Repetition using Java – Loops

Repetition is a programming concept that allows the computer to run one or more commands multiple times, without having to write them out each time. Instead, the commands are placed inside a *looping structure*, where they will be executed over and over, until some *condition* is met (i.e., becomes true).

Consider an example from cooking, where we have to stir a gravy until all of the lumps are gone. Using pseudocode, the instructions might look like

```
while (there are lumps in the gravy)
  give the gravy a stir
end while
```

The idea is to keep checking for lumps. If there are still lumps, stir the gravy. If the lumps are gone, the loop ends.

The While Loop – Decision at the Beginning

The execution of a *while loop* is similar to the execution of an *if statement*. A condition is checked on the first line (`true` or `false`). If the condition is `true`, we enter the loop and execute any commands found within the *body of the loop*. If the condition is `false`, we exit the loop without executing any commands.

```
while (<boolean expression>)
{
  <statements>
}
```

Consider the following example, which asks the user for integer values, and then outputs the square of the value. It continues to do this until the user enters zero, which causes the loop to exit.

```
class PrintSquares
{
  public static void main (String[] args)
  {
    System.out.print("Give an integer (zero to stop): ");
    int value = In.getInt();
    while (value != 0)
    {
      System.out.println(value + "   " + value*value);
      System.out.print("Next integer (zero to stop): ");
      value = In.getInt();
    }
  }
}
```

Some important notes about this program:

1. It is necessary to **initialize** the variable **`value`** before testing the condition of the while loop. Otherwise, `value` would have been undefined when the loop was first checked, and an error would occur.
2. There must be some way within the loop that will eventually cause the loop to exit. In this case, we keep asking the user for a new `value`, and we will exit when `value` is zero. We say zero acts as the **sentinel** value for this loop.
3. If there is no *sentinel* value, or if there is no way to set the sentinel value, the loop will never end. This results in an **infinite loop**, which must be terminated from the operating system of your computer.
4. If the first value entered in the program were zero (the *sentinel* value), the contents of the loop would never execute.

The Do-While Loop – Decision at the End

In the do-while loop, sometimes simply called a do loop, the condition is checked at the end of the loop. This means that the statements inside the do loop will always be executed at least once (whereas a while loop, if it starts with a false condition, might never execute).

```
do
{
  <statements>
}
while (<boolean expression>);
```

Note: A semi-colon is required at the of the while statement in a do-while loop.

Like a while loop, we need a **sentinel** value, and this value must be changed inside the loop or we risk an *infinite loop*.

The For Loop – A Counted Loop

A counted loop is used when you know, or could calculate, the exact number of times that the loop needs to execute. Both a while loop and a do loop can also be used as a counted loop, but since counted loops are so common and useful, they get their own specialized structure, which improves efficiency and clarity in the program.

Suppose we return to our first example of stirring a gravy. Instead of trying to get rid of the lumps, suppose the recipe called for 100 stirs. To implement this using a while loop, we might code the following:

```
int count = 0; // initialize variable before the loop
while (count < 100)
{
  sauce.stir();
  count = count + 1;
}
```

As a for loop, the same code would look like

```
for ( count = 0 ; count < 100 ; count++ )
{
  sauce.stir();
}
```

All of the same steps occur in the for loop. The counting variables is declared and initialized (count = 0), the boolean condition is set (count < 0), and the counter is *incremented* for each iteration of the loop. The main differences are (a) how compact the code is, and (b) that the structures makes it obvious that it is a counting loop.

Another change in the code worth noting is *how* we incremented the counter. The "++" at the end of count++ is called the **increment operator**, and it increases the value of an integer variable by one. There is also a **decrement operator**, which reduces the value of an integer variable by one.

`count++` is equivalent to `count = count + 1`

`count--` is equivalent to `count = count – 1`