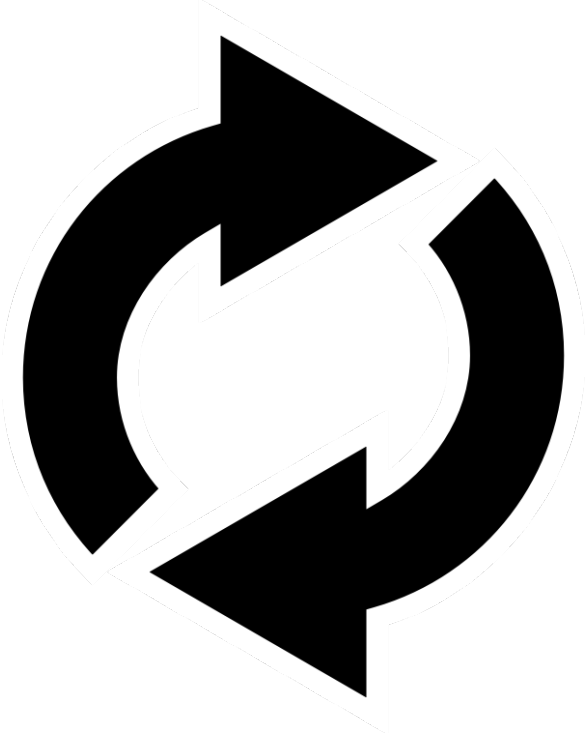
Chapter 3

Repeating Code

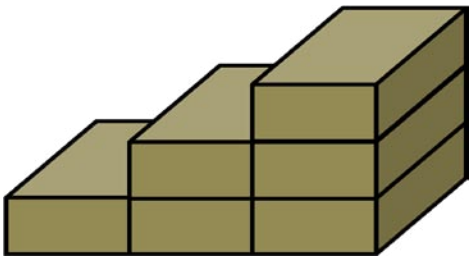
What is in this Chapter ?

When programming, it is often necessary to repeat a selected portion of code a specific number of times, or until some condition occurs. We will look here at the **FOR** and **WHILE** loop constructs that are available in most programming languages.



3.1 Repeating Code Using For Loops

Assume that we want to stack concrete slabs on top of each other to form a staircase. Write a program that will determine how many slabs would be needed to create a staircase **n** stairs high. So, for example, the staircase shown here is 3 stairs high and would require 6 slabs.



To begin, you should realize that **n** may be a very large number. What is our mathematical model ?

Noticing how the number of slabs for each stair increases by one each time, here is a formula that shows how many slabs we need:

1 + 2 + 3 + 4 + ... + n

Some of you may realize that this value can be computed as **n(n+1)/2**. However, assume that we are unaware of that nifty formula. How would you go about solving this problem ? You might realize that some kind of counter is required (i.e., a variable) and that we need to keep adding an increasingly large integer to the count.

Well, in JAVA, we *do* have a way of repeating code using something called a ***FOR loop***. Here is the solution that uses a **for** loop:

```
int total = 0;
for (int height = 1; height<=n; height++) {
    total = total + height;
}
System.out.println(total + " slabs are needed for staircase of " + n + " stairs.");
```

Notice that the **for** loop has parentheses **( )** and then is followed by braces **{ }** which contains the body of the loop (i.e., the code that is to be repeated). Let us take a look at what is inside the **( )** parentheses.

Notice that it declares an **int** variable called **height** and gives it a value of **1** to begin. This variable **height** is called a ***loop variable*** and in our example it represents a *counter*.

After the first semi-colon **;** there is a conditional JAVA expression **height<=n** … which is called the ***loop stopping condition***. That is, the loop will keep repeating as long as our counter is less than or equal to the value of **n**. The moment the **height** reaches **n+1**, the loop stops and our answer is printed.

After the next semi-colon **;** there is the ***loop update expression*** code **height++** which is evaluated each time the loop completes a cycle. In our case, each time through the loop we just need to increase the counter by **1**.

Notice that we can even use the value of the **height** variable within the loop. In general, a loop variable can be used any time within the loop but it cannot be used outside the loop body. The only exception to this is if we declare the type of the loop variable outside the loop as follows:

```
int height;
for (height = 1; height<=n; height++) {
    total = total + height;
}
```

Here is another example. Suppose that we wanted to print out the odd numbers from **1** to **100**. How could we do this ? Do you know how to check whether or not a number is odd ?

We can check if the remainder after dividing by two is zero. The modulus operator **%** gives the remainder after dividing, so we do **n%2** on number **n**. We just need to put this into a **for** loop:

```
for (int n=1; n<=100; n++) {
      if ((n%2) > 0)
            System.out.println(n);
}
```

Notice that we can use the same counter (but called it **n** this time). Then in the loop we just check if the modulus is non-zero and print out the number in that case since it would be odd.

