CHAPTER: 03

Iteration

Objectives:

By the end of this chapter you should be able to:

- *explain the term **iteration**;*
- *repeat a section of code with a `for` loop;*
- *repeat a section of code with a `while` loop;*
- *repeat a section of code with a `do...while` loop;*
- *select the most appropriate loop for a particular task;*
- *explain the term **input validation** and write simple validation routines.*

3.1 Introduction

So far we have considered sequence and selection as forms of program control. One of the advantages of using computers rather than humans to carry out tasks is that they can repeat those tasks over and over again without ever getting tired. With a computer we do not have to worry about mistakes creeping in because of fatigue, whereas humans would need a break to stop them becoming sloppy or careless when carrying out repetitive tasks over a long period of time. Neither sequence nor selection allows us to carry out this kind of control in our programs.

Iteration is the form of program control that allows us to instruct the computer to carry out a task over and over again by repeating a section of code. For this reason this form of control is often also referred to as **repetition**. The programming structure that is used to control this repetition is often called a **loop**. There are three types of loops in Java:

- `for` loop;
- `while` loop;
- `do...while` loop.

3.2 The 'for' loop

Consider a program that needs to display a square of stars (five by five) on the screen as follows:

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

This could be achieved with five output statements executed in sequence:

```
System.out.println("*****");
System.out.println("*****");
System.out.println("*****");
System.out.println("*****");
System.out.println("*****");
```

While this would work, all the program is really doing is executing the following instruction five times.

```
System.out.println("*****");
```

Writing out the same line many times is somewhat wasteful of our precious time as programmers. Imagine what would happen if we wanted a square 40 by 40!

Rather than write out this instruction five times we would prefer to write it out once and get the program to *repeat that same line* five times.

If we wish to repeat a section of code a fixed number of times (five in the example above) we would use Java's **`for`** loop.

The **`for`** loop is usually used in conjunction with a **counter**. A counter is just another variable (usually integer) that has to be created. We use it to keep track of how many times we have been through the loop so far. The loop works as follows:

1 the counter is set to some initial value (usually 0 or 1);

2 the counter is tested before each repetition of the loop. When the test returns a **`boolean`** value of **`true`** the loop repeats, when it returns a **`boolean`** value of **`false`** the loop ends;

3 after each repetition of the loop the value of the counter is changed (usually by adding 1 to it) so that eventually the test will stop the loop.

These three tasks are assembled as follows to construct the **`for`** loop:

```
for( /* start counter */ ; /* test counter */ ; /* change counter */)
{
      // instruction(s) to be repeated go here
}
```

We can construct a loop to display a square of stars consisting of five rows as follows:

```
for(int i = 1; i <= 5; i++)
{
       System.out.println("*****");
}
```

As with **if** statements, the braces can be omitted when only a single instruction is required in the loop – but for clarity we will always use braces with our loops. Notice that the loop counter 'i' is declared as well as initialized in the header of the loop. The counter is just like any other variable, and so can take any name – often though, simple names like 'i' and 'j' are chosen. Although it is possible to declare the counter prior to the loop, declaring it within the header restricts the use of this variable to the loop itself. This is often preferable.

The loop works in the following way. First the loop counter is set:

```
for(int i = 1; i <= 5; i++) // counter initialised to 1
{
       System.out.println("*****");
}
```

The counter is then tested to see if it is less than or equal to 5:

```
for(int i = 1; i <= 5; i++) // counter tested
{
       System.out.println("*****");
}
```

Since the counter was set to 1, this test is **true** and the body of the loop is entered. All the instructions within the braces of the loop are executed. In this case there is only one instruction to execute:

```
for(int i = 1; i <= 5; i++)
{
       System.out.println("*****"); // this line is executed
}
```

This line prints a row of stars on the screen. Once the instructions inside the braces are complete the loop returns to the beginning, where the counter is changed. In this case the counter is incremented:

```
for(int i = 1; i <= 5; i++) // counter is changed
{
       System.out.println("*****");
}
```

Notice that we have used the increment operator here. As you know, this is just a shorthand for

```
i = i+1
```

This assignment can be used in place of the increment operator, if so desired. After the increment, the counter now has the value of 2. The test is checked to see if the loop should repeat:

```
for(int i = 1; i <= 5; i++) // counter tested
{
       System.out.println("*****");
}
```

This test is still **true** as the counter is still not greater than 5. Since the test is **true** the body of the loop is entered again and another row of stars printed. This process of checking the test, entering the loop and changing the counter repeats until five rows of stars have been printed. At this point the counter is incremented as usual:

```
for(int i = 1; i <= 5; i++) // counter eventually equals 6
{
       System.out.println("*****");
}
```

Now when the test is checked it is **false** as the counter is greater than 5:

```
for(int i = 1; i <= 5; i++) // now test is false
{
       System.out.println("*****");
}
```

When the test of the **for** loop is **false** the loop stops. The instructions in the loop are skipped and the program continues with any instructions after the loop.

