6. Programming – Loops

***Note**: some example programs are from, or based on, examples from Java for Everyone (C Horstmann), the course text.*

So far we have only seen programs that do things once – but most programs involve repeated code. For example:

- Asking for user input repeatedly until the user enters it correctly.
- Performing iterative calculations on data.

In practice very few programs do not involve iteration, or loops.
Java has three basic ways of building loops:

- **while loops** – you use these when you don't know at the start of the loop how many times it will go around.
- **do-while loops** - you use these when you don't know at the start of the loop how many times it will go around but you *do* know it will be at least once.
- **for loops** – you use these when you *do* know at the start of the loop how many times it will go around. These also have a newer variant that we'll get back to in a later chapter – usually called **for-each loops**.

First we'll introduce each type of loop and the loop at some examples of how to use them.

While Loops

Probably the most common kind of loop – the syntax is:

```
while(condition) {
      code
}
```

The code between the {...} is executed over and over again, as long as the condition is true.
As long as the condition is true, the code is executed – the condition is essentially the same as the condition we use for if statements: a Boolean expression that evaluates to true or false. We usually use the term *loop body* for the code inside the loop. For example.

```
while (currentAmount < TARGET) {
      currentAmount += currentAmount * 0.05;
      years ++;
}
```

The loop body is in red. This is a simplified version of the code you need if you want to work out how long it will take for an initial amount of money to grow to a target amount given a particular interest rate – `TARGET` is the amount of money we want our investment to grow to; and `currentAmount` is how much we have at any particular point in time (0.05 is the interest rate –

5%). The time taken for an amount of money to grow to some target value is not something you can (easily) know before you start the loop – which is why we use a while loop. In this case, we are using the value of the variable called years to count the number of years.

Here is a complete program based on this loop:

```
final double TARGET = 1000.0;
double initialAmount = 500.0;
final double INTEREST_RATE = 0.05;
int years = 0;

double currentAmount = initialAmount;
while (currentAmount < TARGET) {
     currentAmount += currentAmount * 0.05;
     years ++;
}

System.out.println("It takes " + years + " years at " +
     INTEREST_RATE*100 + "% interest to grow " +
     initialAmount + " to " + TARGET);
```

KEY POINT – No ; at the end of While Statements

Just like the if statement, the while statement does not have a ";" after it normally, and just like the if statement it's usually wrong. If we take the loop above:

```
//WRONG!
while (currentAmount < TARGET) ;
{
     currentAmount += currentAmount * 0.05;
     years ++;
}
```

Again we've added an incorrect ";" in read – so this says "while currentAmount is less than `TARGET` do nothing". Since doing nothing will not change whether currentAmount is less than `TARGET` or not, if `currentAmount` starts off less than `TARGET` this will *never* change *and the loop will never end*.

KEY POINT: Infinite Loops

One risk with loops is that – usually by mistake – you end up with one where the condition never changes to false, and so the loop never ends. This is called an ***infinite loop*** (also the name Apple chose for the road it built its main headquarters on for some reason). Of course loops are never *really* infinite – they finish when you terminate the program. But you need to be careful when you're writing code to avoid infinite loops.

KEY POINT: The Loop Must Change the Condition

Unless you intend to write an infinite loop, then in order for it to end, the condition must change at some point from true to false. In order for this to happen, there has to be something inside the loop that changes the condition. In the case above, the loop will end when `currentAmount < TARGET` - and since we are adding 5% to `currentAmount` every time we go around the loop, we can be sure this will eventually be true in this case. However, if the code in the body of the loop didn't change `currentAmount` at all, then unless it was *already* >= TARGET, then the loop would never end. (And if you intend to write an infinite loop, then the "accepted" way is to write:

```
while (true) {…
```

KEY POINT: Loops that Never Execute

In the Key Point above we raised the possibility that `currentAmount` might already be greater than `TARGET` – what happens in this case? Because the condition is never true, the loop body will never be executed. This is completely reasonable – and you will sometimes write programs which have loops that will "normally" not be executed. For example, suppose you write a program that requires a user to enter data, and then checks to see if it's correct – if it isn't, it prints a message and asks them to enter it again. You would put the error message and the request to re-enter the data in a loop. But if the user entered the data correctly first time (which would, probably, usually be what happened), then the loop never gets executed.