We could eliminate the **if** statement by simply counting by twos (starting at 1) as follows …

```
for (int n=1; n<=100; n=n+2) {
      System.out.println(n);
}
```

The above code too will print out only the odd numbers from **1** to **100** since now the counter **n** increases by two each time, thereby skipping over all the even numbers. You should realize that the update expression can be any JAVA code.

Here is another example that prints out all the even numbers backwards from **100** to **1**:

```
for (int n=100; n>0; n=n-2) {
      System.out.println(n);
}
```

Notice how we started **n** at a higher value now and that we subtract by two each time (i.e., **100**, **98**, **96**, etc..). Also, notice that the *stopping condition* now uses a **>** instead of **<=** (i.e., as long as **n** is above zero we keep decreasing it by **2**).

What would happen if the stopping-expression evaluated to **false** right away as follows:

```
for (int n=1; n>=100; n++) {
      System.out.println(n);
}
```

In this case, **n** starts at **1** and the stopping condition determines that it is not greater than or equal to **100**. Thus, the loop body never gets evaluated. That is, the **for** loop does nothing … your program ignores it.

A similar unintentional situation may occur if you accidentally place a semi-colon **;** after the round brackets by mistake:

```
for (int n=1; n<=100; n++); {
      System.out.println(n);
}
```

In this situation, JAVA assumes that the **for** loop ends at that semi-colon **;** and that it has no body to be evaluated. In this case, the body of the loop is considered to be regular code outside of the for loop and it is evaluated once. Hence JAVA "sees" the above code as:

```
for (int n=1; n<=100; n++){
}
System.out.println(n);
```

One last point regarding **for** loops is that you do not need the braces around the loop body if the loop body contains just one JAVA expression:

```
for (int n=100; n>0; n=n-2)
      System.out.println(n);
```

In that case though, you should still indent your code so that it is clear what is in the loop.

Example:

Suppose that you wanted to ask the user for 5 exam marks and then print the average exam mark.

You might write a program that looks like this:

```
int  n1, n2, n3, n4, n5;

System.out.print("Enter exam mark 1:  ");
n1 = new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 2:  ");
n2 = new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 3:  ");
n3 = new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 4:  ");
n4 = new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 5:  ");
n5 = new Scanner(System.in).nextInt();

System.out.println("The average is " + ((n1+n2+n3+n4+n5)/5));
```

The above code gets all the exam marks first and stores them into variables… afterwards computing the sum and average. Instead of storing each number separately, we can add each number to an accumulating **sum** as follows:

```
int  sum;

System.out.print("Enter exam mark 1:  ");
sum = new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 2:  ");
sum = sum + new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 3:  ");
sum = sum + new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 4:  ");
sum = sum + new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 5:  ");
sum = sum + new Scanner(System.in).nextInt();

System.out.println("The average is " + (sum/5));
```

While this code may work fine, what would happen if we needed 100 numbers ? Clearly, some part of the code is repeating over and over again (i.e., adding the next number to the **sum**). Here is how we could modify our program to get **100** numbers by using a **for** loop:

```
int  sum = 0;

for (int count=1; count<=100; count++) {
      System.out.print("Enter exam mark " + count + ":  ");
      sum = sum + new Scanner(System.in).nextInt();
}

System.out.println("The average is " + (sum/100));
```

In our example, what if we did not know how many time to repeat (i.e., we don't know how many exam marks there will be) ? Well, in that case we can ask the user of the program for the total number as follows …

```
import java.util.Scanner;

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

      int  nums, sum;
      System.out.println("How many exam marks do you want to average ?");

      nums = new Scanner(System.in).nextInt();
      sum = 0;

      // Get the numbers one at a time, and add them
      for (int count=1; count<=nums; count++) {
          System.out.print("Enter exam mark " + count + ":  ");
          sum += new Scanner(System.in).nextInt();
      }
      System.out.println("The average is " + (sum / nums));
    }
}
```

Notice that the program is now flexible in the number of exam marks that it is able to average. Here is an example of the output:

```
How many exam marks do you want to average ?
5
Enter exam mark 1:    10
Enter exam mark 2:    67
Enter exam mark 3:    43
Enter exam mark 4:    96
Enter exam mark 5:    20
The average is 47
```

Example:

Consider a common situation in which you want to count the total of a set of values. For example, assume that you are having a pizza lunch and you want your employees to tell you how many slices they each want. Your goal is to determine how many large pizzas to buy (assume 8 slices per large pizza … and all just plain pepperoni). Write a program that does this by first asking how many people will be at the pizza lunch and then asking for their desired slice counts.