As you can see it is very straightforward to repeat a section of code many times. If, for example, we needed to print the row of stars 100 times instead of just five, how would you change the **for** loop? Well, all you would need to do would be to change the test as follows:

```
for(int i = 1; i <= 100; i++) // this loop repeats 100 times
{
       System.out.println("*****");
}
```

This is a very common way of using a **for** loop. Start the counter at 1 and add 1 to the counter each time the loop repeats. However, you may start your counter at any value and change the counter in any way you choose when constructing your **for** loops.

As an example, look at program 3.1 that prints out a countdown of the numbers from 10 down to 1.

Program 3.1

```
public class Countdown
{
  public static void main(String[] args)
  {
    System.out.println("***Numbers from 10 to 1***");
    for (int i=10; i>=1; i--)  // counter moving from 10 down to 1
    {
      System.out.println(i);
    }
  }
}
```

Here the counter starts at 10 and is reduced by 1 each time. Note the use of the loop counter inside the loop:

```
System.out.println(i); // counter 'i' used here
```

When you do this, however, be careful not to inadvertently *change* the loop counter within the loop body as this can throw the test of your **for** loop off track!

Finally, before moving on to look at another form of loop in Java, a reminder that the loop body can contain any number of instructions, including **if** statements, **switch** statements, or even another loop! In other words, you may nest loops just as you nested **if**s. As an example of this, consider again the **for** loop we constructed to display a square of stars.

```
for(int i = 1; i <= 5; i++)
{
      System.out.println("*****");
}
```

The body of this loop has an instruction that displays five stars on the screen in a row. We could, if we had wanted, have displayed only a *single* star on the screen as follows:

```
System.out.print("*");
```

Now, to get the program to display five of these stars we could put this instruction within another loop. We will need to use another loop counter for this new loop – let's call this counter 'j'. The inner loop would look like this:

```
for (int j = 1; j<=5; j++)
{
      System.out.print("*");
}
```

We now need an instruction to move the cursor to a new line so that the next row of stars appears on the next line:

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

This can be put together to give the following:

```
for(int i = 1; i <= 5; i++) // outer loop control
{
   for (int j = 1; j<=5; j++) // inner loop control
   {
       System.out.print("*");
   }  // inner loop ends here
   System.out.println(); // necessary to start next row on a new line
}  // outer loop ends here
```

Let's look at how the control in this program flows.

First the outer loop counter is set to 1:

```
for(int i = 1; i <= 5; i++) // outer loop counter initialised
{
   for (int j = 1; j<=5; j++)
   {
       System.out.print("*");
   }
   System.out.println();
}
```

The test of the outer loop is then checked:

```
for(int i = 1; i <= 5; i++) // outer loop counter tested
{
   for (int j = 1; j<=5; j++)
   {
       System.out.print("*");
   }
   System.out.println();
}
```

This test is found to be **`true`** so the instructions in the outer loop are executed.

The outer loop itself contains an inner **`for`** loop followed by a blank **`println`** statement. First the inner loop repeats five times.

```
for(int i = 1; i <= 5; i++)
{
   for (int j = 1; j<=5; j++) // this loop repeats 5 times
   {
       System.out.print("*");
   }
   System.out.println();
}
```

The inner loop prints five stars on the screen as follows:

```
*****
```

After the inner loop stops, there is one more instruction to complete: the command to move the cursor to a new line:

```
for(int i = 1; i <= 5; i++)
{
   for (int j = 1; j<=5; j++)
   {
       System.out.print("*");
   }
   System.out.println(); // last instruction of outer loop
}
```

This completes one cycle of the outer loop so the program returns to the beginning of this loop and increments its counter:

```
for(int i = 1; i <= 5; i++) // counter moves to 2
{
   for (int j = 1; j<=5; j++)
   {
       System.out.print("*");
   }
   System.out.println();
}
```

The test of the outer loop is then checked and found to be **`true`** and the whole process repeats, printing out a square of five stars as before.