KEY POINT: {..} and Loops

Just like if statements, if the body of a while loop – or any loop – is only one line long, you don't need the {…}. But just like if statements, *you should always put the {…} in.*

Complete Program

We haven't seen a complete program for a while, so here's the interest rate one, where we get the initial amount from the user.

```
import java.util.Scanner;

class TimeToDouble {
     public static void main(String[] args) {
          Scanner in = new Scanner(System.in);

          final double TARGET = 1000.0;

          System.out.print("Enter amount: ");

          double initialAmount = in.nextInt();
          final double INTEREST_RATE = 0.05;
          int i = 0;

          double currentAmount = initialAmount;
          while (currentAmount < TARGET) {
               currentAmount += currentAmount * 0.05;
               i++;
          }

          System.out.println("It takes " + i +
               " years at " +
               INTEREST_RATE*100 +
               "% interest to grow " +
               initialAmount + " to " + TARGET);
     }
}
```

KEY POINT: Using i

Notice, I've changed the variable years to i in this version. One thing we've said is that variable names should be meaningful; and so we can't normally use single letter names for variables. However, in this case we have used i. Using i (and j, k, l) is a bit of an exception when we need variables to *count* things in loops. From the very earliest days of programming in high level languages (the mid 1950s) i (and j, k, l) have been used for counters (especially *loop* counters) and so they are traditional and so widely-understood. Which means it's OK to use them but *only* as counters. (And if you want to use more meaningful things like "count", or "years", that's fine.)

Do-While loops

These are very commonly used when you are getting – and checking – input; because you have to ask for the input at least once, so you know it will go around at least once. The syntax is:

```
do {
     code
while(condition);
```

KEY POINT: This One *Does* Have a ;

Note – unlike the other kinds of loops (and if statements) do-while loops **do** have a ";" at the end. (This is frankly annoyingly unhelpful; but it is what it is.)

As an example of a do-while loop, this code repeatedly asks the user to enter a string until they enter either "yes" or "no" – this is something that you will often find useful.

```
Scanner in = new Scanner(System.in);
String input;
do {
     System.out.print("Enter either yes or no.");
     input = in.nextLine();
} while (!input.equals("yes") && !input.equals("no"));
```

This loop continues as loop as the input is not equal to "yes" *and* it's not equal to "no" (so by the time it's finished, you know it can only be one of "yes" or "no").

Aside: equalsIgnoreCase

One problem with the code above is that it only accepts "no" and "yes" in lower case – what about "No" or "Yes"; or "NO" or "YES"? We could check for each of these but the code would be very complex. Fortunately, we can write:

```
…
} while (!input.equalsIgnoreCase("no") &&
         !input.equalsIgnoreCase("yes"));
```

`equalsIgnoreCase` checks if two strings are equal regardless of their case. Better yet, we should write something like:

```
final String END = "no";
final String CONT = "yes";
…
} while (!input.equalsIgnoreCase(END) &&
         !input.equalsIgnoreCase(CONT));
```

By using final strings like this we can, for example, make our code independent of language. For example, by just changing two words:

```
//Language is Greek
final String END = "όχι";
final String CONT = "ναί";
…
} while (!input.equalsIgnoreCase(END) &&
         !input.equalsIgnoreCase(CONT));
```

we can make it work in a completely different language.

For Loop

The last kind of loop differs in that it's meant to be used when we know when the loop starts how many times it will go around. However, things are a little complicated and first we need a bit of history.