Here is the structure of the program:

```
import java.util.Scanner;

public class PizzaLunchProgram {
    public static void main(String args[]) {
        int  numEmployees;

        // Find out how many people will be at the lunch
        System.out.print("How many employees will be at the lunch ? ");
        numEmployees = new Scanner(System.in).nextInt();

        // Now ask each for the number of slices that they want and add to total
        ...

        // Now find out how many whole large pizzas to buy
        ...
    }
}
```

Now we need to repeatedly ask each employee how many slices they want. Using a **for** loop, we need to do this **numEmployees** number of times:

```
    total = 0;
    for (int i=0; i<numEmployees; i++) {
        System.out.print("How many slices for employee " + i + " ?  ");
        slices = new Scanner(System.in).nextInt();
        total += slices;
    }
```

Notice how **i** is used as an *index* to count. This is the most popular loop variable name. Finally, we need to determine how many pizzas are needed by dividing by the number of slices per pizza (i.e., 8) and making sure to round up to the next whole number of pizzas. Here is the final program:

```
import java.util.Scanner;

public class PizzaLunchProgram {
    public static void main(String[] args) {
        int  numEmployees, slices, total;

        // Find out how many people will be at the lunch
        System.out.print("How many employees will be at the lunch ? ");
        numEmployees = new Scanner(System.in).nextInt();

        // Now ask each for the number of slices that they want and add to total
        total = 0;
        for (int i=1; i<=numEmployees; i++) {
            System.out.print("How many slices for employee " + i + " ?  ");
            slices = new Scanner(System.in).nextInt();
            total += slices;
        }

        // Now find out how many whole large pizzas to buy
        System.out.println("You need to order " +
                           Math.ceil(total /8.0) + " large pizzas.");
    }
}
```

Notice that we divided by **8.0** (i.e., a double) instead of just **8** (i.e., an int). That is because if we divide an int by another int, we get an int result which discards any remainders. For example, **20/8** would give us **2** instead of **2.5**. That means we would order **2** pizzas, when we really needed **2** and a half pizzas. The **Math.ceil()** function brings the **2.5** up to the nearest whole number of **3** ... so we would order **3** large pizzas ... having a few leftover slices.

Homework:

Modify the above code to show the average number of pizza slices that each person will eat ?

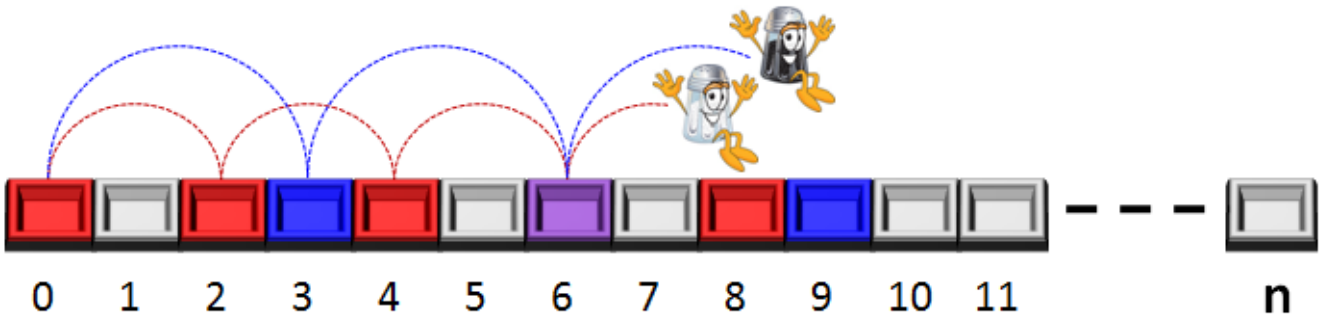
Example:

Another kind of counting involves searching through some items to enumerate (i.e., count) them. Perhaps we need to count the number of items that match some kind of search criteria. For example, how would we write a program that asks for the ages of **n** people and then indicates how many of them are adults (i.e., 18 or over) ?

```
import java.util.Scanner;

public class AdultCountingProgram {
    public static void main(String[] args) {
        int  numPeople, age, adults;

        // Find out how many people there are
        System.out.print("How many people are there ? ");
        numPeople = new Scanner(System.in).nextInt();

        // Now count the adults
        adults = 0;
        for (int i=1; i<=numPeople; i++) {
            System.out.print("How old is person " + i + " ?  ");
            age = new Scanner(System.in).nextInt();
            if (age >= 18)
                adults++;
        }
        System.out.println("There are " + adults + " adults.");
    }
}
```

Example:

Here is an example that involves a bit more thinking. Imagine that you are creating a game where two players are moving along a one-dimensional grid (i.e., path). One player always jumps forward **2** steps at a time, while the other always jumps forward **3** steps at a time. Develop an algorithm that figures out how many grid locations have not been landed on if the two players start at the same location (i.e., 0) and they each jumped up to grid location **n**.



