Chapter 4
Loops

We've been using loops since our third program, where we did simple animation with an endless loop as shown below.

```java
while(true)
{
    label1.move(1, 1);
    pause(100);
} //move the label
```

Java supports three looping statements. We will look at two of them, the `while` and `for` statements. The `do-while` statement will be left to the Quick Reference appendix.

### 4.1 While Loops

The `while` loop is typically used when the number of repetitions is not known before execution of the loop begins.

#### 4.1.1 while(true) break Loops

An endless loop is very simple but not sufficient for any sophisticated application. Games and other applications require that action eventually end or at least change. We might want a loop that does something like:

- while not all of the data has been read from the file continuing reading data
- while the game is not over continuing playing

The `while(true)` `break` loop works well for this sort of purpose.

Let's look at this example, taken from the `MoveGreenBall2` program. It is typical of this form of the `while` loop. Note that we are assuming the diameter of the ball is 50 pixels.

```java
while(true)
{
    pause(20);
    ball.move(1, 1);
    if (ball.getY() + 50 >= APPLICATION_HEIGHT)
    {
```
We've used this form of the while loop several times already. Its general form is

```
while(true) {
    ...statements...
    if (boolean expression) {
        break;
    }
    ...statements...
} //comment
```

Some additional details are worth noting:

- there may be more than one `break` statement, each representing a reason to end the loop.
- there should be at least one `break` statement. Otherwise you have an endless loop. This isn't a Java requirement—we've already written programs with endless loops. But normally we don't want a program to run forever, so at least one `break` statement is needed.

### 4.1.2 While (boolean expression) Loops

The `while(true)` `break` loop has one notable advantage which is also possibly a disadvantage. The `if(boolean expression)`- `break` can be anywhere within the loop, and so execution can also jump out of the loop anywhere. If there are many such jumps it can be difficult to understand the code. The `while(boolean expression)` loop is often considered a more understandable way to write the code.

Here's the above example, written in the `while(boolean expression)` form.

```
while(ball.getY() + 50 < APPLICATION_HEIGHT) {
    pause(20);
    ball.move(1, 1);
} //move the ball
```

In this simple example all that is necessary is to transform the logic of

*If (boolean expression is true) break out of the loop*
to

while (a slightly different boolean expression is true) continue the loop

logic. In the example above, the

while(true)
{
    ...statements...
    if(ball.getY() + 50 >= APPLICATION_HEIGHT)
    {
        break;
    }
} //move the ball

becomes

while(ball.getY() + 50 < APPLICATION_HEIGHT)
{
    ...statements...
} //move the ball

Note that the boolean expression changes to the logical opposite. This is typical of the change needed when a while(true) break loop is rewritten as a while(boolean expression) loop, but it isn’t always that simple. Such boolean expressions need to be written very carefully, particularly if there are multiple conditions that cause a break out of the loop.

Here’s a simple example of a while(boolean expression) loop that inputs an integer representing an age, repeating until an age of at least 21 is entered.

```java
int x;
x = readInt("Enter age: ");
while(x < 21)
{
    x = readInt("Enter age of *21 or over* please: ");
} //get age
println("Okay, serve that man a beer!");
```

Note that x must have a value before the while begins. If the statement

```java
x = readInt("Enter age: ");
```

is removed then x would not have a value the first time the x < 21 expression executes. The Java compiler is pretty smart about identifying this problem and displays error message
Giving the loop control variable, \( x \) in this case, an initial value that guarantees the body of the loop will execute at least once is often called **priming the loop**.

By the way, whenever you log into a computer network or email account you use a loop just like this—think about entering your username and password and you'll see the connection.

### 4.1.3 More to Know About While Loops

The **while** loop is a **pre-test** loop—the **boolean** expression is tested **before** the body of the loop. The body of the loop will not execute if the **boolean** expression is initially **false**. Thus the **while** loop is most naturally used when it is possible that the body of the loop may never need to execute. In the age reading code above, the body of the **while** loop will never be executed if the first age entered is at least 21.

