Iteration

4

Outcomes:

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*;
- *use a* **break** *statement to terminate a loop*;
- *use a* **continue** *statement to skip an iteration of a loop*;
- *explain the term* ***input validation*** *and write simple validation routines.*

4.1 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. As an example, consider a program that needs to display a square of stars (five by five) on the screen as follows:

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


Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_4

This could be achieved with five output statements executed in sequence, as shown in below in a program which we have called `DisplayStars`:

DisplayStars

```
public class DisplayStars
{
      public static void main (String[] args)
      {
            System.out.println("*****"); // instruction to display one row
            System.out.println("*****"); // instruction to display one row
            System.out.println("*****"); // instruction to display one row
            System.out.println("*****"); // instruction to display one row
            System.out.println("*****"); // instruction to display one row
       }
}
```

While this produces the desired result, the program actually consists just of the following instruction to print out one row, but repeated 5 times:

```
System.out.println("*****"); // this instruction is written 5 times
```

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* another four times. Something like:

```
public class DisplayStars
{
    public static void main (String[] args)
    {
        // CARRY OUT THE FOLLOWING INSTRUCTION 5 TIMES
        System.out.println("*****");
    }
}
```

Iteration is the form of program control that allows us to instruct the computer to carry out a task several times 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**; we say that the loop **iterates** a certain number of times. There are three types of loop in Java:

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

We will consider each of these in turn.

4.2 The 'for' Loop

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. For example, the program below re-writes `DisplayStars` by making use of a **`for`** loop. Take a look at it and then we will discuss it:

DisplayStars2

```
public class DisplayStars2
{
    public static void main (String[] args)
    {
        for(int i = 1; i <= 5; i++) // loop to repeat 5 times
        {
            System.out.println("*****"); // instruction to display one row
        }
    }
}
```

As you can see there are three bits of information in the header of the **`for`** loop, each bit separated by a semi-colon:

```
for(int i = 1; i <= 5; i++) // three bits of information in the brackets
{
     System.out.println("*****");
}
```

All three bits of information relate to 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. In this case we have called our counter `i`, but we could give it any variable name—often though, simple names like `i` and `j` are chosen.

Let's look carefully at how this **`for`** loop works. First the counter is initialized to some value. We have decided to initialize it to 1:

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

Notice that the loop counter `i` is *declared* as well as initialized in the header of the loop. Although it is possible to declare the counter variable prior to the loop, declaring it within the header restricts the use of this variable to the loop itself. This is often preferable.

The second bit of information in the header is a test, much like a test when carrying out selection. When the test returns a **`boolean`** value of **`true`** the loop repeats; when it returns a **`boolean`** value of **`false`** the loop ends. In this case the

counter is tested to see if it is less than or equal to 5 (as we wish to repeat this loop 5 times):

```
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 loop is entered. We sometimes refer to the instructions inside the loop as the **`body`** of the loop. As with **`if`** statements, the braces of the **`for`** loop can be omitted when only a single instruction is required in the body of the loop—but for clarity we will always use braces with our loops. When 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 third bit of information in the header of the **`for`** loop is executed. The third bit of information *changes* the value of the counter so that eventually the loop test will be **`false`**. If we want the loop to repeat 5 times and we have started the counter off at 1, we should *add* 1 to the counter each time we go around the loop:

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

After the first increment, the counter now has the value of 2. Once the counter has been changed the test is examined again to see if the loop should repeat:

```
for(int i = 1; i <= 5; i++) // counter tested again
{
   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 five:

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

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

Now that you have seen one example of the use of a **for** loop, the general form of a **for** loop can be given as follows:

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

Be very careful that the loop counter and the test achieve the desired result. For example, consider the following test:

```
for(int i = 1; i >= 10; i++) // something wrong with this test!
{
   // instruction(s) to be repeated go here
}
```

Can you see what is wrong here?

The test to continue with the loop is that the counter be *greater* than or equal to 10 (`i >= 10`). However, the counter starts at 1 so this test is immediately **false**! Because this test would be **false** immediately, the loop does not repeat at all and it is skipped altogether!