How do we approach this problem ? First, examine the grid locations covered by player 1:

0, 2, 4, 6, 8, 10, 12, 14, … (seems to be the even numbered locations)

and those covered by player 2:

0, 3, 6, 9, 12, 15, 18, 21, … (seems to be the locations that are multiples of 3)

We could try to figure out a formula… but can we do this with some kind of loop counter ? What if we examine each location at a time … can we determine by the location number whether or not it would be landed on ?

```
int spotsNotLandedOn = 0;

for (int i=0; i<=n; i++) {
    if (((i%2) != 0) && ((i%3) != 0))
       spotsNotLandedOn++;
}
```

How could we change the code so that we display a list of all the locations that the two players meet at along the way ?

We would just need to find the locations that were multiples of 2 or 3:

```
for (int i=0; i<=n; i++) {
    if (((i%2) == 0) && ((i%3) == 0))
       System.out.println(i);
}
```

3.2 Nested Loops

It is sometimes necessary to have loops within loops. Whenever we have a loop placed inside of another loop, this is called a ***Nested Loop***. Nested loops are often used when 2-dimensional tables need to be traversed or displayed.

Example:

Write a program that displays the following table of values by using nested **for** loops:

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
6 5 4 3 2 1
7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
```

To do this, we can display one line at a time. Notice that each line has numbers decreasing in their values. So, for example, we can do the last line very simply as follows:

```
for (int i=9; i>0; i--)
    System.out.print(i + " ");
```

Notice that the index value goes backwards ... starting at 9 and decreasing to 1. Also notice that we are not using **println()**, but **print()** instead, so that consecutive values are displayed on the same line, separated by spaces. We can place the above code in a for loop so that it repeats 9 times. Notice to the right, however, the table that would be produced:

```
for (int r=1; r<=9; r++) {

    for (int i=9; i>0; i--)
        System.out.print(i + " ");

    System.out.println();
}
```

```
9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
```

This is an example of a nested loop. Notice that the ***outer*** loop uses a different loop variable (i.e., **r**) than the ***inner*** loop (i.e., **i**). This code, however, does not produce exactly what we want. Notice as well the use of **System.out.println()** AFTER the inner loop, which prepares for displaying onto the next line. Thinking backwards ... we need to do the following for the 2nd last line in the table:

```
for (int i=8; i>0; i--)
    System.out.print(i + " ");
```

This code is identical to what we have for the inner loop ... except that the starting loop value of **8** is used instead of **9**. I think that you can see that each row, when going up from the

bottom, has a starting value of one less than the row below it. Hence, instead of fixing a value of 9 as the starting value for the inner loop, we can have it be adjustable according to the row number as follows:

```
for (int r=1; r<=9; r++) {

    for (int i=r; i>0; i--)
        System.out.print(i + " ");

    System.out.println();
}
```

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
6 5 4 3 2 1
7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
```

Now we have the result that we want.

Example:

This one is a bit trickier. Write a program that displays the following table of values by using nested **for** loops (ignore the bold values for the moment):

```
1 2 3 4 5 6 7 8 9
2 1 2 3 4 5 6 7 8
3 2 1 2 3 4 5 6 7
4 3 2 1 2 3 4 5 6
5 4 3 2 1 2 3 4 5
6 5 4 3 2 1 2 3 4
7 6 5 4 3 2 1 2 3
8 7 6 5 4 3 2 1 2
9 8 7 6 5 4 3 2 1
```

At first glance, you may notice that the bottom left triangle is the same as our previous example. So, maybe we can start with that code as a template. But what is different ? Well, it appears that non-bold values are almost the reverse of our inner loop. So perhaps, after displaying one line from our inner loop, we can have a second inner loop to continue that line by increasing the numbers again:

```
for (int r=1; r<=9; r++) {

    for (int i=r; i>0; i--)
        System.out.print(i + " ");

     // for (???)
     //    System.out.print(???);

    System.out.println();
}
```

The second inner loop will simply require counting from 2 to (10 - **r**) as follows:

```
    for (int r=1; r<=9; r++) {

        for (int i=r; i>0; i--)
            System.out.print(i + " ");

        for (int i=2; i<=10-r; i++)
            System.out.print(i + " ");

        System.out.println();
    }
```