'Traditional' For Loops

The syntax of older programming languages varied but they usually had the same kind of for loops. For example, in Pascal:

```
for i := 0 to 10 do begin
     (* code *)
end
```

is a loop that:

- declares a variable i;
- that starts at 0;
- ends at 10; and

executes some code ten times - comments in Pascal are between (* and *); and begin and end take the place of { and }. You can also count down:

```
for i := 500 downto -20 do
     …
```

The start and end values can also be variables – *but you can only count in steps of 1*. This is how most for statements worked, with some variations. *Notice that it's only possible with this kind of statement to write for loops where you know before the loop starts how many times it will go around.*

For Loops in Java

The designers of one of the ancestor languages of Java (called C) decided to generalize for loops and make them more flexible. The syntax is:

```
for(start; end condition; update {
     code
}
```

In this:

- **start** is code that will be executed once, immediately before the loop begins.
- **end condition** is a Boolean condition (like the ones in a while or do loop) – the loop will continue as long as it is true.
- **update** is code that will be executed at the end of every iteration of the loop – so if the loop goes round 10 times, it will be executed 10 times.

Rewriting the two Pascal loops above we get:

```
for (int i = 0; i <= 10; i++) {
     //code
}
```

and

```
for (int i = 500; i >= -20; i--) {
…
```

- The first part of loop before the first ";" defines the initial condition – in this case it declares an integer variable i and sets the initial value to zero for the first one; and 500 for the second.
- The second part (between the two ";") defines the end condition – while the condition is true the loop continues. In the first case it continues looping until i is > 10; in the second until i is < -20.
- The final part is not present in the Pascal version – it says what happens at the end of each loop. In the first case we add one to i and in the second we subtract one (adding one is the default action in Pascal; using 'downto' changes that to subtracting one – no other actions are possible in Pascal).

The reason this is more flexible is that *we can put more or less whatever code we want in the three parts* (including nothing; see below). The examples above are very common in practice, but we can also be more flexible. One example is that we don't have to count up (or down) by one for each loop. For example, the following code prints out only odd numbers up to END_VALUE:

```
for (int i = 1; i < END_VALUE; i +=2) {
     System.out.println(i);
}
```

This flexibility is very helpful – but also dangerous for two reasons.

It encourages 'cleverness' – for example you can re-write the odd number program as:

```
for (int i = 1; i< END_VALUE; System.out.println(i), i +=
2);
```

In this case the ";" at the end of the line is deliberate – we want the loop body to be empty because we've put all the code in the last part of the for loop. Notice the two parts are separated by a "," – this is called the comma operator and can be used to join bits of code in a for loop: but it's rarely a good idea (so don't do it). Compressing code like this makes it shorter, and is 'ingenious' – but it's also hard to read so we don't want to encourage it.

It blurs the line between for and while loops – we'll talk about this a bit more later – first another example.

Above we had a program that computed how many years it would take for an amount of money to grow to a target amount. Suppose instead you want to know how much money you will have after a specific number of years? This is a place to use a for loop *because you know how many years you will be making the calculation for*.

```
double initialAmount = 500.0;
final double INTEREST_RATE = 0.05;
final int NO_OF_YEARS = 15;

double currentAmount = initialAmount;

for(int i = 0; i < NO_OF_YEARS; i++) {
     currentAmount += currentAmount * INTEREST_RATE;
}

System.out.println("After " + NO_OF_YEARS + " years " +
                initialAmount + " will be worth " +
                currentAmount);
```

Counting Loops and Off-by-One Errors

Notice that in our example above the value of i starts at zero. But the continues only as long as i is *less than* NO_OF_YEARS. This means that the value of i ranges from 0..NO_OF_YEARS-1, so the total number of loops (iterations, or repetitions) is NO_OF_YEARS, which is correct. We could equally change it to:

```
for(int i = 1; i <= NO_OF_YEARS; i++)
```

where we start with i equal to one and continue as long as it is less than or equal to `NO_OF_YEARS`. This means that the value of i ranges from `1..NO_OF_YEARS` – which is still equal to `NO_OF_YEARS` (which is still correct).