Now consider this test:

```
for(int i = 1; i >= 1; i++) // something wrong with this test again!
{
   // instruction(s) to be repeated go here
}
```

Can you see what is wrong here?

The test to continue with the loop is that the counter be greater than or equal to 1 (`i >= 1`). However, the counter starts at 1 and increases by 1 each time, so this test will always be **`true`**! Because this test would be **`true`** always, the loop will never stop repeating when it is executed!

As long as you are careful with your counter and your test, however, it is a very easy matter to set your **`for`** loop to repeat a certain number of times. If, for example, we start the counter at 1 and increment it by 1 each time, and we need to carry out some instructions 70 times, we could have the following test in the **`for`** loop:

```
for(int i = 1; i <= 70; i++) // this loop carries out the instructions 70 times
{
   // instruction(s) to be repeated goes here
}
```

4.2.1 Varying the Loop Counter

The `DisplayStars2` program illustrated a 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.

For example, we could have re-written the **`for`** loop of the above program so that the counter starts at 0 instead of 1. In that case, if we wish the **`for`** loop to still execute the instructions 5 times the counter should reach 4 and not 5:

```
// this counter starts at 0 and goes up to 4 so the loop still executes 5 times
for(int i = 0; i <= 4; i++)
{
   System.out.println("*****");
}
```

Another way to ensure that the counter does not reach a value greater than 4 is to insist that the counter stays below 5. In this case we need to use the "less than" operator (<) instead of the "less than or equal to" operator (<=):

```
// this loop still executes 5 times
for(int i = 0; i < 5; i++)
{
   System.out.println("*****");
}
```

We can also change the way we modify the counter after each iteration. Returning to the original **for** loop, we would increment the counter by 2 each time instead of 1. If we still wish the loop to repeat 5 times we could start at 2 and get the counter to go up to 10:

```
// this loop still executes 5 times
for(int i = 2; i <= 10; i = i+2) // the counter moves up in steps of 2
{
    System.out.println("*****");
}
```

Finally, counters can move down as well as up. As an example, look at the following program that prints out a countdown of the numbers from 10 down to 1.

Countdown

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

Here the counter starts at 10 and is reduced by 1 each time. The loop stops when the counter falls below the value of 1. Note the use of the loop counter *inside* the loop:

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

This is perfectly acceptable as the loop counter is just another variable. However, when you do this, 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! Running the `Countdown` program gives us the following result:

```
*** Numbers from 10 to 1 ***

10
9

8

7

6
```

```
5
4
3
2
1
```

4.2.2 The Body of the Loop

The body of the loop can contain any number and type of instructions, including variable declarations, **`if`** statements, **`switch`** statements, or even another loop! For example, the `DisplayEven` program below modifies our `Countdown` by including an **`if`** statement inside the **`for`** loop so that only the *even* numbers from 10 to 1 are displayed:

DisplayEven

```
public class DisplayEven
{
    public static void main(String[] args)
    {
        System.out.println("*** Even numbers from 10 to 1 ***");
        System.out.println();
        for(int i=10; i >= 1; i--) // loop through the numbers 10 down to 1
        {
            // body of the loop contains in 'if' statement
            if (i%2 == 0) // check if number is even
            {
                System.out.println(i); // number displayed only when it is checked to be even
            }
        }
    }
}
```

You can see that the body of the **`for`** loop contains within it an **`if`** statement. The test of the **`if`** statement checks the current value of the loop counter '`i`' to see if it is an even number:

```
for(int i=10; i >= 1; i--)
{
    if (i%2 == 0) // use the modulus operator to check the value of the loop counter
    {
        System.out.println(i);
    }
}
```

An even number is a number that leaves no remainder when divided by 2, so we use the modulus operator (`%`) to check this. Now the loop counter is displayed only if it is an even number. Running the program gives us the obvious results:

```
*** Even numbers from 10 to 1 ***
10
8
6
4
2
```