Notice that we were able to re-use the loop variable **i** again in the 2nd inner loop, since the first inner loop has been completed by the time that the 2nd inner loop is called. Hence, **i** is no longer defined after the first inner loop completes. Notice as well that it can be a bit tricky to get this right, especially if we make some small errors in the start/end bounds. Here are some bad results from some of these kinds of errors:

```
    for (int r=1; r<=9; r++) {
        for (int i=r; i>0; i--)
            System.out.print(i + " ");
        for (int i=2; i<10-r; i++)
            System.out.print(i + " ");
        System.out.println();
    }
```

```
1 2 3 4 5 6 7 8
2 1 2 3 4 5 6 7
3 2 1 2 3 4 5 6
4 3 2 1 2 3 4 5
5 4 3 2 1 2 3 4
6 5 4 3 2 1 2 3
7 6 5 4 3 2 1 2
8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
```

```
    for (int r=1; r<=9; r++) {
        for (int i=r; i>0; i--)
            System.out.print(i + " ");
        for (int i=2; i<9-r; i++)
            System.out.print(i + " ");
        System.out.println();
    }
```

```
1 2 3 4 5 6 7
2 1 2 3 4 5 6
3 2 1 2 3 4 5
4 3 2 1 2 3 4
5 4 3 2 1 2 3
6 5 4 3 2 1 2
7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
```

```
    for (int r=1; r<=9; r++) {
        for (int i=r; i>0; i--)
            System.out.print(i + " ");
        for (int i=1; i<r; i++)
            System.out.print(i + " ");
        System.out.println();
    }
```

```
1
2 1 1
3 2 1 1 2
4 3 2 1 1 2 3
5 4 3 2 1 1 2 3 4
6 5 4 3 2 1 1 2 3 4 5
7 6 5 4 3 2 1 1 2 3 4 5 6
8 7 6 5 4 3 2 1 1 2 3 4 5 6 7
9 8 7 6 5 4 3 2 1 1 2 3 4 5 6 7 8
```

Example:

Now let us do an example where we have to think a little more. Write a program that determines all the prime integers from 1 to 100. An integer is ***prime*** if it is only divisible by itself and 1. So, let us say that we wanted to determine if the number **25** is prime. We can try dividing **25** by all the numbers from 2 through 24 (no need to check 1 and 25 because they will divide evenly for sure).
Here is what we need to do:

```
25/2 = 12.5                                    // 25%2 = 1
25/3 = 8.33                                    // 25%3 = 1
25/4 = 6.25                                    // 25%4 = 1
25/5 = 5                                       // 25%5 = 0
```

We can stop as soon as we divide by 5 because the result is a whole number. Therefore, 25 must not be prime. We can use the modulus operator to determine if a number divides evenly into another. (x%y) will give a zero value if **y** divides evenly into **x**, otherwise it will give the integer remainder (see values on the right above).

So, to write the program, we could simply loop through all 100 numbers from 1 through 100 and try dividing each number by all the numbers before it. Can you sense this structure:

```
for (int i=1; i<=100; i++) {
      for (int j=2; j<i; j++) {
            if (i%j == 0) {
                  ...
            }
      }
}
```

In the above code, the potential prime numbers are the values of **i**. The value of **j** represents all numbers less than **i**, except **1** and **i** itself. All that remains now is to print out the prime numbers when we find them. For the inner loop, when the **if** statement evaluates to **true**, then we found that **i** is not prime because of the numbers below it divided into **i** evenly. So, in that case, we need to continue the inner loop, we can set some kind of boolean flag to **false** to indicate that **i** was not prime. Then after the inner loop, we can check the flag and print out **i** as prime if the flag was not set to **false**. Here is the code:

```
for (int i=1; i<=100; i++) {
    isPrime = true; // Assume that i is prime
    for (int j=2; j<i; j++) {
        if (i%j == 0)
            isPrime = false; // if factor found, not prime
    }
    if (isPrime)
        System.out.println(i);
}
```

There is one small aspect of this program that is a little inefficient. Many numbers will be divisible by two ... which is the first value of **j** when going through the inner loop. In those cases, the flag will be set to **false** right away but the inner loop continues checking all of the other values up to **i**-1. This is wasted time, since we already found out that the number was divisible by 2, and so it is not prime. When programming, there are many times that we want to exit a loop based on some condition occurring.

In JAVA, to indicate that we want to exit a loop early (i.e., without completing it fully), we use the **break** statement. Whenever JAVA encounters a **break** statement, it immediately stops looping and the program continues with the code that immediately follows the loop.

Here is the final code:

```
public class PrimeNumbersProgram {
    public static void main(String[] args) {
        boolean   isPrime;

        System.out.println("The prime numbers from 1 to 100 are:");
        for (int i=1; i<=100; i++) {
            isPrime = true;   // Assume that i is prime
            for (int j=2; j<i; j++) {
                if (i%j == 0) {
                    isPrime = false; // if factor found, not prime
                    break;
                }
            }
            if (isPrime)
                System.out.println(i);
        }
    }
}
```

On a similar note, there is a **continue** statement that can be used instead of **break**. It does not cause the loop to quit altogether, rather it causes JAVA to stop that particular iteration (i.e., round) of the loop and go to the next iteration of the loop. It would not be beneficial in this program. It is more useful when there is a lot of "work" to be done for each item.

For example, consider having to search 500 resumes to hire a worker. You may be looking for a particular attribute (e.g., previous administrative experience). You may go through each application and browse it quickly. If you do not see the desired attribute right away, you may want to quickly move on (i.e., ***continue***) to the next resume instead of spending a lot of time examining this candidate's resume any further. That would save you a lot of time...

```
    for (int i=0; i<yearsExperience.length; i++) {

        if (yearsExperience[i] < 2)
            continue;

        // ... code to determine if there is administrative experience ...
        if (!hasAdminExperience)
            continue;

        // ... otherwise check the resume further...
    }
```

You should use **break** and **continue** in order to speed up your loops, especially when you have a lot of data to sift through.

