REPETITION STATEMENT

One of the great virtues of computers is that they will repeat mindless tasks without complaint. To facilitate such repetition, Java includes statements types called loops, which allow a program to specify some sequence of instructions that should be repeated.

5.1. The `while` loop

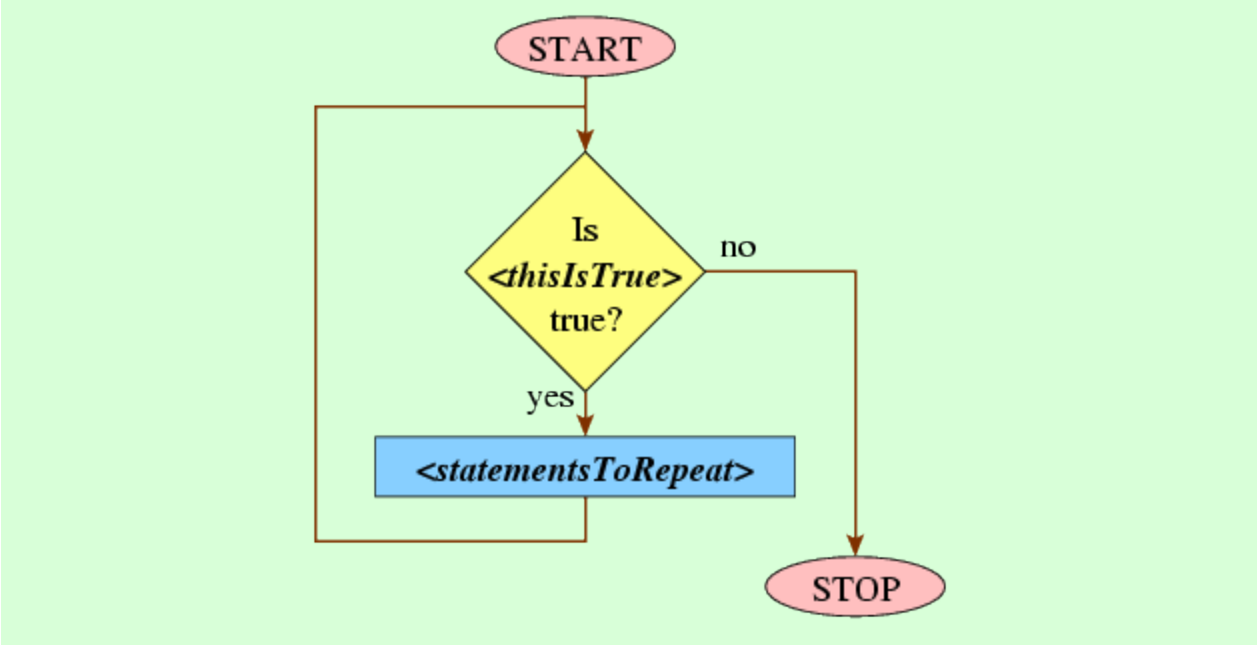
Java has a few categories of loops, and the most fundamental of these is the **`while`** loop.
The **`while`** loop enables you to specify that some sequence of statements should be repeated as long as some condition is true (i.e., *while* the condition holds) . The template for a **`while`** loop looks like the following.

```
while(<thisIsTrue>) {
    <statementsToRepeat>
}
```

You include the word **`while`**, with a true/false expression in parentheses (the parentheses are required), followed by a set of braces including the lines you want to repeat. The parenthesized expression is called the condition of the loop. The part between the braces is the loop's body.

When the computer reaches a **`while`** loop, it tests the condition inside the parentheses to determine whether it is true. If so, it executes the body (between the braces) and then checks the condition again. It repeatedly checks the condition and executes the body, until finally it finishes the body and the condition turns out to be false. Each execution of the loop's body is called an iteration. Once the condition turns out to be false, the computer continues to the first statement following the body (after the closing brace). (If the condition never held in the first place, the computer skips past the body immediately, with no iterations.) Figure 5.1 illustrates this process.

Figure 5.1: A flowchart for the **`while`** statement.