It does not really matter which of these you choose – though 'traditionally' we start counting from zero in Computer Science so you would normally pick the first one. The thing to avoid though is mixing them up:

```
for(int i = 1; i < NO_OF_YEARS; i++)
```

In this case, i only ranges from `1..NO_OF_YEARS-1`, so there are only `NO_OF_YEARS-1` interations.

```
for(int i = 0; i <= NO_OF_YEARS; i++)
```

In this case, i ranges from `0..NO_OF_YEARS`, so there are only `NO_OF_YEARS+1` interations.

In both cases the number of iterations is wrong by one – we call this kind of error (which is very common) an ***off-by-one error***. When writing loops (of any kind) you do need to be careful to ensure that the loop iterations the correct number of times (and not one too many or too few).

Loop Counters and Scope

In chapter 5 on Scope we said that the *scope* of a variable was the {…} it was declared in. Things are slightly different for *for* loops because the loop counter is declared *outside* the loop. So as *a special case* the loop counter declaration is considered to be in the scope of the loop body:

```
for(int i = 0; i < NO_OF_YEARS; i++) {
     currentAmount += currentAmount * INTEREST_RATE;
}
```

The variable declaration (in red) is considered to be in the scope of the loop body (also in red) – even though it's strictly not inside the {..}.

Using the Loop Counter Outside the Loop

One thing that does cause confusion sometimes, and related to scope, is using the loop counter *outside* (usually *after*) the loop. Notice in the example above we defined a variable called `NO_OF_YEARS`. If we tried to rewrite our code like this

```
//WON'T COMPILE
for(int i = 0; i < NO_OF_YEARS; i++) {
     currentAmount += currentAmount * INTEREST_RATE;
}

System.out.println("After " + i + " years " +
                initialAmount + " will be worth " +
                currentAmount);
```

using i (in red again) instead of `NO_OF_YEARS`, the code would not compile because we are trying to use i outside its scope. In this case it's not a huge problem but there are times when it can be something you need (or at least want) to do. The solution is to just move the declaration of i to before the loop:

```
//This works
int i = 0;
for(; i < NO_OF_YEARS; i++) {
     currentAmount += currentAmount * INTEREST_RATE;
}

System.out.println("After " + i + " years " +
               initialAmount + " will be worth " +
               currentAmount);
```

because now the declaration of i is in the scope of the println statement that uses it. *Notice that now the first part of the for loop is empty* – this is fine and it's not a problem if one or more parts are empty (though if they are *all* empty – `for(;;)` – you will have an infinite loop).

Interchangeable While and For Loops – Danger!

One of the consequences of the more general kinds of for loop used in Java is that they have become *functionally identical* to while loops. This makes it *very* easy to use the wrong one. Here's an example – this should be a for loop (shown first) but we can write it as a while loop:

```
for (int i = 0; i < 10; i++) {
     //code
}
```

identical to:

```
int i = 0;
while (i < 10) {
     //code
     i++;
}
```

I've coloured coded the various parts to show which bits correspond in each case:

- Red – setting up the loop counter;
- Blue – the loop condition;
- Green – incrementing the loop counter; and
- Purple – the code.

(Strictly speaking, the second loop is not *quite* identical to the first because the scope of the variable i is wider – but this is a very minor point.)

Because for loops are so general, we can do the same thing the other way around – here's an example of a while loop that calculates the *greatest common divisor* of two integers (largest integer that divides them both without

remainder) `valueA` and `valueB` – first as a while loop and then rewritten as a for loop (again colour coded):

```
//Assume a & b integers already declared
a = valueA;
b = valueB;
while(b !=0 ) {
      int temp = b;
      b = a % b;
      a = temp;
}
System.out.println("Greatest common divisor of " +
      valueA + " and " + valueB + " is " + a);
```

and re-written:

```
for(a = valueA, b = valueB; b != 0;) {
      int temp = b;
      b = a % b;
      a = temp;
}
System.out.println("Greatest common divisor of " +
      valueA + " and " + valueB + " is " + a);
```

Notice again this is an example of a for loop where part is empty – in this case, the last part.