3.3 Conditional Iteration: While Loops

In some situations, we do not know how many times to repeat something. That is, we may need to repeat some code until a particular condition occurs. For example, consider a cashier scanning items at a checkout line in a store. The cashier repeatedly adds the items to a running total. In this scenario there is no way the cashier could know in advance how many items will be scanned ... it could be 1, 10, 50, etc… In real life, the cashier would likely press a particular button (e.g., **done**) once all items are scanned.

Whenever we have a looping situation in which we do not know how many times to loop, we should use the **while** loop ... which has this format:

```
while (stopping condition) {
    ... loop body ...
}
```

The ***stopping condition*** must be a boolean expression that evaluates to **true** or **false**. If the *stopping condition* starts off as **true**, then the loop body code runs once and then the *stopping condition* is checked again. As long as the *stopping condition* remains true, the loop keeps going. Whenever it is found to be **false**, the loop stops and the program continues with the code that follows the **while** loop.

Here is some code that uses a **while** loop to print the numbers from 0 to 99. To the right is an equivalent **for** loop:

```
int count = 0;
while (count < 100) {
      System.out.println(count++);
}
```

```
for (int count = 0; count<100; count++) {
      System.out.println(count);
}
```

In such a situation, we would always use a **for** loop instead, since we know exactly how many times to repeat the code. **while** loops are meant for situations where we are waiting for some condition to occur in order to stop looping.

Just as with **for** loops, you should be careful not to put a semi-colon **;** after the round brackets, otherwise your *loop body* will not be evaluated. Usually your code will loop forever because the *stopping condition* will likely never change to **false**:

```
while (count < 100); {
      System.out.println(count++);
}
```

```
// This code will loop forever
```

As with the **if** statements and **for** loops, the braces **{ }** are not necessary when the *loop body* contains a single JAVA expression:

```
while (count < 100)
      System.out.println(count++);
```

Some students tend to confuse the **while** loop with **if** statements and try to replace an **if** statement with a **while** loop. Do you understand the difference in the two pieces of code below ?

```
if (age > 18)
      discount = 0;
```

```
while (age > 18)
      discount = 0;
```

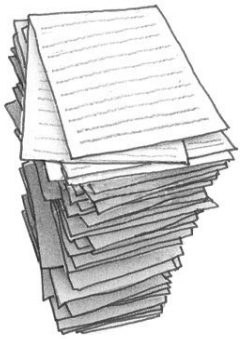
Assume that the person's age is **20**. The code on the left will set the discount to **0** and move on. The code on the right will loop forever, continually setting the discount to **0**.

Example:

Recall our example in which we found the average of a bunch of exam marks. We asked the user how many exams there were before starting and then used a **for** loop as follows:

```
int  nums, sum;

System.out.println("How many exam marks do you want to average ?");
nums = new Scanner(System.in).nextInt();
sum = 0;

// Get the numbers one at a time, and add them
for (int count=1; count<=nums; count++) {
    System.out.print("Enter exam mark " + count + ":  ");
    sum = sum + new Scanner(System.in).nextInt();

}
System.out.println("The average is " + (sum / nums));
```

Consider though, the situation in which we do not know how many exams we have (e.g., someone just plops them down on our desk). Instead of forcing the user to count them all ahead of time so that he can enter the number of exams at the start of the program, we can allow him to just start entering the marks one at a time and then look for a "***special***" value that will indicate the completion of the entries (e.g., **-1**). Then we can use a **while** loop to look for that "***special***" value.



Here is how it can be done:

```
import java.util.Scanner;

public class CalculateAverageProgram2 {
    public static void main(String[] args) {
        int  count=1, sum=0, enteredValue=0;

        while (enteredValue >= 0) {
            System.out.print("Enter exam mark " + count + " (or -1 to quit):  ");
            enteredValue = new Scanner(System.in).nextInt();
            if (enteredValue >= 0) {
                sum = sum + enteredValue;
                count++;
            }
        }
        if (count > 1)
            System.out.println("The average is " + (sum /(count-1.0)));
    }
}
```

Notice that the **enteredValue** must start off with a value that is non-negative ... in order to get into the **while** loop the first time. Any positive values entered are then added to the **sum**. In the code we also need to count how many numbers that we have entered because we will need this count to determine the average later. This is done using the **count** variable ... which starts at 1. This **count** variable is also used in the print statement so that the user knows which mark number is being entered. Notice as well, that this count is checked at the end to make sure that it is not still at 1. If, for example, the user entered -1 as the first value, then the **count** would be **1** and the final calculation would attempt to do a divide by zero ... since there are no numbers to average. Lastly, notice that we do a **(count -1.0)** instead of **(count-1)**. This will ensure that the calculation is done as a double ... so that the results will not be integers but instead the more precise floating point value.