To see an example of a `while` loop, let's look back to our `DrawSquare` program (Figure 4.1). That program was a bit repetitive: We repeated the same two lines four times, once for each side. We could replace lines 10 to 16 with a `while` loop, with each iteration of the loop drawing a single side of the square. Figure 5.2 contains such a program.

Figure 5.2: The `DrawSquareLoop` program.

```
1  import turtles.*;
2
3  public class DrawSquareLoop extends TurtleProgram {
4      public void run() {
5          double sideLength;
6          sideLength = this.readDouble();
7
8          Turtle boxTurtle;
9          boxTurtle = new Turtle(100 - sideLength / 2, 100 -
sideLength / 2);
10         int drawn; // counts how many sides are complete
11         drawn = 0; // so far none are complete
12         while(drawn < 4) {
13             boxTurtle.forward(sideLength);
14             drawn = drawn + 1; // we've completed one more side
15             boxTurtle.right(90);
16         }
```

```
17          boxTurtle.hide();
18      }
19  }
```

The computer would execute the `DrawSquareLoop` program as follows.

1. **Lines 5–9:** These lines are the same as before: They determine the size of the square and initialize the turtle.
2. **Lines 10–11:** The computer creates a variable `drawn` to count how many sides of the square `boxTurtle` has drawn. Because `boxTurtle` hasn't drawn any sides yet, the program initializes `drawn` to 0.
3. **Line 12:** The computer checks whether the condition `drawn < 4` holds. It is, so the computer proceeds to execute each of the statements in the braces.
4. **Line 13:** The computer tells `boxTurtle` to move forward `sideLength` pixels.
5. **Line 14:** The computer now reassigns `drawn` to a new value: In particular, it is assigned to the value of `drawn + 1`. Since `drawn` is currently assigned to the value 0, the value of `drawn + 1` is 1, and so 1 is the value now assigned to the `drawn` name.

   Notice the peculiarity of this line. If this were an algebra class, you'd look at this as an equation, cancel `drawn` from both sides, and end up with 0 = 1. But in Java, the equal sign represents *assignment*, not *equality* as in algebra. So the computer doesn't think about the relationship between the two sides: It simply evaluates the right-hand side first, and then it assigns the left-hand variable to reference whatever result falls out.

6. **Line 15:** The computer tells `boxTurtle` to turn 90° clockwise, so that `boxTurtle` now faces south.
7. **Line 16:** Having reached the end of the **`while`** loop, the computer proceeds to the top of the loop again to check the condition again, in line 12.
8. **Lines 12–16:** The computer checks whether the condition `drawn < 4` still holds. Since `drawn` is 1, it is does. Hence we proceed: The computer tells `boxTurtle` to move forward, reassigns 2 to `drawn`, and tells `boxTurtle` to turn facing west.
9. **Lines 12–16:** The condition `drawn < 4` still holds. The computer tells `boxTurtle` to move forward, reassigns 3 to `drawn`, and tells `boxTurtle` to turn facing north.
10. **Lines 12–16:** The condition `drawn < 4` still holds. The computer tells `boxTurtle` to move forward, reassigns 4 to `drawn`, and tells `boxTurtle` to turn facing east.
11. **Line 12:** The condition `drawn < 4` is finally false. So the computer is done with the **`while`** loop and now proceeds to the first statement following the loop's body, line 17.
12. **Line 17:** The computer tells `boxTurtle` to hide itself.

One subtlety of **`while`** loops is that the computer checks the condition only when it reaches the end of the loop's body. In the last iteration of the **`while`** loop of this example, after`drawn` is assigned the

value 4 in line 14, the turtle nonetheless turns to face east in line 15, because the computer must complete the body of the `while` loop before checking the condition again.

Notice that a `while` loop involves no semicolons! Beginners are often tempted to add one anyway, after the parentheses.

```
while(drawn < 4); {   // Wrong! The semicolon messes up this code.
    drawn = drawn + 1;
}
```

Irritatingly, the Java compiler will accept this with a very different meaning than intended: It will understand the body of the loop to be an empty statement terminated by the semicolon before the opening brace, and the part in braces will be something to be done only once the loop completes. But the loop will never complete, because the computer will end up repeatedly checking whether `drawn < 4` and then doing nothing.

Source : http://www.toves.org/books/java/ch05-while/index.html