In this example we included an **if** statement inside the **for** loop. It is also possible to have one **for** loop inside another. When we have one loop inside another we refer to these loops as **nested** loops. As an example of this consider the program DisplayStars3 below, which displays a square of stars as before, but this time uses a pair of nested loops to achieve this:

DisplayStars3

```
public class DisplayStars3
{
   public static void main (String[] args)
   {
      for(int i = 1; i <= 5; i++) // outer loop as before
      {
         for (int j = 1; j <= 5; j++) // inner loop to display one row of stars
         {
            System.out.print("*");
         }  // inner loop ends here
         System.out.println(); // necessary to start next row on a new line
      } // outer loop ends here
   }
}
```

You can see that the outer **for** loop is the same as the one used previously in DisplayStars2. Whereas in the original program we had a single instruction to display a single row of stars inside our loop:

```
System.out.println("*****"); // original instruction inside the 'for' loop
```

in DisplayStars3 we have replaced this instruction with *another* **for** loop, followed by a blank println instruction:

```
// new instructions inside the original 'for' loop to print a single row of stars
for (int j = 1; j <= 5; j++) // new name for this loop counter
{
   System.out.print("*");
}
System.out.println();
```

Notice that when we place one loop inside another, we need a fresh name for the loop counter in the nested loop. In this case we have called the counter 'j'. These instructions together allow us to display a single row of 5 stars and move to a new line, ready to print the next row.

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 initialized
{
    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 body of the outer loop is executed. 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.

`DisplayStars3` displayed a five by five square of stars. Now take a look at the next program and see if you can work out what it does. Look particularly at the header of the inner loop:

DisplayShape

```
public class DisplayShape
{
   public static void main (String[] args)
   {
       for(int i = 1; i <= 5; i++) // outer loop controlling the number of rows
       {
         for (int j = 1; j <= i; j++) // inner loop controlling the number of stars in one row
         {
                 System.out.print("*");
         }
         System.out.println();
       }
   }
}
```

You can see this is very similar to the previous program, except that in that program the inner loop displayed 5 stars each time. In this case the number of stars is not fixed to a number, but to the value of the outer loop counter `i`:

```
for(int i = 1; i <= 5; i++) // outer loops controls the number of rows
{
    // inner loop determines how many stars in each row
    for (int j = 1; j <= i; j++) // inner loop displays 'i' number of stars
    {
        System.out.print("*");
    }
    System.out.println();
}
```

The first time around this loop the inner loop will display only 1 star in the row as the `i` counter starts at 1. The second time around this loop it will display 2 stars as the `i` counter is incremented, then 3 stars. Eventually it will display 5 stars the last time around the loop when the outer `i` counter reaches 5. Effectively this means the program will display a *triangle* of stars as follows:

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

4.2.3 Revisiting the Loop Counter

Before we move on to look at other kinds of loops in Java it is important to understand that, 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. This number could then be used to test against your loop counter. The program below modifies `DisplayStars3` by asking the user to determine the size of the square of stars.

DisplayStars4

```
import java.util.Scanner;

public class DisplayStars4
{
    public static void main(String[] args)
    {
        int num; // to hold user response
        Scanner keyboard = new Scanner(System.in);
        // prompt and get user response
        System.out.println("Size of square?");
        num = keyboard.nextInt();
        // display square
        for(int i = 1; i <= num; i++) // number of rows fixed to 'num'
        {
            for (int j = 1; j <= num; j++) // number of stars in a row 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 `DisplayStars4`:

```
Size of square?
7
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
* * * * * * *
```

Here is another sample run:

```
Size of square?
3
* * *
* * *
* * *
```

4.3 The 'while' Loop

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 the program `DisplayResult` in the last chapter, which asked the user to enter an exam mark:

```
System.out.println("What exam mark did you get?");
mark = keyboard.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:

```
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 every time 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 = keyboard.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 = keyboard.nextInt();
}
if (mark >= 40)
// rest of code goes here
```

The program below shows the whole of the `DisplayResult` 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.