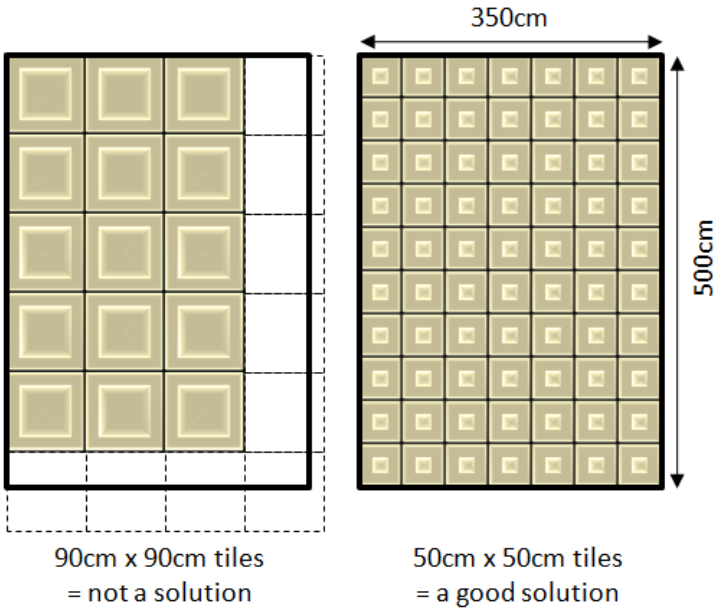
Example:

Assume that you have a nice rectangular room that measures $\mathbf{W}_{cm}$ **x** $\mathbf{H}_{cm}$. You want to place tiles down on the floor arranged in a grid pattern so that the entire floor is covered. However, you do not want to cut any tiles! Assuming that you can buy pre-cut square tiles of any size, what size of tiles should you buy ?

How do we approach the problem ?

Step 1 is to make sure that we **UNDERSTAND THE PROBLEM**.

Consider the picture to the right which has a **350_{cm} x 500_{cm}** room. In order for the tiles to fit properly, we can only have whole tiles across any row and any column. That means, if we have **R** tiles across a row, then for tiles that are **T_{cm} x T_{cm}** , then **RxT** must equal exactly **350**. In other words, **350 / T** must be a whole number, not a fraction. So the **T** must divide evenly into **350**. Similarly, **T** must divide evenly into **500** if we are to fit them properly in each column as well.



Certainly we could use 1_{cm} x 1_{cm} tiles in our example above, but that would require **175000** tiles (surely you would not want to lay those down yourself) !

In fact, here are all the possible solutions for our example:

Tile Size	Tiles Required
1_{cm} x **1_{cm}**	350 x 500 = **175000**
2_{cm} x **2_{cm}**	175 x 250 = **43750**
5_{cm} x **5_{cm}**	70 x 100 = **7000**
10_{cm} x **10_{cm}**	35 x 50 = **1750**
25_{cm} x **25_{cm}**	14x 20 = **280**
50_{cm} x **50_{cm}**	7 x 10 = **70**

Likely, the favored solution is the one that requires the least amount of tiles … which is the **50_{cm}** x **50_{cm}** tile solution. The number **50** happens to be the ***greatest common divisor*** (i.e., **GCD**) … or ***greatest common factor*** (i.e., **GCF**) of the numbers **350** and **500**. In fact, the problem that we are trying to solve requires us to find the GCD of our two numbers (i.e., of the width and the height). Here is a solution that uses a **while** loop:

```
Algorithm: SimpleGCD
        length, width:          numbers to which we need to find the GCD

1.      gcd ← minimum of length and width
2.      found ← false
3.      while (not found)  {
4.          if (gcd divides evenly into length) AND (gcd divides evenly into width) then
5.              found ← true
6.          otherwise
7.              gcd ← gcd - 1
        }
8.      print(gcd)
```

This program will start off with an attempt to see whether or not the smaller number divides evenly into the larger one. If that is true, then we have our answer and the **while** loop quits. Otherwise, the program keeps subtracting **1** from the potential **gcd** until one is found. Ultimately, this number will keep decreasing and eventually reach **1**, and that will be a common divisor to any number (although it is the ***least*** common divisor). The program assumes that neither number is zero or negative to begin with.

```
import java.util.Scanner;

public class GCDProgram {
    public static void main(String[] args) {
        int      length, width, gcd;
        boolean  found;

        System.out.print("Enter the floor length (cm): ");
        length = new Scanner(System.in).nextInt();

        System.out.print("Enter the floor width  (cm): ");
        width = new Scanner(System.in).nextInt();

        gcd = Math.min(length, width);
        found = false;
        while (!found) {
            if (((length%gcd) == 0) && ((width%gcd) == 0))
                found = true;
            else
                gcd--;
        }
        System.out.println("You need (" + gcd + "cm x " + gcd + "cm) tiles.");
    }
}
```