Although a **`for`** loop is used to repeat something a fixed number of times, you don't necessarily need to know this fixed number when you are writing the program. This fixed number could be a value given to you by the user of your program, for example. Program 3.2 asks the user to determine the size of the square of stars.

Program 3.2

```
import java.util.*;

public class DisplayStars
{
  public static void main(String[] args)
  {
      int num; // to hold user response
      Scanner sc = new Scanner(System.in);
      // prompt and get user response
      System.out.println("Size of square?");
      num = sc.nextInt();
      // display square
      for(int i = 1; i <= num; i++) // loop fixed to 'num'
      {
        for (int j = 1; j<=num; j++) // loop fixed to 'num'
         {
             System.out.print("*");
         }
         System.out.println();
      }
   }
}
```

In this program you cannot tell from the code exactly how many times the loops will iterate, but you can say that they will iterate *num* number of times – whatever the user may have entered for *num*. So in this sense the loop is still fixed. Here is a sample run of program 3.2:

```
Size of square?
8
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
```

3.3 The 'while' loop

As we have already said, much of the power of computers comes from the ability to ask them to carry out repetitive tasks, so iteration is a very important form of program control. The **`for`** loop is an often used construct to implement fixed repetitions.

Sometimes, however, a repetition is required that is *not fixed* and a **`for`** loop is not the best one to use in such a case. Consider the following scenarios, for example:

- a racing game that repeatedly moves a car around a track until the car crashes;
- a ticket issuing program that repeatedly offers tickets for sale until the user chooses to quit the program;
- a password checking program that does not let a user into an application until he or she enters the right password.

Each of the above cases involves repetition; however, the number of repetitions is not fixed but depends upon some condition. The **`while`** loop offers one type of non-fixed iteration. The syntax for constructing this loop in Java is as follows:

```
while ( /* test goes here */ )
{
      // instruction(s) to be repeated go here
}
```

As you can see, this loop is much simpler to construct than a **`for`** loop. As this loop is not repeating a fixed number of times, there is no need to create a counter to keep track of the number of repetitions.

When might this kind of loop be useful? The first example we will explore is the use of the **`while`** loop to check data that is input by the user. Checking input data for errors is referred to as **input validation**.

For example, look back at program 2.2 in the last chapter, which asked the user to enter an exam mark:

```
System.out.println("What exam mark did you get?");
mark = sc.nextInt();
if (mark >= 40)
// rest of code goes here
```

The mark that is entered should never be greater than 100 or less than 0. At the time we assumed that the user would enter the mark correctly. However, good programmers never make this assumption!

Before accepting the mark that is entered and moving on to the next stage of the program, it is good practice to check that the mark entered is indeed a valid one. If it is not, then the user will be allowed to enter the mark again. This will go on until the user enters a valid mark.

We can express this using pseudocode as follows:

```
DISPLAY prompt for mark
ENTER mark
KEEP REPEATING WHILE mark < 0 OR  mark > 100
BEGIN
    DISPLAY error message to user
    ENTER mark
END
// REST OF PROGRAM HERE
```

The design makes clear that an error message is to be displayed as long as the user enters an invalid mark. The user may enter an invalid mark many times so an iteration is required here.

However, the number of iterations is not fixed as it is impossible to say how many, if any, mistakes the user will make.

This sounds like a job for the **`while`** loop.

```
System.out.println("What exam mark did you get?");
mark = sc.nextInt();
while (mark < 0  || mark > 100) // check for invalid input
{
   // display error message and allow for re-input
   System.out.println("Invalid mark: Re-enter!");
   mark = sc.nextInt();
}
if (mark >= 40)
// rest of code goes here
```

Program 3.3 below shows the whole of the previous program rewritten to include the input validation. Notice how this works – we ask the user for the mark; if it is within the acceptable range, the **`while`** loop is not entered and we move past it to the other instructions. But if the mark entered is less than zero or greater than 100 we enter the loop, display an error message and ask the user to input the mark again. This continues until the mark is within the required range.