DisplayResult2

```
import java.util.Scanner;

public class DisplayResult2
{
    public static void main(String[] args)
    {
        int mark;
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What exam mark did you get?");
        mark = keyboard.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 = keyboard.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
```

4.4 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 the 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 the 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 the 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 = keyboard.next().charAt(0); // get user reply
} while (response == 'y' || response == 'Y'); // test must be at the end of the loop
```

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 (response == 'y' || response == 'Y');
```

As an example of this application of the **do...while** loop, the program below amends the FindCost3 program of Chap. 2, which calculated the cost of a product, by allowing the user to repeat the program as often as he or she chooses.

FindCost4

```
import java.util.Scanner;

public class FindCost4
{
    public static void main(String[] args)
    {
        double price, tax;
        char reply;
        Scanner keyboard = new Scanner(System.in);
        do
        {
            // these instructions as before
            System.out.println("*** Product Price Check ***");
            System.out.print("Enter initial price: ");
            price = keyboard.nextDouble();
            System.out.print("Enter tax rate: ");
            tax = keyboard.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 = keyboard.next().charAt(0);
            System.out.println();
        } 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. The program below is a reworking of the time table program of the previous chapter using this technique.

TimetableWithLoop

```
import java.util.Scanner;

public class TimetableWithLoop
{
    public static void main(String[] args)
    {
        char group, response;
        Scanner keyboard = 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 = keyboard.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 interpreted 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
```

4.5 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, especially as it is technically possible to pick *any type of loop* to implement *any type of repetition*! For example, **`while`** and **`do...while`** loops *can* be used for fixed repetitions by introducing your own counter and checking this counter in the test of the loop. However, it is always best

to pick the most appropriate loop construct to use in each case, as this will simplify the logic of your code. 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 iterations—use a **while** loop;
- if the number of repetitions required cannot be determined before the loop, and you require at least one iteration of the loop—use a **do...while** loop.

4.6 The 'break' Statement

In the previous chapter we met the **break** statement when looking at **switch** statements. Here for example is a **switch** statement from the previous chapter that processed a student's timetable:

```
switch(group)
{
   case 'A': System.out.print("10.00 a.m ");
             break; // terminates switch
   case 'B': System.out.print("1.00 p.m ");
             break; // terminates switch
   case 'C': System.out.print("11.00 a.m ");
             break; // terminates switch
   default: System.out.print("No such group");
}
```

Here the **break** statement allowed the **switch** to terminate without processing the remaining **cases**. The **break** statement can also be used with Java's loops to terminate a loop before it reaches its natural end. For example, consider a program that allows the user a maximum of three attempts to guess a secret number. This is an example of a non-fixed iteration—but the iteration does have a fixed upper limit of three.

We could use any of the loop types to implement this. If we wished to use a **for** loop, however, we would need to make use of the **break** statement. Take a look at the following program that does this for a secret number of 27:

SecretNumber

```
import java.util.Scanner;

// This program demonstrates the use of the 'break' statement inside a 'for' loop

public class SecretNumber
{

   public static void main(String[] args)
   {
      Scanner keyboard = new Scanner (System.in);
      final int SECRET = 27; // secret number
      int num; // to hold user's guess
      boolean guessed = false; // so far number not guessed

      System.out.println("You have 3 goes to guess the secret number");
      System.out.println("HINT: It is a number less than 50!");
      System.out.println();

      // look carefully at this loop
      for (int i= 1; i <= 3; i++) // loop repeats 3 times
      {
         System.out.print("Enter guess: ");
         num = keyboard.nextInt();
         // check guess
         if (num == SECRET) // check if number guessed correctly
         {
            guessed = true; // record number has been guessed correctly
            break; // exit loop
         }
      }

      // now check to see if the number was guessed correctly or not
      if (guessed)
      {
         System.out.println("Number guessed correctly");
      }
      else
      {
         System.out.println("Number NOT guessed");
      }
   }
}
```

The important part of this program is the **`for`** loop. You can see that it has been written to repeat three times:

```
for (int i= 1; i <= 3; i++) // loop executes 3 times
{
   System.out.print("Enter guess: ");
   num = keyboard.nextInt();
   // code here to check the guess
}
```

Each time around the loop the user gets to have a guess at the secret number. We need to do two things if we determine that the guess is correct. Firstly, set a **`boolean`** variable to **`true`** to indicate a correct guess. Then, secondly, we need to terminate the loop, even if this is before we reach the third iteration. We do so by using a **`break`** statement if the guess is correct:

```
for (int i= 1; i <= 3; i++)
{
    System.out.print("Enter guess: ");
    num = keyboard.nextInt();
    if (num == SECRET) // check if number guessed correctly
    {
        guessed = true; // record number has been guessed correctly
        break; // exit loop even if it has not yet finished three iterations
    }
}
```

Here is a sample program run:

```
You have 3 goes to guess the secret number
HINT: It is a number less than 50!
Enter guess: 49
Enter guess: 27
Number guessed correctly
```

Here the user guessed the number after two attempts and the loop terminated early due to the **break** statement. Here is another program run where the user fails to guess the secret number:

```
You have 3 goes to guess the secret number
HINT: It is a number less than 50!

Enter guess: 33
Enter guess: 22
Enter guess: 11
Number NOT guessed
```

Here the **break** statement is never reached so the loop iterates three times without terminating early.

4.7 The 'continue' Statement

Whereas the **break** statement forces a loop to terminate, a **continue** statement forces a loop to skip the remaining instructions in the body of the loop and to *continue* to the next iteration. As an example of this here is a reminder of the earlier program that displayed the even numbers from 10 down to 1:

***DisplayEven* – a reminder**

```
public class DisplayEven
{
   public static void main(String[] args)
   {
      System.out.println("*** Even numbers from 10 to 1 ***");
      System.out.println();
      for(int i=10; i >= 1; i--) // loop through the numbers 10 down to 1
      {
         // body of the loop contains in 'if' statement
         if (i%2 == 0) // check if number is even
         {
            System.out.println(i); // number displayed only when it is checked to be even
         }
      }
   }
}
```

Here the body of the loop displayed the loop counter if it was an even number. An alternative approach would have been to skip a number if it was odd and move on to the next iteration of the loop. If the number is not skipped then it must be even, so can be displayed. This is what we have done in the following program:

DisplayEven2

```
public class DisplayEven2
{
   public static void main(String[] args)
   {
      System.out.println("*** Even numbers from 10 to 1 ***");
      System.out.println();
      for(int i=10; i>=1; i--)
      {
         if (i%2 != 0) // check if number is NOT even
         {
            continue; // skips the rest of this iteration and moves to the next iteration
         }
         System.out.println(i); // even number only displayed if we have not skipped this iteration
      }
   }
}
```

The **`if`** statement checks to see if the number is odd (not even). If this is the case the rest of the instructions in the loop can be skipped with a **`continue`** statement, so the loop moves to the next iteration:

```
if (i%2 != 0) // check if number is NOT even
{
   continue; // skips the rest of the loop body and moves to the next iteration
}
System.out.println(i); // this line only executed if this iteration is not skipped
```

The last `println` instruction is only executed if the number is even and the iteration has not been skipped. Of course, the result of running this program will be the same as the result of running the original program.

4.8 Self-test Questions

1. Consider the following program:

```
public class IterationQ1
{
    public static void main(String[] args)
    {
        for(int i = 1; i <= 4; i++)
        {
            System.out.println("YES");
        }
        System.out.println("OK");
    }
}
```

   (a) How many times does this **for** loop repeat?
   (b) What would be the output of this program?

2. Consider the following program:

```
public class IterationQ2
{
    public static void main(String[] args)
    {
        for(int i = 1; i < 4; i++)
        {
            System.out.println("YES");
            System.out.println("NO");
        }
        System.out.println("OK");
    }
}
```

   (a) How many times does this **for** loop repeat?
   (b) What would be the output of this program?

3. Consider the following program:

```
import java.util.Scanner;

public class IterationQ3
{
    public static void main(String[] args)
    {
        int num;
        Scanner keyboard = new Scanner(System.in);

        System.out.print("Enter a number: ");
        num = keyboard.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?