The body of the **while** loop usually contains multiple statements and so we have braces surrounding the body.

```java
while(...)
{
    statement1
    statement2
    ...more statements...
} //comment
```

However, if the body consists only one statement it is legal (though poor practice) to leave out the braces. We would have

```java
while(...) statement;
```

and unfortunately that leads us to a **very** easy to make error. Examine the code below.

```java
while(boolean expression);
{
    statement1
    statement2
    ...more statements...
} //comment
```

The problem lies in semi-colon at the end of the very first line.

```java
while(boolean expression);  the ; is the problem
```
That semi-colon is a legitimate, though vacuous, statement. The loop body is only the ;. If the boolean expression is true when first evaluated it will always be true (because the ; statement doesn't change anything) and the loop (containing only the semi-colon) will run forever. What we think is the body of the while loop, the block

```java
{  
    statement1  
    statement2  
    ...more statements...
} //comment
```

is actually just another set of statements that will never execute because the loop never ends. And it is very hard to see this error in your source code. Unfortunately the compiler doesn’t give us a warning about this.

Many professionals consider writing endless while loops and then making them non-endless with a break to be bad programming practice, particularly if the loop body is long and there is more than one break statement. They believe that having more than one exit from the loop is confusing and prefer the while(boolean expression) form.

### 4.2 For Loops

Generally while loops are used when the programmer doesn’t know how many times the loop will execute. But very often a loop will need to execute a specific number of times or perhaps create a set of values via a formula. In cases like this a for loop is generally preferred.

#### 4.2.1 Examples

Example: display the digits from 1 to 3

```java
for(int i = 1; i <= 3; i++)
{
    println(i);
} //count up 1 to 3
```

Consider the first line of the for loop (called the loop header) and its three parts.

- **int i = 1;** This is the initialization statement, which is used to initialize the loop control variable, i in this case. The loop control variable is typically used to control the number of executions of the body of the loop. The initialization statement runs only once, when execution of the for loop begins.
- **i <= 3;** This is the boolean expression. It is first evaluated before the body of the loop is executed, and then after every repetition of the loop. If the boolean expression evaluates to true, the body is executed. If it is false, the body is skipped and the for loop is complete.
• $i++$; The **increment statement**, which is usually used to change the value of the loop control variable so that eventually the boolean expression becomes **false**, thus ending the loop. The increment statement is executed immediately after the body of the loop is executed.

The body is executed each the **boolean** expression evaluates to **true**. When all of the statements in the body have executed, execution jumps to the increment expression and then to the **boolean** expression. Below is a trace of the execution of this **for** loop.

<table>
<thead>
<tr>
<th></th>
<th>$i$</th>
<th>$i \leq 3$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop header</td>
<td>1</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>first pass through loop</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>loop header</td>
<td>2</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>second pass through loop</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>loop header</td>
<td>3</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>third pass through loop</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>loop header</td>
<td>4</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

Example: display the integers from 20 down to 6, by twos (20, 18, ..., 6)

```java
for(int m = 20; m >= 6; m = m – 2) {
    println(i);
} //count down 20 to 6 by 2s
```

In this loop the increment is actually a decrement statement, or perhaps we might view it as an increment by -2.

Example: generate the values from 3.25 to 6, by increments of .05.

Here's our first solution, but it **DOES NOT WORK**.

```java
for(double value = 3.25; value != 6; value = value + .05) {
    ...statements...
} //count up 3.25 to 6
```

This doesn't work because arithmetic with **doubles** is not necessarily completely accurate, and so we can't check for exactly equal to because it is quite likely that `value` is never exactly equal to 6. This could easily result in an endless loop.

To avoid an endless loop, we rewrite the **boolean** expression so that the loop executes as long as value is not very close to 6, using Math.abs(value – 6) > **EPS** for the test.
final double EPS = .001;
for(double value = 3.25; Math.abs(value - 6) > EPS; value = value + .05)
{
    ...statements...
} //count up 3.25 to 6

Example: display the first 5 Fibonacci numbers. The Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21, ... Except for the first two numbers, which seed the sequence, each Fibonacci number is the sum of the previous two Fibonacci numbers: $x_n = x_{n-1} + x_{n-2}$. Here's a complete program.