Program 3.3

```
import java.util.*;

public class DisplayResult2
{
  public static void main(String[] args)
  {
    int mark;
    Scanner sc = new Scanner (System.in);
    System.out.println("What exam mark did you get?");
    mark = sc.nextInt();
    // input validation
    while (mark < 0 || mark > 100) // check if mark is invalid
    {
      // display error message
      System.out.println("Invalid mark: please re-enter");
      // mark must be re-entered
      mark = sc.nextInt();
    }
    // by this point loop is finished and mark will be valid
    if (mark >= 40)
    {
      System.out.println("Congratulations, you passed");
    }
    else
    {
      System.out.println("I'm sorry, but you failed");
    }
    System.out.println("Good luck with your other exams");
   }
}
```

Here is a sample program run:

```
What exam mark did you get?
101
Invalid mark: please re-enter
-10
Invalid mark: please re-enter
10
I'm sorry, but you failed
Good luck with your other exams
```

3.4 The 'do...while' loop

There is one more loop construct in Java that we need to tell you about: the **do...while** loop.

The **do...while** loop is another variable loop construct, but, unlike the **while** loop, the **do...while** loop has its test at the *end* of the loop rather than at the *beginning*.

The syntax of a **do...while** loop is given below:

```
do
{
      // instruction(s) to be repeated go here
} while ( /* test goes here */ ); // note the semi-colon at the end
```

You are probably wondering what difference it makes if the test is at the end or the beginning of the loop. Well, there is one subtle difference. If the test is at the end of the loop, the loop will iterate *at least once*. If the test is at the beginning of the loop, however, there is a possibility that the condition will be **`false`** to begin with, and the loop is never executed. A **`while`** loop therefore executes *zero or more times* whereas a **`do...while`** loop executes *one or more times*.

To make this a little clearer, look back at the **`while`** loop we just showed you for validating exam marks. If the user entered a valid mark initially (such as 66), the test to trap an invalid mark (`mark <0 || mark > 100`) would be **`false`** and the loop would be skipped altogether. A **`do...while`** loop would not be appropriate here as the possibility of never getting into the loop should be left open.

When would a **`do...while`** loop be suitable? Well, any time you wish to code a non-fixed loop that must execute at least once. Usually, this would be the case when the test can take place only *after* the loop has been entered.

To illustrate this, think about all the programs you have written so far. Once the program has done its job it terminates – if you want it to perform the same task again you have to go through the whole procedure of running that program again.

In many cases a better solution would be to put your whole program in a loop that keeps repeating until the user chooses to quit your program. This would involve asking the user each time if he or she would like to continue repeating your program, or to stop.

A **`for`** loop would not be the best loop to choose here as this is more useful when the number of repetitions can be predicted. A **`while`** loop would be difficult to use, as the test that checks the user's response to a question cannot be carried out at the beginning of the loop. The answer is to move the test to the end of the loop and use a **`do...while`** loop as follows:

```
char response; // variable to hold user response
do // place code in loop
{
     // program instructions go here
      System.out.println("another go (y/n)?");
      response = sc.next().charAt(0); // get user reply
}  while (response == 'y'); // this test must be at the end of the loop
```

For example, program 3.4 below amends program 1.4, which calculated the cost of a product, by allowing the user to repeat the program as often as he or she chooses.

Program 3.4

```
import java.util.*;

public class FindCost4
{
 public static void main(String[] args )
 {
   double price, tax;
   char reply;
   Scanner sc = new Scanner(System.in);
   do
   {
     // these instructions as before
     System.out.println("*** Product Price Check ***");
     System.out.print("Enter initial price: ");
     price = sc.nextDouble();
     System.out.print("Enter tax rate: ");
     tax = sc.nextDouble();
     price = price * (1 + tax/100);
     System.out.println("Cost after tax = " + price);
     // now see if user wants another go
     System.out.println();
     System.out.print("Would you like to enter another product(y/n)?: ");
     reply = sc.next().charAt(0);
     System.out.println();
   } while (reply == 'y' || reply == 'Y');
 }
}
```

Notice the test of the **do...while** loop allows the user to enter either a lower case or an upper case 'Y' to continue running the program:

```
while (reply == 'y' || reply == 'Y');
```

Here is sample program run:

```
*** Product Price Check ***
Enter initial price: 50
Enter tax rate: 10
Cost after tax = 55.0
Would you like to enter another product (y/n)?: y


*** Product Price Check ***
Enter initial price: 70
Enter tax rate: 5
Cost after tax = 73.5

Would you like to enter another product (y/n)?: Y
```

```
*** Product Price Check ***
Enter initial price: 200
Enter tax rate: 15
Cost after tax = 230.0

Would you like to enter another product (y/n)?: n
```

Another way to allow a program to be run repeatedly using a **do…while** loop is to include a *menu* of options within the loop (this was very common in the days before windows and mice). The options themselves are processed by a **switch** statement. One of the options in the menu list would be the option to quit and this option is checked in the **while** condition of the loop. Program 3.5 is a reworking of program 2.5 of the previous chapter using this technique.