4. Consider the following program

```
public class IterationQ4
{
    public static void main(String[] args)
    {
        for(int i=1; i<=15; i= i +2)
        {
            System.out.println(i);
        }
    }
}
```

(a) How many times does this **for** loop repeat?
(b) What would be the output of this program?
(c) What would be the consequence of changing the test of the loop to (`i >= 15`)?

5. Consider the following program:

```
public class IterationQ5
{
    public static void main(String[] args)
    {
        for(int i=5; i>=2; i--)
        {
            switch (i)
            {
                case 1: case 3: System.out.println("YES"); break;
                case 2: case 4: case 5: System.out.println("NO");
            }
        }
        System.out.println("OK");
    }
}
```

(a) How many times does this **for** loop repeat?
(b) What would be the output of this program?
(c) What would be the consequence of changing the loop counter to (`i++`) instead of (`i--`)

6. What would be the output from the following program?

```
public class IterationQ6
{
    public static void main(String[] args)
    {
        for(int i=1; i <= 3; i++)
        {
                for(int j=1; j <= 7; j++)
                {
                        System.out.print("*");
                }
                System.out.println();
        }
    }
}
```

7. Examine the program below that aims to allow a user to guess the square of a number that is entered. Part of the code has been replaced by a comment:

```
import java.util.Scanner;

public class IterationQ7
{
    public static void main(String[] args)
    {
        int num, square;
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter a number ");
        num = keyboard.nextInt();
        System.out.print("Enter the square of this number ");
        square = keyboard.nextInt();
        // loop to check answer
        while (/* test to be completed */)
        {
            System.out.println("Wrong answer, try again");
            square = keyboard.nextInt();
        }
        System.out.println("Well done, right answer");
    }
}
```

(a) Why is a **while** loop preferable to a **for** loop or a **do...while** loop here?

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

8. What would be the output of the following program?

```
public class IterationQ8
{
    public static void main(String[] args)
    {
        for(int i=1; i<=10; i++)
        {
            if (i > 5)
            {
                break;
            }
            System.out.println(i);
        }
    }
}
```

9. What would be the output of the following program?

```
public class IterationQ9
{
    public static void main(String[] args)
    {
        for(int i=1; i<=10; i++)
        {
            if (i <= 5)
            {
                    continue;
            }
            System.out.println(i);
        }
    }
}
```

4.9 Programming Exercises

1. Implement a few of the programs from this chapter, and then implement the programs from the self-test questions above in order to verify your answers to those questions.

2. (a) Modify the `DisplayEven` program from Sect. 4.2.2 so that the program displays the even numbers from 1 to 20 instead of from 10 down to 1.
   (b) Modify the program further so that the user enters a number and the program displays all the even numbers from 1 up to the number entered by the user.
   (c) Modify the program again so that it identifies 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
```

3. Write a program that makes use of nested **`for`** loops to display the following shapes:

   (a)
```
* * * * * *
* * * * * *
* * * * * *
```

   (b)
```
* * *
* * *
* * * * * * * * *
* * * * * * * * *
* * *
* * *
```

   Hint: make use of an ***`if...else`*** *statement inside your for loops.*

   (c)
```
* * * *
* * *
* *
*
```

6. (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) Modify the program further by making use of a **while** loop to carry out some *input validation* that ensures that the user enters a number that is never less than 2. If a number less than 2 is entered an error message should be displayed and the user is asked to enter another number.
   (d) Finally, make use of a **do...while** loop so that the user is asked to enter 'y' or 'n' to indicate if they wish to run the program again. Ideally the program should run again if the user enters 'y' or 'Y'.

7. Implement the program `DisplayStars4` from Sect. 4.2.3 (which 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 20;
   (b) adapt the program further so that the user can choose whether or not to have another go.

8. Modify programming Exercise 7, from Sect. 2.12, that caries out some calculations related to a circle as follows:

   (a) Add input validation to ensure that the radius entered is always non-negative
   (b) Provide a menu interface for this program. For example:

```
[1] Set radius
[2] Display radius
[3] Display area
[4] Display perimeter
[5] Quit
```

9. 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!
```