Fibonacci

```java
//Fibonacci.java
import acm.program.*;
public class Fibonacci extends ConsoleProgram
{
    public void run() {
        //first 2 Fibonaccis
        int fib1 = 1, fib2 = 1, fib3;
        println(fib1);
        println(fib2);

        //remaining 3 Fibonaccis
        for(int i = 1; i <= 3; i++) {
            fib3 = fib1 + fib2;
            println(fib3);
            fib1 = fib2;
            fib2 = fib3;
        } //count up 1 to 3
    } //run
} //Fibonacci
```

Here's a **trace table** for this program, this time showing the executed statements. The arrows indicate the shuffling described above.

<table>
<thead>
<tr>
<th>executed statements</th>
<th>fib1</th>
<th>fib2</th>
<th>fib3</th>
<th>i</th>
<th>boolean</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib1=1, fib2=1</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>println(fib1), println(fib2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>i=0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>i&lt;5</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>fib3=fib1+fib2</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>println(fib3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>fib1=fib2</td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fib2=fib3</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
It should be noted a trace table is usually made much more compact by leaving out the executed statements. Here’s the same trace table as above, done in the common fashion.

<table>
<thead>
<tr>
<th>fib1</th>
<th>fib2</th>
<th>fib3</th>
<th>i</th>
<th>boolean</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>true</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>8</td>
<td>4</td>
<td>false</td>
<td>3</td>
</tr>
</tbody>
</table>

The important thing to be learned here is the concept of 'shuffling values', that is moving the most recent values (fib3 for example) to be stored in other variables. This shuffling technique might be used to stored the last three temperature values obtained from a sensor on the leading edge of the space shuttle, tracking the last three enemies killed, etc. It's a useful technique.

### 4.2.2 More to Know About for Loops

What we’ve learned about for loops will probably cover most of your programming needs, but here are some extra facts worth knowing.

- an identifier declared inside the for loop header is local to the loop, meaning its scope is restricted to the for loop itself.
- the initialization, boolean expression and increment statements within the for loop are all optional. You can leave any of these out, perhaps providing the needed logic before the for statement or in the body of the loop instead.

For example, consider this code segment that prints all the numbers in the Fibonacci sequence that are less than 100. It leaves out the initialization statement, instead...
putting initialization before the loop header. And the increment statements that assign new values to fib1, fib2 and fib3 are in the body. Note that the semi-colons are required even though the associated statements are missing.

```java
int fib1 = 1, fib2 = 1, fib3;
println(fib1);
println(fib2);
fib3 = fib1 + fib2;

for (; fib3 <= 100;)
{
    fib3 = fib1 + fib2;
    if (fib3 <= 100)
    {
        println(fib3);
        fib1 = fib2;
        fib2 = fib3;
    }
}
```

- **for** loops can also contain **break** statements, allowing you to construct a more flexible loop, though multiple exit points can be confusing.

### 4.3 More About Loops

Loops will power our games and of course virtually all other programs. We consider some additional points.

#### 4.3.1 Fundamental Algorithms

There are a few fundamental techniques that show up over and over.

**The Accumulation Algorithm**

Goal: input and sum values until a specific maximum is exceeded.

```java
//SumNumbers.java
import acm.program.*;

public class SumNumbers extends ConsoleProgram
{
    public void run()
    {
        final int MAX_VALUE = 100;
        int inputVal, sum = 0;

        while(true)
        {
            // must initialize the sum to 0; we will add values to it as the loop executes
        }
    }
}
```
And here’s a sample trace table.

<table>
<thead>
<tr>
<th>inputVal</th>
<th>sum</th>
<th>sum &gt;= MAX_VALUE</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35 is entered</td>
<td>35</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>-10 is entered</td>
<td>-10</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>50 is entered</td>
<td>50</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>82 is entered</td>
<td>82</td>
<td>true</td>
<td>157</td>
</tr>
</tbody>
</table>

Using a computer program to do this sort of calculation is of course silly. But the accumulation algorithm can also be used for:

- tracking fuel added to a battle tank
- summing temperatures so that an average can be calculated
- totaling points scored in football game

It’s simple but very useful.

**The Counting Algorithm**

Counting is a fundamental operation and in this example it is combined with a *sentinel value*, which is a value used to trigger an action in the program. Sometimes a sentinel is called a *flag*.