Program 3.5

```
import java.util.*;

public class TimetableWithLoop
{
  public static void main(String[] args)
  {
       char group, response;
       Scanner sc = new Scanner(System.in);
       System.out.println("***Lab Times***");
       do // put code in loop
       {
          // offer menu of options
         System.out.println(); // create a blank line
         System.out.println("[1] TIME FOR GROUP A");
         System.out.println("[2] TIME FOR GROUP B");
         System.out.println("[3] TIME FOR GROUP C");
         System.out.println("[4] QUIT PROGRAM");
         System.out.print("enter choice [1,2,3,4]: ");
         response = sc.next().charAt(0); // get response
         System.out.println(); // create a blank line
         switch(response)  // process response
         {
           case '1': System.out.println("10.00 a.m ");
                     break;
           case '2': System.out.println("1.00 p.m ");
                     break;
           case '3': System.out.println("11.00 a.m ");
                     break;
           case '4': System.out.println("Goodbye ");
                     break;
           default:  System.out.println("Options 1-4 only!");
         }
       }  while (response != '4'); // test for Quit option
    }
}
```

Notice that the menu option is treated as a character here, rather than an integer. So option 1 would be entered as the character '1' rather than the number 1, for example. The advantage of treating the menu option as a character rather than a number is that an incorrect menu entry would not result in a program crash if the value entered was non-numeric. Here is a sample run of this program:

```
***Lab Times***

[1] TIME FOR GROUP A
[2] TIME FOR GROUP B
[3] TIME FOR GROUP C
[4] QUIT PROGRAM
enter choice [1,2,3,4]:  2

1.00 p.m

[1] TIME FOR GROUP A
[2] TIME FOR GROUP B
[3] TIME FOR GROUP C
[4] QUIT PROGRAM
enter choice [1,2,3,4]:  5

Options 1-4 only!

[1] TIME FOR GROUP A
[2] TIME FOR GROUP B
[3] TIME FOR GROUP C
[4] QUIT PROGRAM
enter choice [1,2,3,4]:  1
10.00 a.m

[1] TIME FOR GROUP A
[2] TIME FOR GROUP B
[3] TIME FOR GROUP C
[4] QUIT PROGRAM
enter choice [1,2,3,4]:  3

11.00 a.m
```

```
[1] TIME FOR GROUP A
[2] TIME FOR GROUP B
[3] TIME FOR GROUP C
[4] QUIT PROGRAM
enter choice [1,2,3,4]:  4

Goodbye
```

3.5 Picking the right loop

With three types of loop to choose from in Java, it can sometimes be difficult to decide upon the best one to use in each case. Here are some general guidelines that should help you:

- if the number of repetitions required can be determined prior to entering the loop – use a **`for`** loop;
- if the number of repetitions required cannot be determined prior to entering the loop, and you wish to allow for the possibility of zero repetitions – use a **`while`** loop;
- if the number of repetitions required cannot be determined before the loop, and you require at least one repetition of the loop – use a **`do...while`** loop.

The guidelines above will help you to pick the most appropriate loop construct to use in each case. However, it is possible to pick *any loop* to implement *any type of repetition*.

For example, program 3.1 used a **`for`** loop to display a countdown of numbers from 10 down to 1. The **`for`** loop is the most appropriate loop to use here as the number of repetitions can be determined. However, we could choose either a **`while`** or a **`do...while`** to implement this. Neither of these loops have a counter associated with them, so to use them to implement this task we will need to create our own loop counter prior to the loop. We will also need to modify the loop counter within the loop. Program 3.6 rewrites program 3.1 by using a **`while`** loop instead of the more obvious choice of a **`for`** loop.

Program 3.6

```
public class CountdownUsingWhile
{
  public static void main(String[] args)
  {
    int i = 10; // declare and initialize a loop counter before loop
    System.out.println("***Numbers from 10 to 1***");
    while (i>=1)  // 'while' loop just consists of a test
    {
      System.out.println(i);
      i--; // modify the counter within the 'while' loop
    }
  }
}
```

When run, this program behaves in exactly the same way as program 3.1. Similarly, we could use a **for** loop to implement a repetition that may be more ideally suited to one of the other two loops. We do this by leaving the start condition of the counter, and the counter modifier blank. This just leaves the test, and so effectively reduces the **for** loop to a **while** loop. As an example, program 3.7 rewrites program 3.3 by using a **for** loop to check for valid exam marks, rather than the more obvious choice of a **while** loop.

