Mr G's Java Jive

#9: Two General Methods – Containers and Remaining

When we first wrote the Pots program it had one tiny little problem – it was only capable of shipping full boxes. That was really inefficient and needs to change. This is were we fix that problem.

Short-Term Solution or Long-Term Solution?

As with the problem in the **Converse** program, where users could enter a year before the one they were born in, the problem of wasting boxes when we ship pots has both a short-term, **specific** solution and a long-term, more **general** one. We'll look at the short-term solution first as a way to work on the long-term one. But first we have to consider **maximums** and **minimums**.

Of Max and Min

When we first wrote this program, we established three **final values** for how much each size box could hold. We called them **LG**, **MD**, and **SM**. It's time to rethink that now. These three values really represent the **maximum** that any of these boxes can hold, but what about the **minimum**? What's the **least** number of pots you'd want to put into any of these boxes? With this in mind, we need some **final values** that represent the **minimum** number of pots for each size box. But before we do that, let's change the names of our current **final values** to represent our new reality. Take a look at the code shown at the top of the next page:

```
//variables
  final int LGMAX = 9;
  final int MDMAX = 4;
  final int SMMAX = 1;
  String name;
  int pots, lgnum, mdnum, smnum, left;
```

Now that you know the maximum number of pots you can fit into a certain size box, it's time to consider the **minimum**. It seems that in order to save boxes and postage, the minimum capacity of a box should be just one more than the maximum of the size just below it. So the minimum for a **large** box would be **5** and for a **medium** would be **2**. The minimum for a **small** box would be the same as its maximum: **1**.

Now that we know that, let's make the following additions to our program.

```
//variables
  final int LGMAX = 9;
  final int MDMAX = 4;
  final int SMMAX = 1;
  final int LGMIN = 5;
  final int MDMIN = 2;
  final int SMMIN = 1;
  String name;
  int pots, lgnum, mdnum, smnum, left;
```

Once you've done this, you'll have to carefully go through your program, find every place where you used **LG**, **MD**, or **SM**, and change them to **LGMAX**, **MDMAX**, and **SMMAX**.

The Short-Term Solution

Now that we have our three new finals in place, it's time to look at the short-term solution to our problem of box waste. If you take a look at the changes shown below, you'll see that I've done it here just for the large boxes. It says that if the number of pots leftover is at least the minimum needed for a large box, then do two things. The first is to add one more box. The second is to say that there are no more pots left over (after all, they've all been put into that new box).

```
//do calculations
        lgnum=pots/LGMAX;
        left=pots%LGMAX;
        if (left>=LGMIN)
        {//start large if
           lgnum++;
           left=0;
        }//end large if
        mdnum=left/MDMAX;
        left=left%MDMAX;
        smnum=left/SMMAX;
        left=left%SMMAX;
```

But do we really want to write this two more times, once for medium boxes and once for small? Is there maybe a better way to do this? A long-term solution that's a lot more general? You bet there is.

The Long-Term Solution Part 1: Mine.containers

Since there will be many times when you'll want to try to put things into boxes of one kind or another, it's better to write a method for it that can be called in any program you write. We'll call this one **containers** and it's going into your **Mine** class, in the parking space just below **getHiInt**. Check out the example below:

```
}//end getHiInt
   //=====
      public static int containers(int items, int min, int max)
     {//start containers
     }//end containers
   //=====
```

This very simple beginning shows us that you need to bring in three pieces of information in order to make this work: an int value to show the number of **items**, an int value to show the **min**imum number of items to put into the container, and an int value to show the **max**imum that can fit into that container.

The next example **declares** and **defines** one **local variable** that we'll use to get the result:

```
//=====
  public static int containers(int items, int min, int max)
  {//start containers
     //variables
       int result = items/max;
  }//end containers
//=====
```

We've said here that the beginning value of **result** will be equal to the number of **items** divided by the **maximum capacity** of the container. But suppose there are enough items left over to make it worth using another container of this size? That gets taken care of with the **if** statement shown on the next page.

```
//=====
  public static int containers(int items, int min, int max)
  {//start containers
     //variables
       int result = items/max;

     //do we need another container?
       if(items%max>=min)
         {result++;}
  }//end containers
//=====
```

Here we've said that if the number of items left over is greater than the minimum capacity of the container, then we should add another container.

What's that **result++** thing all about? It's Java's very simple way of saying "add one to the previous value." We'll call that the **increment operator**. Similarly, there's a **decrement operator**, and that's **--**.

