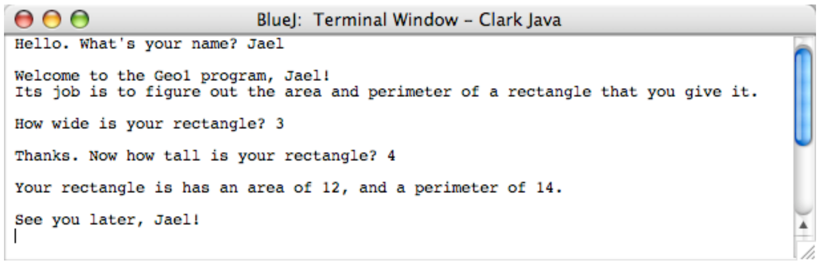
Let's go to the next page to actually use our new format.

Using Our New Format

This is going to be really simple. just make the following two changes in your code:

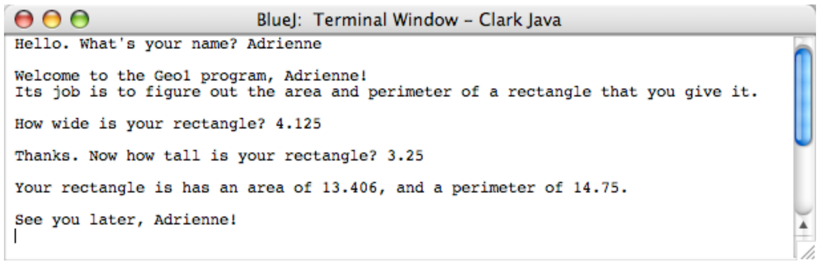
```
//do calculations
  area=height*width;
  perimeter=2*(height+width);
  astring=d3~~f~~o.format(area);
  pstring=d3~~f~~o.format(perimeter);
```

In case that was unclear, I simply changed the two instances of **d3f** to **d3o**. Now when you compile and run with the values **3** and **4**, you should get output like this:



```
Hello. What's your name? Jael

Welcome to the Geo1 program, Jael!
Its job is to figure out the area and perimeter of a rectangle that you give it.

How wide is your rectangle? 3

Thanks. Now how tall is your rectangle? 4

Your rectangle is has an area of 12, and a perimeter of 14.

See you later, Jael!
```

Since the decimal places weren't needed, they weren't shown. But what about Adrienne's values? Here's what happens when we run them through with the new format:



```
Hello. What's your name? Adrienne

Welcome to the Geo1 program, Adrienne!
Its job is to figure out the area and perimeter of a rectangle that you give it.

How wide is your rectangle? 4.125

Thanks. Now how tall is your rectangle? 3.25

Your rectangle is has an area of 13.406, and a perimeter of 14.75.

See you later, Adrienne!
```

Here, the **area** maxes out at **three** decimal places, but the **perimeter** maxes out at **two**, and there are different reasons for each. **Area** maxes out at **three** places because our format was set up to show **at most** three decimal places. **Perimeter** maxes out at **two** places because the third place is a **0**, which in this case is not a **significant digit**, so it gets dropped.

Fixed vs Optional Places

So now that you know how to set up formats for both, how do you know when to use **fixed** or **optional** decimal places? Unfortunately (or maybe fortunately), as with many things in life, there is no hard and fast rule. A lot just depends on how you want it to look and how many decimal places you want a number to go to.

Obviously, if you're working with money, you'll always want **two fixed** decimal places (unless you're working with large amounts of money, and then, who cares about the pennies). However, aside from that obvious consideration, the number of decimal places, and whether or not they're optional, is pretty much up to you – within reason (only the most geeky users want to see more than three or four decimal places).

It's Not Perfect

As good as it is, to my mind, the **NumberFormat** class and its methods aren't perfect. It would be perfect if it let you set up the total **column width** for the number (**C++** lets you do that). But since it doesn't, I took matters into my own hands, and wrote a few methods that will do that for you. We'll look at them later – much later.

This page intentionally left **almost** blank