KEY POINT – Use the Right One

Clearly Java does not care which kind of loop – *while* or *for* - you use, and in fact there is no technical need for both in the language. However, you should care – because it affects the readability of your program.

- If you do not know at the start of a loop how many repetitions there will be, use a while loop (or do-while if you know there will be at least one).
- If you do know at the start of a loop how many repetitions there will be, use a for loop.

Programmers in Java (and other languages) understand that this is normally the convention – and so if they see a for loop they expect that it computes something where you know the number of iterations before the loop starts; and for a while loop, the opposite. This knowledge helps their understanding. Now, there are a few cases later on where we'll break this 'rule' – but in general, it's something that helps understanding when reading programs and you should stick to it.

Programming Style and Loops – Structured Programming

All the loops we've seen so are quite structured – we test if we are going to continue to loop at the beginning or end; we cannot exit the loop from the middle – only if the test at the start (or end) tells us the loop should finish. However, there are (sort of unfortunately) two commands in Java that let us change that. This section explains what they are and why you should not use them.

The break Statement

Recall the last chapter where we discussed switch statements:

```
switch(card) {
case "Ace" :   //do something
          break;
case "King":   //do something else
          break;
…
case "one":    //do one last thing
          break;
}
```

The break statement is not only used as part of the switch statement – it also has a more general meaning. Simplistically, if the command 'break' appears in a program, it means 'jump to the first line of code after the immediately following '}' .You can see that's exactly what break does in a switch – it causes execution to jump to the code following the switch, after the '}' that ends the switch.

You can use break within loops in your code (*but you normally shouldn't – see later*). In particular, if you put it in a loop it will cause execution to exit the loop at that point. For example, remember this loop:

```
while (currentAmount < TARGET) {
     currentAmount += currentAmount * 0.05;
     i++;
}
```

This is (not quite) exactly the same as this loop:

```
while (true) {
     currentAmount += currentAmount * 0.05;
     i++;
     if (currentAmount >= TARGET) {
          break;
     }
}
```

(It's not quite the same because this loop will always execute at least once.)

Because it says while(true) the loop itself is infinite – true can never be false so the loop will go on forever. But the if statement contains a break, which will exit the loop if executed. I hope you would agree though, the second one is confusing and hard to read compared to the first.

The continue Statement

Very similar to the break statement, the continue statement does not exit a loop, instead it jump to the test to see if the loop should continue. For example:

```
int sum = 0;
for(int i = 1; i <= 100 ; i++){
    if(i % 2 == 0) {
         continue;
    }
    sum += i;
}
```

This code adds up only the odd numbers between 1 and 100 – the if statement checks if the value of i is even, and if it is, the continue statement means it jumps straight back to the for statement, skipping the line sum += i;

Structured Programming – and Why not to use break and continue

This looks like it might come in handy – and very occasionally, it does (and so does break). But more usually it cases problems. In fact, we can much more easily write the example above as:

```
int sum = 0;
for(int i = 1; i <= 100 ; i++){
    if(i % 2 != 0) {
        sum += i;
   }
}
```

or even better, this:

```
int sum = 0;
for(int i = 1; i <= 100 ; i += 2){
    sum += i;
}
```

without using continue at all.

Very occasionally – usually when you're dealing with some error condition or special case deep in a load of nested loops – these commands can come in handy. But:

- Usually there's a better way (including exceptions which are in the next module CS-115).
- They are open to abuse and lead to very bad code.