Goal: input values until a sentinel value is input, then calculate the average of the inputs. Here -1 is used as the sentinel to stop data input.

```java
final int SENTINEL = -1;
int sum = 0, count = 0, inputVal;

while (true) {
    inputVal = readInt("Enter a positive value (-1 to quit): ");
    if (inputVal == SENTINEL)
    { take sum, add inputVal to it and assign the new value back into sum
```
break;
}
    sum = sum + inputVal;
    count++;
} //read and sum values

double average = (double) sum/count;

And the trace table.

<table>
<thead>
<tr>
<th>initialization</th>
<th>inputValue == SENTINEL</th>
<th>sum</th>
<th>count</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 is entered</td>
<td>false</td>
<td>8</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6 is entered</td>
<td>false</td>
<td>14</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5 is entered</td>
<td>false</td>
<td>19</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-1 is entered</td>
<td>true</td>
<td></td>
<td></td>
<td>6.333</td>
</tr>
</tbody>
</table>

The accumulation and counting algorithms and sentinels can also be used with a `while(boolean expression)` loop as shown below.

```java
final int SENTINEL = -1;
int sum = 0, count = 0, inputVal = 0;
while (inputVal != SENTINEL)
{
    inputVal = readInt("Enter a positive value (-1 to quit): ");
    if (inputVal != SENTINEL)
    {
        sum = sum + inputVal;
        count++;
    }
} //read values and sum them

double average = (double) sum/count;
```

And of course these techniques can be used with `for` loops also.

### 4.3.2 Nested Loops

It is sometimes very useful to put one loop inside another. Here's a very simple example that outputs the multiplication table shown below. Note that only the results of the multiplications are printed, not the row and column headings.
for(int row = 1; row < 4; row++)
{
    for (int col = 1; col < 6; col++)
    {
        print('row*col + "   "');
    } //col values
    println();
} //row values

If we look at this abstractly we see

run the outer loop 3 times
{
    for each outer loop run the inner loop 5 times
}

This means that the inner loop actually runs 15 times.

Producing the row and column headings requires the modifications as shown below.

println("1 2 3 4 5");
println("-------------------");
println("-------------------");
for(int row = 1; row < 4; row++)
{
    print("  + row + " || ");
    for (int col = 1; col < 6; col++)
    {
        print('row*col + "   "');
    } //col values
    println();
} //row values

Note that this really doesn't produce a beautifully formatted table. In particular the formatting breaks down as soon as our multiplication results in two digits numbers. But it's the best we can do with the tools we've got right now.
4.3.3 Pausing Execution in a Loop

Looping, particularly unintended infinite loops, has the potential to put information in the window faster than you can use it. It is very useful to be able to pause a loop temporarily.

Pausing in a console program
Below is an example of an infinite loop, intended to print the numbers 1, 3, ..., and stop at 111. But it doesn’t work and the data runs by so fast it is hard to tell what values are actually being printed out.

```java
for(int i = 1; i != 112; i = i + 2)
{
    println(i);
}
```

Here's the same code segment with a modification that halts it every 10 loops by requiring user input. This lets the user examine the values of i as the program runs. We don't actually do anything with the new variable x. It's just there so we can halt the program by requiring keyboard input, which allows us to observe what's going on.

```java
int x;
int counter = 1;
for(int i = 1; i != 112; i = i + 2)
{
    println(i);
    counter++;
    if (counter % 10 == 0)
    {
        x = readInt();
    }
}
```

Pausing in a graphics program – just use the familiar `waitForClick()` or `pause()` statements as required. You can also use data input as shown above. The input will occur in the command window that appears when you run a graphics program.

Objective Questions

1. Evaluate the result of execution of each expression
   a. int x = 10;
      while(true)
      {
          println(x-1);
          if (x == 0) exit;
      }
   b. int x = 10;
      while(true)
{     println(x+1);  }

o  int x = 10
    while(x <= 20)
    {
        x++;
        println(x);
    }

o  int x = 10;
    while(x > 0);
    {
        print(x);
        x--;
    }

o  for(int a = 1; a <= 10; a++)
    {     println(2*a);  }

o  for(double r = 1.10; r != 5; r = r + .01)
    {     println(r);  }

o  for(int a = 100; a <= 99; a++)
    {     println(a);  }

o  int f = 0;
    int s = 0;
    for(double t = 0; t < 10; t = t + 2)
    {
        s = f;   f = t;
        println(f + "   " + s);
    }

o  int sum = 0, count = 0;
    for()
    {
        int x = ReadInt("Value? ");
        sum = sum - x;
        count = count--;
        if (count == -5) break;
    }
    println(sum/count);

o  for(int i=0; i<4; i++)
    {
        for(int j=i; j < 5; j++)
        {
            println(i*j);
        }
    }