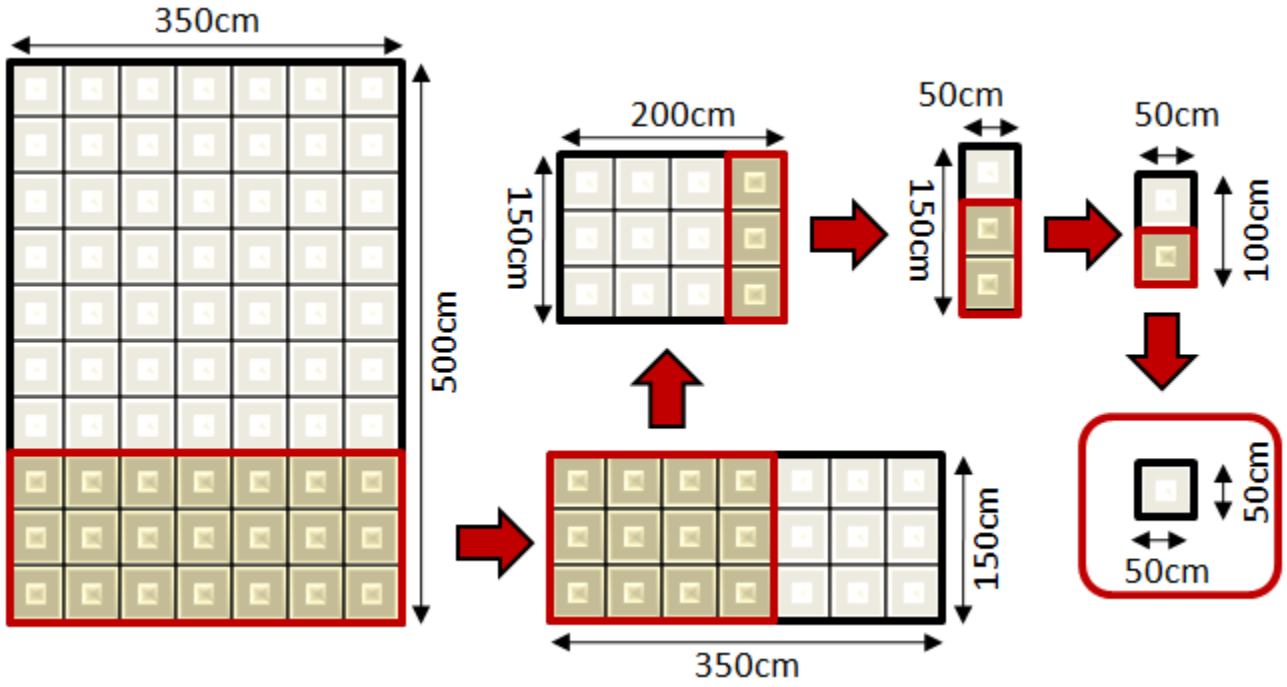
Here is some sample output:

```
Enter the floor length (cm): 350
Enter the floor width  (cm): 500
You need (50cm x 50cm) tiles.
```

The above solution will require **300** iterations of the **while** loop (i.e., **gcd** decreases from 350, 349, 348, 347, … down to **50**).

There are more efficient solutions...

For example, since the **gcd** divides both **350** and **500**, we can see that it is still possible to find the **gcd** by ignoring a large 350_{cm} x 350_{cm} portion of the floor area and concentrating on the remaining area:



As seen in the diagram above, given that we have a **W** x **L** floor area remaining, we can continually extract a **W** x **W** floor area (if **W** < **L**) or a **L** x **L** floor area (if **L** < **W**) until we end up with a remaining floor area in which **W** = **L**. In this case, **W** (or **L**, since they are equal) is the **gcd**. We can adjust our algorithm do compute the answer in this manner by repeatedly extracting the minimum of the dimensions:

```
import java.util.Scanner;

public class GCDProgram2 {
    public static void main(String[] args) {
       int      length, width;

       System.out.print("Enter the floor length (cm): ");
       length = new Scanner(System.in).nextInt();

       System.out.print("Enter the floor width  (cm): ");
       width = new Scanner(System.in).nextInt();

       while (length != width) {
           if (length > width)
               length = length - width;
           else
               width = width - length;
       }
       System.out.println("You need (" + length + "cm x " + length + "cm) tiles.");
    }
}
```

This algorithm produces a better solution … which requires only **5** iterations of the **while** loop!

Homework:

In fact, it can be improved even further (i.e., only **3** iterations of the **while** loop) by using the modulus operator which takes multiples of the lower dimension away in one step. See if you can change the program to do this (put a print statement in the **while** loop to see how many times it gets evaluated):