Program 3.7

```
import java.util.*;

public class DisplayResult3UsingFor
{
  public static void main(String[] args)
  {
    int mark;
    Scanner sc = new Scanner(System.in);
    System.out.println("What exam mark did you get?");
    mark = sc.nextInt();
    for (; mark < 0 || mark > 100; ) // just leave the test here
    {
      System.out.println("Invalid mark: please re-enter");
      mark = sc.nextInt();
    }
    if (mark >= 40)
    {
      System.out.println("Congratulations, you passed");
    }
    else
    {
      System.out.println("I'm sorry, but you failed");
    }
    System.out.println("Good luck with your other exams");
   }
}
```

Again, this program will behave in exactly the same way as the original program 3.3. Notice that when the **for** loop is reduced to containing just a test, semi-colons are still required to indicate that there is no start condition and no counter modification:

```
// semi-colons either side of the test still required
for (; mark < 0 || mark > 100; )
```

While the loops can be used interchangeably in this way, each is designed for a specific purpose; so it is best to use the most appropriate loop for each situation.

Self-test questions

1 How does *iteration* differ from *selection*?

2 Consider the following program:

```
import java.util.*;

public class IterationQ2
{
    public static void main(String[] args)
    {
        int num;
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number ");
        num = sc.nextInt();
        for(int i= 1; i< num; i++)
        {
            System.out.println("YES");
            System.out.println("NO");
        }
        System.out.println("OK");
    }
}
```

a) What would be the output of this program if the user entered 5 when prompted?

b) What would be the output of this program if the user entered 0 when prompted?

3 What would be the output from the following program?

```
public class IterationQ3
{
    public static void main(String[] args)
    {
        for(int i=1; i<=10; i++)
        {
            if (i%2 == 0)
            {
                System.out.println(i);
            }
        }
    }
}
```

4 Examine the program below that aims to allow a user to keep guessing a secret number. Part of the code has been replaced by a comment:

```
import java.util.*;

public class IterationQ4
{
    public static void main(String[] args)
    {
        final int SECRET = 321;
```

```
        int num;
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number ");
        num = sc.nextInt();
        do
        {
            System.out.println("Wrong number, try again");
            num = sc.nextInt();
        }  while (/* test to be completed */)
        System.out.println("Well done, right number");
    }
}
```

a) Why is a **do...while** loop not a good choice for this repetition?

b) Replace the **do...while** loop with a more appropriate loop construct.

c) Replace the comment with an appropriate test for this loop.

Programming exercises

1 Modify the program given in self-test question 3 above, so that the user enters a number and the program displays all the numbers from 1 up to the number entered. The program should identify which of these numbers are odd and which are even. For example, if the user entered 5 the program should display something like the following:

1 is odd

2 is even

3 is odd

4 is even

5 is odd

2 a) Using a **for** loop, write a program that displays a "6 times" multiplication table; the output should look like this:

```
1 × 6 = 6
2 × 6 = 12
3 × 6 = 18
4 × 6 = 24
5 × 6 = 30
6 × 6 = 36
```

```
 7 × 6 = 42
 8 × 6 = 48
 9 × 6 = 54
10 × 6 = 60
11 × 6 = 66
12 × 6 = 72
```

b) Adapt the program so that instead of a "6 times" table, the user chooses which table is displayed.

c) Adapt the program further by replacing the **`for`** loop with a **`while`** loop.

3 Implement program 3.2 that allows the user to determine the size of a square of stars and then

a) adapt it so that the user is allowed to enter a size only between 2 and 100;

b) adapt the program further so that the user can choose whether or not to have another go.

4 Consider a vending machine that offers the following options:

```
[1] Get gum
[2] Get chocolate
[3] Get popcorn
[4] Get juice
[5] Display total sold so far
[6] Quit
```

Design and implement a program that continuously allows users to select from these options. When options 1–4 are selected an appropriate message is to be displayed acknowledging their choice. For example, when option 3 is selected the following message could be displayed:

```
Here is your popcorn
```

When option 5 is selected, the number of each type of item sold is displayed. For example:

```
3 items of gum sold
2 items of chocolate sold
6 items of popcorn sold
9 items of juice sold
```

When option 6 is chosen the program terminates. If an option other than 1–6 is entered an appropriate error message should be displayed, such as:

```
Error, options 1-6 only!
```