---

**Console Problem Set**
1. **NumbersWhileLoops** write one program that uses several loops, once for each task. Use two `println()` statements to provide separation between the output for each task.
   - Display the numbers 20, 19, 18, ..., 2, 1 using a while loop.
   - Display the fractions 2/3, 3/4, 4/5, ..., 9/10 using a while loop
   - Using a while loop, input values from the keyboard. Sum those values that are less than 100, until a 0 is input. Ignore values that are non less than 100. Display the sum
   - Using a while loop, input values from the keyboard, until a 0 has been input. Ignoring the 0, output the average of the input

2. **NumbersForLoops** write one program that uses several loops, once for each task. Use two `println()` statements to provide separation between the output for each task.
   - display the integers 10, 9, 8, ... 1
   - display the integers 10, 9, 8, ... 1
   - display the real numbers 1/2, 1, 1 1/2, 2, ..., 5 in the format .5, 1, 1.5, and so on. Note that you do not as yet know how to set the precision, so what you actually get might look like .500000, 1.0000, 1.5000, etc.
   - display the rational numbers 1/2, 2/2, 3/2, 4/2, ..., 10/2
   - display the numbers 2, -4, 8, -16, ..., -64
   - sum the numbers 1 though 100. Display the sum.
   - sum the numbers 5, 8, 11, ...,20. Display the sum
   - display the first 6 powers of 3: 1, 3, 9, 27, 81, 243

3. **SumTill0** Write a program input integers from the keyboard until a 0 is input. Display the average, count and sum of the integers. Do not include the 0 when computing the statistics.

4. **Tables** Write one program that solves the following tasks. Use nested loops for each.
   - display an addition table
     
     | 1 | 2 | 3 | 4 | 5 |
     |-------------------|
     | 1 | 2 | 3 | 4 | 5 | 6 |
     | 2 | 3 | 4 | 5 | 6 | 7 |
     | 3 | 4 | 5 | 6 | 7 | 8 |
     | 4 | 5 | 6 | 7 | 8 | 9 |
     | 5 | 6 | 7 | 8 | 9 |10 |
   - display this table of descending rows
     
     | 4 | 3 | 2 | 1 |
     | 4 | 3 | 2 |
     | 4 | 3 |
     | 4 |

**Graphics Problem Set**

1. **ExplodingUFO1** – Recall the UFO that you flew across the screen. Modify that program so that when it hits a "mine" in the sky the ufo crashes to the ground, falling down and to the right.
2. **ExplodingUFO2** When the mine is hit, a green alien (just a green circle) pops out and falls to the ground.

3. **Dribble1** Bounce a ball up and down. The ball bounces forever, always reaching the same height.

4. **Dribble2** Modify Dribble1 so the bounce height decreases and eventually the bouncing ball settles to the ground, just like a real ball would do.

5. **Ornithopter1** An ornithopter is an airplane that flies by "flapping" its wings, rather than through turning a propellor. Write a program to simulate a nose-on view of an ornithopter with flapping wings in 3 stages.

6. **Ornithopter2** Modify the ornithopter so that the beat of the wings slows, and the ornithopter settles gently to the bottom of the window.

**objective questions**

1. TRUE/FALSE A **for** loop cannot be an infinite loop.
2. TRUE/FALSE A **while** loop can be written so that its body never executes.
3. What is the output of this loop?
   ```java
   for(int i = 0; i < 10; i++)
   {
       if (i % 2 == 0)
       {
           println(i);
       }
   }
   ```

4. **short answer**

1. Write a **for** loop that runs, inputing integer values. The loop should sum only odd integer values, and output the sum as soon as it exceeds 50.
2. Write a **while(true)-break** loop that solves #1.
3. Write a **while(booleanExpression)** loop that solves #1.
4.