For example, here's a fragment of a good solution to a coursework I set in previous years – this coursework was to simulate the card game Blackjack. The point of this loop is to ask the user if they want to take another card, and if they do, to add a random number in the range 1 to 10 to their current total; and to keep doing that until they either stop asking or they go over the maximum allowed value.

```
do {
     System.out.println("Your current score is: "
          + playerValue);
     System.out.print("Enter stick or twist: ");
     input = in.next();
     if (input.equals("twist")) {
         playerValue += rnd.nextInt(MAX_VAL) + MIN_VAL;
     }
} while ((input.equals("twist")) && (playerValue <=
MAX_SCORE));
```

This code isn't perfect (it should really also check if the user typed 'stick' and if they didn't type 'twist' *or* 'stick' if should ask them again). But it's pretty clear:

- You ask the user for input
- If they type "twist" you add a random number to their score
- The loop ends when they either don't type "twist" or they go over the maximum value.

Compare it to this code:

```
while (true) {
     System.out.println("Your current score is: "
          + playerValue);
     System.out.print("Enter stick or twist: ");
     input = in.next();
     if (input.equals("stick")) {
         break;
    }
     playerValue += rnd.nextInt(MAX_VAL) + MIN_VAL;
     if (playerValue > MAX_SCORE) {
         System.out.println("You are bust!");
         break;
    }
}
```

- First it's an infinite loop – we have no idea at first sight when it will end.
- Second it has two break statements, so it can exit from the middle at *two different places* – which is not at all obvious until you start to read it in detail.

KEY POINT: Structured Programming

Practice and experience has shown that this type of loop is much harder to understand than the first kind, and much harder to change without introducing errors than the first kind. Code in which loops only start and end in one place (all the ones we've seen except the examples in this section) are categorized as examples of structured programming, and that is what we should aspire to do where possible. There are cases where not doing this is clearer, but:

- they are rare
- none of them will come up in your first year of programming

So try not to use break and continue in loops – with an exception (see below)

Advanced Interlude – Edsger Dijkstra and Goto Considered Harmful

If you want to know more about this try googling *Edsger Dijkstra Goto Considered Harmful* (the name is spelt correctly – it's Dutch – and pronounced 'eds-ger dike-stra'). In early programming languages the primary control command was the 'goto' statement that allowed you to jump wherever you wanted within your code. Prof Dijkstra wrote a letter in 1968 – normally referred to as 'Go To Statement Considered Harmful' – to the journal *Communications of the Association for Computing Machinery* in which he criticized this, leading to the development of modern structured programming. The 'break' and 'continue' statements are basically 'controlled' versions of goto and the same principles apply – and they should *usually* be avoided for the same reasons.

Skip on First Reading if New to Programming: Using Break and Continue – When it's OK

You should mainly try to avoid using break and continue in loops, *but there is an exception.* It can make sense to use break when you have a for loop *which could end early*. To illustrate what I mean, suppose you have a list of data items (how do you make lists? See next chapter) – and suppose that list had 100 items. Then suppose you were looking for something in that list that might not be there. You might have code that looks a bit like this

```
String item = "not found";
for (int i = 0; i < 100; i++) {
     if (ith item what we're looking for) {
          item = value of ith item;
          break;
     }
}
```

We have used *pseudo code* to represent 'lists of items' – we'll see how to really do this in the next chapter. The thing about this loop is that it has two conditions to manage – has it reached the end of the list? has it found the item? To do this with a while loop, we need:

```
String item = "not found";
int i = 0;
boolean found = false;
while (!found && i < 100) {
     if (ith item what we're looking for) {
          item = value of ith item;
          found = true;
     }
     i++;
}
```

This works and avoids the break – but it's actually a bit harder to understand and needs an extra variable. So in this case break is better - *but part of the reason for that is the loop is short* – breaks buried in the middle of long loops generally make things worse. **So break is OK in short for loops that could end early.**

What about continue? Again, the loop needs to be short – but it can be handy if you have a loop where you need to do things only part of the time. For example:

```
for(int i = 0; i < 100; i++) {
     if(some condition) {
          continue; //condition true, don't do it
     }
     do stuff
}
```

vs

```
for(int i = 0; i < 100; i++) {
     if(!some condition) {
          do stuff
     }
}
```

Depending on the condition, and what 'stuff' you are 'doing', the version with continue *might* be clearer.