We're almost done with our **containers** method. All we need to do now is to **return** the value to the calling program or method. Make the following changes, and you should be all set:

```
//=====
  public static int containers(int items, int min, int max)
  {//start containers
     //variables
       int result = items/max;

     //do we need another container?
       if(items%max>=min)
         {result++;}

     //return the result
       return result;
  }//end containers
//=====
```

Once you've done this, it would be a good idea to try to compile. This is where you'll find out if you've made any **syntax** errors. We won't be able to look for **logic** errors until we actually run with this method, and we're not ready for that yet.

The Long-Term Solution Part 2: Mine.remaining

The **containers** method was only half of the solution. That's because we not only need to know how many containers to use, but also how many items are **left over**. This sounds like a job for a whole new method, in a parking space of its own just below **containers**. We'll call this one **remaining**. Check out the beginning code shown below:

```
}//end containers
   //=====
     public static int remaining(int items, int min, int max)
     {//start remaining
     }//end remaining
   //=====
```

Does this look strangely familiar? It should. That's because you'll need to bring the very same information into **remaining** that you for **containers**. Now let's move on to the next steps.

```
//=====
  public static int remaining(int items, int min, int max)
  {//start remaining
     //variables
       int result = items%max;
  }//end remaining
//=====
```

Once again, this looks very similar to what we've already done in **containers**, but the difference here is that it's saying that the value of **result** is equal to the **remainder** of **items** divided by **max**. But suppose those leftover items were able to fit into another container of that size? That's where the next set of changes come in:

```
//=====
  public static int remaining(int items, int min, int max)
  {//start remaining
     //variables
       int result = items%max;

     //did they fit into another container?
       if(items%max>=min)
         {result=0;}
  }//end remaining
//=====
```

Here we've said that if the number of items left over is greater than the minimum capacity of the container, then there really aren't any items left over. That was simple enough. Now let's return the result and we're almost done!

```
//=====
  public static int remaining(int items, int min, int max)
  {//start remaining
     //variables
       int result = items%max;

     //did they fit into another container?
       if(items%max>=min)
         {result=0;}

     //return the result
       return result;
  }//end remaining
//=====
```

Once again, once you've done this, it would be a good idea to try to compile. If it compiles properly, then we're ready to use both methods in our **pots** program. We'll take care of that on the next page, where there's a bit more room.

Using containers and remaining

Now that we've written our long-term solutions to the problem of box waste, let's put them to use, starting off with the following changes in **pots**:

```
//do calculations
  lgnum=pots/LGMAX;
  left=pots%LGMAX;
  if (left>=LGMIN)
  {//start large if
     lgnum++;
     left=0;
  }//end large if
  lgnum=Mine.containers(pots,LGMIN,LGMAX);
  left=Mine.remaining(pots,LGMIN,LGMAX);
  mdnum=left/MDMAX;
  left=left%MDMAX;
  smnum=left/SMMAX;
  left=left%SMMAX;
```

That takes care of replacing our short-term solution for the **large** boxes with our much more elegant long-term one. Now let's finish the job by applying our new solution to the **medium** and **small** ones:

```
//do calculations
  lgnum=Mine.containers(pots,LGMIN,LGMAX);
  left=Mine.remaining(pots,LGMIN,LGMAX);
  mdnum=Mine.containers(left,MDMIN,MDMAX);
  left=Mine.remaining(left,MDMIN,MDMAX);
  smnum=Mine.containers(left,SMMIN,SMMAX);
  left=Mine.remaining(left,SMMIN,SMMAX);
  mdnum=left/MDMAX;
  left=left%MDMAX;
  smnum=left/SMMAX;
  left=left%SMMAX;
```

Compile it and run it with **17** pots. Run it again with **7**, **3**, and **11**. Are there any wasted boxes this time?

What Now?

Well, the program still isn't perfect. Some idiot user could say that they wanted **–27** pots. You could also fix the program up so that it **didn't** show you the boxes you didn't need and only showed you the boxes that you **did**. You could have it give you the results in a **full sentence**. All of these things are easily doable, but won't be covered here. Why not? Because I have to let you do some thinking on your own and give you something to do for homework!

What's Next

In handout **#10**, we finally get to work with some **real numbers** by writing a program that uses the **double** type. And that brings its own little set of issues along with it.

This page intentionally left **almost** blank